Lecture 14: Introduction to Memory Consistency

Vivek Kumar

Computer Science and Engineering

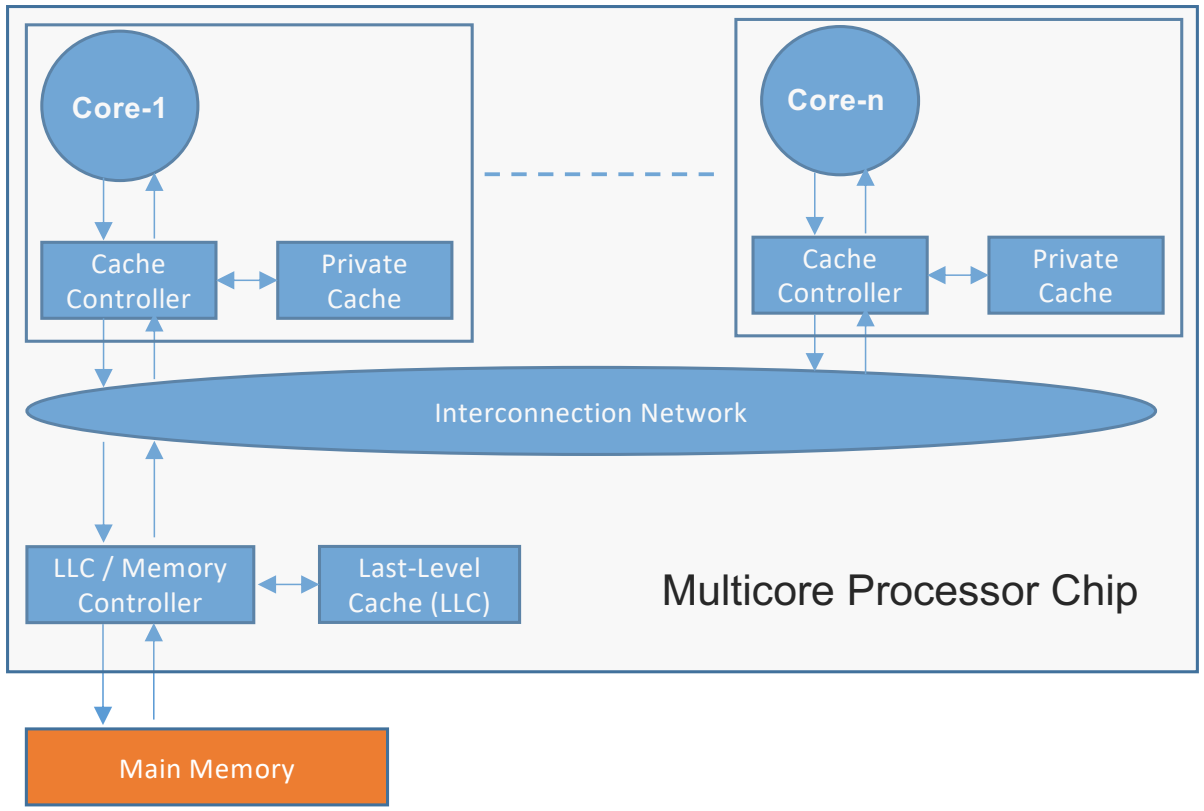
IIIT Delhi

vivekk@iiitd.ac.in

Today's Class

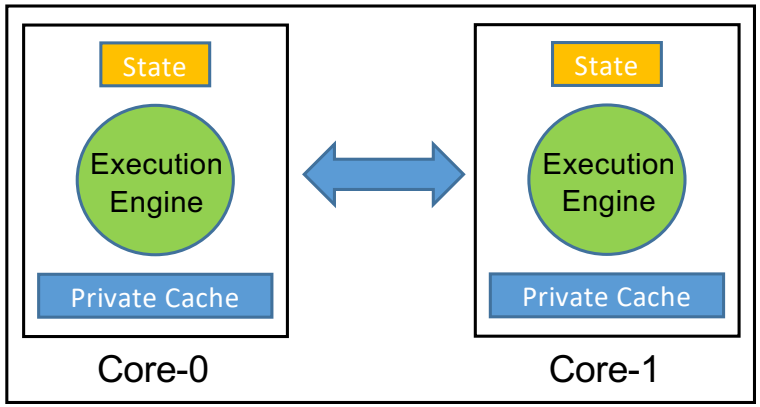
- Memory consistency problem
- Difference between coherence and consistency
- Sequential consistency
- X86 TSO memory model

Multicore System Overview

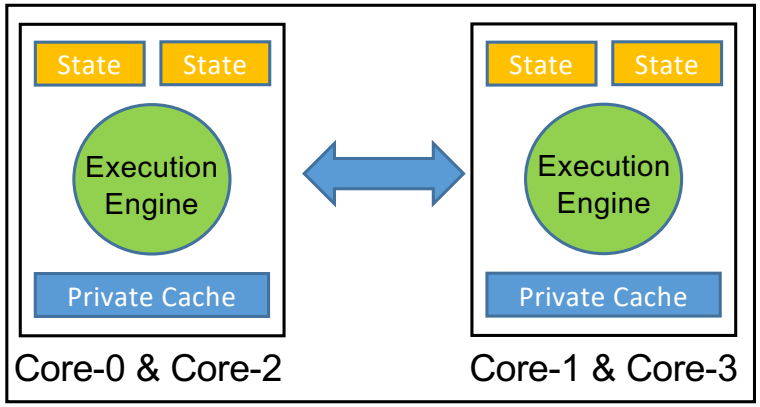


- Cache controller copies code/data to/from main memory
  - It runs the actual cache coherence algorithm
- LLC is shared with all the cores
- LLC is logically a memory side cache as it primarily serves to reduce the latency and increase bandwidth of memory accesses

Physical VS. Logical Cores



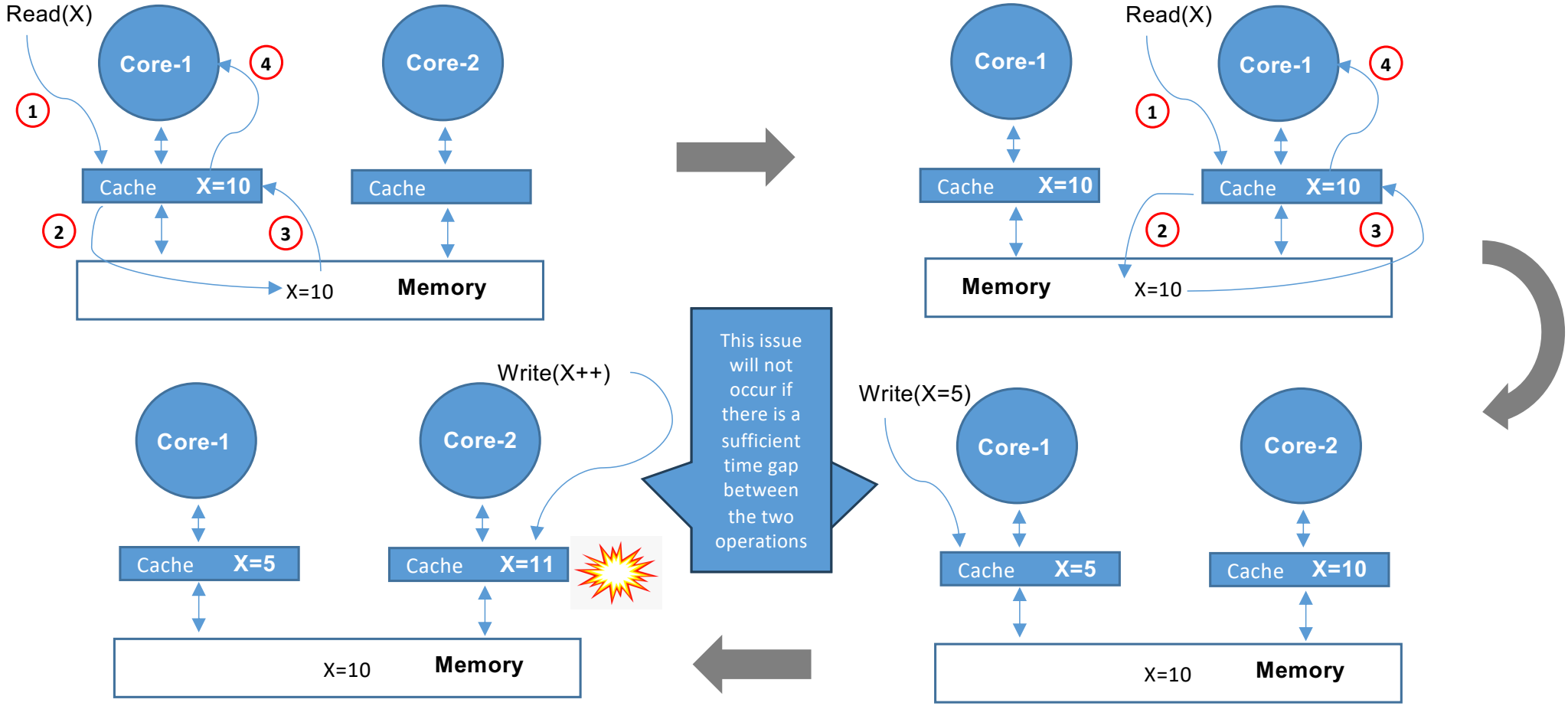
Dual-core processor with hyperthreading **DISABLED**



Dual-core processor with hyperthreading **ENABLED**

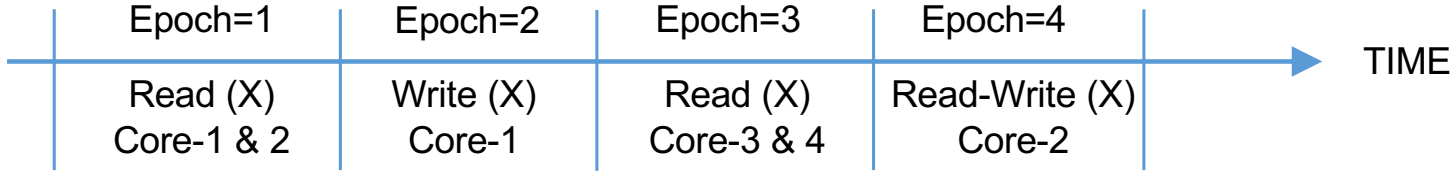
- Architectural state of a core are the registers (EBP, ESP, EIP, etc.)
- Logical cores of a processor share
  - Private cache
  - Execution engine
  - System bus interface
- If the execution of one of the logical core blocks (e.g., Core-0 waiting for a memory fetch from the DRAM) then the other logical core (Core-2) can resume its execution with its own state

The Cache Coherence Problem



Cache Coherence (CC) Protocols

- The issue highlighted in previous slide still happens even when CC protocols are being followed
- CC definition / requirements
  - Single-Writer-Multiple-Reader (**write serialization**)
    - If Core-1 writes X=5 and Core-2 writes X=11, all other cores must either see X=5 followed by X=11 or directly X=11. It will never be the case that some cores see X=5, while other see X=11 at the same time
  - Value of a memory location is ***eventually*** propagated to all readers (**value propagation**)



Dividing a memory location's lifetime into epochs

Multithreaded Programs

Initially X = Y = 0

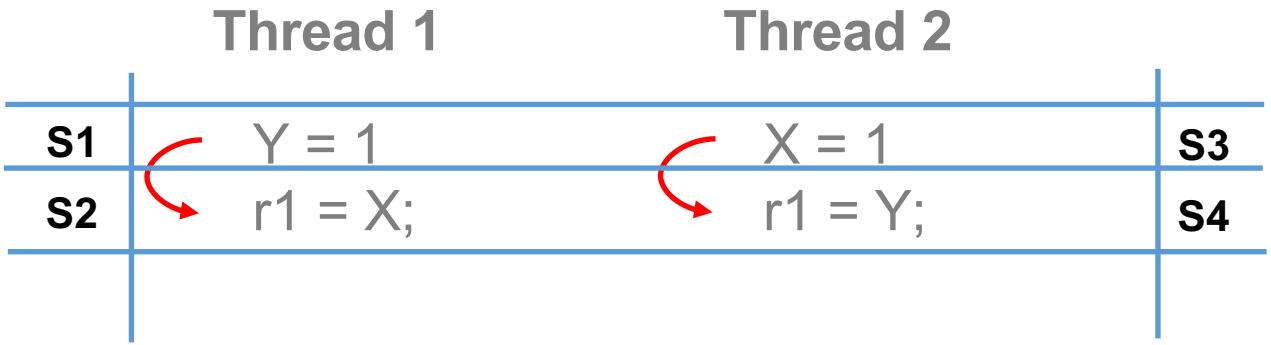
We are referring to two different memory locations. Scope of CC protocols is limited to an individual memory location

Thread 1	Thread 2
Y = 1	X = 1
r1 = X;	r2 = Y;

Question: What can be printed for “r1” and “r2” ?

Why Memory Consistency Model?

- What does programmer think?
  - Cores atomically executes one instruction at a time in **program order**
    - Valid over a single core (sequential) processor



Let's see if we can reproduce it on our machine

Question: According to you as a programmer, what are the valid program orders here?

1. S1 – S2 – S3 – S4
2. S3 – S4 – S1 – S2
3. S1 – S3 – S2 – S4
4. S1 – S3 – S4 – S2
5. S2 – S1 – S3 – S4
6. S2 – S1 – S4 – S3

Code available online:
https://github.com/hipec/cse513/tree/main/lec14

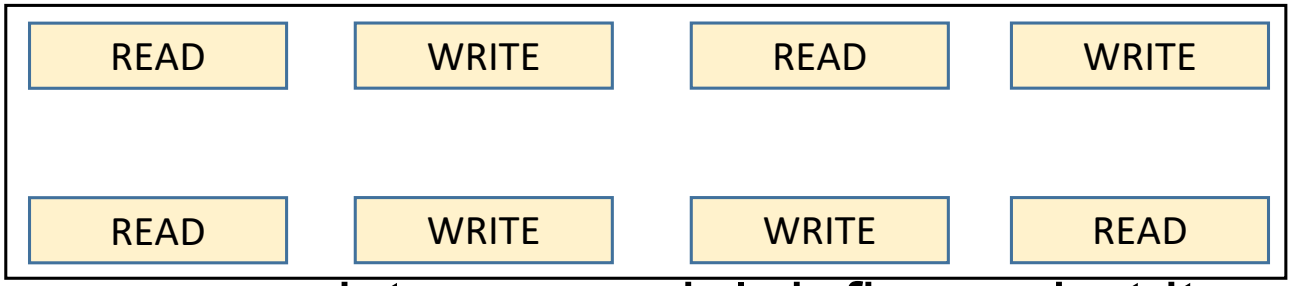
```
int X, Y;
int r1, r2;

void *threadFunc(void *param) {
  int id = *((int*)param);
  while(!shutdown) {
    wait(id);
    switch(id) {
      case 1:
        X = 1;
#ifdef USE_CPU_FENCE
        asm volatile("mfence" ::: "memory");  // Prevent CPU reordering
#else
        asm volatile("" ::: "memory");  // Prevent compiler reordering
#endif
        r1 = Y;
        break;
      case 2:
        Y = 1;
#ifdef USE_CPU_FENCE
        asm volatile("mfence" ::: "memory");  // Prevent CPU reordering
#else
        asm volatile("" ::: "memory");  // Prevent compiler reordering
#endif
        r2 = X;
        break;
    }
    signal(0);
  }
  return NULL;  // Never returns
}

int main()
```

Why Memory Consistency Model?

