CS614: Advanced Compilers

Memory Models

Manas Thakur

CSE, IIT Bombay

Spring 2025

Multithreaded Programs

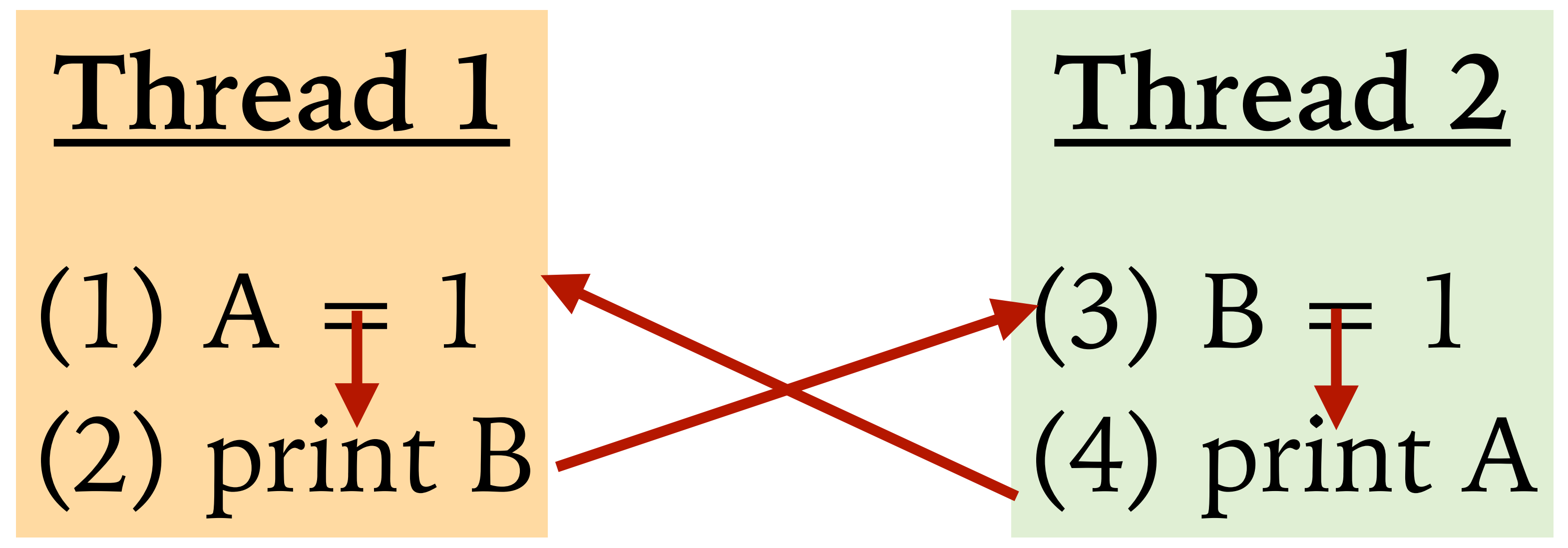
Initially, A = B = 0.

Thread 1	**Thread 2**
(1) A = 1	(3) B = 1
(2) print B	(4) print A

- **What all can be printed?**
  - (1) —> (2) —> (3) —> (4): **01**
  - (3) —> (4) —> (1) —> (2): **01**
  - (1) —> (3) —> (2) —> (4): **11**
  - (1) —> (3) —> (4) —> (2): **11**
  - A few others with the same effect.

Things that shouldn't happen

- It should not be possible to print **00**.



- In order to print **00**:
  - For line (2) to print 0, (2) should ***happen before*** (3) writes 1 to B.
  - For line (4) to print 0, (4) should happen before (1) writes 1 to A.
  - Each thread's events should be in order: (1) before (2) and (3) before (4).
  - This implies (1) should happen before (1)!
  - A contradiction — means **00** cannot be printed by this multithreaded program.

Sequential Consistency

1. All operations executed in some sequential order.
   - ➤ As if they were manipulating a *single shared memory*.
2. Each thread's operations happen in program order.

