CMPE110 Lecture 15
Cache Coherency

Heiner Litz

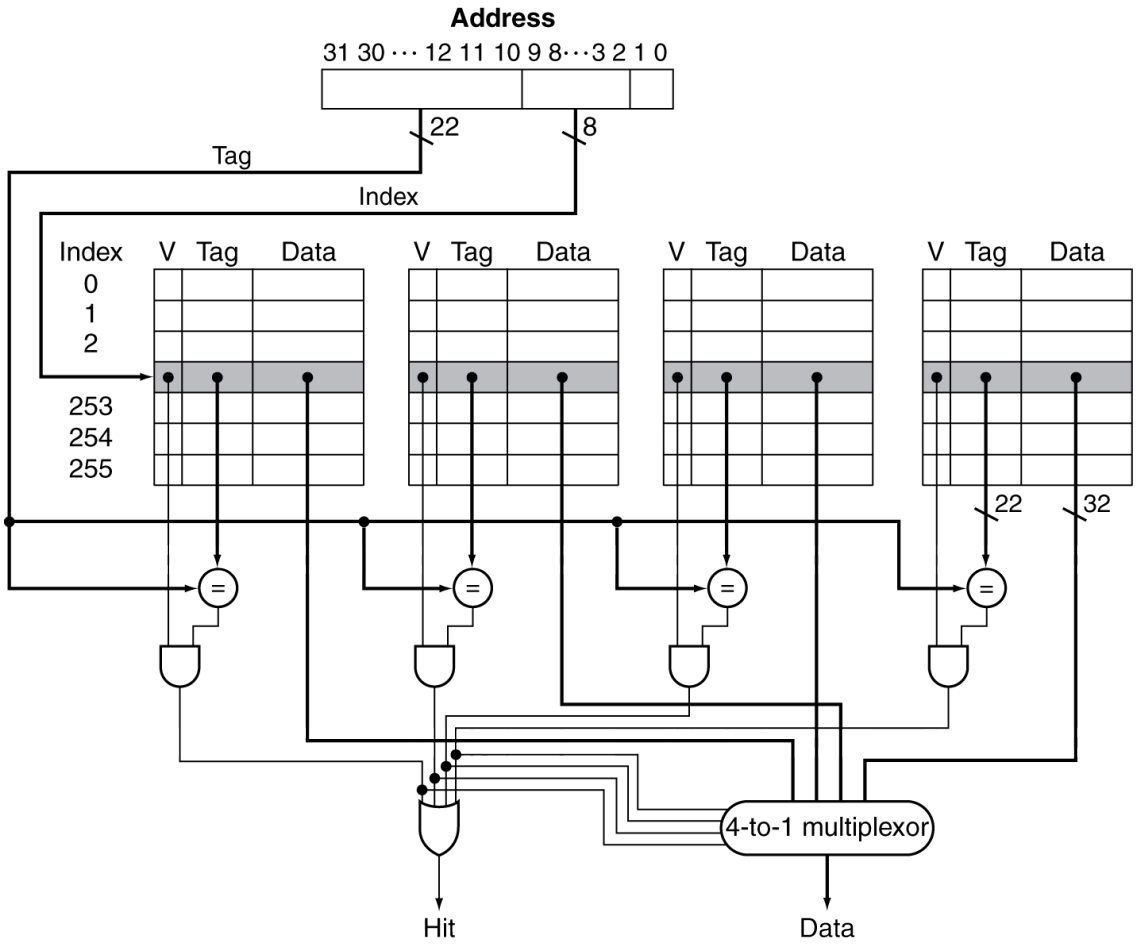
https://canvas.ucsc.edu/courses/12652

Announcements

- Midterm has been moved to Monday May 14th
- Review session on Friday May 11th
- HA4 has been released
- Readings: 5.1-5.4, 5.9-5.10

Review

Set Associative Cache Design



Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative	8	2	tag: 5 | index: 3 | blk off: 4
4-way set associative	4	4	tag: 6 | ind: 2 | blk off: 4
8-way set associative	2	8	tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative	1	16	tag: 8 | blk off: 4

Write Miss Actions

Only works for **DM cache**

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

Quiz

- How can we patch our pipeline to have a single unified DRAM for data and instructions but two ports for accessing instructions & data?

Splitting Caches

- Most chips have separate caches for instructions & data
  - Often noted ad $I and $D or I-cache and D-cache
- Advantages
  - Extra access port, bandwidth
  - Low hit time
  - Customize to specific patterns (e.g. line size)
- Disadvantages
  - Capacity utilization
  - Miss rate

