信息科学与技术学院
School of Information Science and Technology

# CS 110
# Computer Architecture
## Hardware Multithreading

Instructors:
Siting Liu & Yuan Xiao
Course website: https://faculty.sist.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2026/5/28

# Administratives

- Project 2.2 released, ddl June 4th.
- Project 3 released, ddl June 11th.
- HW6 released, ddl TODAY!
- Lab 13 released, to be checked on Jun. 4th/8th/9th.
- Discussion May 29th on SIMD/profiling/etc., useful for project 3.
- Multiple assignments overlapped, start early!

# Parallelism Overview

*Software* | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search “CS110”
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

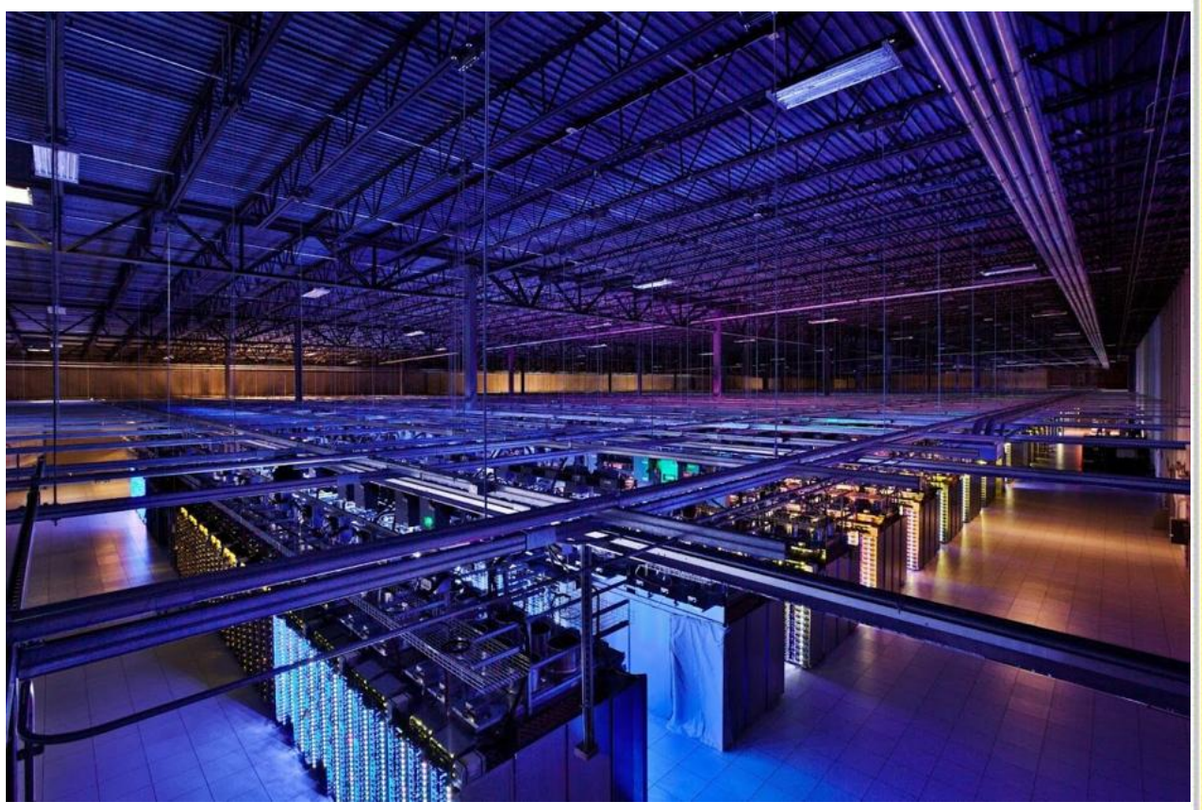
Warehouse Scale Computer

Smart Phone

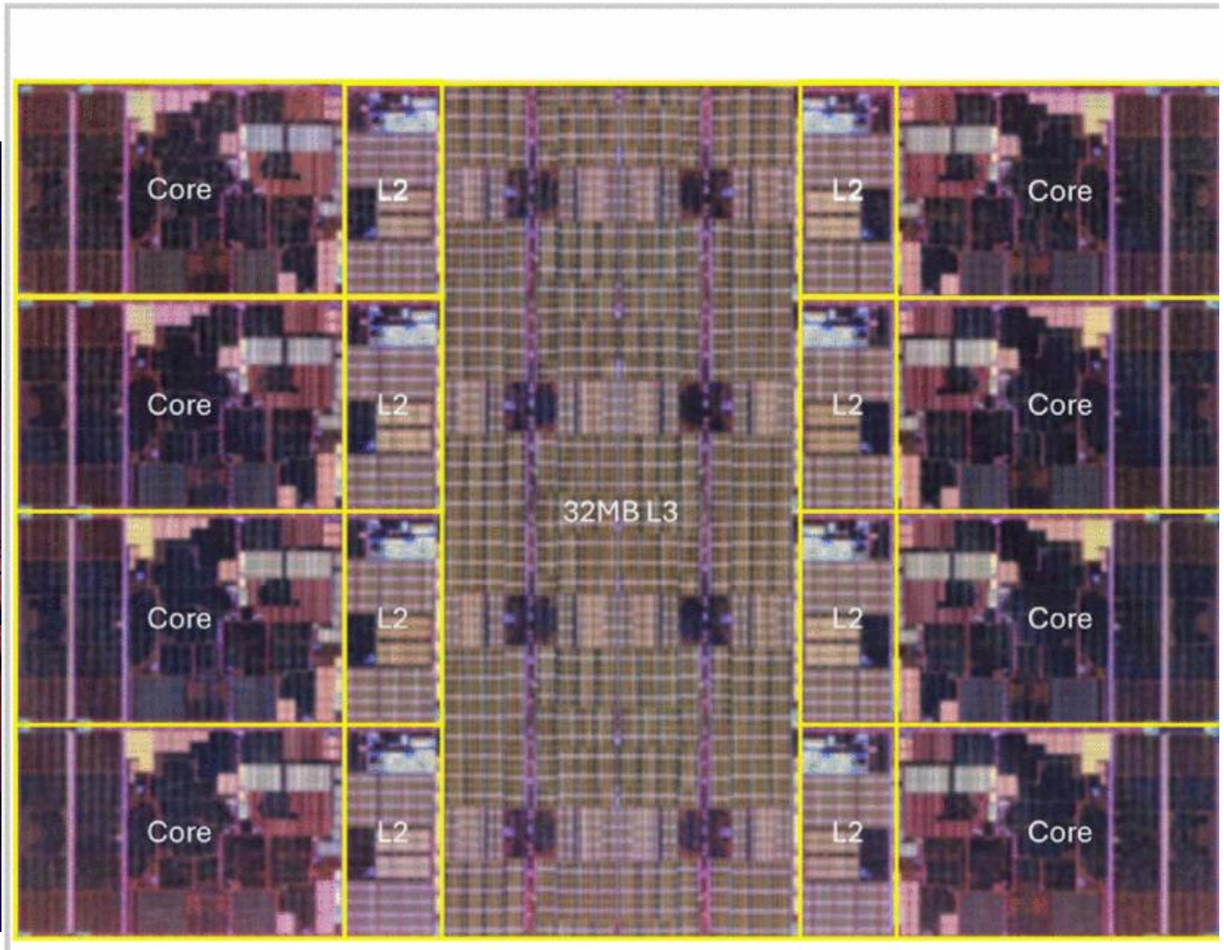



**Today’s Lecture**

# Parallel Computer Architecture



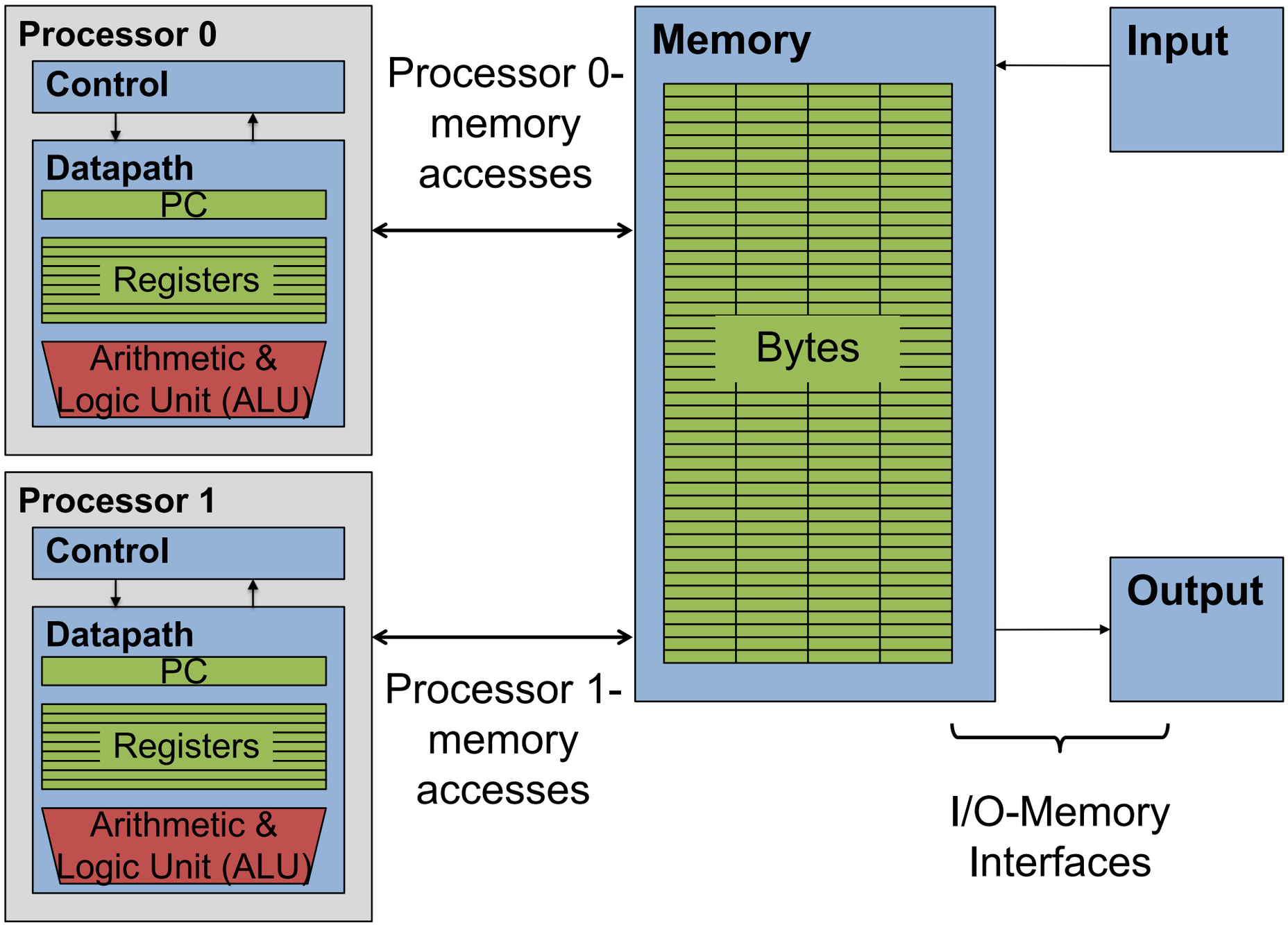
Warehouse-scale computers




AMD Zen 5 Die photo [ISSCC 2025]

# Multicore Processor

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.



