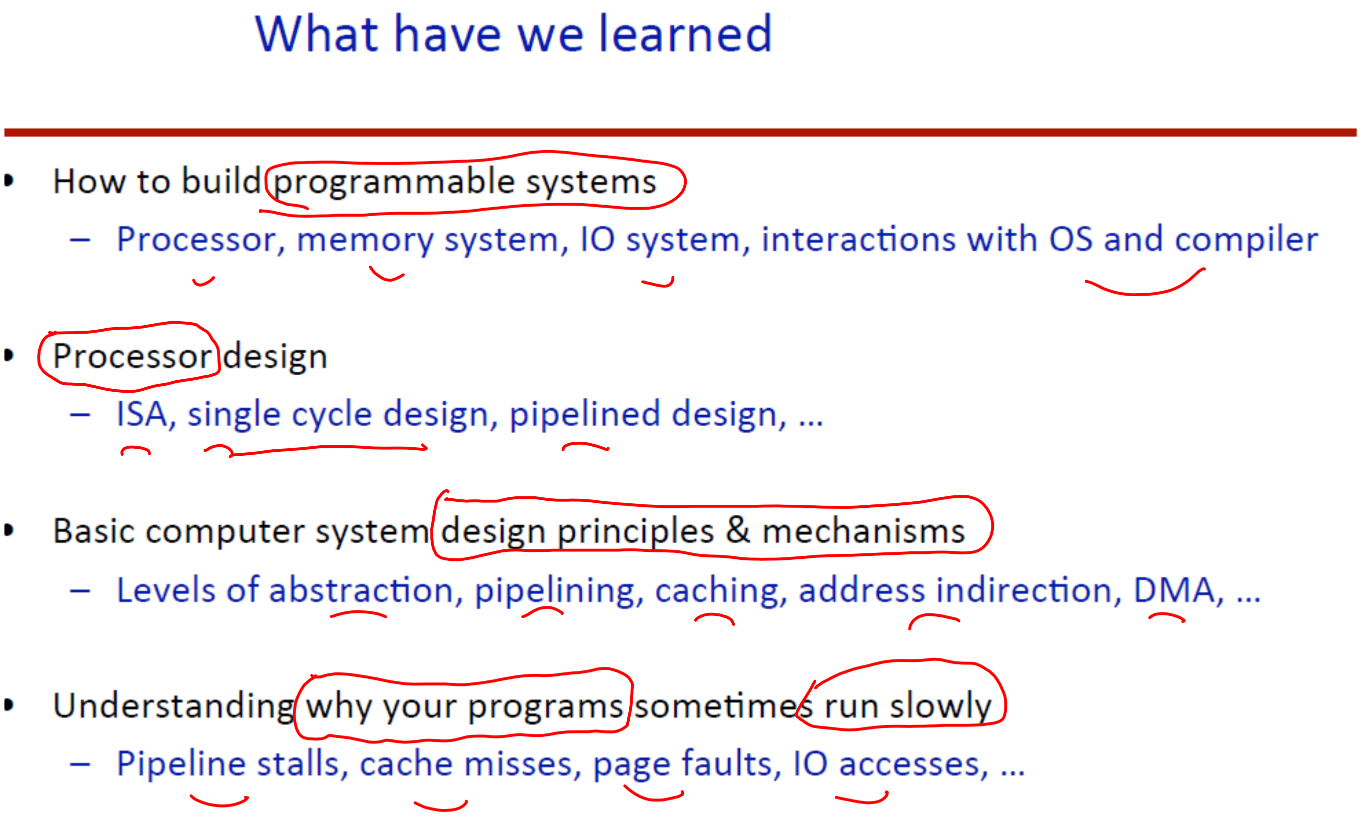
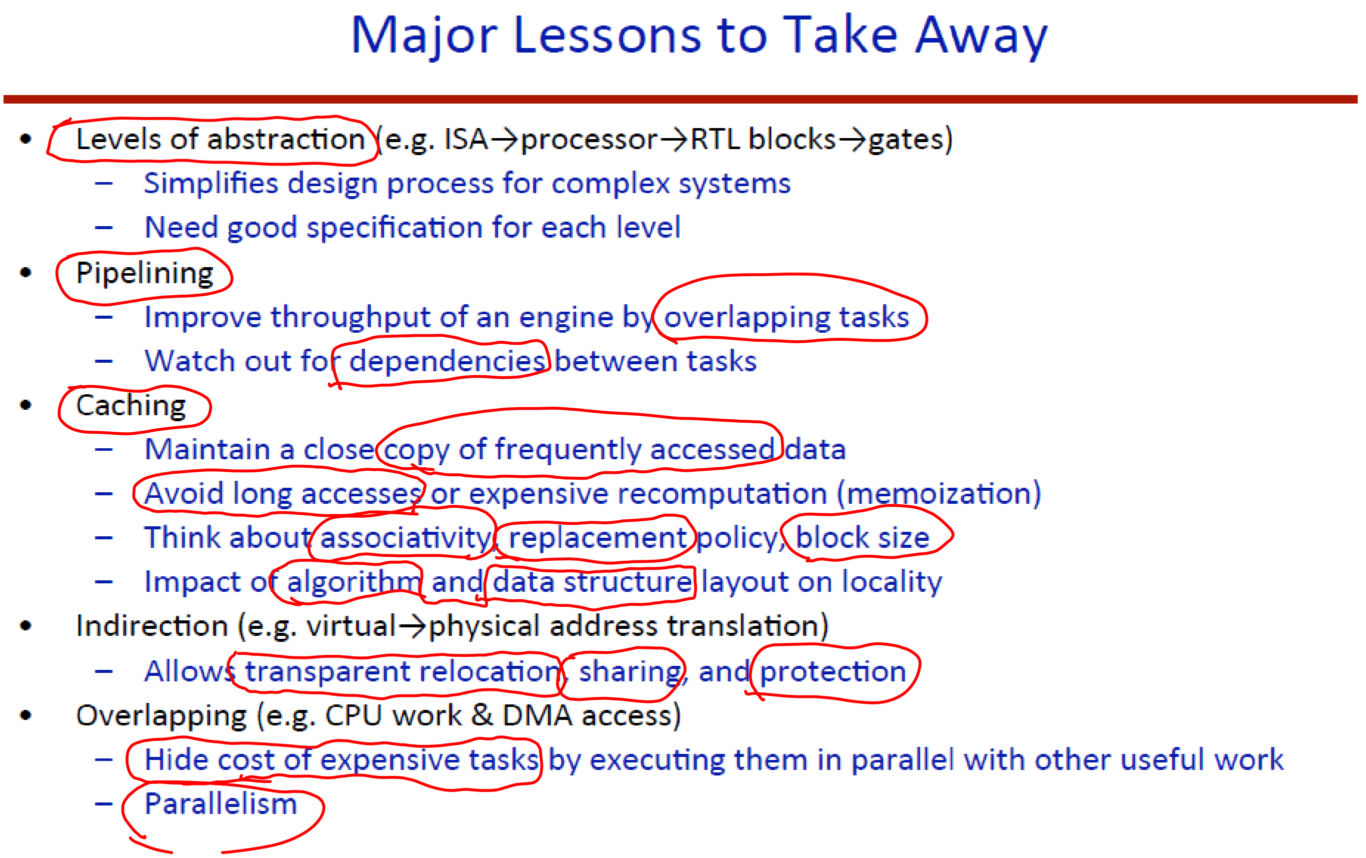
Multi



What have we learned

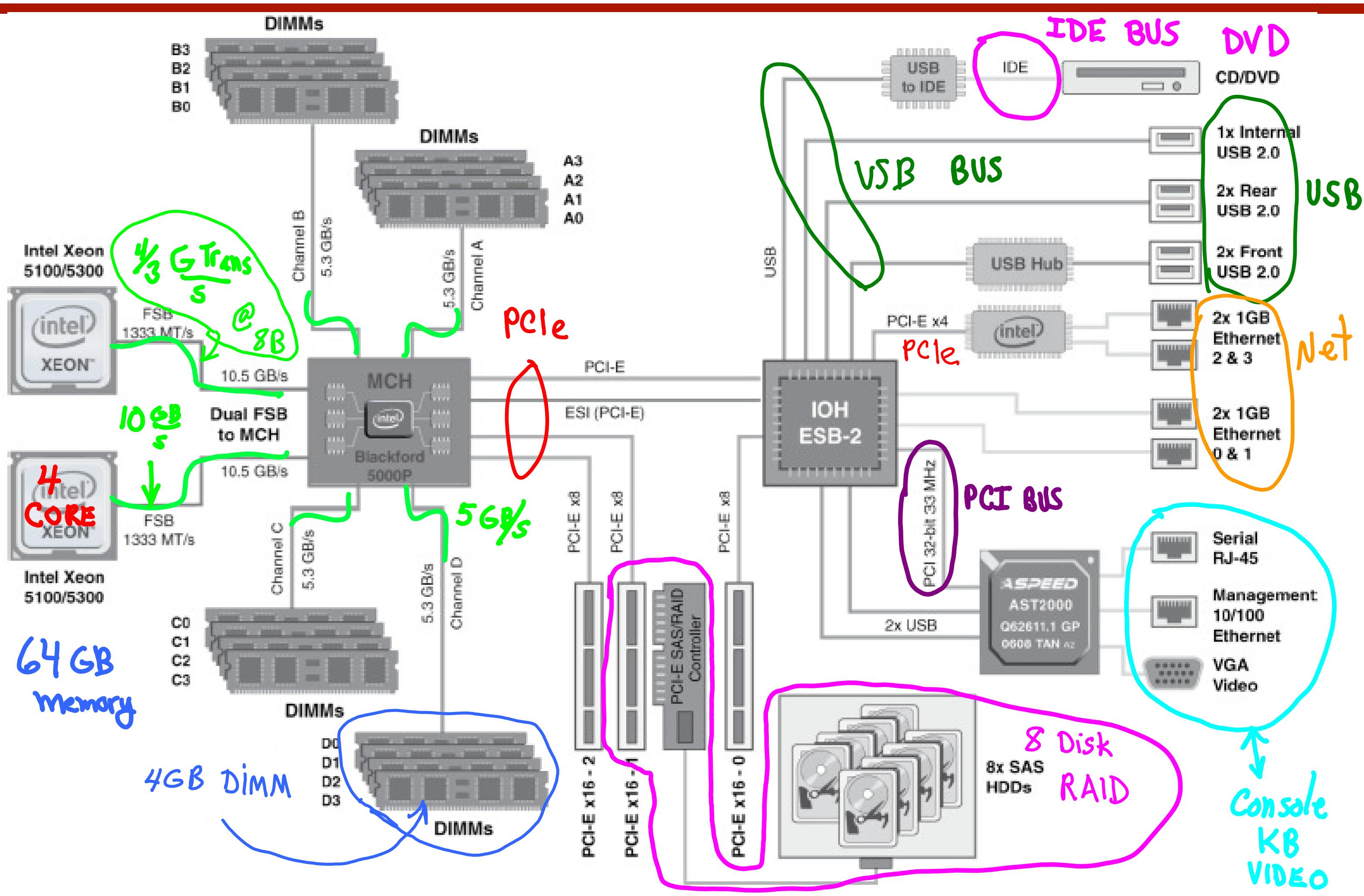
- How to build programmable systems
  - Processor, memory system, IO system, interactions with OS and compiler
- Processor design
  - ISA, single cycle design, pipelined design, …
- Basic computer system design principles & mechanisms
  - Levels of abstraction, pipelining, caching, address indirection, DMA, …
- Understanding why your programs sometimes run slowly
  - Pipeline stalls, cache misses, page faults, IO accesses, …

