CS-200
Computer Architecture
—
Part 5c. Multiprocessors Memory Consistency

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

Coherence? Consistency?

Informally:

- Coherence
  - **What values** should be returned by a set of reads and writes to a **single address**
  - An issue of data integrity: if violated, I get wrong data

- Consistency
  - **When** written values will be returned by reads to **multiple addresses**
  - Not an issue of data integrity: data are correct from the perspective of single processors, but observations from different processors may show different orderings and **contradict the expectations of the programmer**

Examples in the following will help clarify the distinction…

What about Consistency?

Processor or Thread #1

```
A = 0;
…
A = 1;
if ( B == 0 ) …
```

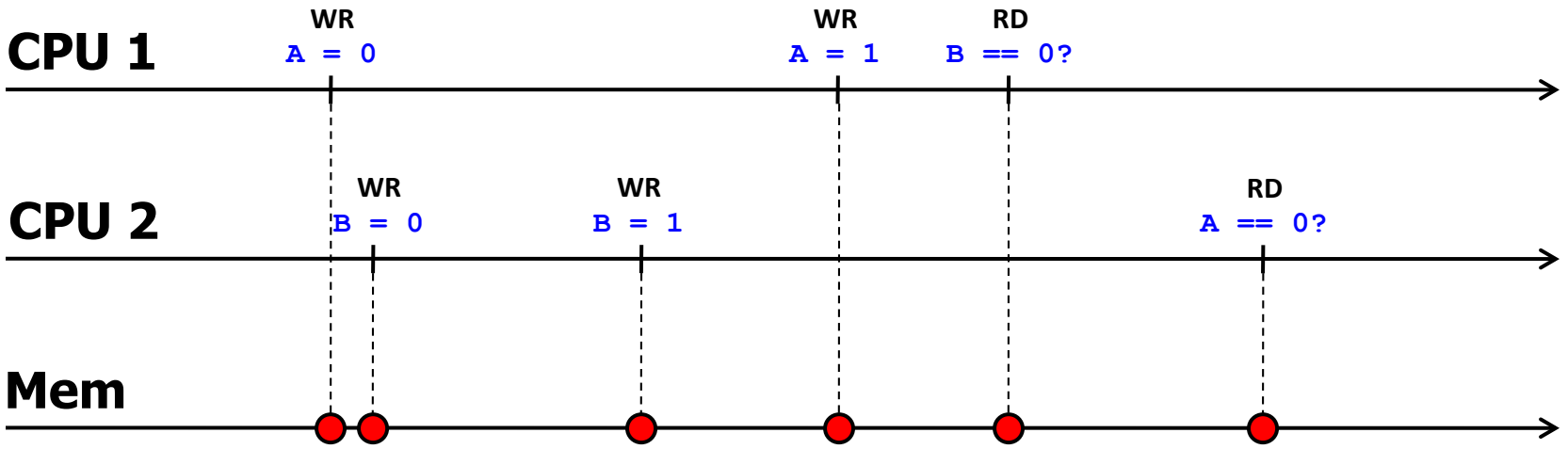
Processor or Thread #2

```
B = 0;
…
B = 1;
if ( A == 0 ) …
```

- Can **both tests be true**? Not really…
- Note that this is not a problem of accesses to a single location (coherence) but of the interaction of **several accesses to more than one location**
- It is also the problem of **when** a new value must be visible

Ideally: Strict Consistency

- Everything happens in memory **exactly in the order it has been issued**

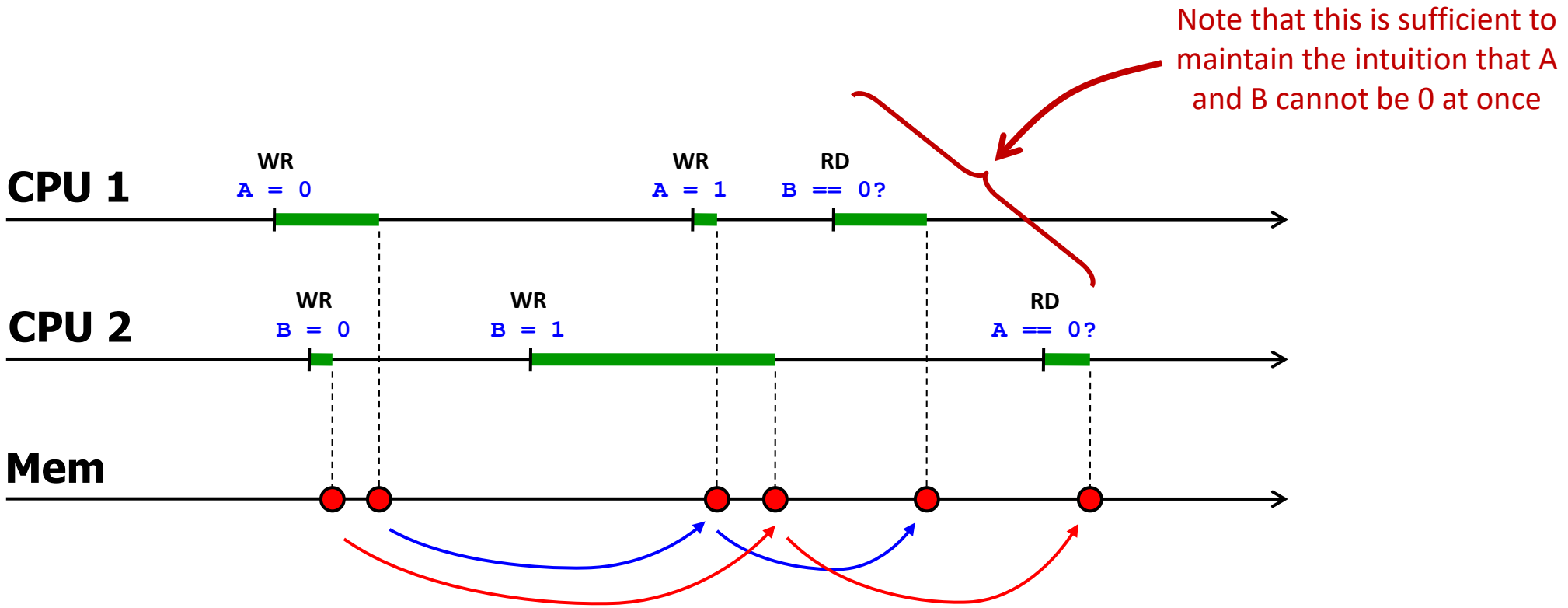


- But in any distributed system is (virtually) **impossible to obtain a global time**, hence let's forget about it…

