Κεφ. 5: Πολυεπεξεργαστές Ι
Multiprocessors
(σημειώσεις από U Berkeley και το Βιβλίο)

Εαρινό Εξάμηνο 2017

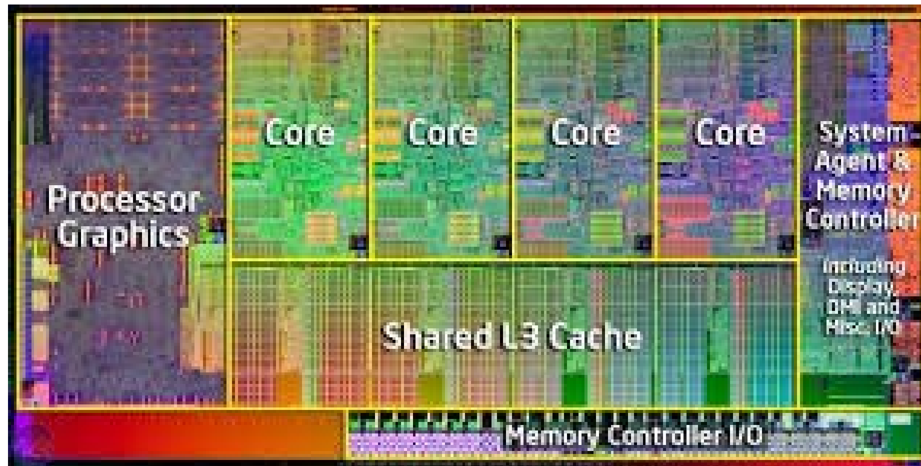
M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

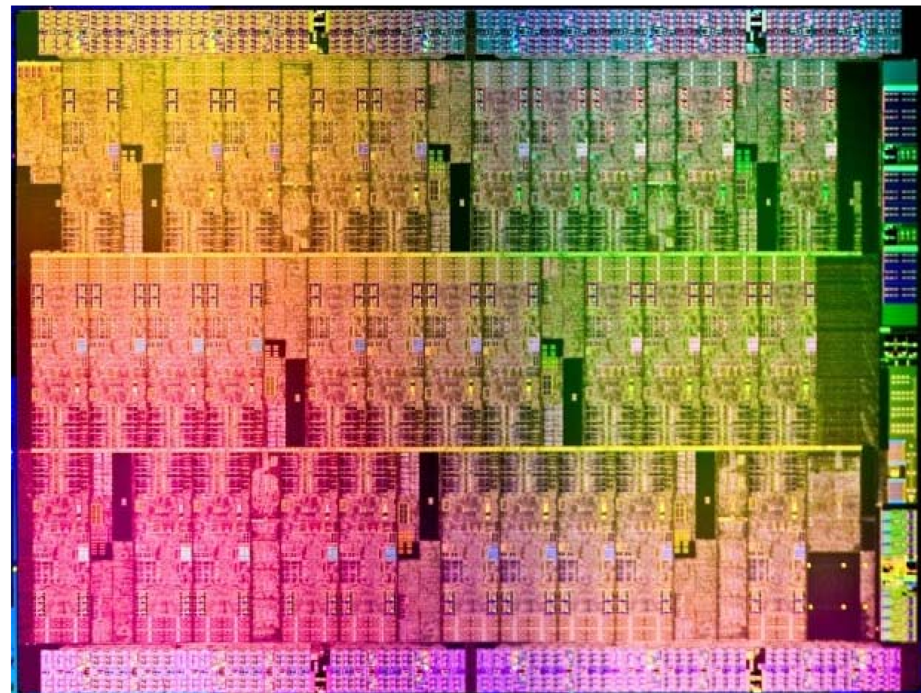
Flynn's Taxonomy

- Flynn classified by data and control streams in 1966

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (single PC: Vector, SIMD ext, GPU)
Multiple Instruction Single Data (MISD) (????)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers, DC)

Multicore, Manycore, Multiple Sockets





Rack, Supercomputers, Data Centers

MIMD, TLP

- Thread-Level parallelism
  - Have multiple program counters (sequencers)
  - With *n* cores, can run *n* threads
- Amount of computation assigned to each thread = grain size
  - For multi-core typical threads large (coarse grain)
  - data-level parallelism fine grain

Who benefits: Parallel Programs

Matrices a[n][n], b[n][n], c[n][n],

```
for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            double s = 0;
            for (int k = 0; k < n; k++) {
                s += a[i][k] * b[k][j];
            }
            c[i][j] = s;
}
```

Assign to each processor fraction of the work (n^3)

Who benefits: Internet Services

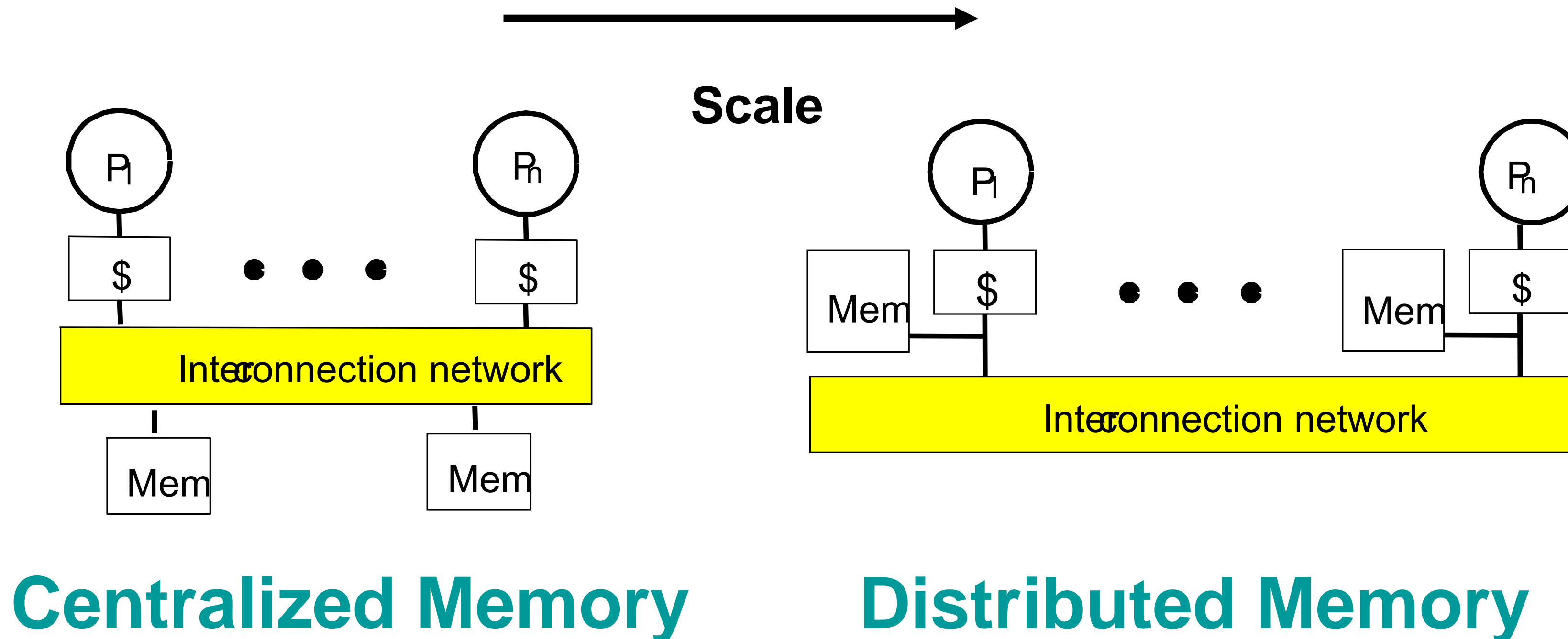
- Numerous requests per second
- Mostly read-only
- Data parallel
  - divided and processed by independent tasks.
- Amenable to computing infrastructures with large number of computing elements and high throughput => Data Centers.
- Data are replicated and partitioned
  - replicated for throughput
  - partitioned for parallel execution and shorter response latency