Multilevel Caches

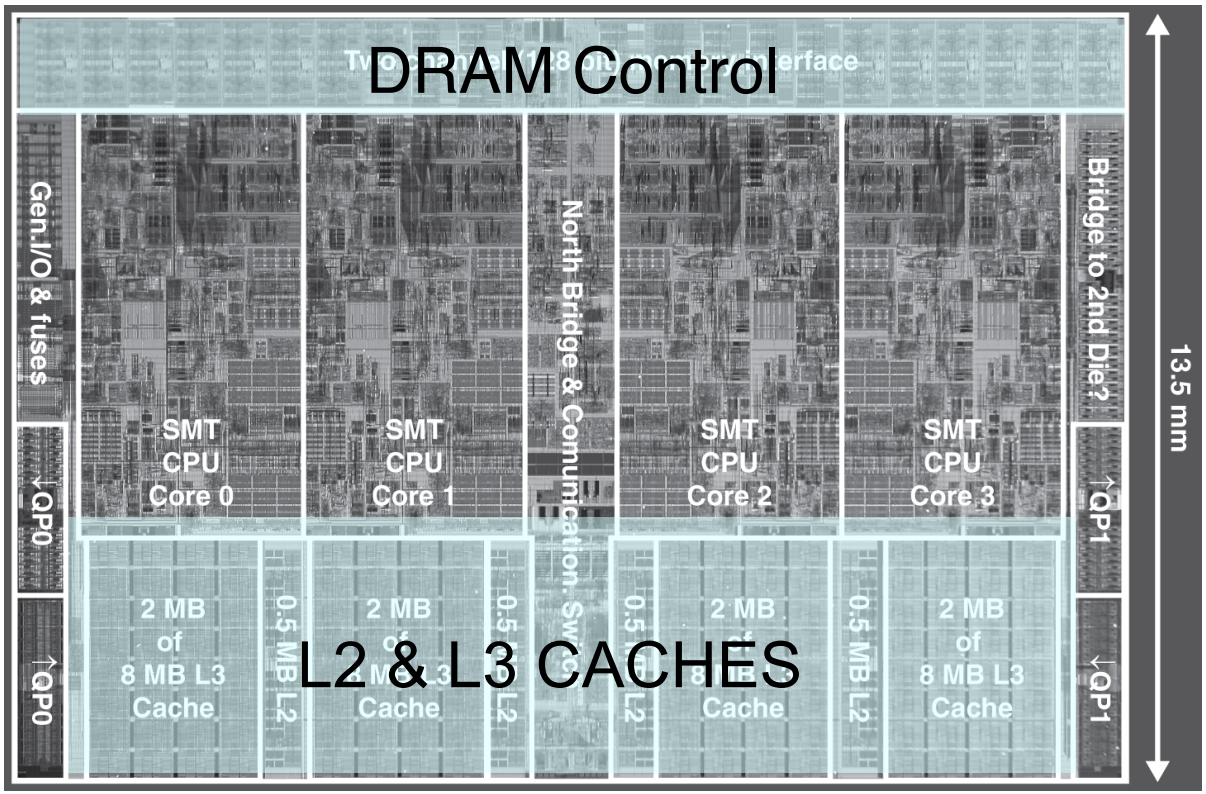
- Primary (L1) caches attached to CPU
  - Small, but fast
  - Focusing on hit time rather than miss rate
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Unified instruction and data (why?)
  - Focusing on low miss rate rather than low hit time (why?)
- Main memory services L2 cache misses
  - Many chips include L3 cache

