CS-200
Computer Architecture
—
Part 5a. Multiprocessors
Cache Coherence

Paolo Ienne
<paolo.ienne@epfl.ch>

Some slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh
of University of Illinois, EPFL, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University

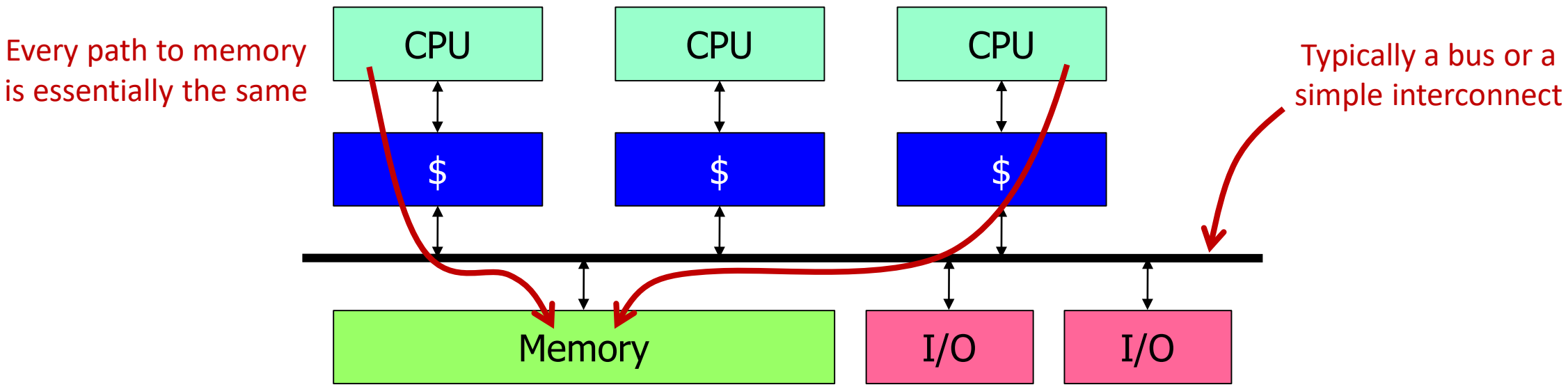
Flynn's Taxonomy (1966)

Single/Multiple Instruction Streams (= programs)

Single/Multiple Data Streams

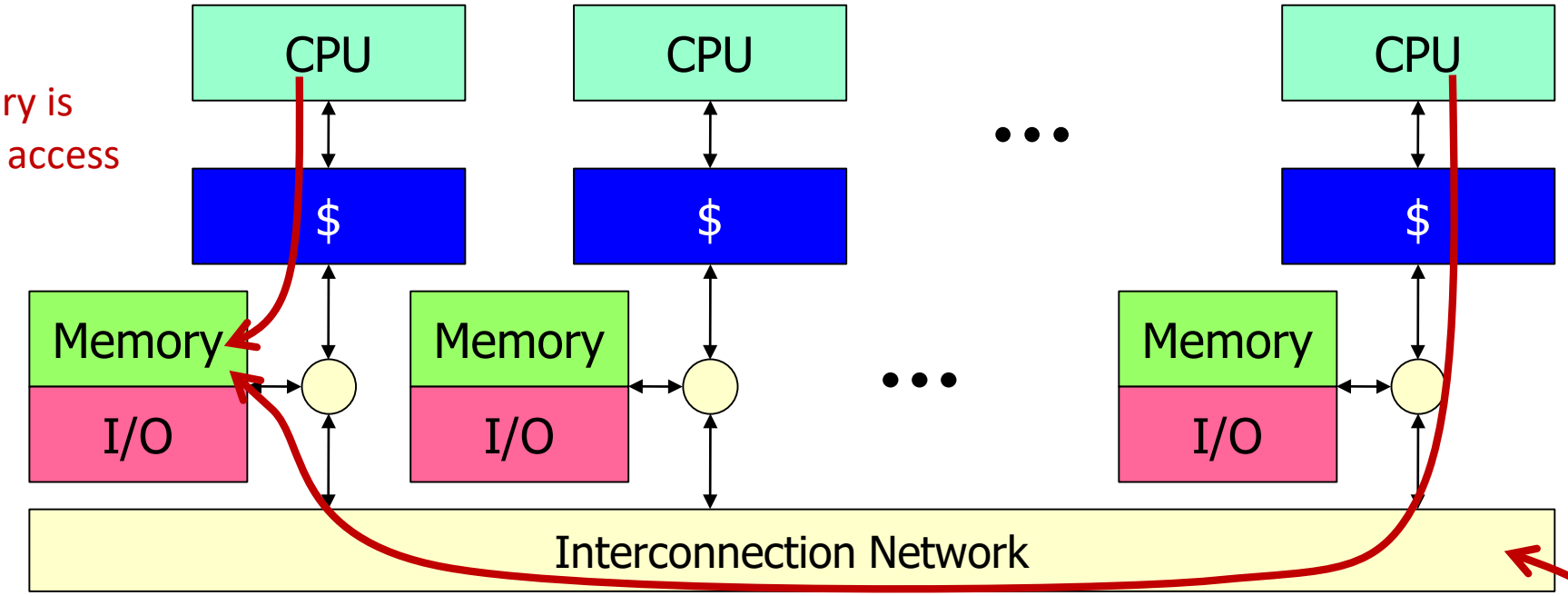
- **SISD**: Uniprocessors, what we have seen so far
- **SIMD**: A single program is run on multiple data sets at once. Classic examples are Vector Architectures for high-performance computing, now fairly rare. More commonly, x86 supports SIMD through various ISA extensions: MMX (1996), SSE (1999-2008), and AVX (2011-2016)
- **MIMD**: General form of parallelism with each processor executing its own program on its own data

Shared-Memory Multiprocessors (UMA = Uniform Memory Access)



- Limited scalability (4-16 processors?)
- Very simple and fairly traditional architecture

Distributed-Memory Multiprocessors (NUMA = Nonuniform Memory Access)



- More scalable and cost effective way to grow memory system
  - Communication is more complex and higher latency

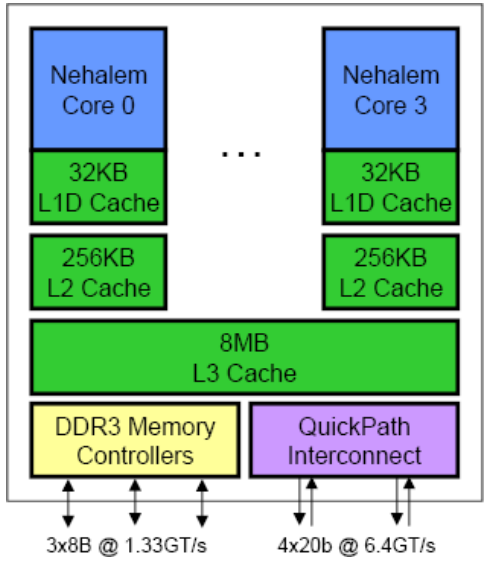
Programming Paradigms

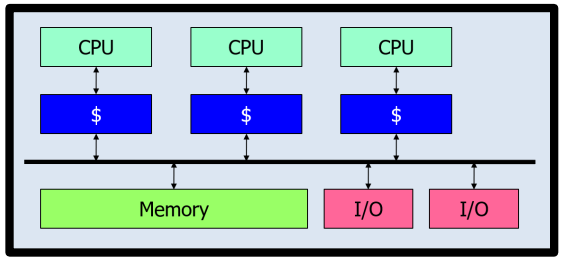
- **Shared-Memory**
  - Data exchanged **implicitly** through shared variables in a common memory space
  - Standard libraries (e.g., **OpenMP**) simplify programming
  - Natural on shared-memory architectures (e.g., **SMP**, **NUMA**)
  - Can be implemented as **Distributed Shared Memory (DSM)** on systems with physically distributed memory, leveraging virtual memory abstractions (e.g., **TreadMarks** for DSM; **Apache Spark** for a DSM-like abstraction in big data) or using hardware-based approaches at the cache-block level (e.g., **HPE Superdome**)
- **Message Passing**
  - Data exchanged **explicitly** by sending and receiving messages over a network or interconnect
  - Standard libraries (e.g., **MPI**) are widely used
  - Natural on distributed-memory systems with private memory per processor
  - Can also be implemented on shared-memory systems (e.g., **NUMA**), though it may introduce unnecessary overhead compared to native shared-memory programming