- What does programmer think?
  - Cores atomically executes one instruction at a time in **program order**
    - Valid over a single core (sequential) processor
- What cores can do?
  - Reorder memory operations to two different memory locations to reduce memory access latency

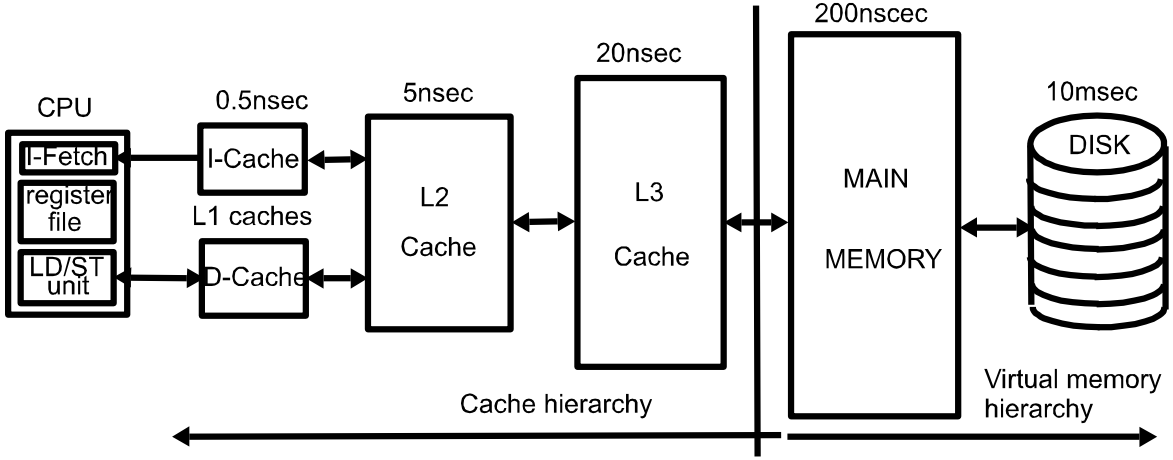


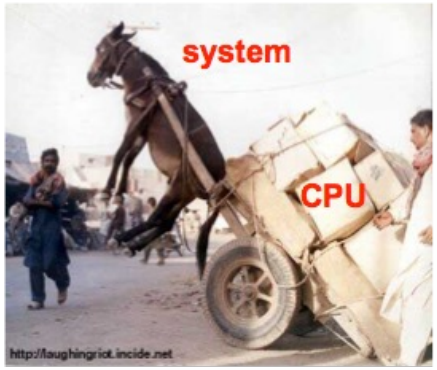
- Memory consistency model defines what it means to “read a memory”, “write at a memory”, and the ”respective ordering”

Today's Class

- Memory consistency problem
- Difference between coherence and consistency
- Sequential consistency
- X86 TSO memory model

Typical Memory Hierarchy

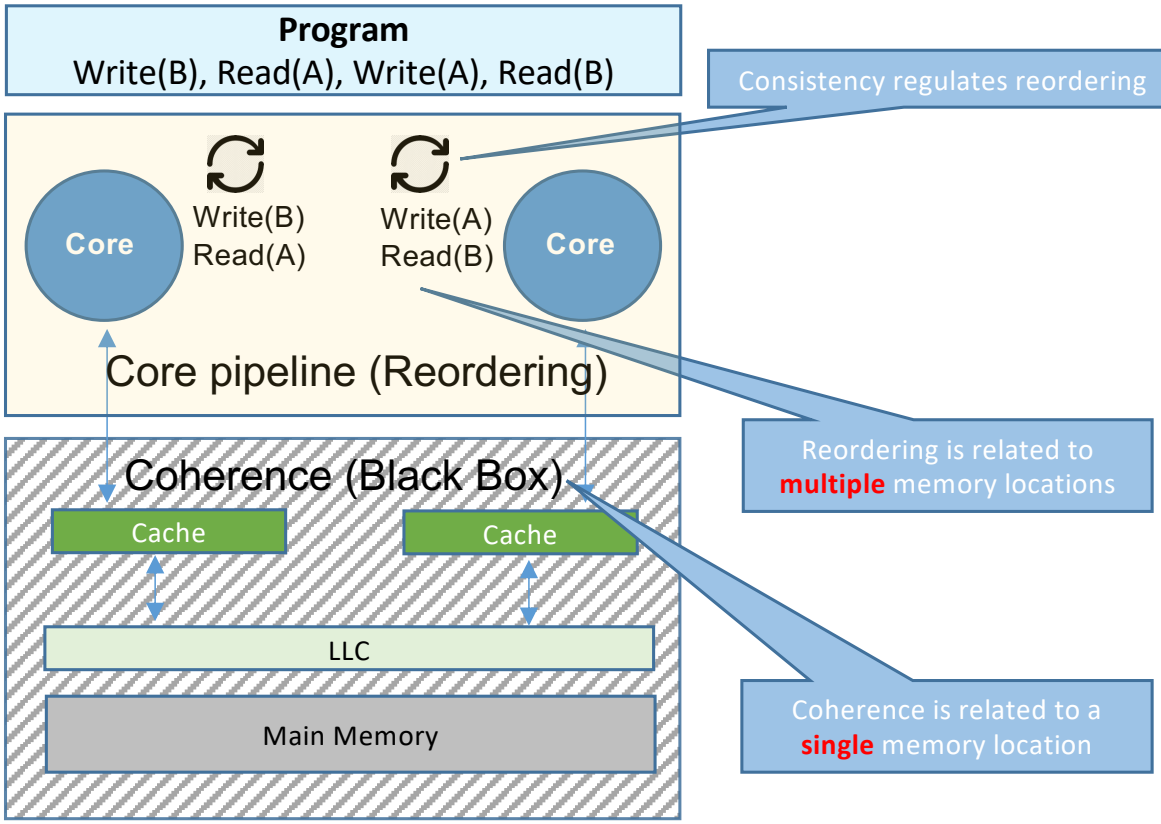




- Memory latency continues to limit the performance of modern out-of-order cores
  - Loads and Stores are very expensive

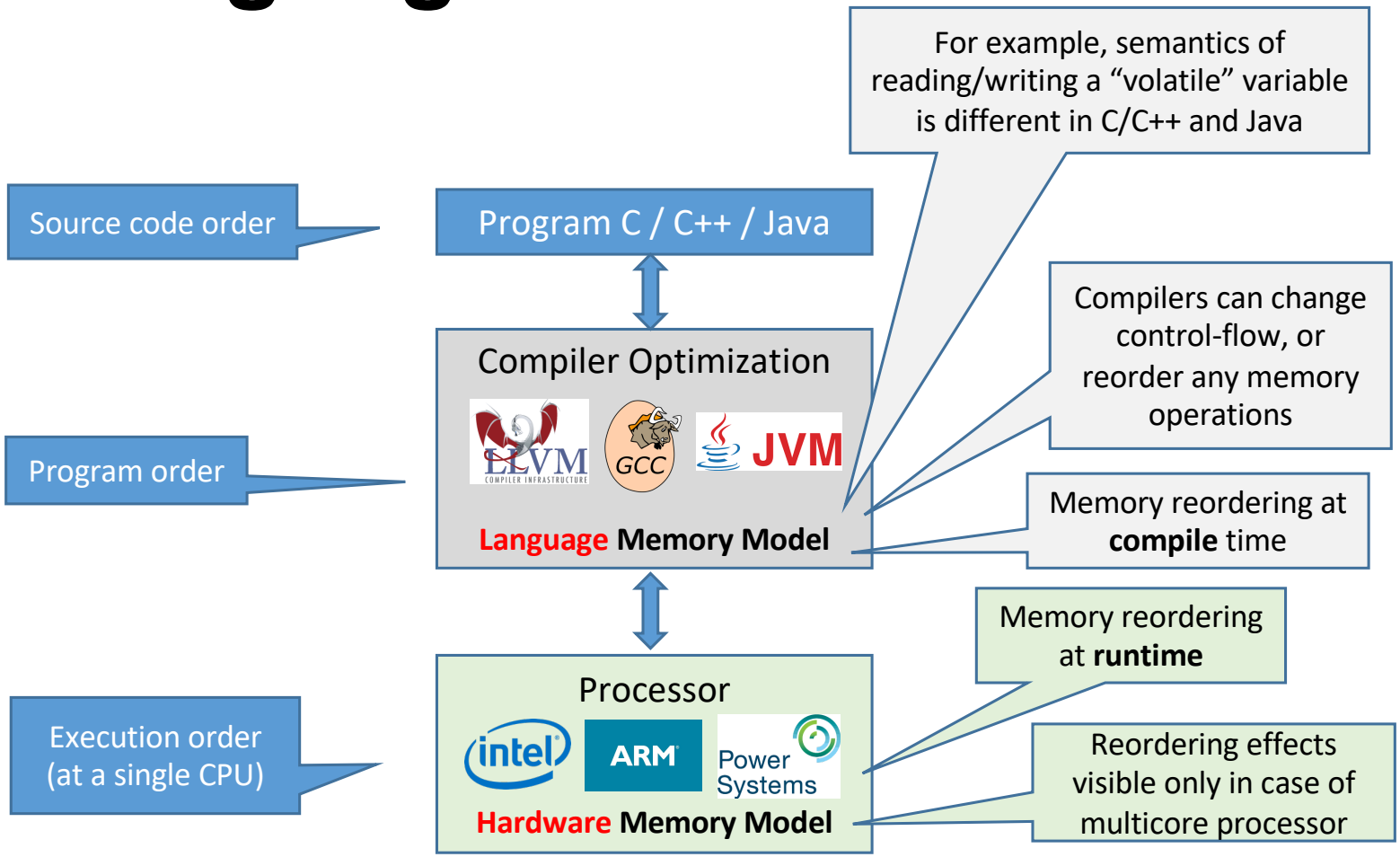
- Hiding Read (Load) latency
  - Prefetching loads that are delayed due to consistency model (e.g., prefetching data for future iterations of a for loop)
  - Out-of-order execution of loads
    - Speculative execution to allow the CPUs to proceed even though consistency model requires memory accesses be delayed (e.g., Intel, AMD)
      - CPUs speculate on instruction execution, but they rollback if speculation is incorrect
- Hiding Write (Store) latency
  - Store Buffers – available on almost all modern processors (discussed later)

Coherence VS. Consistency



- Coherence assures that value written to a memory location by one core is made visible to other cores (value propagation)
- However, coherence says nothing about when writes will become visible
  - It may not be instantly
- Consistency insures that writes to different locations will be seen in an order that makes sense, given the source code

Language v/s Hardware Memory Model



Memory reordering happens at two different levels

Today's Class

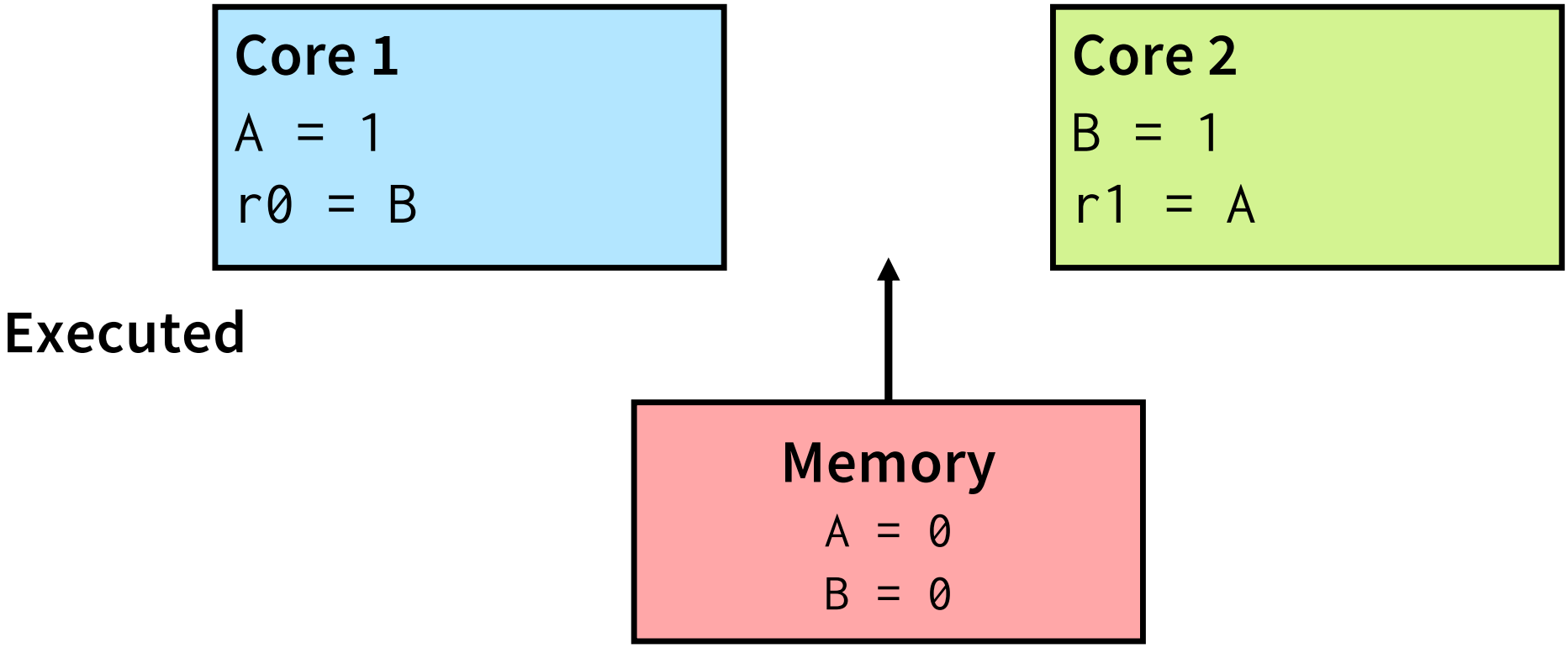
- Memory consistency problem
- Difference between coherence and consistency
- Sequential consistency
- X86 TSO memory model

Sequential Consistency

- Programmer's productivity is high if the memory operations in his parallel program executes in the source code order
  - Sequential consistency
    - Lamport defined SC (1979)
- Let's assume there is a processor that supports sequential consistency to provide high productivity for the programmer
  - Although it suffers on performance
- What that processor has to do for supporting sequential consistency?