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Multilevel On-Chip Caches

Intel Nehalem 4-core processor

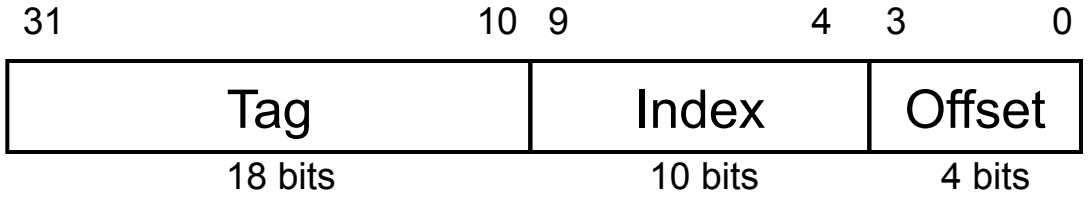


Per core:
-32KB, 4-way L1 $I
-32KB, 8-way L1 $D
-256KB, 8-way L2

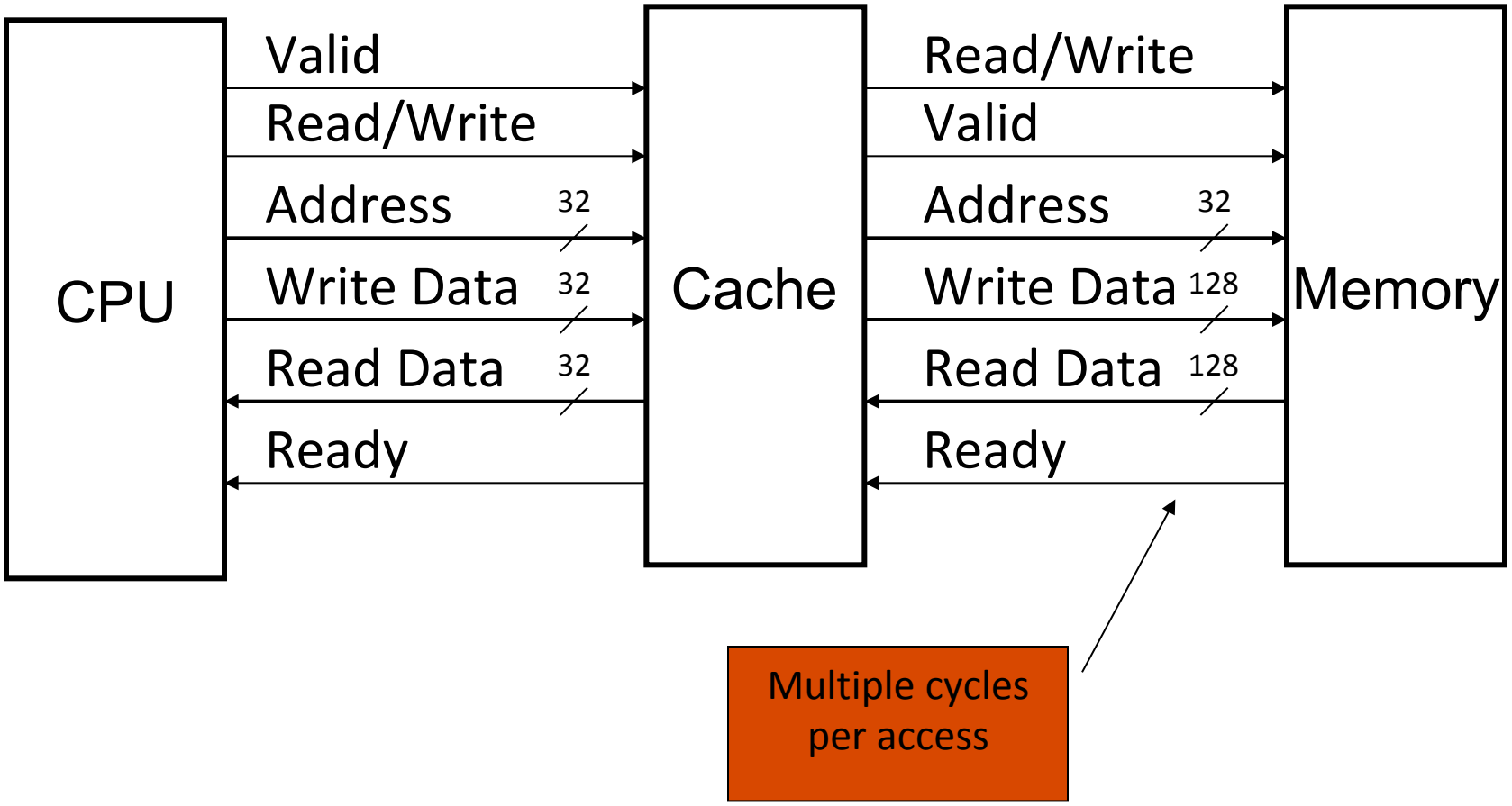
Shared
- 8 MB, 16-way L3

Cache Implementation: Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

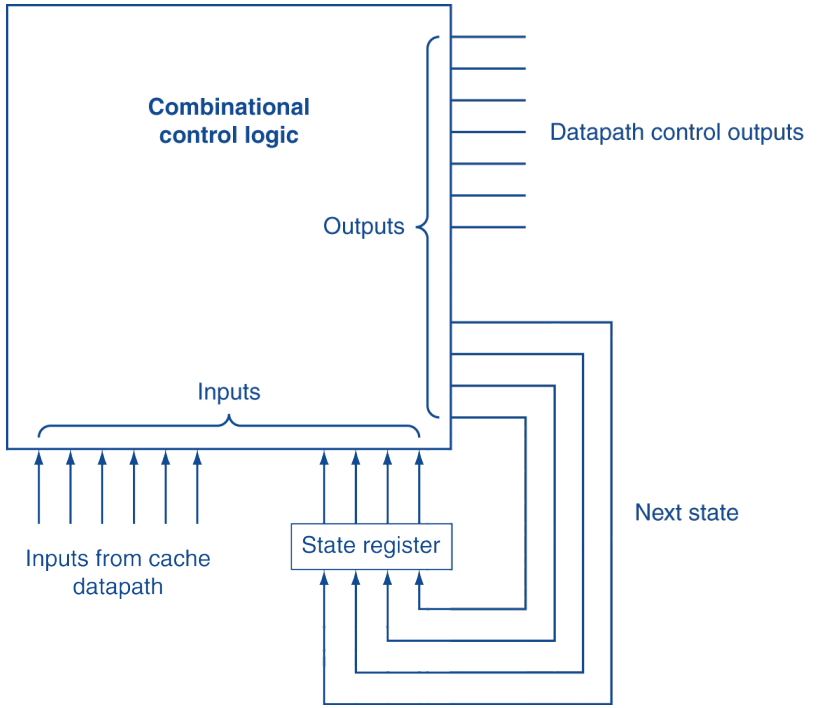


Interface Signals



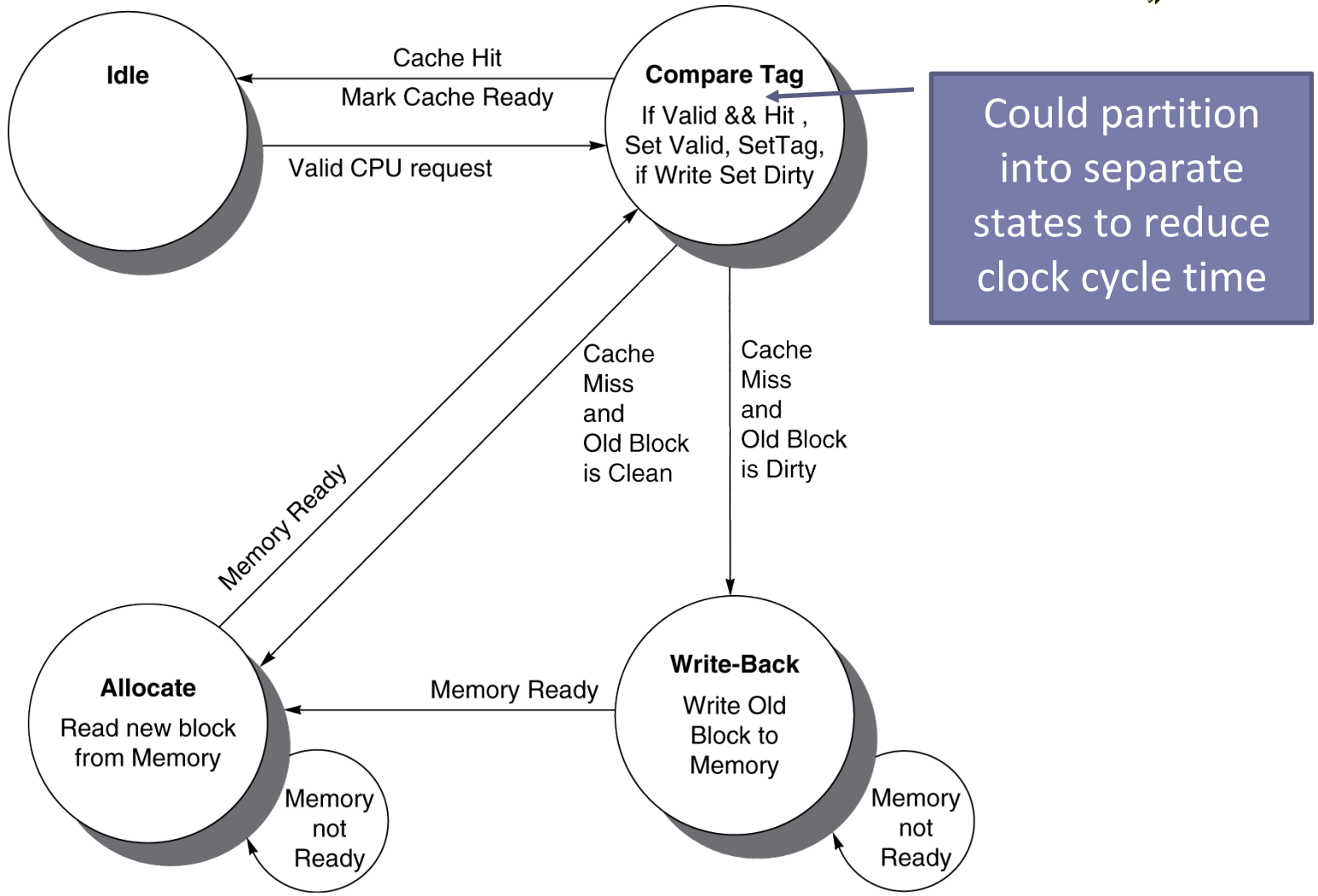
Reminder: Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state
    = fn (current state, current inputs)