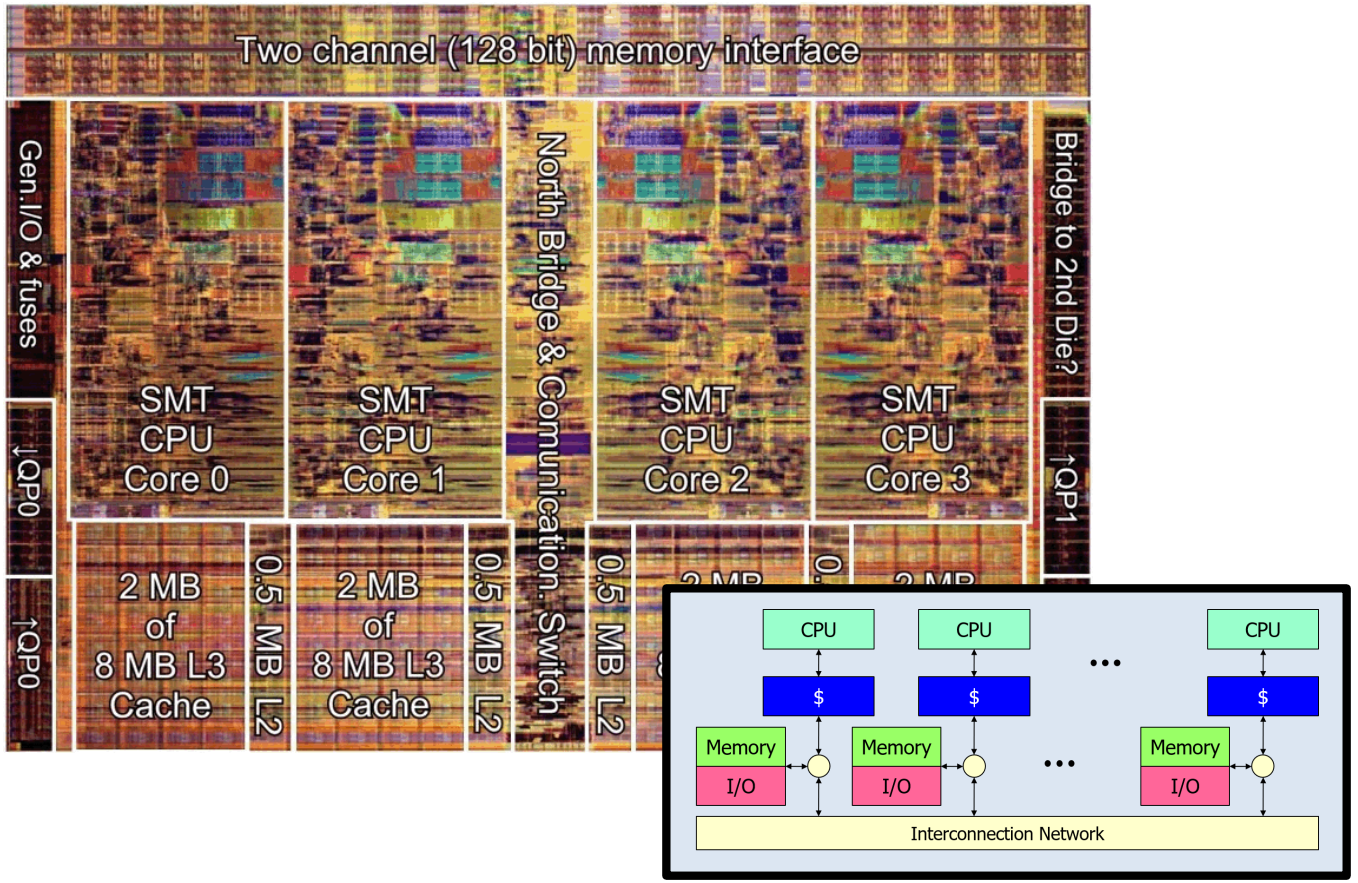
Why (Hardware) Shared Memory?

- **Advantages:**
  - For applications looks like a multitasking uniprocessor
  - For OS only evolutionary extensions required
  - Easy to do communication without OS
  - Software can worry about correctness first, then performance
- **Disadvantages:**
  - Communication is implicit, hence harder to optimize
  - Proper synchronization is complex
  - Hardware designers must implement
- **Result:**
  - **Symmetric Multiprocessors (SMPs)** were the foundation of early supercomputers but gave way to distributed-memory message-passing systems as scaling issues made SMPs less efficient
  - **Chip Multiprocessors (CMPs)** or **multicore** processors dominate as the most widespread form of parallel computing, driving multibillion-dollar markets

Intel Nehalem (2008)



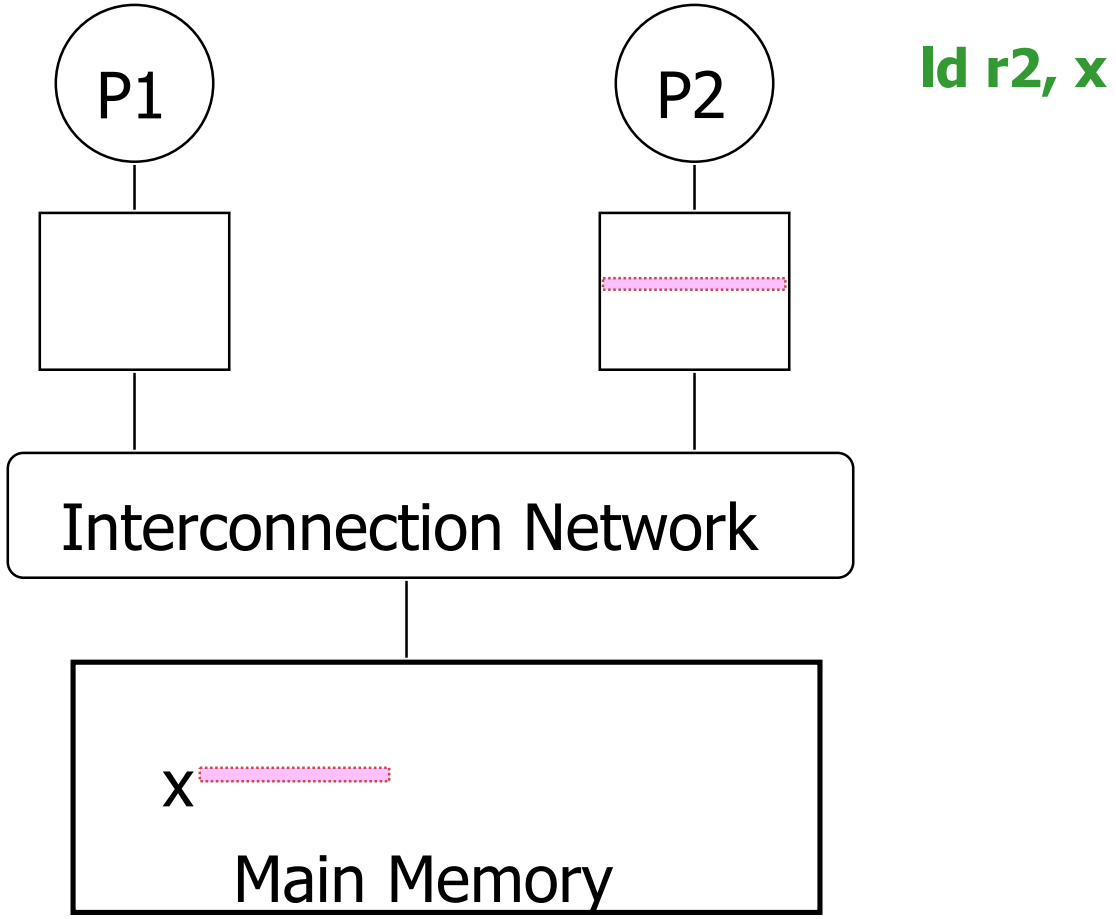




Source: http://www.chip-architect.com/

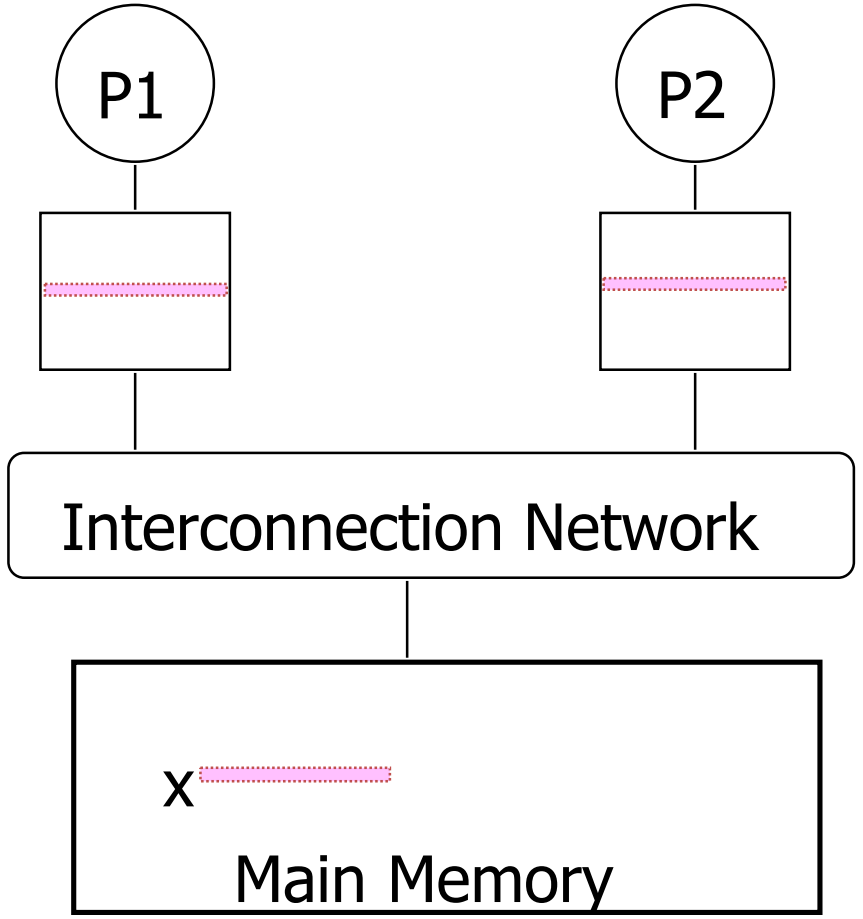
Note that the L3 cache is **logically shared** but **physically distributed** across the four cores
→ even **a single CMP** is in fact **implemented** as a **NUMA** architecture

