Lecture 23: Thread Level Parallelism -- Introduction, SMP and Snooping Cache Coherence Protocol

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

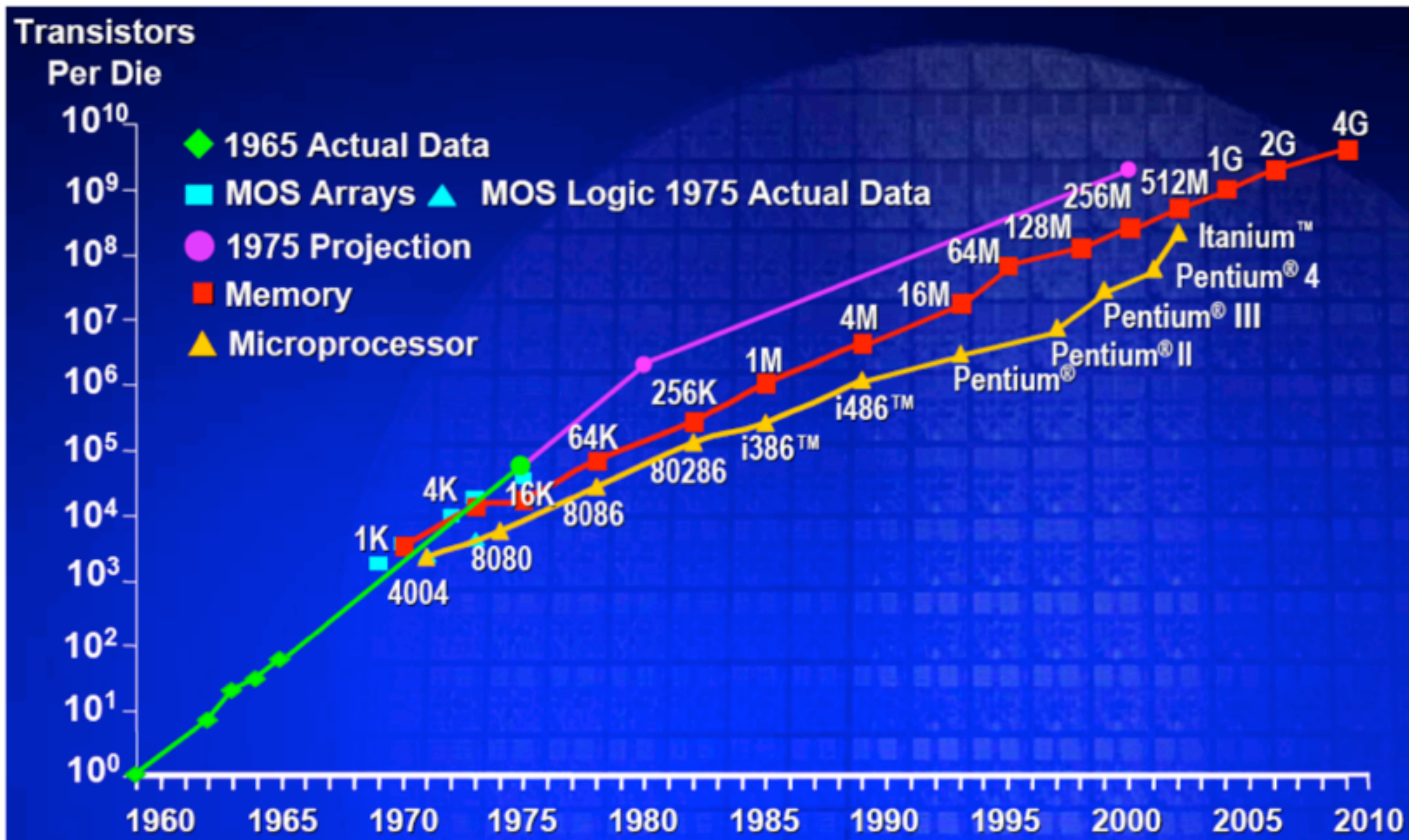
https://passlab.github.io/CSCE513

Topics for Thread Level Parallelism (TLP)

- Parallelism (centered around …)
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism
- TLP Introduction
  - 5.1
- SMP and Snooping Cache Coherence Protocol
  - 5.2, 5.3
- Distributed Shared-Memory and Directory-Based Coherence
  - 5.4
- Synchronization Basics and Memory Consistency Model
  - 5.5, 5.6
- Others

Moore's Law

- Long-term trend on the density of transistor per integrated circuit
- **Number of transistors/in^2 double every ~18-24 month**



What do we do with that many transistors?

- Optimizing the execution of a single instruction stream through
  - Pipelining
    - » Overlap the execution of multiple instructions
    - » Example: all RISC architectures; Intel x86 underneath the hood
  - Out-of-order execution:
    - » Allow instructions to overtake each other in accordance with code dependencies (RAW, WAW, WAR)
    - » Example: all commercial processors (Intel, AMD, IBM, Oracle)
  - Branch prediction and speculative execution:
    - » Reduce the number of stall cycles due to unresolved branches
    - » Example: (nearly) all commercial processors

What do we do with that many transistors? (II)

- Multi-issue processors:
  - » Allow multiple instructions to start execution per clock cycle
  - » Superscalar (Intel x86, AMD, …) vs. VLIW architectures
- VLIW/EPIC architectures:
  - » Allow compilers to indicate independent instructions per issue packet
  - » Example: Intel Itanium
- SIMD units:
  - » Allow for the efficient expression and execution of vector operations
  - » Example: Vector, SSE - SSE4, AVX instructions

Everything we have learned so far

Limitations of optimizing a single instruction stream

- **Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to**
  - **data dependencies**
  - **limitations of speculative execution across multiple branches**
  - **difficulties to detect memory dependencies among instruction (alias analysis)**
- **Consequence: significant number of functional units are idling at any given time**
- **Question: Can we maybe execute instructions from another instructions stream**
  - **Another thread?**
  - **Another process?**

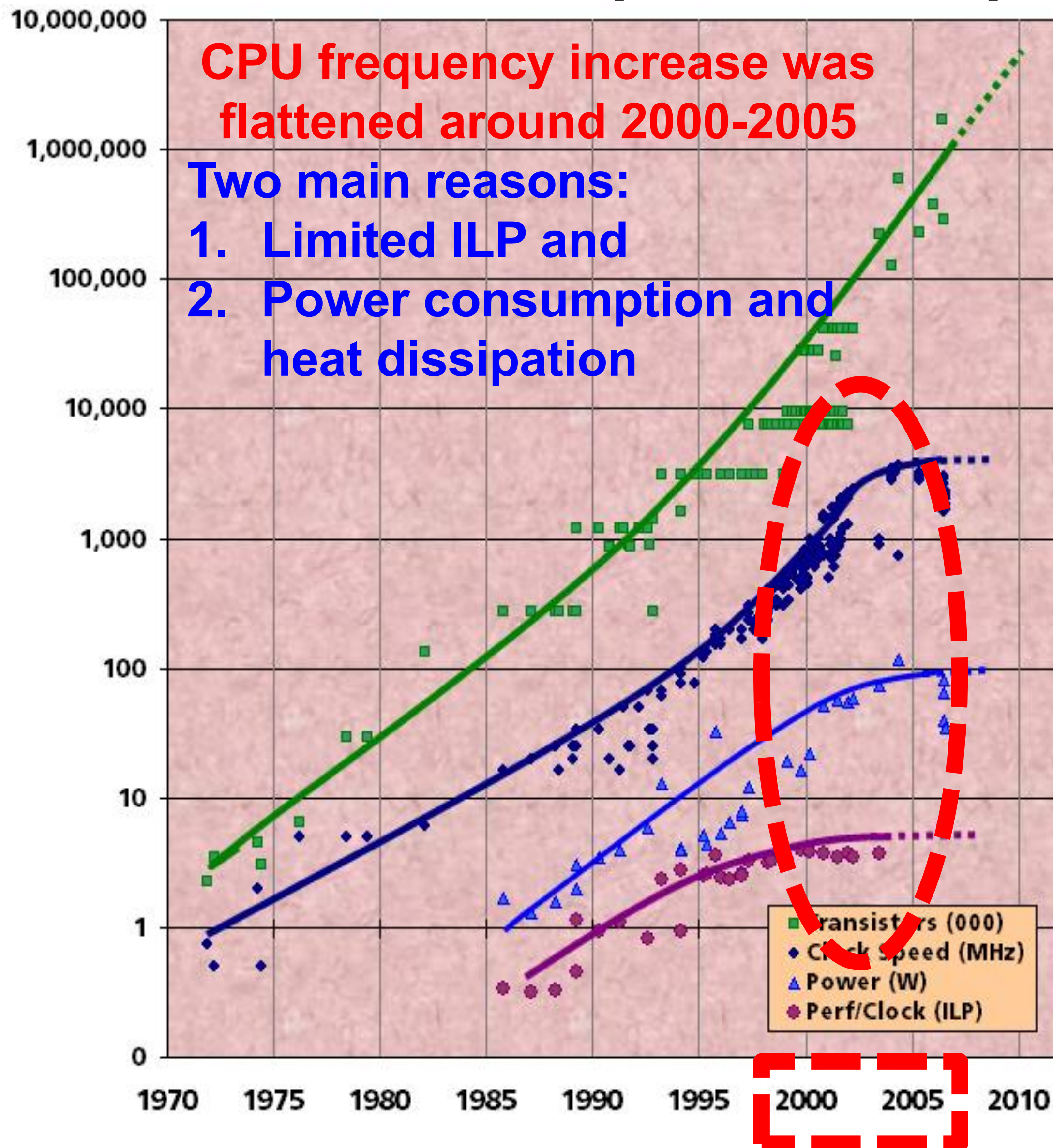
Thread-level parallelism

- **Problems for executing instructions from multiple threads at the same time**
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter
- **Virtual memory management allows for the execution of multiple threads and sharing of the main memory**

- **When to switch between different threads:**
  - Fine grain multithreading: switches between every instruction
  - Course grain multithreading: switches only on costly stalls (e.g. level 2 cache misses)

Power, Frequency and ILP

- **Moore's Law to processor speed (frequency)**



Note: Even Moore's Law is ending around 2021:

http://spectrum.ieee.org/semiconductors/devices/transistors-could-stop-shrinking-in-2021

https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/

http://www.forbes.com/sites/timworstall/2016/07/26/economics-is-important-the-end-of-moores-law

History – Past (2000) and Today

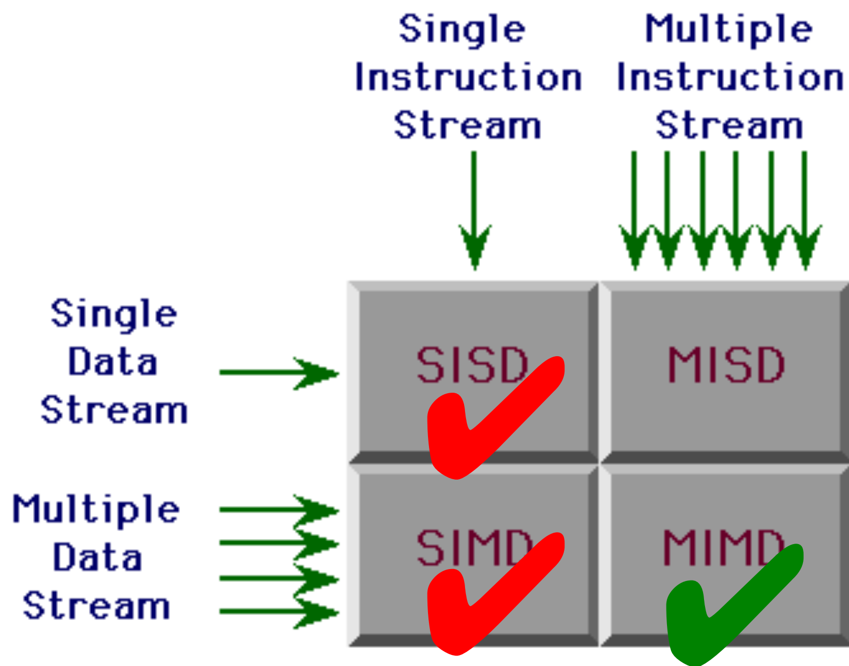
The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

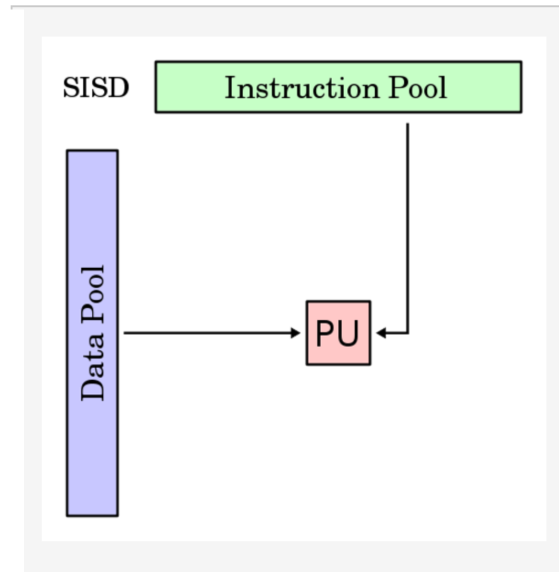
W. Jack Bouknight et al.
The Illiac IV System (1972)

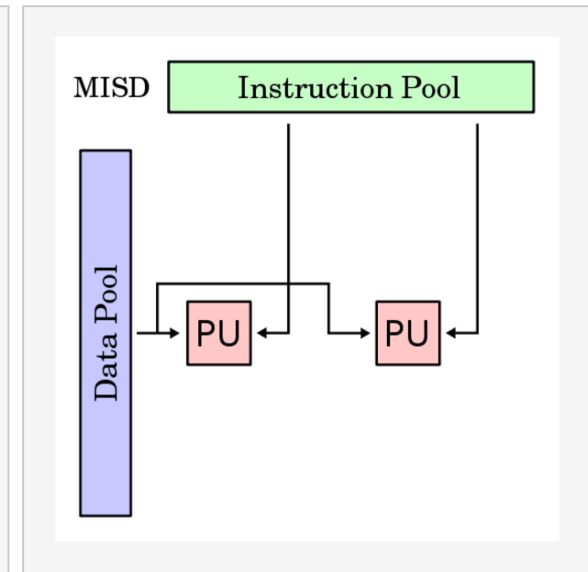
We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

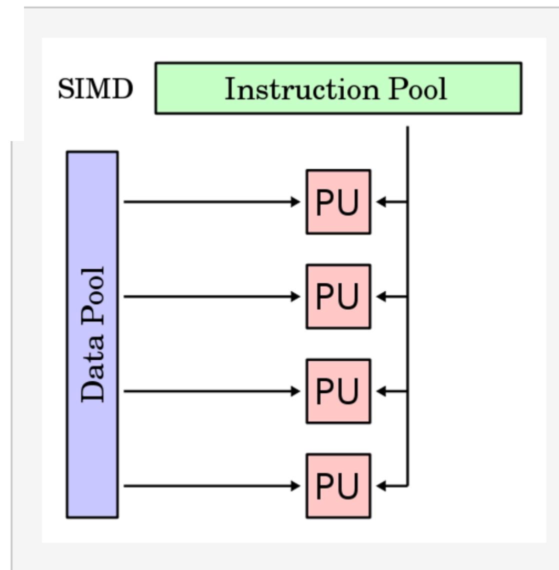
Intel President Paul Otellini,
describing Intel's future direction at the Intel Developer Forum in 2005