# Multicore Processor

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.

  Execution Model:

  - Each processor executes an independent stream of instructions.
    - Also separate: high-level caches (e.g., L1 & L2 cache)
- All processors access the same shared memory.
  - Shared: DRAM and (perhaps) L3 cache
  - Communicate via shared memory by storing to/loading from common locations.

# Multicore Processor Use Cases

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.

Parallel-processing program

- Improve the runtime of a single program that has been specially crafted to run on a multiprocessor
- Example: Multithreaded program

Process-level parallelism (i.e., job-level parallelism, not covered in CS 110):

- Deliver high throughput for independent jobs
- Example: Your operating system and different programs

# Key Design Questions

- How many processors (or cores) should be supported in this multiprocessor?
  - Depends on the target workload!
  - Most systems: Multiple “best available single core within constraints”
  - Power-critical systems (e.g., phones): “some of the best available single cores” and “some of the most power-efficient single cores”

- How do different processors coordinate/communicate?
  - Shared variables in memory and load/store instructions
  - Coordinated access to shared data through synchronization primitives (e.g., locks) that restrict access to one processor at a time

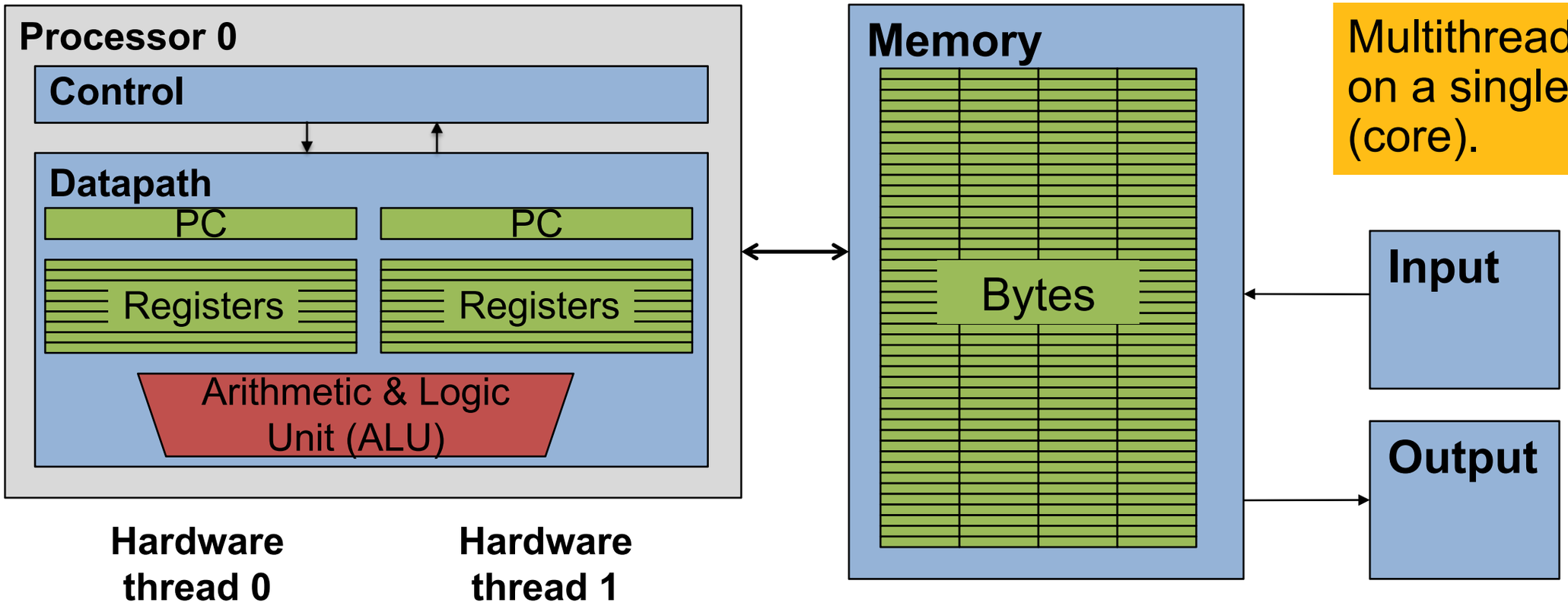
- How do different processors (cores) share data?
  - Via symmetric (shared-memory) multiprocessor (SMP)

# Hardware Multithreading

- Processor resources are expensive; should not be left idle
  - High memory latency cost on cache miss (~100 cycles)
  - Furthermore, the cost of thread context switch should be much less than cache miss latency!
- HW optimization is to have **redundant hardware** so that not every context switch needs to “save context”
  - Thanks to Moore’s Law, transistors are plenty
  - Add multiple PCs, registers to the same core

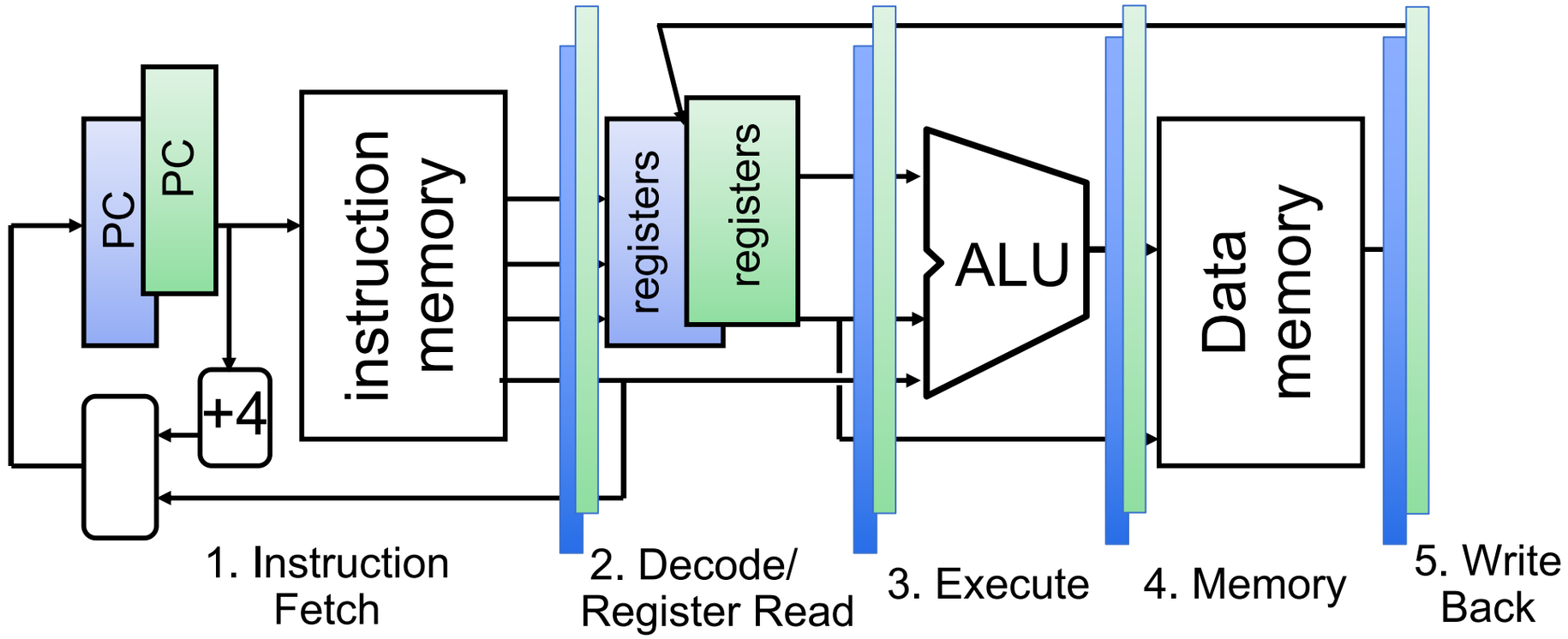
# Hardware Multithreading

- HW **multithreading** means having multiple thread ”active” in the same processor, e.g., by storing thread state (PC, registers).
  - Control logic decides which instruction to issue next
  - Can mix from different threads
- To software, this looks like **multiple processors** (hardware thread 0, hardware thread 1, etc).