Cache Coherence Problem
Step 1



Cache Coherence Problem
Step 2

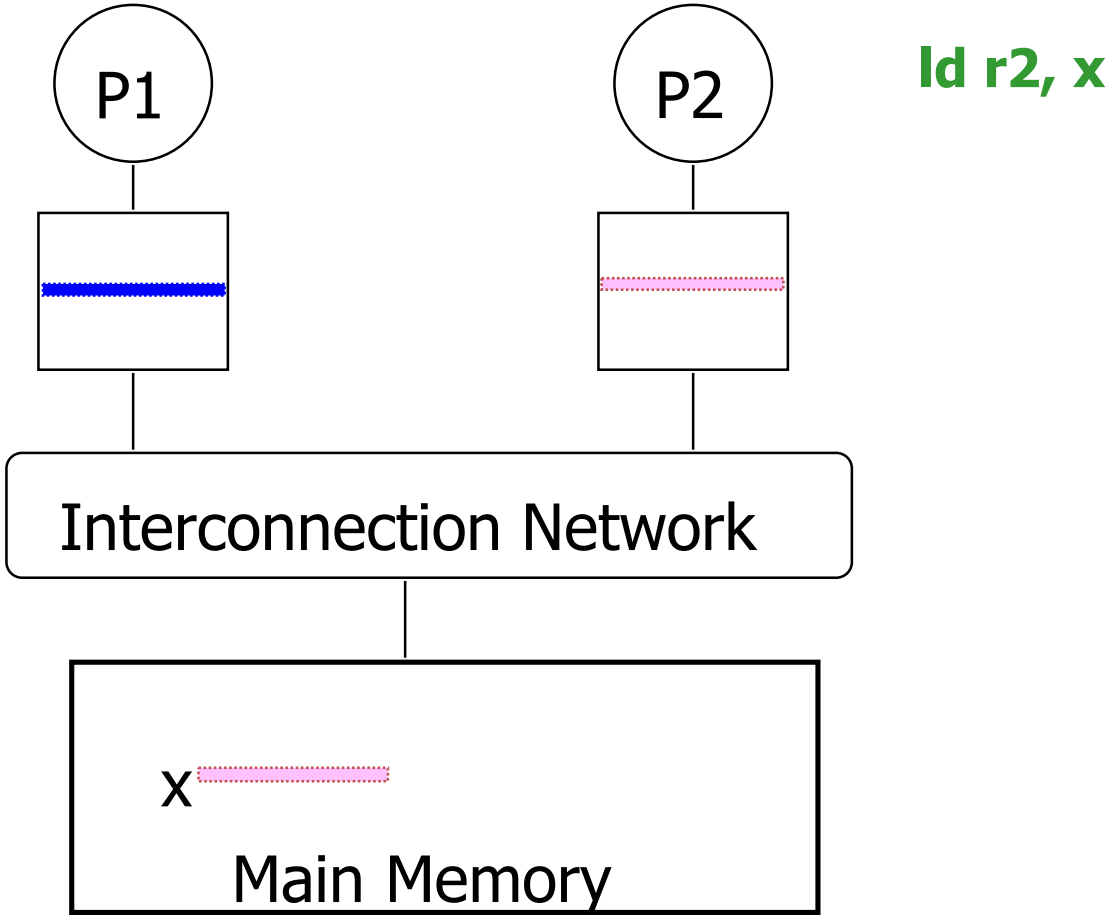
ld r2, x



ld r2, x

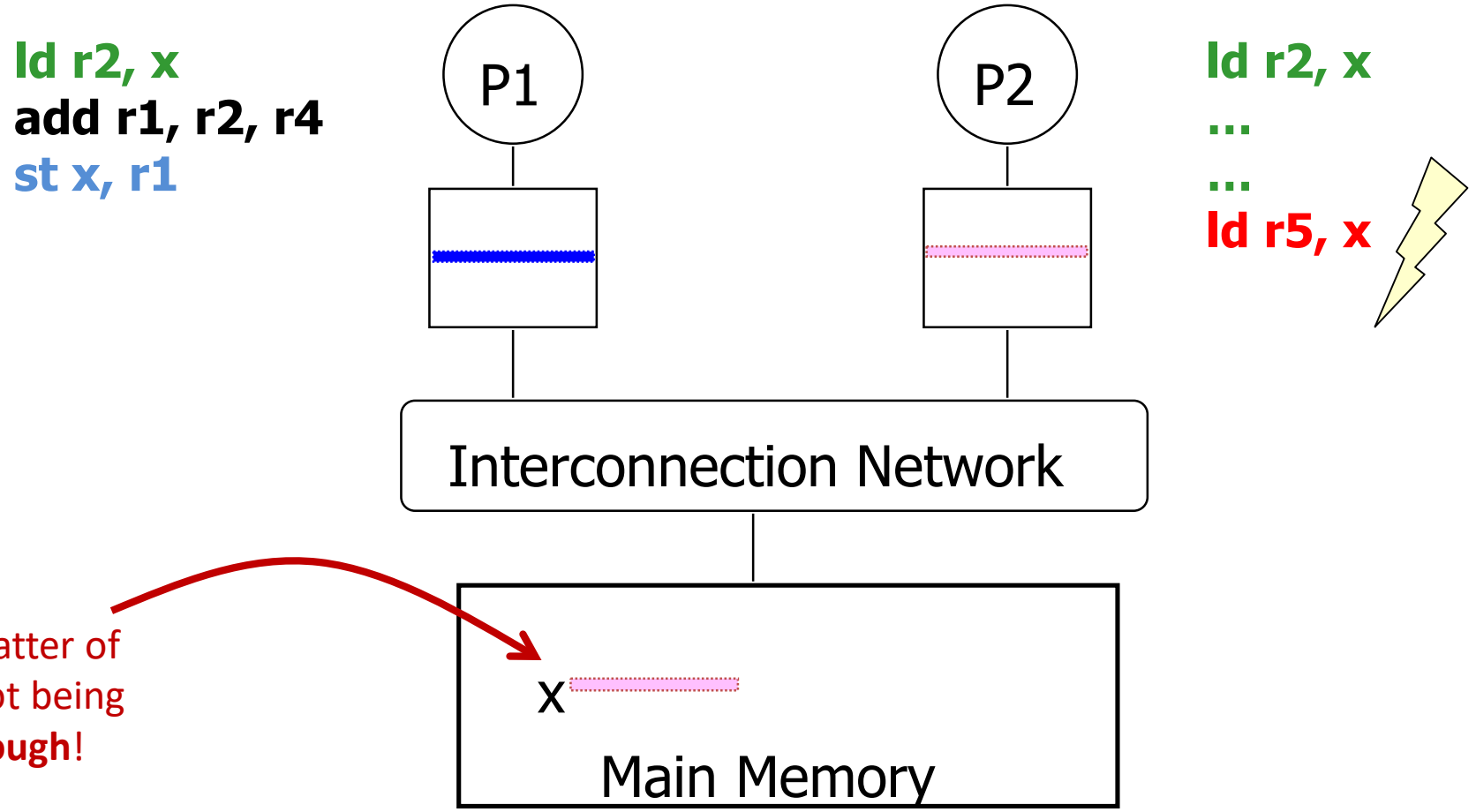
Cache Coherence Problem
Step 3

ld r2, x
add r1, r2, r4
st x, r1



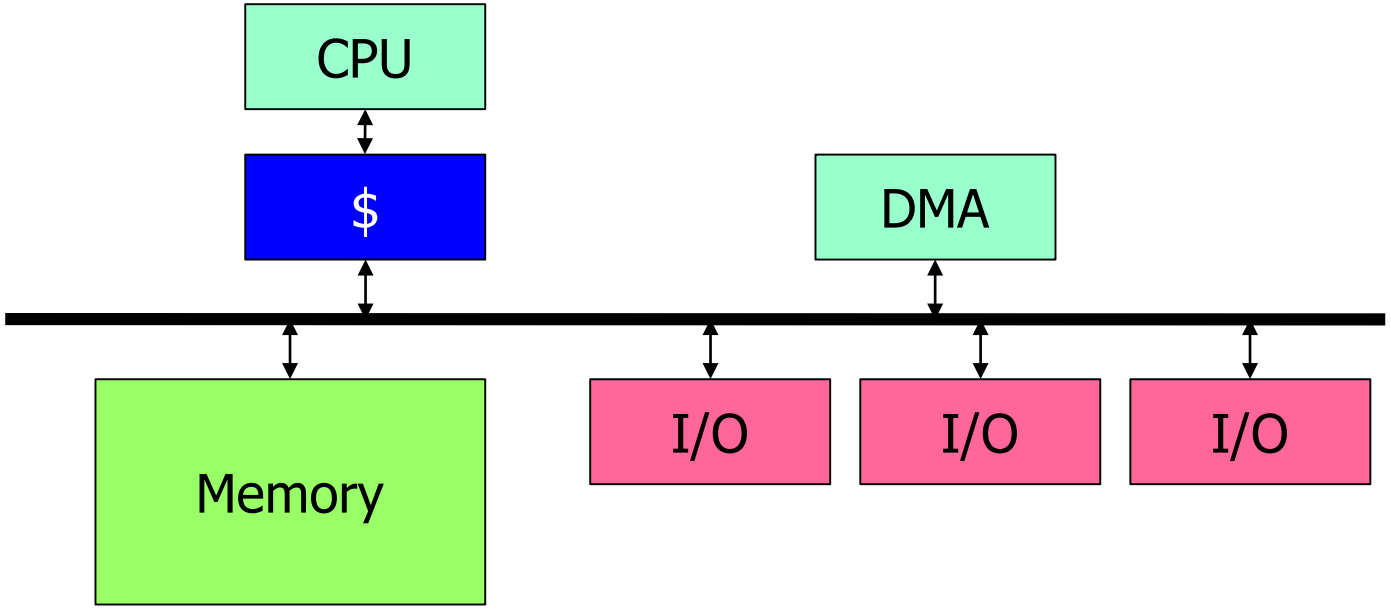
ld r2, x

Cache Coherence Problem
Step 4



Really New in Multiprocessors?