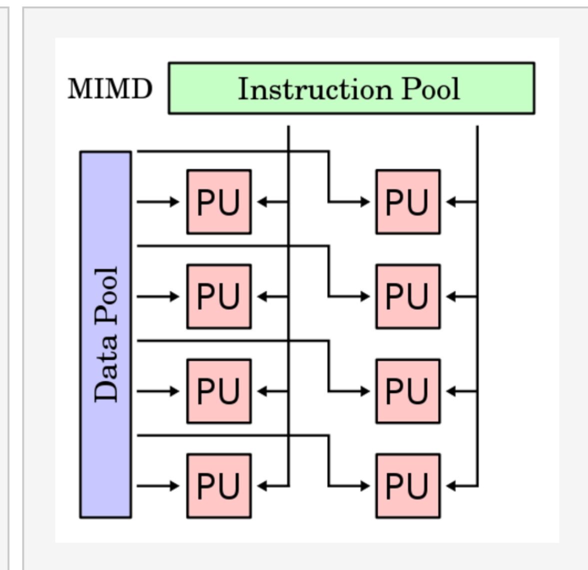
Flynn’s Taxonomy







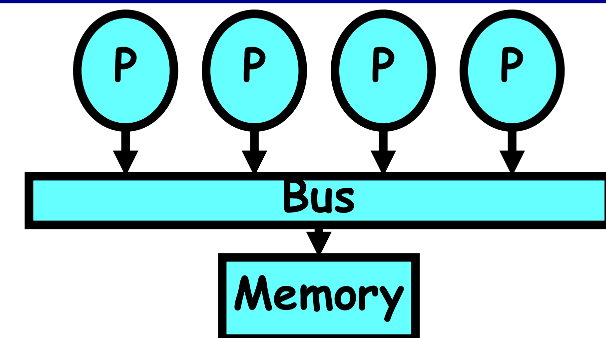


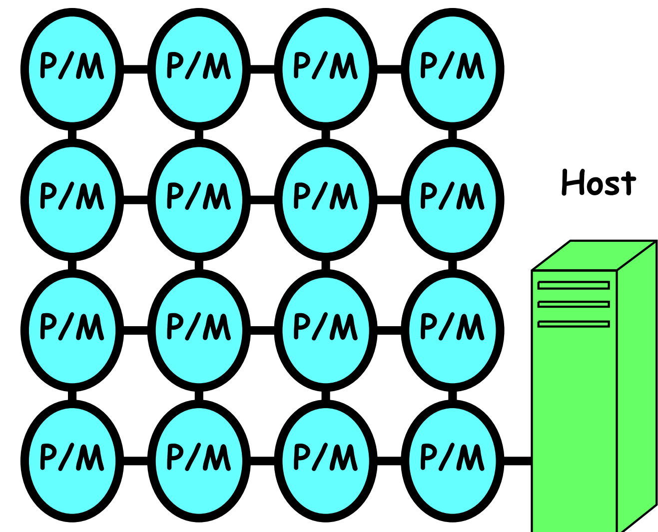


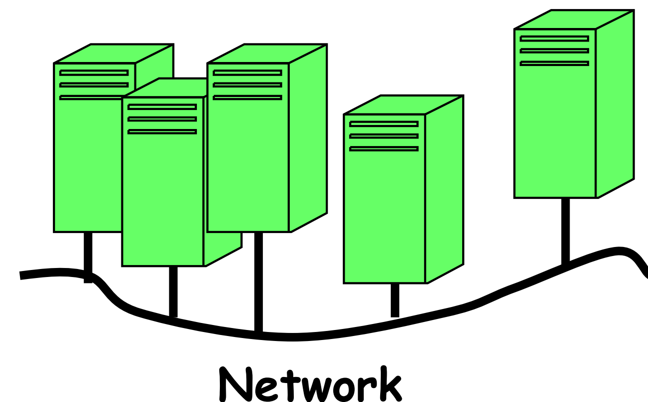
https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

Examples of MIMD Machines

- Symmetric Shared-Memory Multiprocessor (SMP)
  - Multiple processors in box with shared memory communication
  - Current Multicore chips like this
  - Every processor runs copy of OS
- Distributed/Non-uniform Shared-Memory Multiprocessor
  - Multiple processors
    - » Each with local memory
    - » general scalable network
  - Extremely light "OS" on node provides simple services
    - » Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages







Symmetric (Shared-Memory) Multiprocessors (SMP)

- **Small numbers of cores**
  - **Typically eight or fewer, and no more than 32 in most cases**
- **Share a single centralized memory that all processors have equal access to,**
  - **Hence the term *symmetric*.**
- **All existing multicores are SMPs.**
- **Also called *uniform memory access* (UMA) multiprocessors**
  - **all processors have a uniform latency**

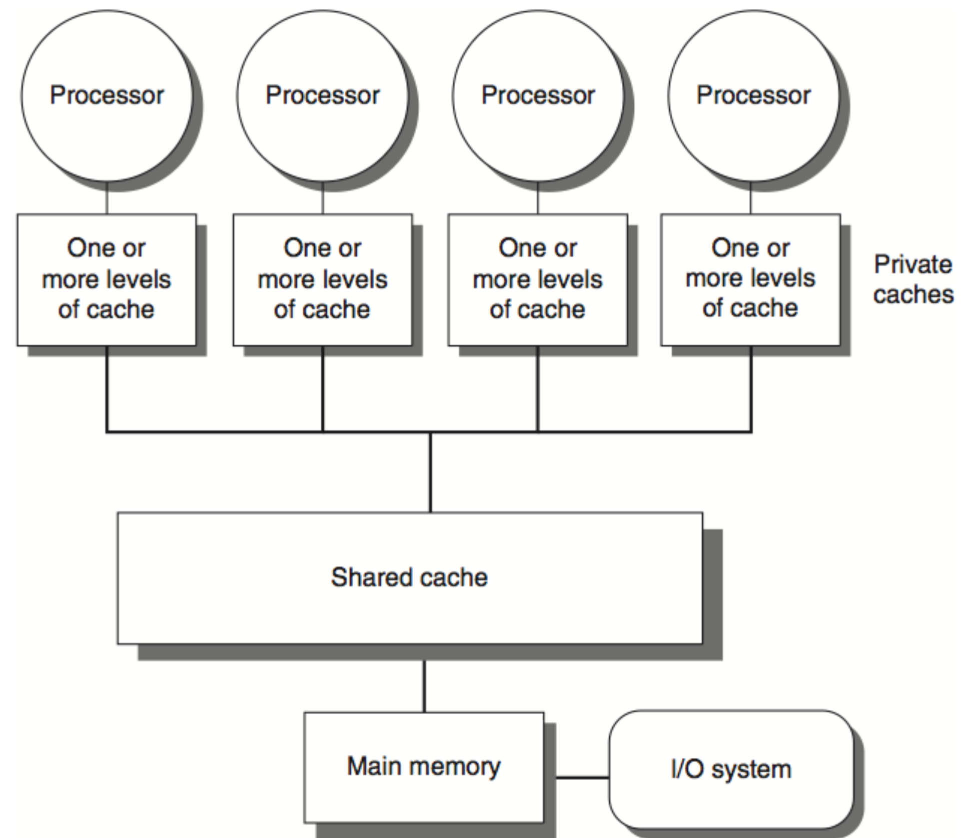
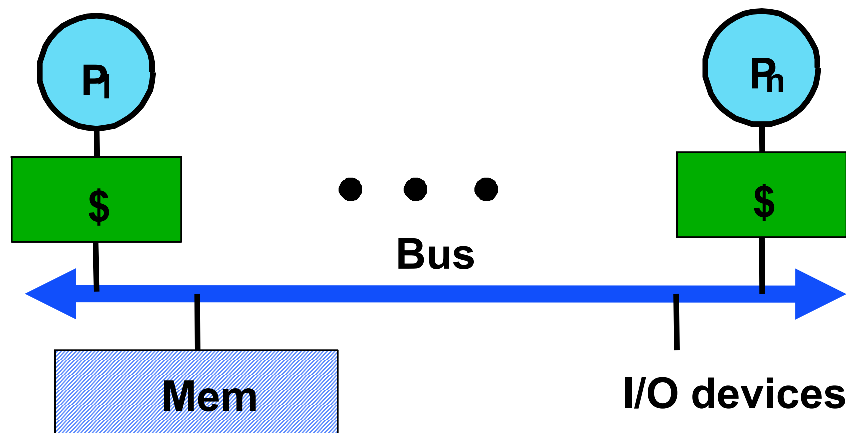


Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.

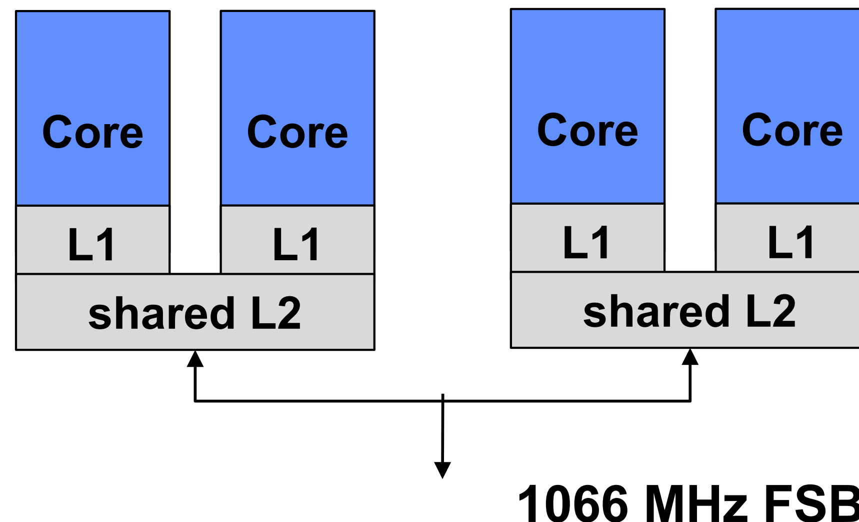
Bus-Based Symmetric Shared Memory

- Still an important architecture – even on chip (until very recently)
  - 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



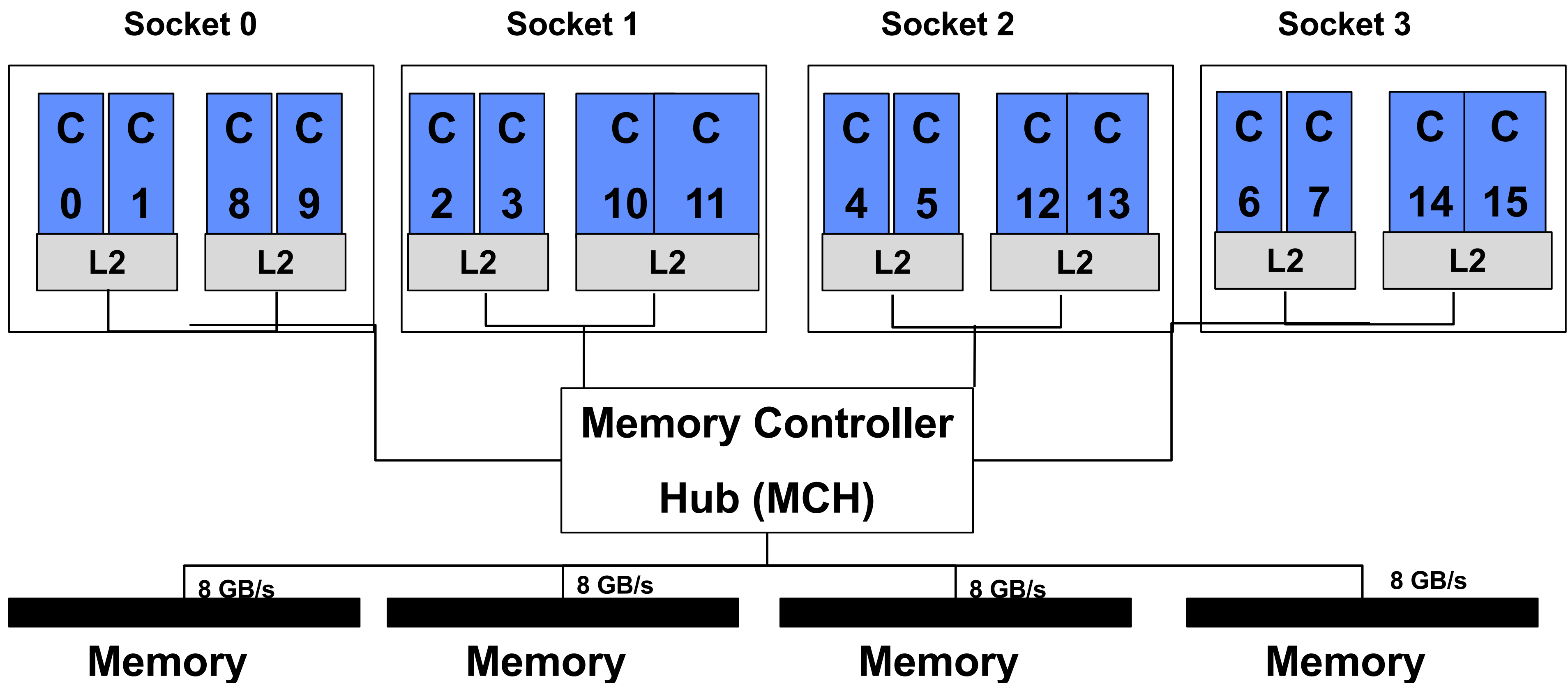
Centralized shared memory system (I)

- **Multi-core processors**
  - Typically connected over a cache,
  - Previous SMP systems were typically connected over the main memory
- **Intel X7350 quad-core (Tigerton)**
  - Private L1 cache: 32 KB instruction, 32 KB data
  - Shared L2 cache: 4 MB unified cache



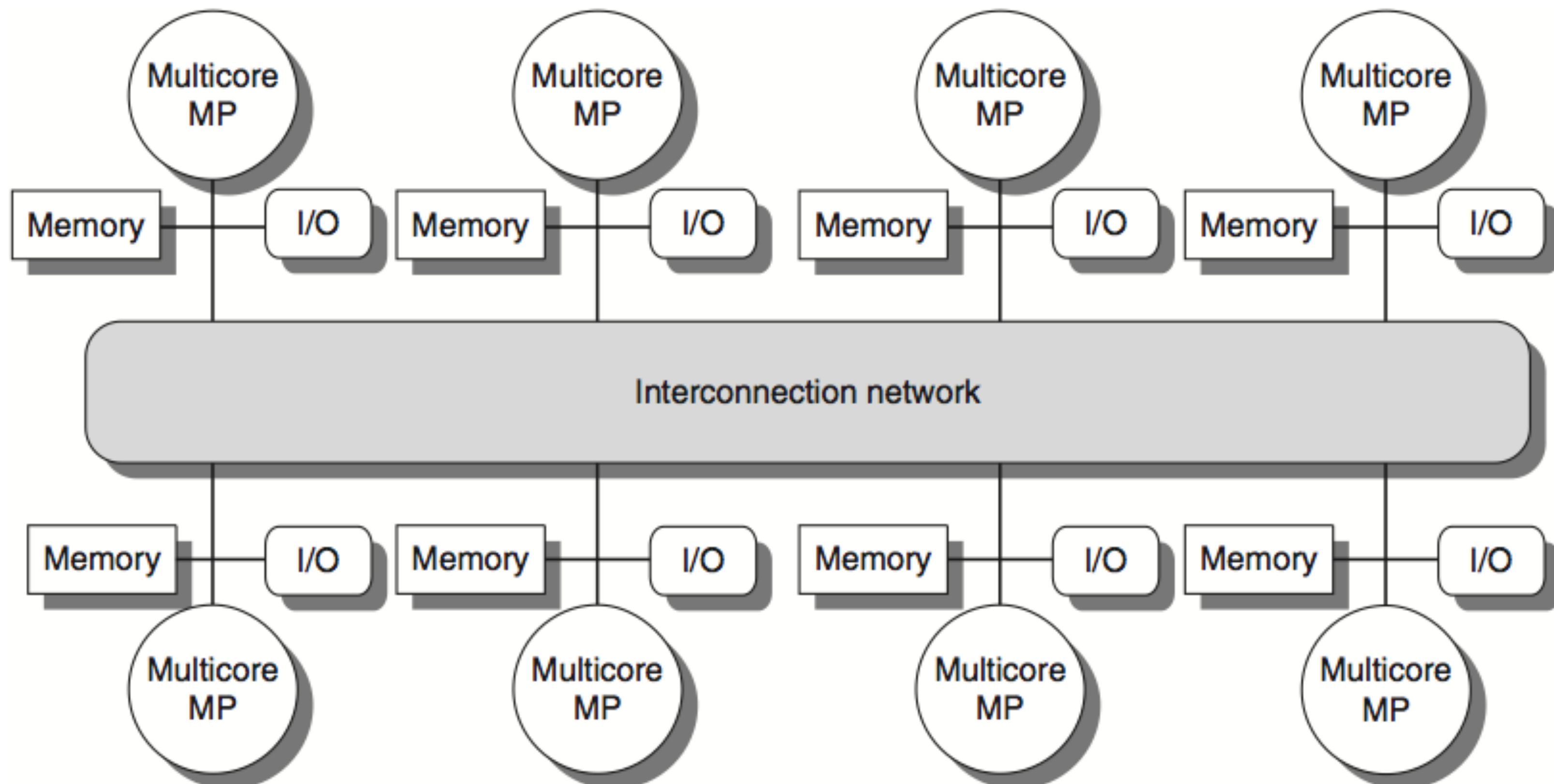
Centralized shared memory systems (II)

- **Intel X7350 quad-core (Tigerton) multi-processor configuration**



Distributed Shared-Memory Multiprocessor

- Large processor count
  - 64 to 1000s
- Distributed memory
  - Remote vs local memory
  - Long vs short latency
  - High vs low latency
- Interconnection network
  - Bandwidth, topology, etc
- Nonuniform memory access (NUMA)
- Each processor may has local I/O



Distributed Shared-Memory Multiprocessor (NUMA)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors

Shared-Memory Multiprocessor

- SMP and DSM are all shared memory multiprocessors
  - UMA or NUMA
- Multicore are SMP shared memory
- Most multi-CPU machines are DSM
  - NUMA

- Shared Address Space (Virtual Address Space)
  - Not always shared memory

Performance Metrics

- **Speedup: how much faster does a problem run on *p* processors compared to 1 processor?**

$$S(p) = \frac{T_{total}(1)}{T_{total}(p)}$$

  - **Optimal: *S(p)* = *p* (linear speedup)**

- **Parallel Efficiency: Speedup normalized by the number of processors**

$$E(p) = \frac{S(p)}{p}$$

  - **Optimal: *E(p)* = *1.0***

Amdahl's Law

- **Most applications have a (small) sequential fraction, which limits the speedup**

$$T_{total} = T_{sequential} + T_{parallel} = fT_{Total} + (1-f)T_{Total}$$

f: **fraction of the code which can only be executed sequentially**

$$S(p) = \frac{T_{total}(1)}{(f + \frac{1-f}{p})T_{total}(1)} = \frac{1}{f + \frac{1-f}{p}}$$

- **Assumes the problem size is constant**
  - **In most applications, the sequential part is independent of the problem size**
  - **The part which can be executed in parallel depends.**

Challenges of Parallel Processing

- **1. Limited parallelism available in programs**
  - Amdahl's Law

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's law is

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

- **0.25% can be sequential**

Simplifying this equation yields:

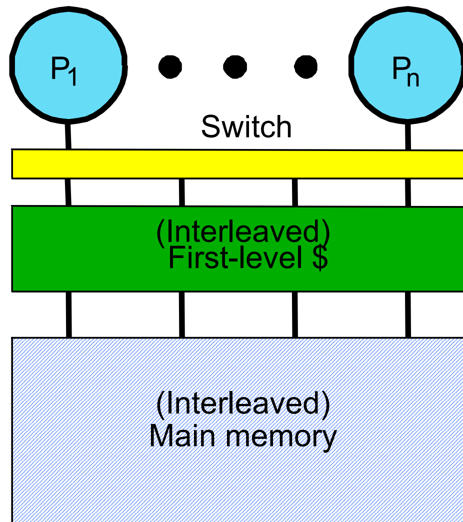
$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

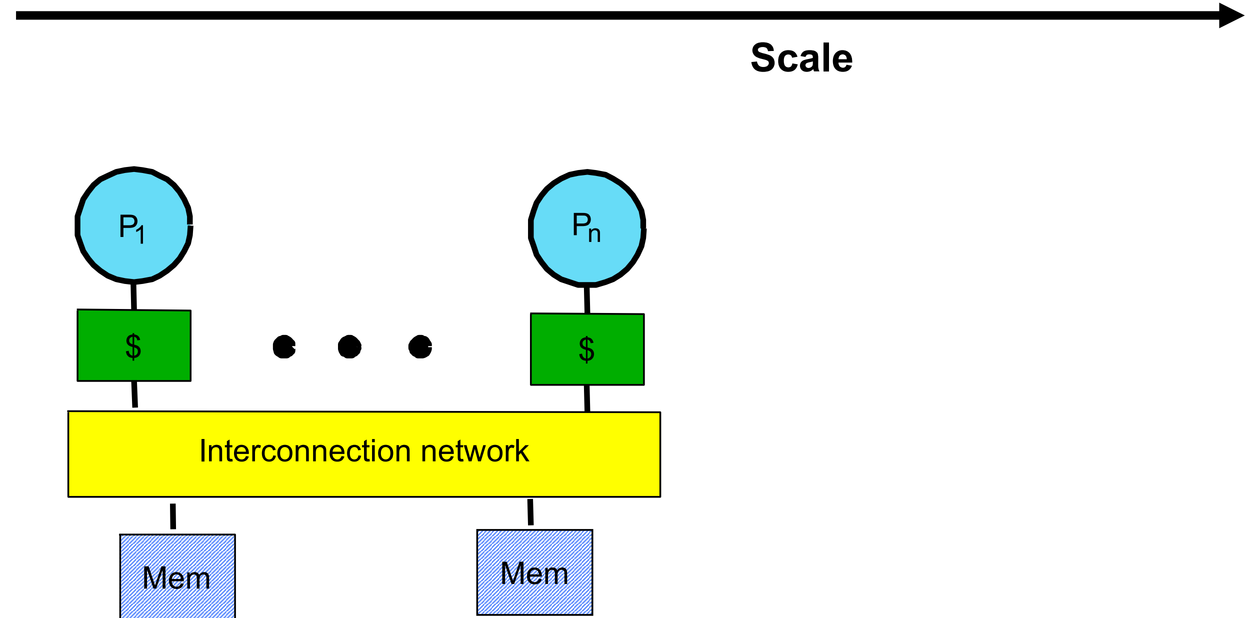
$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

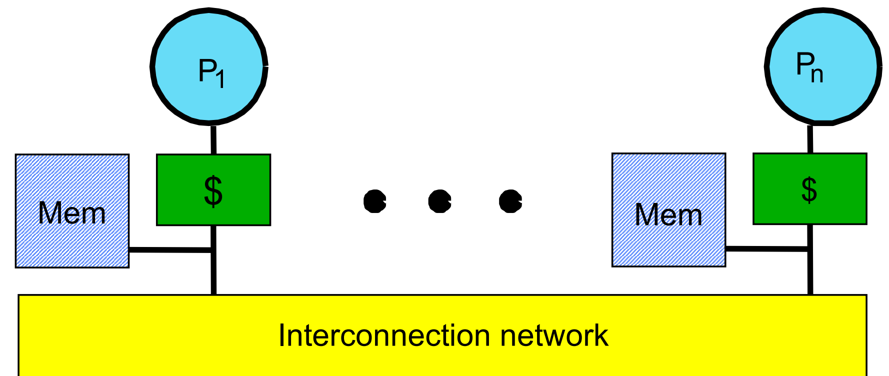
Cache in Shared Memory System (UMA or NUMA)



Shared Cache



UMA



NUMA

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

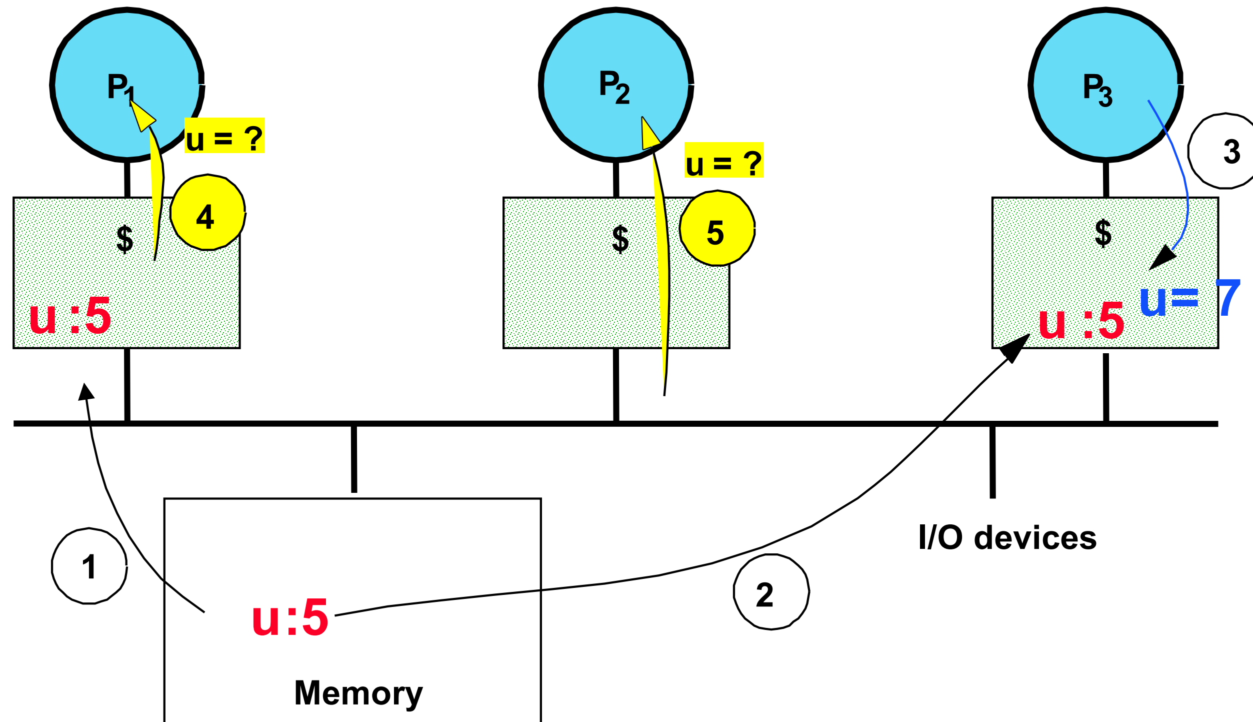
⇒ *Cache coherence* problem

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

Example Cache Coherence Problem

```
int count = 5;
int * u= &count;
….
```

T1 (P1)	T2 (P2)	T3 (P3)
a1 = *u;		
		a3 = *u;
		*u = 7;
b1 = *u		
	a2 = *u	



Things to note:

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 and when

- Processes accessing main memory may see very stale value

Unacceptable to programs, and frequent!

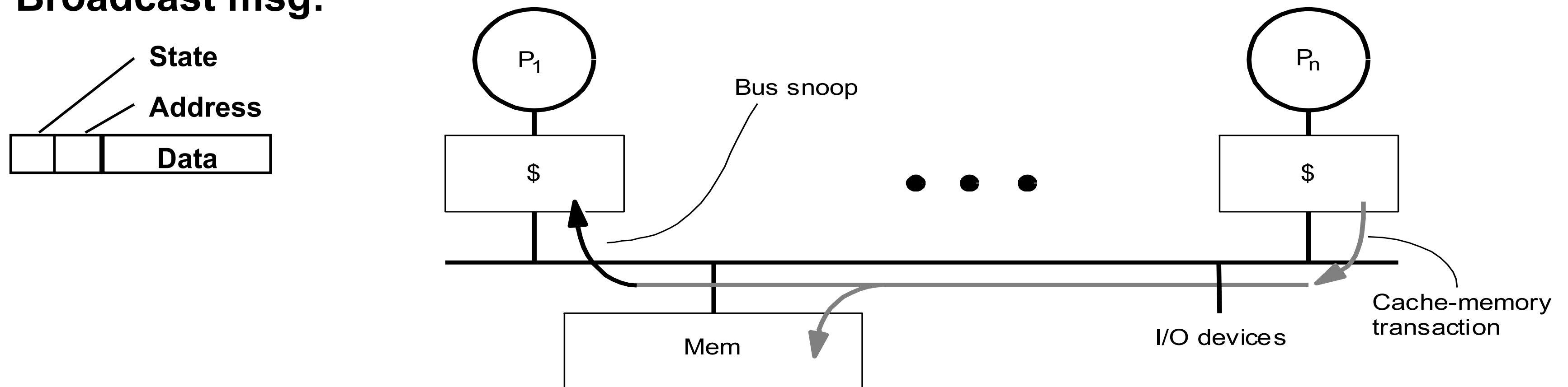
Cache Coherence

- **Typical solution:**
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, 'notification' of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- **Cache coherence is only an issue in case multiple tasks access the same item and one is to write**
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

Cache Coherence Protocols

- Snooping Protocols
  - Send all requests for data to all processors, the address
  - Processors snoop a bus to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for centralized shared memory machines
- Directory-Based Protocols
  - Keep track of what is being shared in centralized location
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Commonly used for distributed shared memory machines

Snoopy Cache-Coherence Protocols



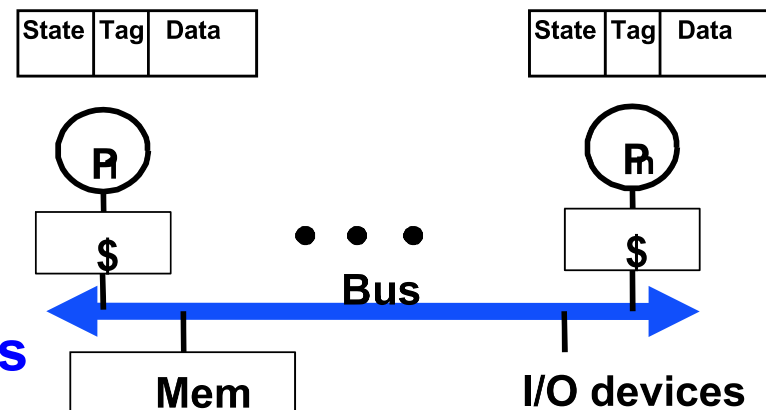
- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and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

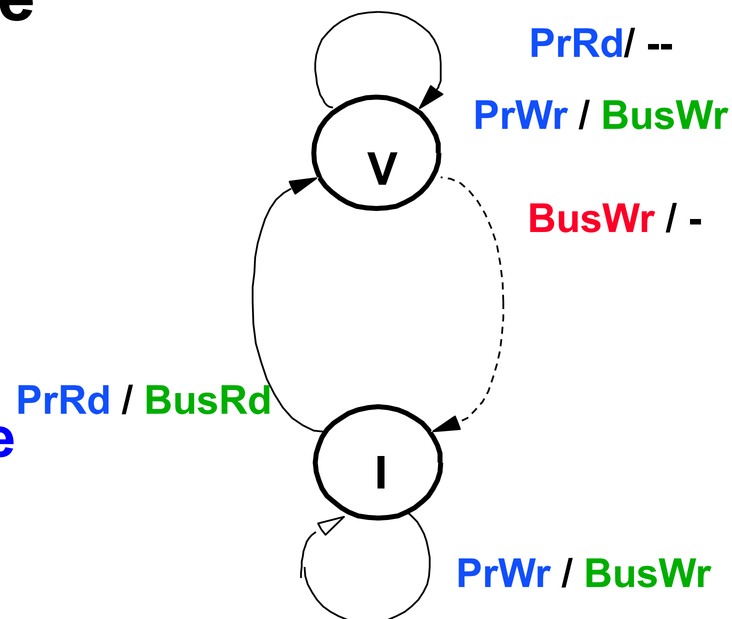
Basic Snoopy Protocols

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
  - Read Miss:
    - » Write-through: memory is always up-to-date
    - » Write-back: snoop in caches to find most recent copy
- Write Update Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

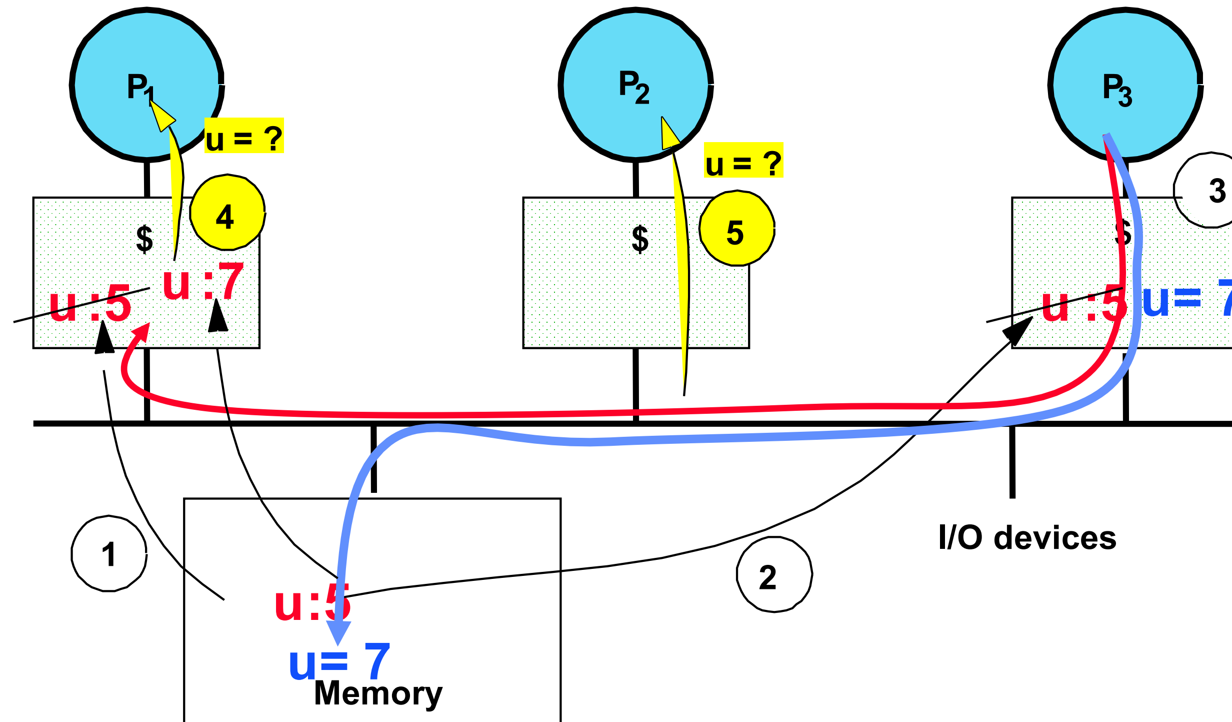
Write Invalidate Protocol

- Basic Bus-Based Protocol
  - Each processor has cache, state
  - All transactions over bus snooped
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block, but write invalidates them
- 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