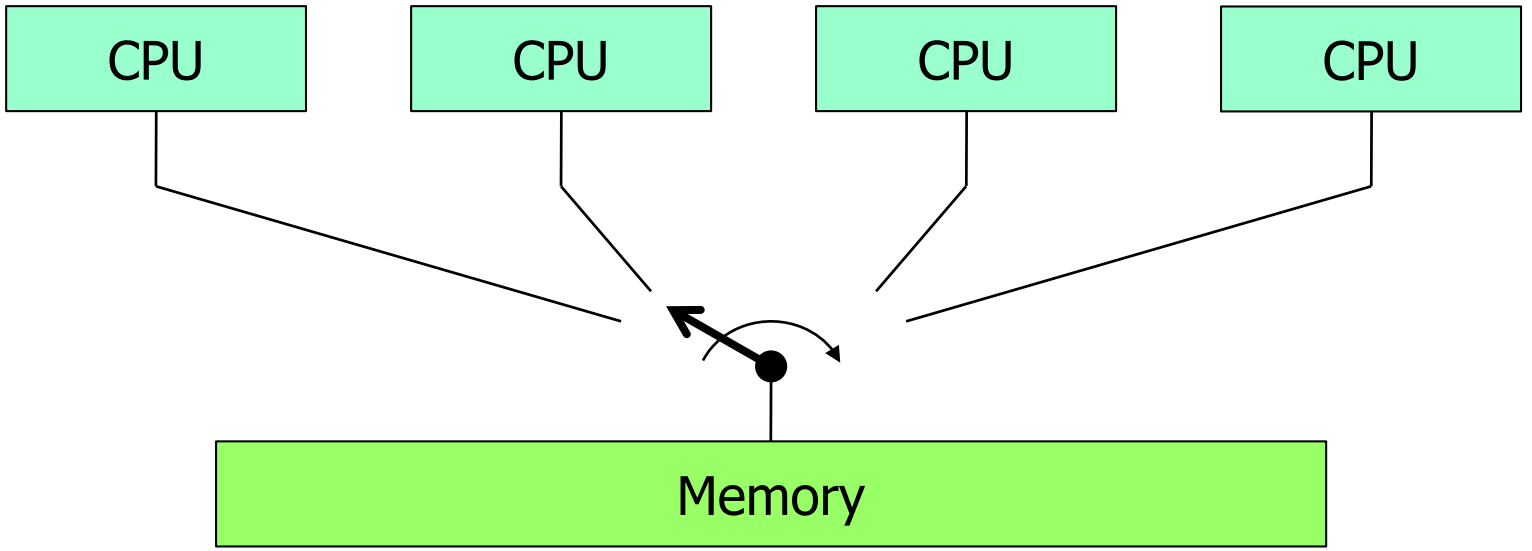
More Practical: Sequential Consistency

Sequential Consistency: the result of any execution is as if

- the operations of each individual processor were executed the order specified by its program, and
- the operations of the different processors were arbitrarily interleaved



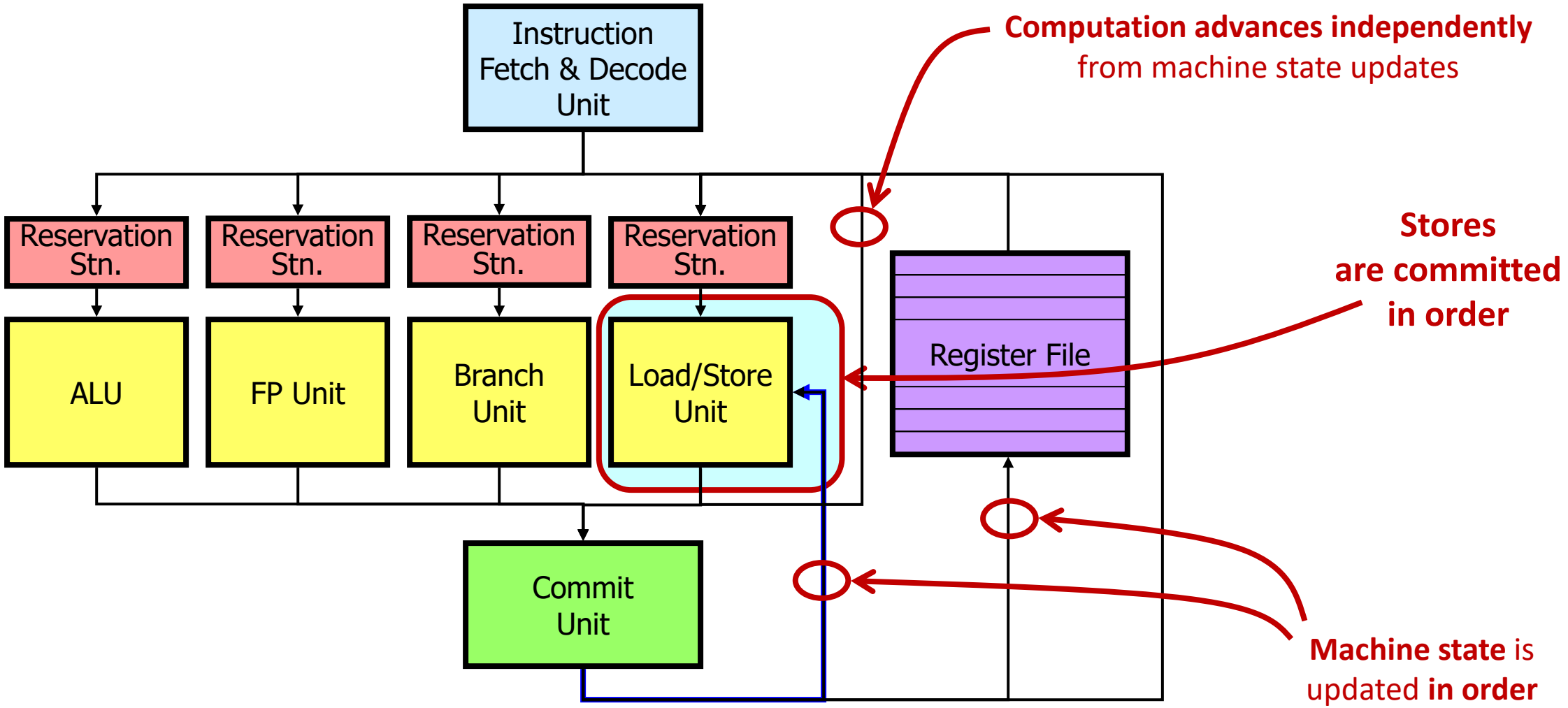
Sequential Consistency



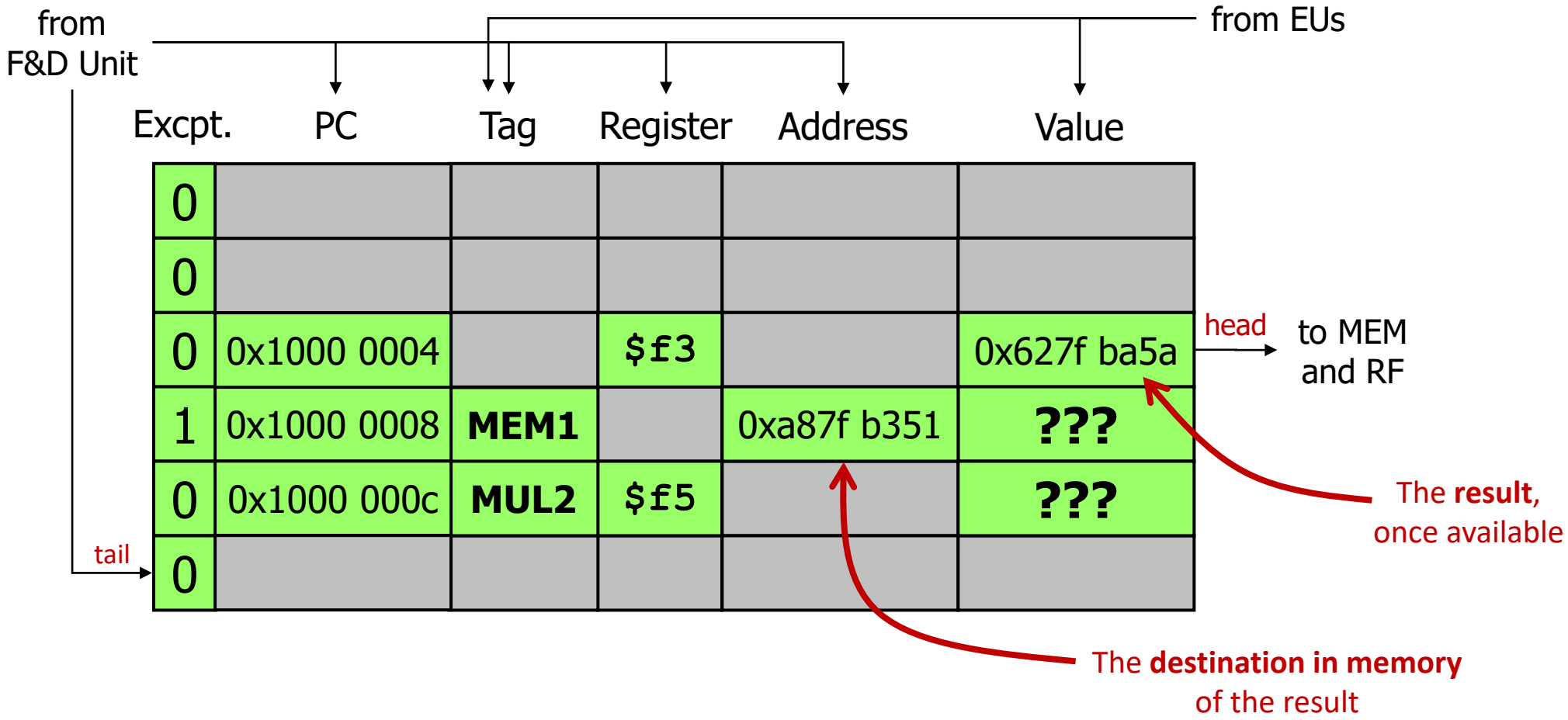
Do they?!