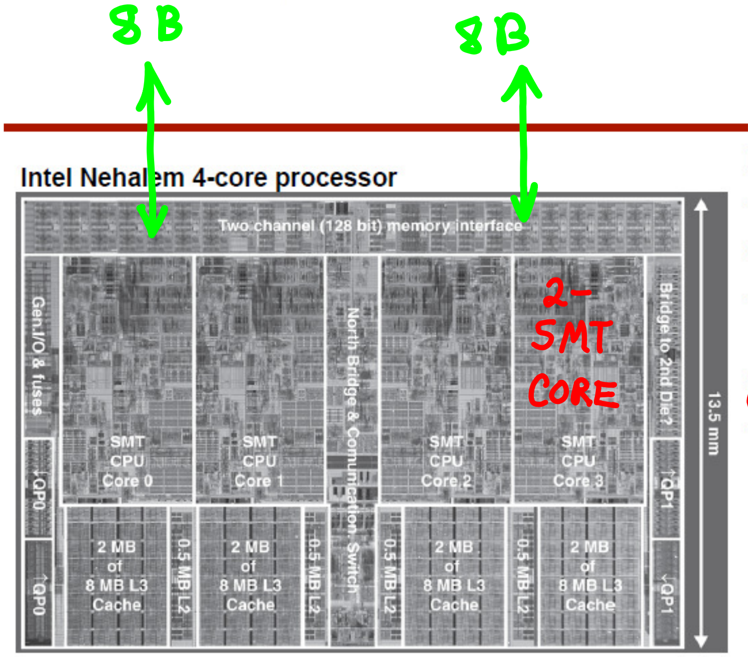


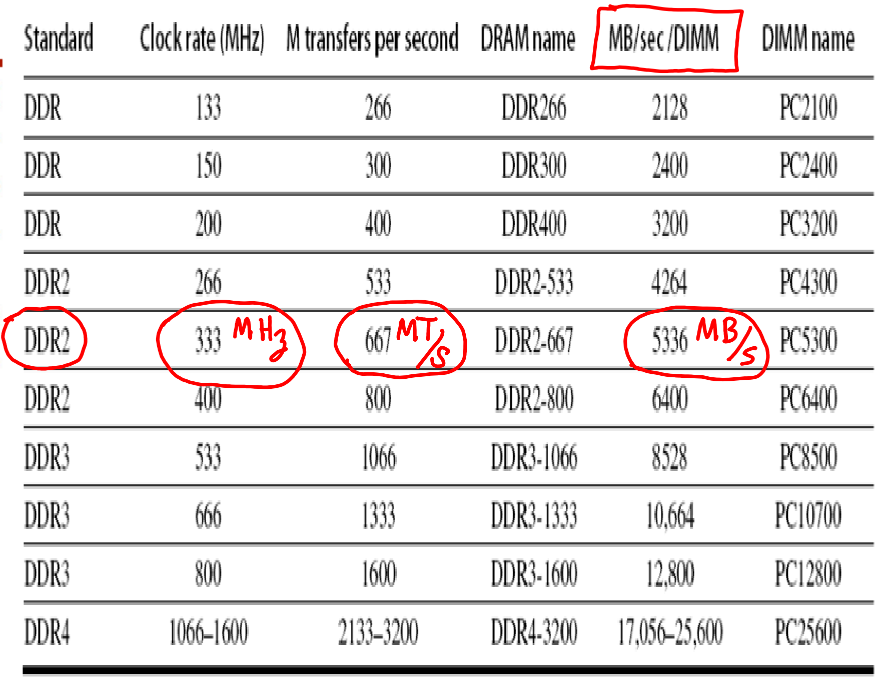
Major Lessons to Take Away

- Levels of abstraction (e.g. ISA→processor→RTL blocks→gates)
  - Simplifies design process for complex systems
  - Need good specification for each level
- Pipelining
  - Improve throughput of an engine by overlapping tasks
  - Watch out for dependencies between tasks
- Caching
  - Maintain a close copy of frequently accessed data
  - Avoid long accesses or expensive recomputation (memoization)
  - Think about associativity, replacement policy, block size
  - Impact of algorithm and data structure layout on locality
- Indirection (e.g. virtual→physical address translation)
  - Allows transparent relocation, sharing, and protection
- Overlapping (e.g. CPU work & DMA access)
  - Hide cost of expensive tasks by executing them in parallel with other useful work
  - Parallelism

Sun Fire x4150 1U server

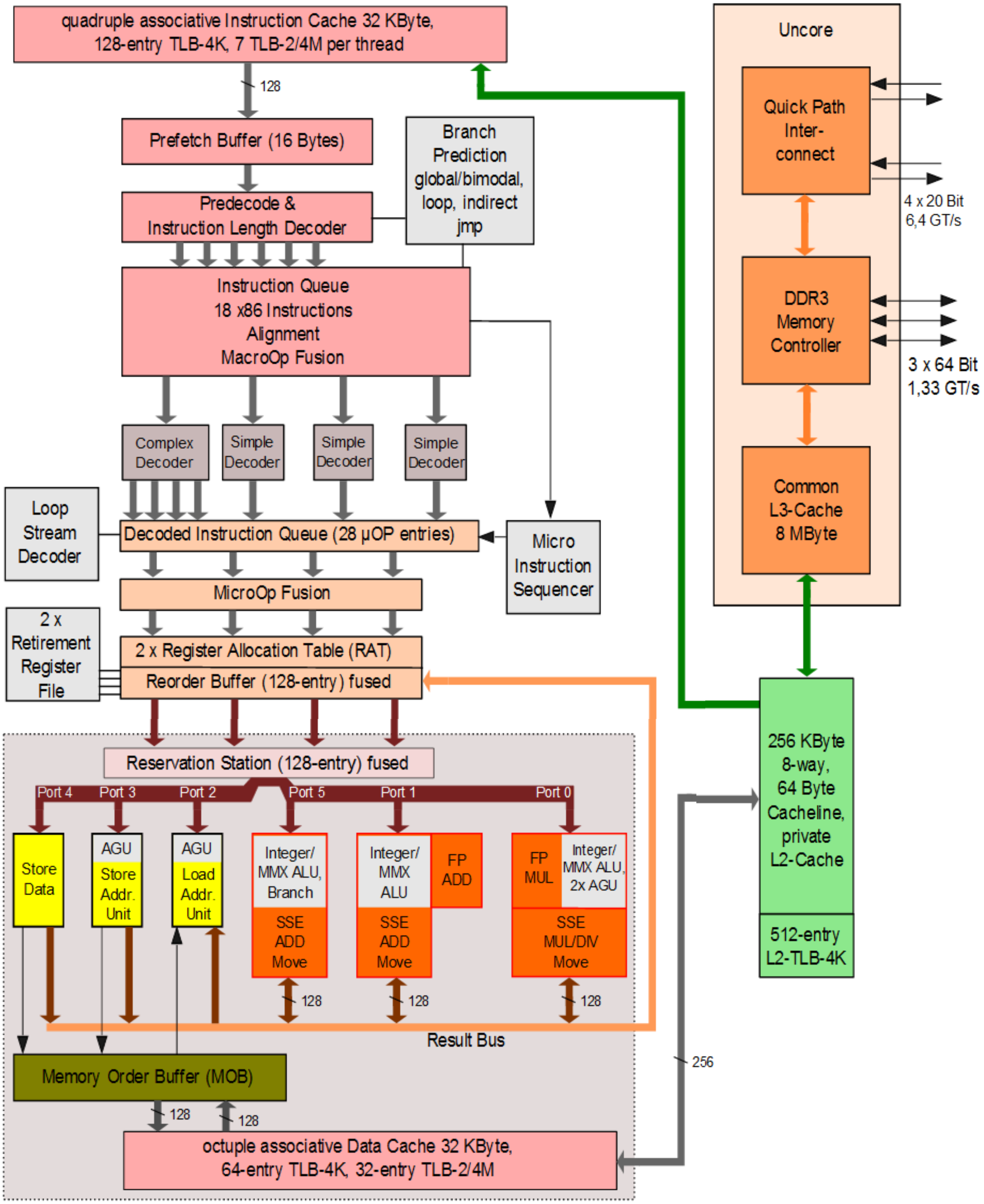






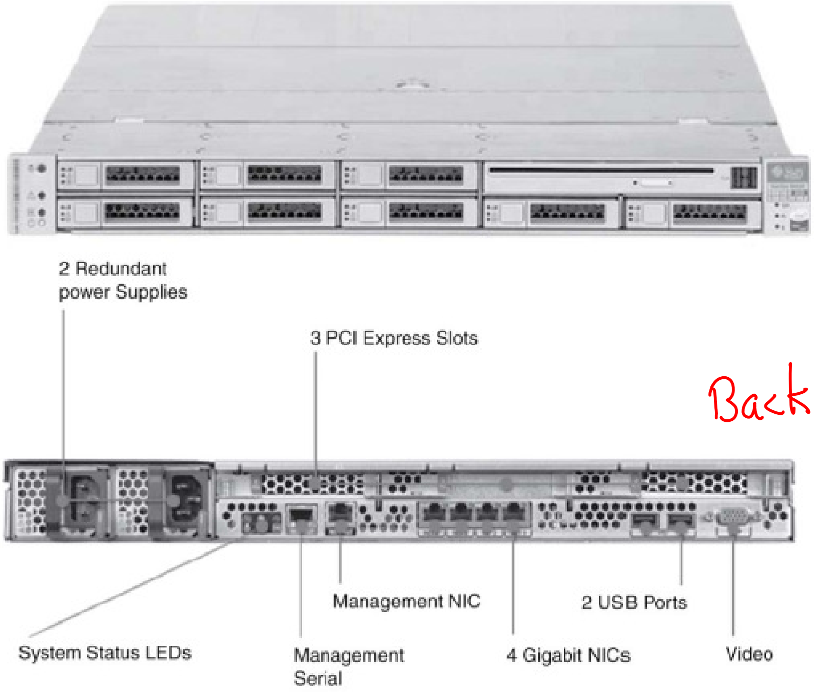
Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333 MHz	667 MT/s	DDR2-667	5336 MB/s	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Intel Nehalem microarchitecture


quadruple associative Instruction Cache 32 KByte,
128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode &
Instruction Length Decoder
Instruction Queue
18 x86 Instructions
Alignment
MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU
Store Addr. Unit
AGU
Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte,
64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit
6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit
1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