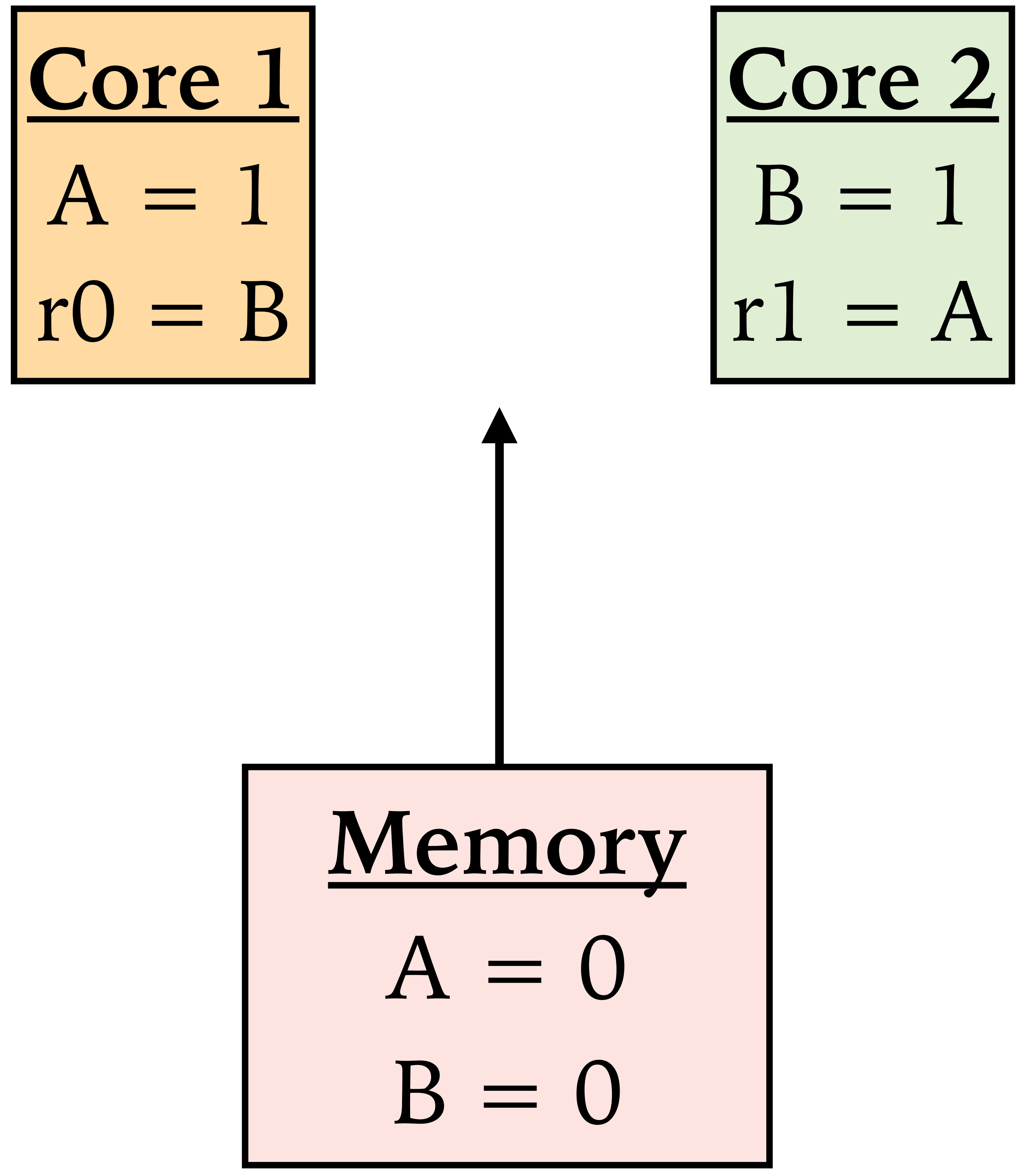
Initially, A = B = 0.

Thread 1	**Thread 2**
A = 1	B = 1
r0 = B	r1 = A

➤ Not allowed: r0 = 0 and r1 = 0.