Basics

- "A parallel computer = collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors (memory view):

1. Centralized Memory Multiprocessor (SMP)
   - Chip Multiprocessors: several cores/chip
   - < few dozen processor chips (and ~100s of cores) in 2017
   - Large caches filter memory requests
   - Small enough to share single, centralized memory
2. Physically Distributed-Memory multiprocessor
   - Larger number processors
   - BW demands ⇒ Memory distributed among processors

Centralized vs. Distributed Memory



- This picture true also for on-chip cache (instead of memory replace with LLC – last level cache and banking)

Distributed Memory Multiprocessor

- Pro: Cost-effective way to scale memory bandwidth (no special interconnect)
  - If most accesses are to local memory
- Pro: Reduces latency of local memory accesses
- Con: Communicating data between processors more complex
- Con: need software support to take advantage of increased memory BW

2 Models for Communication and Memory Architecture

1. Communication occurs through a shared address space (via loads and stores): shared memory multiprocessors two forms:
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP, banking

2. Communication occurs by explicitly passing messages among the processors: message-passing multiprocessors

Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Goal 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 10%

  b. 5%

  c. 1%

  d. <1%

Amdahl’s Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \dfrac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

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

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

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. (Remote access = 200/0.5 = 400 clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?

  a. 1.5X

  b. 2.0X

  c. 2.5X

CPI Equation

- CPI = Base CPI +
  Remote request rate
  x Remote request cost
- CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
- Without communication:1.3/0.5 or 2.6 faster as compared with 0.2% instructions involve remote access

Challenges of Parallel Processing

1. Application parallelism $\Rightarrow$ primarily via new algorithms that have better parallel performance, easy programming language
2. Long remote latency impact $\Rightarrow$ both by architect and by the programmer
   - For example, reduce frequency of remote accesses either by
     - Caching shared data, prefetching, reduce traffic (HW)
     - Restructuring the data layout to make more accesses local (SW)
3. BW Limited CMP: many cores on a chip but not
   enough BW to off-chip DRAM =>
   technology (3D stacking)

- **Today's lecture how HW helps memory latency via caches**

Shared-Memory Architectures

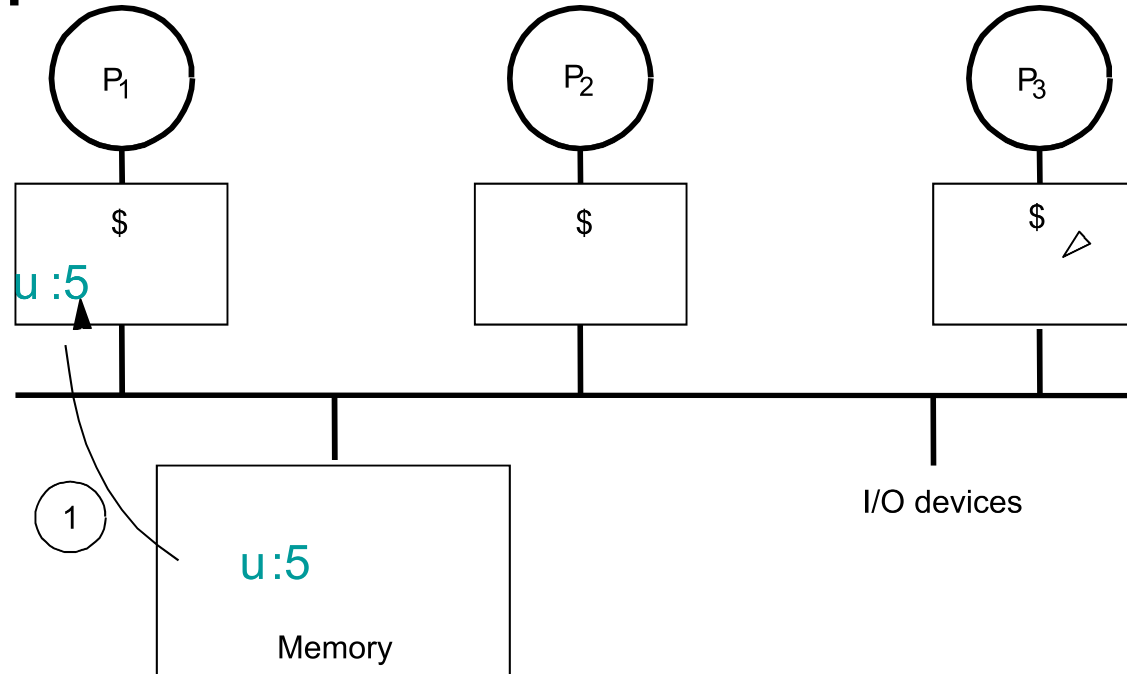
- Caches in multiprocessors hold both
  - Private data used by a single core
  - Shared data used by multiple cores

- Caching shared data (creates duplicates):

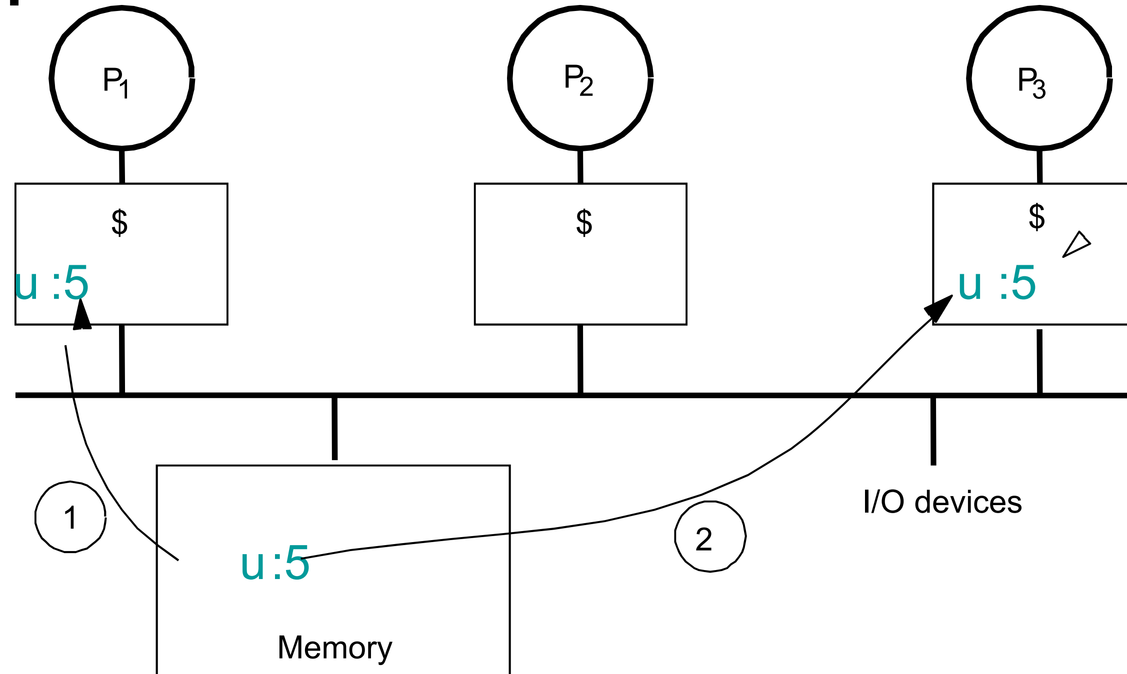
  \+ reduces: latency to shared data, memory bandwidth for shared data and interconnect bandwidth