Rack-Mounted Servers

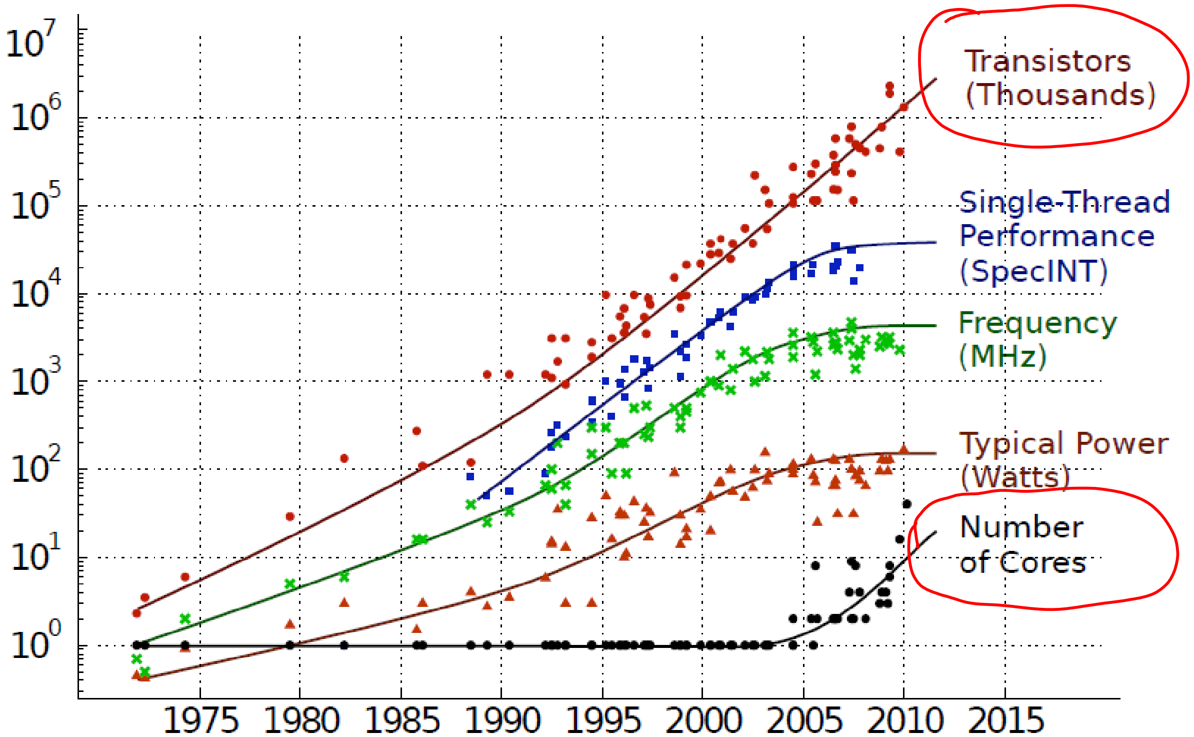
Sun Fire x4150 1U server



Back

10s – 1,000s

Motivation: Single Processor Performance Scaling

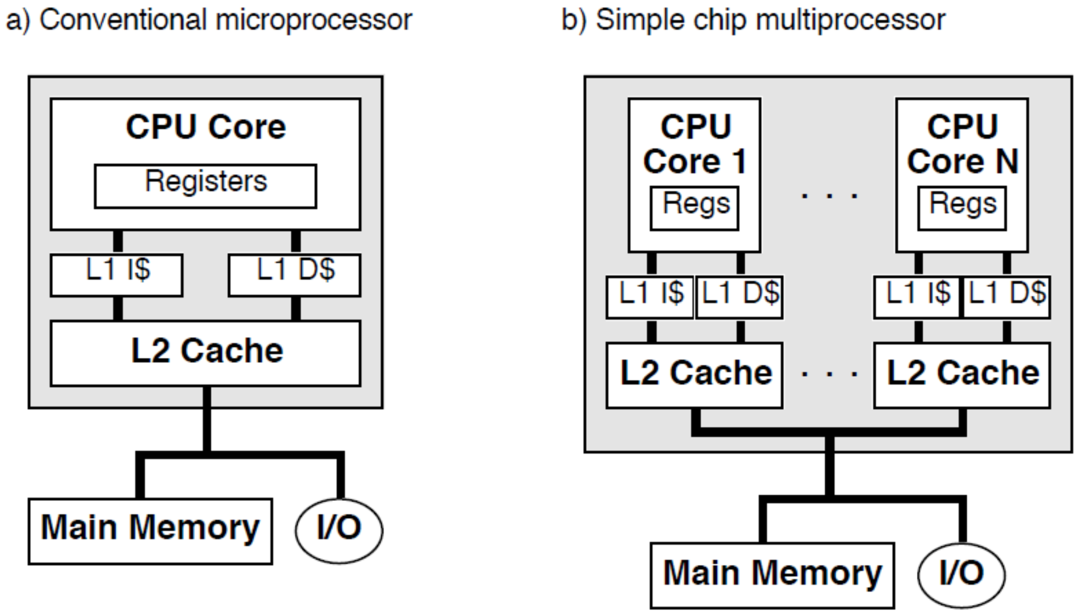


Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

Multi-core Chips
(aka Chip MultiProcessors or CMPs)

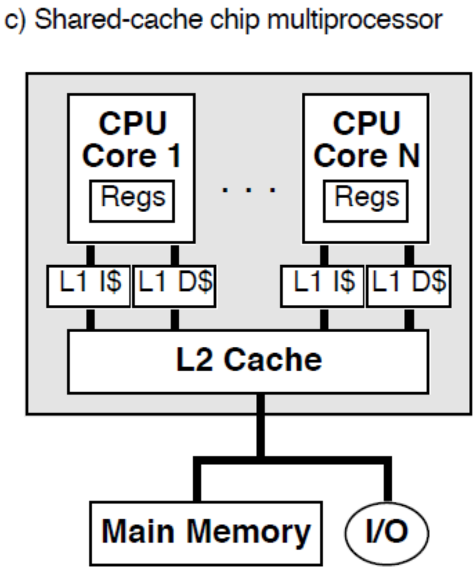
- Put multiple cores on a chip
  - Modular, scalable, simple, fault-tolerant
  - Relies on request-level, task-level, or data level parallelism
    - Hopefully there is lots of them
  - Can trade-off parallelism for power
- All processor vendors implement multi-core chips
  - In embedded, server, and even desktop systems
- Challenges
  - HW: what type of parallelism do we optimize for; what support do we provide for inter-processor communication; …
  - SW: how do we write parallel programs
    - A major crisis for the IT industry…

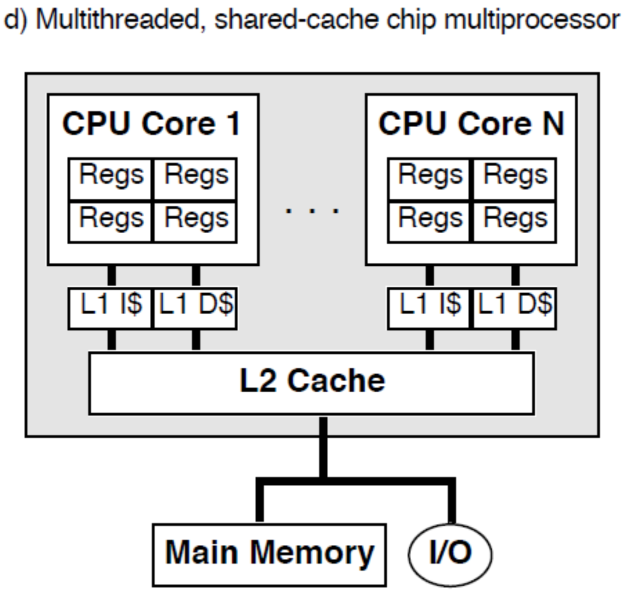
Sample of Multi-core Options


a) Conventional microprocessor

b) Simple chip multiprocessor

Early multi-core designs


c) Shared-cache chip multiprocessor


d) Multithreaded, shared-cache chip multiprocessor

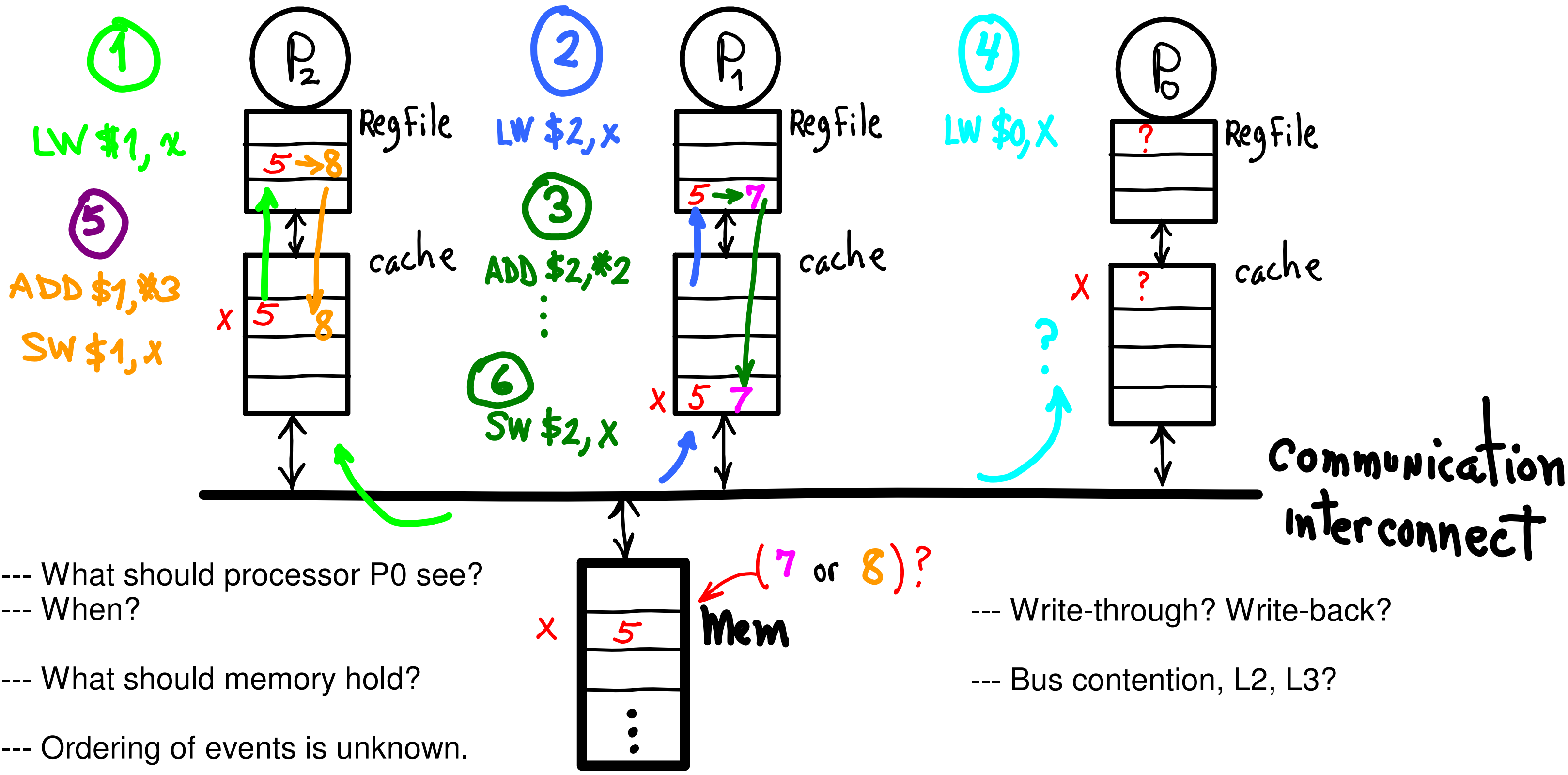
And There is Much More…

- Data parallel architectures
  - Vector processors, GPUs, …
- Heterogeneous multi-core
  - Already the mainstream in cellphones
- Multi-socket parallel systems
  - Multiple chips with multiple cores each
- Clusters
- Hybrids of the above…

Another HW Issue: Memory Model for Multi-core

- One approach: shared memory multi-cores
  - All core view the same physical address space for memory ?
- Advantages
  - Parallel tasks can communicate implicitly using loads and stores
  - Requires evolutionary changes to OS
  - App development: first focus on correctness, then on performance
- Disadvantages
  - Implicit communication is hard to optimize
  - Synchronization can get tricky
  - Introduces (again) the issue of cache coherence
    - The problem we had with DMAs, but now it's the common case…

Cache Coherence / Memory Consistency



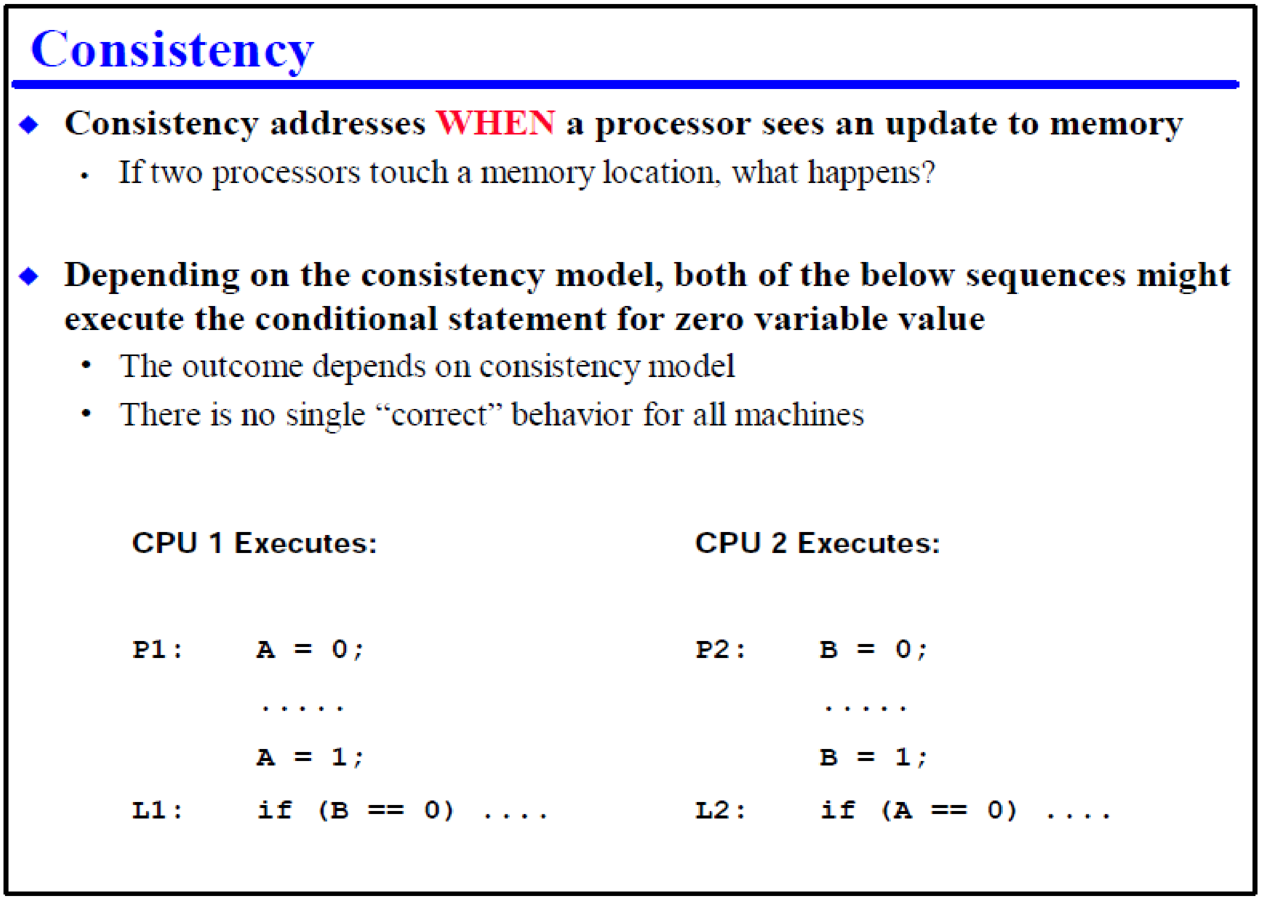
--- What should processor P0 see?
--- When?