# Hardware Multithreading

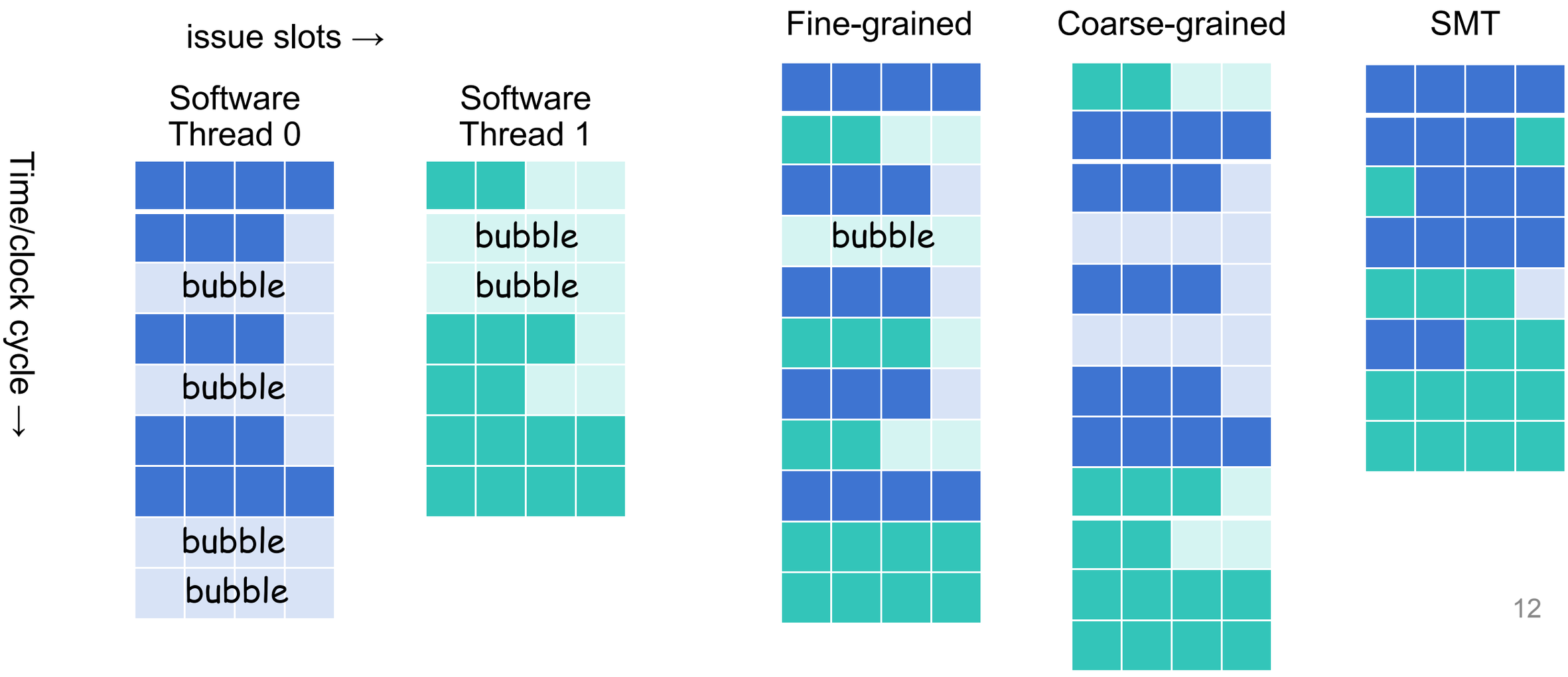
- Simplified implementation




- Use muxes to select which thread/state to use every clock cycle
- Run 2 independent processes
  - No Hazards: different registers; different control flow;
    Threads: race condition should be solved by software (e.g., lock sync. …)
- Speedup?
  - No obvious speedup; Complex pipeline: switch to another hardware thread in case of unavailable resources (e.g. wait for memory)

# Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT) lowers the cost of multithreading by leveraging multiple issue, dynamically scheduled microarchitecture.
  - Superscalar architecture
  - Also called “hyperthreading” in Intel processors
  - Downside: excessive power consumption



# SMT vs. Multicore

- Modern machines do both:
  - Multiple cores, with multiple threads per core.
- Simultaneous Multithreading (SMT):
  - ~1% more hardware, ~1.10X better performance
  - Shared: integer ALU, floating point units, all caches, memory controller
  - Better utilization from reducing latency of:
    - OS context switches
    - Major stalls like instruction cache misses
- Multiple cores:
  - Duplicate processors entirely
  - ~2X more hardware, ~2X better performance
  - Share: outer caches (e.g., L3 cache), memory controller
  - Better utilization from executing instructions on different processors in parallel

# Summary and Comparison

| | Multi-issue | SIMD | SMT | Multi-core |
|---|---|---|---|---|
| Parallelism | ILP | DLP | TLP | TLP |
| Datapath | Shared IF/PC/register file (RF), multiple datapaths for different types of instructions | Multiple processing elements/ ALUs, indepedent vector RF | Shared ALU, multiple PC/register files | Indepedent multiple full datapaths |
| Core | Within single core | Within single core | Within single core, but looks like multi-core (multiple logical cores) | Multiple (phsical) cores or full datapaths (IF, PC, RF, ALU, etc.) |
| Main issues | Combined with pipeline, may lead to data hazards | Data should be indepedent with the same operations | Requires Synchronization | Requires Synchronization |

# Cache: Key Design Questions

How many processors (or cores) should be supported in this multiprocessor?

- Depends on the target workload!
- Most systems: Multiple “best available single core within constraints”
- Power-critical systems (e.g., phones): “some of the best available single cores” and “some of the most power efficient single cores”

How do different processors coordinate/communicate?

- Shared variables in memory and load/store instructions
- Coordinated access to shared data through synchronization primitives (e.g., locks) that restrict access to one processor at a time

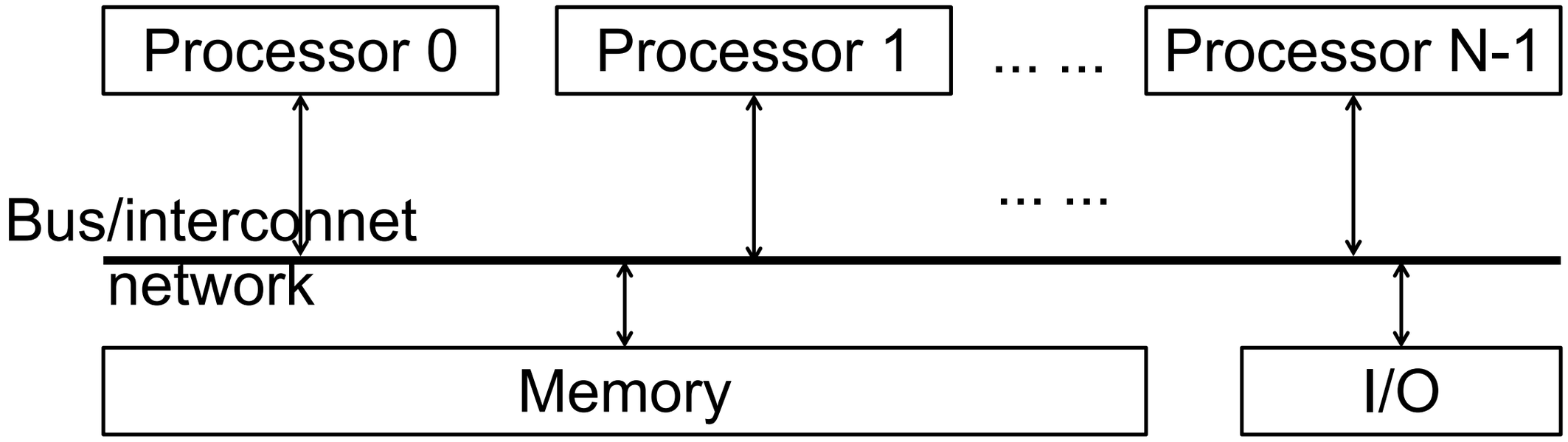
How do different processors (cores) share data?

- Via shared-memory

Effectively all multicore computers today use SMP.

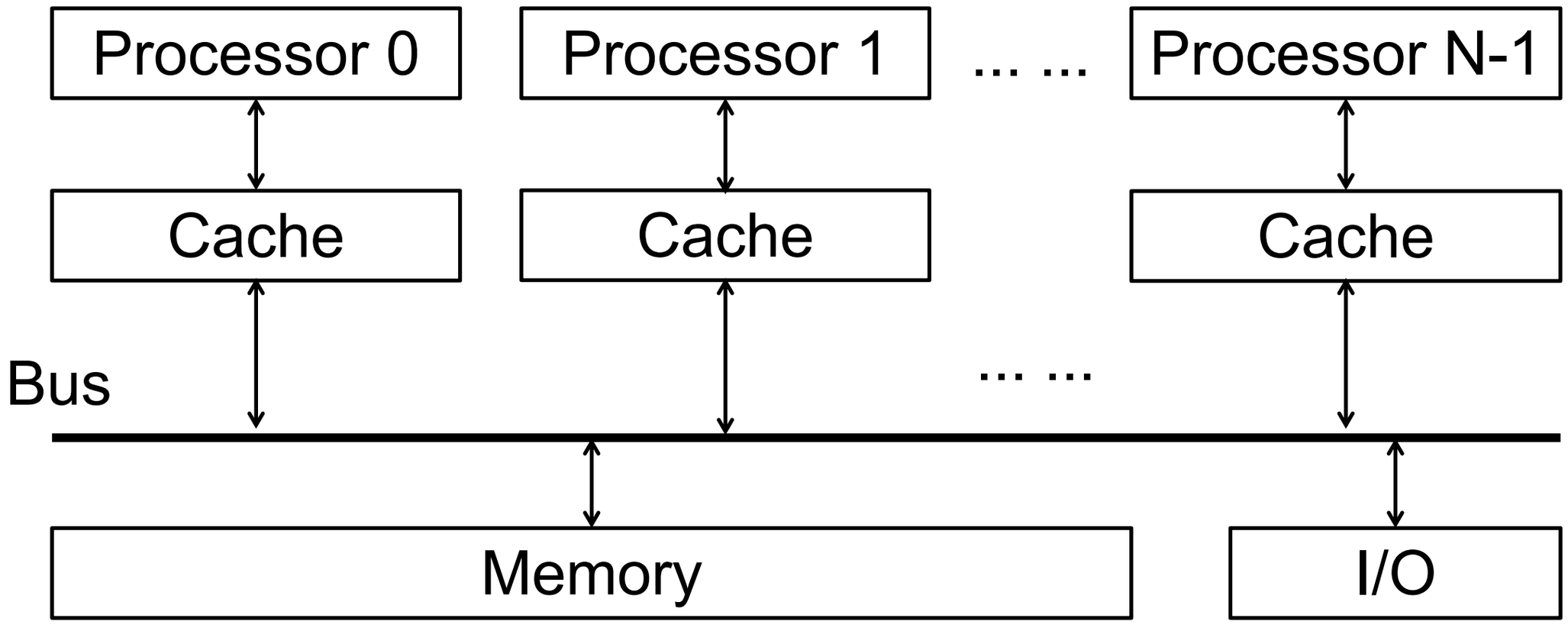
# Shared-memory Multiprocessor

- A **shared-memory multiprocessor** offers multiple cores/CPUs a single, shared, coherent memory.
  - Sometimes called symmetric multiprocessor (SMP)
  - Should be called shared-address multiprocessor, because all processors share single physical address space (more later, VM)