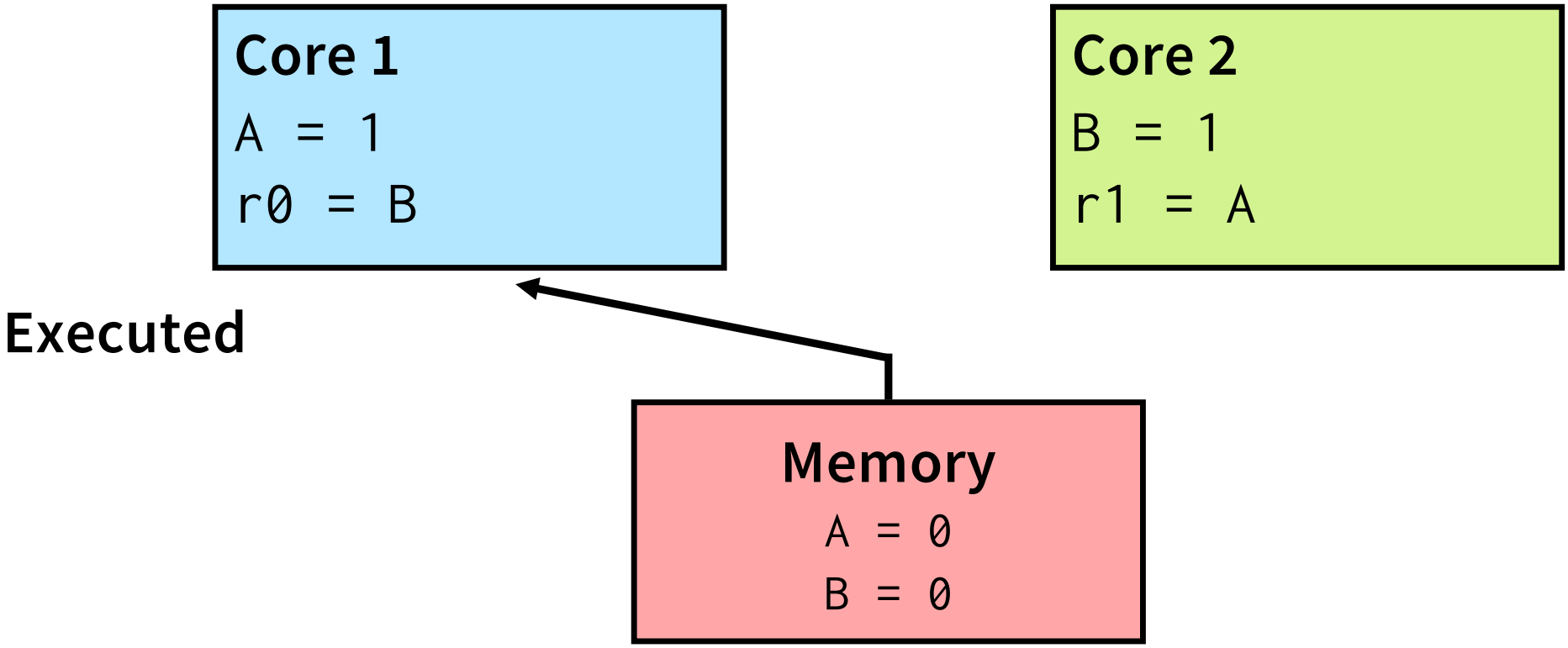
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



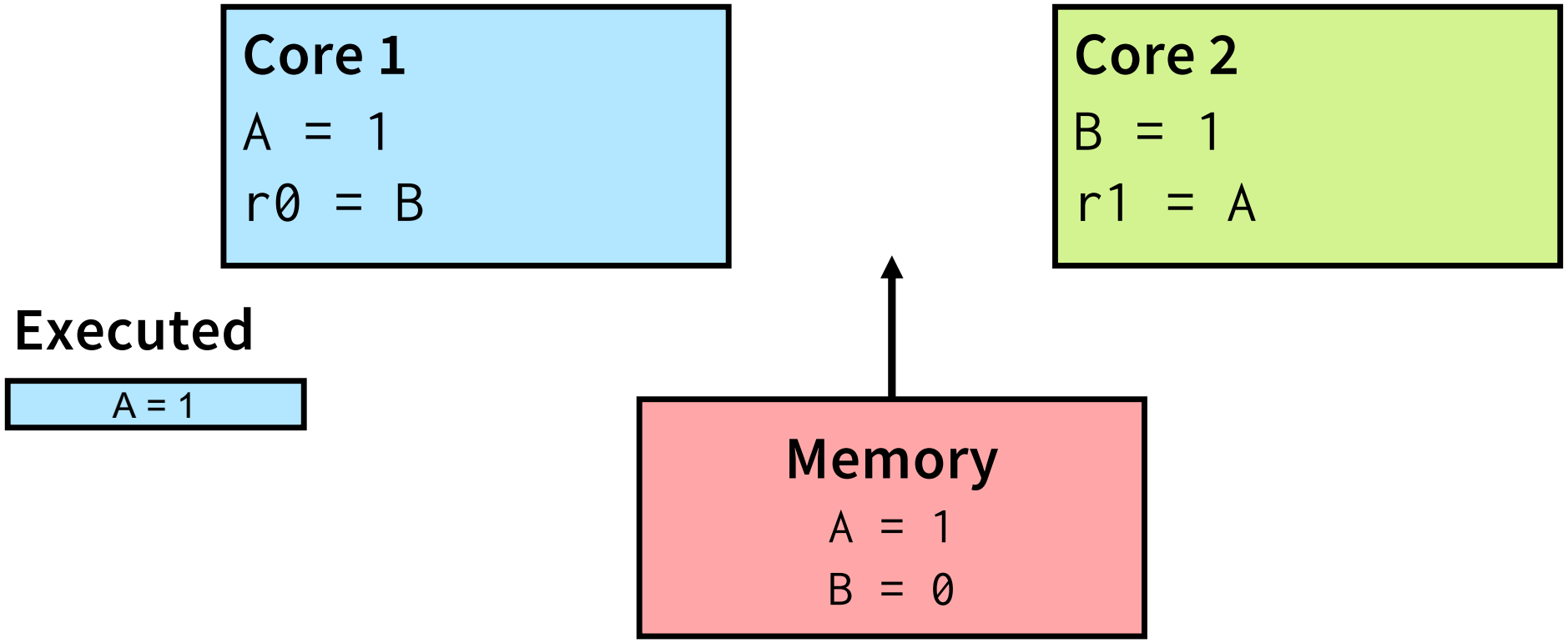
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



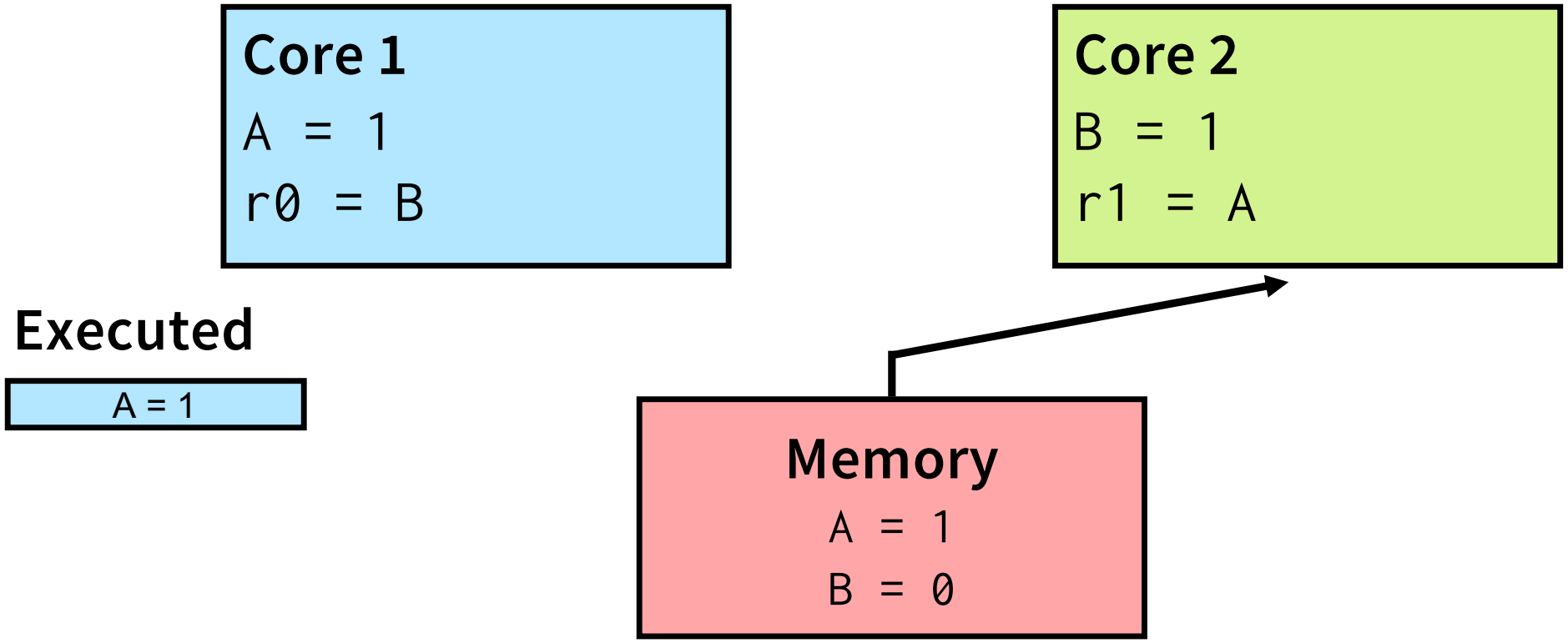
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



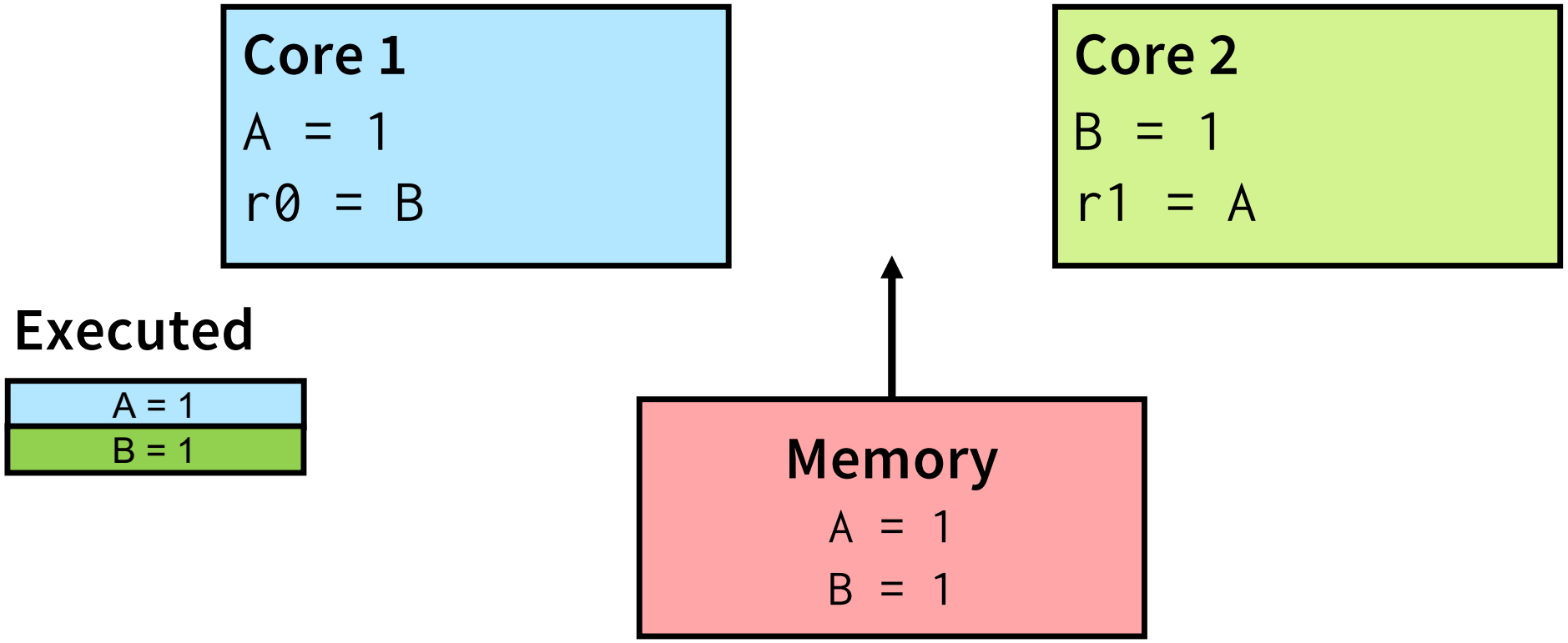
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



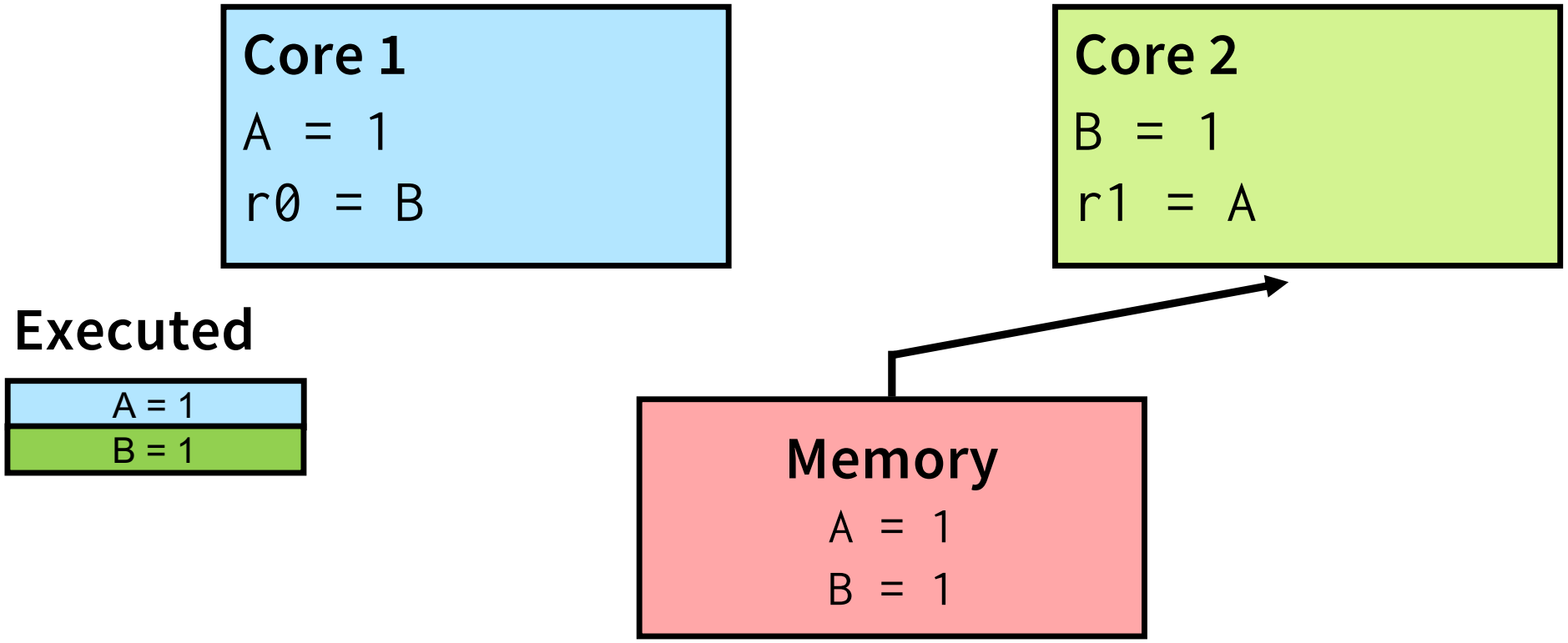
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