--- What should memory hold?

--- Ordering of events is unknown.

--- Write-through? Write-back?

--- Bus contention, L2, L3?



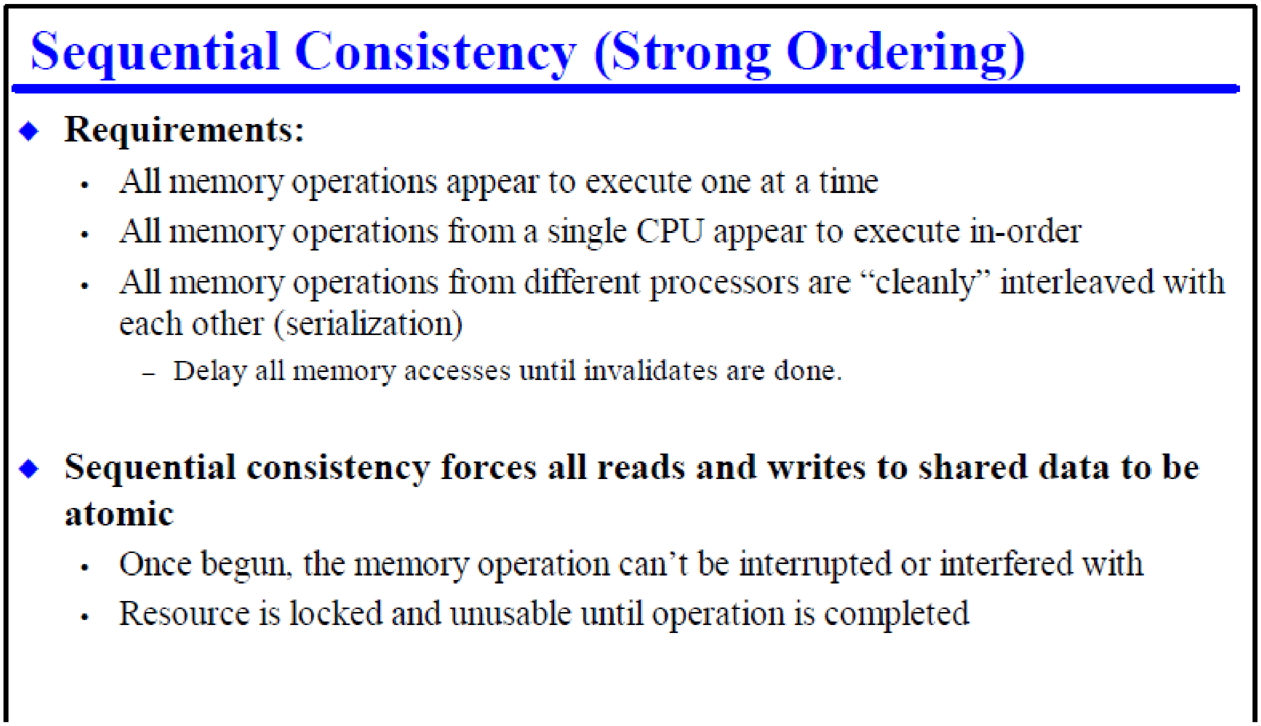
Consistency

- **Consistency addresses WHEN a processor sees an update to memory**
  - If two processors touch a memory location, what happens?

- **Depending on the consistency model, both of the below sequences might execute the conditional statement for zero variable value**
  - The outcome depends on consistency model
  - There is no single “correct” behavior for all machines

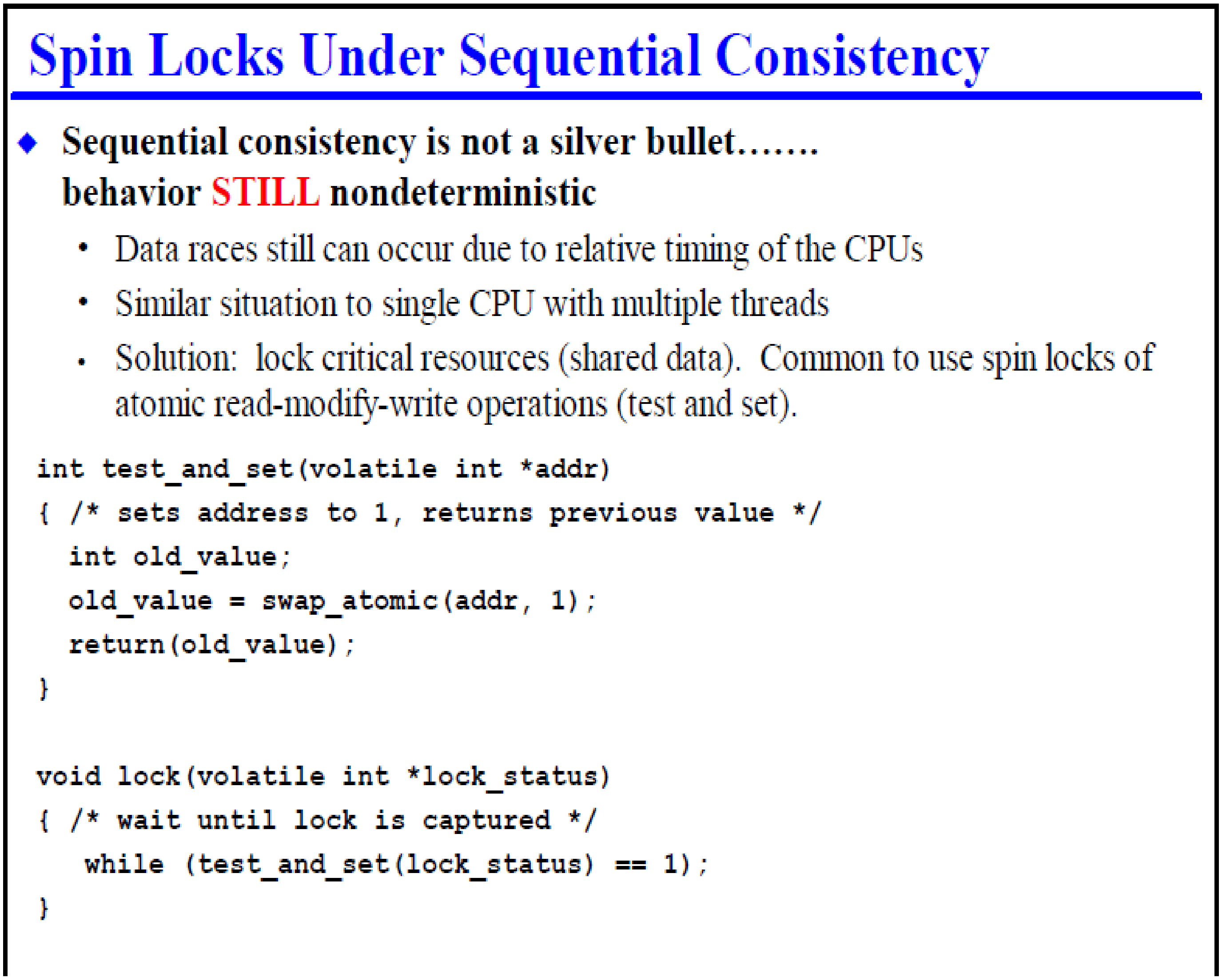
CPU 1 Executes:	CPU 2 Executes:

```
P1:     A = 0;                        P2:     B = 0;
        .....                                 .....
        A = 1;                                B = 1;
L1:     if (B == 0) ....              L2:     if (A == 0) ....
```



Sequential Consistency (Strong Ordering)

- **Requirements:**
  - All memory operations appear to execute one at a time
  - All memory operations from a single CPU appear to execute in-order
  - All memory operations from different processors are "cleanly" interleaved with each other (serialization)
    - Delay all memory accesses until invalidates are done.
- **Sequential consistency forces all reads and writes to shared data to be atomic**
  - Once begun, the memory operation can't be interrupted or interfered with
  - Resource is locked and unusable until operation is completed



Spin Locks Under Sequential Consistency

- **Sequential consistency is not a silver bullet……. behavior STILL nondeterministic**
  - Data races still can occur due to relative timing of the CPUs
  - Similar situation to single CPU with multiple threads
  - Solution: lock critical resources (shared data). Common to use spin locks of atomic read-modify-write operations (test and set).

```
int test_and_set(volatile int *addr)
{ /* sets address to 1, returns previous value */
  int old_value;
  old_value = swap_atomic(addr, 1);
  return(old_value);
}

void lock(volatile int *lock_status)
{ /* wait until lock is captured */
   while (test_and_set(lock_status) == 1);
}
```

Sequential Consistency Problems

- **Can’t use important hardware optimizations**
  - Problem with anything that interferes with strict execution order
    - Write buffers, Write assembly caches, Non-blocking caches…
  - Not a problem with uniprocessors
- **May not be able to use important software optimizations**
  - If you want to be really strict about it, source code must execute as-is, so no:
    - Code motion, register allocation, eliminating common subexpressions…
  - Same problem exists with uniprocessor concurrency
- **Relaxed memory consistency models:**
  - Permit performance optimizations
  - BUT, require programmer to take responsibility for concurrency issues

Total Store Ordering

- **Relaxed Consistency**
  - Stores must complete in-order
  - But, stores need not complete before a read to a given location takes place
- **Allows reads to bypass pending writes.**
  - Store buffers allowed!
  - But, writes MUST exit the store buffer in FIFO order.
- **Problem: Other CPUs don’t check the store buffer for data.**
  - So, a read from CPU #2 might not see that data has “already” been changed by CPU #1
  - Synchronization of some sort required before reading potentially shared data

Partial Store Ordering

- **Even more relaxed consistency**
  - Stores to any given memory location complete in-order
  - But, stores to different locations may complete out of order
  - And, stores need not complete before a read to a given location takes place
  - Like total store ordering, but ordering concept applied only on a per-location basis

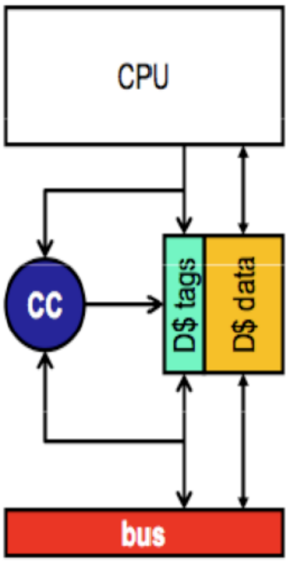
- **Additional Problem: Spin locks may not work**
  - Modifying a shared variable involves:
    - Writing to the variable's memory location
    - Changing the spin lock value to "available"
    - But, what if the spin lock write completes before the variable write?
  - Solution: hardware must support some sort of barrier synchronization
    - All CPUs wait at barrier until global memory state is synchronized
    - Release spin lock only after barrier synch.