- Not really…

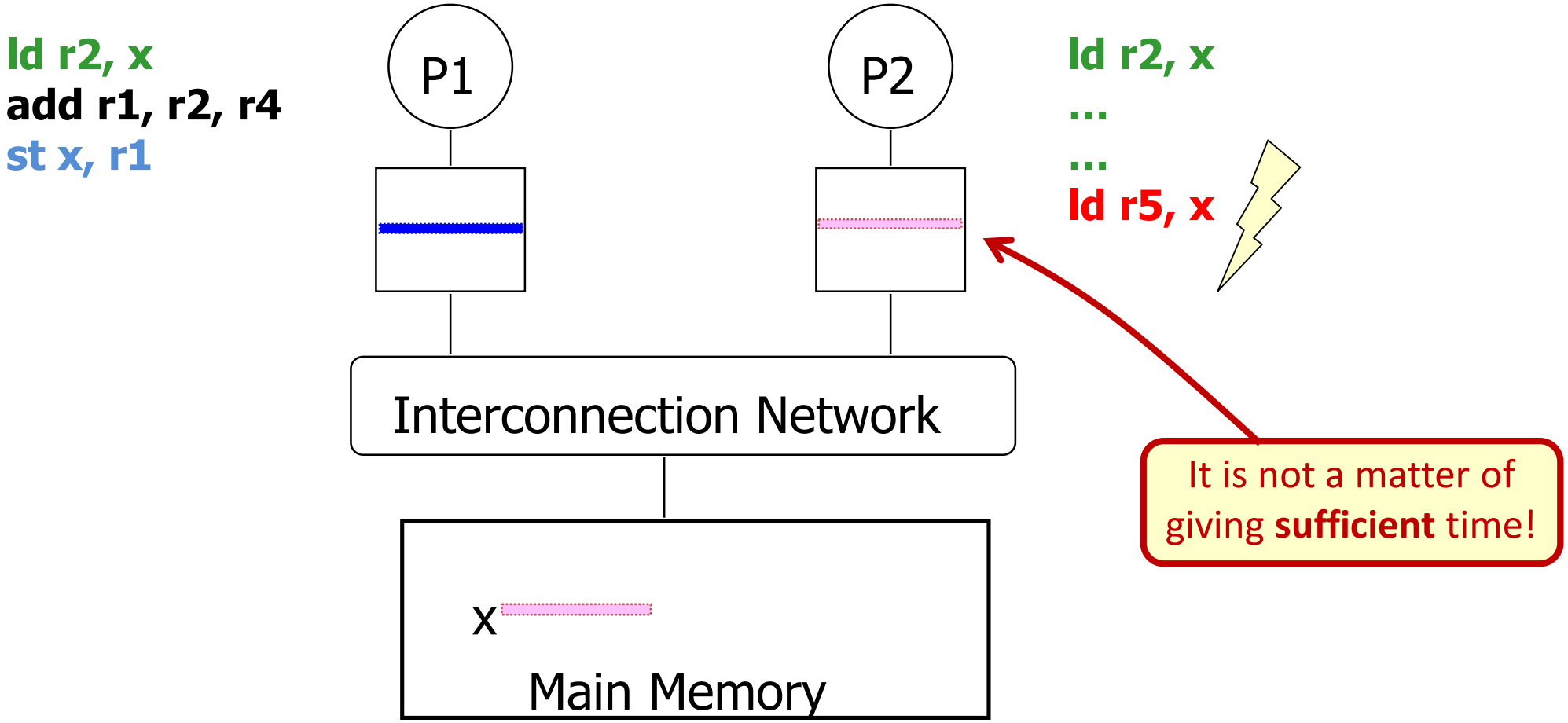


- But there we have simpler ad-hoc ways to handle this:
  - Flush the cache in software
  - Invalidate cache lines in software
  - Define **noncacheable areas of memory** and perform I/O there
  - …

Coherent Memory System

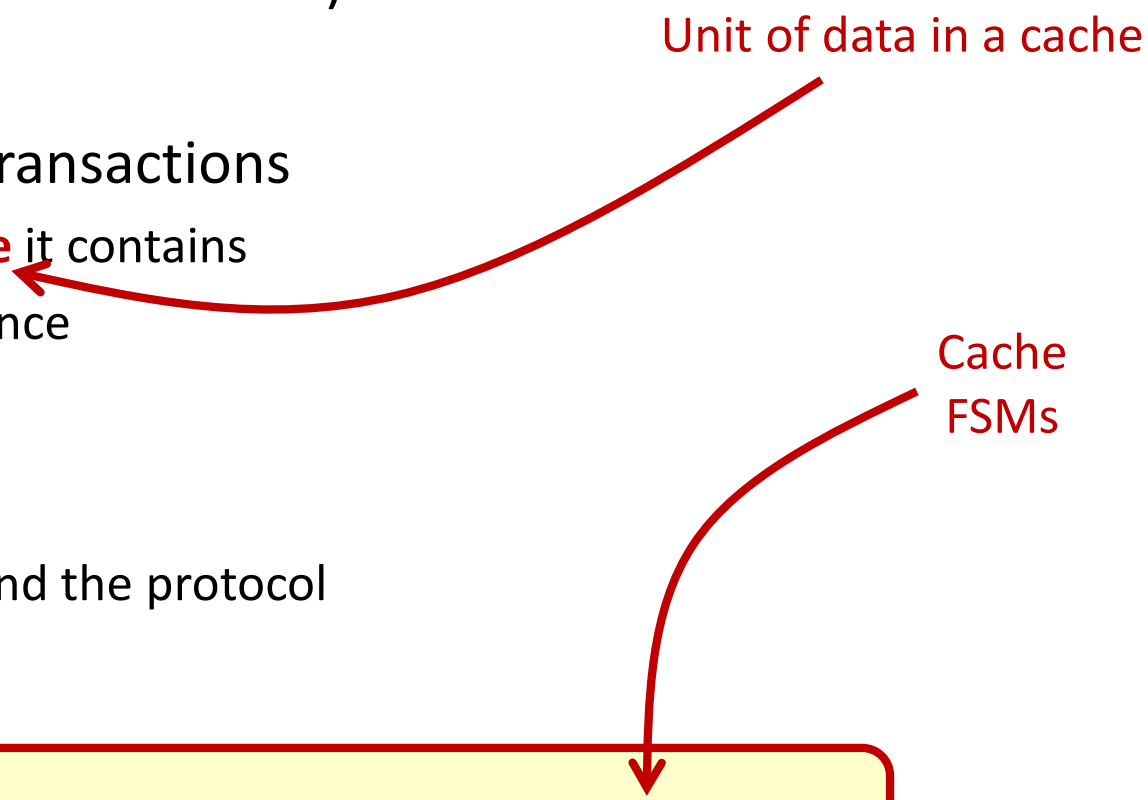
1. **Preservation of program order.** If P writes in X and then P reads in X, and in the meantime no other processor has written X, the value returned is the value previously written by P
2. **Coherent view.** If P1 writes X and P2 reads X, and in the meantime no other processor has written X, the value returned is the value previously written by P1, if the read and write are *sufficiently separated in time*
3. **Write Serialization.** If P1 writes X and P2 writes X, all processors see the writes in the same order

Cache Coherence Problem
Step 4



Snoopy Cache-Coherence Protocols

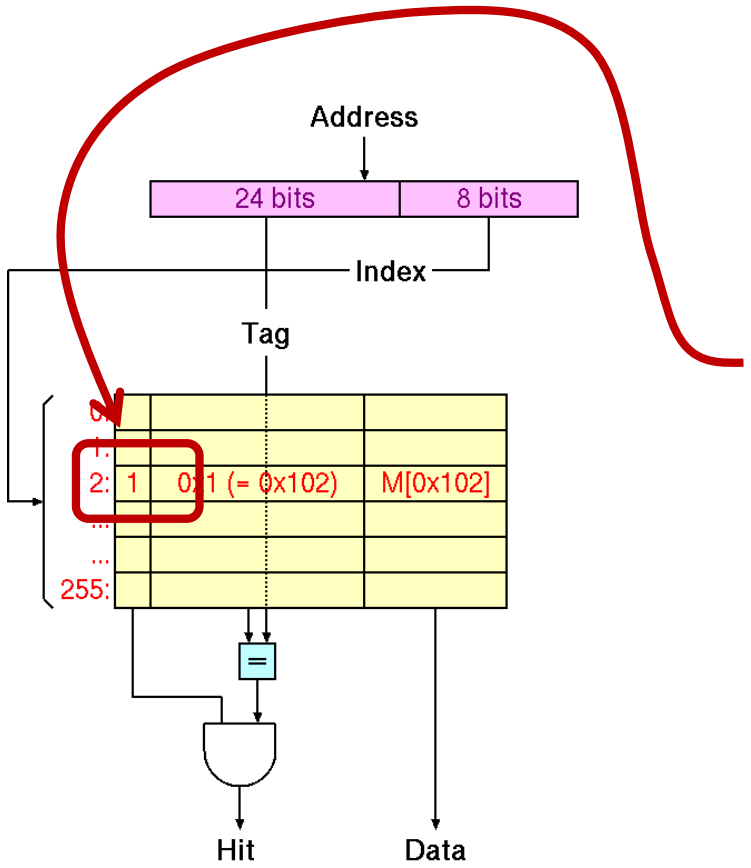
- Bus provides serialization point (more on this later)
- Each cache controller **snoops** all bus transactions
  - Relevant transactions if for a **line of cache** it contains
  - Take appropriate action to ensure coherence
    - Invalidate
    - Update
    - Supply value
  - Which one depends on state of the line and the protocol

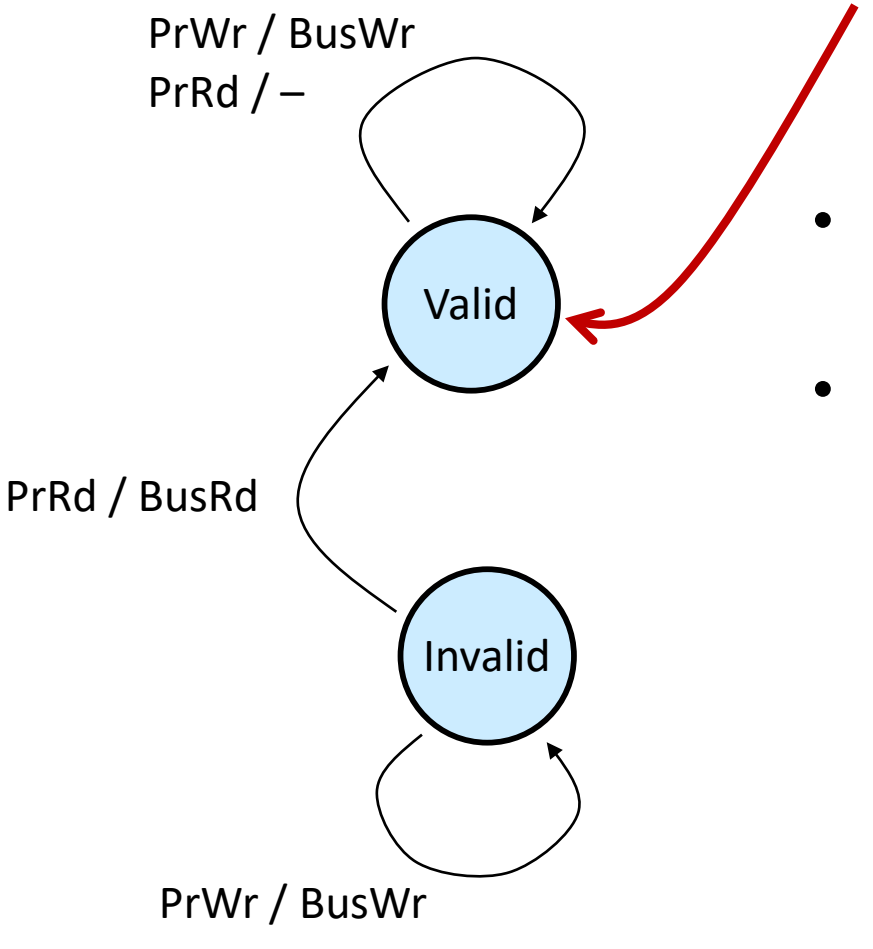


FSM of a Cache

- Write-through, write-no-allocate cache
- Actions:
  - PrRd
  - PrWr
  - BusRd
  - BusWr