Sequential Consistency

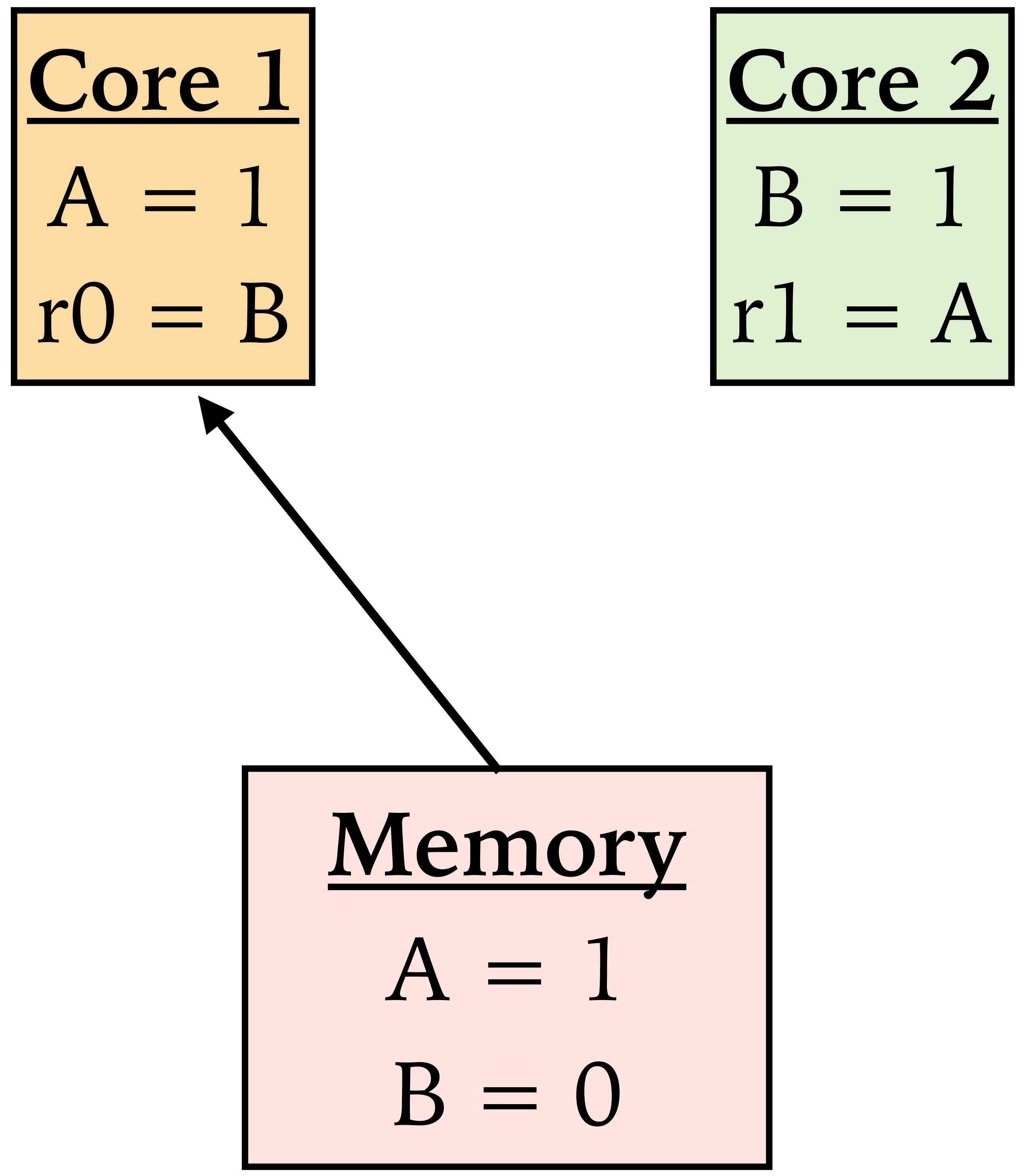
- Can be seen as a "switch" running one instruction at a time from different threads.



Executed

Sequential Consistency

- Can be seen as a “switch” running one instruction at a time from different threads.