Weak Consistency

- **Really relaxed consistency**
  - Anything goes, except at barrier synchronization points
  - Global memory state must be completely settled at each synchronization
  - Memory state may correspond to *any* ordering of reads and writes between synchronization points

- **Permits fastest execution**
  - But, managing concurrency is entirely the programmer's responsibility

Hardware Cache Coherence Using Snooping

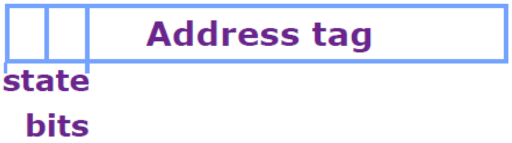


- Hardware guarantees that loads from all cores will return the value of the latest write
- Coherence mechanisms
  - Metadata to track state for cached data
  - Controller that snoops bus (or interconnect) activity and reacts if needed to adjust the state of the cache data
- There needs to be a serialization point
  - Bus, shared L2/L3, memory controller, …

- Suggest a snooping protocol for assuming write-through caches
  - What happens when a core writes what another cores caches?
  - What happens when a core reads what another core caches

MSI:
Simple Coherence Protocol for Write Back Caches

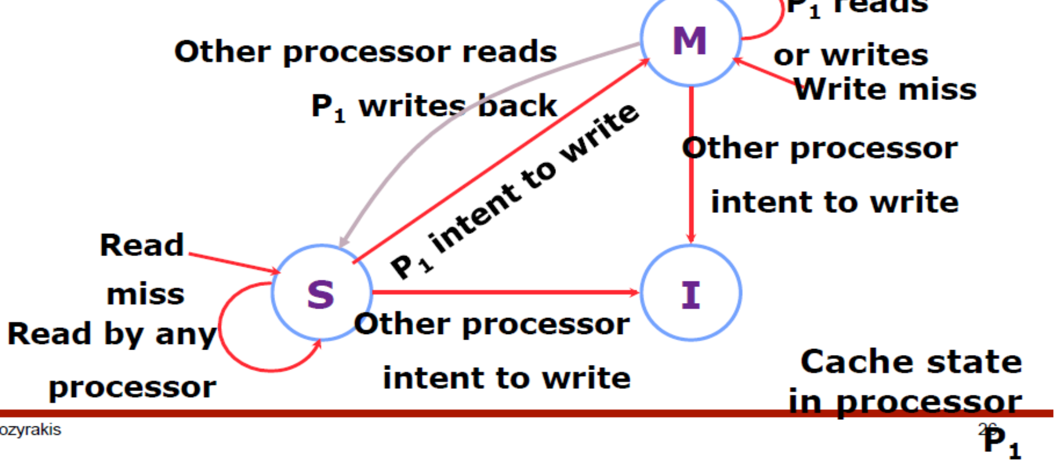
Each **cache line has a tag**



M: Modified

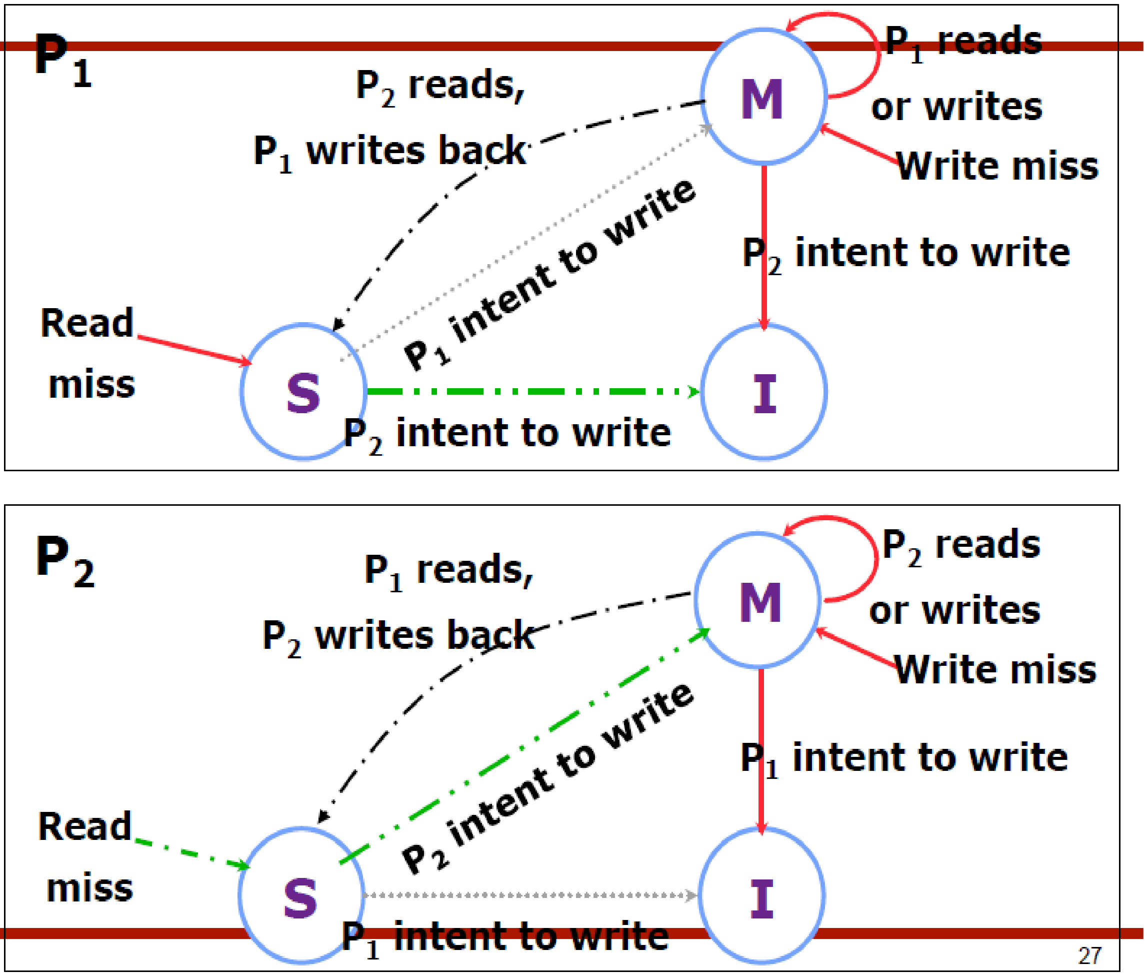
S: Shared

I: Invalid



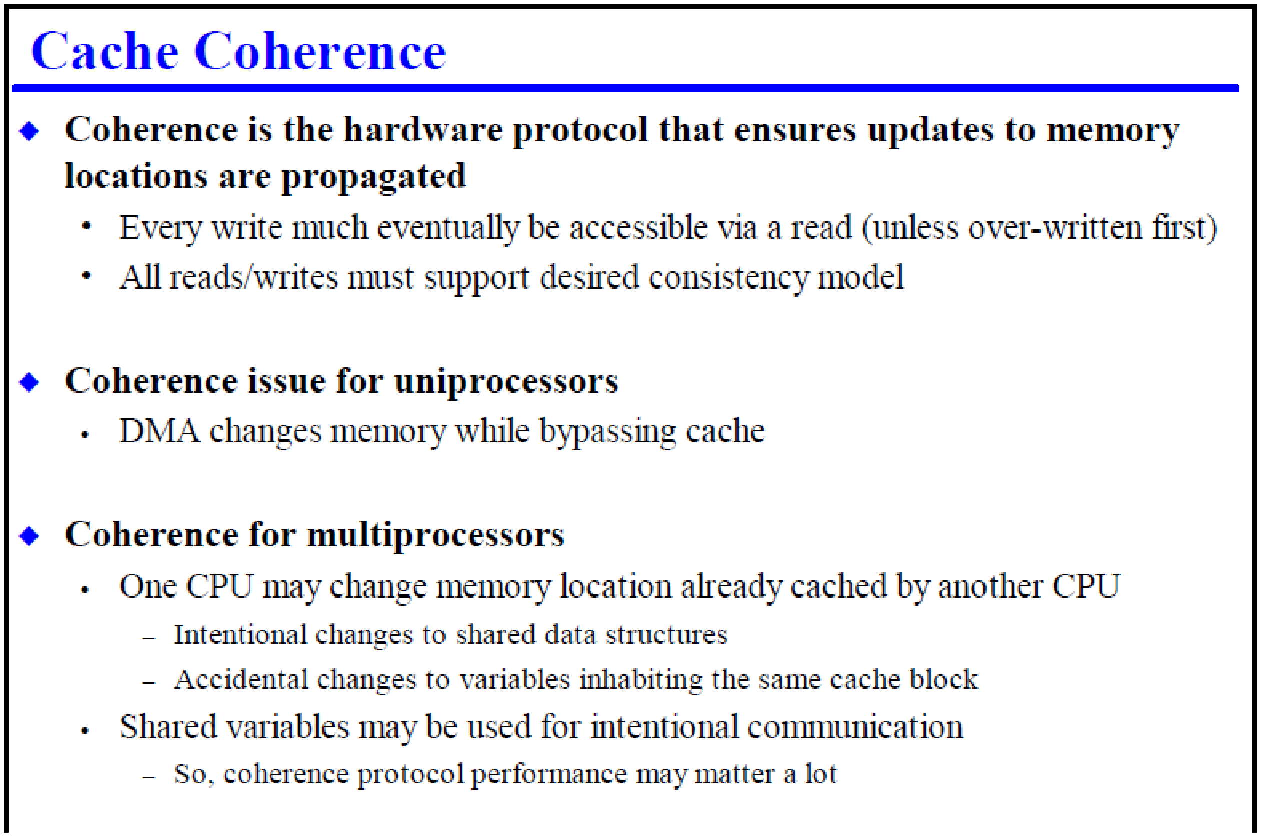
MSI Example with 2 Cores

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes



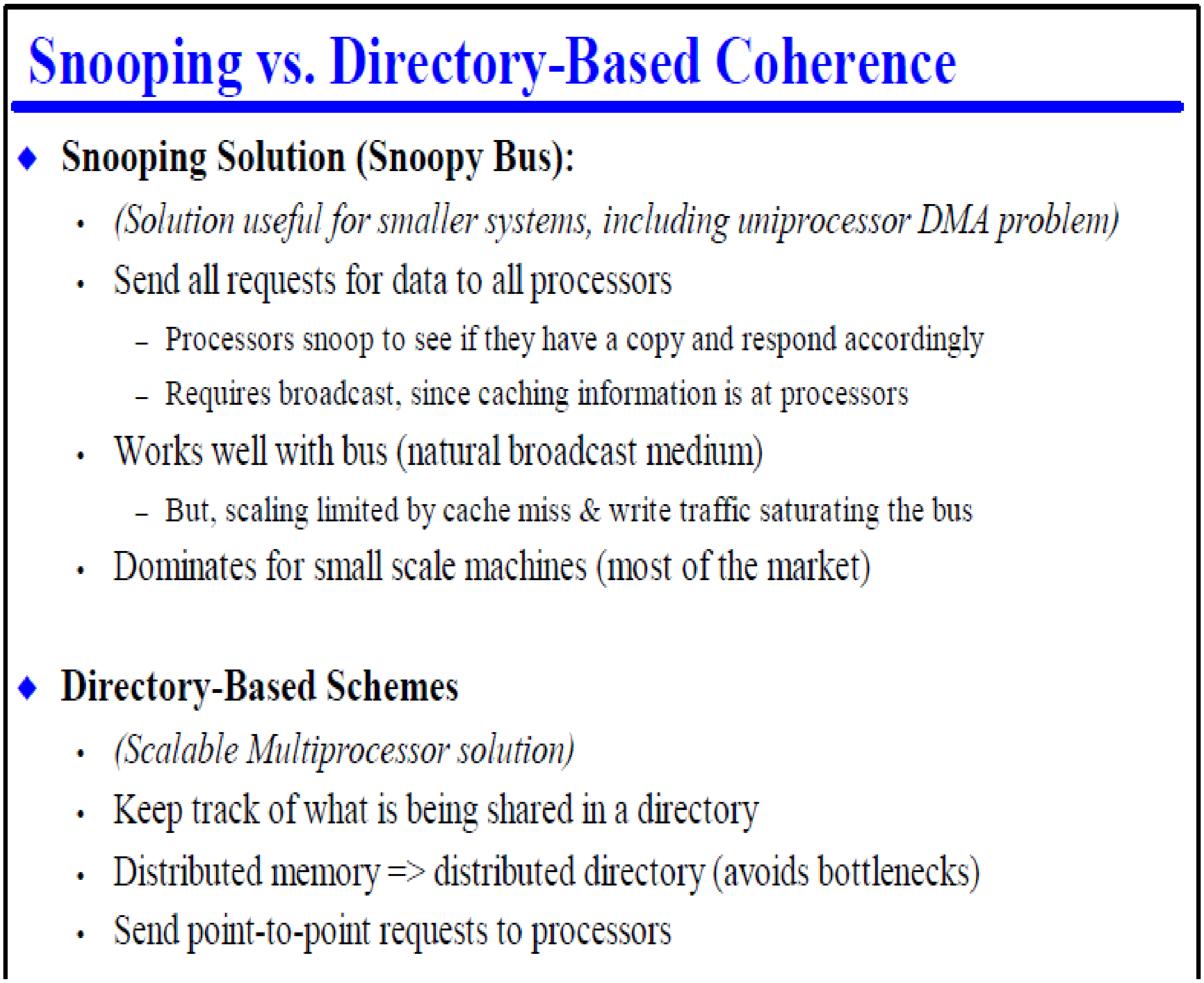
Quick Questions

- How many copies of a cache line can you have in S state?
- How many copies can you have in M state?
- Why is serialization important?