- Control output signals = fn (current state)

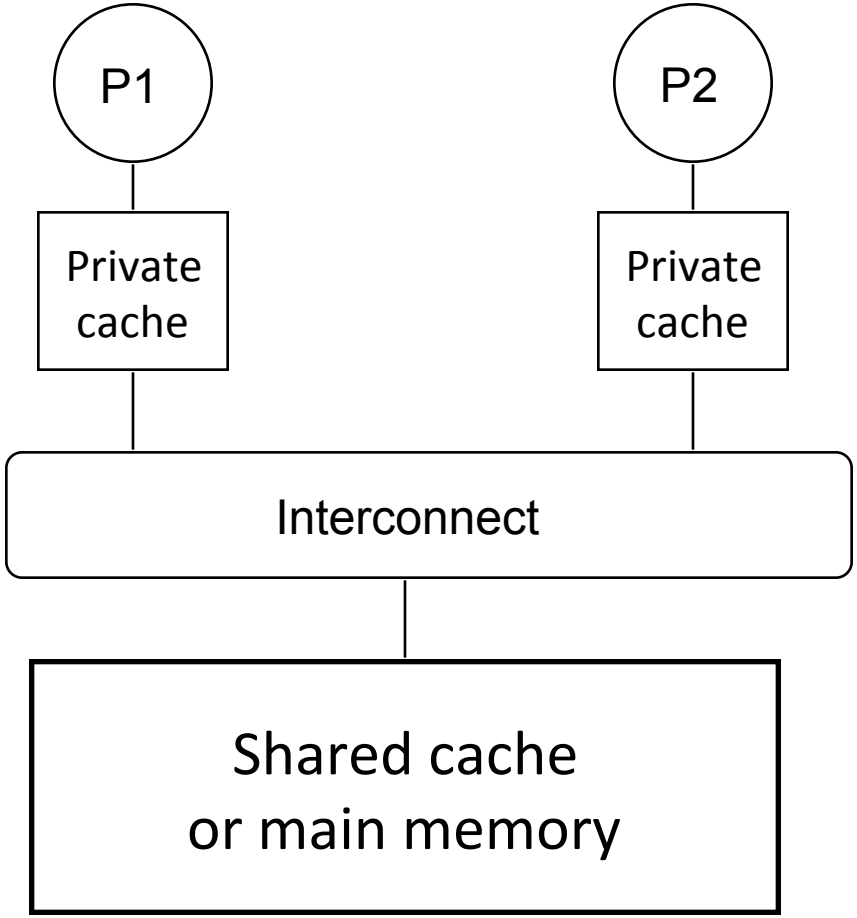
Cache Controller FSM - WRITE



Multi-core & Caches

- See any issues?