The FSM of a Simple Cache Protocol



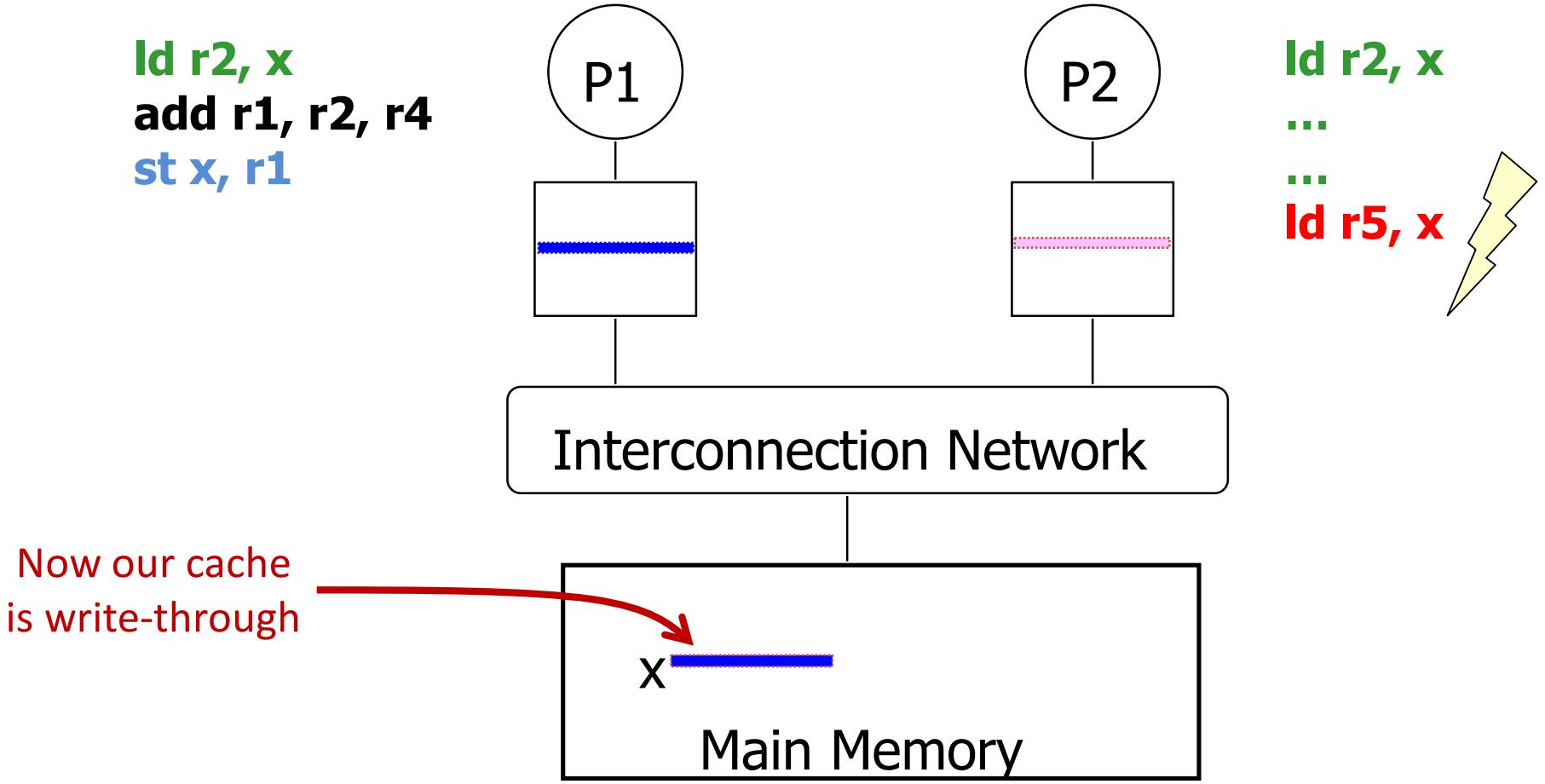


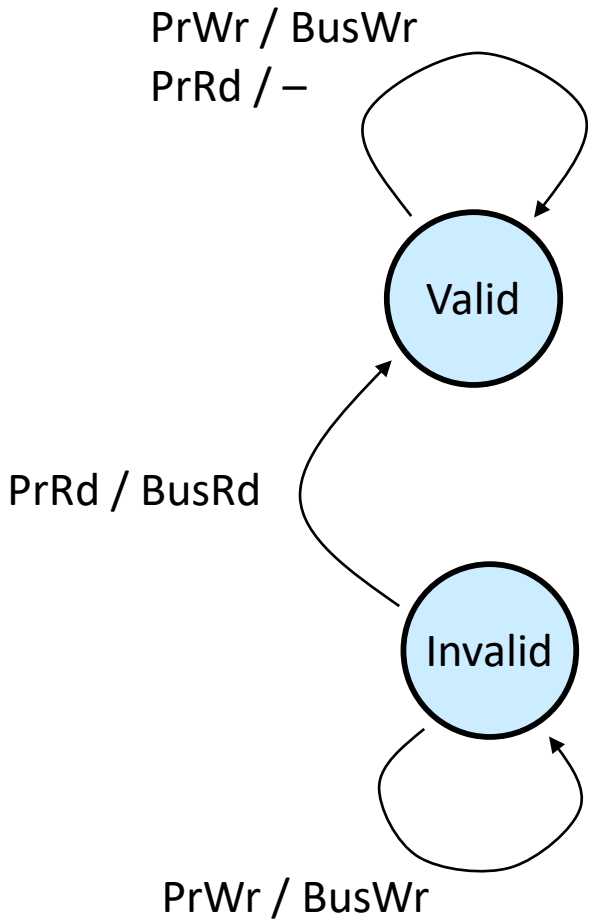
If evicted, the FSM is "destroyed"

- Write-through, write-no-allocate cache
- Actions:
  - PrRd
  - PrWr
  - BusRd
  - BusWr