Cache Coherence

- **Coherence is the hardware protocol that ensures updates to memory locations are propagated**
  - Every write much eventually be accessible via a read (unless over-written first)
  - All reads/writes must support desired consistency model
- **Coherence issue for uniprocessors**
  - DMA changes memory while bypassing cache
- **Coherence for multiprocessors**
  - One CPU may change memory location already cached by another CPU
    - Intentional changes to shared data structures
    - Accidental changes to variables inhabiting the same cache block
  - Shared variables may be used for intentional communication
    - So, coherence protocol performance may matter a lot



Snooping vs. Directory-Based Coherence

- **Snooping Solution (Snoopy Bus):**
  - *(Solution useful for smaller systems, including uniprocessor DMA problem)*
  - Send all requests for data to all processors
    - Processors snoop 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)
    - But, scaling limited by cache miss & write traffic saturating the bus
  - Dominates for small scale machines (most of the market)
- **Directory-Based Schemes**
  - *(Scalable Multiprocessor solution)*
  - Keep track of what is being shared in a directory
  - Distributed memory => distributed directory (avoids bottlenecks)
  - Send point-to-point requests to processors

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: force other caches to update copy in main memory, then snoop that value
  - Can use a separate invalidate bus for write traffic

- **Write Broadcast Protocol:**
  - Write to shared data: broadcast on bus, processors snoop, and *update* copies
  - Read miss: memory is always up-to-date
  - Higher bandwidth (transmit data + address), but lower latency for readers
    - From a bandwidth point of view, looks like write-through cache

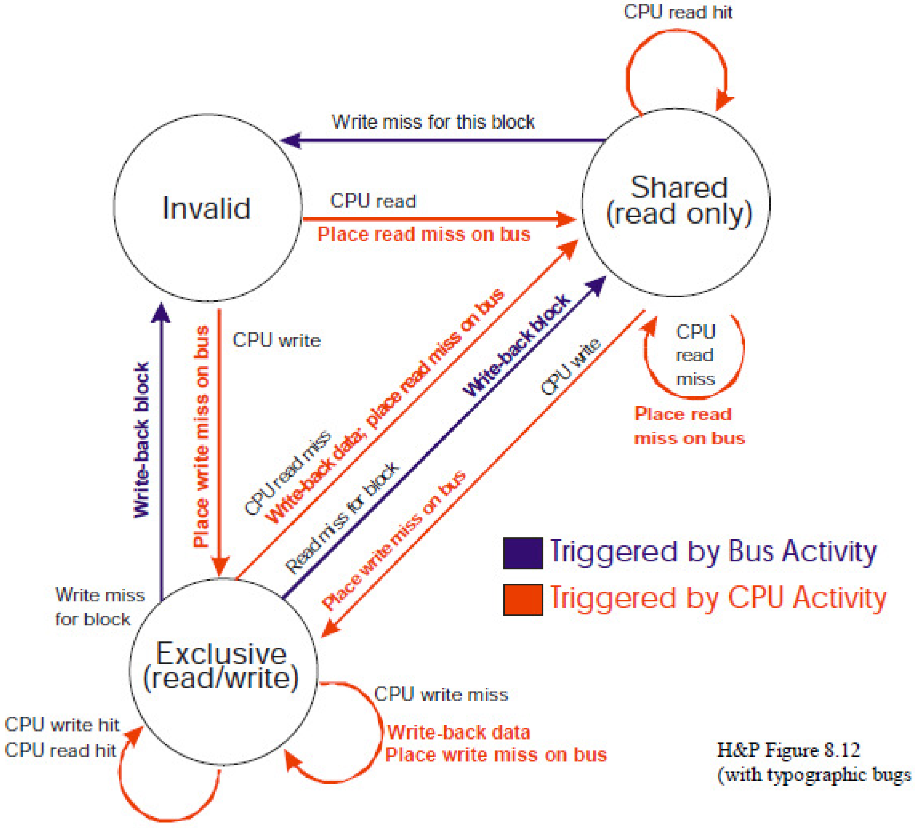
An Example Snoopy Protocol

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
  - Clean in some subset caches and up-to-date in memory
  - OR Dirty in exactly one cache
  - OR Not in any caches

- **Each cache block is in one state:**
  - 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**
- **Writes to clean line are treated as misses**

Snoopy Protocol Example

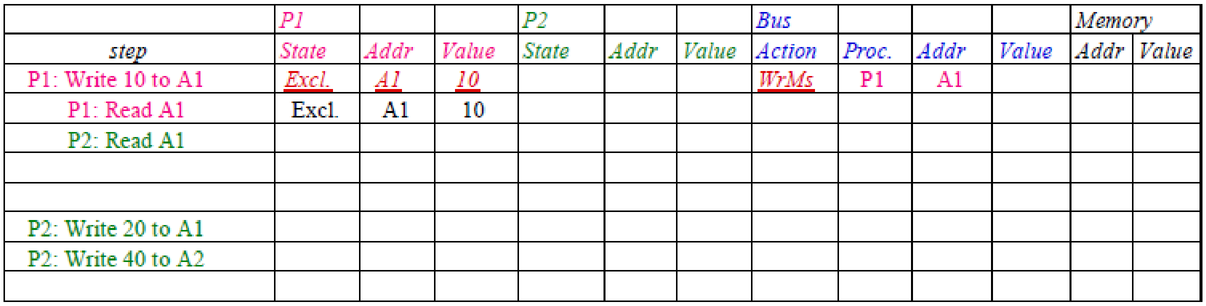


H&P Figure 8.12
(with typographic bugs fixed)

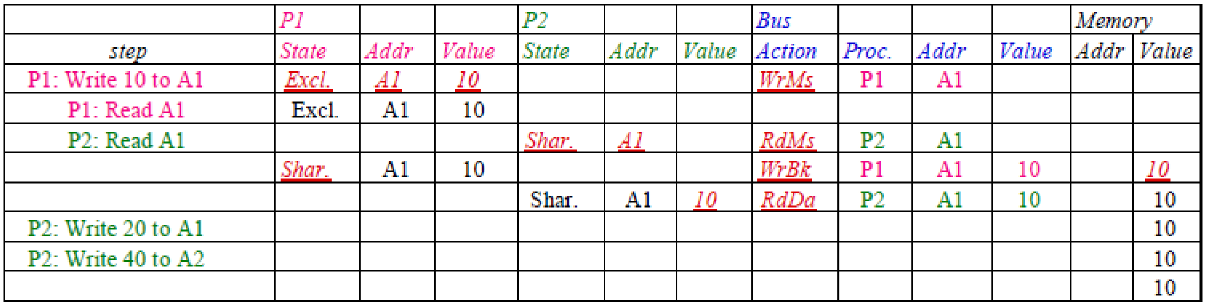
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

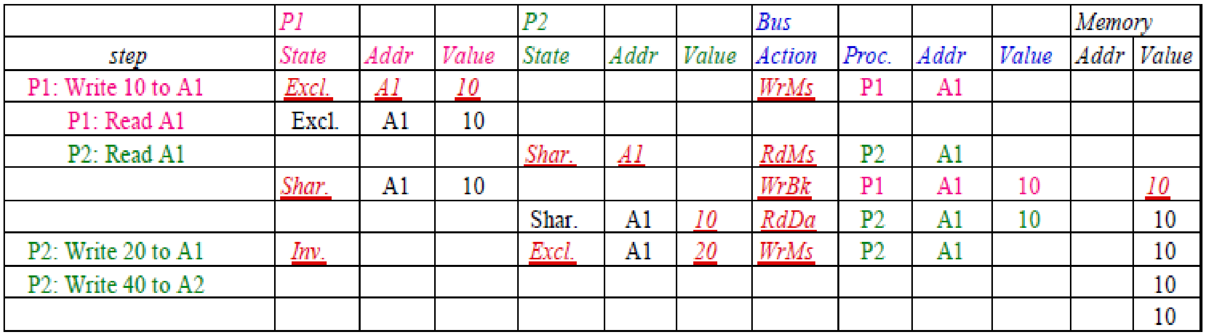
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												



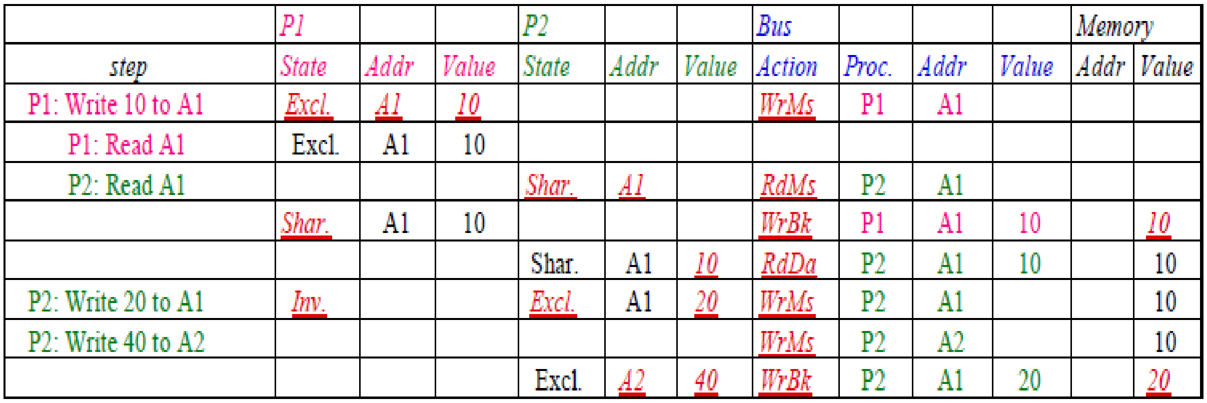
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												



	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		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1												10
P2: Write 40 to A2												10
												10



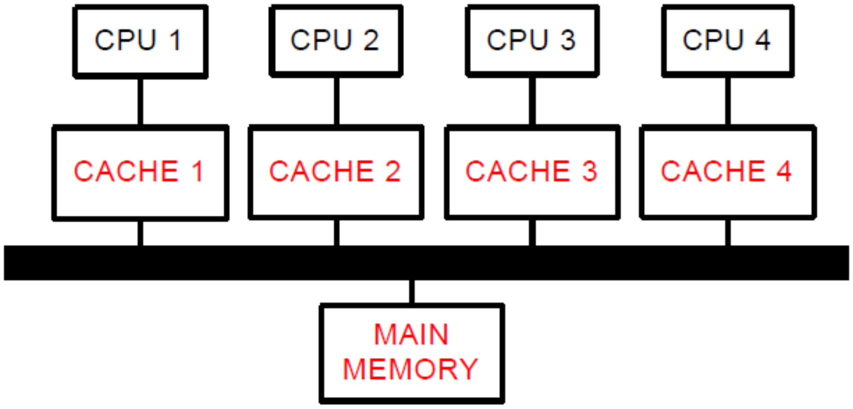
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to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



	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		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20		20