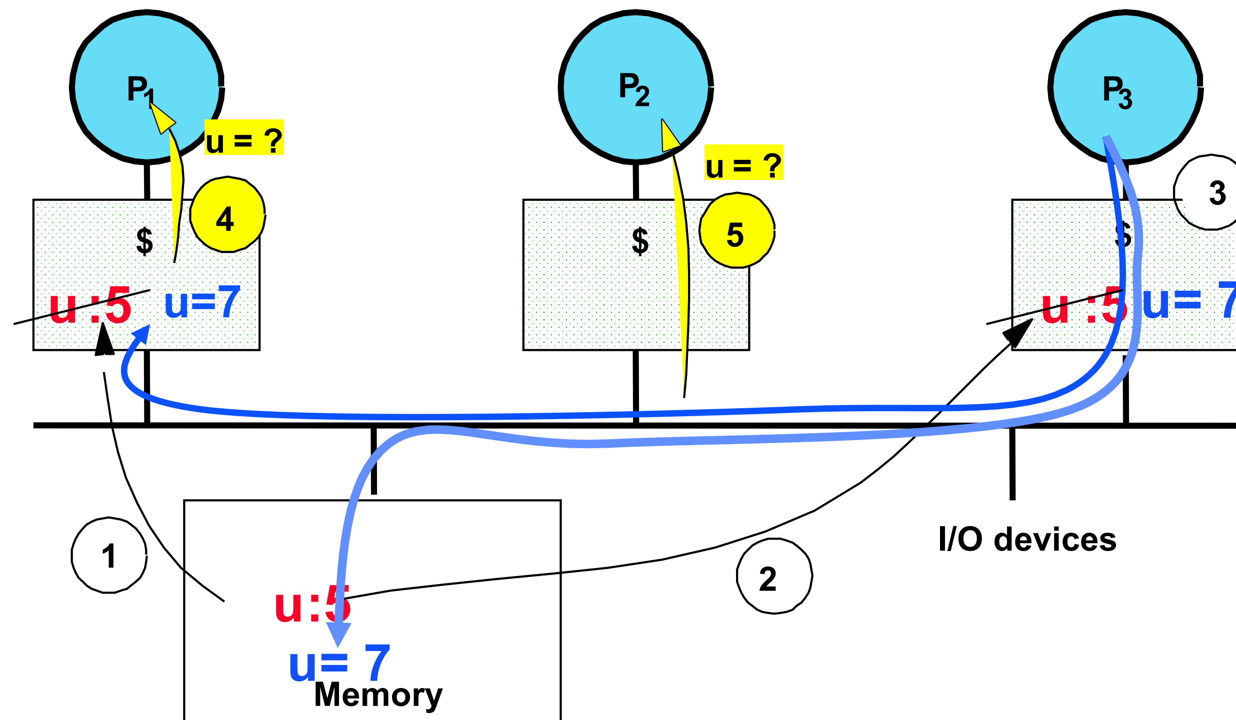


Example: Write Invalidate

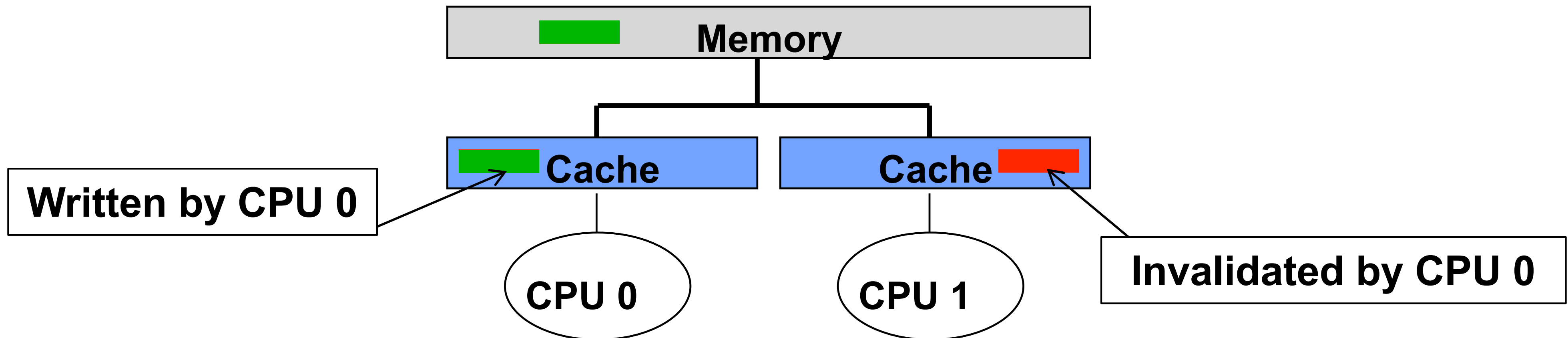


Write-Update (Broadcast)

- Update all the cached copies of a data item when that item is written.
  - Even a processor may not need the updated copy in the future
- Consumes considerably more bandwidth
- Recent multiprocessors have opted to implement a write invalidate protocol



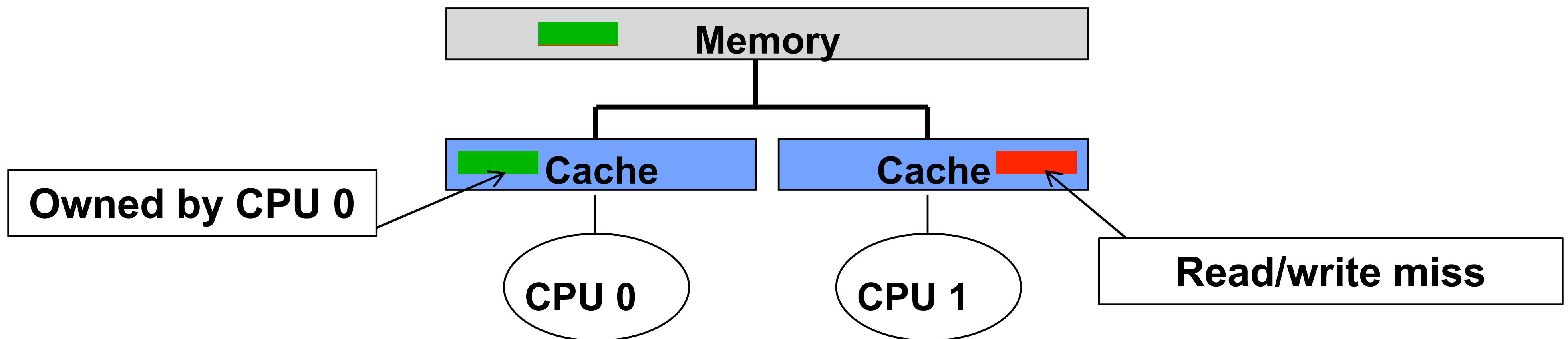
Implementation of Cache Coherence Protocol -- 1



- When data are coherent, the cache block is shared
  - "Memory" could be the last level shared cache, e.g. shared L3

1. When there is a write by CPU 0, Invalidate the shared copies in the cache of other processors/cores
   - Copy in CPU 0's cache is exclusive/unshared,
   - CPU 0 is the owner of the block
   - For write-through cache, data is also written to the memory
     - » Memory has the latest
   - For write-back cache: data in memory is obsoleted
   - For snooping protocol, invalidate signals are broadcasted by CPU 0
     - » CPU 0 broadcasts; and CPU 1 snoops, compares and invalidates

Implementation of Cache Coherence Protocol -- 2



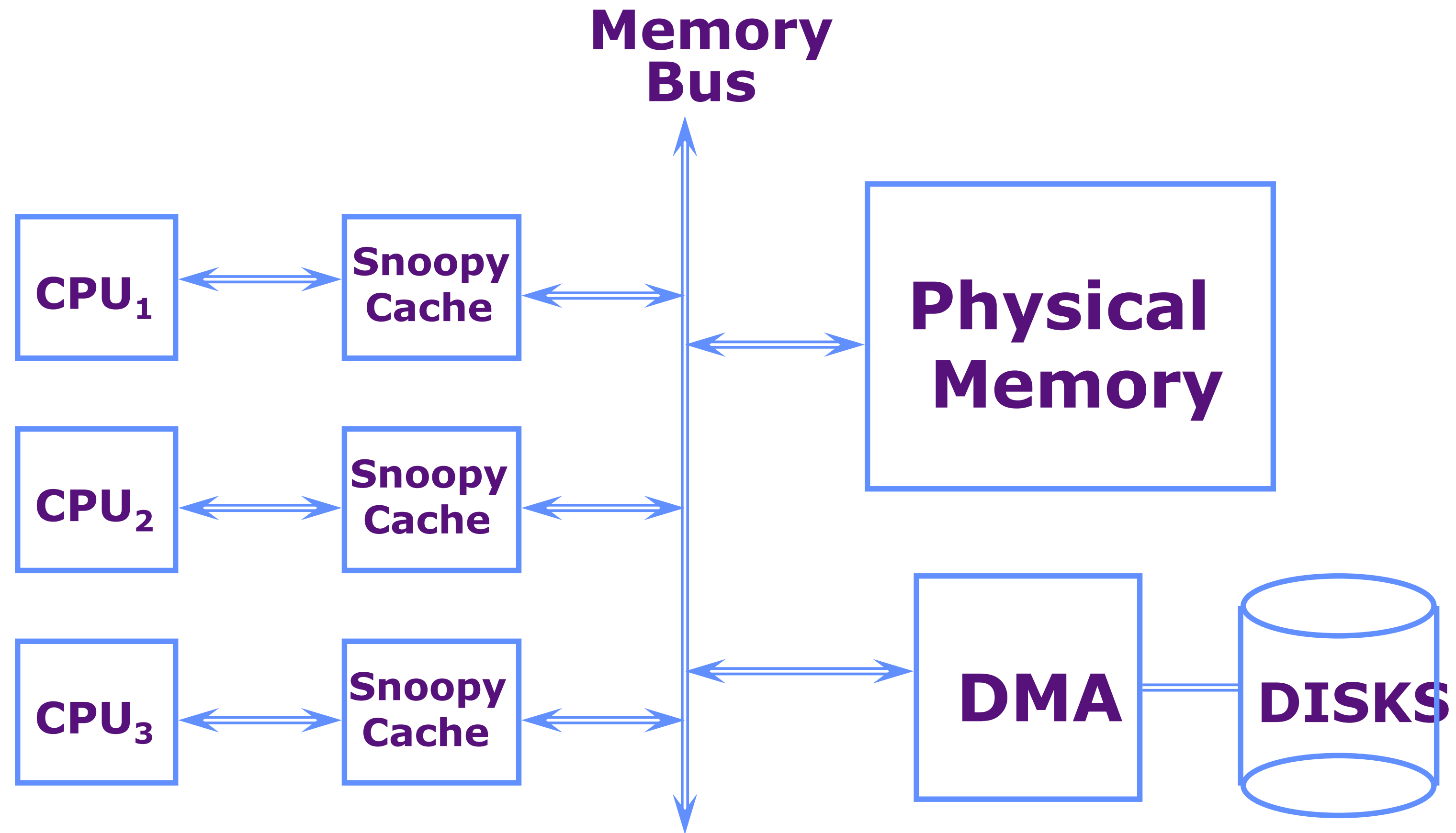
- CPU 0 owned the block (exclusive or unshared)

2. When there is a read/write by CPU 1 or others → Miss since already invalidated
   - For write-through cache: read from memory
   - For write-back cache: supply from CPU 0 and abort memory access
   - For snooping: CPU 1 broadcasts mem request because of a miss; CPU 0 snoops, compares and provides cache block (aborts the memory request)

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

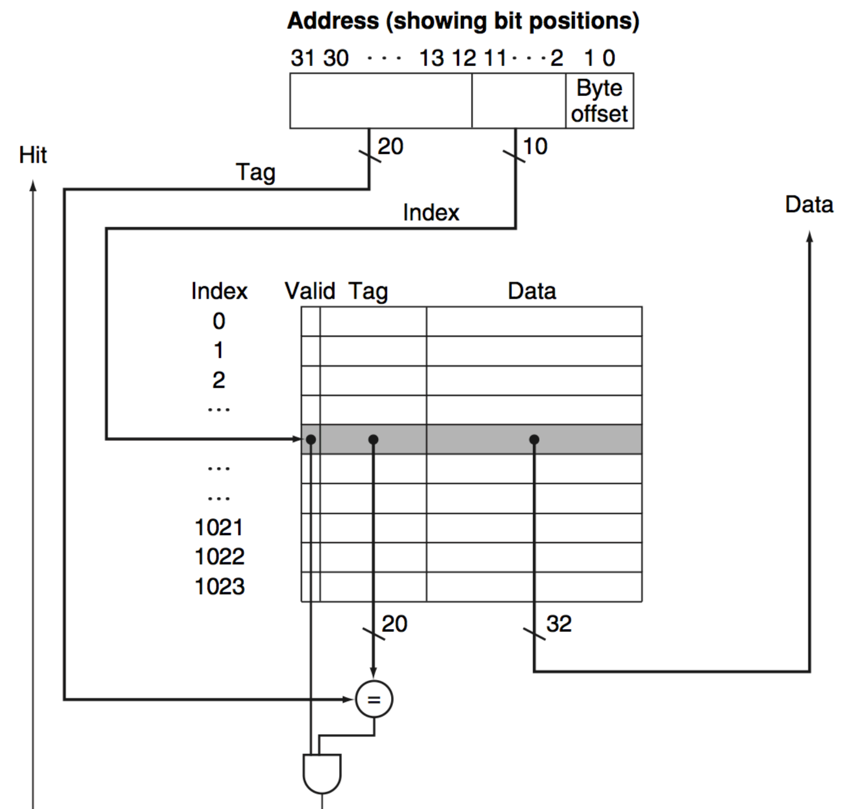
Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Cache Line for Snooping

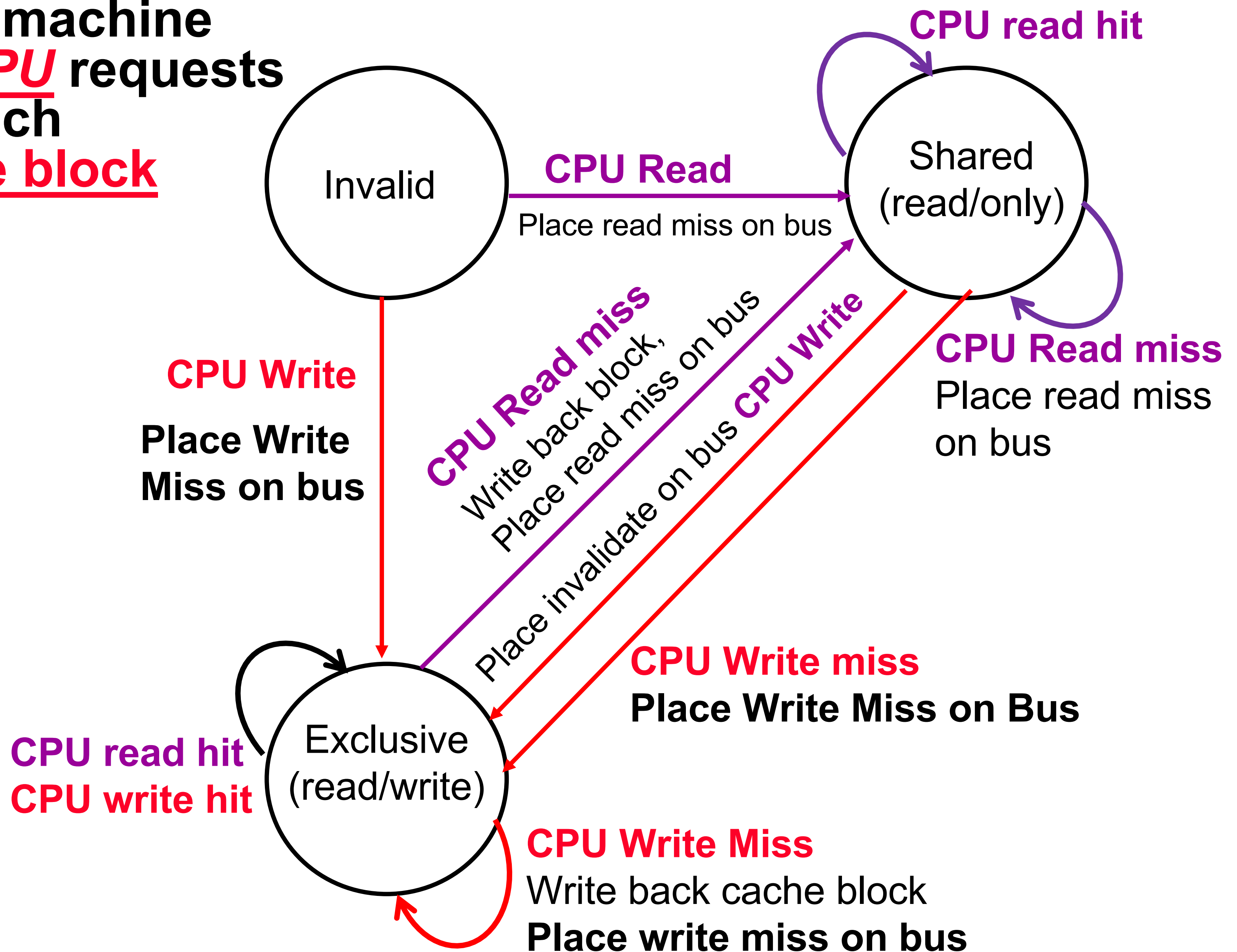
- Cache tags for implementing snooping
  - Compares the addresses on the bus with the tags of the cache line
- Valid bit for being invalidated
- State bit for shared/exclusive