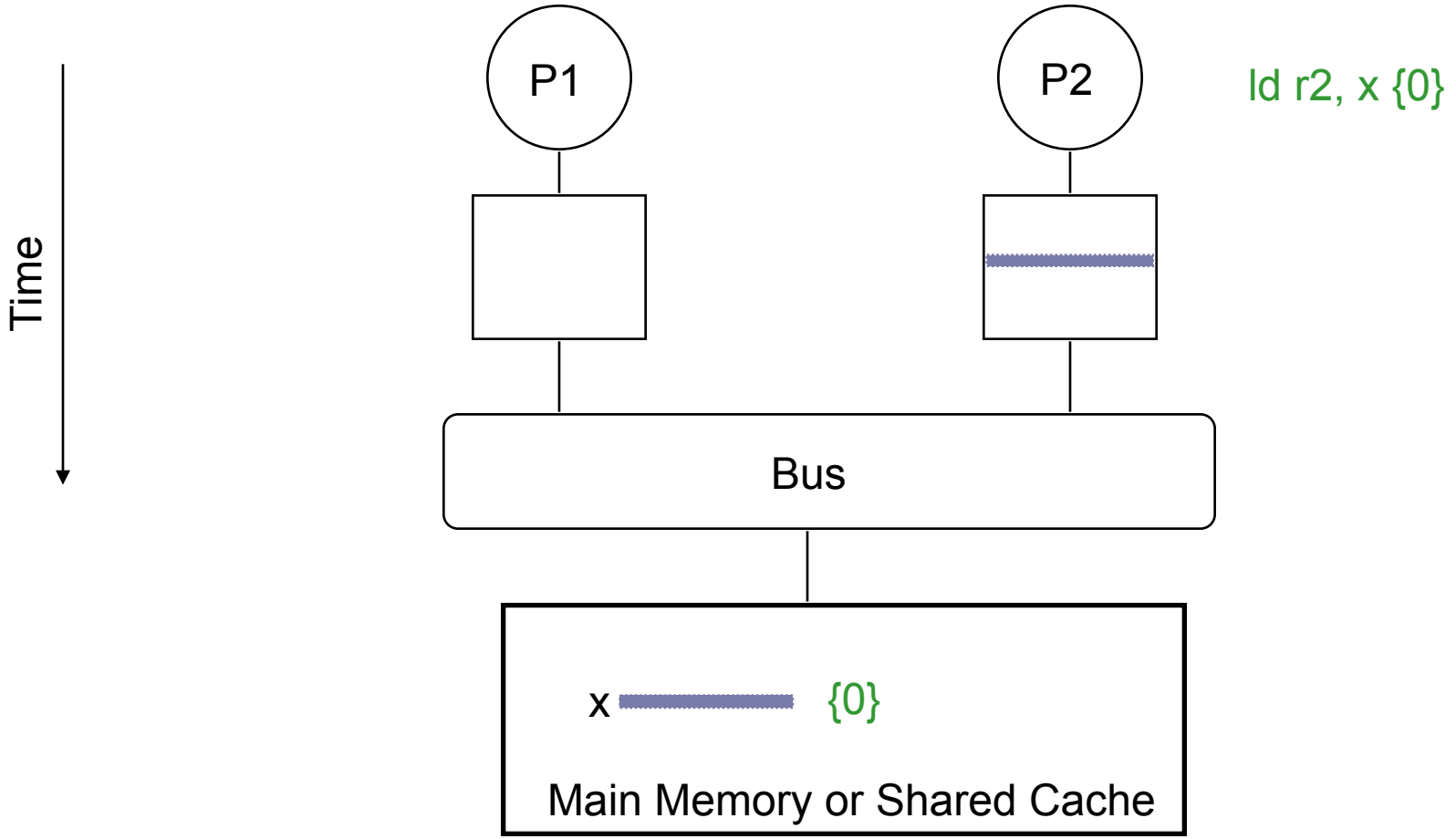
Cache Coherence Problem



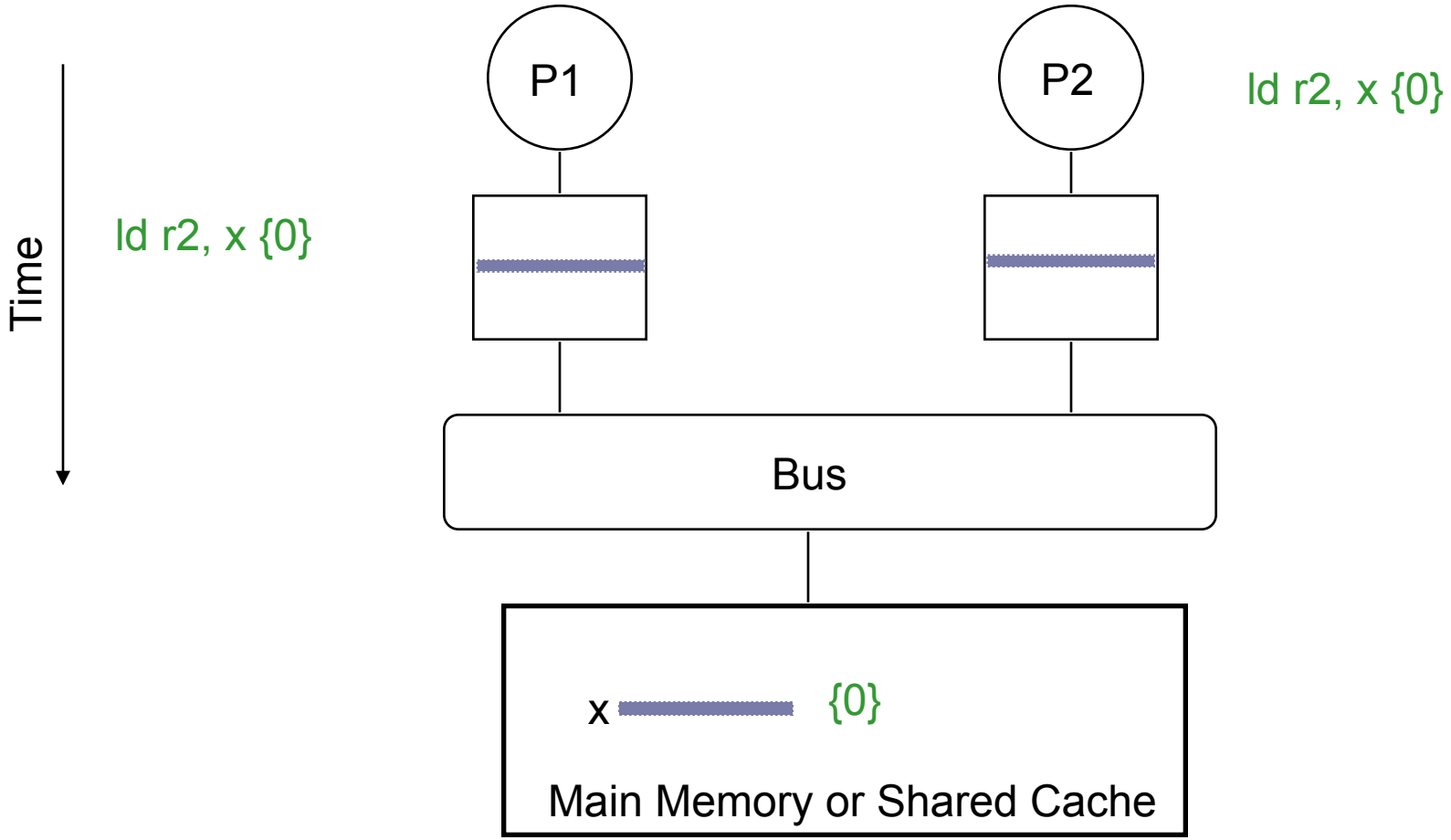
Cache Coherence

- Informally
  - Reads return most recently written value
- Formally
  1. P writes X; P reads X (no intervening writes)
     ⇒ read returns written value
  2. P1 writes X; P2 reads X (sufficiently later)
     ⇒ read returns written value
  3. P1 writes X, P2 writes X
     ⇒ all processors see writes in the same order
     ⇒ End up with the same final value for X