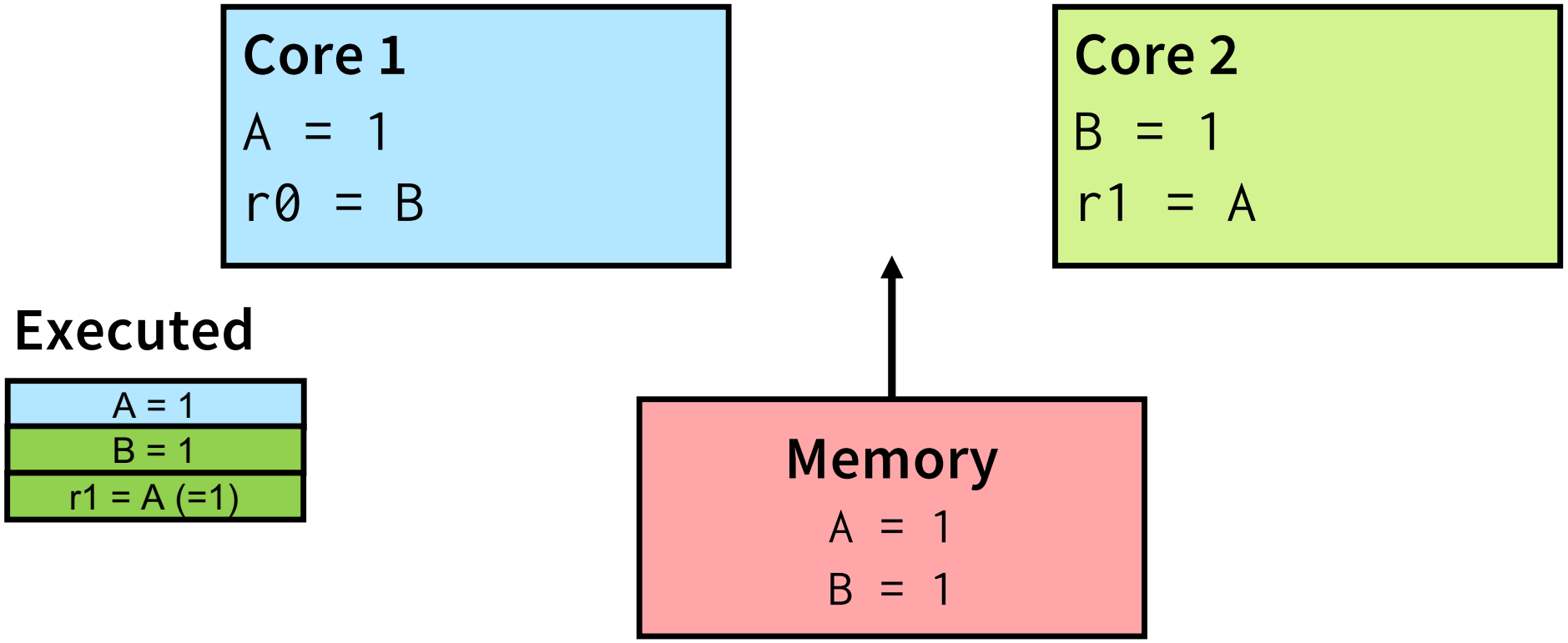
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



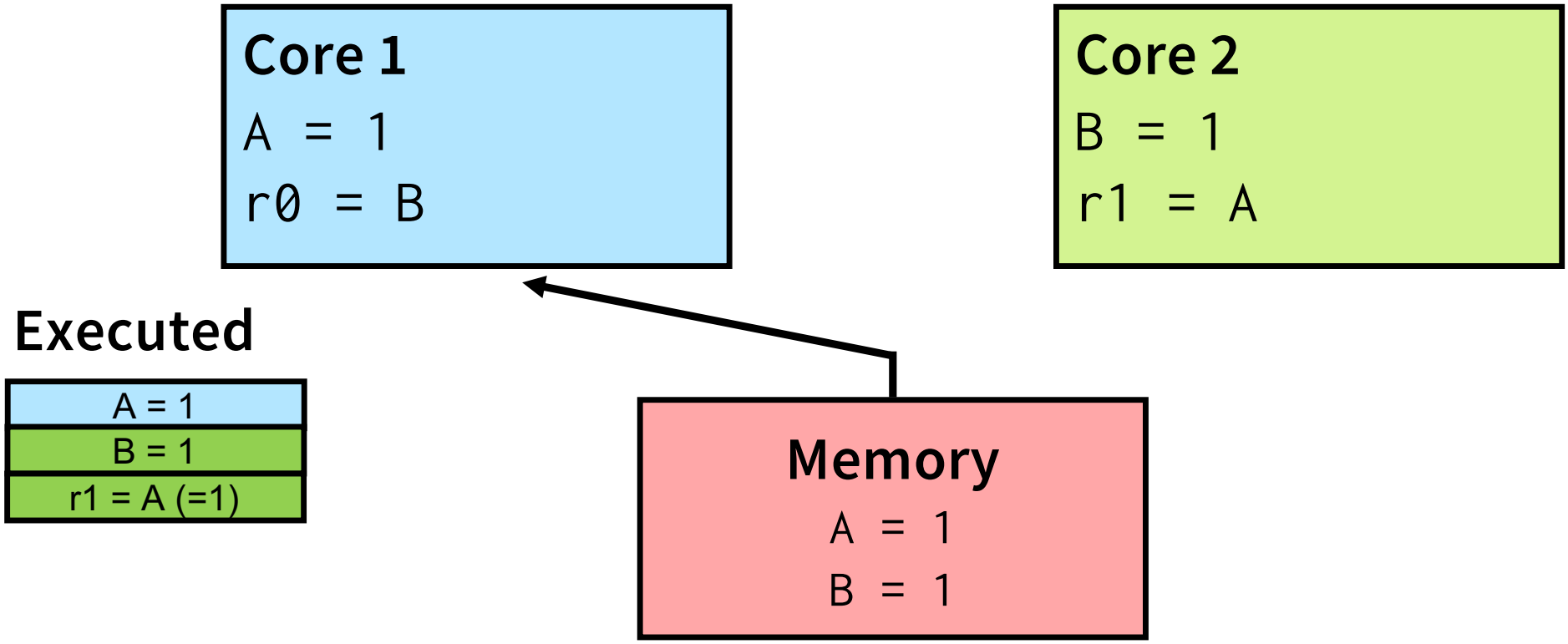
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



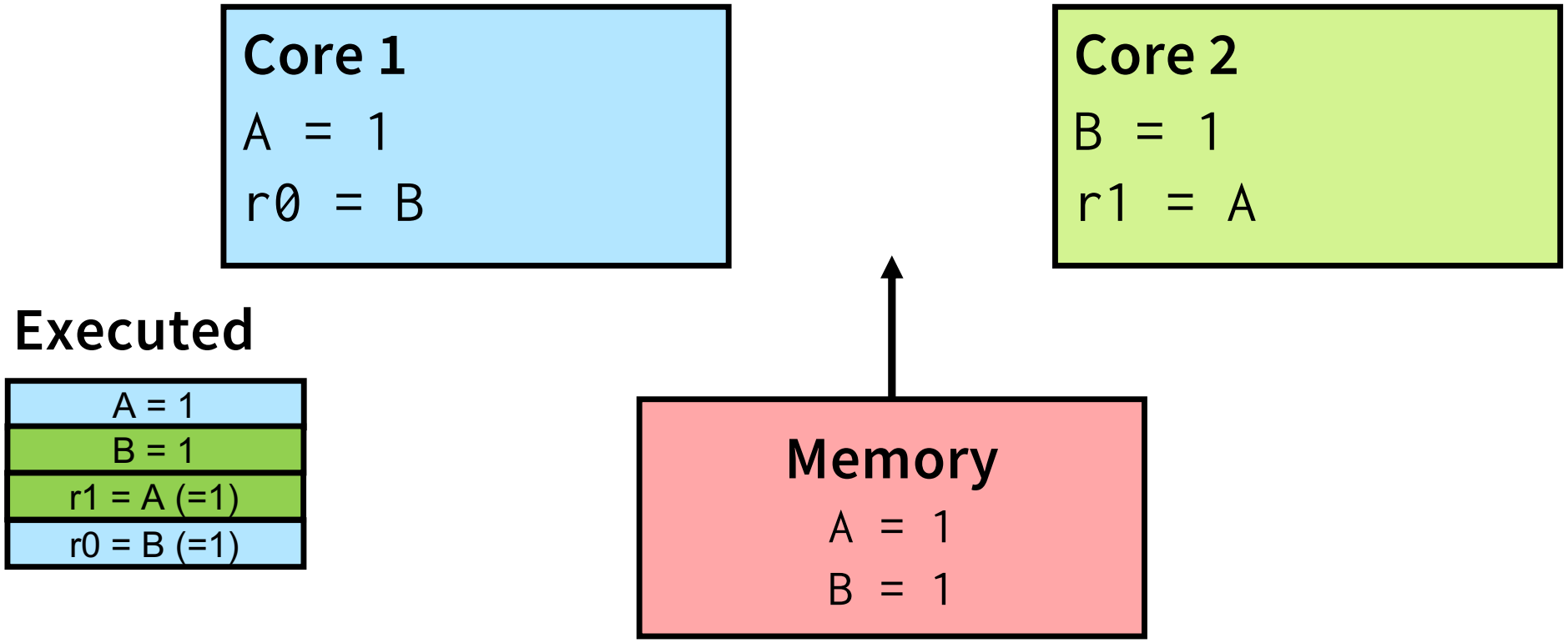
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time



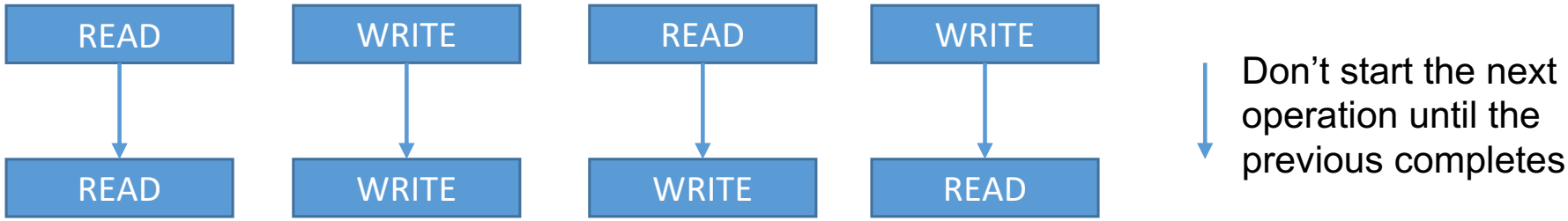
Sequential Consistency

Can be seen as a ”switch” running one instruction at a time

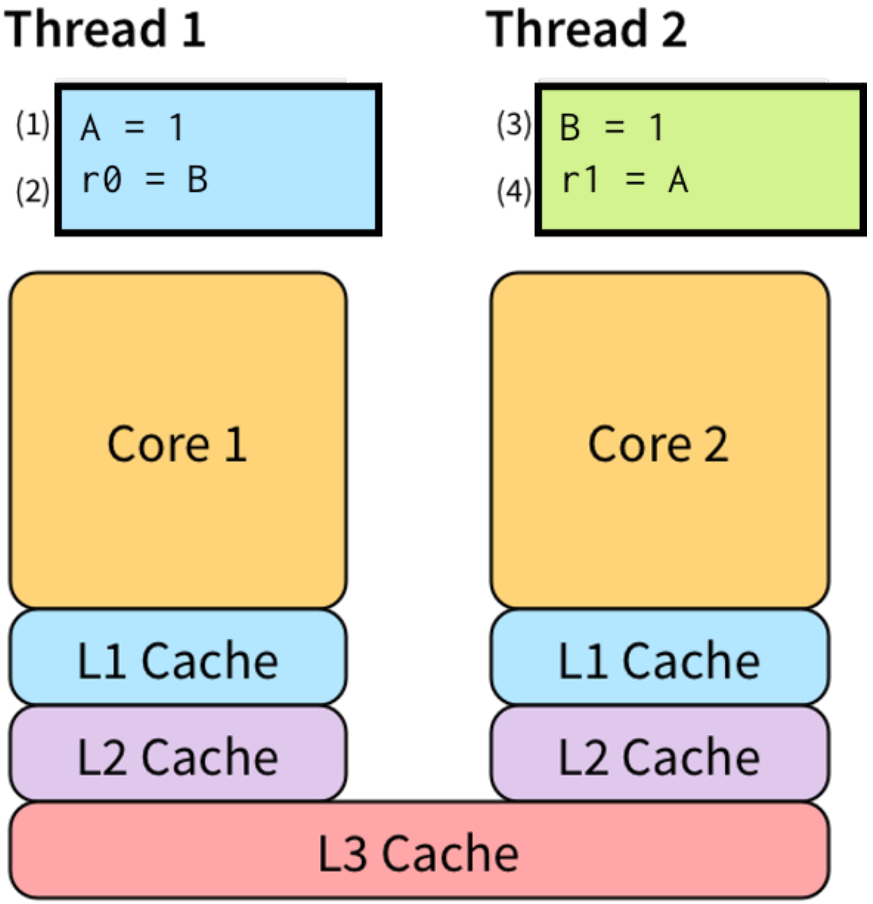


Summary: Sequential Consistency

- Each core execute its read/write in the program order
  - Irrespective of whether they are at same or different addresses
- Every read from address A gets its value from the last write before it to the same address A
  - Operations must appear atomic—one operation must complete before the next one is issued



The Problem with SC



- With a single view of memory, we can't run (2) until (1) has become visible to every other thread
  - On a modern CPU, that's a very expensive operation due to the cache hierarchy

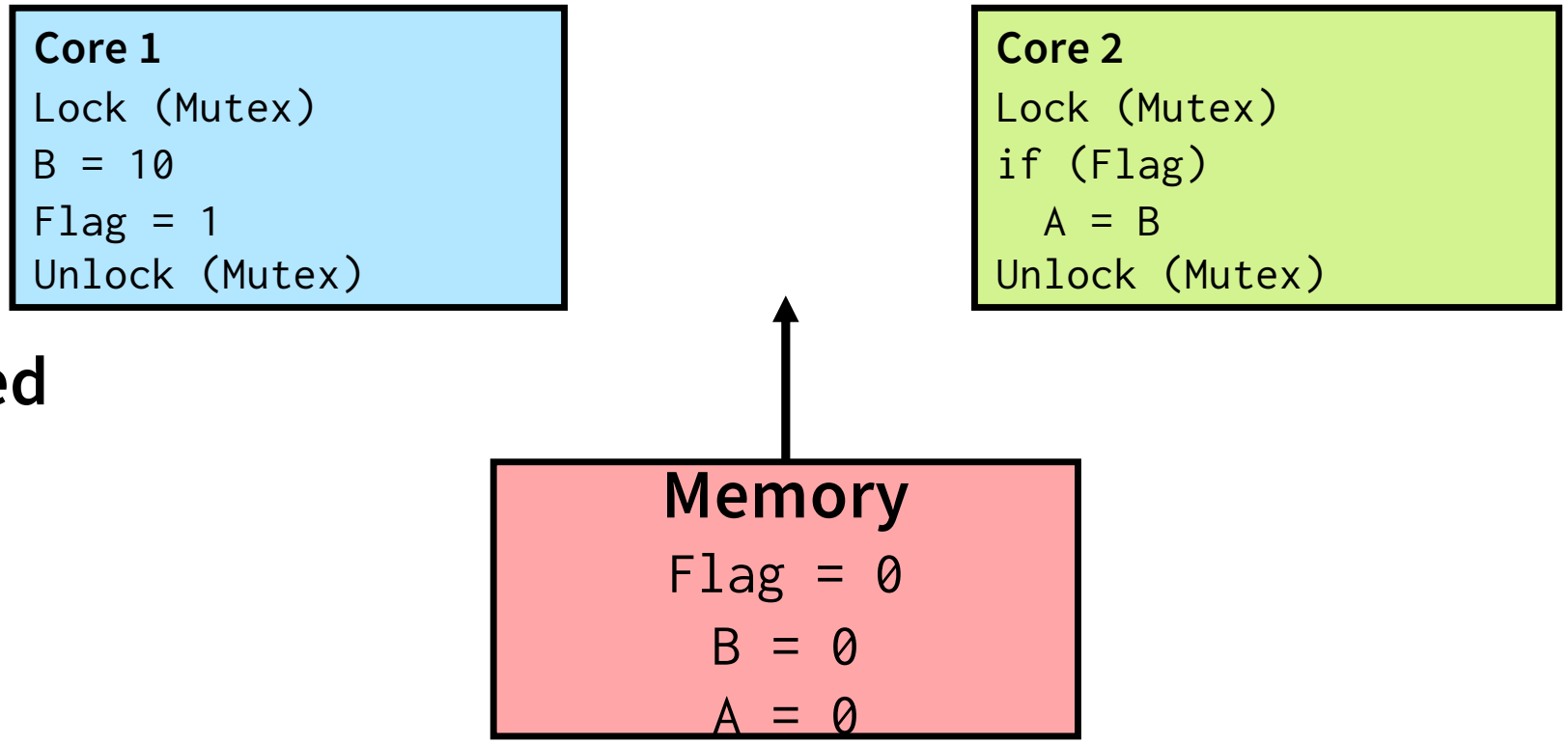
Sequential Consistency Supported!