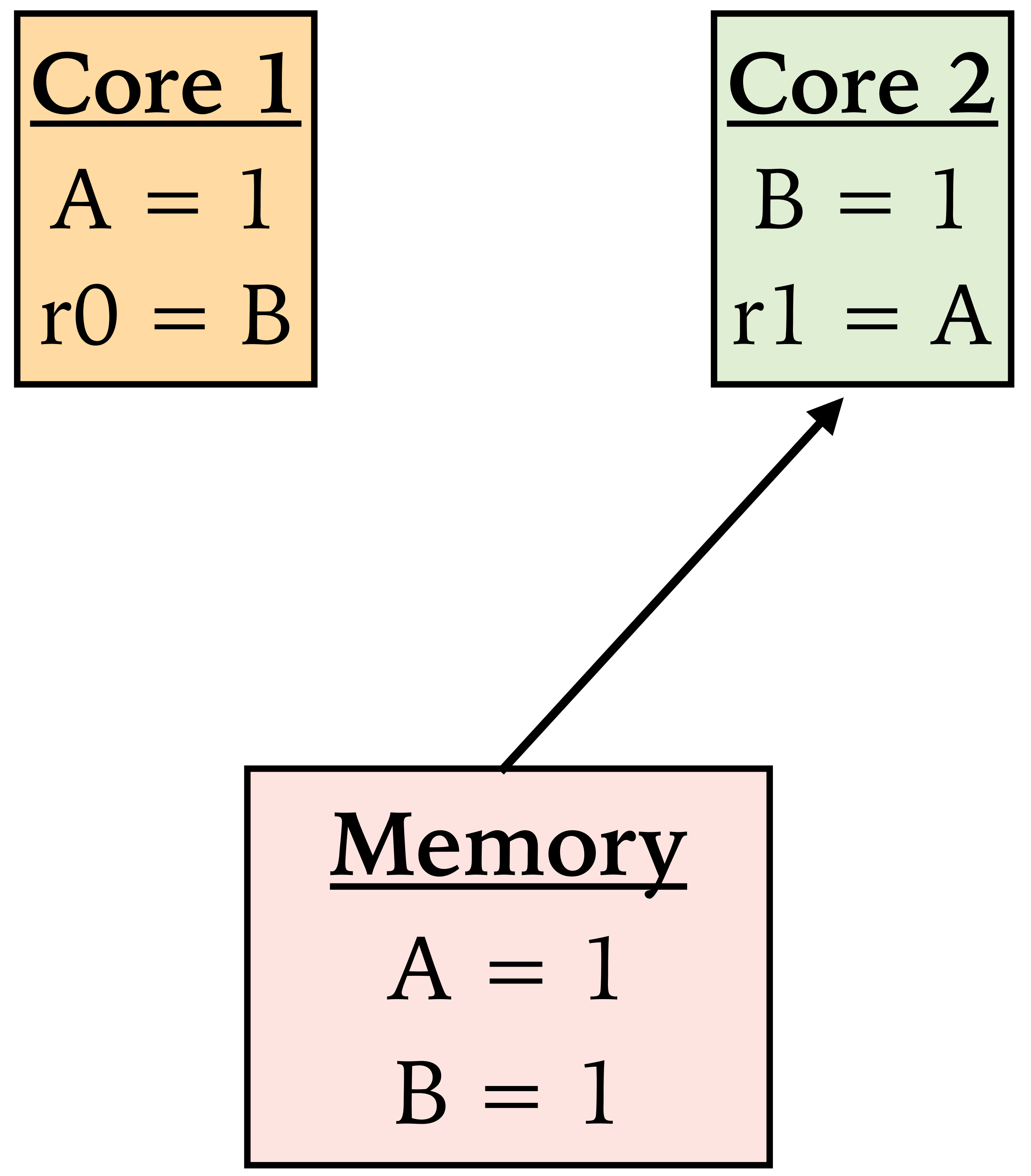


Executed

A = 1

Sequential Consistency

- Can be seen as a “switch” running one instruction at a time from different threads.



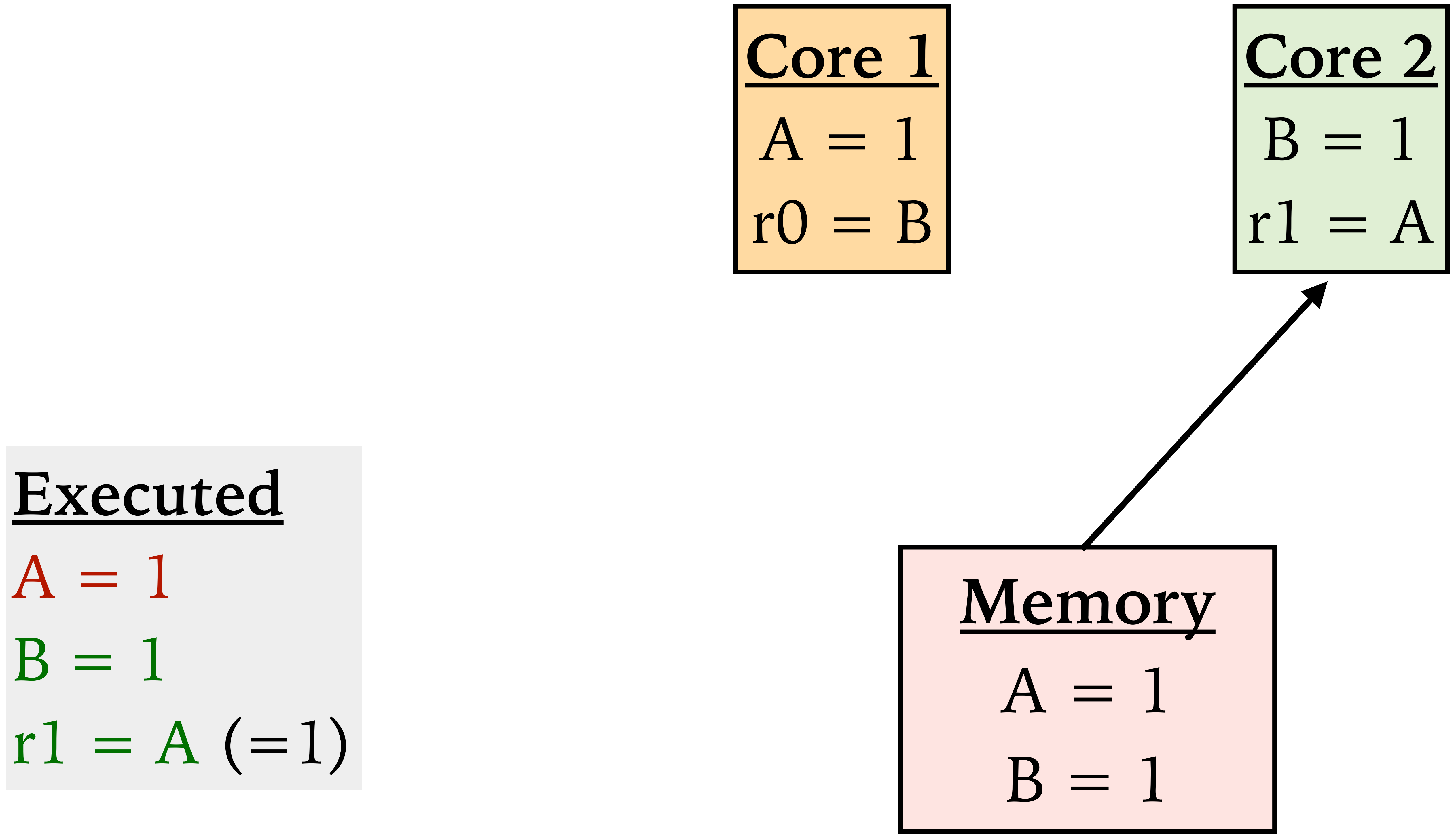
Executed

A = 1

B = 1

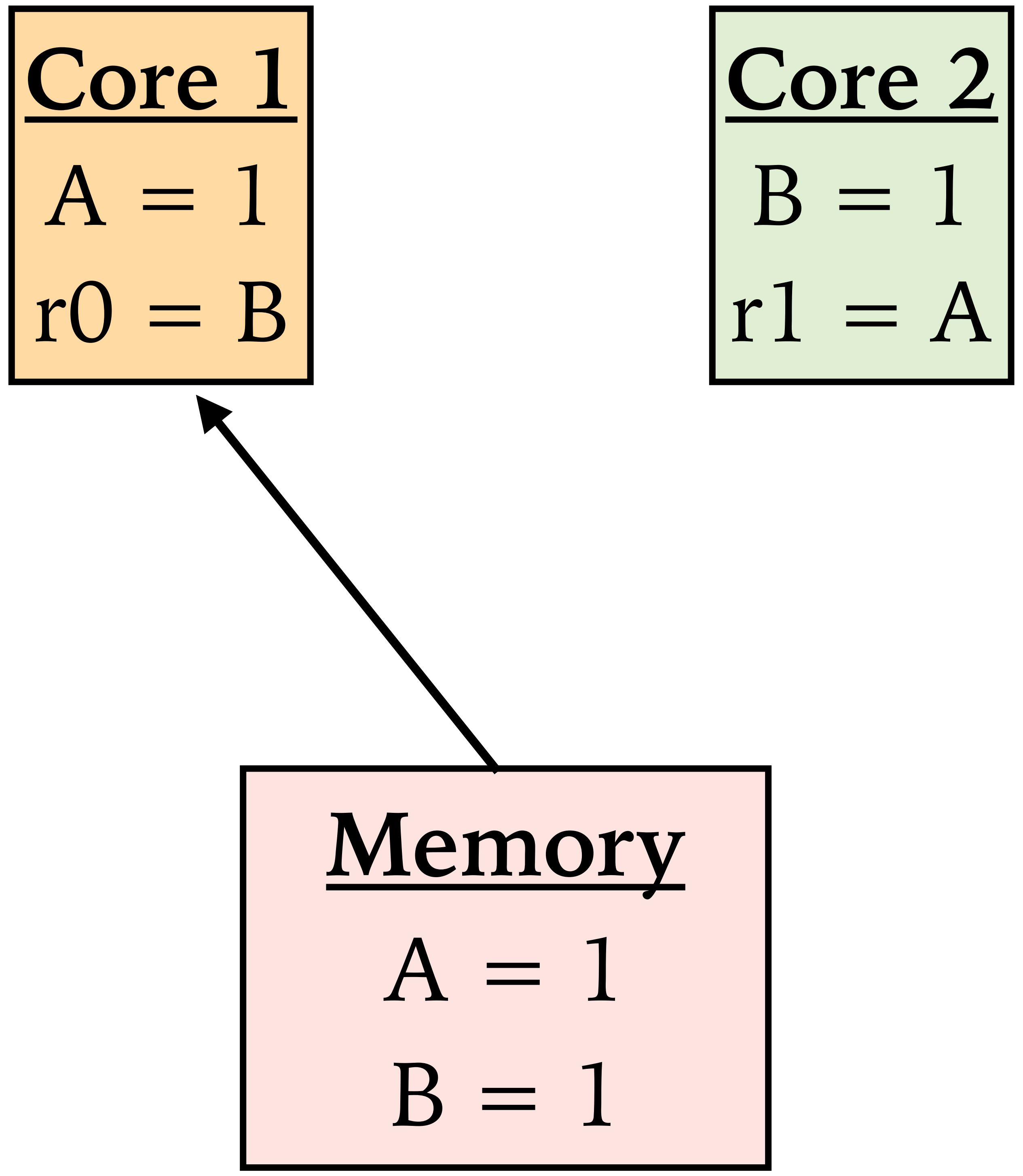
Sequential Consistency

- Can be seen as a "switch" running one instruction at a time from different threads.



Sequential Consistency

- Can be seen as a “switch” running one instruction at a time from different threads.



Executed

A = 1

B = 1

r1 = A (=1)

r0 = B (=1)

Sequential Consistency

- Two invariants:
  - All operations executed in some sequential order.
  - Each thread's operations happen in program order.

Order across threads

Order within a thread

- Says nothing about **which** order all operations happen in
  - Any interleaving of threads is allowed

- Developed by **Leslie Lamport** in 1979 (Turing Award in 2013).