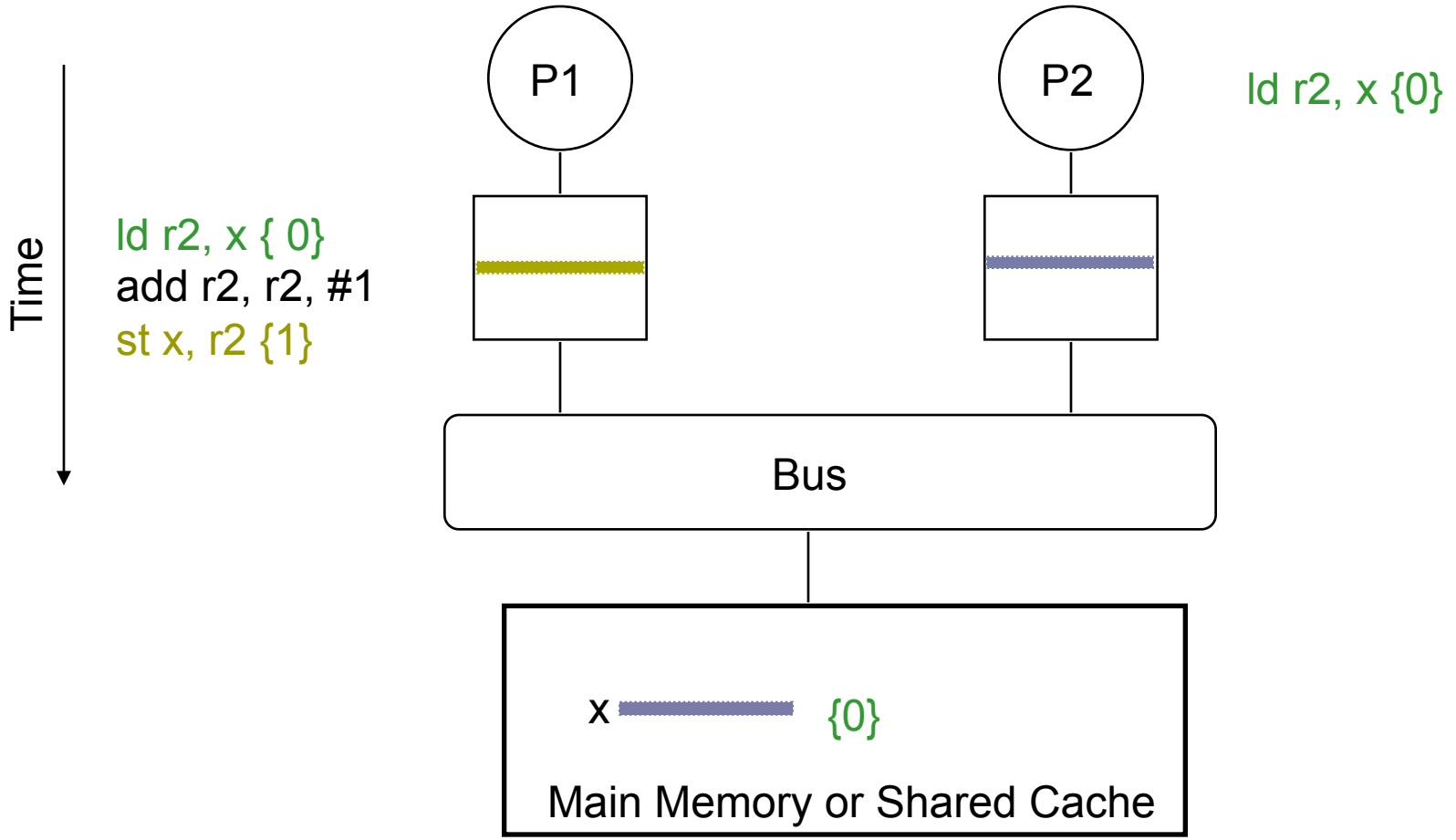
Cache Coherence Problem (Step 1)