1. Processors **load and store in program order**
2. **Access to memory are atomic** (no other memory operation is started while the previous has not completed)
3. After every memory operation, the **switch is randomly changed**

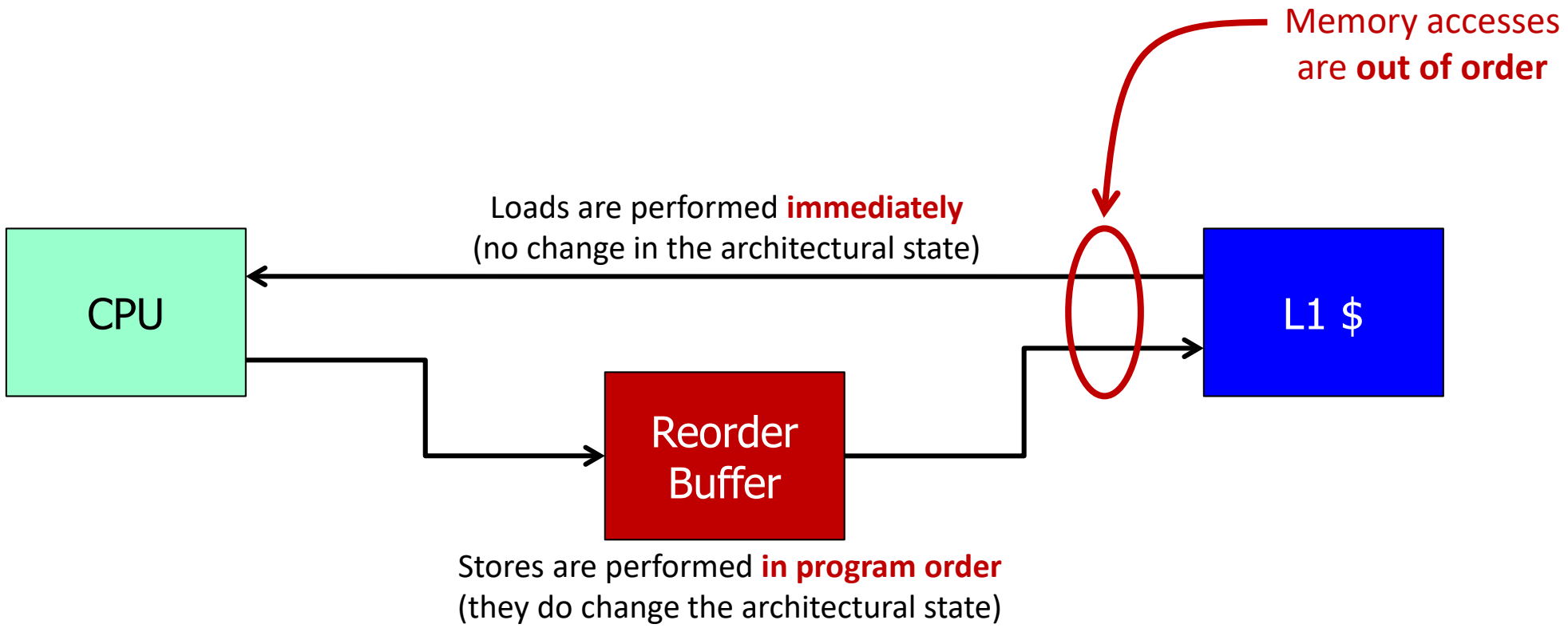
Ordering?


Instruction Fetch & Decode Unit
Reservation Stn.
Reservation Stn.
Reservation Stn.
Reservation Stn.
ALU
FP Unit
Branch Unit
Load/Store Unit
Register File
Commit Unit
Computation advances independently from machine state updates
Stores are committed in order
Machine state is updated in order

Reordering Instructions at Writeback

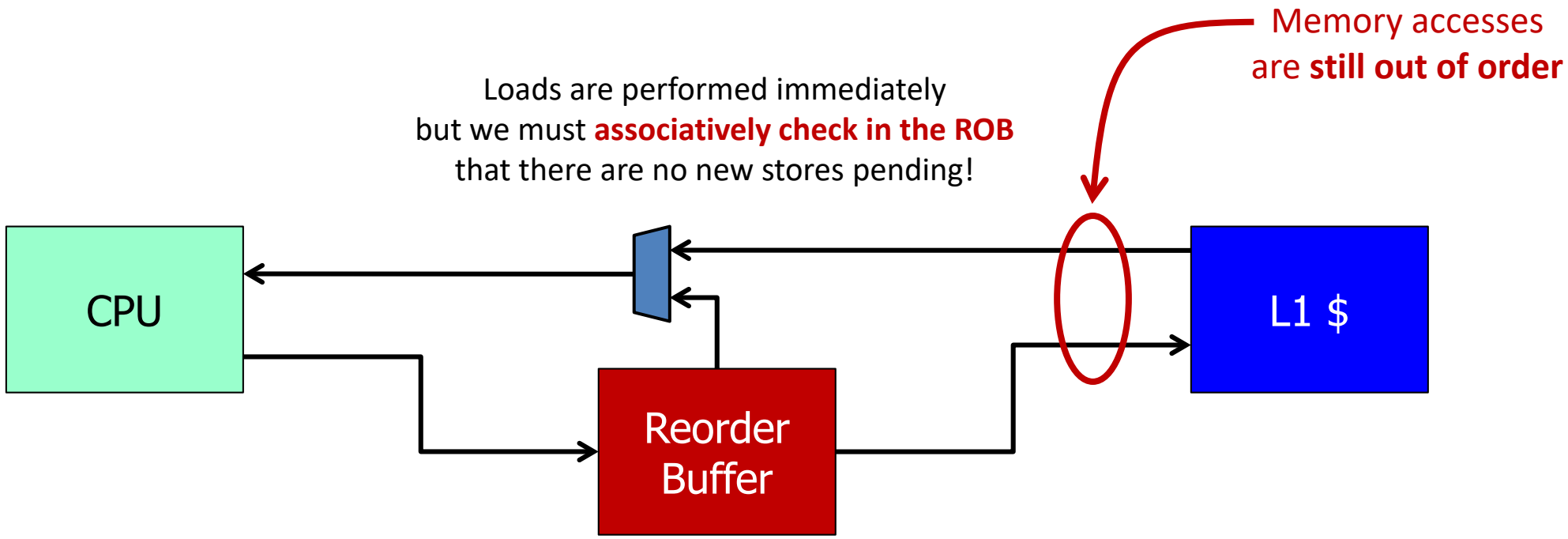


Actual Memory Path



- Is there a mistake? **Yes**, we have badly violated consistency even on a uniprocessor: the relative order of read and writes is now possibly wrong!