  \- **cache coherence problem**

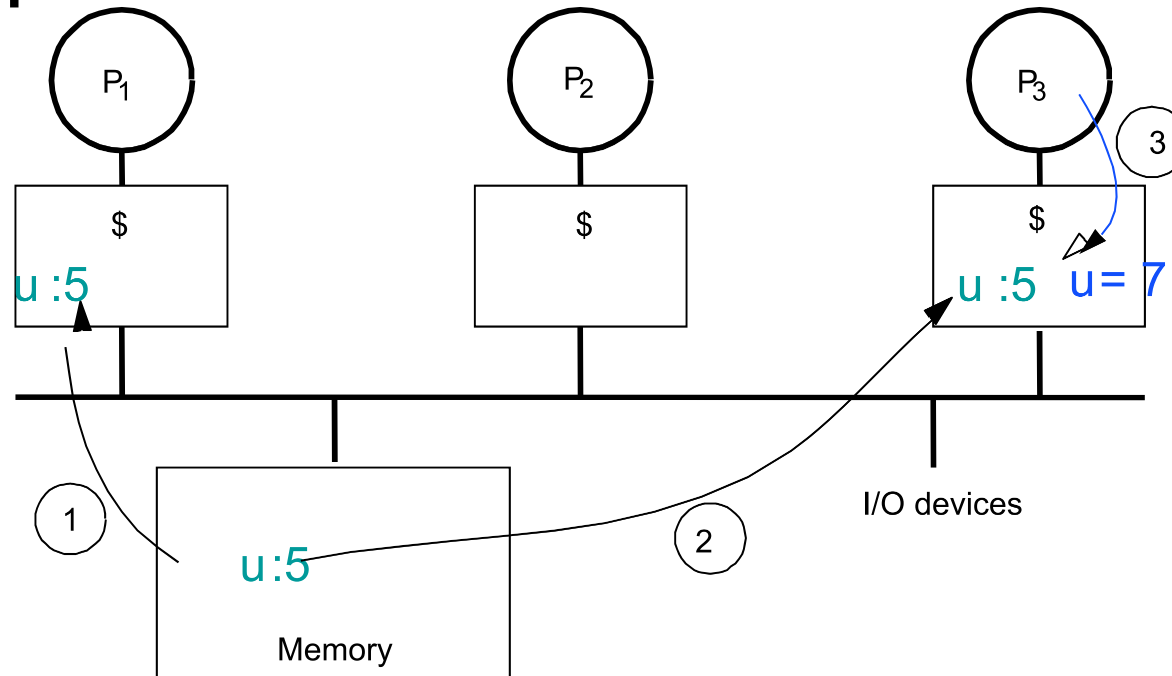
Example Cache Coherence Challenges



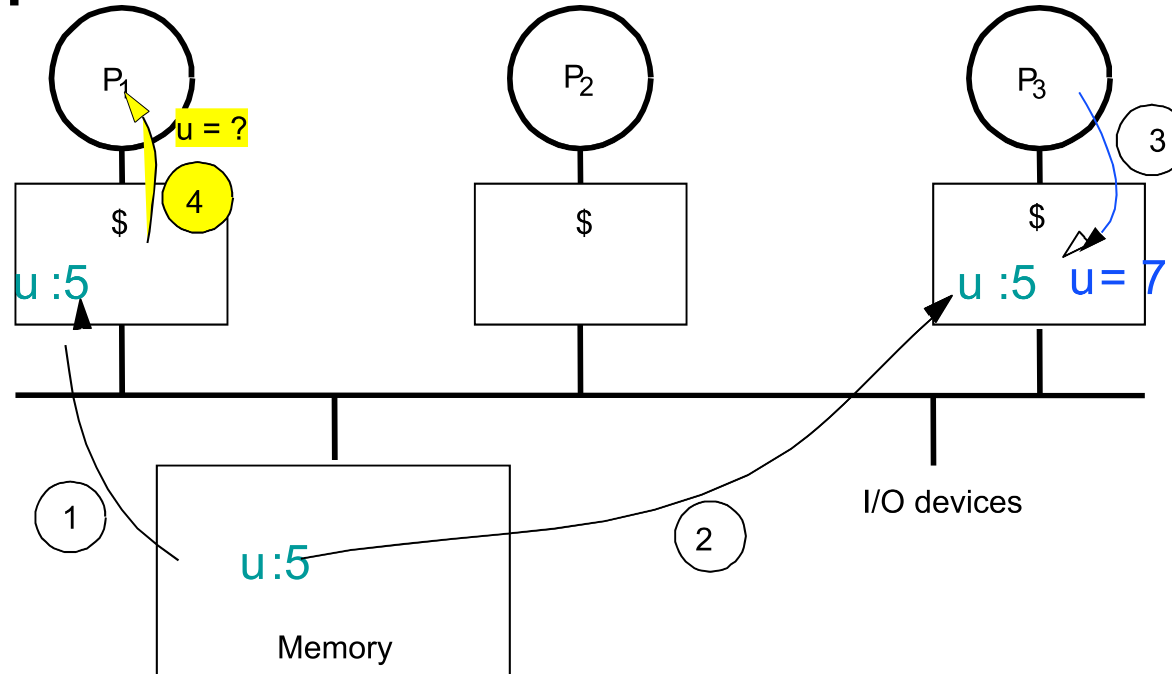
Example Cache Coherence Challenges



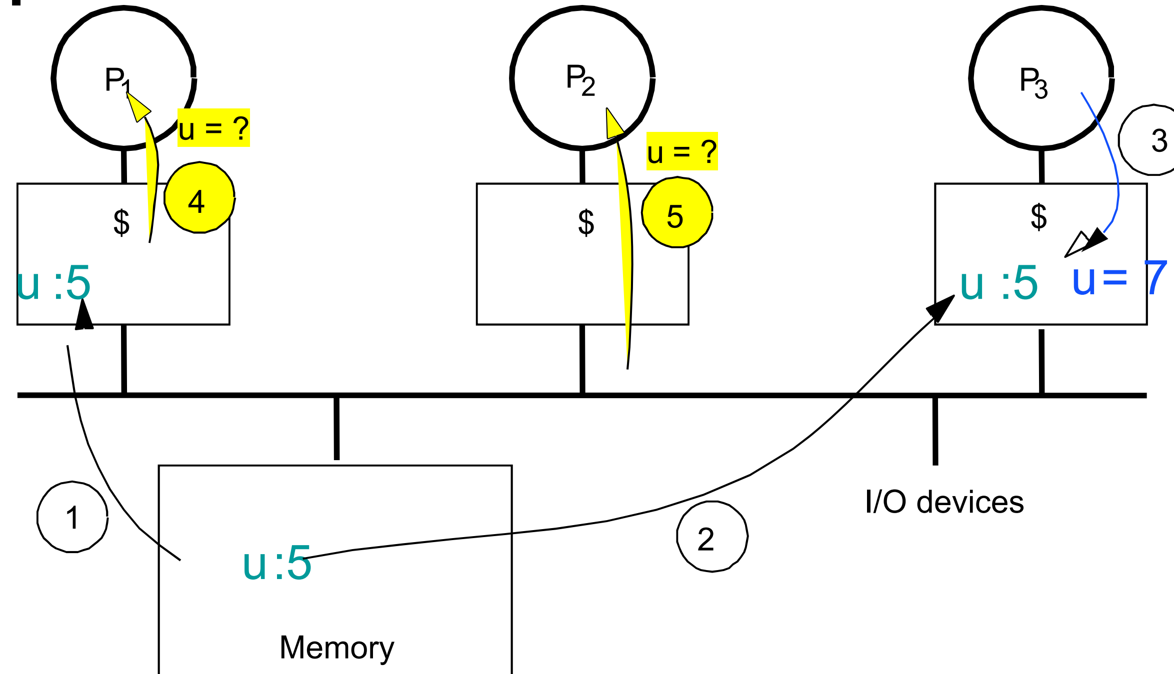
Example Cache Coherence Challenges



Example Cache Coherence Challenges

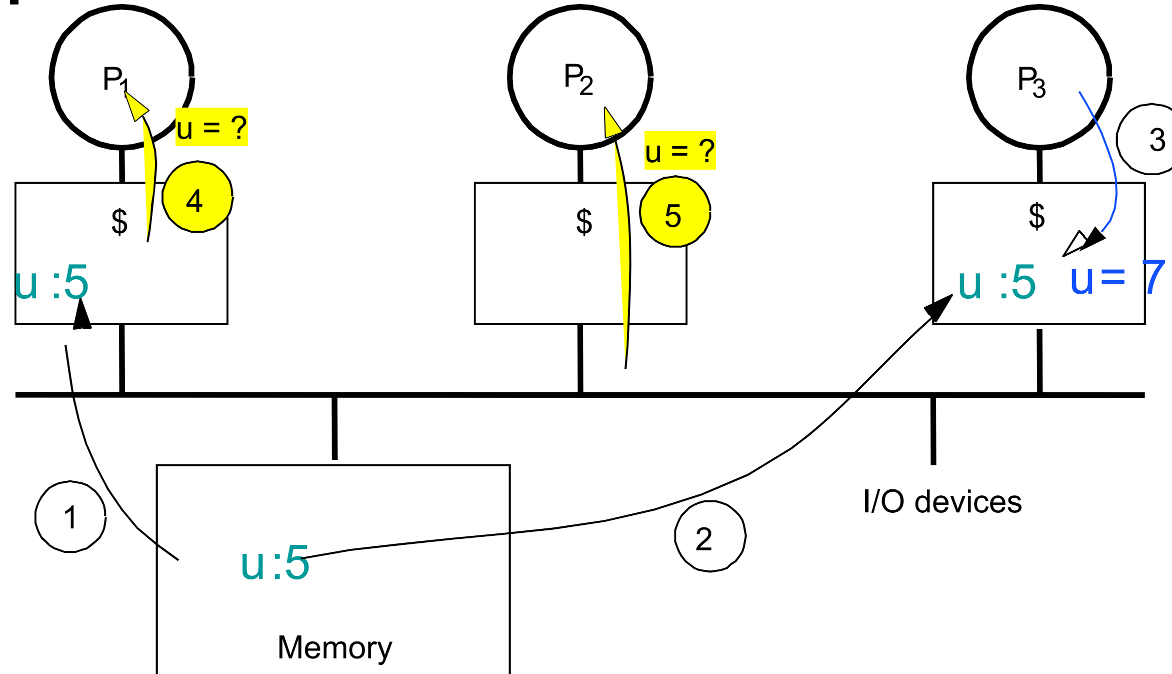


Example Cache Coherence Challenges