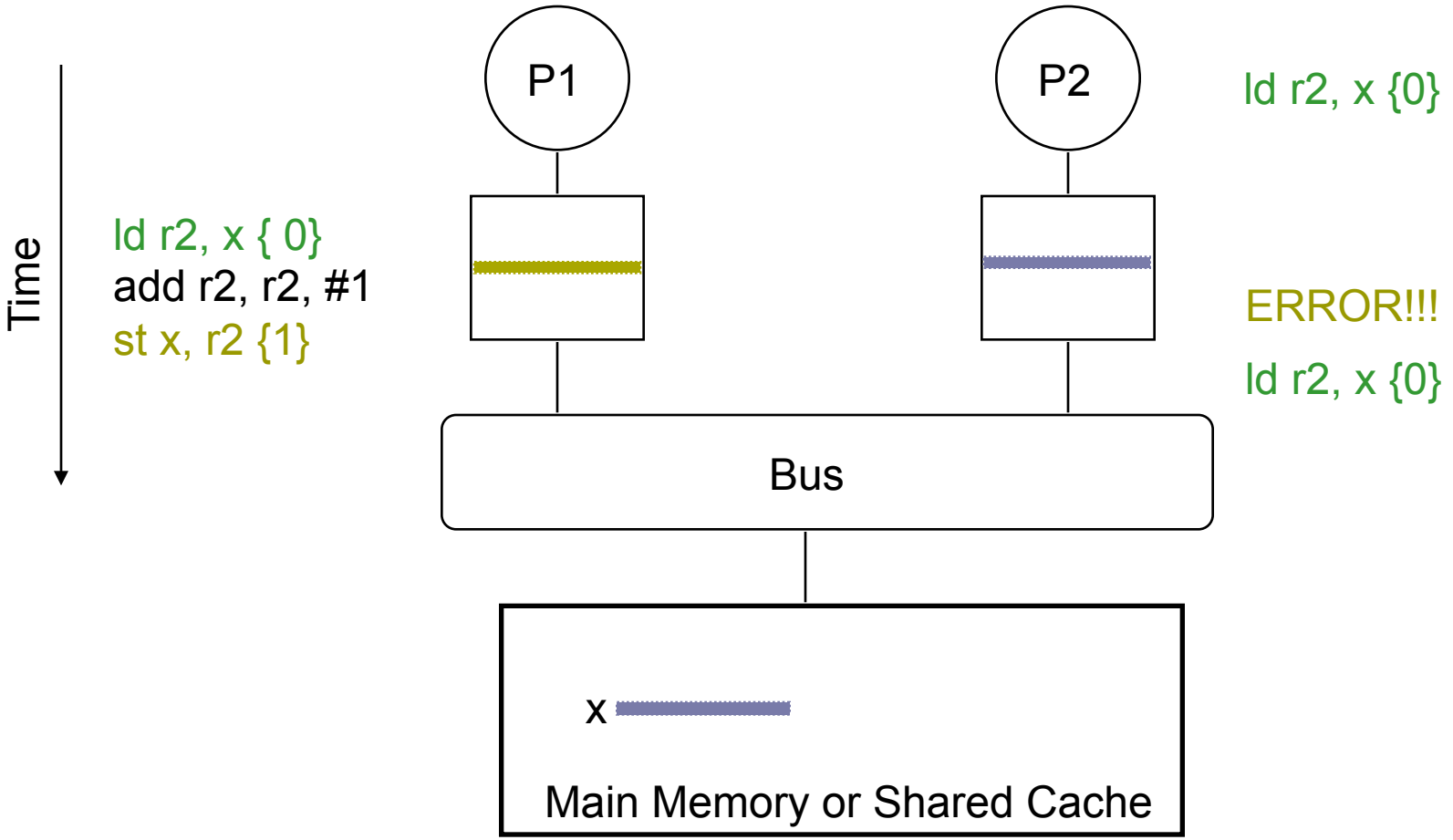
Cache Coherence Problem (Step 2)



Cache Coherence Problem (Step 3)



Cache Coherence Problem (Step 4)



Cache Coherence Protocols

- Cache coherence protocols
  - State and sequence of actions needed to ensure caches are coherence in multi-core systems
- Naïve protocol: turn off caching
  - Does not allow caching of read-shared data
  - Does not allow caching of private data
- Smarter protocol: allow one writer at the time
  - All caches can have copies of data while read-shared
  - On a write, invalidate all copies, allow a single writer

Cache Coherence Protocol

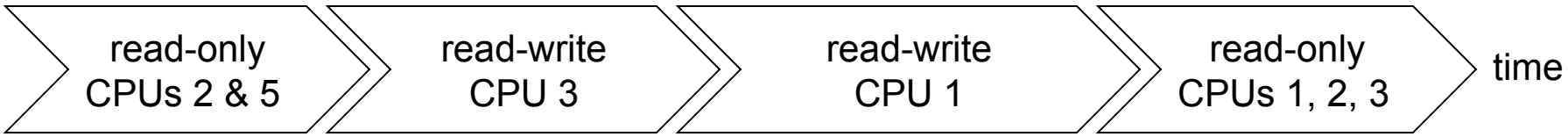
Divide lifetime of a memory value into epochs (time periods)

1. Single-Writer, Multiple-Read (SWMR) Invariant

   For any memory location A, at any given time (epoch), there exists only a single CPU that may write to A (and can also read it) or some number of CPUs that may only read A