- Is there support for sequential consistency in modern programming language?
  - Yes, **but** only for **one** particular case in almost every language!
    - Only for code block that is Data Race Free (DRF)
    - Even without using any special construct, you get it for free inside critical sections that are executed within mutex lock/unlock operations
      - We will see in next lecture about other features in C++11 to support DRF
  - No sequential consistency for rest of the program (racy code!)

Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

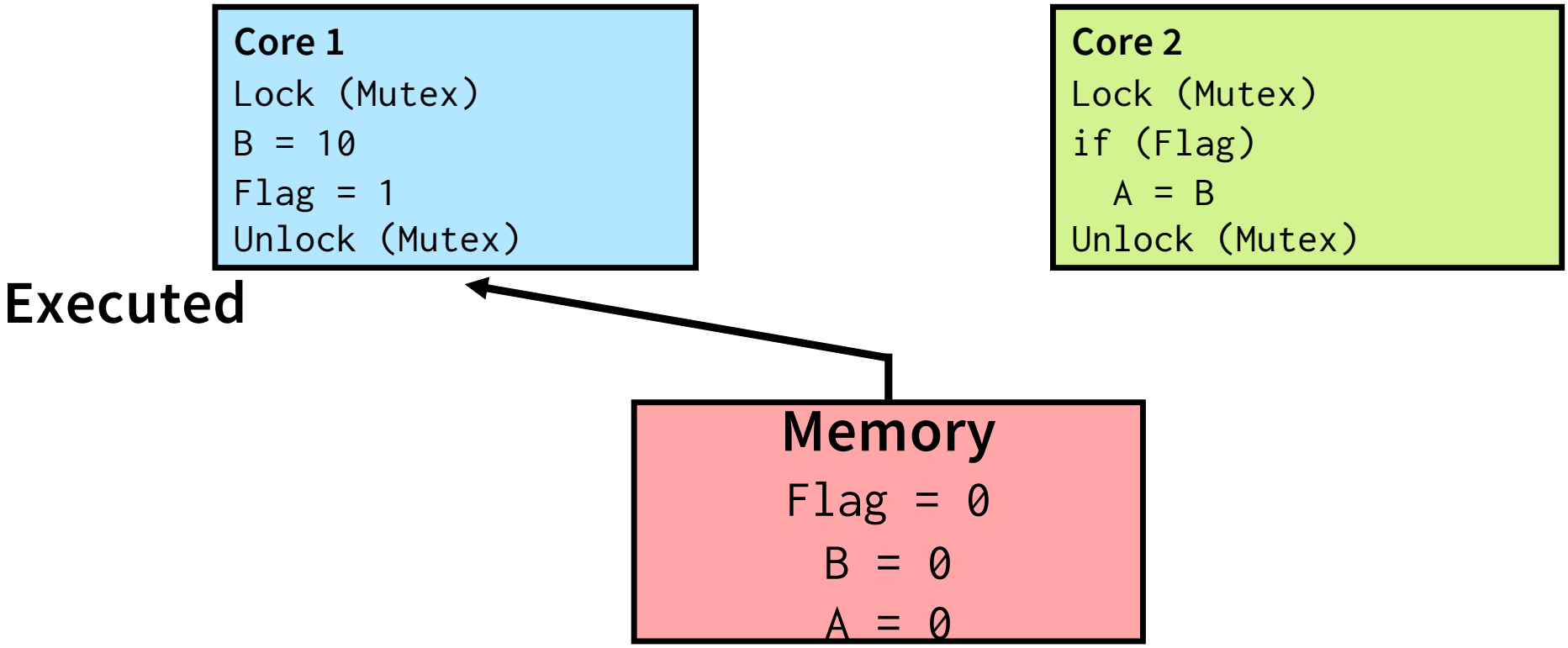
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

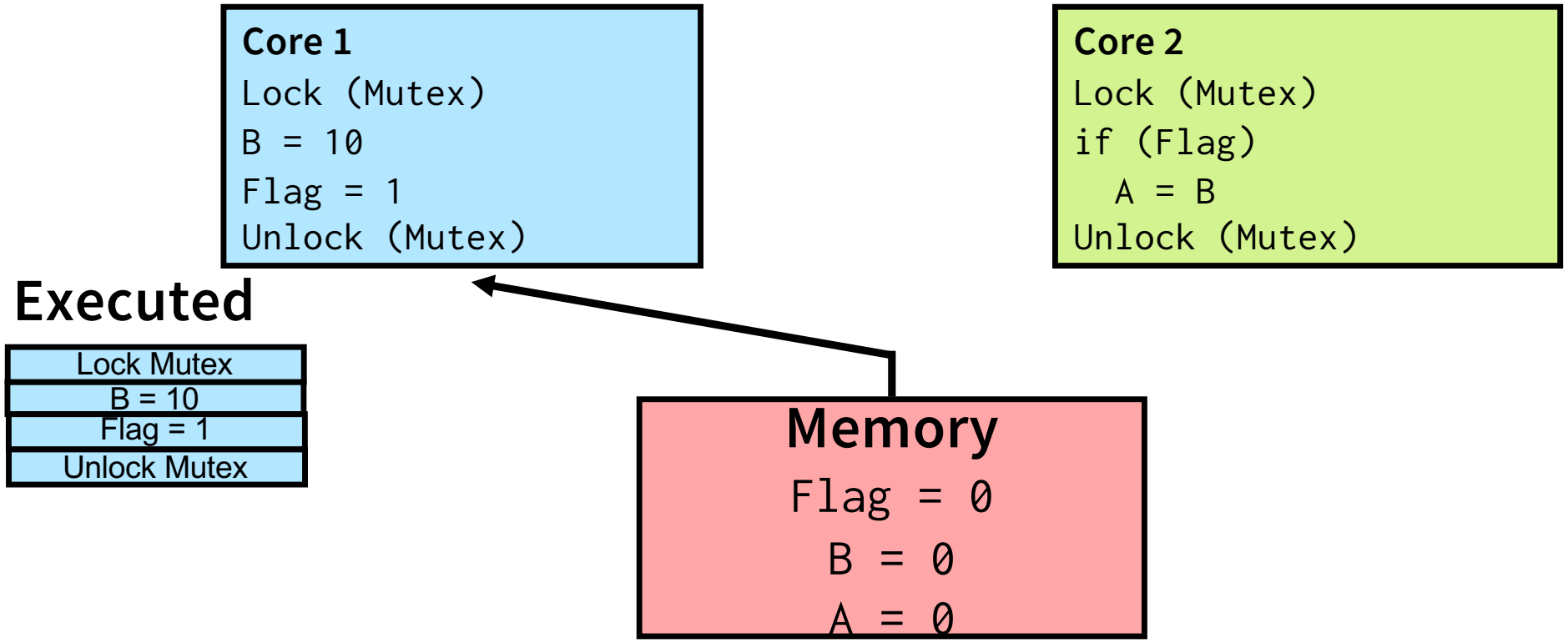
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

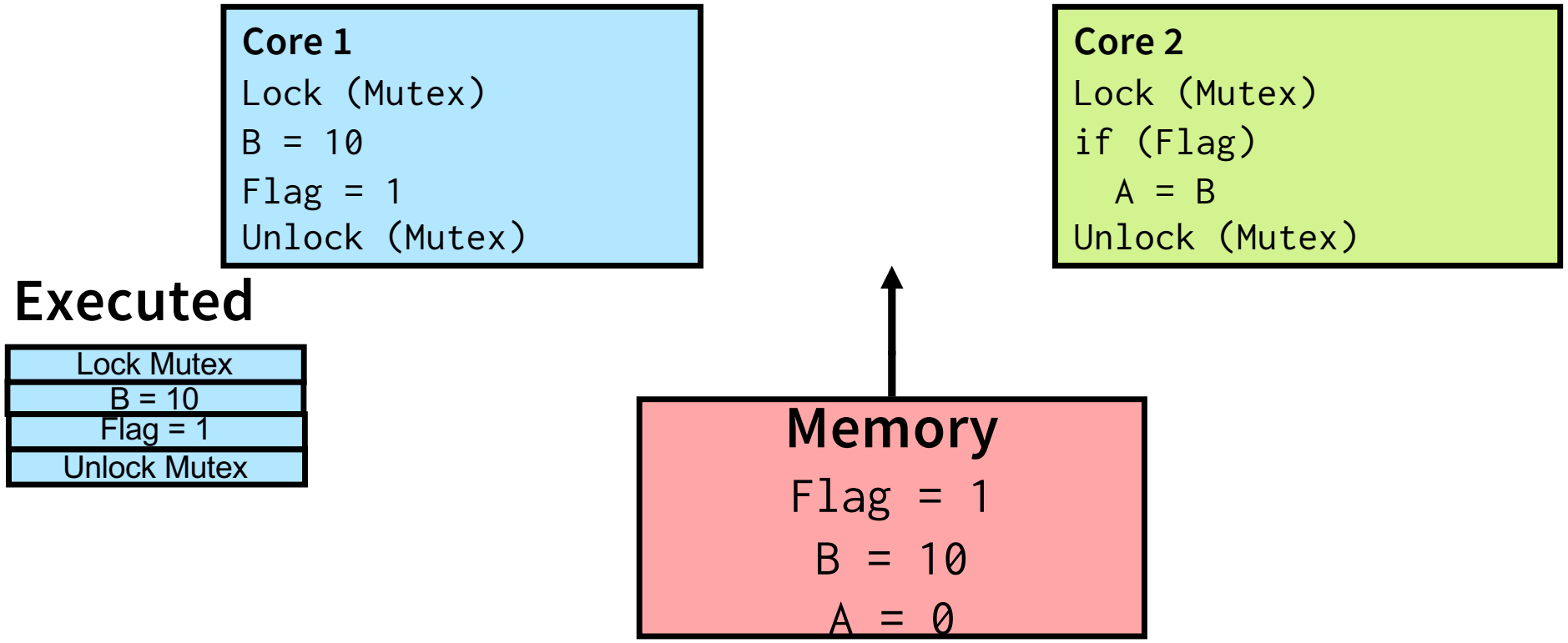
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

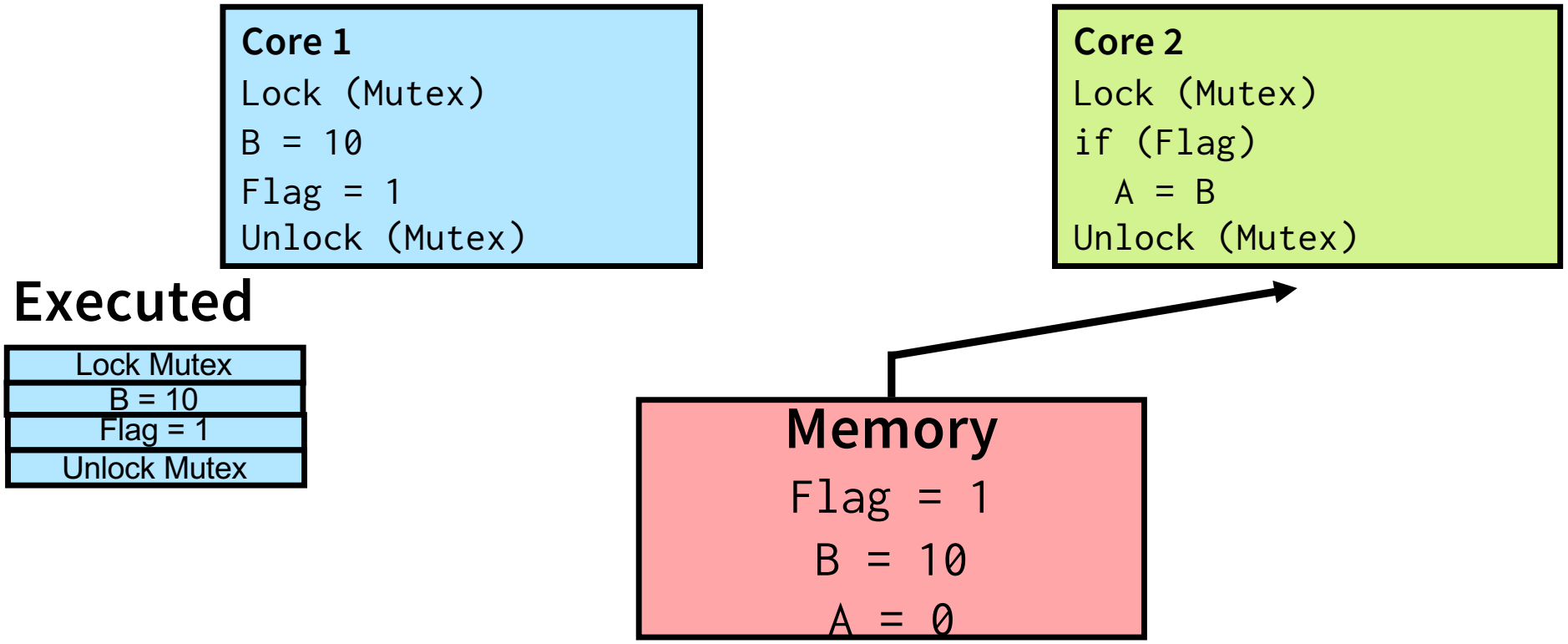
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