The cache only *obeys* the processor

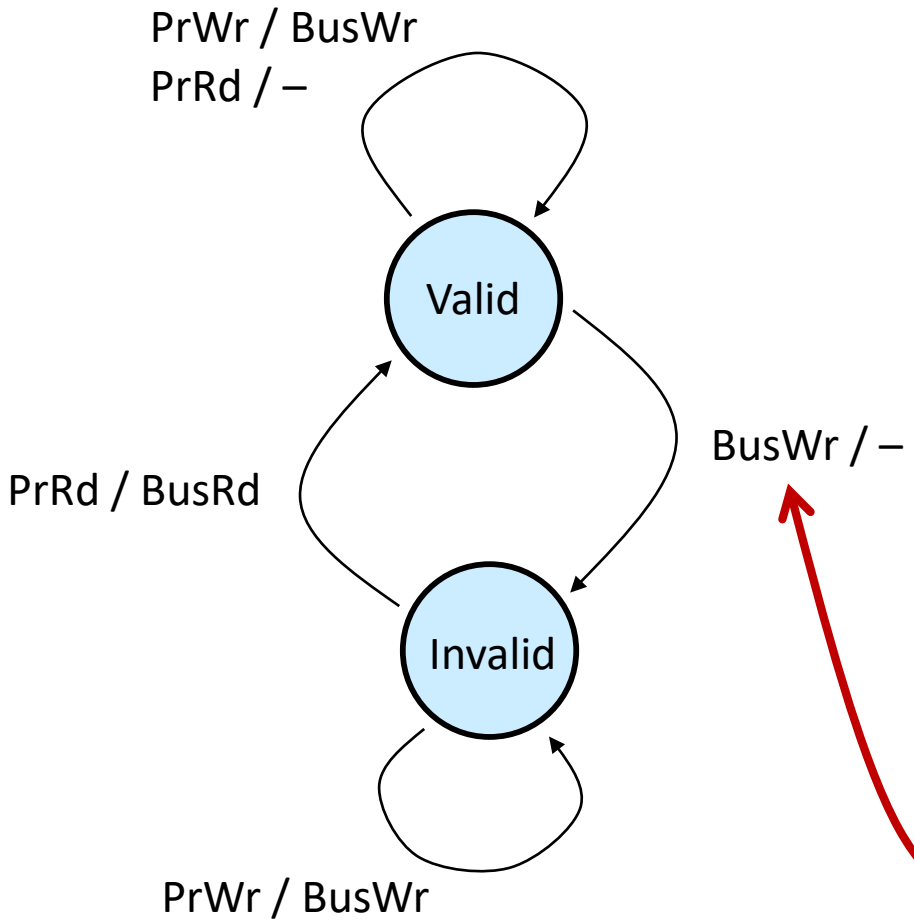
Cache Coherence Problem
Step 4





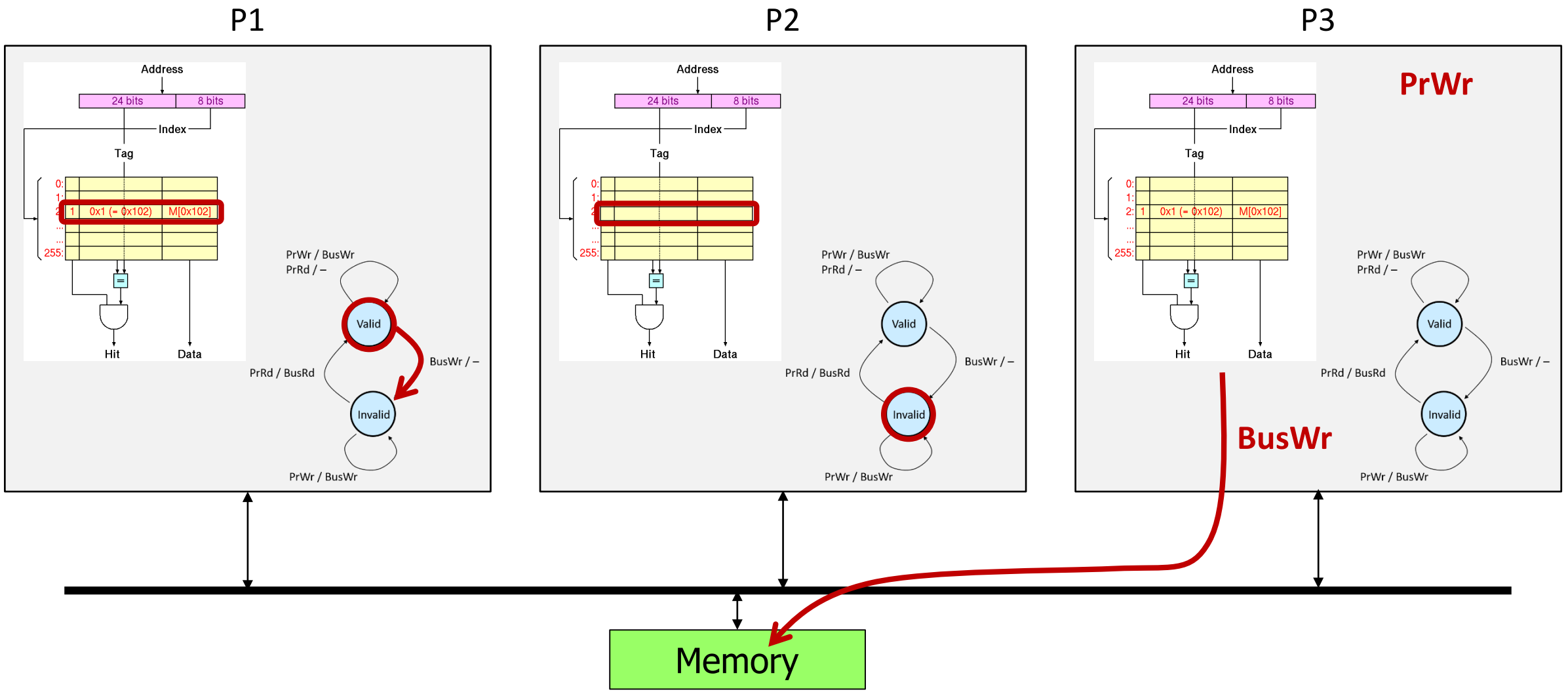
- Write-through, write-no-allocate cache
- Actions:
  - PrRd
  - PrWr
  - BusRd
  - BusWr

Simple Invalidate Snooping Protocol



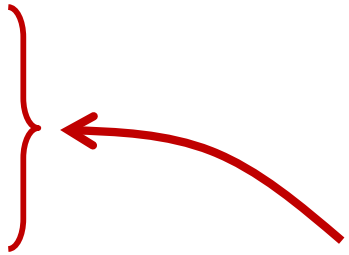
- Write-through, write-no-allocate cache
- Actions:
  - PrRd
  - PrWr
  - BusRd
  - BusWr

Cache FSMs Work Simultaneously



A 3-State Write-Back Invalidation Protocol (MSI)

- 2-State Protocol
  - Simple hardware and protocol
  - **Bandwidth (every write goes on bus!)**
- 3-State Protocol (**MSI**)
  - **Modified**
    - Only one cache has valid/latest copy
    - Memory is stale, that is the content is not up-to-date
  - **Shared**
    - One or more caches have valid copy
  - **Invalid**
- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)



Functional issues **solved**
(caches are **coherent**)
but performance may be **abysmal**!

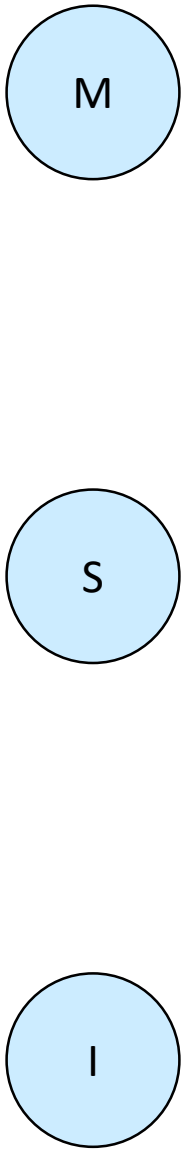
MSI: Processor and Bus Actions

- Processor
  - PrRd
  - PrWr
- Bus
  - **BusRd** (Bus Read) Read **without intent to modify**, data could come from memory or another cache
  - **BusRdX** (Bus Read Exclusive) Read **with intent to modify**, must invalidate all other caches copies
  - **BusWB** (Writeback) cache controller puts contents on bus and memory is updated

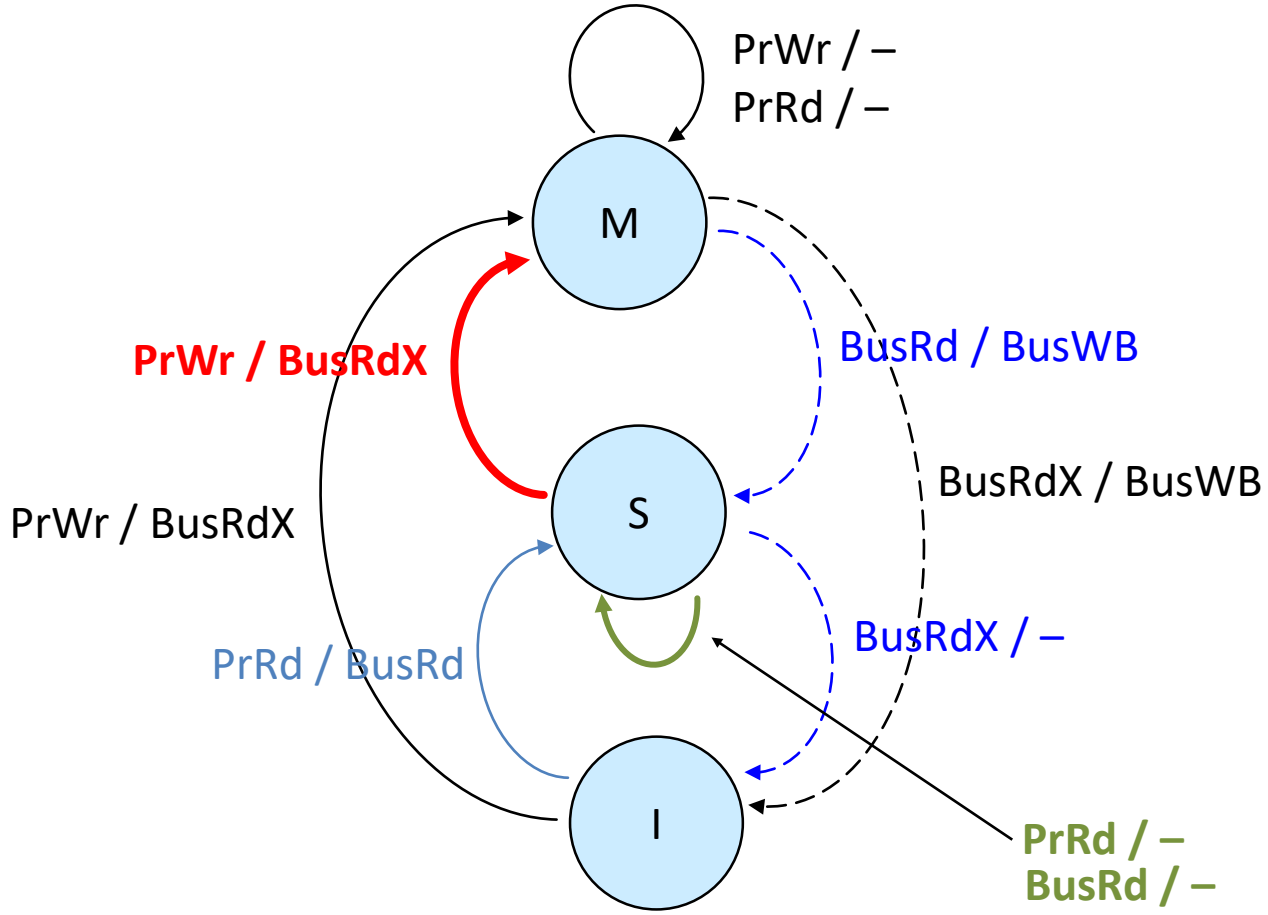
- Definition: **cache-to-cache transfer** occurs when another cache satisfies **BusRd** or **BusRdX** request with a **BusWB**

- Let's draw it!