2. Data-Value Invariant

   The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch

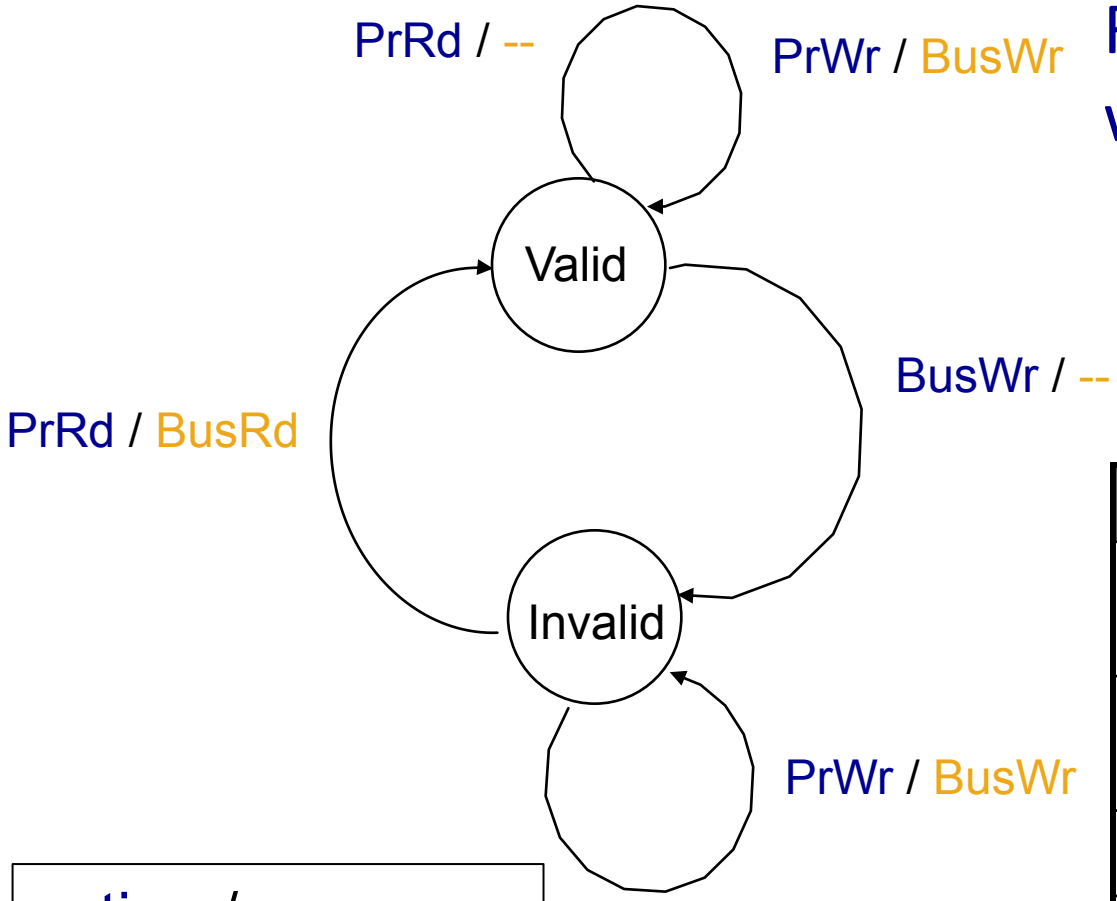


Implementation of Coherence Protocols

- Protocols rely on monitoring core actions
  - For now, assume all misses visible to all cores
    - Interconnect makes all actions visible
    - Cache FSM also reacts to requests from other caches
  - Known as a **snooping-based** protocols

- Protocols avoid stale copies
  - Whenever a core becomes a writer: invalidate copies

Simple Coherence Protocol



For write-through, no-write-allocate caches

Action	Abbreviation
Processor Read	PrRd
Processor Write	PrWr
Bus Read	BusRd
Bus Write	BusWr

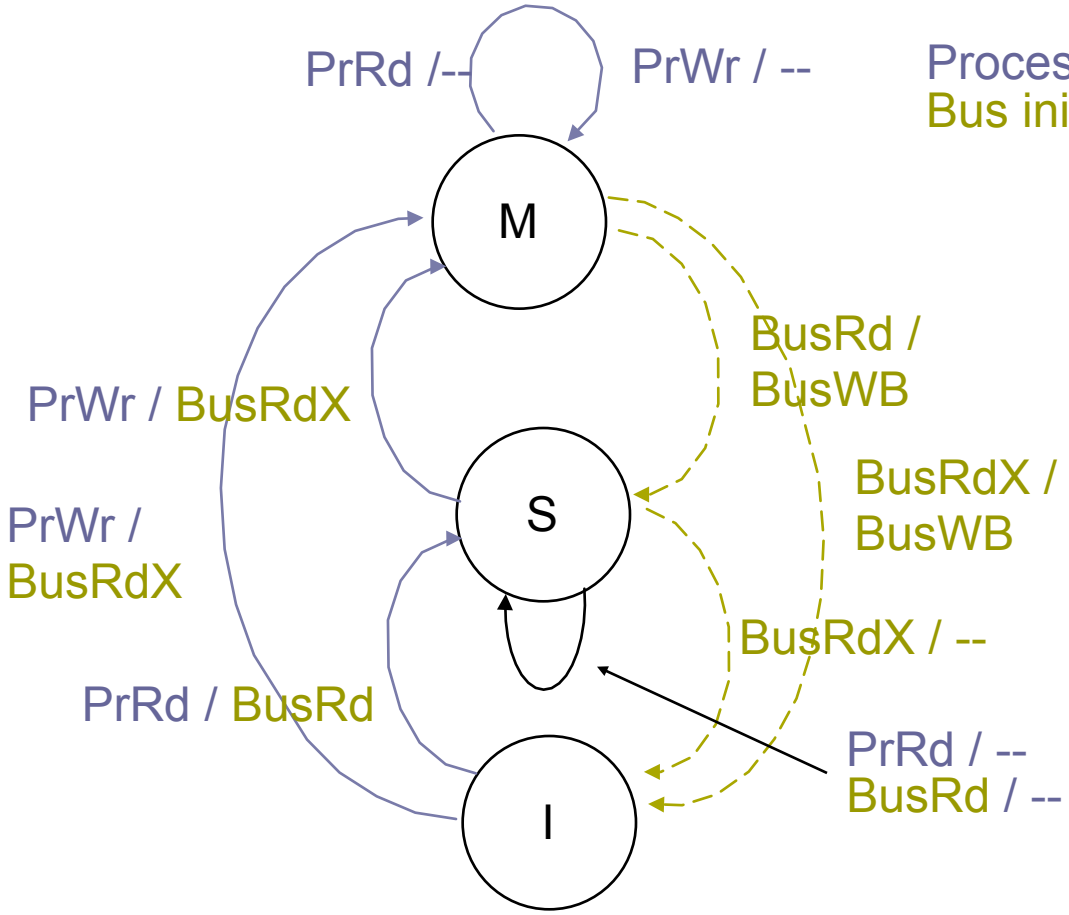
Is 2-state Protocol Coherent?

- Assume bus transactions and memory operations are atomic
  - Processor waits for memory operation to finish before issuing next
  - With one-level cache, assume invalidations applied during bus xaction
- SWMR Invariant
  - All writes go to bus + atomicity, causes transition to invalid state
- Data-Value Invariant
  - Read misses appear on bus, and will “see” last write in bus order
  - Read hits: do not appear on bus
  - But value read was placed in cache by either
    - Most recent write by this processor or most recent read miss
    - Both these transactions appeared on the bus
  - So reads hits also see values as produced bus order

A 3-State Write-Back Invalidation Protocol

- 2-State protocol
  - + Simple hardware and protocol (used in Sun Niagara)
  - - Bandwidth: write-through, every write goes on bus
    - Can work in multi-core with additional level of shared cache
- 3-State protocol (MSI)
  - Modified (called Exclusive in some books)
    - One cache has valid/latest copy
    - Memory is stale
  - Shared
    - One or more caches (and memory) have valid copy
  - Invalid
- Must invalidate all other copies before entering modified state
  - Requires bus transaction (order and invalidate)

MSI Coherence Protocol



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback