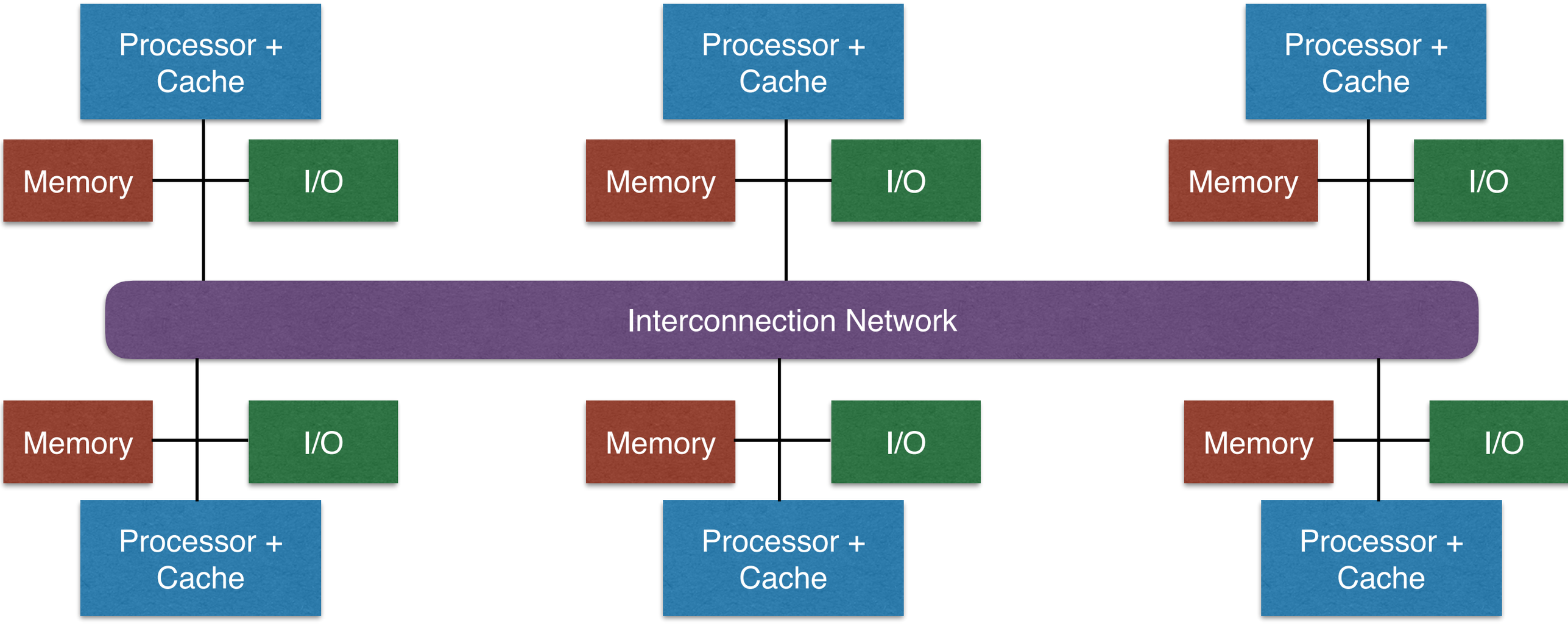
Multiprocessors

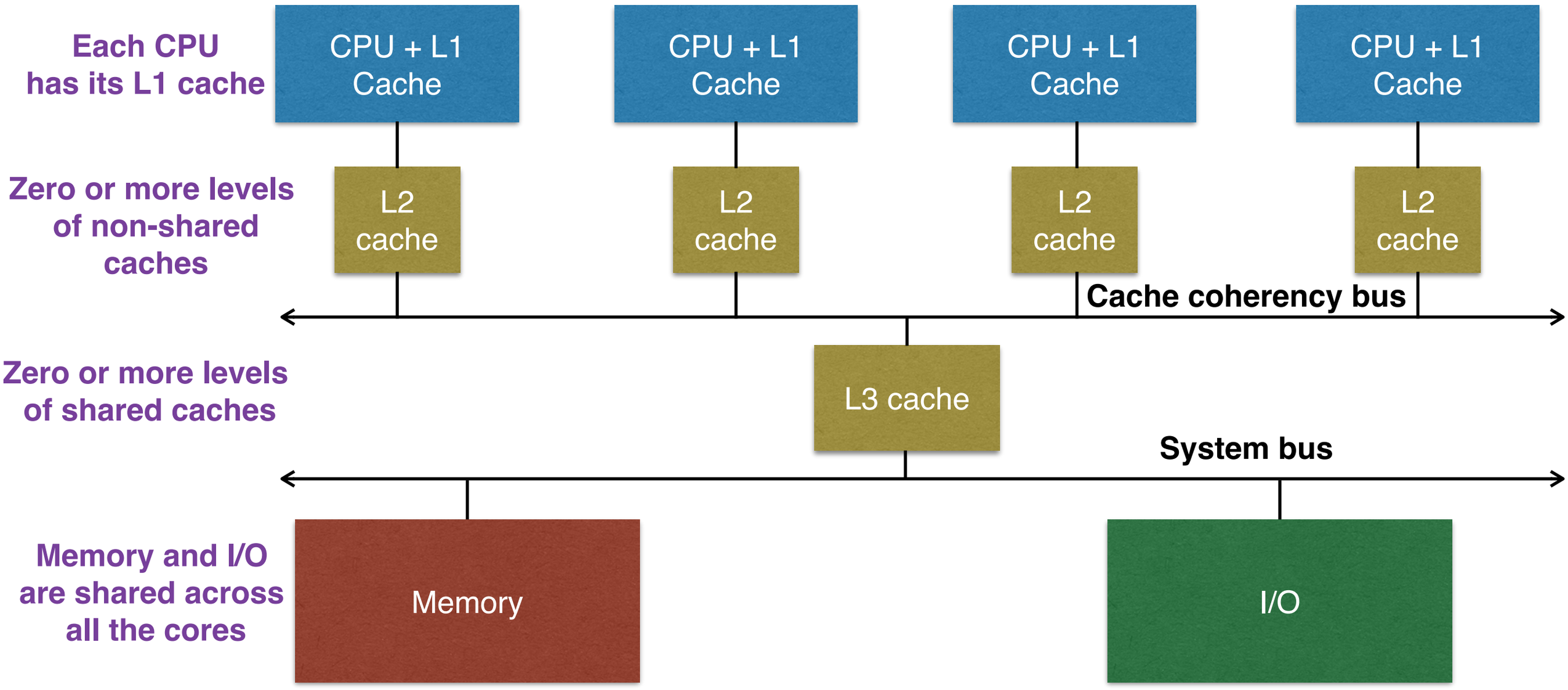
CSU34021 - Computer Architecture II

Loosely-coupled / Distributed-memory computers



- Each CPU has its own (not shared) memory and I/O devices.
- Used for distributed systems, clouds, clusters with many, many processors…