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on which cache flushes or writes back value when
  - Processes accessing main memory may see stale(old) value

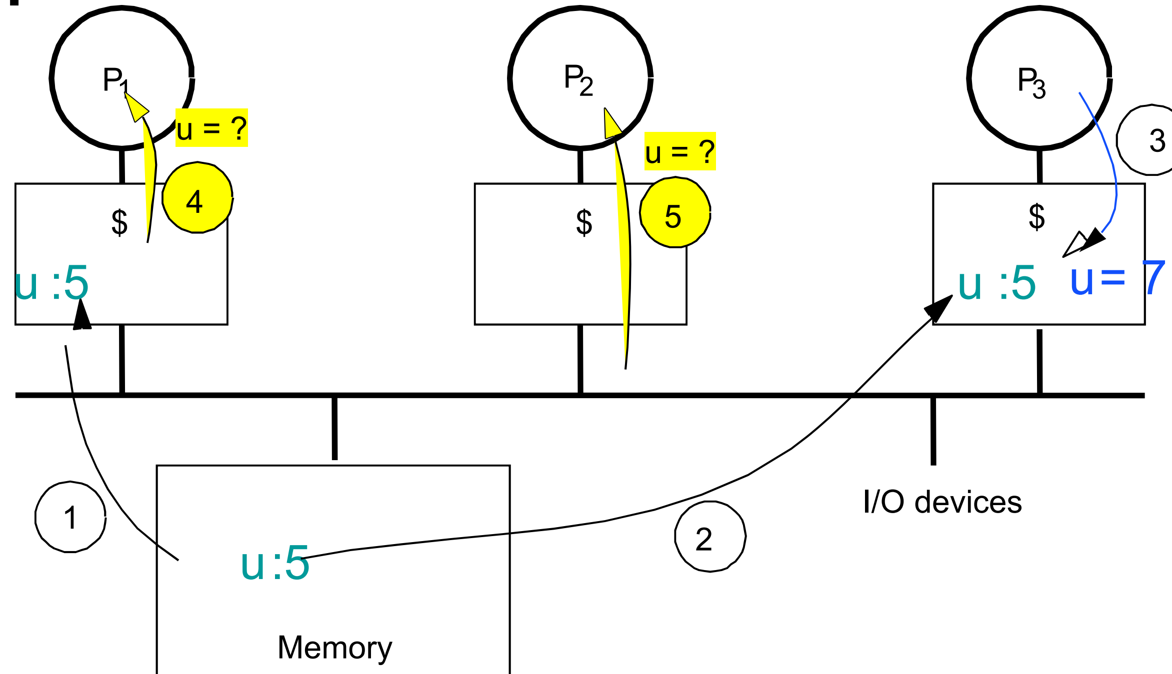
Example Cache Coherence Challenges



1. If P3 gets u=7 and then P1 or P2 reads u=5
2. If P2 gets value u=7 and then P1 gets later u=5

Coherent or non-coherent?