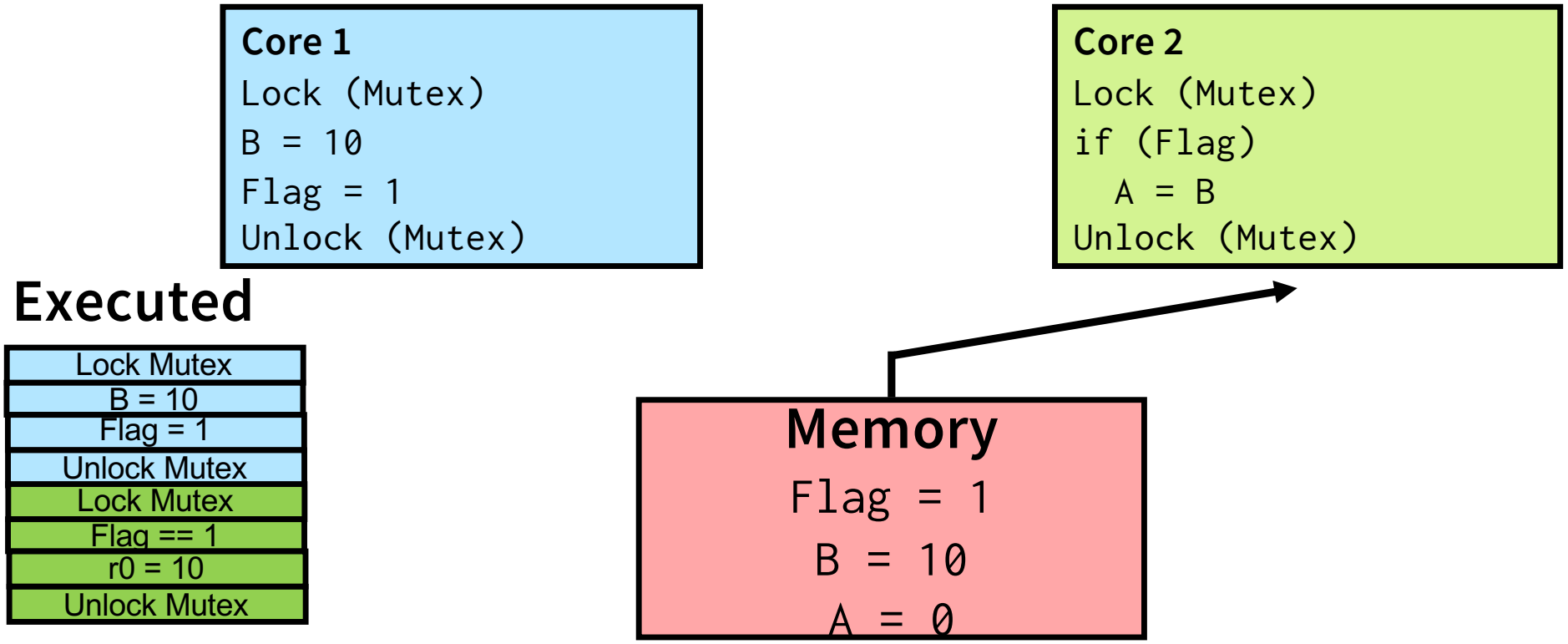
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

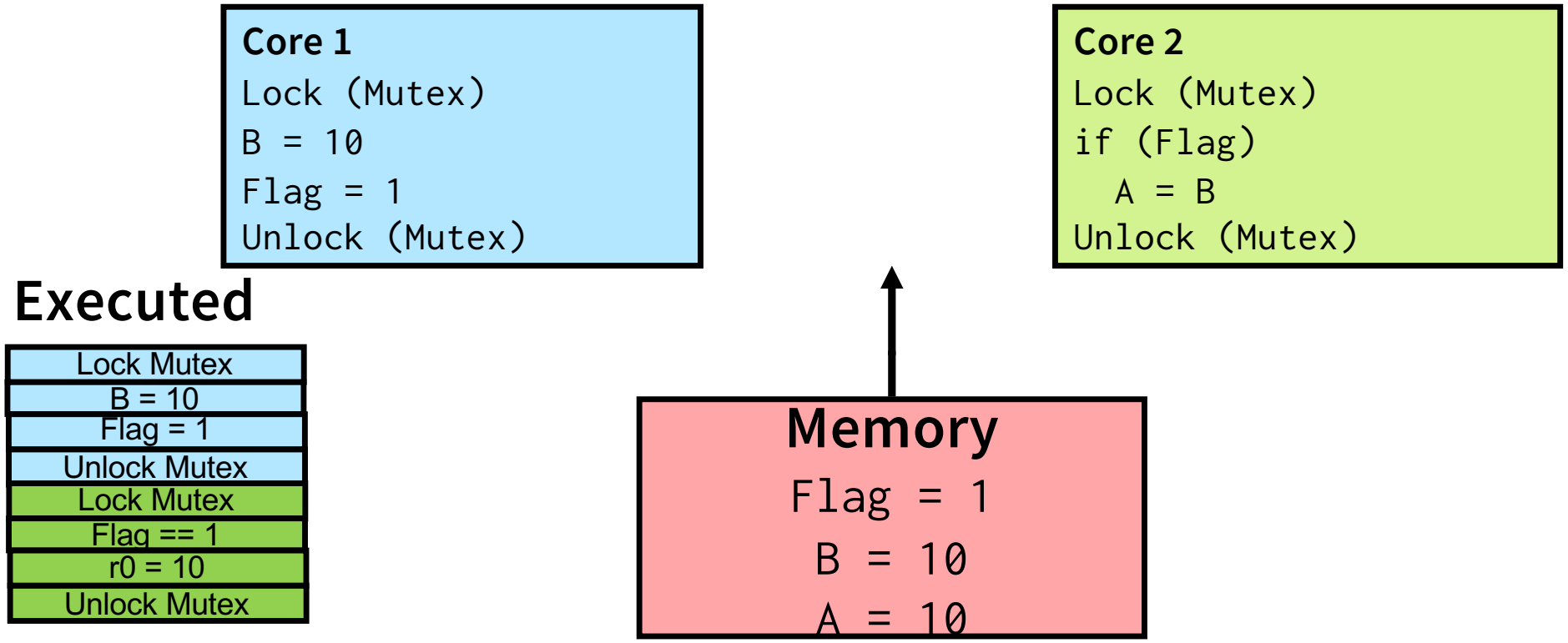
- Can be seen as a ”switch” running one instruction at a time



Sequential Consistency for Data Race Free (DRF)

Cores executing synchronized code blocks

- Can be seen as a ”switch” running one instruction at a time

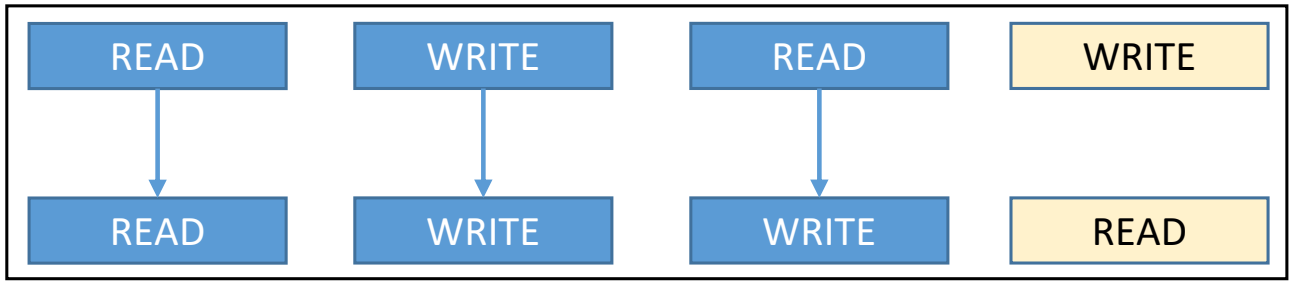


Today's Class

- Memory consistency problem
- Difference between coherence and consistency
- Sequential consistency
- X86 TSO memory model

X86-TSO (Total Store Order) Intel/AMD

Don't start the next operation until the previous completes



READ & WRITE to **TWO** different memory locations

Store Buffers

Store Buffer [edit]

A store buffer is used when writing to an invalid cache line. Since the write will proceed anyway, the CPU issues a read-invalid message (hence the cache line in question and all other CPUs' cache lines that store that memory address are invalidated) and then pushes the write into the store buffer, to be executed when the cache line finally arrives in the cache.

A direct consequence of the store buffer's existence is that when a CPU commits a write, that write is not immediately written in the cache. Therefore, whenever a CPU needs to read a cache line, it first has to scan its own store buffer for the existence of the same line, as there is a possibility that the same line was written by the same CPU before but hasn't yet been written in the cache (the preceding write is still waiting in the store buffer). Note that while a CPU can read its own previous writes in its store buffer, other CPUs *cannot see those writes* before they are flushed from the store buffer to the cache - a CPU cannot scan the store buffer of other CPUs.

From Wikipedia

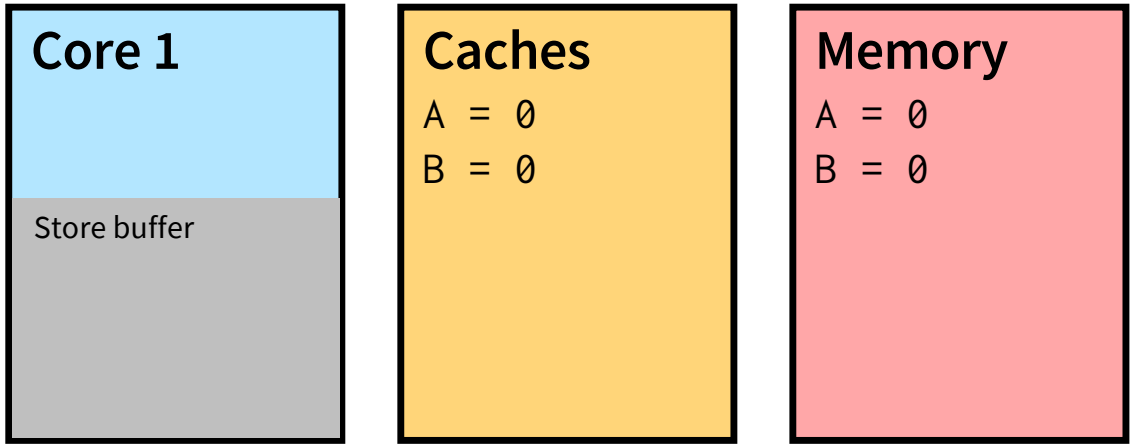
MESI protocol. (2022, May 2). In *Wikipedia*. https://en.wikipedia.org/wiki/MESI_protocol

Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic

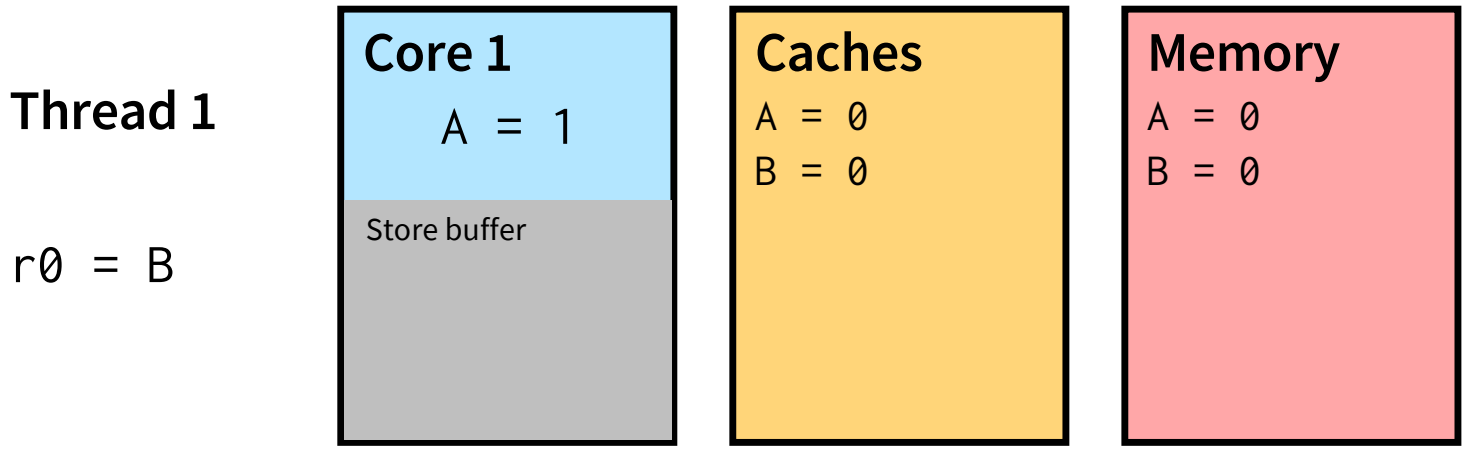
Thread 1

```
A = 1
r0 = B
```



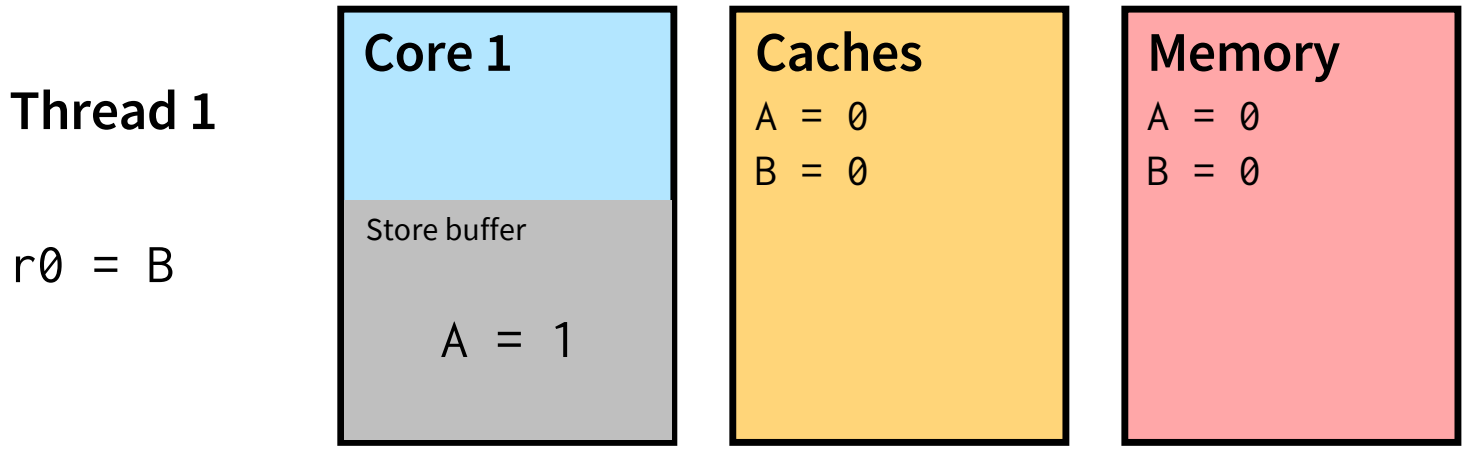
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