(Correct) Actual Memory Path



- Now we do honour RAW dependencies and **uniprocessor consistency** is correctly implemented
- Still, other processors do not have such a bypass...
- Uniprocessor memory accesses out of order → no sequential consistency

Dependences through Memory

The way we detect and resolve dependences through memory (a store at some address and a subsequent load from the same address) is the same as for registers

For every load, check the ROB:

Implements the multiplexer in the slide before

a) If there is **no store to the same address** in the ROB, get the value from memory (i.e., from the cache)

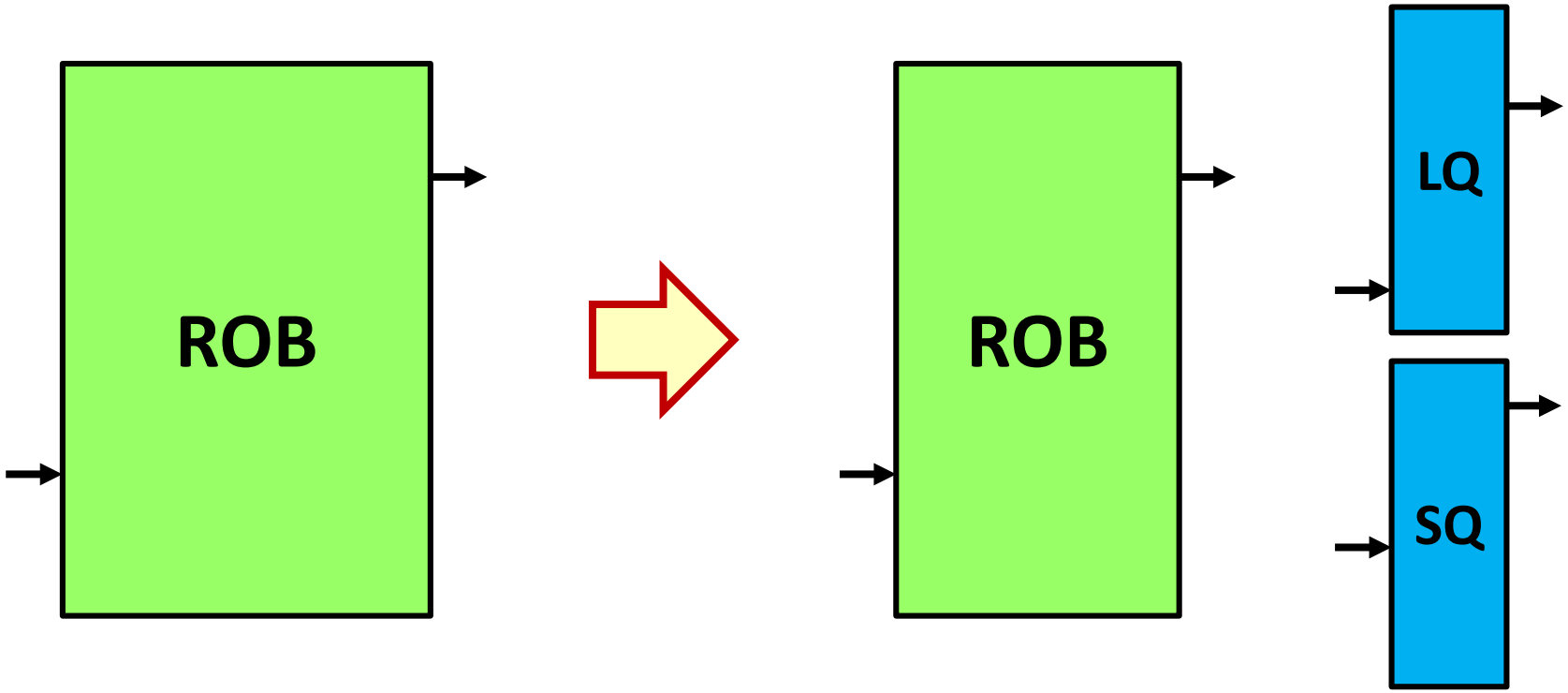
b) If there is a **store to the same address** in the ROB, either get the value (if ready) or the tag

but there is an additional situation now

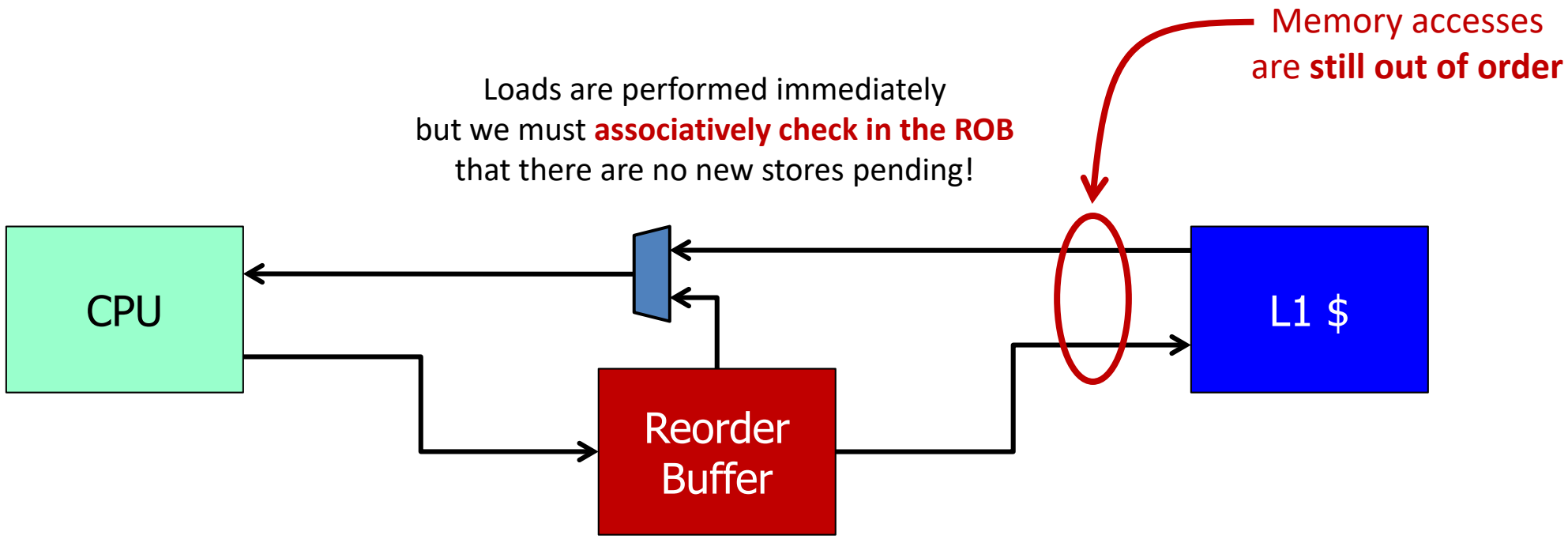
c) If there is a store to an **unknown address** in the ROB or if the address of the load is unknown, **wait!**