- We will use write-back cache
  - Lower bandwidth requirement than write-through cache
  - Dirty bit for write-back
  - Write-buffer complicates things

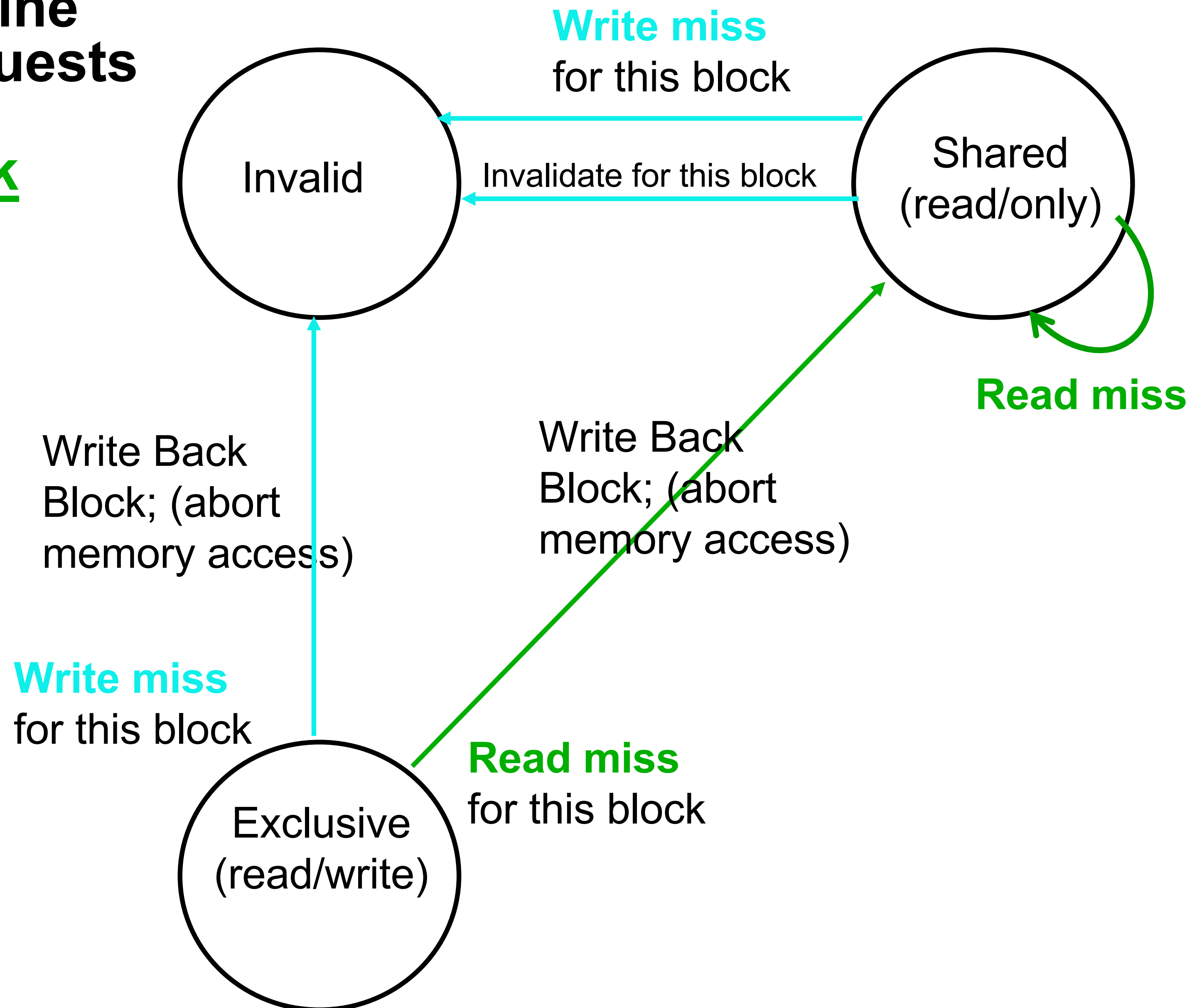
Snoopy-Cache State Machine-I

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



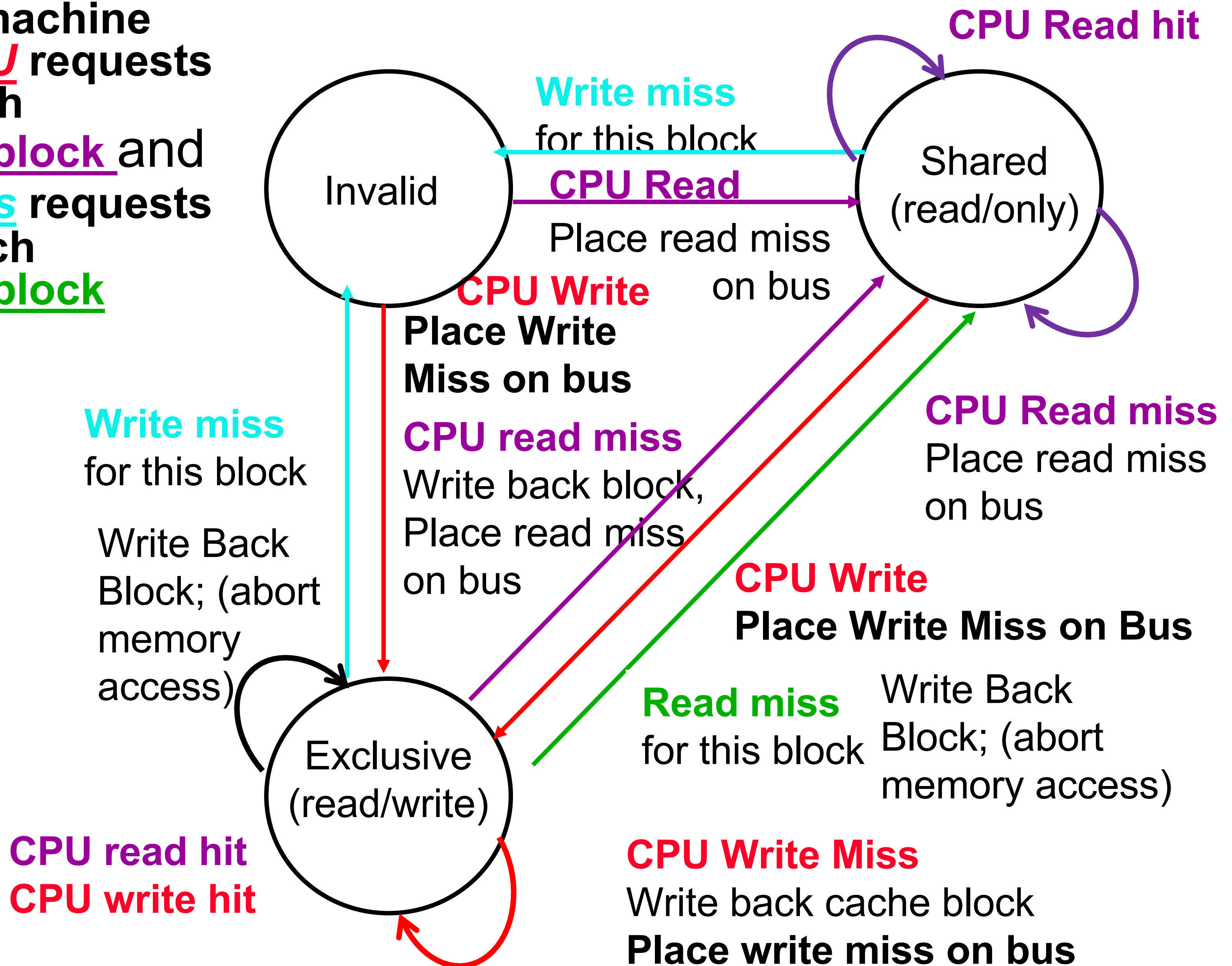
Snoopy-Cache State Machine-II

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



Snoopy-Cache State Machine-III

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



State Table of Snoopy Protocol

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Figure 5.5 The cache coherence mechanism receives requests from both the core's processor and the shared

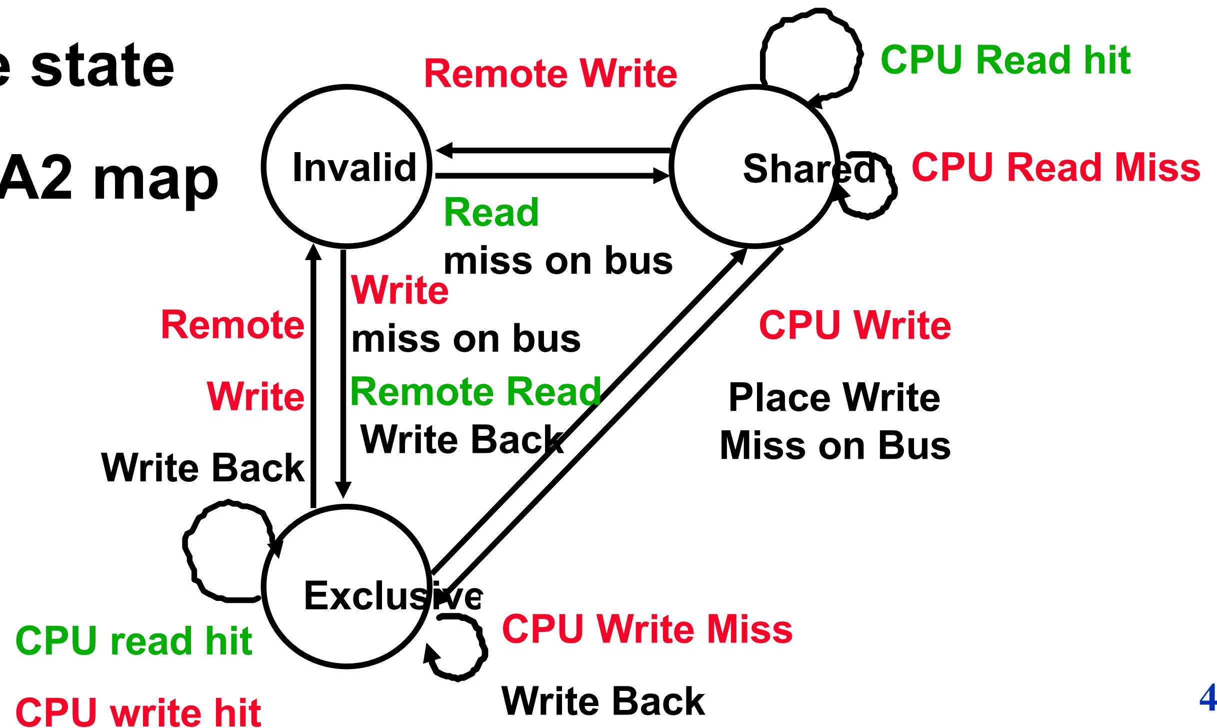
Example

	Processor 1			Processor 2			Bus				Memory	
	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to 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



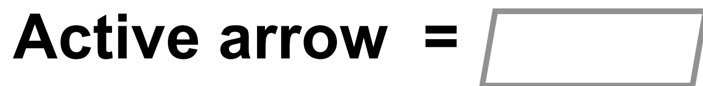
Example: Step 1

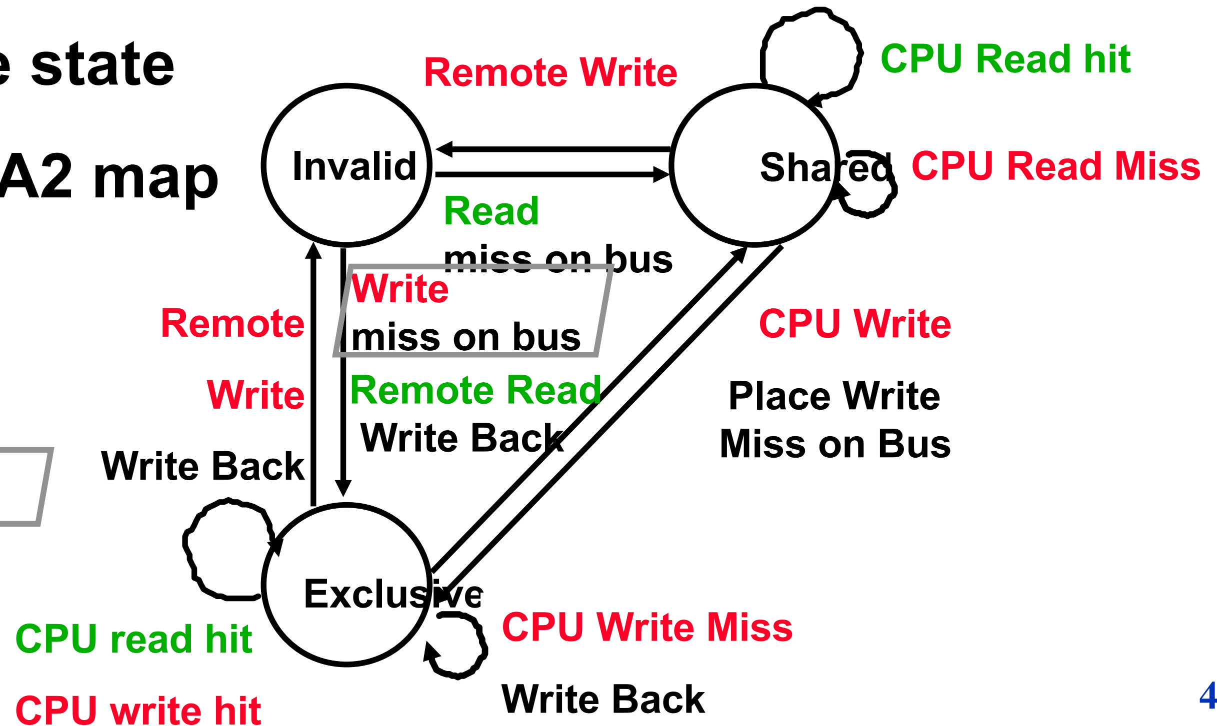
	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2.