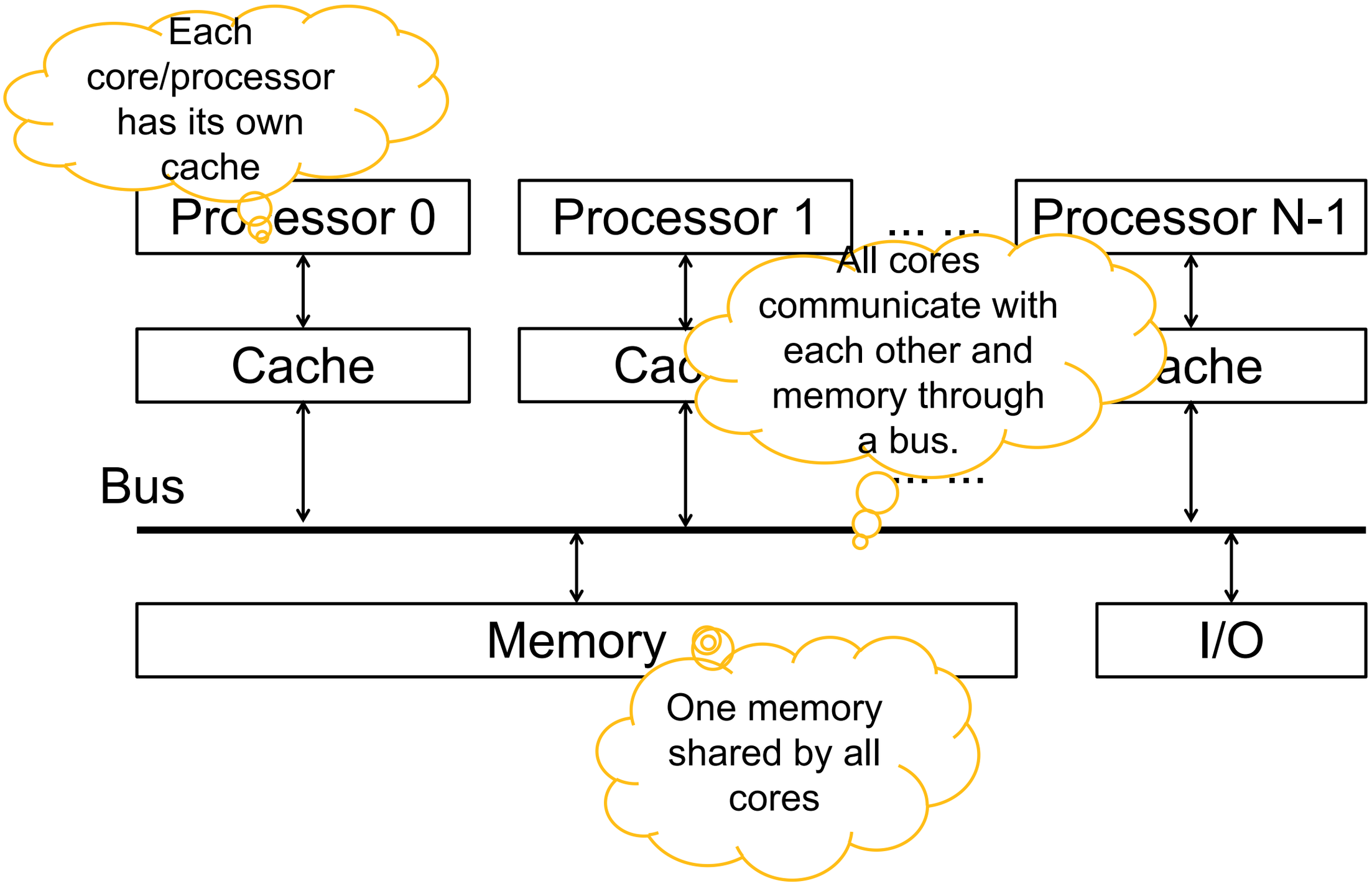


# Multiprocessor Cache

- Memory is a performance bottleneck even with one processor.
- Use private **caches** to reduce bandwidth demands on main memory!
- Only cache misses have to access the **shared** common memory

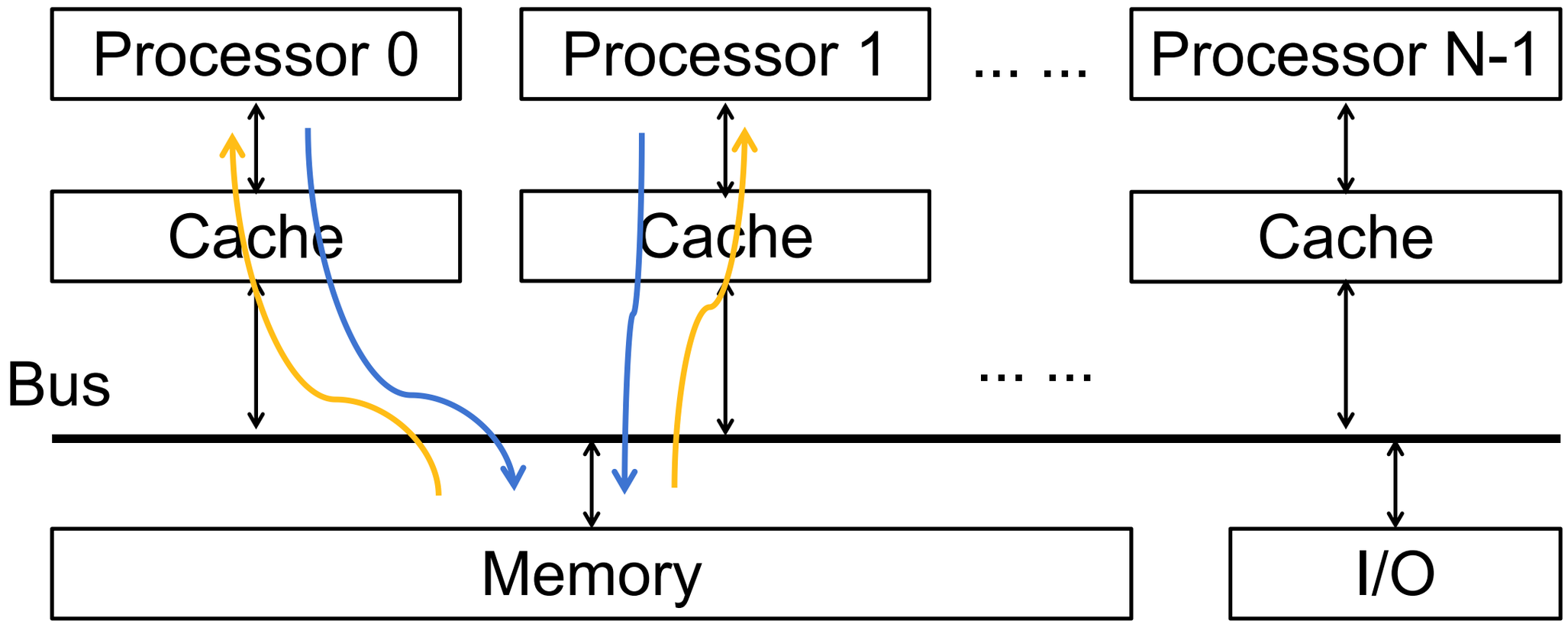


# Multiprocessor Cache (1/3)



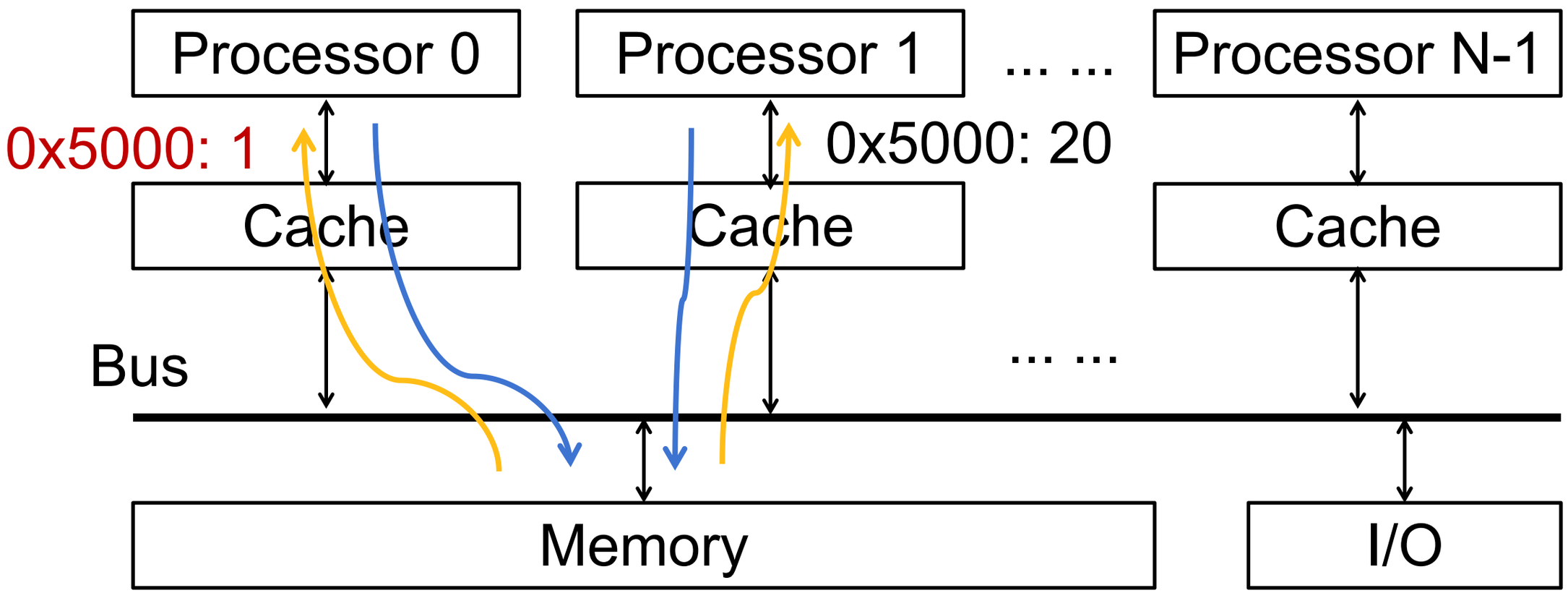
# Multiprocessor Cache (2/3)

- Consider the following scenario
  - Assume value “20” initially @ Mem[0x5000]):
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];