Load-Store Queues

In practice, the **memory part of the ROB** is implemented separately and is called a **Load-Store Queue** (in turn, usually implemented as a Load and a Store queues)



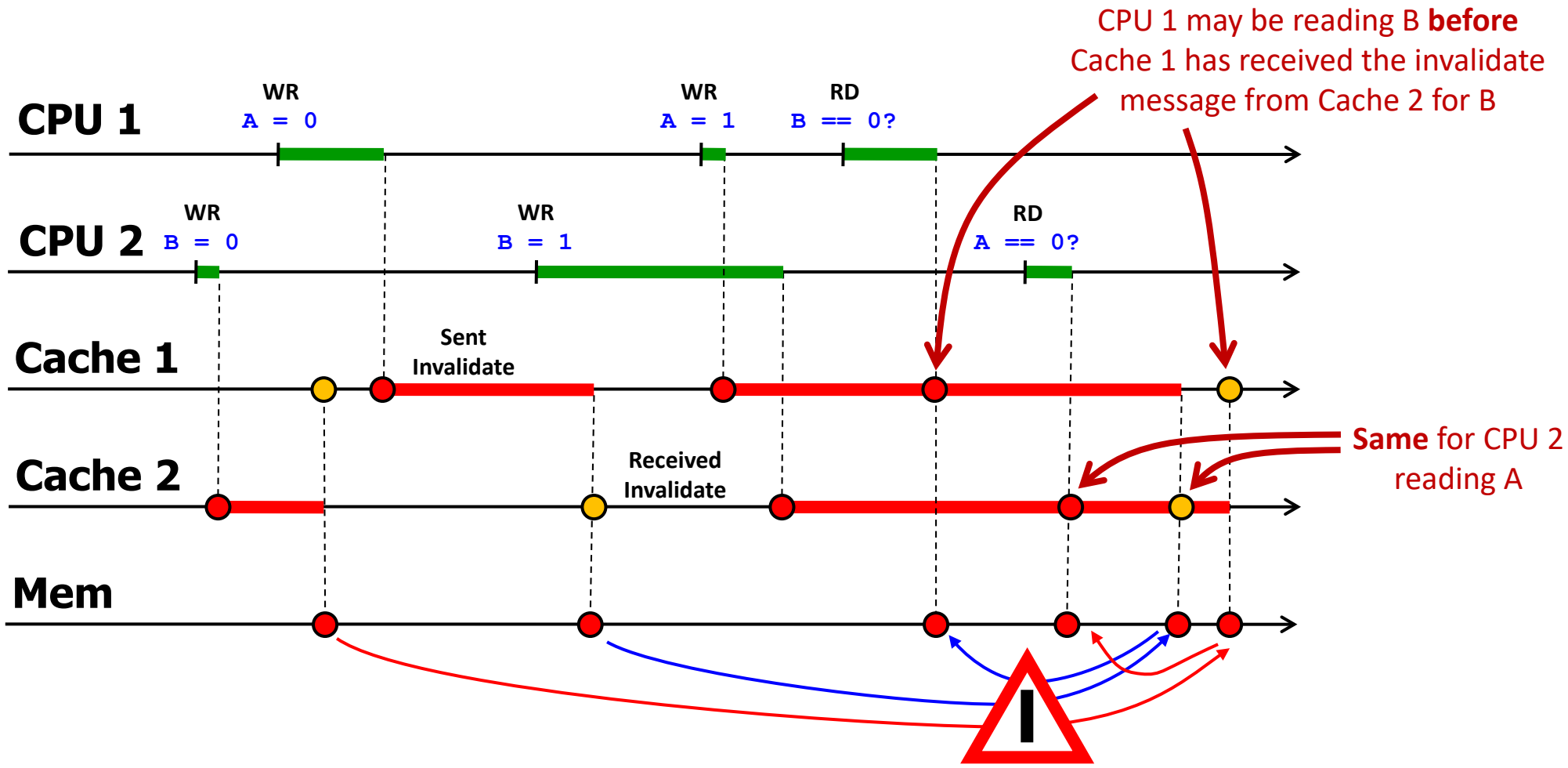
(Correct) Actual Memory Path



- Now we do honour RAW dependencies and **uniprocessor consistency** is correctly implemented
- Still, other processors do not have such a bypass...
- **Uniprocessor memory accesses out of order → no sequential consistency**