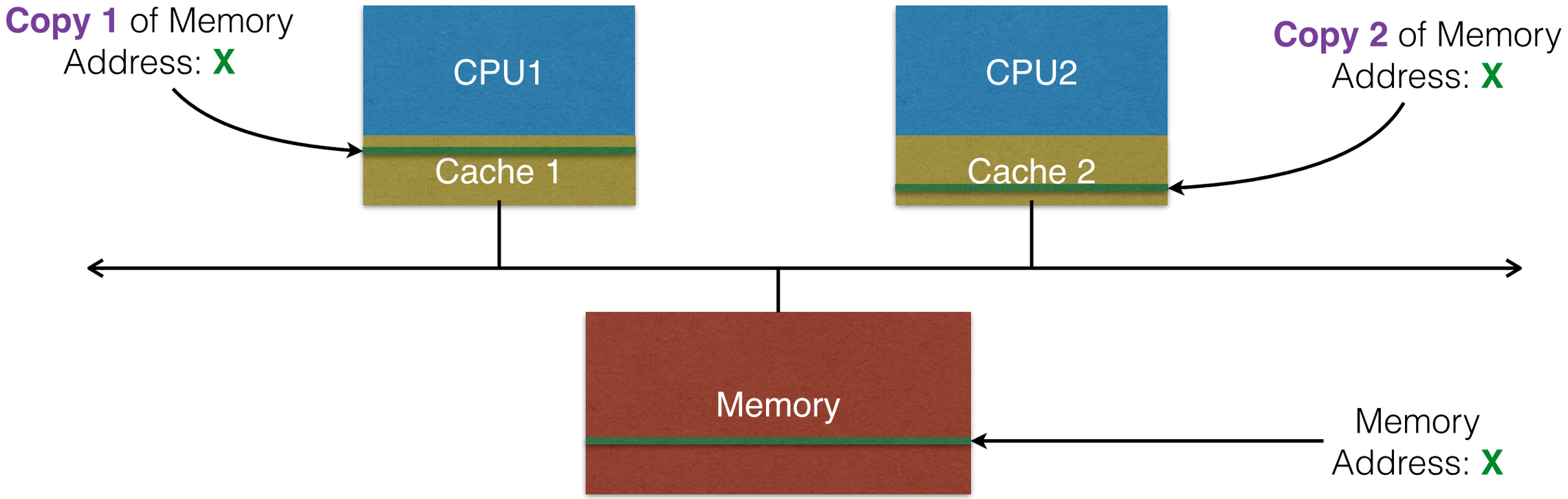
Tightly-coupled / Shared-Memory processors



[For simplicity we will ignore the L3 cache, and go to memory directly. It doesn't change the protocols in any meaningful way]

Shared-Memory Processor – What happens to the Cache?

- Consider two processors operating on the same memory address.



- What if **CPU1** read its copy of **X** right after **CPU2** writes on its copies?
- If **CPU1** reads **X** from **Cache 1**, it will have the wrong value!

Cache Coherency

- We need to enforce **cache coherency**. Formally:

    1. If **P** writes on **X**, and then reads from it, while no other processors modifies it, then reading will return whatever **P** wrote.
    2. If **P** writes on **X**, and **Q** waits "enough" time it will read whatever **P** wrote.
    3. If **P** writes on **X**, and then **Q** write on **X**, then the same ordering needs to be maintained when writing in memory (*serialisation*).

- A complimentary problem is that of **cache consistency** – i.e., how long after a **write** will an another **processor** be able to **read** that value? [we will not talk about it].

Cache Coherency Protocols

- Various cash coherency protocols have been developed…
- We will discuss solutions based on ***snoopy caches***, i.e. able to snoop all bus transactions happening around them.
- Write-once, write through, MESI… But many, many more exist…
- Protocols important: coherence bus can get quite busy with many cores…