A write to memory that can be simultaneously read also by other caches

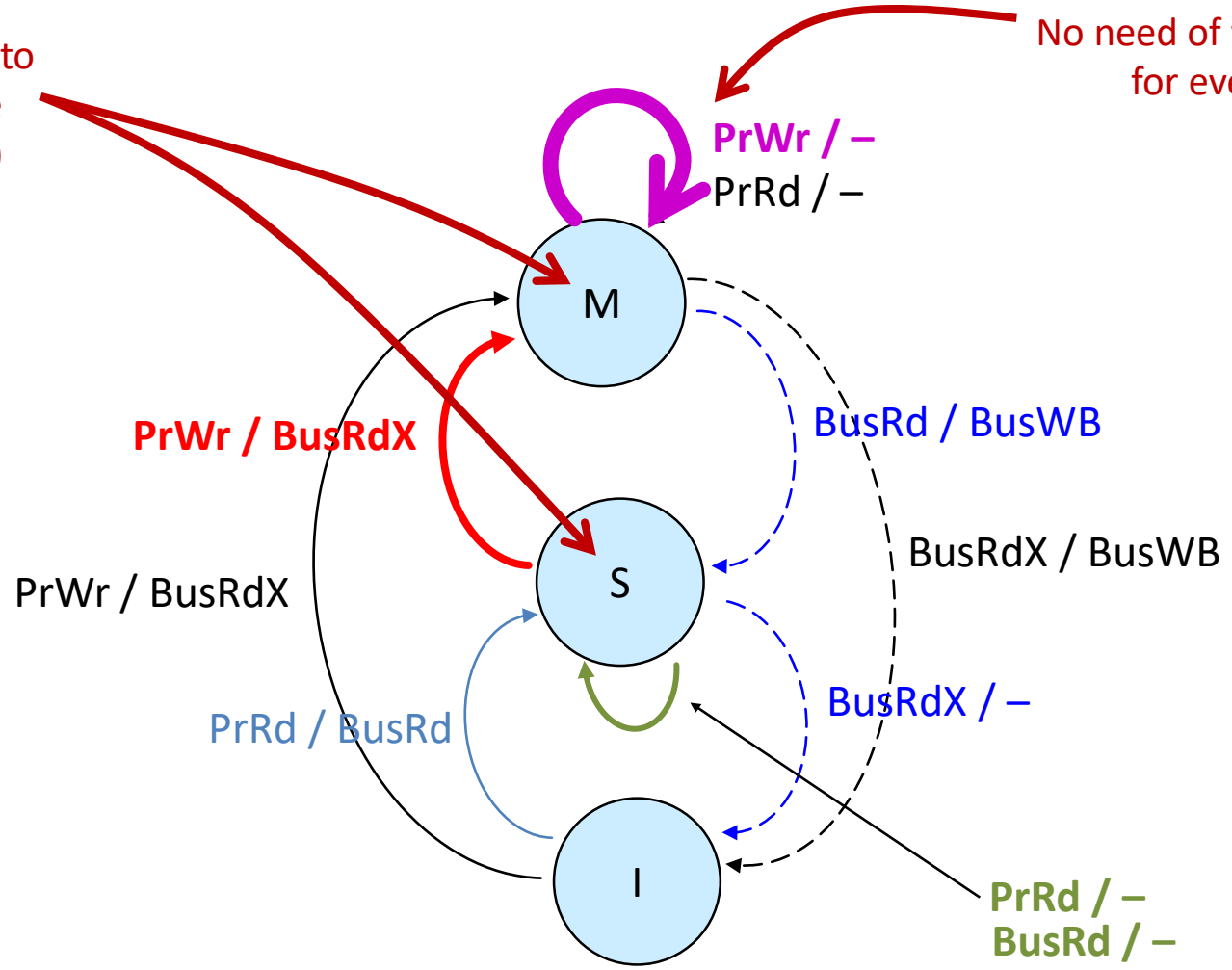

M
S
I

MSI State Diagram



What Has Changed?

We split the original **Valid** state into two states, one **Shared** and one **exclusive**/dirty (a.k.a. Modified)



Local variables cause **no memory traffic**!

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		– (= I)	– (= I)	– (= I)		
1	P1 reads x					
2	P3 reads x					
3	P3 writes x					
4	P1 reads x					
5	P2 reads x					

Example of MSI Transactions

	CPU Action	P1 state	P2 state	P3 state	Bus Action	Data from
0		– (= I)	– (= I)	– (= I)		
1	P1 reads x	S	– (= I)	– (= I)	BusRd	Memory
2	P3 reads x	S	– (= I)	S	BusRd	Memory
3	P3 writes x	I	– (= I)	M	BusRdX	Memory or not
4	P1 reads x	S	– (= I)	S	BusRd	P3's cache
5	P2 reads x	S	S	S	BusRd	Memory

4-State (MESI) Invalidation Protocol

- Often called the **Illinois** protocol
  - **M**odified (dirty)
  - **E**xclusive (unshared clean = only copy, not dirty)
  - **S**hared
  - **I**nvalid
- Requires **shared** signal to detect if other caches have a copy of block
- Cache Flush for cache-to-cache transfers
  - Only one can do it though
- What does state diagram look like?

Many Other Similar Protocols

- More states, different transactions
- Lots of research on how to minimize the coherence traffic
  (= better detect when it is not necessary)

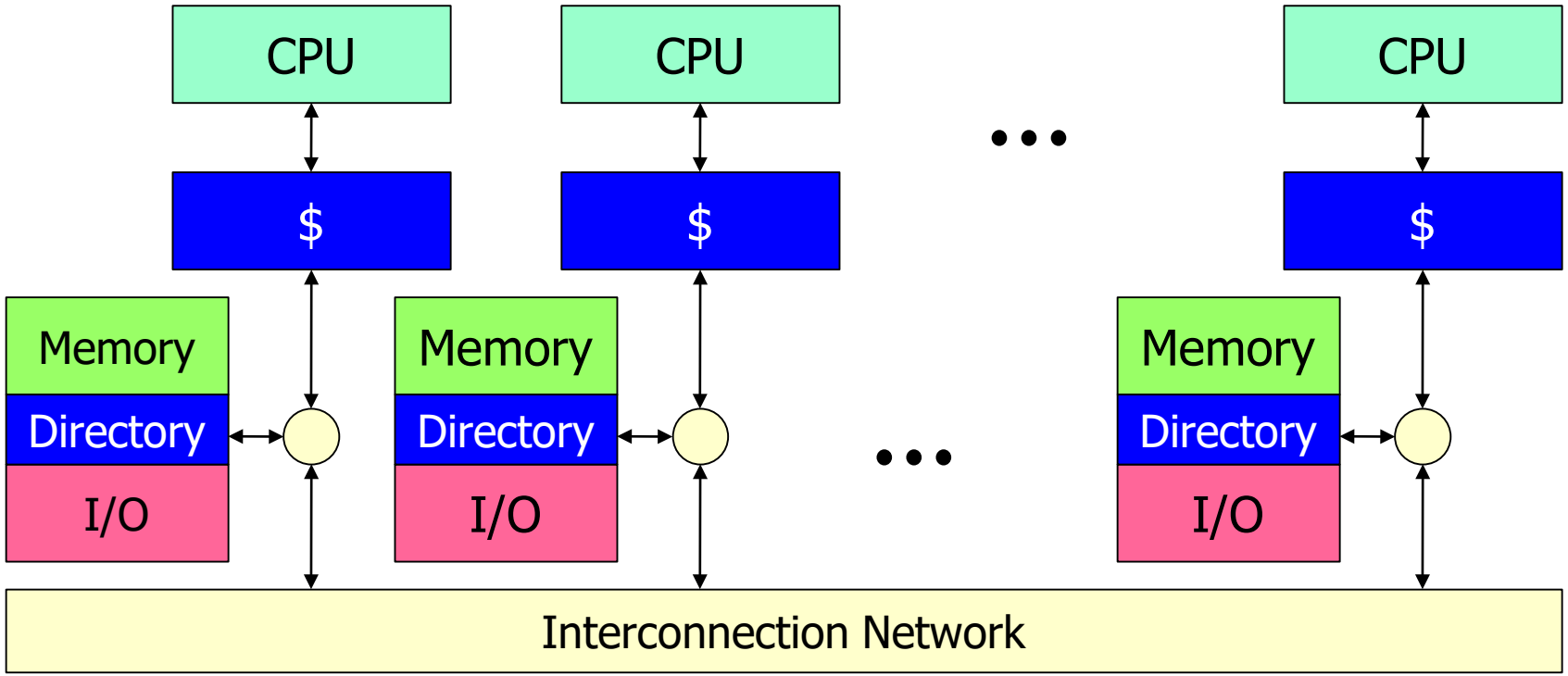
An important problem is scalability

beyond a few processors/cores

→ **Directory-Based Cache Coherence**

Directory-Based Cache Coherence

- Typically needed in Distributed Memory Architectures



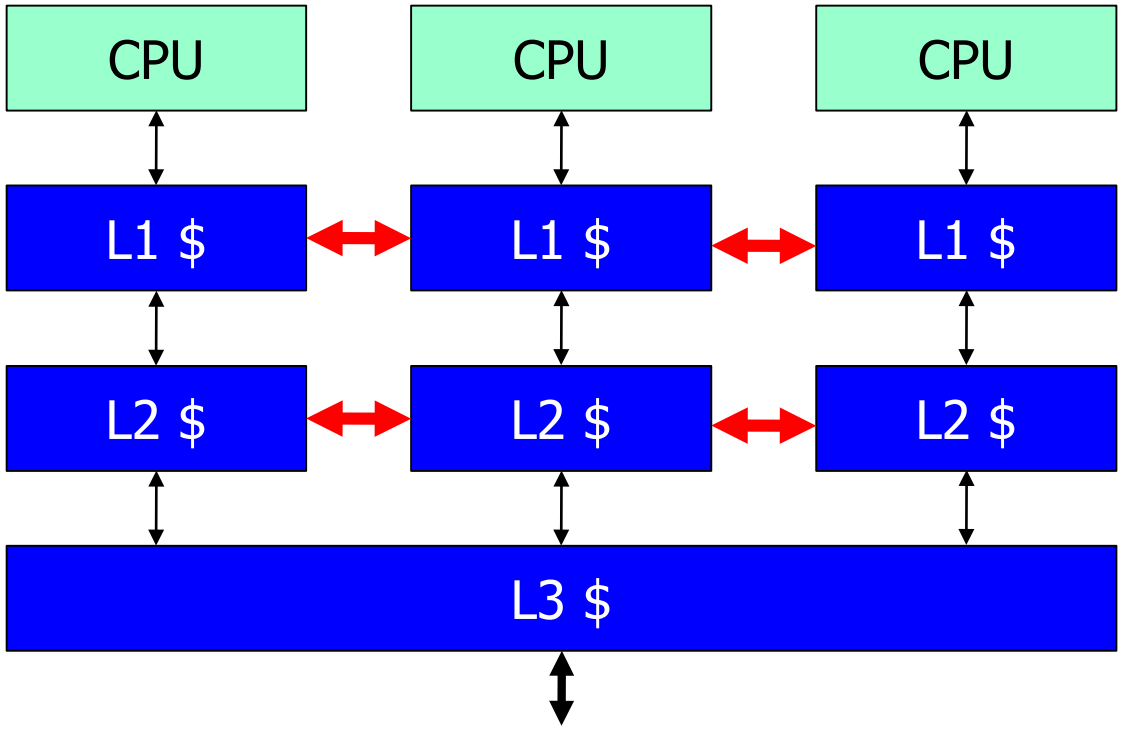
Directory keeps unique and “central” track
of existing cache copies and state