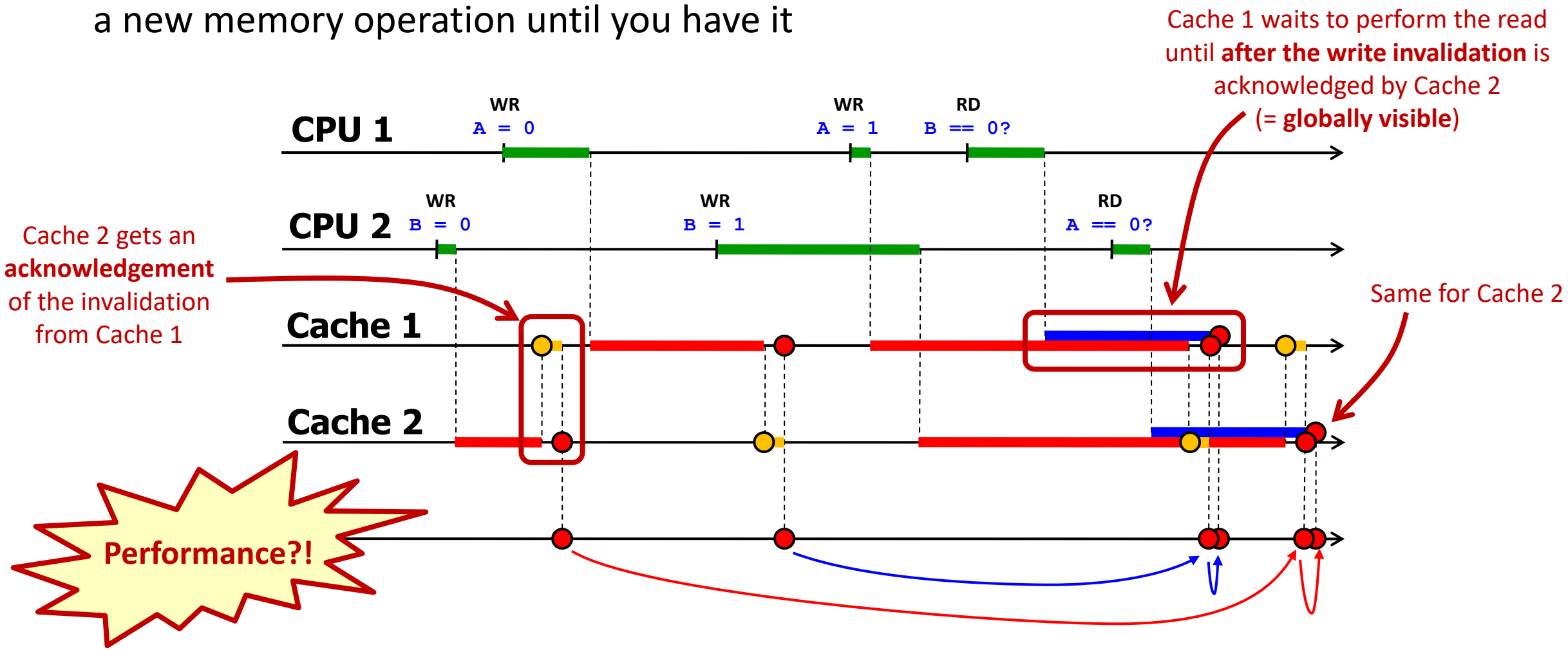
More Challenges to Sequential Consistency?

- Consider a normal system with caches and using a simple invalidate snooping protocol



How to Get Sequential Consistency

- A easy but naïve fix: wait for the acknowledgement of the invalidation and do not start a new memory operation until you have it



Relaxing Write → Read Order: *Processor Consistency* Model

- **Exposes write buffers** to the programmer (and thus grants the possibility to the architect to improve performance)

- We **now admit that both tests can be true at once** (Read advanced over independent Write)
- Still, we **enforce write order so that the values of A and B will eventually be both 1** (in some unspecified future)
- IA-32 and some other processors (IBM 370) implement this model