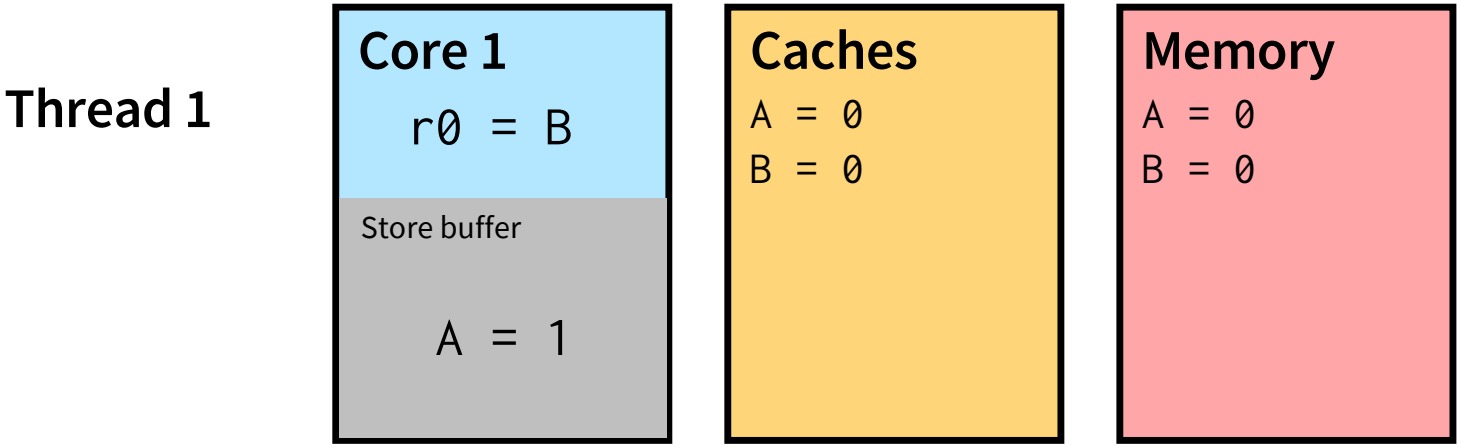
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic

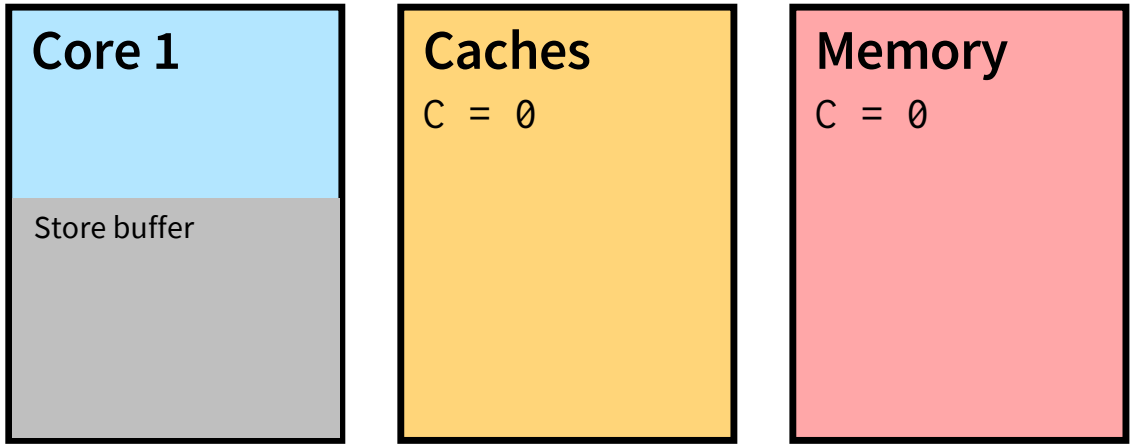


Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic

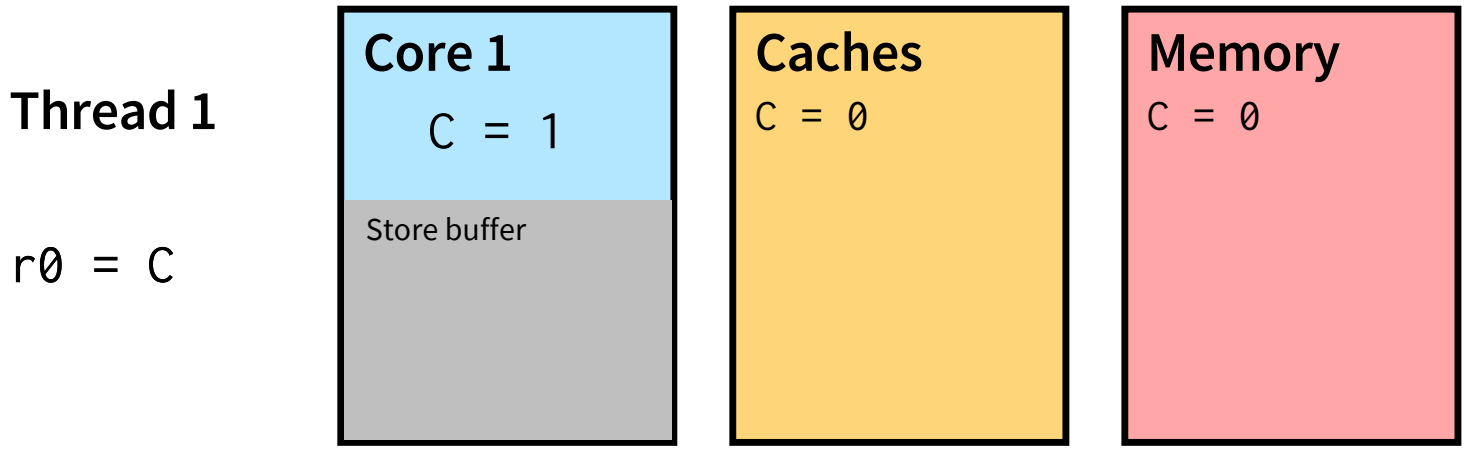
Thread 1

```
C = 1
r0 = C
```



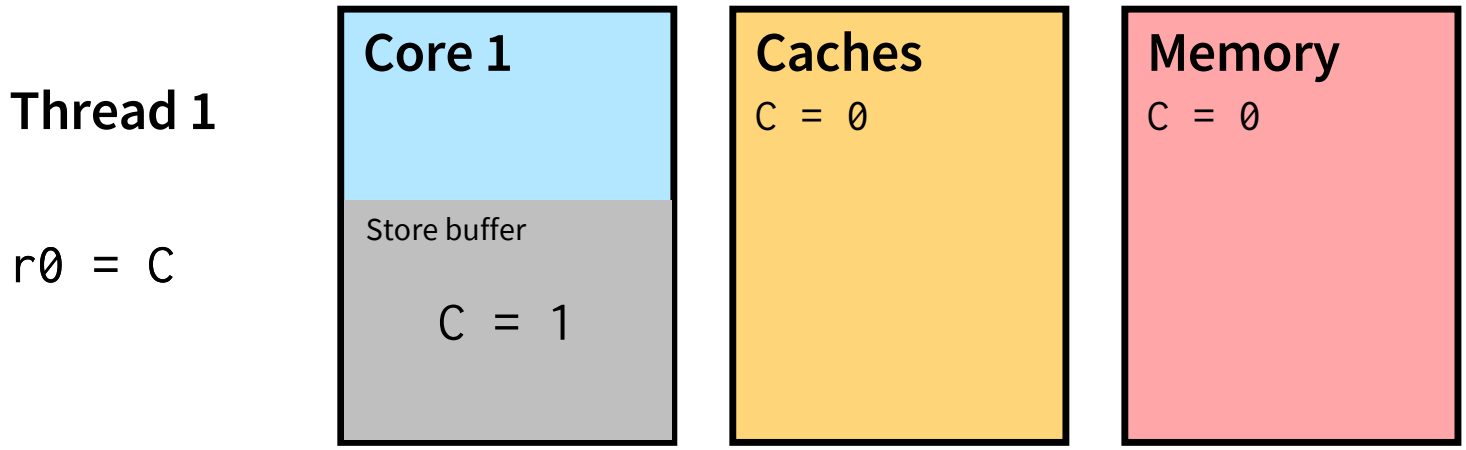
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



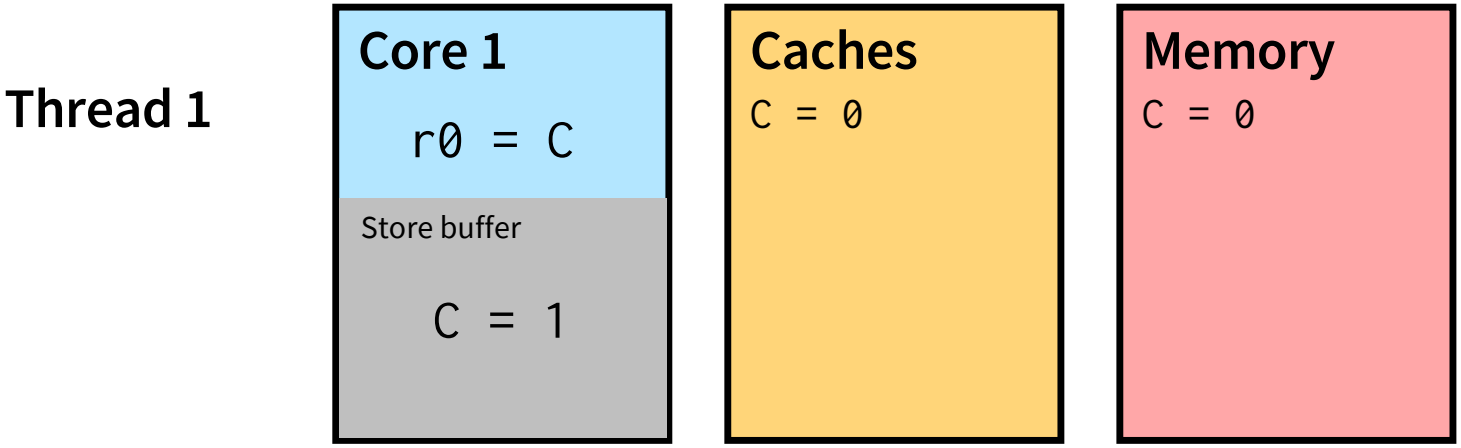
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



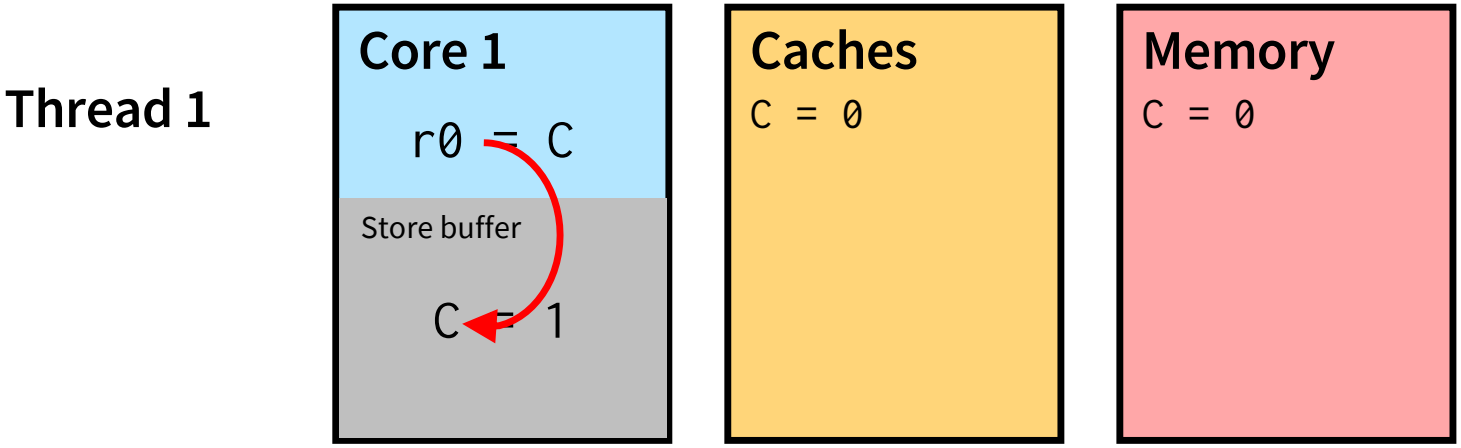
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



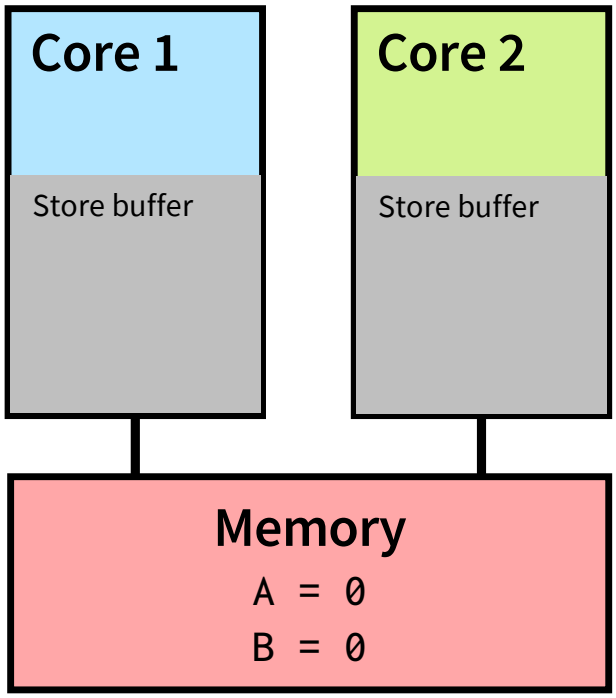
Store Buffers

- Store writes in a local buffer and then proceed to next instruction immediately
  - Absorbs writes faster than the next cache => prevents stalls
- The cache will pull writes out of the store buffer when it's ready
  - Aggregates writes to same cache line => reduces cache traffic



x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

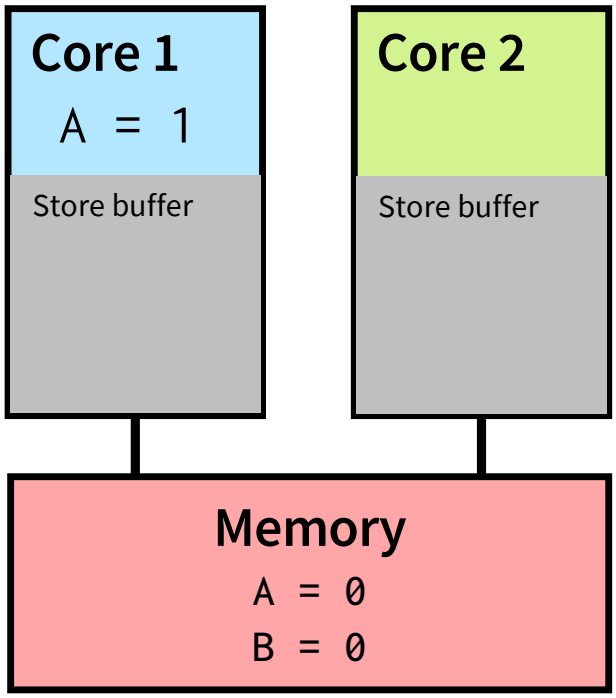


Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

Can `r0 = 0` and `r1 = 0`?