Write-through Protocol

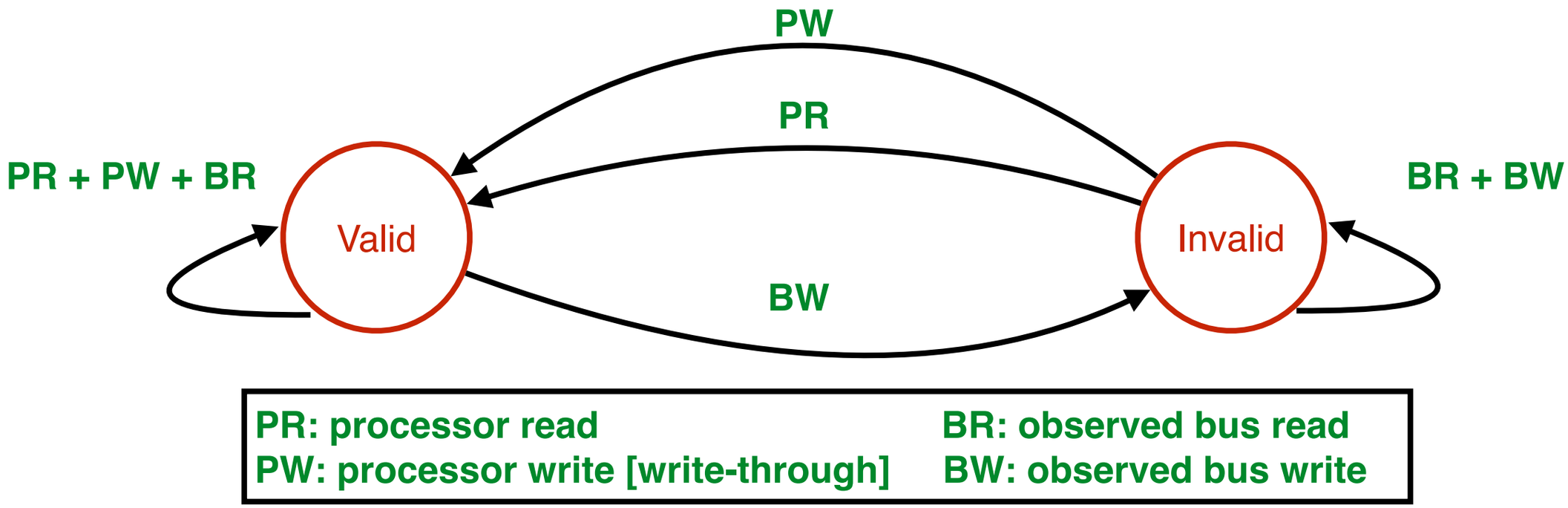
- Simple approach based on **write-through** cache [avoids the problem of delayed writing on memory…].

- Implements **write-invalidate** strategy: when a **cache** snoops from the bus a writing on a **location** it currently holds, it simply **flips its valid bit flag to zero**.

- The next time the **CPU** will try to access that **location**, the valid bit set to zero will trigger a **cache miss**, and the location will be fetched from memory (and memory is updated because of **write-through**).

Write-through State Transition Diagram

- Coherency protocols are generally described using state-transition diagrams.



- Each cache line is either valid or invalid [not cached = invalid]
- Bus traffic scales as the number of writes
- Easy to implement, because of data traffic effective only for small scale parallelism

Write-once Protocol

- It follows similar principles of write-through protocol, but is for **write-back** caches.
- Using **write-back** caches reduces the bus traffic, but introduces some complications [Why?]
- It works by adding an **extra bit flag** in the cache. Each entry can be in either one of the following **states**:
    1. **INVALID**: **cache line** is invalid.
    2. **VALID**: **cache line** valid and in one or **more** caches.
    3. **RESERVED**: **cache line** in one cache **only** and it's valid.
    4. **DIRTY**: **cache line** in one cache **only**, and it is the only up to date copy.