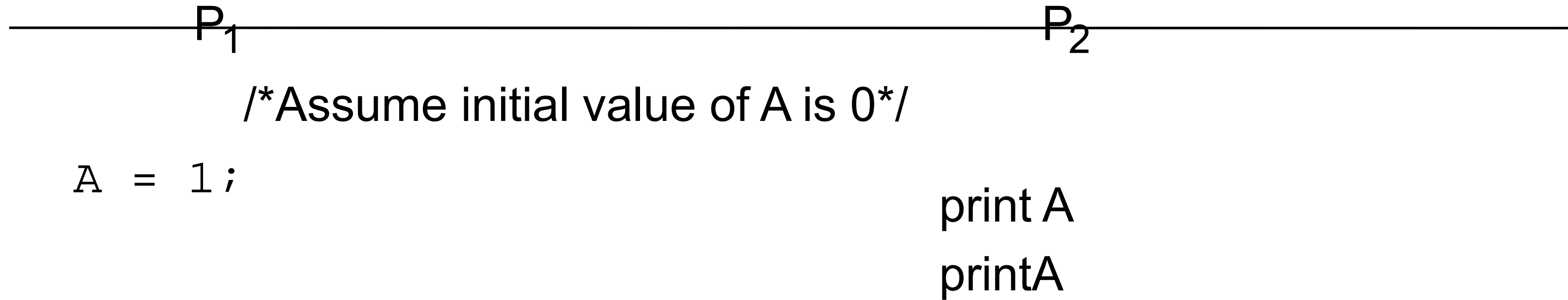
Example Cache Coherence Challenges



If P3 gets u=7 then P1 or P2 reads u=7

Coherent or non-coherent?

Example



- Coherent:

Cache Coherence

- Coherence
  - Requirement: writes to the same location by any two processors are seen in the same order by all processors
  - Order of stores observed in the same location
- HOW: on a write to a shared variable inform others variable copies (caches) learn that that their value is incoherent
  - How? before a write inform other copies about the write
  - How long it takes? It depends…

Example Illustrating Consistency Problem

P_1	P_2
/*Assume initial value of A and flag 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 processor
- Coherence is not enough!
  - pertains only to single location
- Memory Order seen by one processor different from another!

Consistency

- Requirement: if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- Order of stores observed in different locations
- When a written value will be returned by a read
- Simplest consistency: sequential (but too strict)

Basic Schemes for Enforcing Coherence

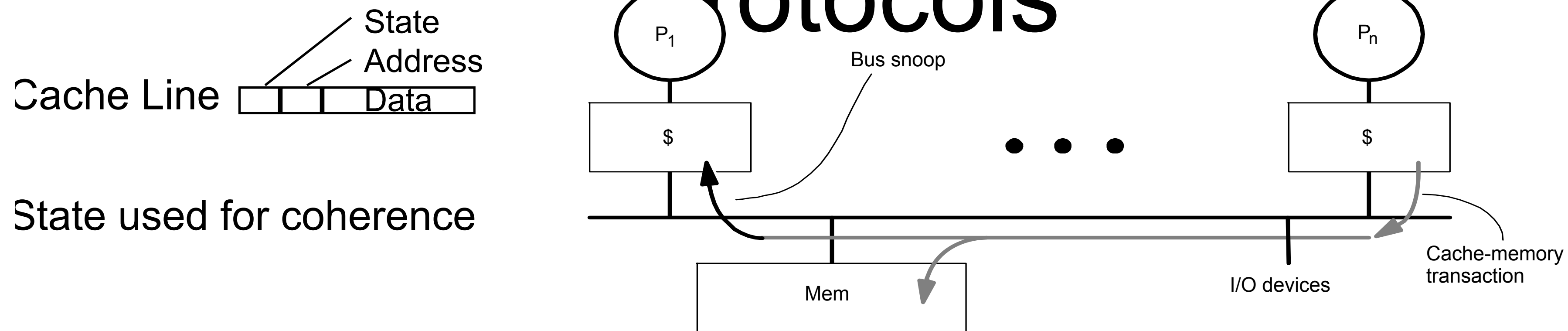
- Program on multiple processors will normally have copies of the same data in several caches
  - Reduces both latency of access and contention for read shared data and bandwidth demand on the shared memory

- Use a HW protocol to maintain coherent caches

2 Classes of Cache Coherence Protocols

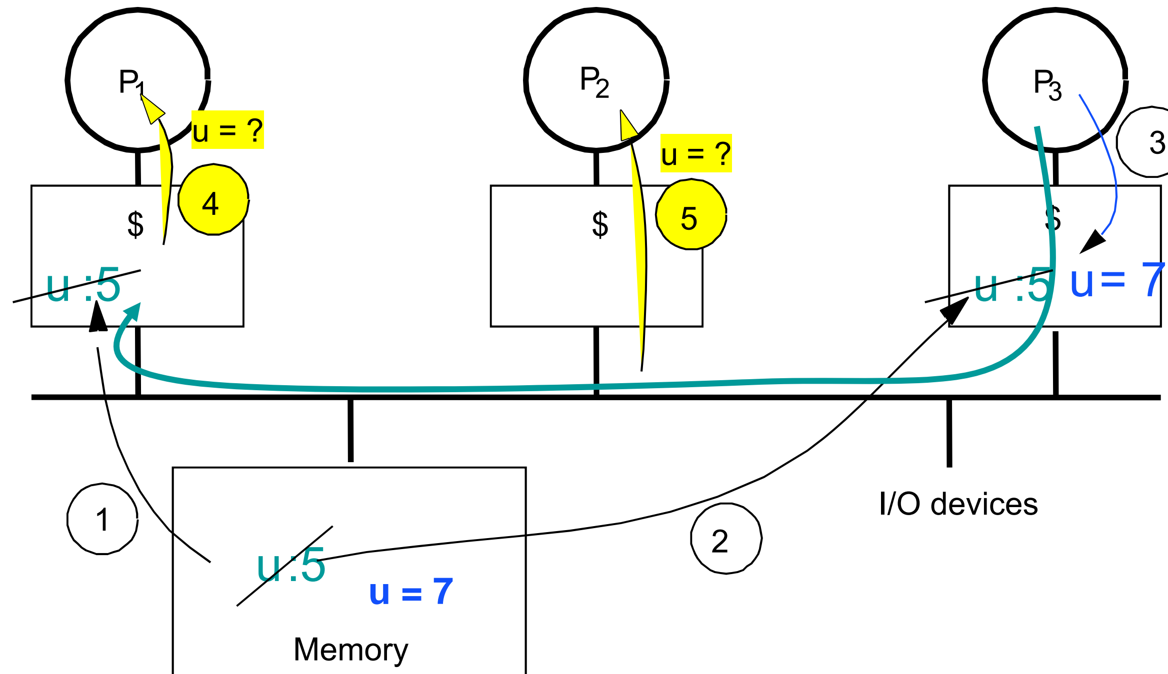
1. Snooping — Every core tracks sharing status of blocks
2. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory (directory can be distributed)

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or network)
  - Check addresses on bus
  - take action to ensure coherence if having a match
    - invalidate, update, or supply value
  - depends on state of the block and the protocol
- Writes: get exclusive access before write via write invalidate or update all copies on write