# Multiprocessor Cache (3/3)

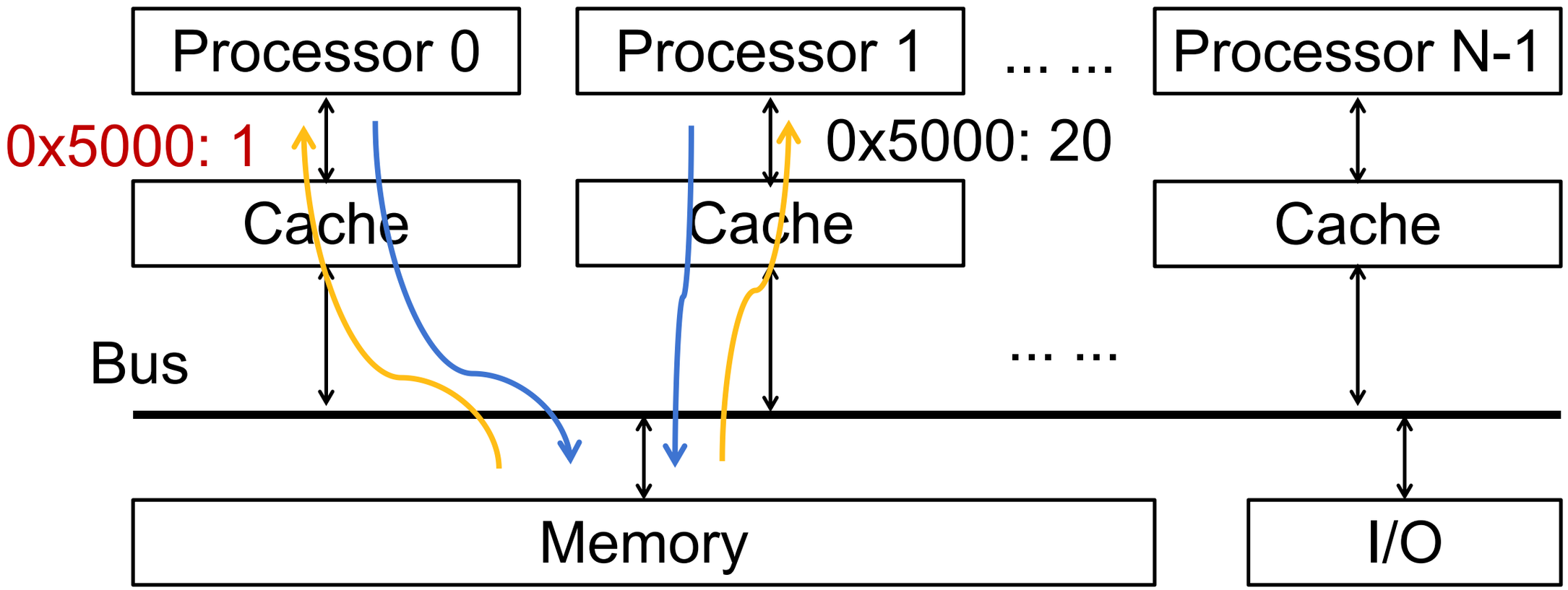
- Consider the following scenario
  - Assume value “20” initially @ Mem[0x5000]):
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];
  - Processor 0 write ‘1’ to Mem[0x5000];



# Cache (In)coherence

- Consider the following scenario
  - Assume value “20” initially @ Mem[0x
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];
  - Processor 0 write ‘1’ to Mem[0x5000];

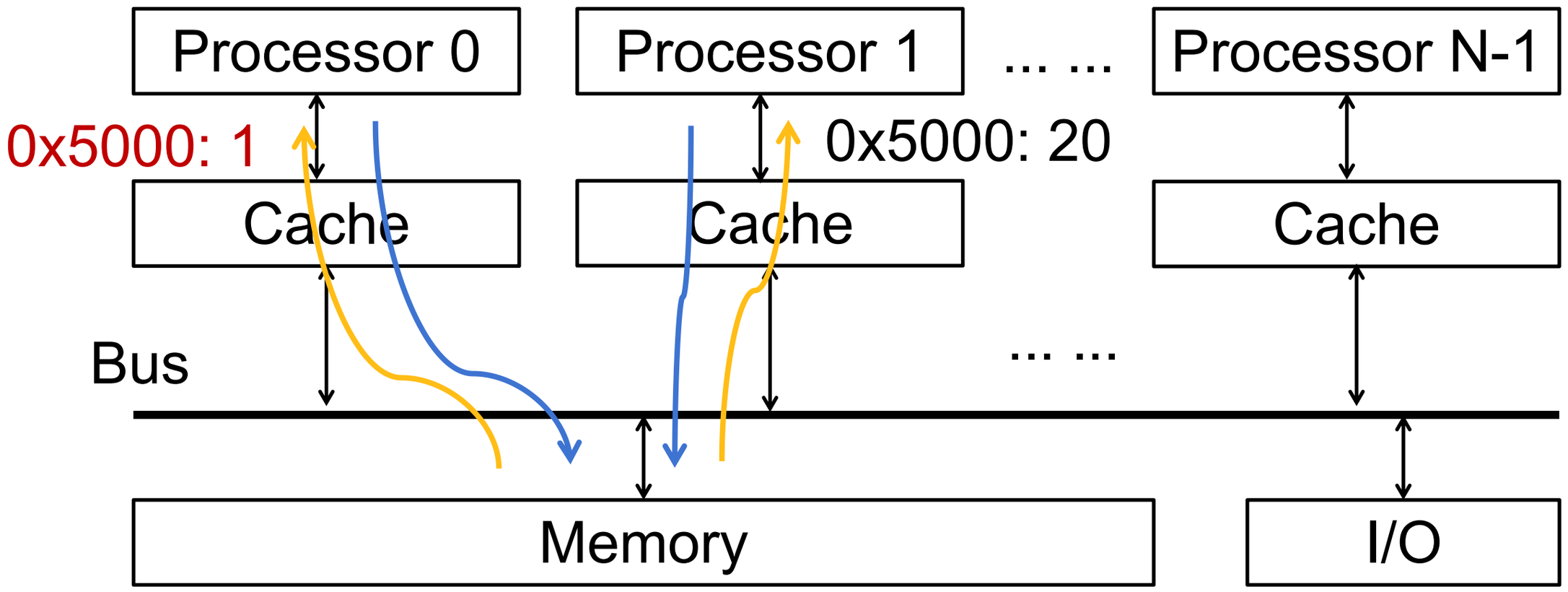
New cache miss type: **coherence miss** (a.k.a. communication miss), caused by writes to shared data made by other processors.




- For some parallel programs, coherence misses can dominate total misses;
- The 4th “C” of cache misses

# Cache (In)coherence

- The processor 0 write invalidates other copies in other processors' caches.



# Cache Coherence and Snooping

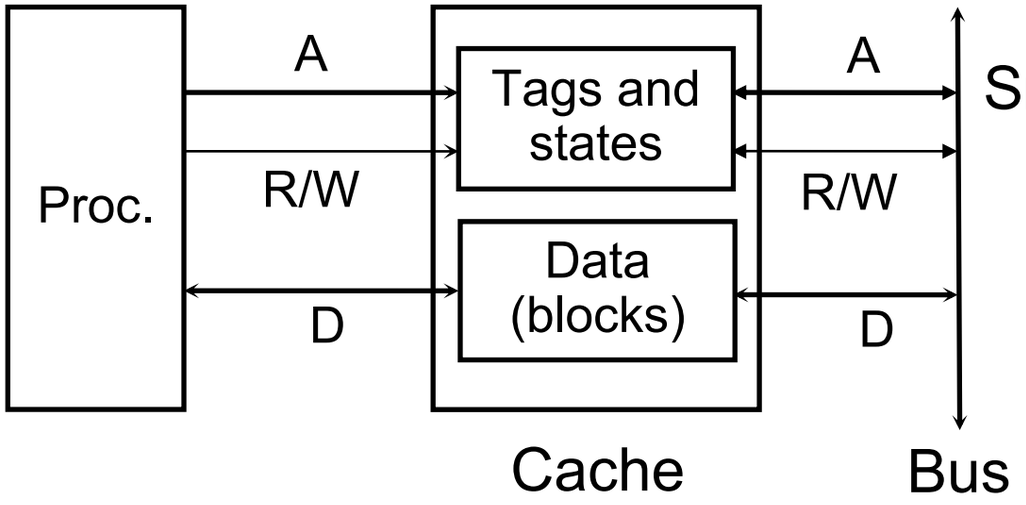
- **Coherent**: any read of a data item returns the most recently written value of that data item
- Because there is shared memory, a computer architect must design the system to keep cache values **coherent**.
- Idea: When any processor has cache miss or writes, use the bus to **notify other processors**.
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies.
- One **cache coherence protocol**: Each cache controller "**snoops**" for write transactions on the common bus
  - Bus is a broadcast medium
  - On any block request to the bus, check if own cache has a copy
    - If exists, then invalidate own cache's copy

# How to Keep Cache Coherent?

- ***Cache coherent protocol***, things to think about:
  - How do we communicate when one processor changes the state of shared data?
  - Does every processor action cause data to change state?
  - Who should be responsible for providing the updated data?
  - What happens to memory while all of this is happening?

# Snooping/Snoopy Protocols

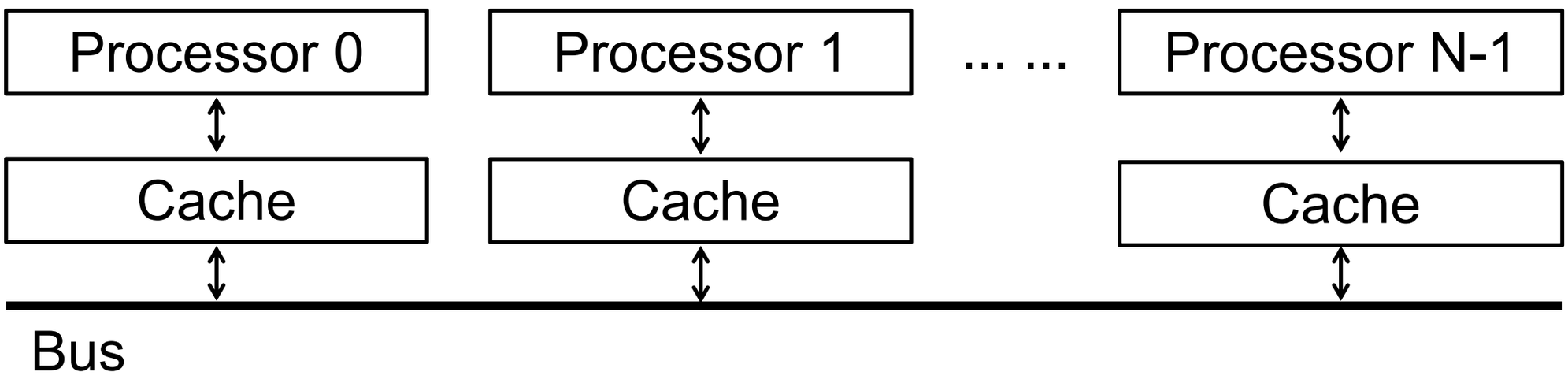
- Snoopy Cache, *[Goodman 1983]*
  - Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
  - Snoopy cache tags are dual-ported




Courtesy: Baidu Baike

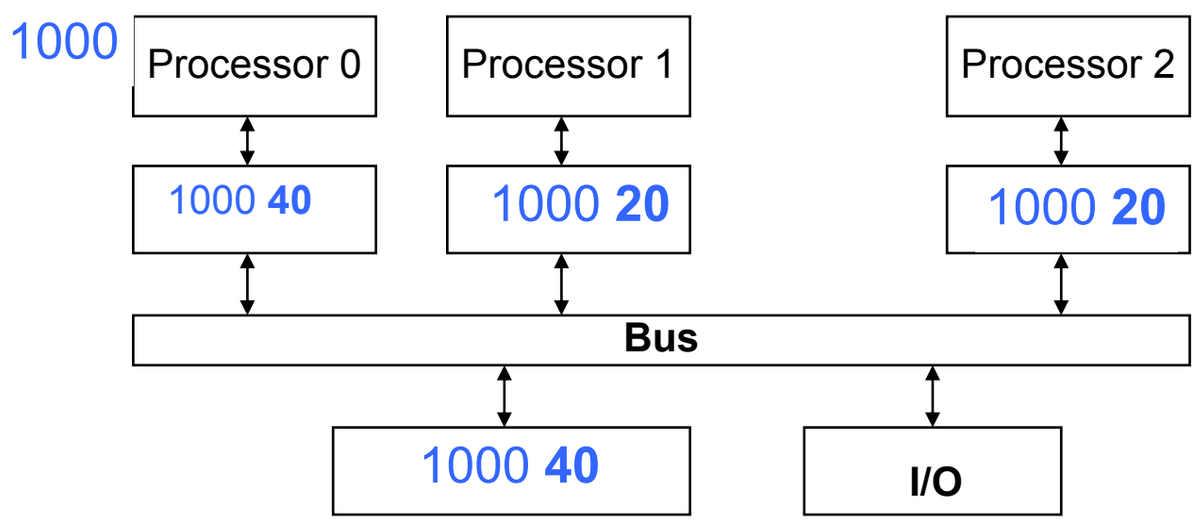
# Optimized Snoop with WAW

- Use valid bit to “unload” cache lines (in processors 1~N-1);
- If write-back cache, processor 0 holds a dirty bit;
  - Dirty bit tells me: “I am the only one using this cache line”! => no need to announce on bus again for a second write by processor 0!