Active arrow = 



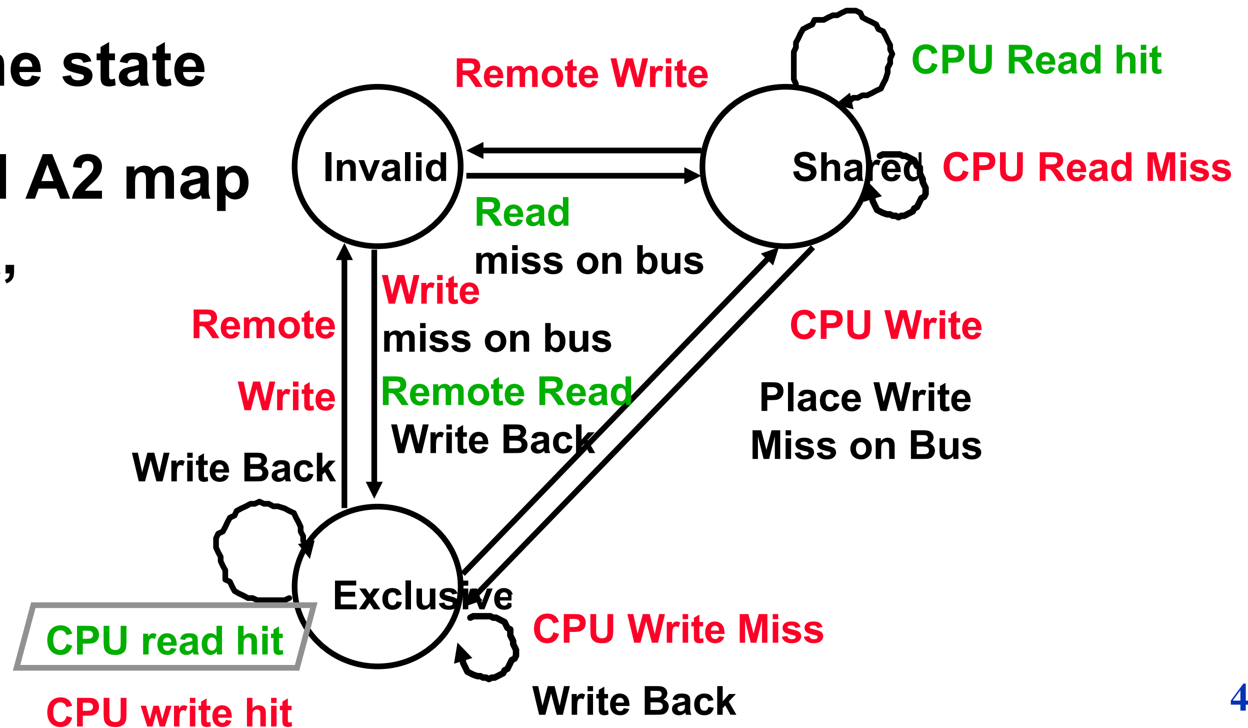
Example: Step 2

	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



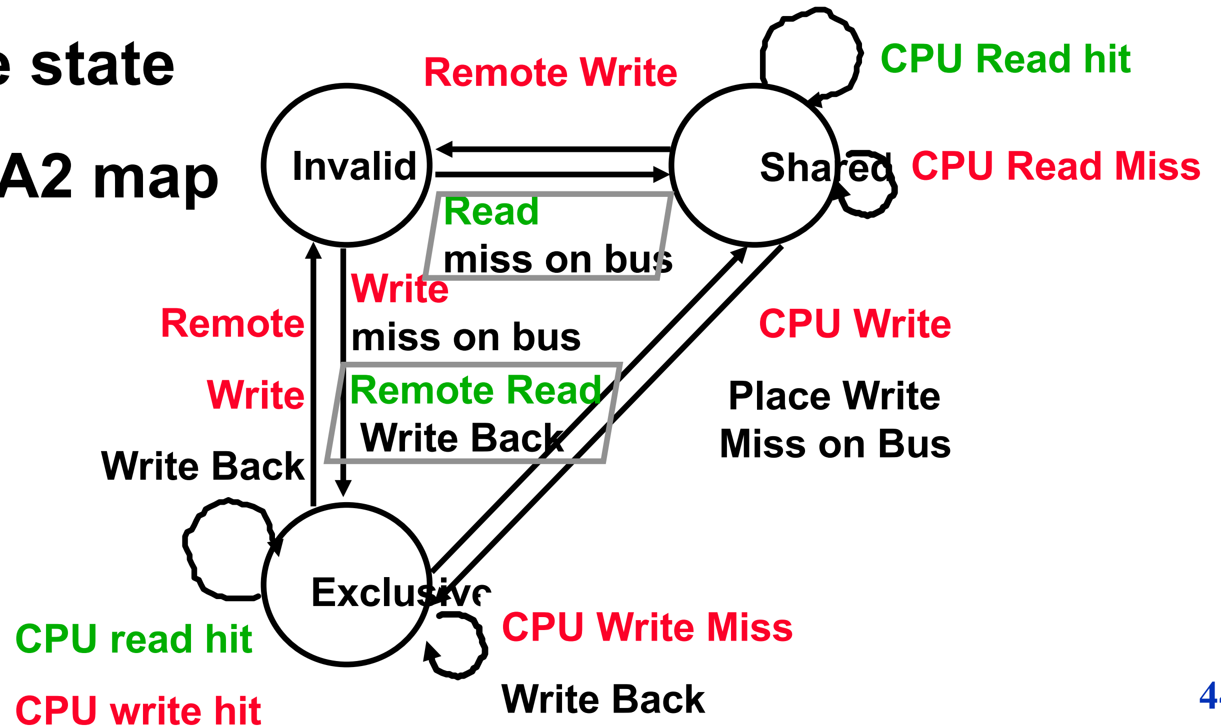
Example: Step 3

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 initial cache state

is invalid and A1 and A2 map to same cache block,

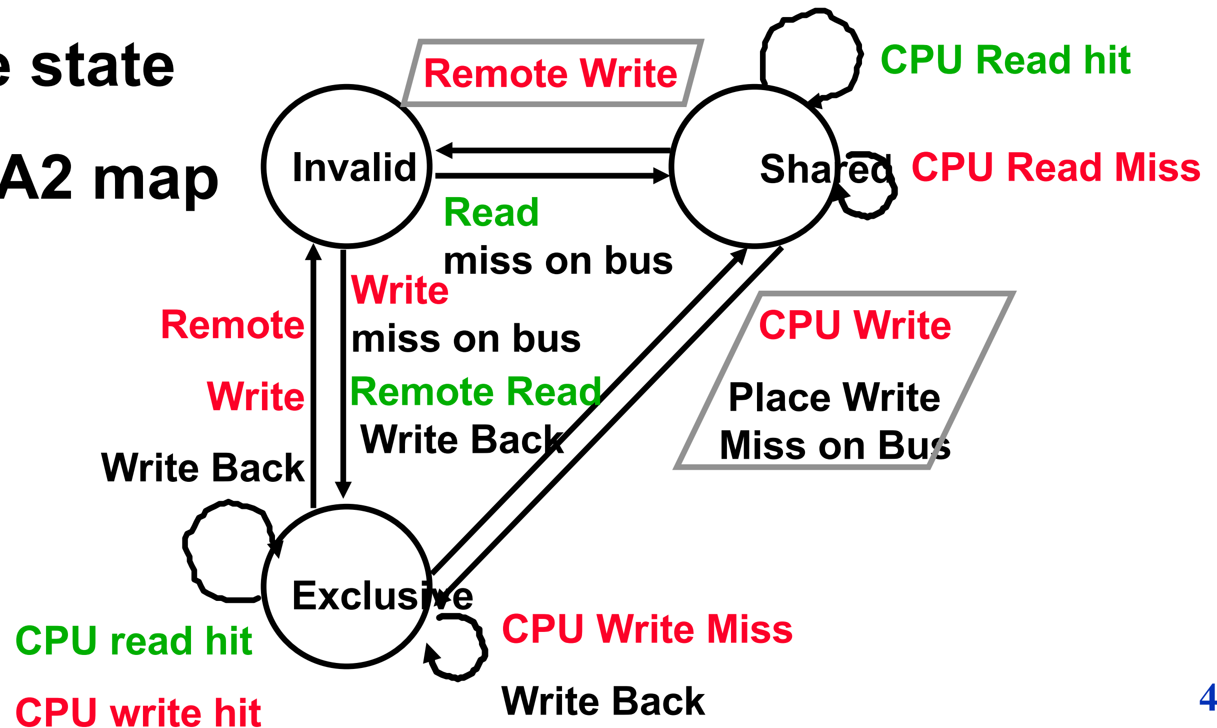
but A1 != A2.



Example: Step 4

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		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



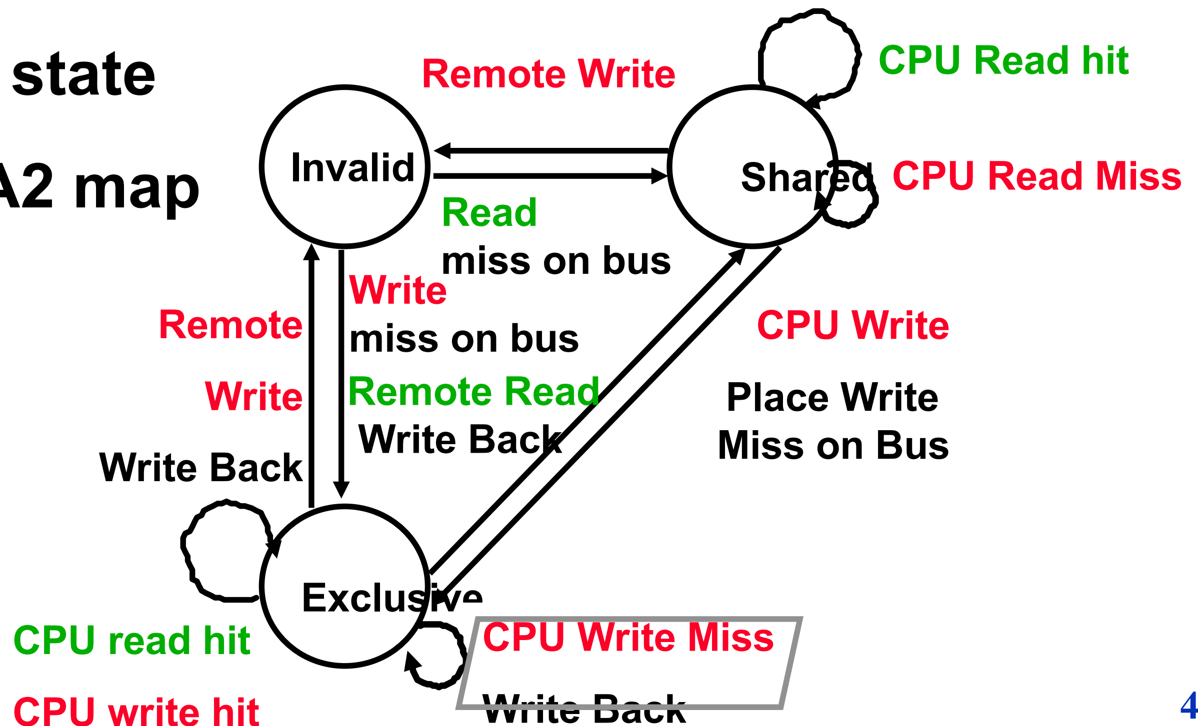
Example: Step 5

	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



Categories of cache misses

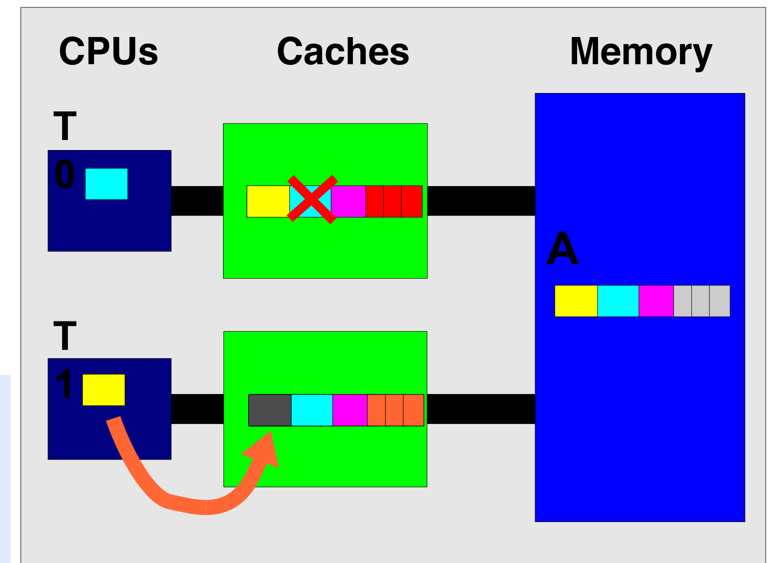
- **Up to now:**
  - **Compulsory Misses: first access to a block cannot be in the cache (cold start misses)**
  - **Capacity Misses: cache cannot contain all blocks required for the execution**
  - **Conflict Misses: cache block has to be discarded because of block replacement strategy**
- **In multi-processor systems:**
  - **Coherence Misses: cache block has to be discarded because another processor modified the content**
    - » **true sharing miss: another processor modified the content of the request element**
    - » **false sharing miss: another processor invalidated the block, although the actual item of interest is unchanged.**

False Sharing

- A cache line contains more than one word
- Cache-coherence is done at the line-level and not word-level
- Suppose M1 writes wordi and M2 writes wordk and
  – both words have the same line address.
- What can happen?

Avoid False Sharing in Programming

- **False sharing**
  - **When at least one thread write to a cache line while others access it**
    - » **Thread 0: = A[1]   (read)**
    - » **Thread 1: A[0] = … (write)**




```
int a[16];  // can fill in a full 64-byte cache line

#pragma omp parallel for num_threads (16)
schedule(static,1)

for(int i=0; i<16; i++) a[i] +=i;
```

- **Solution: use array padding**

```
int a[16][16];

#pragma omp parallel for num_threads (16) schedule(static,1)

for(int i=0; i<16; i++) a[i][0] +=i;
```

Class Lectures End Here!

Performance

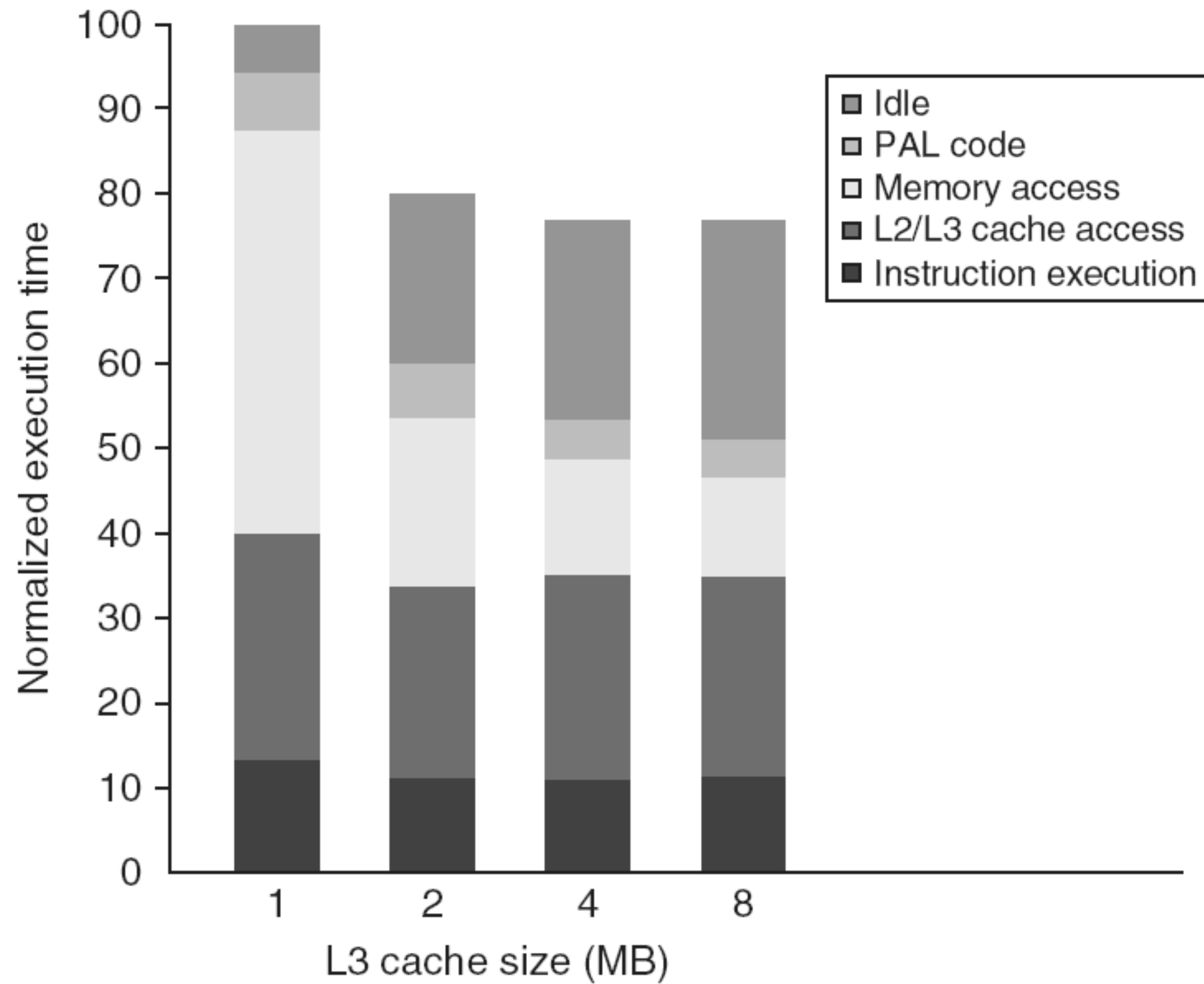
- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

Example: True v. False Sharing v. Hit?