Example: Write-thru Invalidate

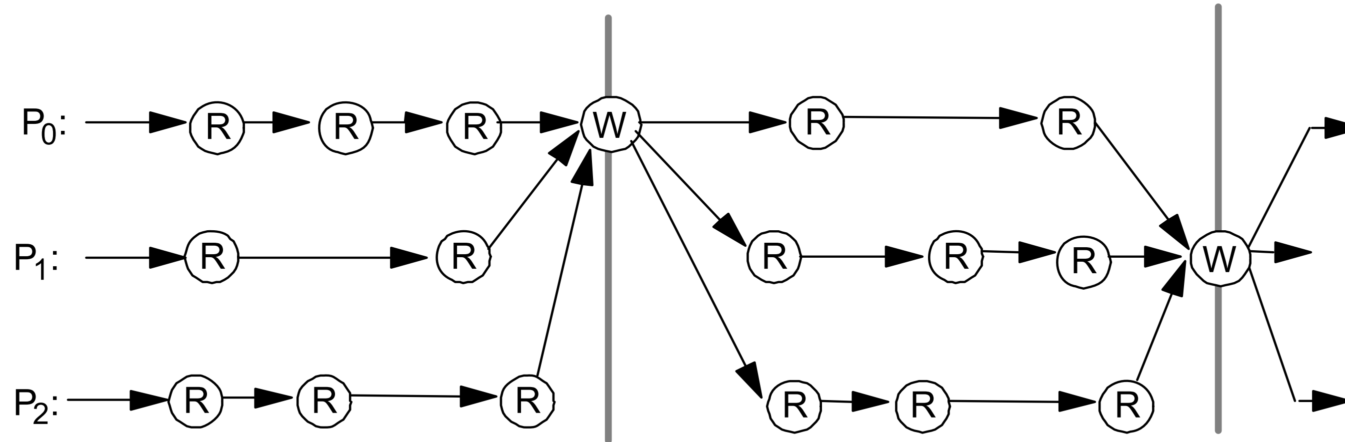


- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ typically MPs use write invalidate

Coherence Building Blocks

1. Cache block state transition diagram for each cache block
   1. FSM specifying how state of block changes
      - invalid, valid, dirty, shared etc
2. Broadcast Medium Transactions (e.g., bus)
   1. Logically single set of wires connect several devices
   2. Protocol: arbitration, command/addr, data
   3. SNOOPY: Every device observes every transaction
3. Broadcast medium helps enforces serialization of read or write accesses $\Rightarrow$ Write serialization
   1. Cannot complete write until it obtains bus
   2. 1st processor to get medium invalidates other copies
   3. All coherence schemes require serializing accesses to same cache block
4. Need to find up-to-date copy of cache block
   1. Cache coherence requests from other processors

Bus Orders Writes



- Writes establish a partial order
- Doesn’t constrain ordering of reads

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW

- Write-back harder
  - Most recent copy can be in a cache
  - Use snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory (chip2chip)

- Write-back consumers less memory bandwidth (why?)
  $\Rightarrow$ Support larger numbers of faster processors
  $\Rightarrow$ Most multiprocessors use write-back caches

Cache Resources for WB Snooping

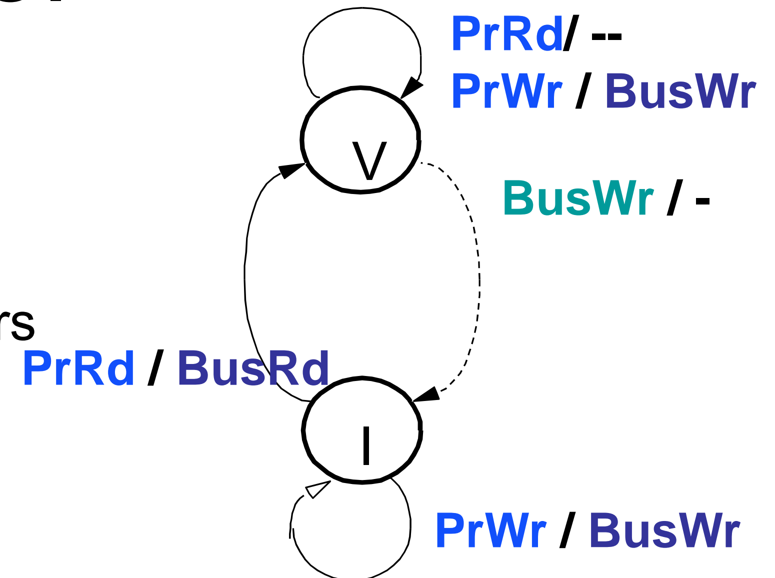
- Normal cache tags can be used for snooping
  - check address on bus if in the processor cache
- Valid bit per block used to invalidate
- Read misses handled by snooping
- Writes $\Rightarrow$ Need to know whether any other copies of the block are cached
  - Extra state per block indicates this
  - No other copies $\Rightarrow$ No need to place write on bus for WB (if a core owner can write it)
  - Other copies $\Rightarrow$ Need to place invalidate on bus (if not owner inform others)

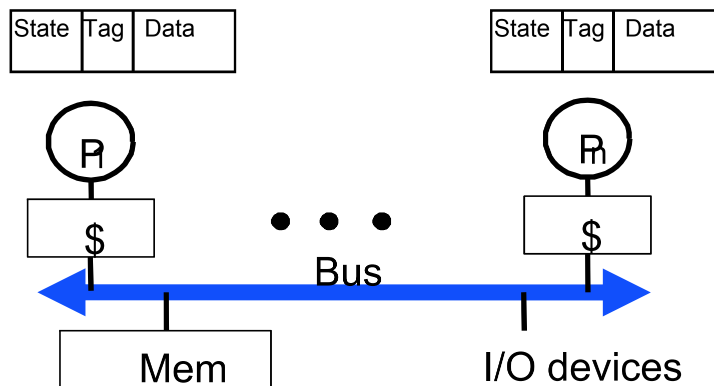
Example Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each core
- Each block separate state (not per word)
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Write-through Invalidate Protocol

- 2 states per block in each cache
  - as in uniprocessor
- Writes invalidate all other cache copies in other processors
  - can have multiple simultaneous readers of block, but write invalidates them





Processor Action/ Bus Action

PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write

What happens on a bus read?
What happens when block evicted?