# Snooping Protocols

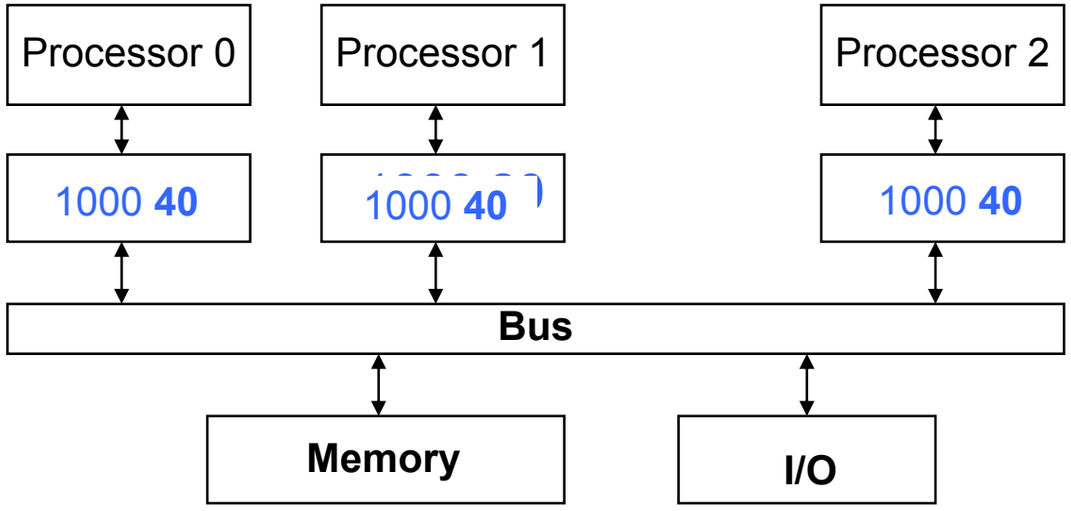
- *Write invalidate*
  - Processor *k* wanting to write to an address, grabs a bus cycle and sends a 'write invalidate' message
  - All the other snooping caches invalidate their copy of appropriate cache line
  - Processor *k* writes to its cached copy (assume for now that it also writes through to memory)
  - Any shared read in the other processors will now miss in cache and re-fetch new data.




Processor 0 write invalidates other copies

# Snooping Protocols

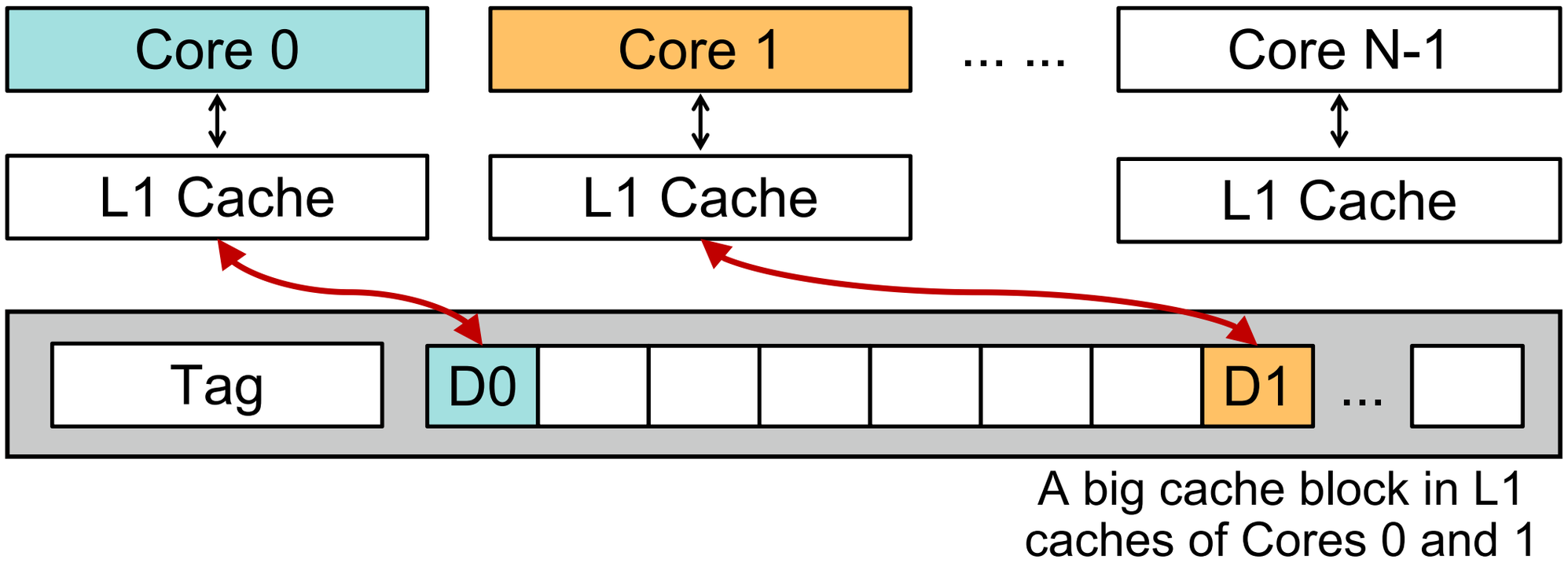
- *Write update*
  - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
  - All snooping caches update their copy



# Snooping Protocols

- Write invalidate:
  - Processor *k* wanting to write to an address, grabs a bus cycle and sends a 'write invalidate' message
  - All the other snooping caches invalidate their copy of appropriate cache line
  - Processor *k* writes to its cached copy (assume for now that it also writes through to memory)
    - Any shared read in the other processors will now miss in cache and re-fetch new data.
- Write update:
  - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
  - All snooping caches update their copy
- In either case, problem of simultaneous writes is taken care of by bus arbitration, i.e., only one processor can use the bus at any one time.

# Cache Coherence Tracked by Block

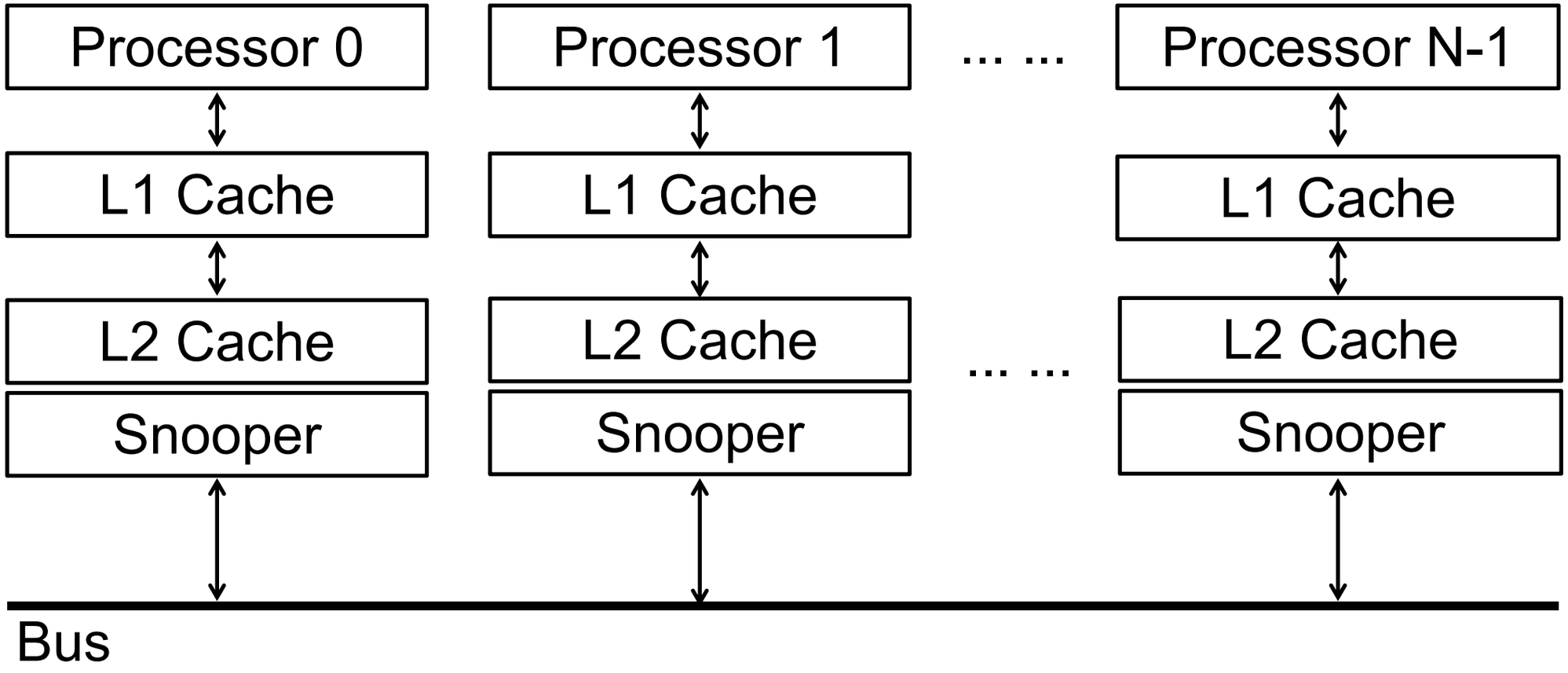



A big cache block in L1
caches of Cores 0 and 1

- Suppose core 0 reads and writes D0, core 1 reads and writes D1
- What will happed?
- ***False sharing*** effect
  - From hardware perspective, use relatively small cache block;
  - Once the hardware is given, keep variables far apart (at least block size away)

# Optimized Snoop with L2 Cache

- Processors often have two or more-level caches
  - Small L1, large L2 (usually both 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



# Implementation Issues

- Knowing if a cached value is not shared (copy in another cache) can avoid sending messages
  - But when combined with “write-back” policy, the other processors may re-fetch the old value;
- Requires protocol to handle this;
- The cache coherence protocols ensure that there is a coherent view of data, with migration and replication.
  - A cache line has a state

# Example: M(O)ESI Protocols

- For each block in a cache, track its state:
  - **Shared**: up-to-date data, other caches may have a copy; can evict the data without writing it to backing store;
  - **Modified**: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date (i.e., write-back); can be further modified freely;
  - **Invalid**: not in cache (from before: valid flag), must be fetched.
  - and optional performance optimizations:
  - **Exclusive**: up-to-date data, no other cache has a copy, OK to write, memory up-to-date;
  - **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state), the only copy that can update the memory;
  - There are different combinations of them (and the other newly invented states)
    - MSI/MESI/MOESI (AMD processor family)/MESI+F (Intel processor)

More in CAII

# True or False?

- Using write-through caches removes the need for cache coherence.