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

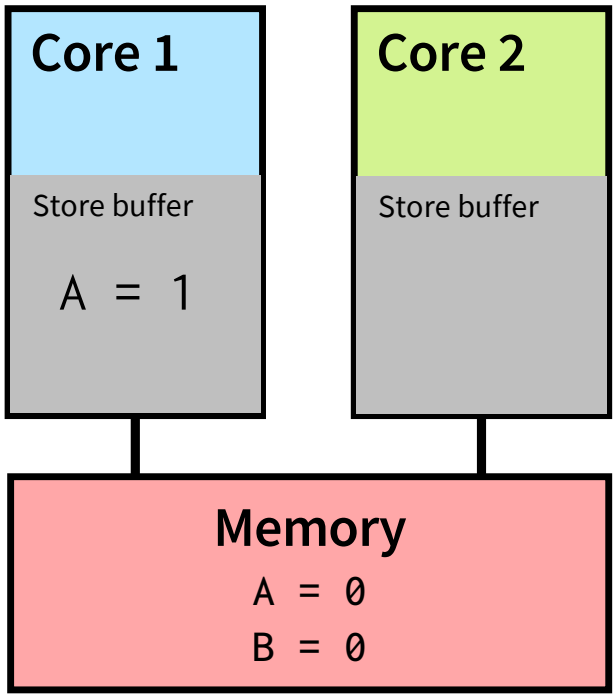


Thread 1	Thread 2
(1)	(3) B = 1
(2) r0 = B	(4) r1 = A

Can r0 = 0 and r1 = 0?

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

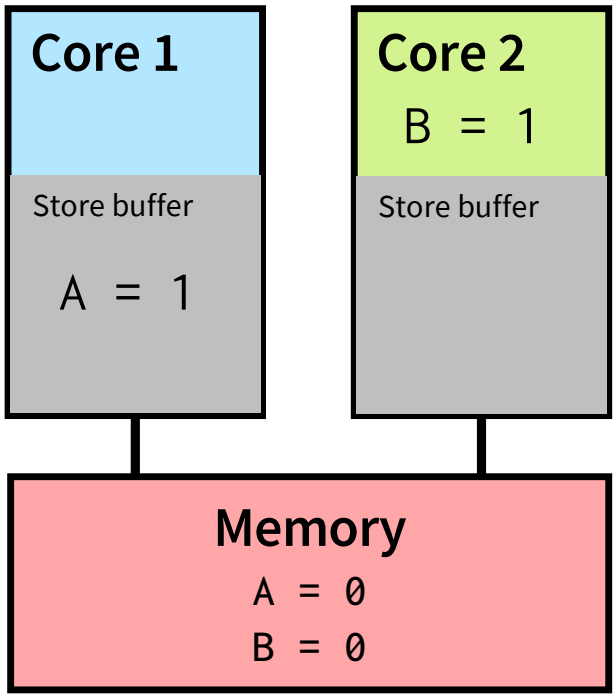


Thread 1	Thread 2
(1)	(3) B = 1
(2) r0 = B	(4) r1 = A

Can r0 = 0 and r1 = 0?

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

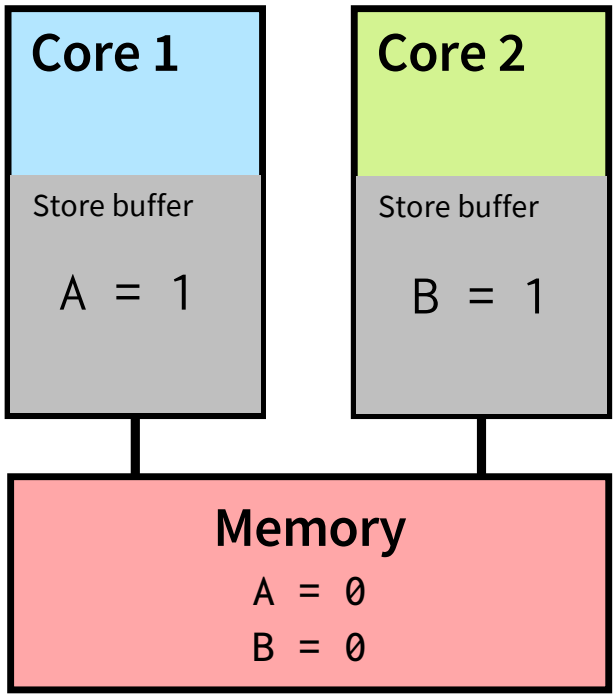


Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

Can `r0 = 0` and `r1 = 0`?

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

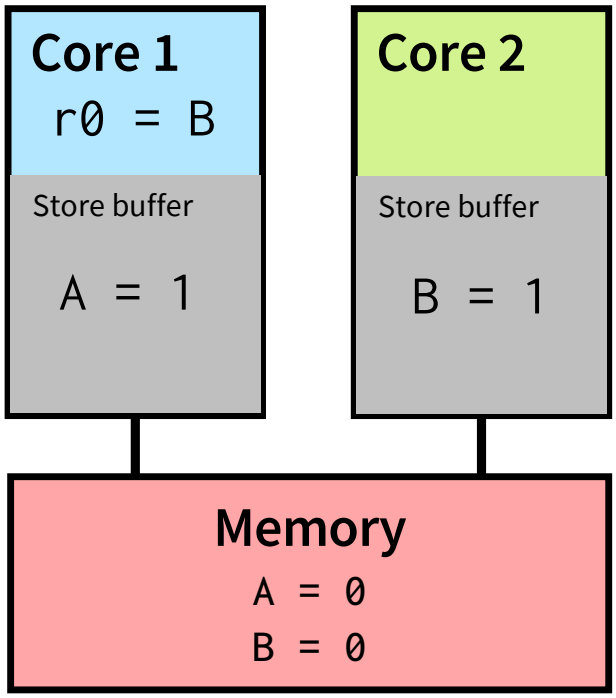


Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

Can `r0 = 0` and `r1 = 0`?

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Thread 1	Thread 2
(1)	(3)
(2)	(4) r1 = A

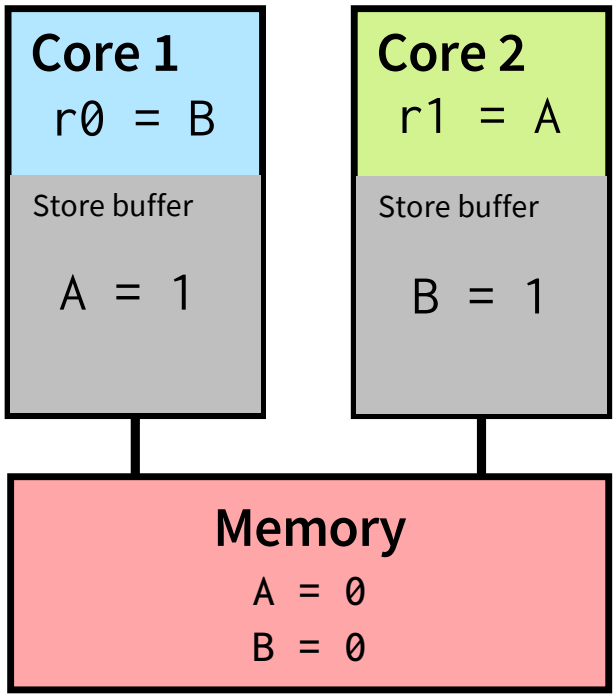
Can r0 = 0 and r1 = 0?

Executed

```
r0 = B (= 0)
```

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Thread 1	Thread 2
(1)	(3)
(2)	(4)

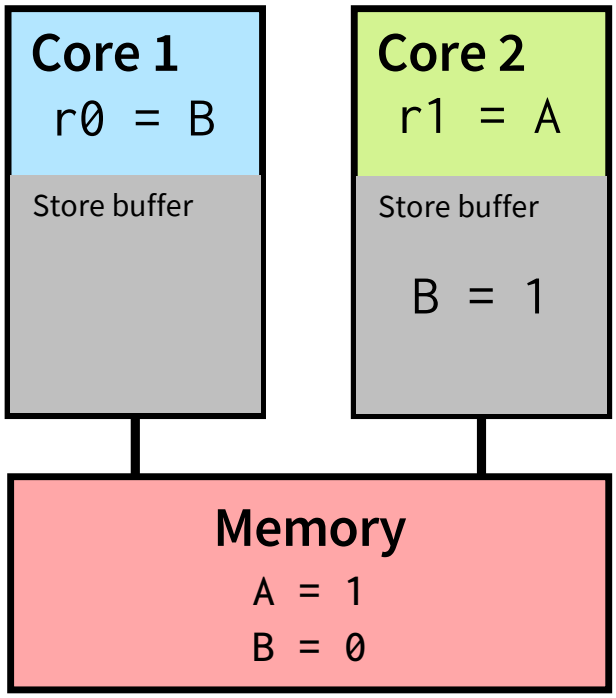
Can `r0 = 0` and `r1 = 0`?

Executed

```
r0 = B (= 0)
r1 = A (= 0)
```

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Thread 1	Thread 2
(1)	(3)
(2)	(4)

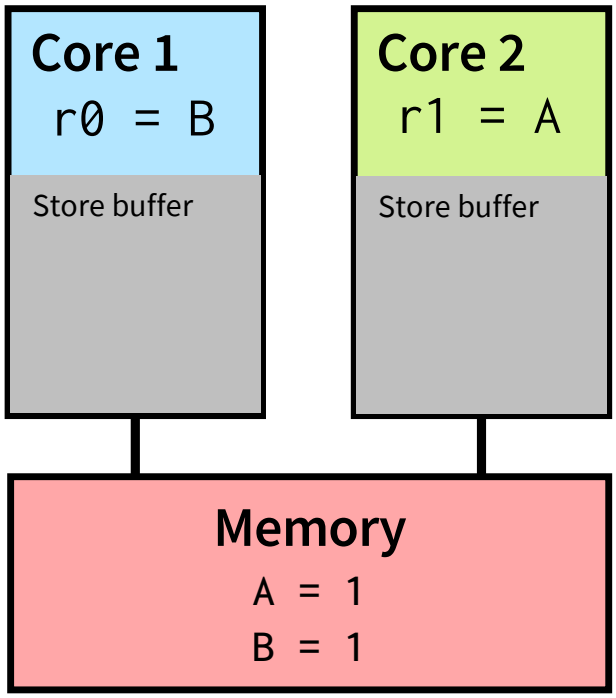
Can `r0 = 0` and `r1 = 0`?

Executed

```
r0 = B (= 0)
r1 = A (= 0)
A = 1
```

x86-TSO Behavior

- Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Thread 1	Thread 2
(1)	(3)
(2)	(4)

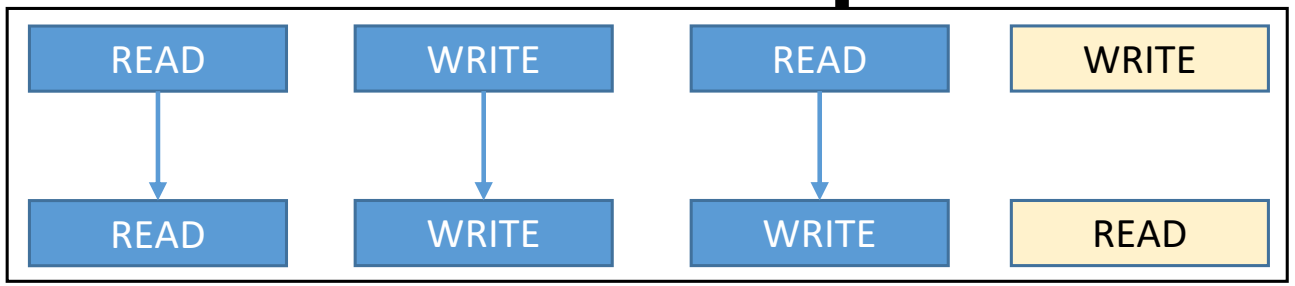
Can `r0 = 0` and `r1 = 0`?

Store buffers: Yes!

Executed

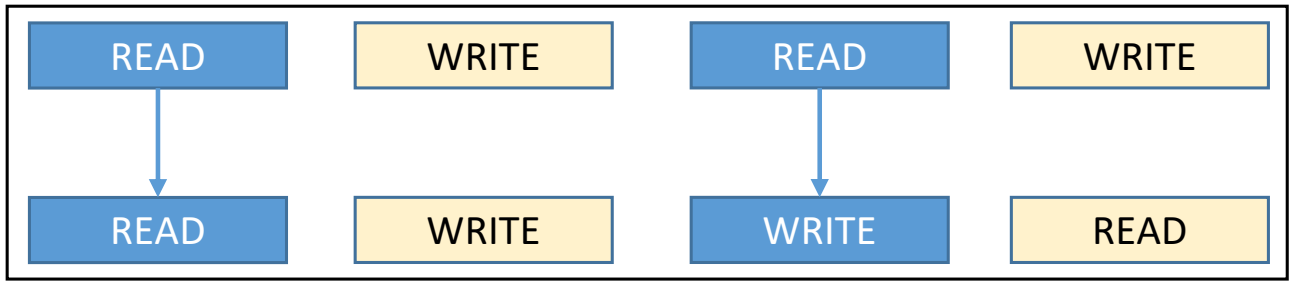
```
r0 = B (= 0)
r1 = A (= 0)
A = 1
B = 1
```

Few Alternatives to Sequential Consistency

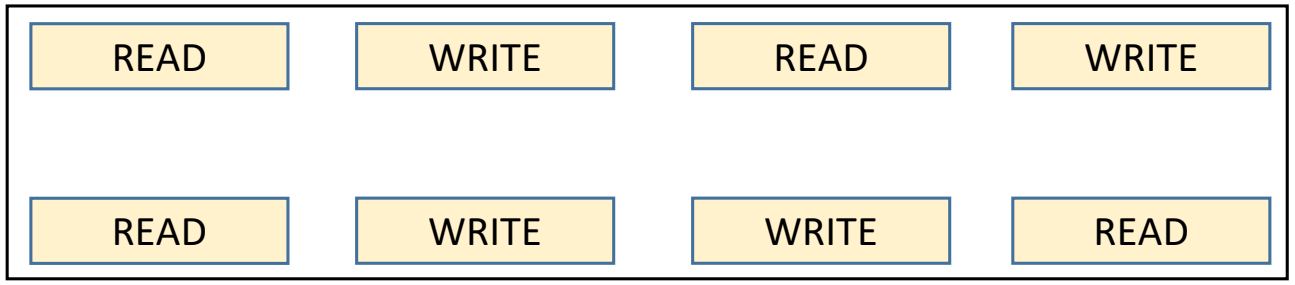


X86-TSO (Total Store Order)
Intel/AMD processors

Don't start the next operation until the previous completes



PSO (Partial Store Order)
SPARC processor



Relaxed Memory Ordering
IBM Power PC, ARM, etc.

Reference Materials

- https://www.youtube.com/watch?v=mDYLRX2pbFw
- Preshing
  - https://preshing.com/20120515/memory-reordering-caught-in-the-act/
- Intel processor manual
  - https://cdrdv2.intel.com/v1/dl/getContent/671200
    - Section 8.2
- Book
  - Sorin et al., A Primer on Memory Consistency and Cache Coherence
  - https://course.ece.cmu.edu/~ece847c/S15/lib/exe/fetch.php?media=part2_2_sorin12.pdf

Next Lecture

- Language memory consistency model