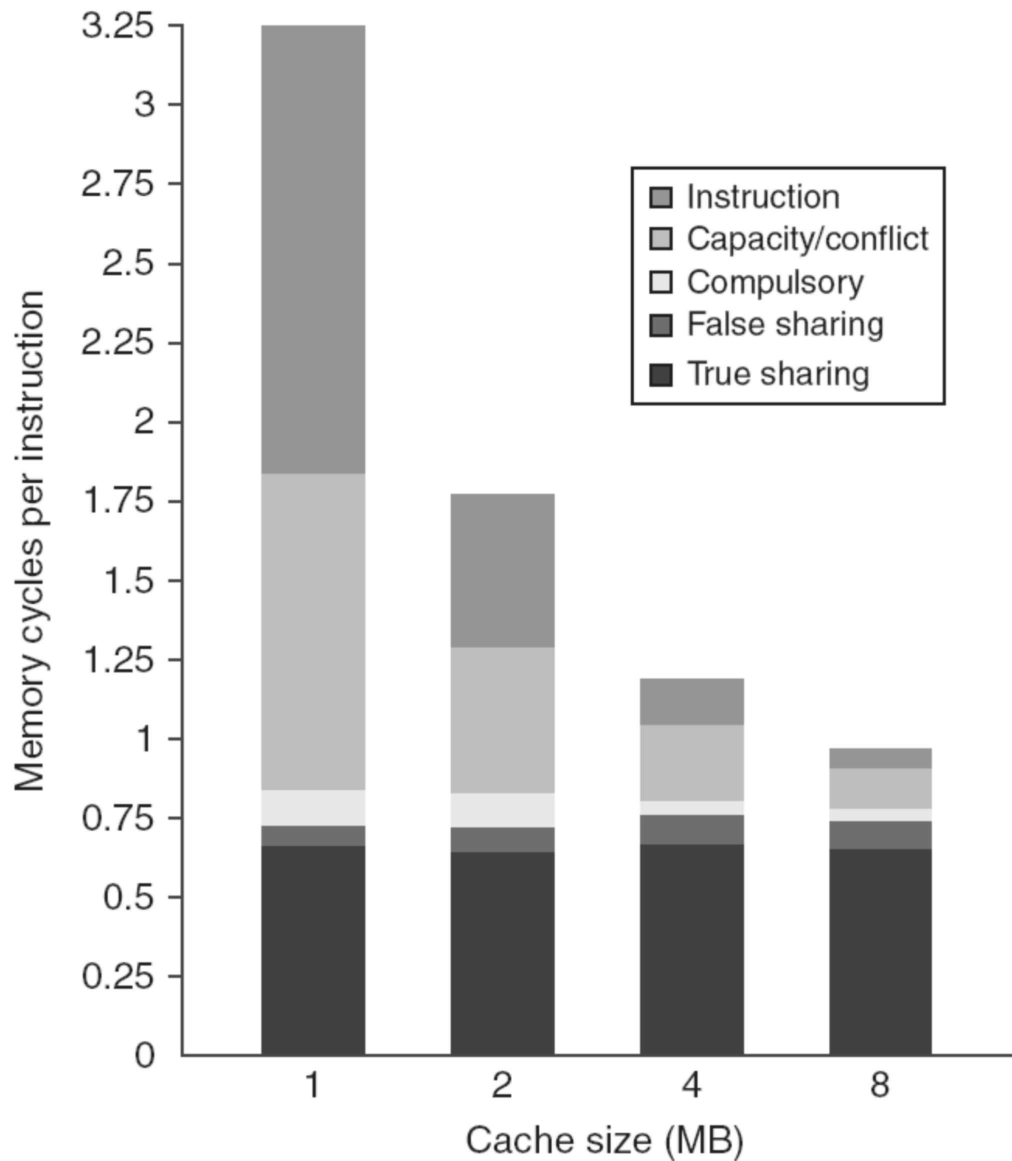
- **Assume x1 and x2 in same cache line.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x2 irrelevant to P1
4		Write x2	False miss; invalidate x2 in P1
5	Read x2		True miss; invalidate x2 in P1

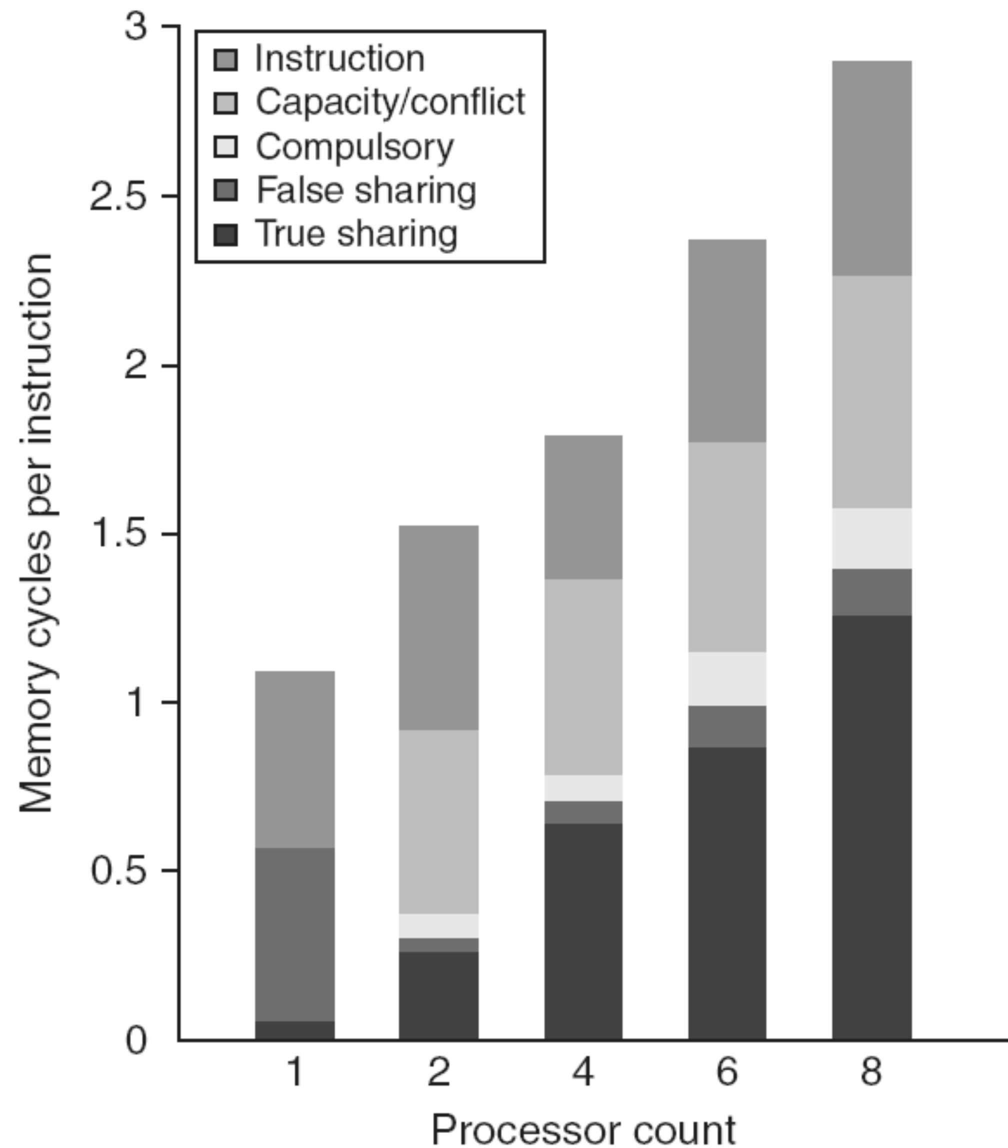
Performance Study: Commercial Workload



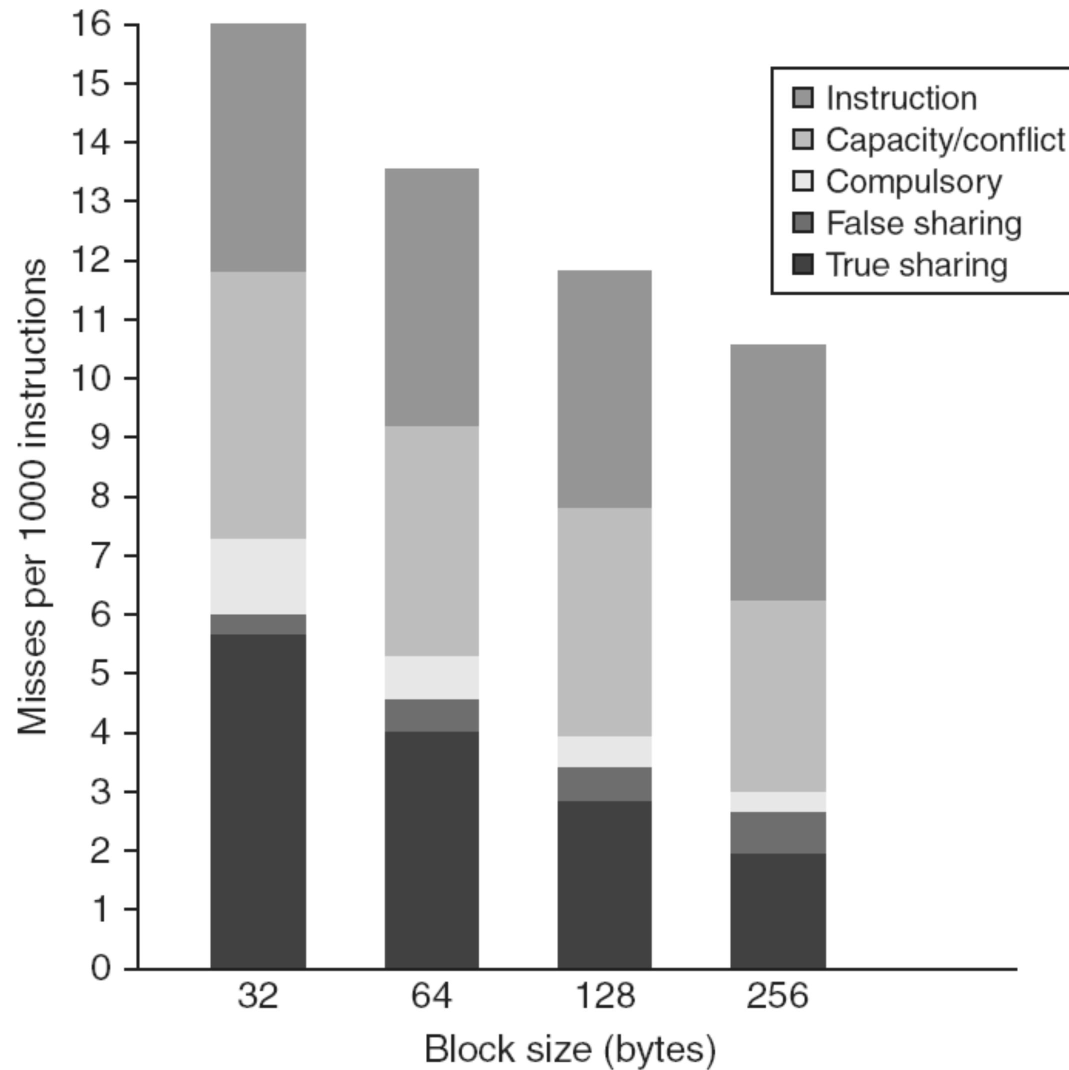
Performance Study: Commercial Workload



Performance Study: Commercial Workload



Performance Study: Commercial Workload



Snooping Cache Variations

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

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 cache access:
  - 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

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 of the cache block and its dirty, so must reply
  - Add extra state bit to cache to determine shared or not
  - Add 4th state (MESI)

Complications

- The simple cache protocol is correct, it omits a number of complications that make the implementation much trickier.
- Operations are *atomic*
  - For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality this is not true.
  - In fact, even a read miss might not be atomic; after detecting a miss in the L2 of a multi-core, the core must arbitrate for access to the bus connecting to the shared L3.
- Nonatomic actions introduce the possibility that the protocol can *deadlock*
- With multicore processors, the coherence among the processor cores is all implemented on chip, using either a snooping or simple central directory protocol.

Extensions to Snoopy: MESI

- *MESI:* adds the state Exclusive to the basic MSI
  - Exclusive: indicate when a cache block is resident only in a single cache but is clean.
  - If a block is in the E state, it can be written without generating any invalidates
    - optimizes the case where a block is read by a single cache before being written by that same cache.
  - When a read miss to a block in the E state occurs, the block must be changed to the S state to maintain coherence.
  - The advantage: a subsequent write to a block in the exclusive state by the same core need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this local cache; the processor merely changes the state to modified.
  - This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a bock is modified.
  - The popular MESI protocol (Modified, Exclusive, Shared, and Invalid), is used in Intel i7 as a variant called MESIF, which adds a state (Forward) to designate which sharing processor should respond to a request. It is designed to enhance performance in distributed memory organizations.

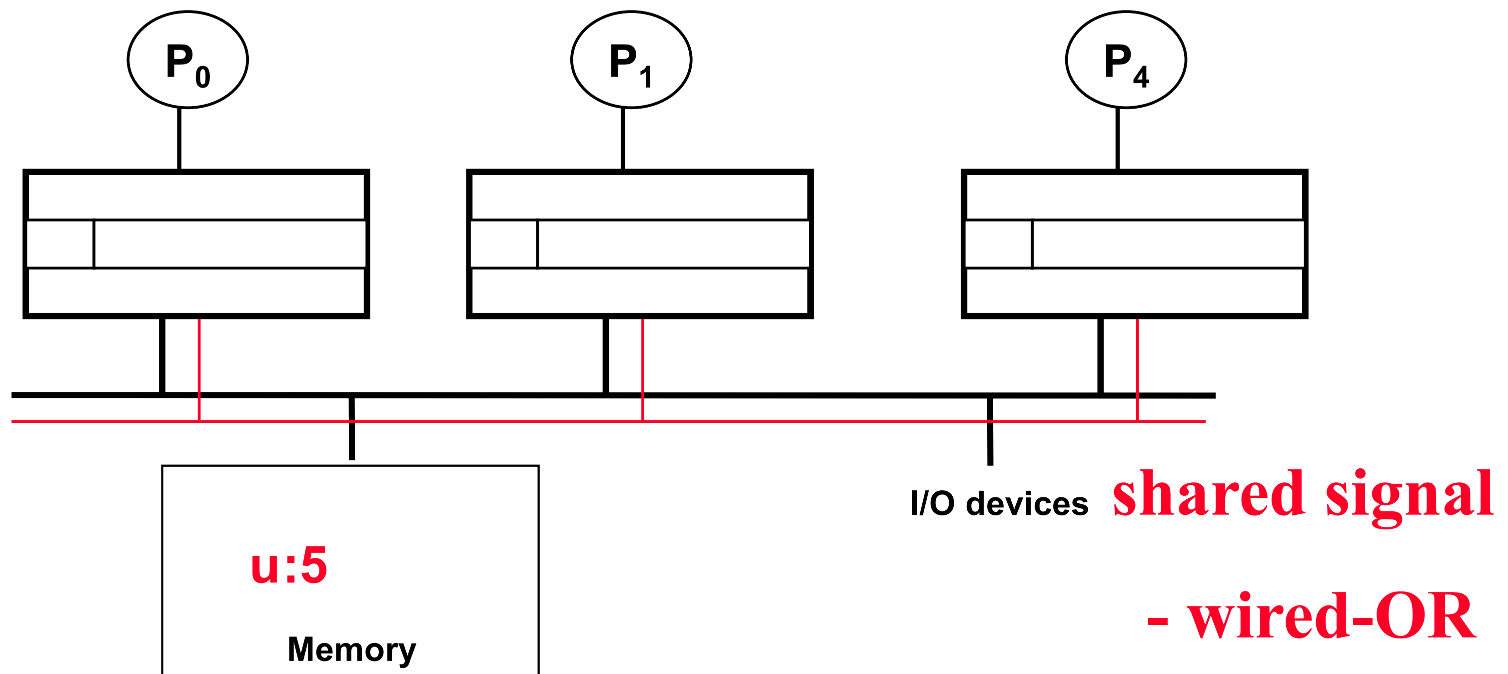
MESI (4-state) Invalidation Protocol

- Four States:
  - "M": "Modified"
  - "E": "Exclusive"
  - "S": "Shared"
  - "I": "Invalid"
- Add *exclusive* state
  - distinguish exclusive (writable) and owned (written)
  - Main memory is up to date, so cache not necessarily owner
  - can be written locally
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if no cache has copy

  => How can you tell?