- Every processor store instruction must check contents of other caches.

- Only one processor can cache any memory location at one time.

# True or False?

- Using write-through caches removes the need for cache coherence.

FALSE. You have a copy. I do a write (through, to memory). How do you get updated when you do a read?

- Every processor store instruction must check contents of other caches.

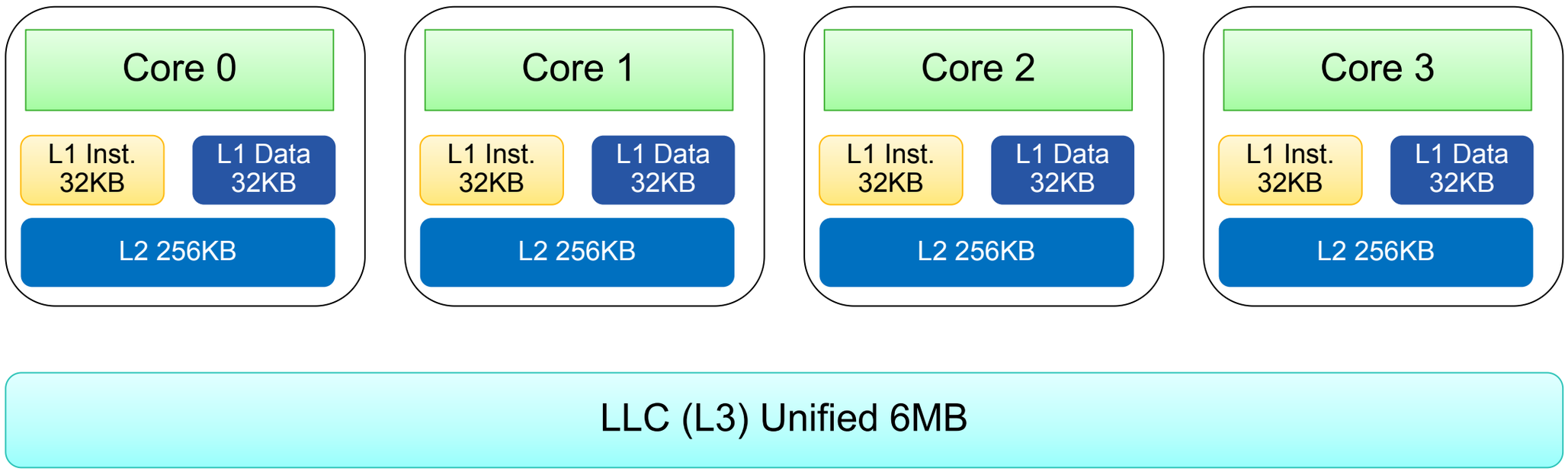
FALSE. That's the point of these protocols, to know if others have copies and whether I need to just do a store or do other work.

- Only one processor can cache any memory location at one time.

FALSE. What if they're all doing reads? That would be inefficient.

# Advanced Cache: Inclusiveness

- Inclusiveness of multi-level caches

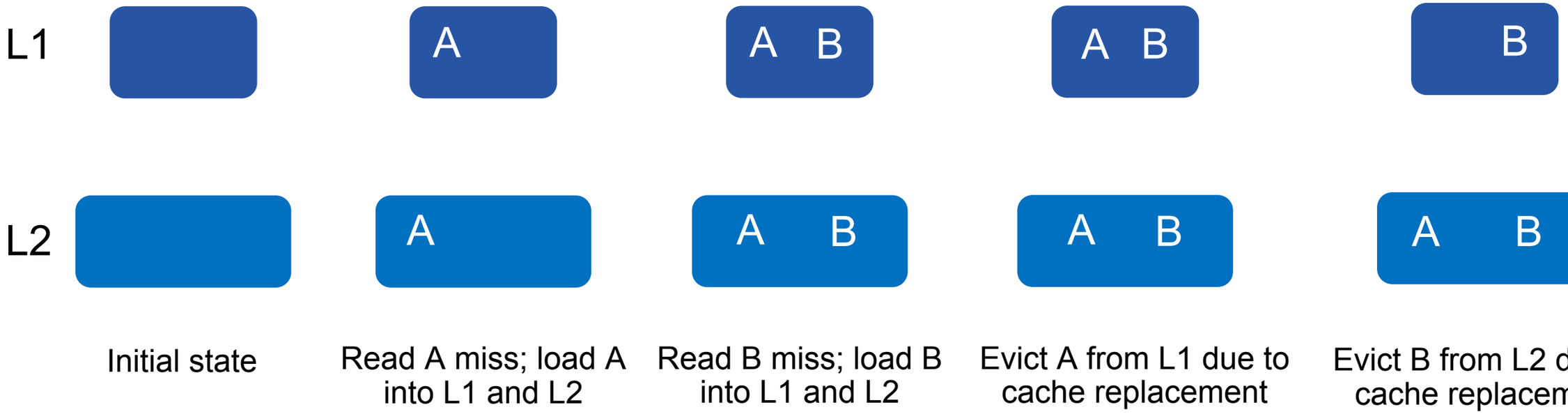



Intel Ivy Bridge Cache Architecture (Core i5-3470)

If all blocks in the higher level cache are also present in the lower level cache, then the lower level cache is said to be **inclusive** of the higher level cache.

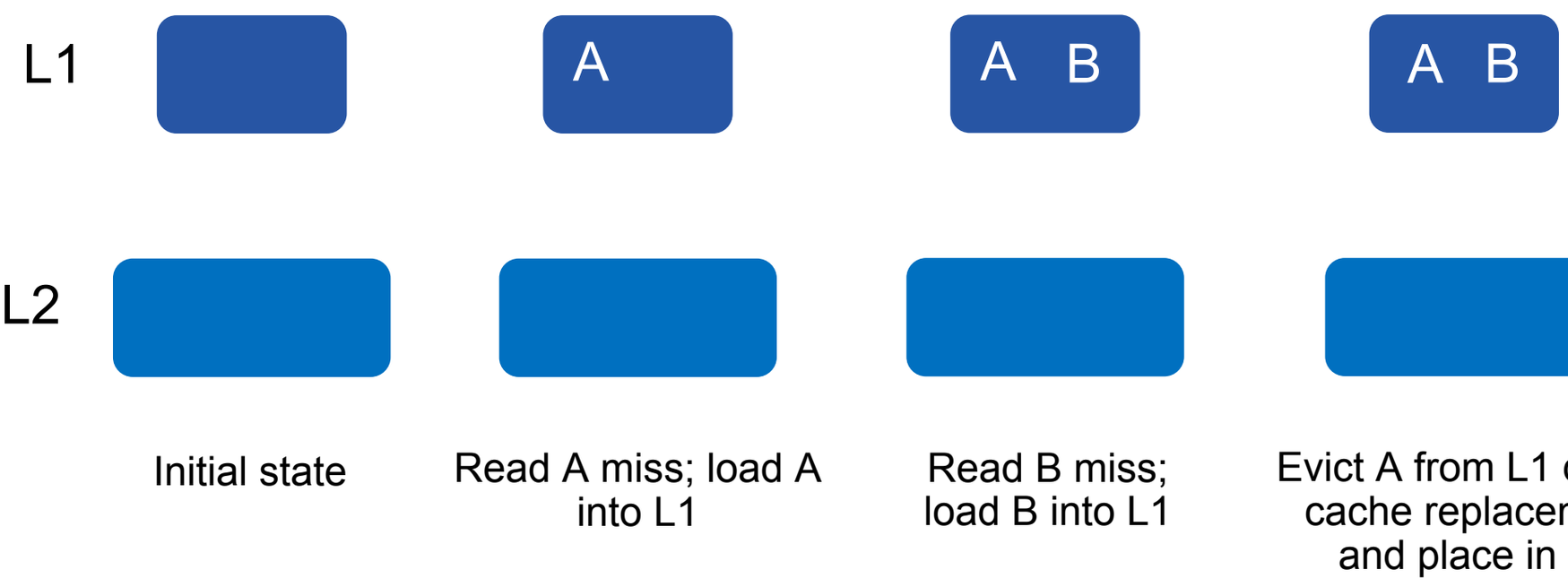
# Inclusiveness

$$L_n \subsetneqq L_{n+1} \ (n \geq 1)$$

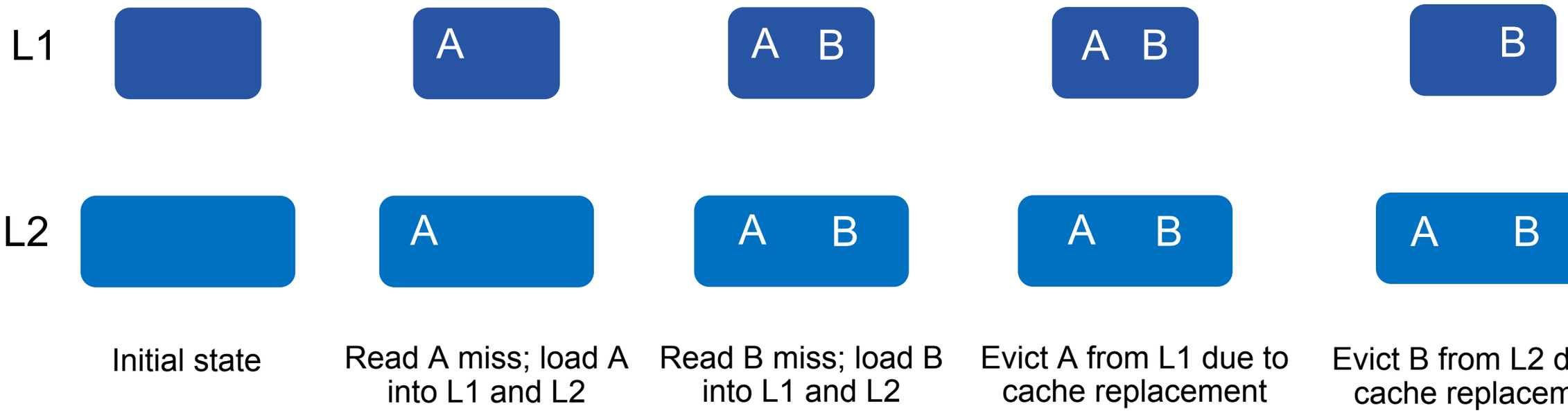


# Exclusive

$$L_n \cap L_{n+1} = \emptyset \ (n \geq 1)$$



# Non-inclusive



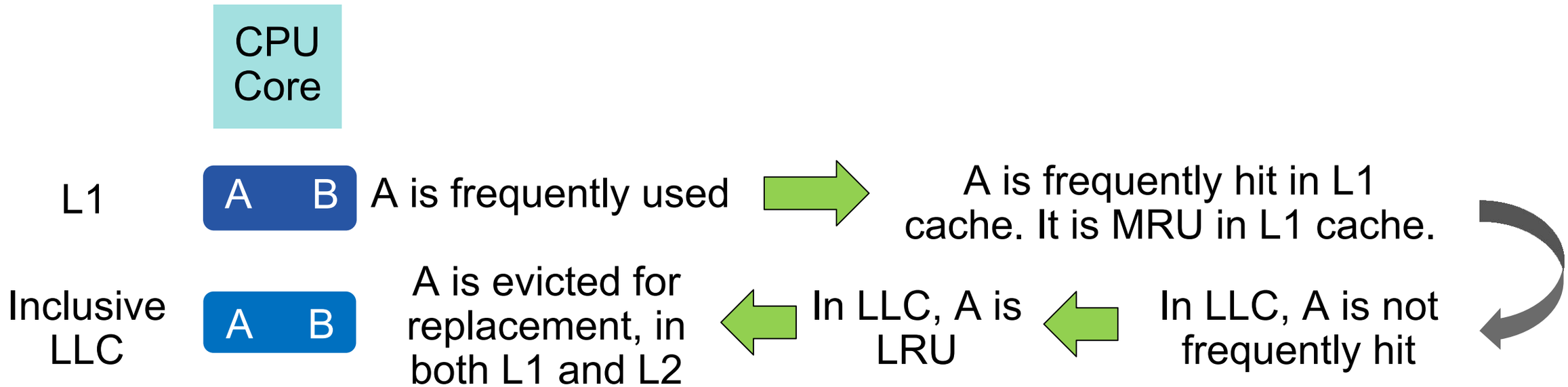
# Real Staff

- Intel processors
  - Sandy bridge, inclusive
  - Haswell, inclusive
  - Skylake-S, inclusive
  - Skylake-X, non-inclusive
- ARM processors
  - ARMv7, non-inclusive
  - ARMv8, non-inclusive
- AMD processors
  - K6, exclusive
  - Zen, inclusive
  - Shanghai, LLC non-inclusive

# Inclusive or Not?

- Inclusive cache eases coherence
  - A cache block in a higher-level surely exists in lower-level(s)
- Non-inclusive cache yields higher performance though, why?
  - No back invalidation
  - More data can be cached, larger capacity

# “Sneaky” LRU for Inclusive Cache

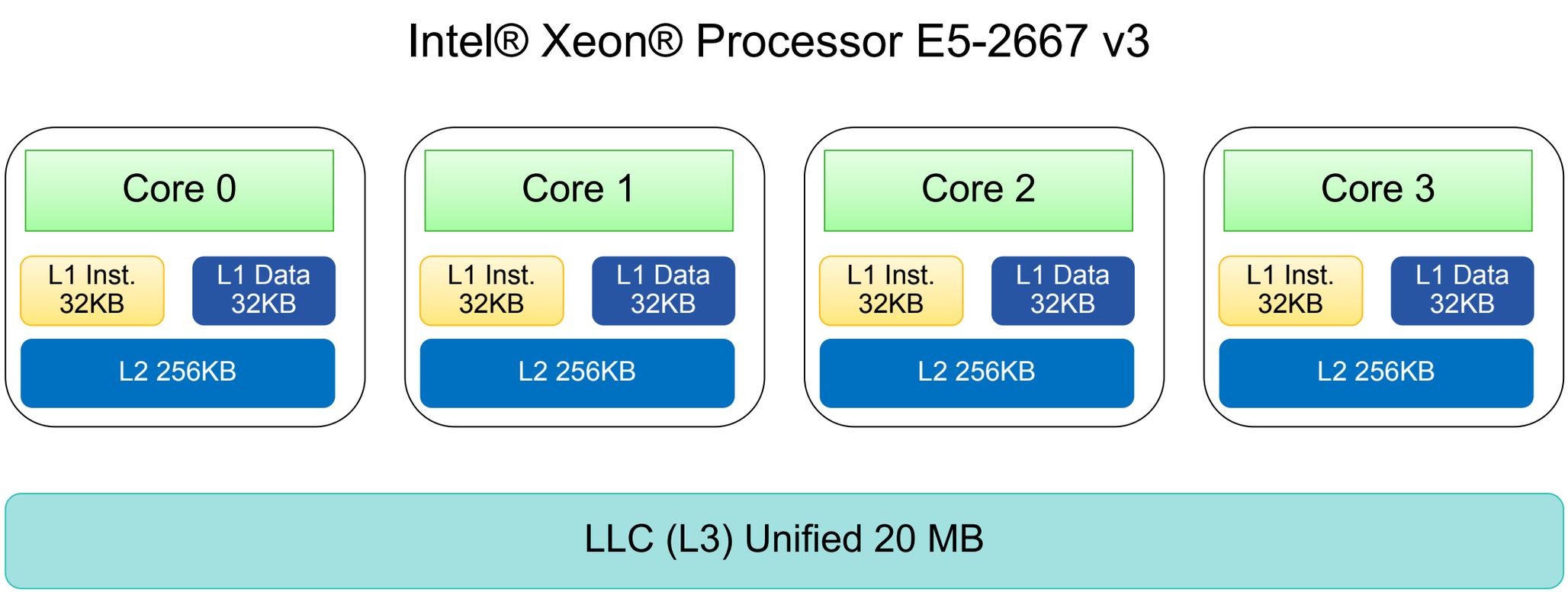



As a result, MRU block that should be retained might be evicted, which causes performance penalty.

What if LLC is non-inclusive?

Should you be interested, you can click https://doi.org/10.1109/MICRO.2010.52 to read the related research paper for details.

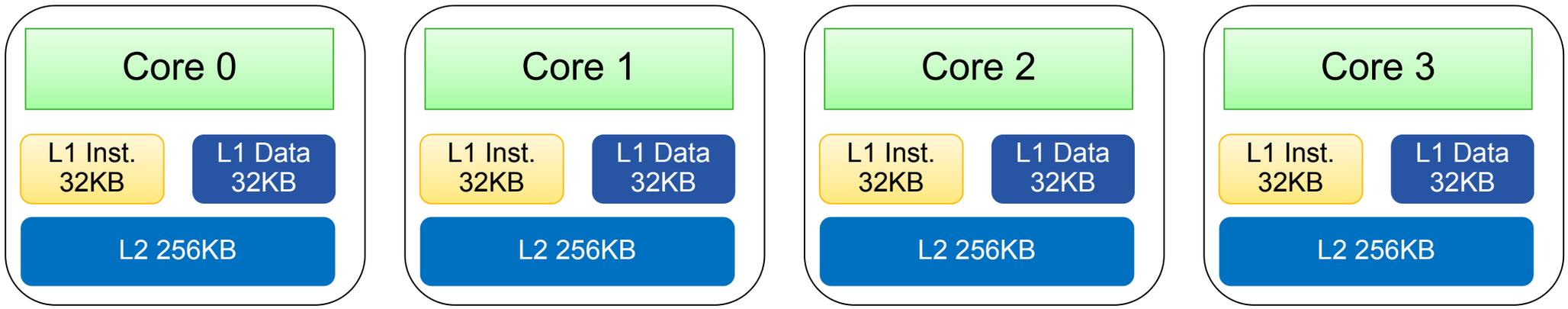
# Last-Level Cache (LLC) is not Monolithic



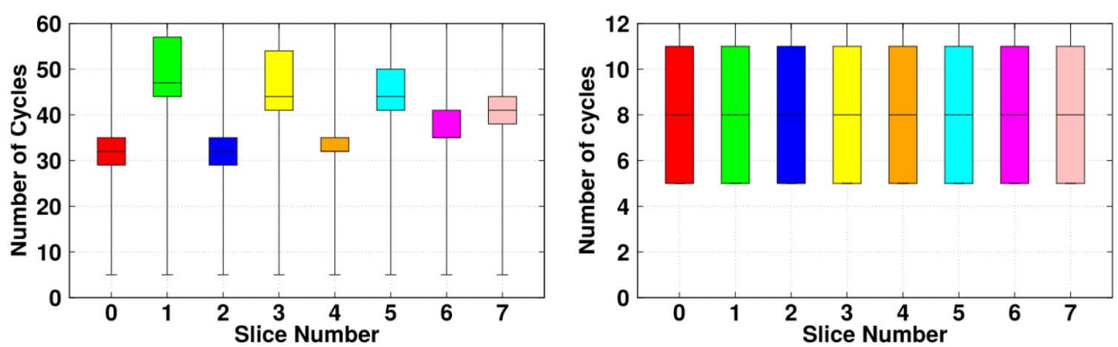

- Previously, it's considered that, to CPU cores, LLC is monolithic. No matter where a cache block in the LLC, a core would load it into private L2 and L1 cache with the same time cost.

# Last-Level Cache (LLC) is not Monolithic

Intel® Xeon® Processor E5-2667 v3




LLC (L3) Unified 20 MB




(a) Read. (b) Write.

LLC is fine-grained
LLC in 8 slices

From the paper https://doi.org/10.1145/3302424.3303977

# Slice-aware Memory Management

- The idea seems simple
  - Put your data closer to your program (core)
- But it not *EASY* to do so
  - Cache management is undocumented, not to mention fine-grained slices
  - Researchers did a lot of efforts
    - Click https://doi.org/10.1145/3302424.3303977 for details
    - They managed to improve the average performance by 12.2% for GET operations of a key-value store.
    - 12.2% is a lot, if you consider the huge transactions every day for Google, Taobao, Tencent, JD, etc.

# Summary

- There is a huge design space for CPU cache
- To make the best use of cache can boost your program’s performance!