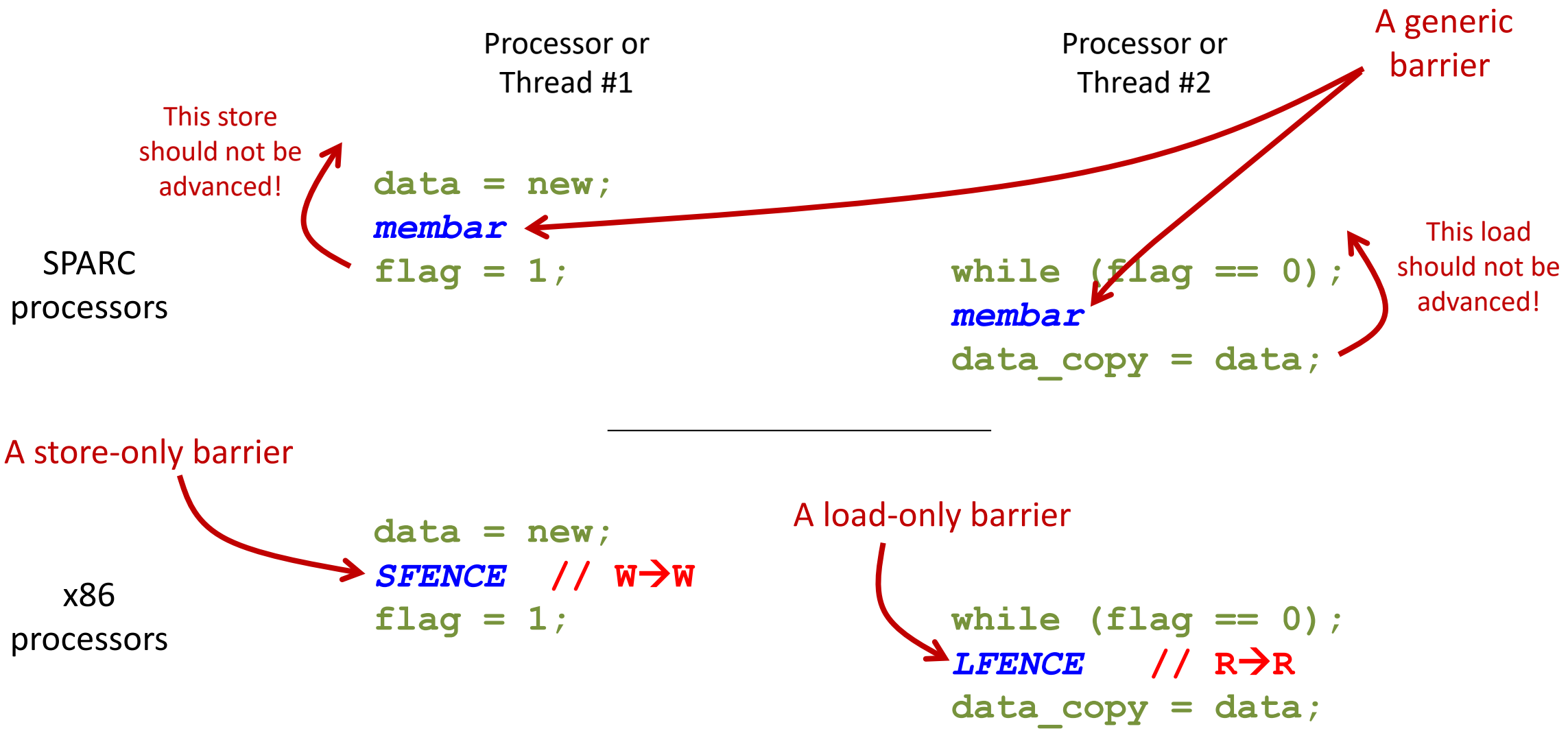
Many Relaxed Consistency Models

- The goal is to choose consistency models which are **efficiently implementable but do not "surprise too much" programmers**
- Different combinations on **what can be disordered** (W→R, W→W, R→W, R→R,…) and other details
  - Wisconsin/Stanford processor consistency
  - IBM 370
  - Intel IA-32
  - Sun Total Store Order
  - USC/Rice weak ordering
  - Stanford release consistency
  - DEC Alpha
  - IBM PowerPC
  - Sun's Relaxed Memory Order
  - …
- Only system programmers (OS, libraries, middleware) typically see these details and act on them to implement higher level functions, uniform across all or most systems

Relax Everything: *Release Consistency* Model

- Still **honour every dependence locally** in a processor, but **otherwise completely disregard ordering** across normal loads and stores
- Introduce special synchronization operations that have strict ordering
  - Typically some instruction are used to **acquire access** (S_A) to a shared variable and enforce the orderings $S_A \rightarrow W$ and $S_A \rightarrow R$, while other instructions are used to **release access** (S_R) to a shared variable and enforce the orderings $W \rightarrow S_R$ and $R \rightarrow S_R$
  - Another approach is to have memory **barriers** or **fences** (**S**) that act like $S_A + S_R$ and enforce all orderings $W \rightarrow S$, $R \rightarrow S$, $S \rightarrow W$, and $S \rightarrow R$ (i.e., the execution of a memory barrier waits for all pending loads and stores to **complete and be globally visible**, and does not let any successive load or store start).
- Put the **burden on the programmer/compiler** and be as aggressive as you can in the hardware

Memory Barriers / Fences



Atomic Instructions

- Combinations of load and store **without interference from others**
- A typical way to implement **acquire access**

- **Test-and-set**: interchanges a fixed value for a value in memory
- **Atomic exchange** or **swap**: interchanges a value in a register for a value in memory
- **Compare-and-swap**: compare a register value to a value in memory addressed by another register, and if they are equal, then swap a third register value with the one in memory

Good because it writes **only** if the comparison is successful
Bad because it needs **three source** registers

RISC-V: Load-Reserved/Store-Conditional

- Acquire access/lock

Behaves like a normal load but sets a **reservation** on M[a0]; expects to be followed by an `sc.w`

```
        li    t0, 1            # t0 = 1 = locked value; 0 = unlocked
again:  lr.w  t1, (a0)         # load-reserved to read lock
        bnez  t1, again        # try again if someone else has the lock
        sc.w  t2, t0, (a0)     # attempt to store t0 in the lock
        bnez  t2, again        # try again if store fails = someone took it
locked:                        # lock acquired: S_A→W and S_A→R
```

`sc.w t2, t0, (a0)` is like `sw t0, 0(a0)` but
(i) does not store if M[a0] has changed since the last `lr.w` and
(ii) returns nonzero in t2 if it fails to store

- Release access/lock

```
sw      zero, 0(a0)        # free lock by writing 0 = unlocked
```

Consistency is Hard!

- Memory **Consistency is Hard**
  - Subtle interactions between hardware optimizations (e.g., store buffers, reordering) and memory models make **reasoning about correctness challenging**
- Code is **Subtly Processor-Dependent**
  - Programs can **behave differently** based on the processor's memory consistency model (e.g., x86 vs ARM), requiring careful design for portability
- **Simplified for Software Programmers**
  - To shield developers, **consistency mechanisms are encapsulated** in
    - System libraries (e.g., synchronization primitives, atomics)
    - APIs (e.g., C++ std::atomic, pthreads, Java volatile)
  - These APIs are simple, intuitive, and **uniform across platforms** while hiding processor-specific details

Multiprocessors

- Multiprocessors have **come to the consumer market** and are here to stay
- Peculiar multiprocessors (e.g., heterogeneous) have been for many years in high-end embedded systems
- They can usually **take advantage of most of the progress in uniprocessor design** and performance optimization
- Yet, they involve **major challenges when it comes to preserve the multithreaded performance** of uniprocessors (interconnection, coherence, consistency, etc.)
- **Scalability** is one of the greatest architectural issues of the future

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Sections 2.11** (Synchronization)
  - **Sections 5.10** (Parallelism and Memory Hierarchy)