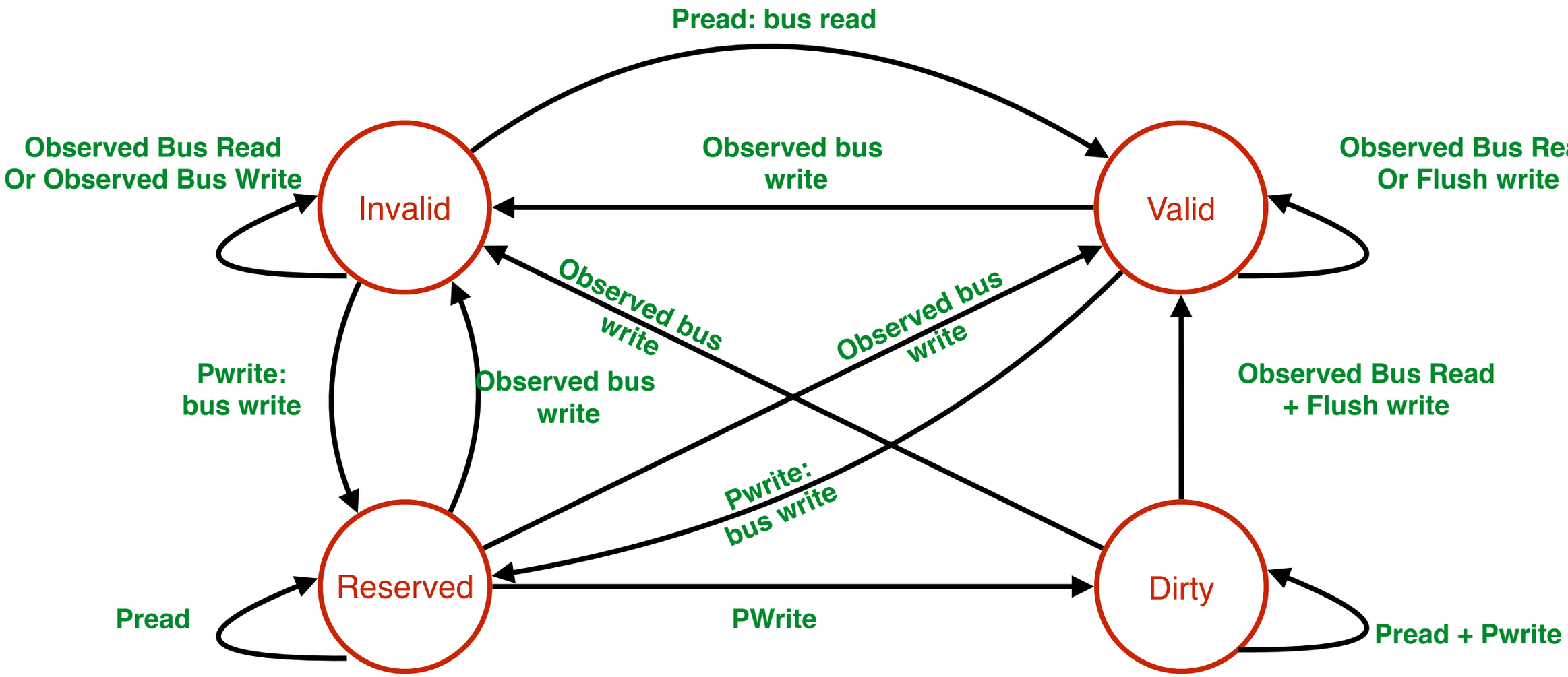
Write-once Protocol Workings

- When a memory location is first read, it enters the cache in the **VALID** state.
- A **VALID** cache line changes to **RESERVED** when written for the first time [**write-once**].
- A write which causes a transition to **RESERVED** is **written through to memory** so that all **other caches** can observe the transaction and **invalidate** their copies.
- Subsequent writes to a **RESERVED** line will use write-back and mark the line entry as **DIRTY**.

Write-once Protocol Workings

- A cache must monitor the bus for any reads or write to its **RESERVED**/ **DIRTY** lines:
  - If it observes read on **RESERVED** line it changes it to **VALID**.
  - If it observes a read from **DIRTY** line, the cache intervenes by (1) supplying the line to the requesting cache and (2) by writing it to memory. The two cache then updates their line to **VALID**.