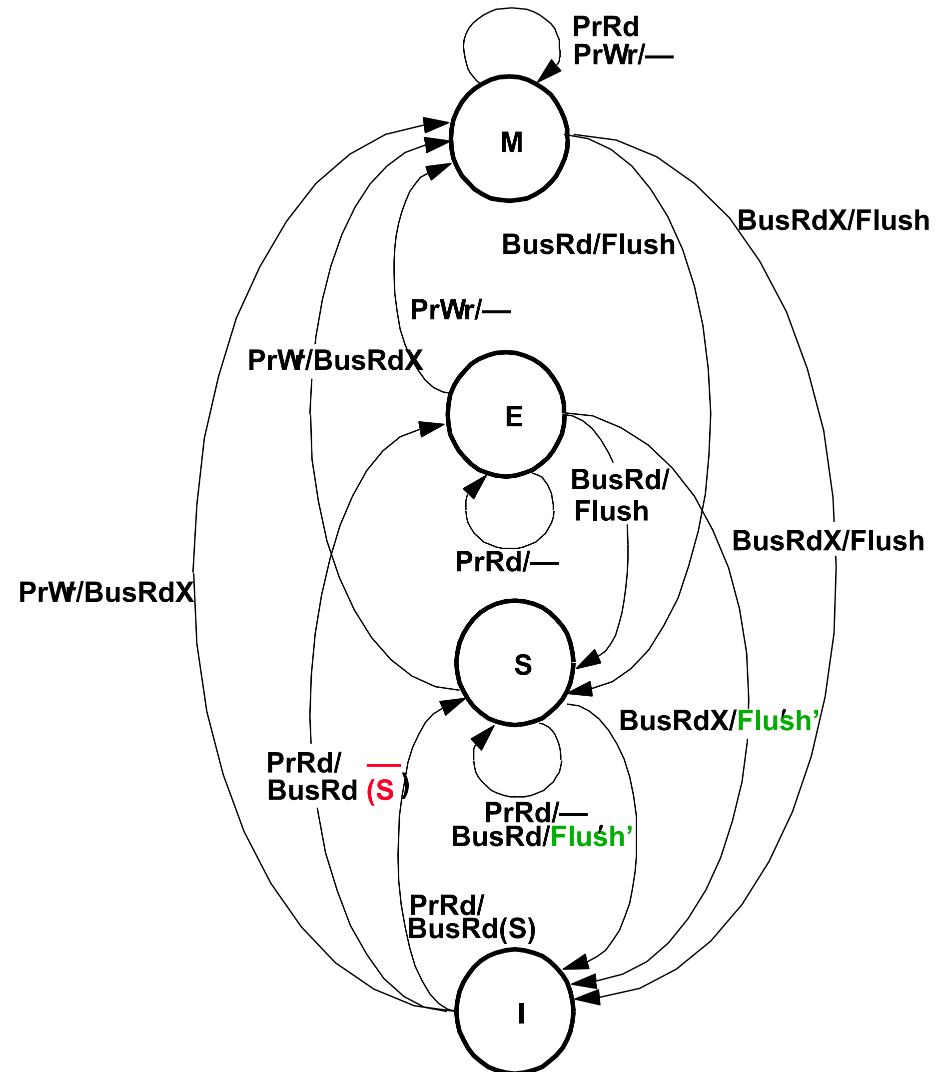
Hardware Support for MESI

- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
  - how does it know when all have voted?



MESI State Transition Diagram

- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
  - only one cache flushes data
- Replacement:
  - S→I can happen without telling other caches
  - E→I, M→I
- MOESI protocol: Owned state: exclusive but memory not valid



MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid

Write miss

M

E

S

I

P_1 write or read

P_1 write

P_1 read

Read miss, not shared

P_1 intent to write

Other processor reads

Other processor reads
P_1 writes back

Other processor intent to write

Other processor intent to write, P1 writes back

Read miss, shared

Read by any processor

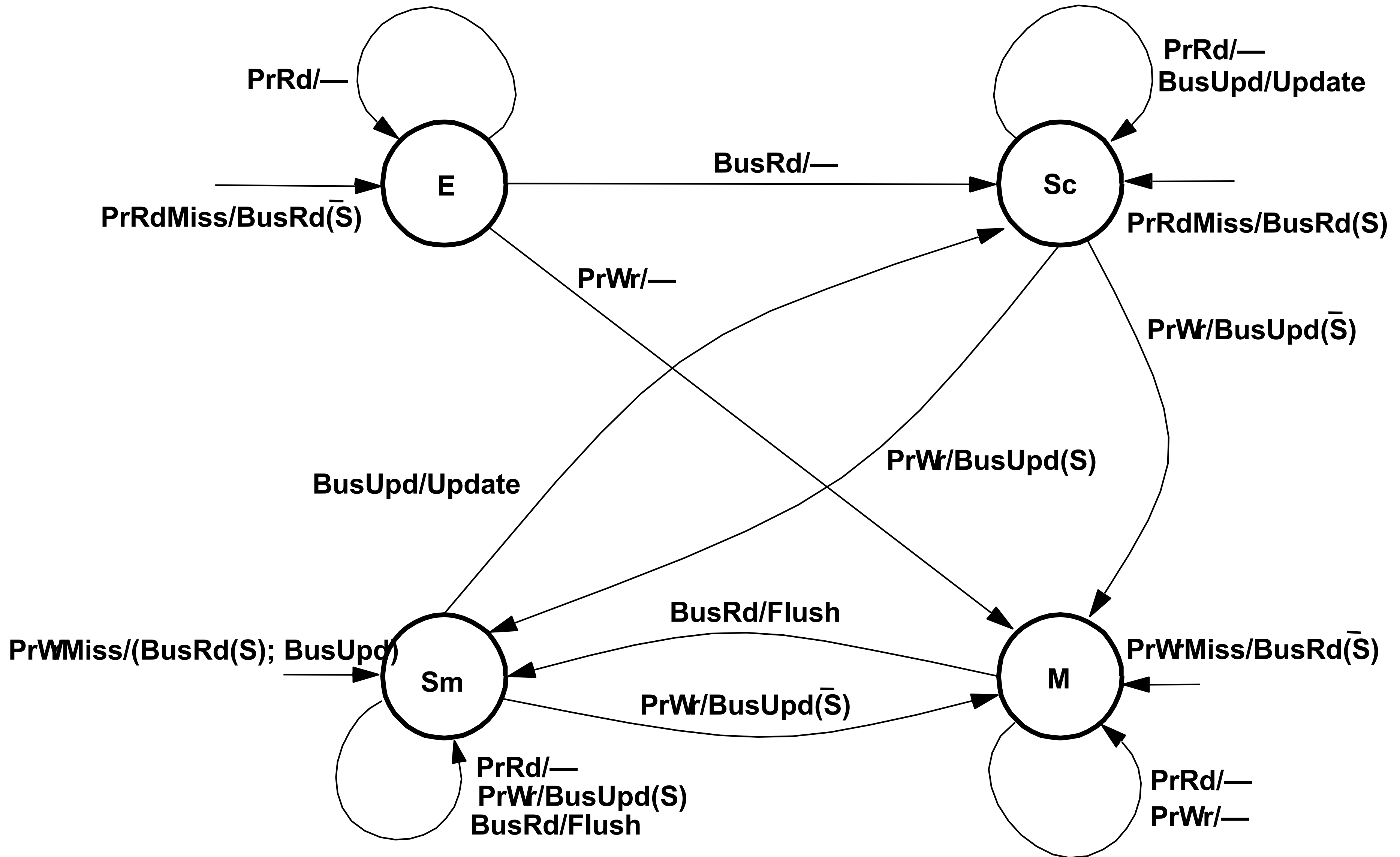
Other processor intent to write

Cache state in processor P_1

Dragon Write-back Update Protocol

- **4 states**
  - Exclusive-clean or exclusive (E): I and memory have it
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner
  - Shared modified (Sm): I and others but not memory, and I'm the owner
    - Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, noone else
- **No invalid state**
  - If in cache, cannot be invalid
  - If not present in cache, view as being in not-present or invalid state
- **New processor events: PrRdMiss, PrWrMiss**
  - Introduced to specify actions when block not present in cache
- **New bus transaction: BusUpd**
  - Broadcasts single word written on bus; updates other relevant caches

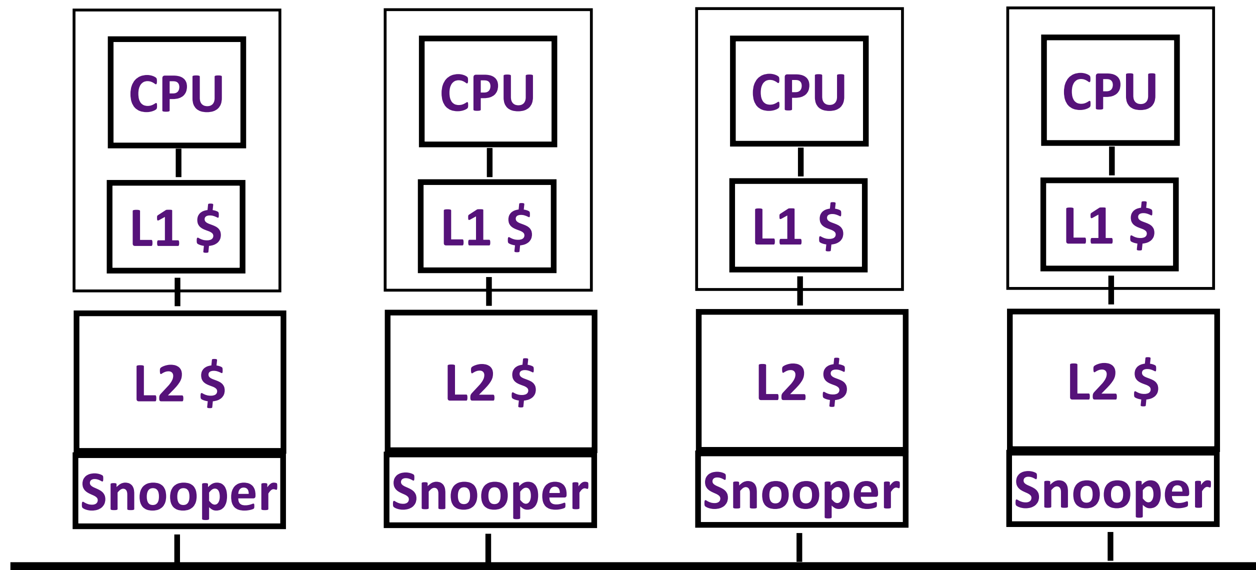
Dragon State Transition Diagram



Extensions to Snoopy: MOESI

- ***Add the*** **state Owned to the MESI protocol**
  - **indicate that the associated block is owned by that cache and out-of-date in memory.**
- **In MSI and MESI protocols, when there is an attempt to share a block in the Modified state, the state is changed to Shared (in both the original and newly sharing cache), and the block must be written back to memory. In a MOESI protocol, the block can be changed from the Modified to Owned state in the original cache without writing it to memory. Other caches, which are newly sharing the block, keep the block in the Shared state; the O state, which only the original cache holds, indicates that the main memory copy is out of date and that the designated cache is the owner. The owner of the block must supply it on a miss, since memory is not up to date and must write the block back to memory if it is replaced. The AMD Opteron uses the MOESI protocol.**

Optimized Snoop with Level-2 Caches

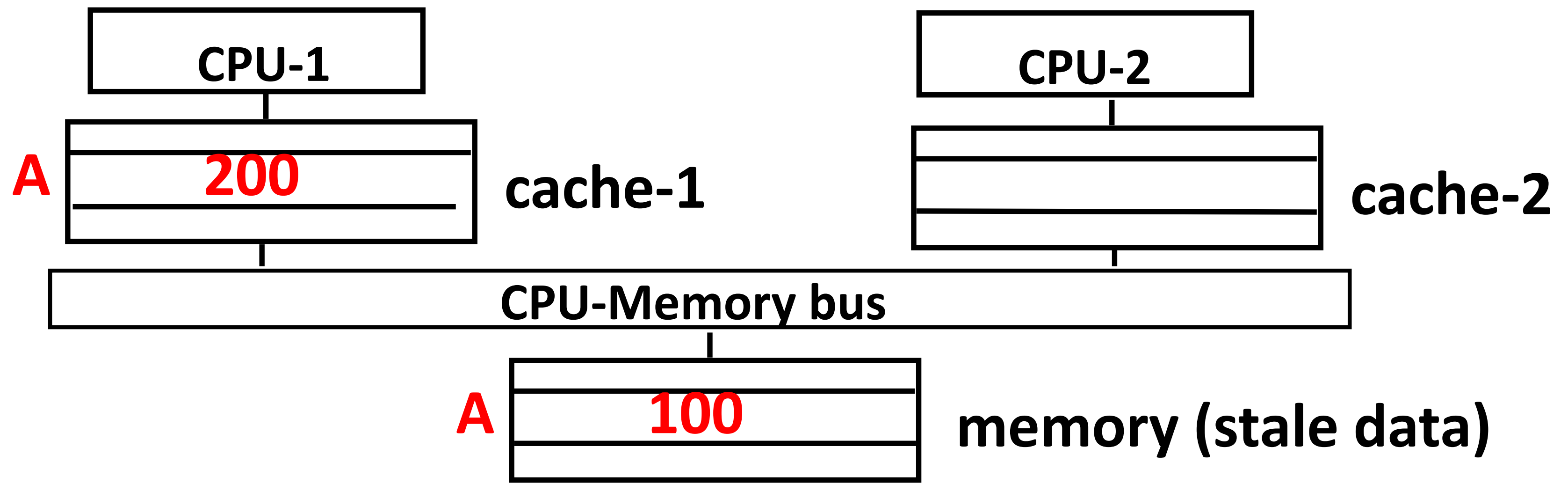


- **Processors often have two-level caches**
  - **small L1, large L2 (on chip)**
- ***Inclusion property:*** **entries in L1 must be in L2**

  invalidation in L2 ⇒ invalidation in L1
- **Snooping on L2 does not affect CPU-L1 bandwidth**

What problem could occur?

Intervention



When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

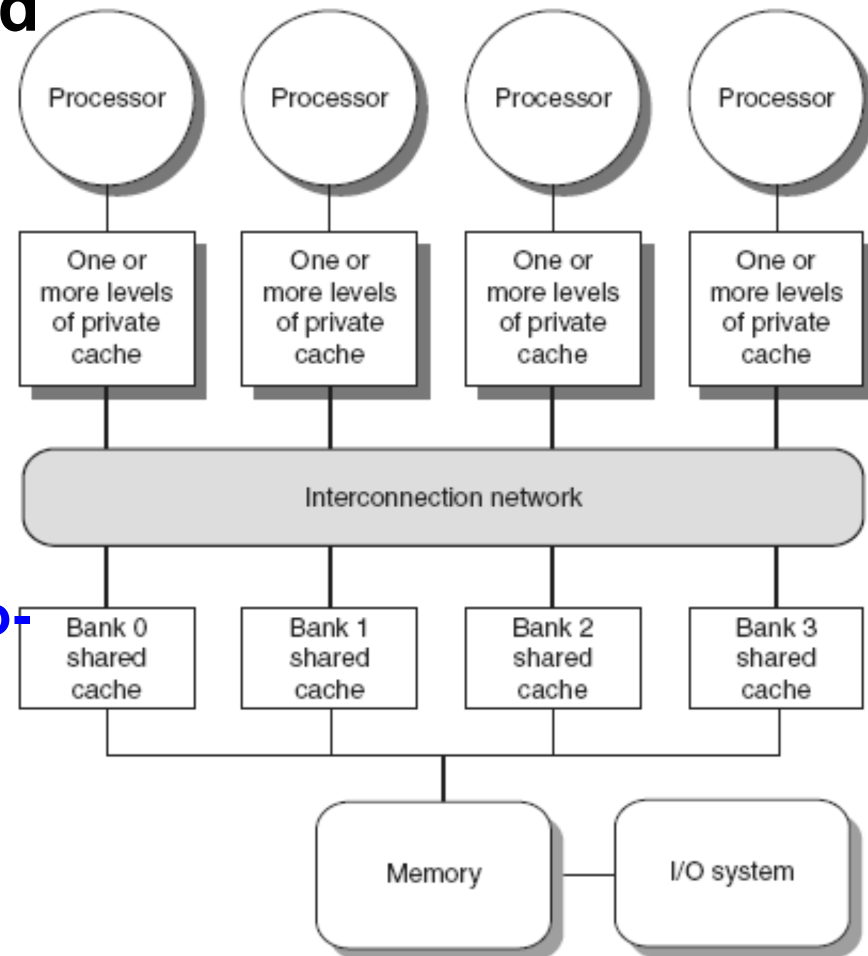
- **Cache-1 needs to supply & change its state to shared**
- **The memory may respond to the request also!**

Does memory know it has stale data?

Cache-1 needs to intervene through memory controller to supply correct data to cache-2

Coherence Protocols: Extensions

- **Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors**
  - **Duplicating tags**
  - **Place directory in outermost cache**
  - **Use crossbars or point-to-point networks with banked memory**



Coherence Protocols

- **AMD Opteron:**
  - **Memory directly connected to each multicore chip in NUMA-like organization**
  - **Implement coherence protocol using point-to-point links**
  - **Use explicit acknowledgements to order operations**