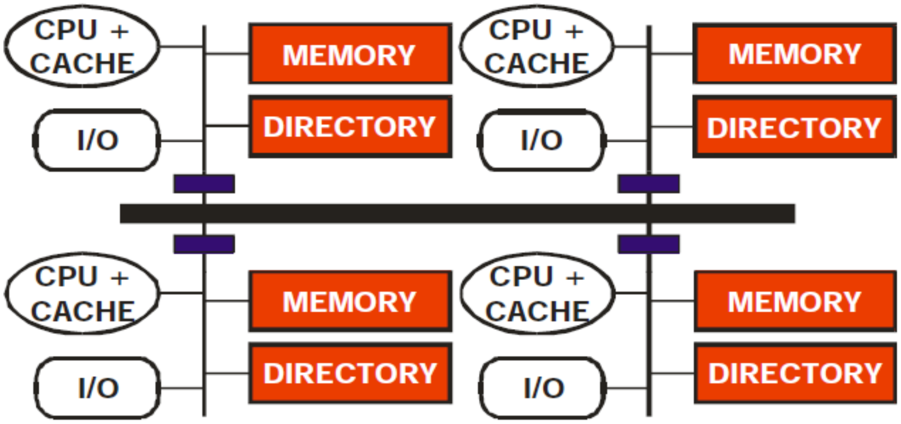
Multiprocessors -- UMA

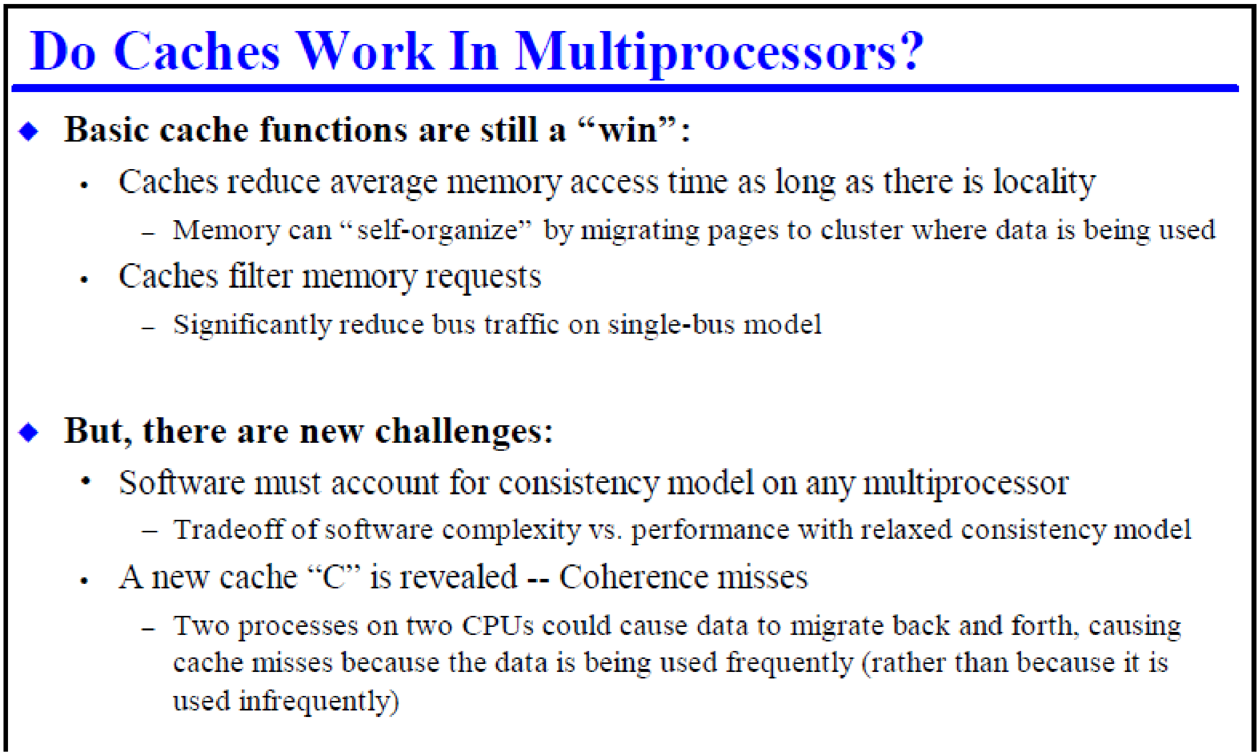
- **UMA - Uniform Memory Access**
  - Several CPUs interconnect with shared memory/common bus
  - Caches used to filter bus traffic
  - Works well up to 8-16 nodes (*e.g.,* Encore Multimax)



Multiprocessors -- NUMA

- **CC-NUMA - Cache Coherent Non-Uniform Memory Access**
  - Numerous clusters with interconnect; global address space
  - Scales to many CPUs (*as long as application has locality*)
  - Becomes a "multicomputer" if each cluster has a separate address space instead of global memory addressing



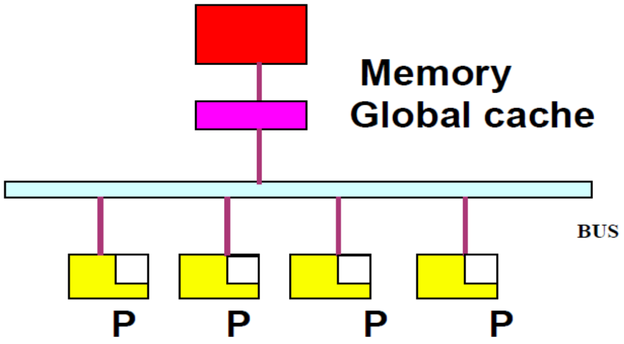


Do Caches Work In Multiprocessors?

- **Basic cache functions are still a "win":**
  - Caches reduce average memory access time as long as there is locality
    - Memory can "self-organize" by migrating pages to cluster where data is being used
  - Caches filter memory requests
    - Significantly reduce bus traffic on single-bus model
- **But, there are new challenges:**
  - Software must account for consistency model on any multiprocessor
    - Tradeoff of software complexity vs. performance with relaxed consistency model
  - A new cache "C" is revealed -- Coherence misses
    - Two processes on two CPUs could cause data to migrate back and forth, causing cache misses because the data is being used frequently (rather than because it is used infrequently)

Software Solutions

Compiler tags data as cacheable and non-cacheable. Only read-only data is considered cachable and put in private cache. All other data are non-cachable, and can be put in a global cache, if available.

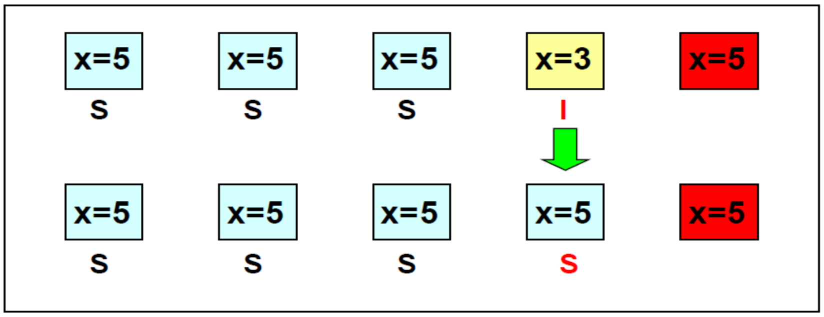


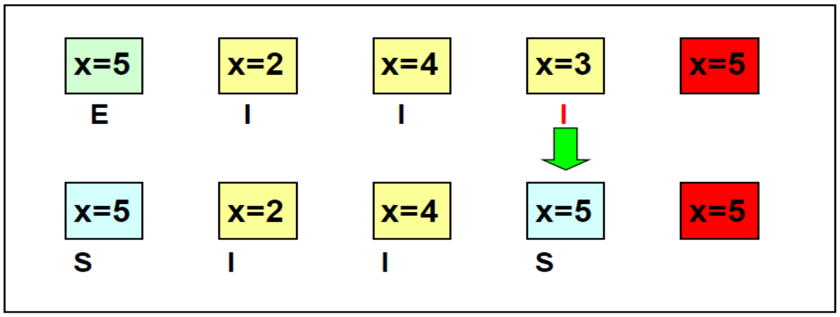
INVALID	Not valid
SHARED	Multiple caches may hold valid copies.
EXCLUSIVE	No other cache has this block, M-block is valid
MODIFIED	Valid block, but copy in M-block is not valid.

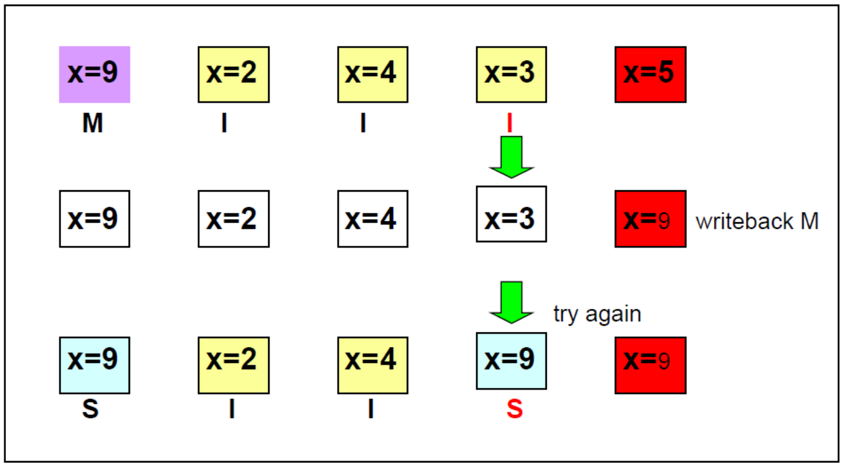
Event	*Local*	*Remote*
Read hit	Use local copy	No action
Read miss	I to S, or I to E	(S,E,M) to S
Write hit	(S,E) to M	(S,E,M) to I
Write miss	I to M	(S,E,M) to I

When a cache block changes its status from M, it first updates the main memory.

A. Read Miss

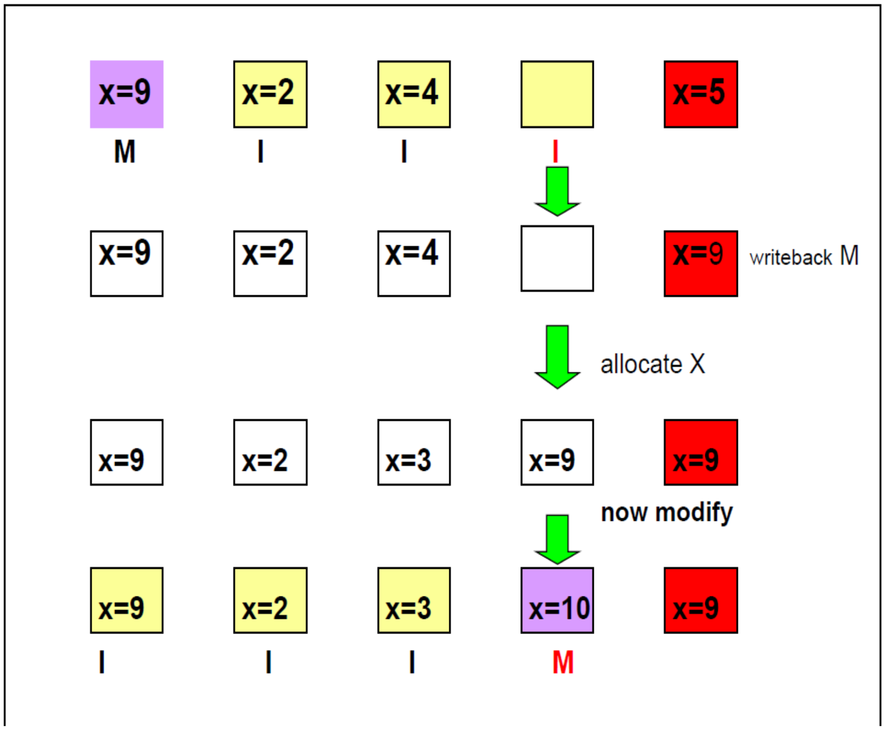






Following the read miss, the holder of the modified copy signals the initiator to try again. Meanwhile, it seizes the bus, and write the updated copy into the main memory.

C. Write Miss



Directory-based cache coherence

The snooping cache protocol does not work if there is no bus. Large-scale shared memory multiprocessors may connect processors with memories through switches.

A directory has to beep track of the states of the shared variables, and oversee that they are modified in a consistent way. Maintenance of the directory in a distributed environment is another issue.

Naïve solutions may lead to deadlock. Consider this:

P1 has a read miss for x2 (local to P2)
P2 has a read miss for x1 (local to P1)

Each will block and expect the other process to send the correct value of x: deadlock (!)

Cache coherence protocols guarantee that *eventually* all copies are updated. Depending on how and when these updates are performed, a read operation may sometimes return unexpected values.

Consistency deals with *what values can be returned to the user* by a read operation (may return unexpected values if the update is not complete). **Consistency model** is a contract that defines what a programmer can expect from the machine.

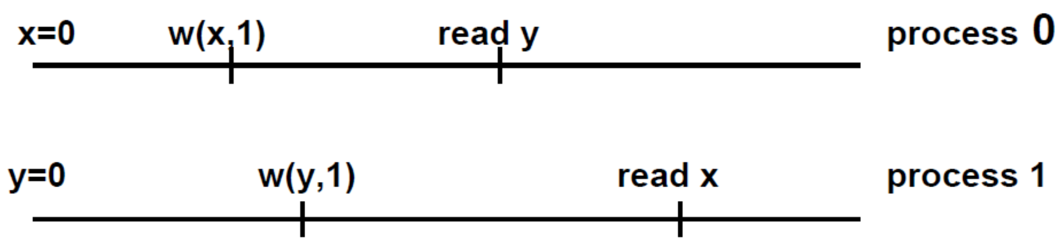
Sequential Consistency

Program 1.