Snoopy vs. Directory-Based

- **Snoopy protocols** are distributed coherence protocols at the cache level
  - Scalability issues
  - Unnecessary coherence traffic
  - Fast
- **Directory-based protocols** are centralized protocols at the memory level
  - Scalable
  - Coherence traffic only as actually needed
  - Latency issues, due to centralization
  - Granularity issues, linked to latency issues

Multilevel Caches

- What happens if instead of **CPU → $ → Mem** we have **CPU → L1 $ → L2 $ → L3 $ → Mem**?
- One could just replicate snooping at all levels



Inclusion Property between Caches at Levels n-1 and n

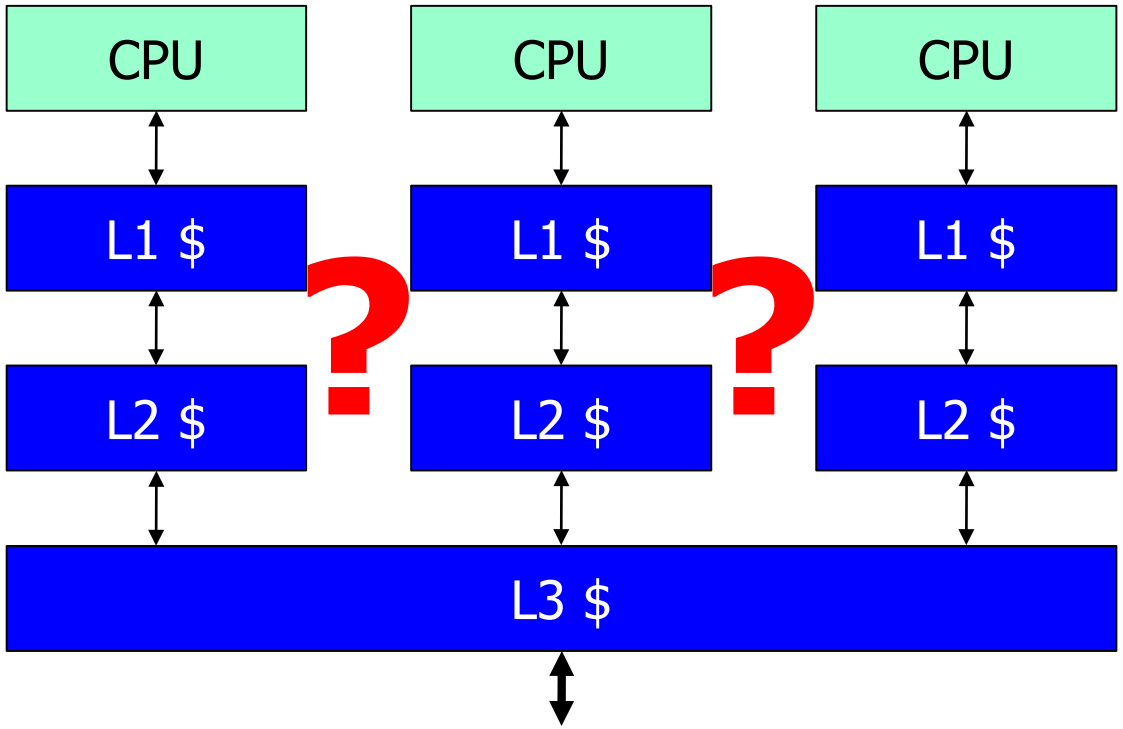
1. **Same content:** Content of L(n-1) cache is always a subset of the corresponding L(n) cache

   A bus transaction from an L2 cache is also always relevant for the L3 cache, hence **a snoop at L3 is sufficient**

2. **Same state:** If a cache line is marked as owned/modified in L(n-1) cache, then it should also be so marked in the corresponding L(n) cache

   If a bus transaction requests a cache line in owned/modified state in L3, the **L3 cache can determine this on its own without checking L2**

No Snooping Any More? Magic?

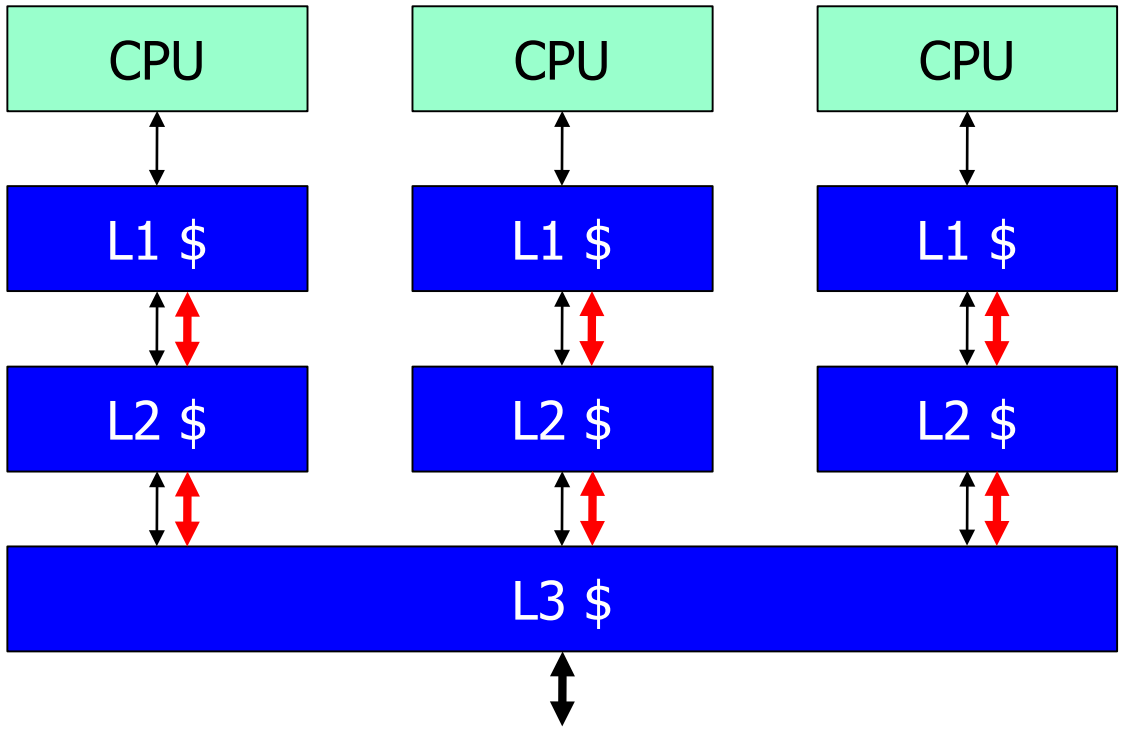


Is Inclusion Naturally Maintained?

- In some cases yes: in a Miss, it is, as L1 misses go to L2 and eventually the data will be in both
- In the general case **no**: for instance, L2 will eventually decide to evict a given line and that may be also in L1 → if nothing special is done, **inclusion will be violated**

Essentially, we will need to **propagate invalidations** and **evictions** up the hierarchy to keep, for instance, L1 informed of what happens in L2

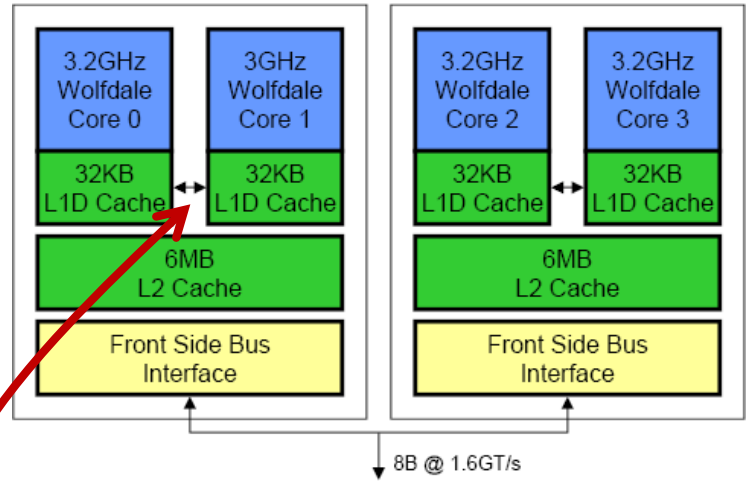
No Free Lunch! Only Different Messages



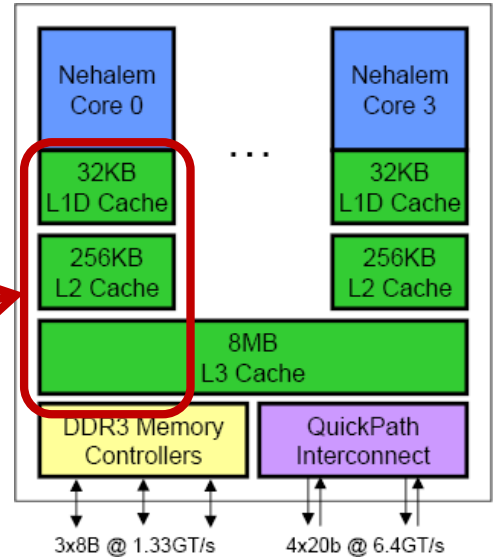
Which one is easier between snooping and maintaining inclusion is not trivially determined and well beyond this course…

Intel Core and Intel Nehalem

Intel Core



Intel Nehalem



Snooping

Inclusion

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Sections 6.3** (SIMD & MIMD)
  - **Sections 6.5** (Multicores)
  - **Sections 5.10** (Cache coherence)