Write-once Protocol Transition Diagram



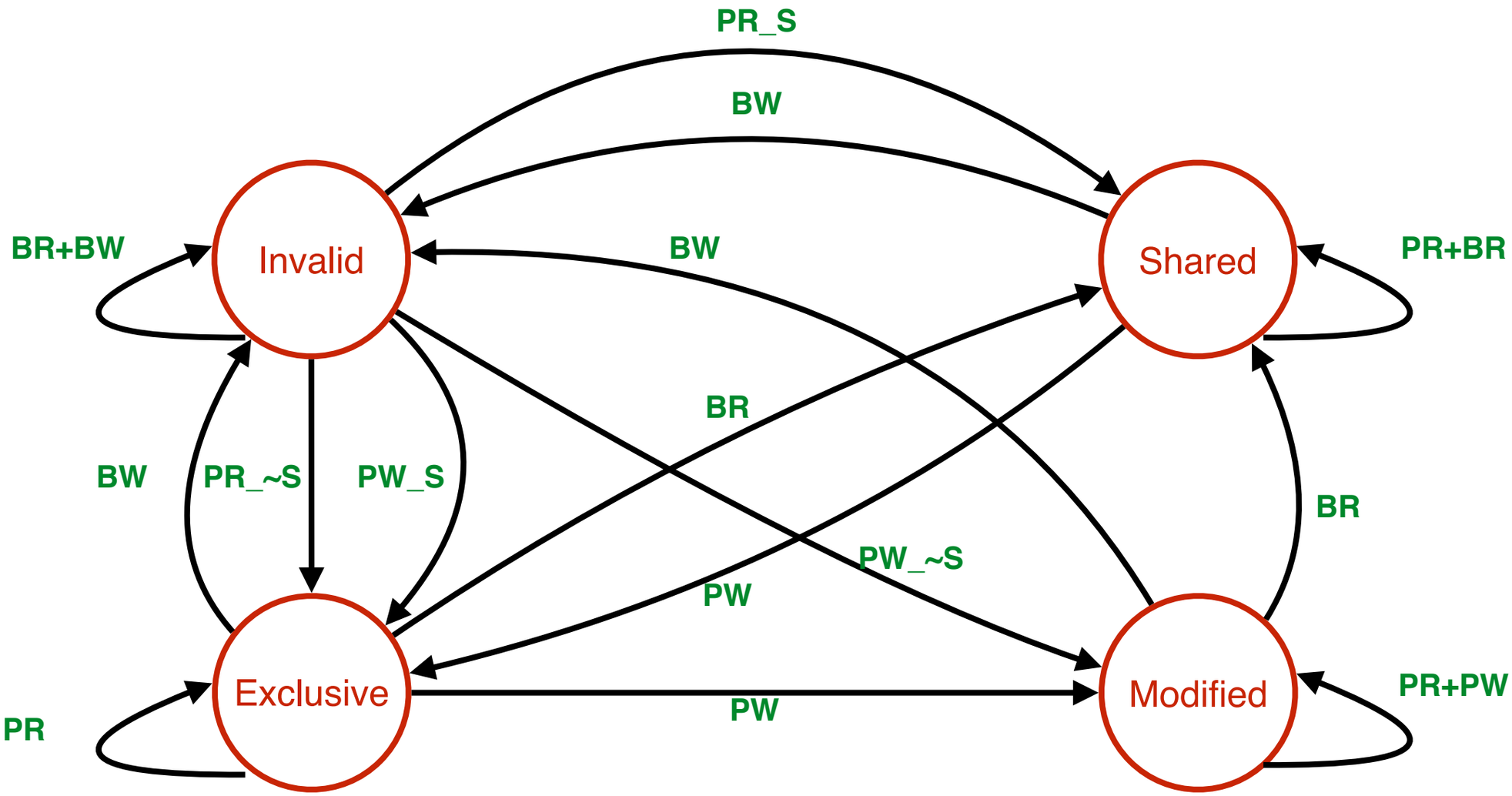
MESI Protocol

- MESI is used by Intel CPUs.
- Similar to write-once protocol but implements a **SHARED** state. A cache line can enter the cache and put directly on **SHARED** or **EXCLUSIVE** state.
- It's a **write-invalidate** protocol.
- The allowed states are:
  - **MODIFIED** [=dirty]
  - **EXCLUSIVE** [=reserved]
  - **SHARED** [=valid]
  - **INVALID** [=invalid]

MESI Protocol working

- Only a few differences with **write-once** protocol.
- Upon entering the cache a cache line might be put into **EXCLUSIVE** state immediately.
- **Write-once** protocols need for “write-through” cycles not needed if cache is in **EXCLUSIVE** mode.

MESI Protocol Transition Diagram



PR: processor read
PW: processor write
BR: observed bus read
BW: observed bus write
S/~S: shared/not-shared