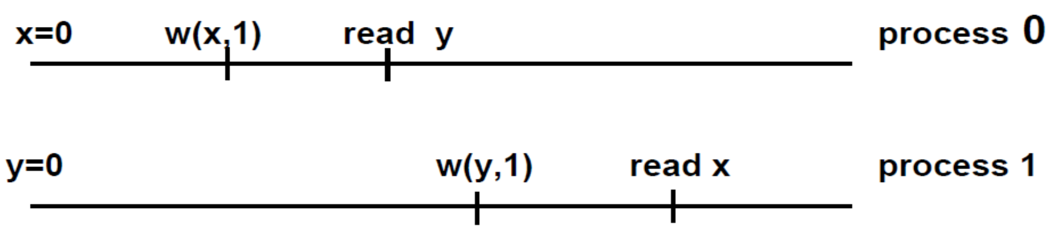
```
process 0                       process 1

{initially,x=0 and y=0}
x:=1;                      y:=1;
if (y=0) then x:=2;        if (x=0) then y:=2;
print  x;                  print  y;
```

If both processes run concurrently, then can we see a printout (x=2, y=2)?

A key question is: Did process 0 read y before process 1 modified it? One possible scenario is:



Here, the final values are: (x=1, y=1)



Here, the final values are: (x=2, y=1)

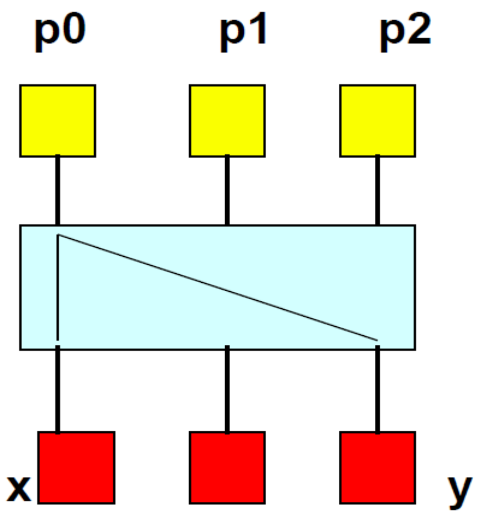
Properties of SC.

SC1. All operations in a single process are executed in program order.

SC2. The result of any execution is the same as if a single sequential order has been maintained among all operations.

Consider a switch-based multiprocessor.

Assume there is no cache.



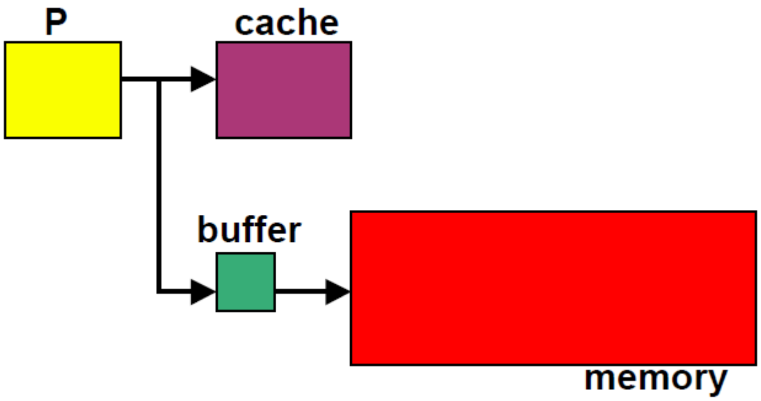
Process 0 executes: (x:=1; y:=2)

To prevent p1 or p2 from seeing these in a different order, p0 must receive an acknowledgement after every write operation.

Case 2

In a multiprocessor where processors have private cache, all invalidate signals must be acknowledged.

Write-buffers and New problems



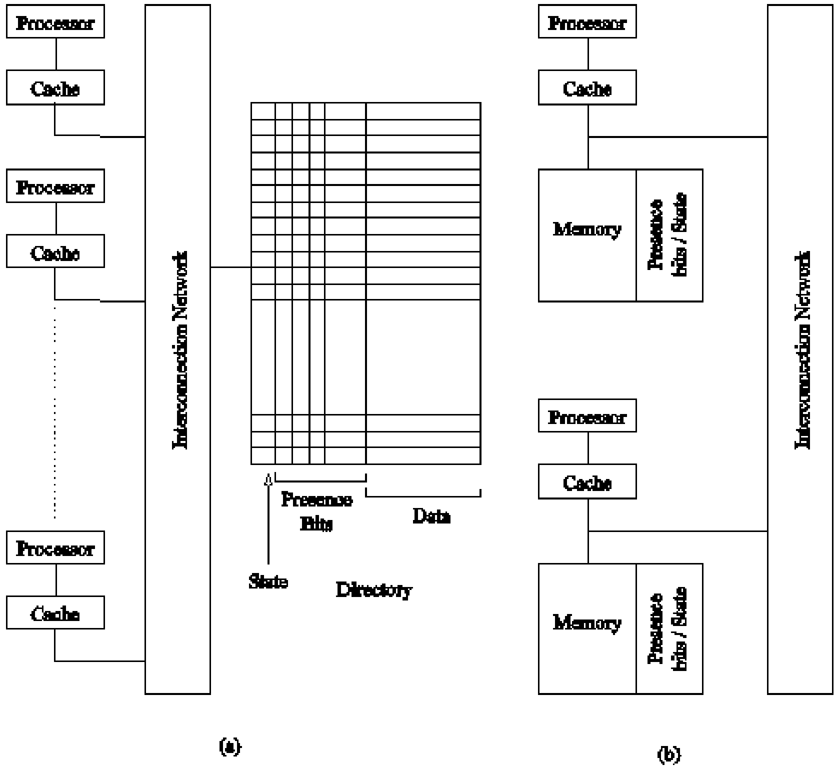
```
process 0                          process 1

{initially,x=0 and y=0}
x:=1;                              y:=1;
if (y=0) then x:=2;                if (x=0) then y:=2;
print  x;                          print  y;
```

Let both x:=1 and y:=1 be written into the write buffers, but before the memory is updated, let the two if statements be evaluated.

Both can be true, and (x:=2, y:= 2) are possible!

This violates sequential consistency.

Directory Based Systems

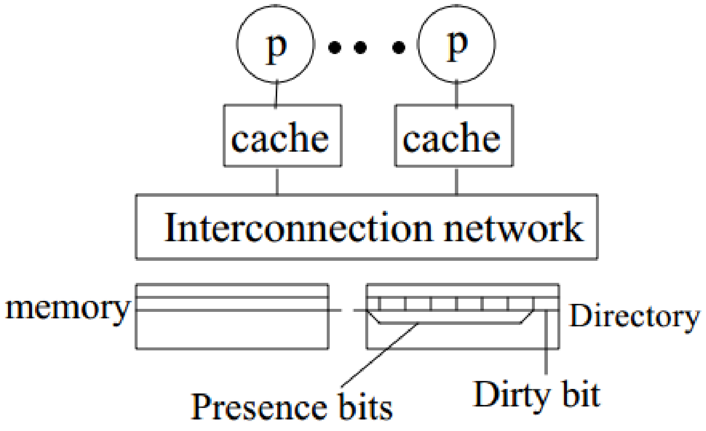


Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.

Key idea :keep track in a global directory (in main memory) of which processors are caching a location and the state.

- Directory based schemes allow scaling
  - They avoid broadcasts by keeping track of all Pes caching a memory block, and then using point-to-point messages to maintain coherence
  - They allow the flexibility to use any scalable point-to-point network

Basic Scheme (Censier and Feautrier)



- Assume K processors
- With each cache-block in memory: K presence bits and 1 dirty bit
- With each cache-block in cache : 1 valid bit and 1 dirty (owner) bit

READ MISS

Read from main-memory by PE_i

- If dirty bit is off then {read from main memory;turn p[i] ON; }
- If dirty bit is ON then {recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF;turn p[i] ON; supply recalled data to PE_i;}

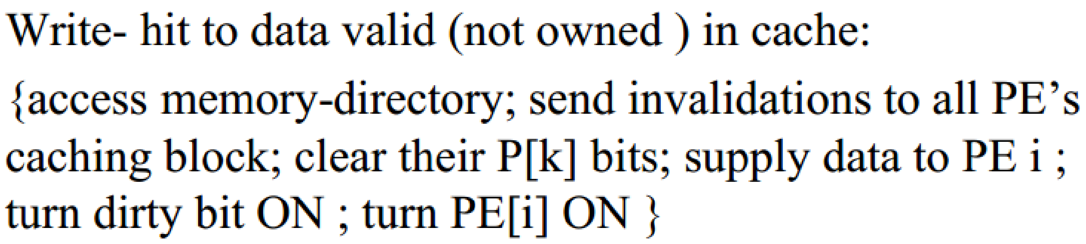
WRITE MISS

If dirty-bit OFF then

{supply data to PE_i; send invalidations to all PE's caching that block and clear their P[k] bits; turn dirty bit ON; turn P[i] ON; .. }

If dirty bit ON then

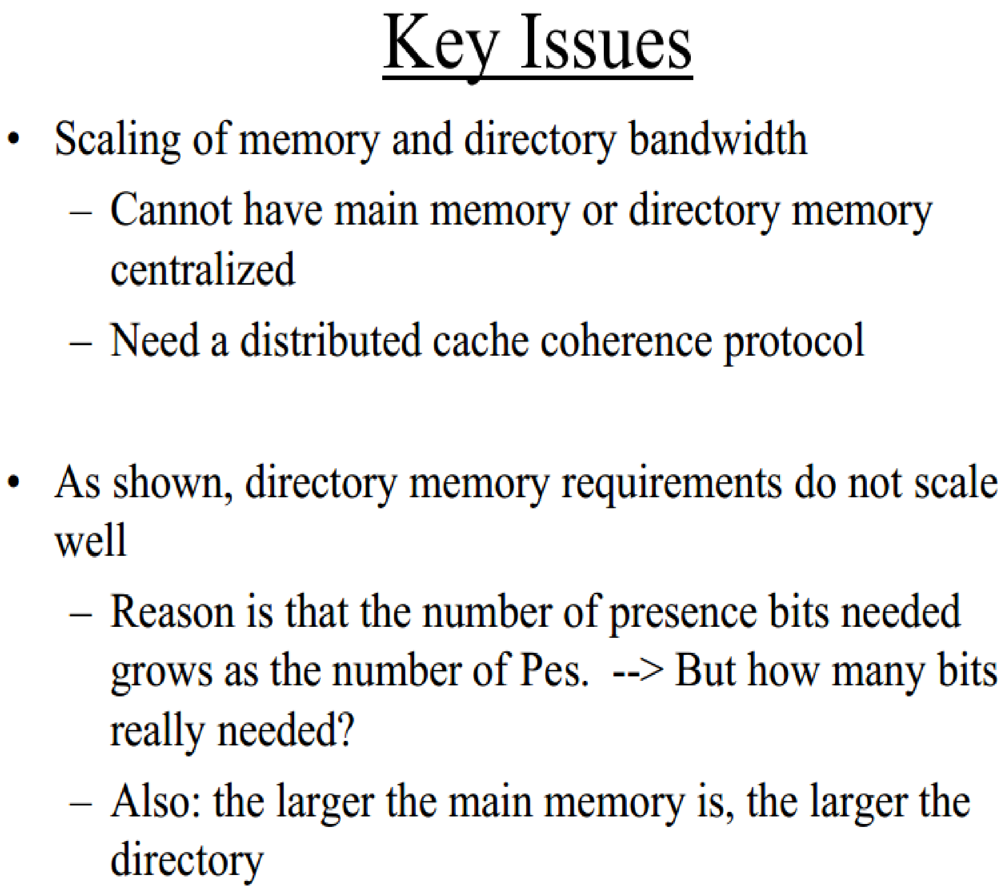
{recall the data from owner PE which invalidates itself; (update memory); clear bit of previous owner; forward data to PE i; turn bit PE[I] on; (dirty bit ON all the time) }



Write- hit to data valid (not owned) in cache:

{access memory-directory; send invalidations to all PE’s caching block; clear their P[k] bits; supply data to PE i ; turn dirty bit ON ; turn PE[i] ON }

WRITE HIT NOT OWNED



Key Issues

- Scaling of memory and directory bandwidth
  - Cannot have main memory or directory memory centralized
  - Need a distributed cache coherence protocol

- As shown, directory memory requirements do not scale well
  - Reason is that the number of presence bits needed grows as the number of Pes. --> But how many bits really needed?
  - Also: the larger the main memory is, the larger the directory

http://www.icsa.inf.ed.ac.uk/research/groups/hase/models/dir-cache/

But there is much more

- There are many alternative coherence models
  - Track different states to optimize communication cases
  - Broadcast vs directory based coherence
  - Protocols that deal with multiple levels of caches, multiple chips, etc

- Other alternative: shared memory without cache coherence
  - Do you see any issues?

- Other alternative: message passing memory models
  - Each core has a separate physical memory space
  - Explicit communication (send/receive) needed to exchange data
    - E.g., think of the way computers communicate on Internet