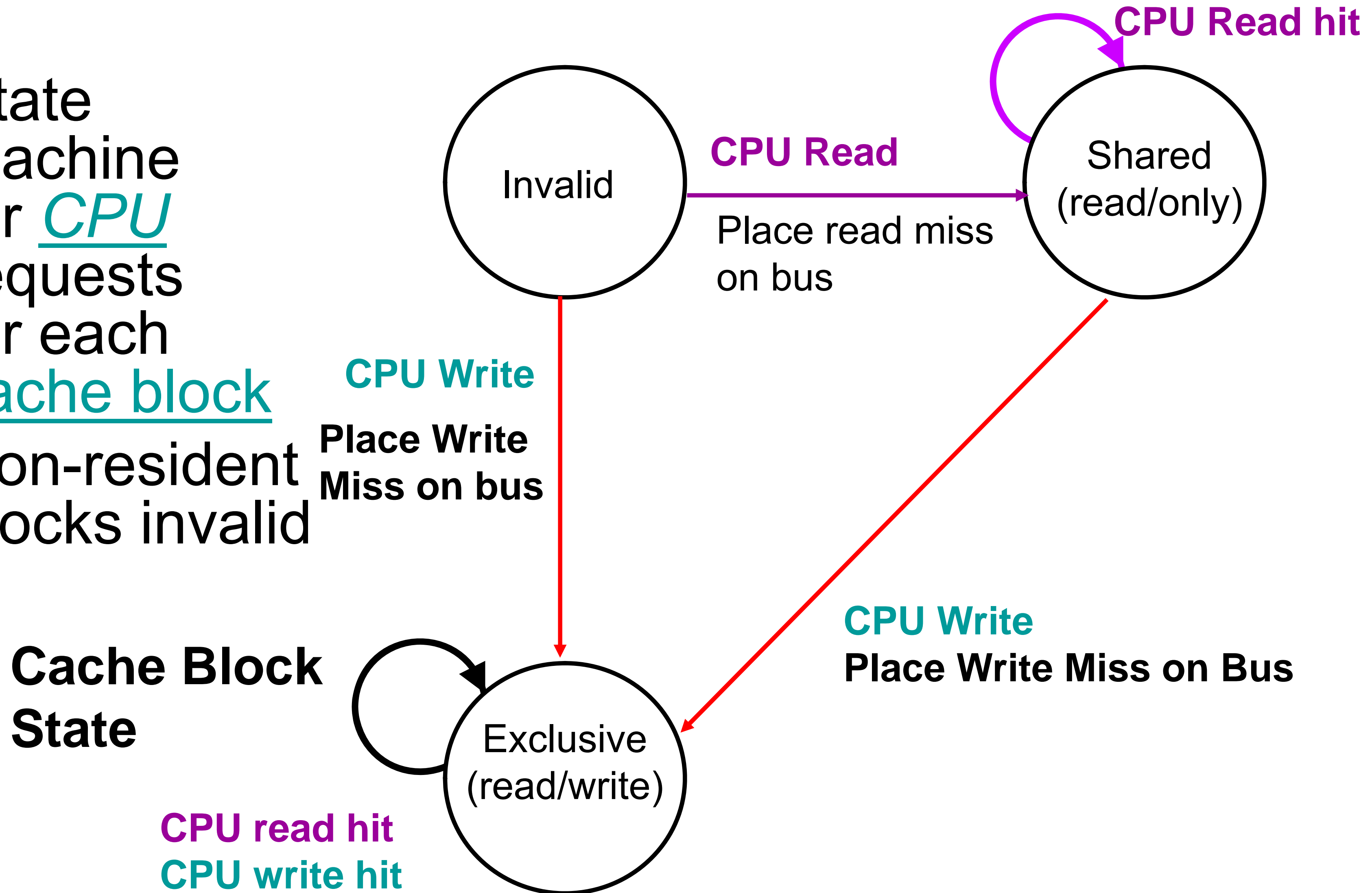
- No-write allocate

Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : one cache has copy, its writeable, and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)

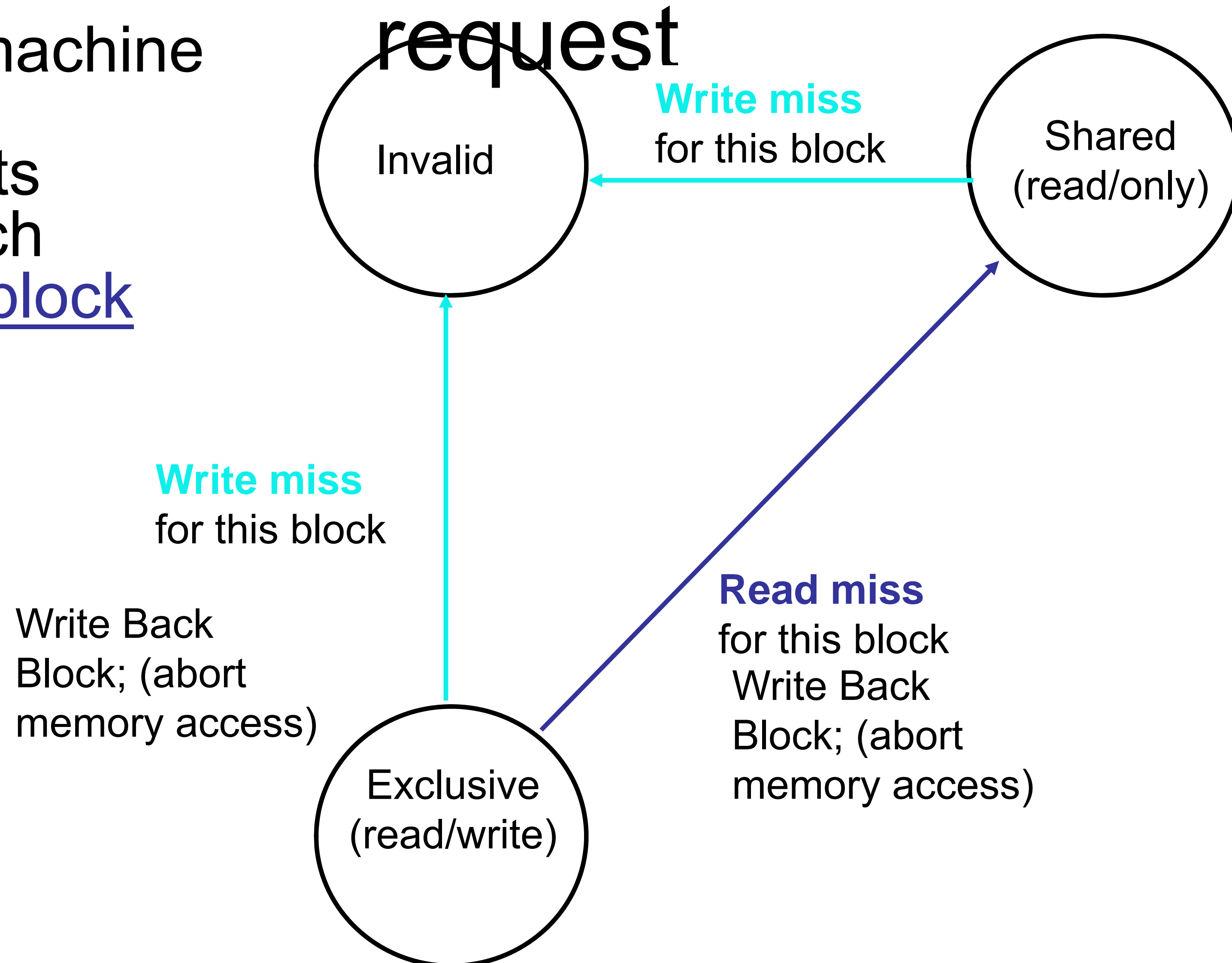
Write-Back State Machine - CPU

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



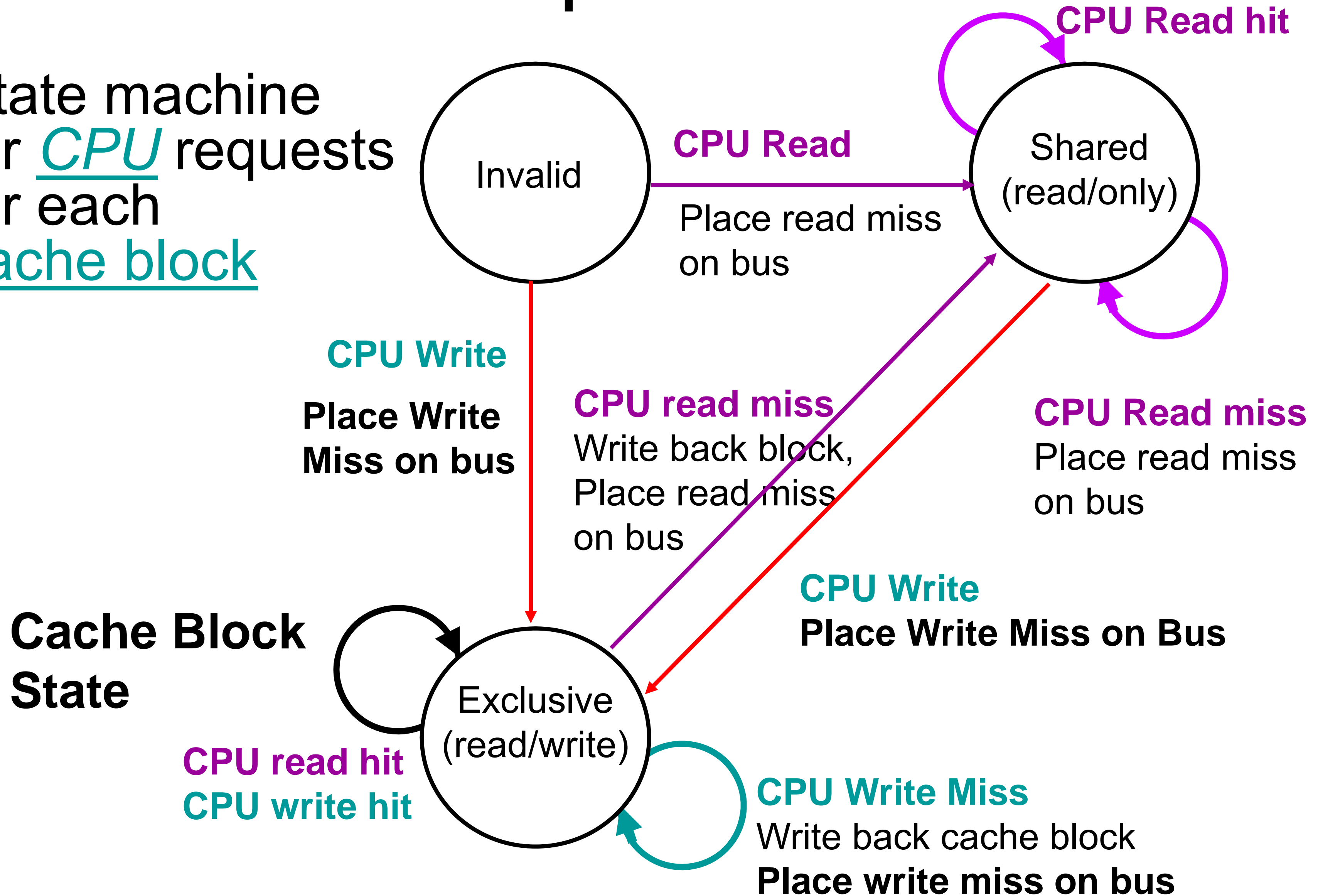
Write-Back State Machine- Bus request

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

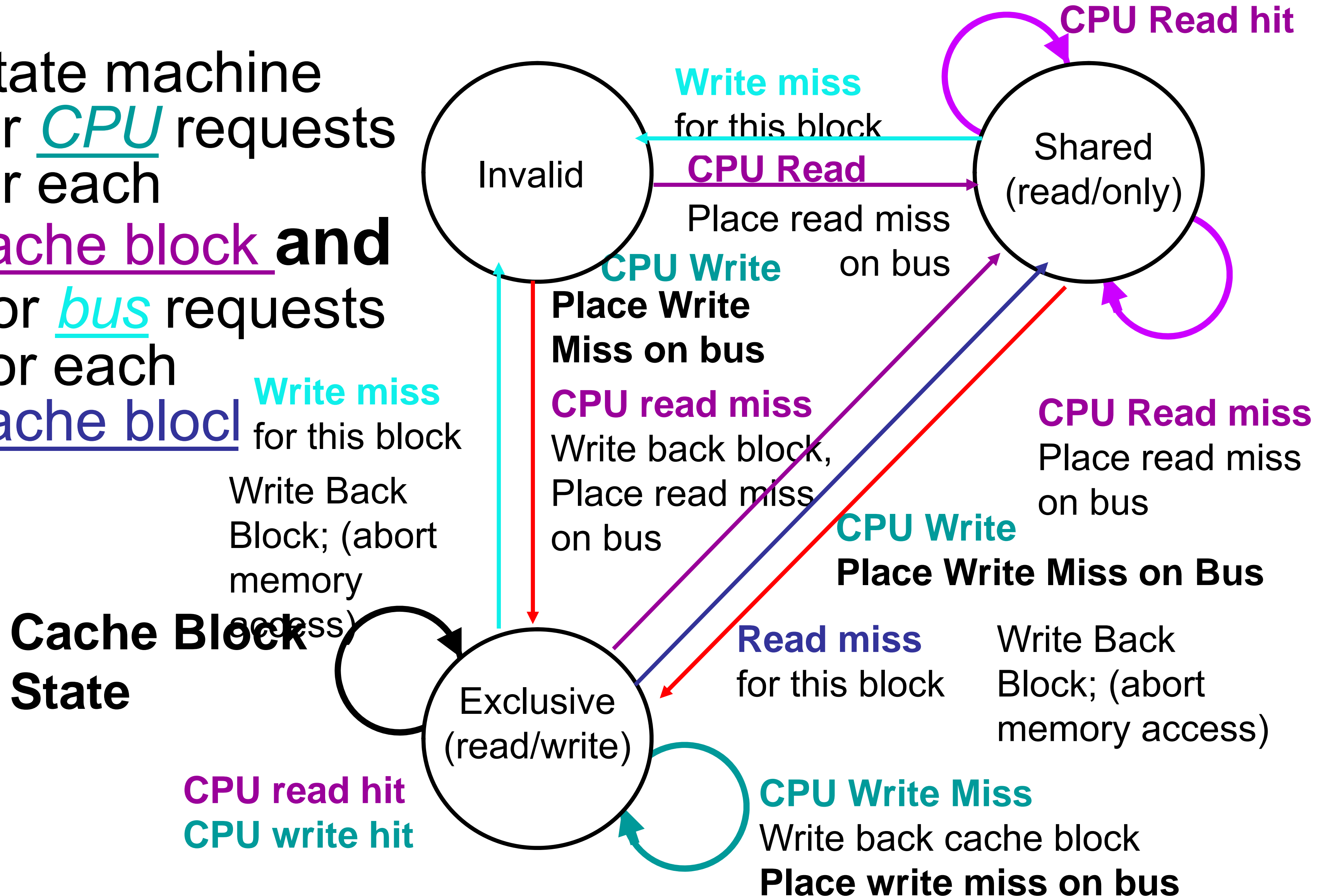


Block-replacement

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



Write-back State Machine-III



Example

	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												

Example

	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												

Example

	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												

Example

	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												

Example

	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												

Example

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Valu*	*Action*	*Proc.*	*Addr*	*Value*	*Add*	*Valu*
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*

- Assume block A2 evicts A1

Cache behavior in response to bus

- For every bus transaction each core must check the cache-address tags
  - could potentially interfere with core cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1
  - $\Rightarrow$ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Summary

- Parallelism challenges: % parallalizable, long latency to remote memory
- Centralized vs. distributed memory
  - Small MP vs. lower latency, larger BW for Larger MP
- Shared Address vs Message Passing
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
  - Uniform access time vs. Non-uniform access time
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes

Write Serialization/Consistency

- A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
- For now assume the processor does not change the order of any write with respect to any other memory access

$\Rightarrow$ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order