Memory (Consistency) Models

- A memory model defines the permitted reorderings of memory operations during execution.

- A **contract between hardware and software**: the hardware will only mess with your memory operations in these ways.

- Sequential consistency is the strongest memory (consistency) model; allows the fewest reorderings.

Practice

- Assume sequential consistency, and all variables to be initially 0.

Thread 1	**Thread 2**
(1) X = 1	(3) r0 = Y
(2) Y = 1	(4) r1 = X

- r0 = 0 and r1 = 0? (3) —> (4) —> (1) —> (2)
- r0 = 1 and r1 = 1? (1) —> (2) —> (3) —> (4)
- r0 = 0 and r1 = 1? (1) —> (3) —> (4) —> (2)
- r0 = 1 and r1 = 0? Not possible.

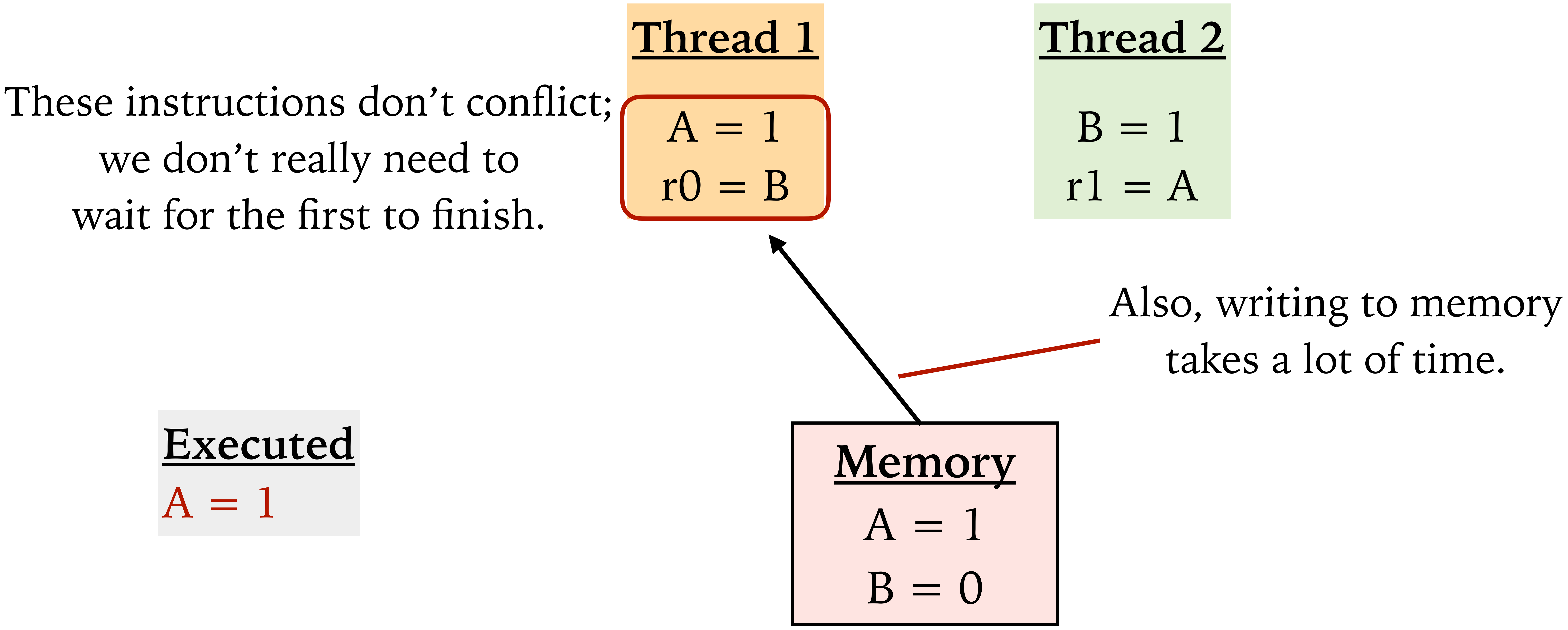
Why SC?

- Agrees with programmer intuition!

Why not SC?

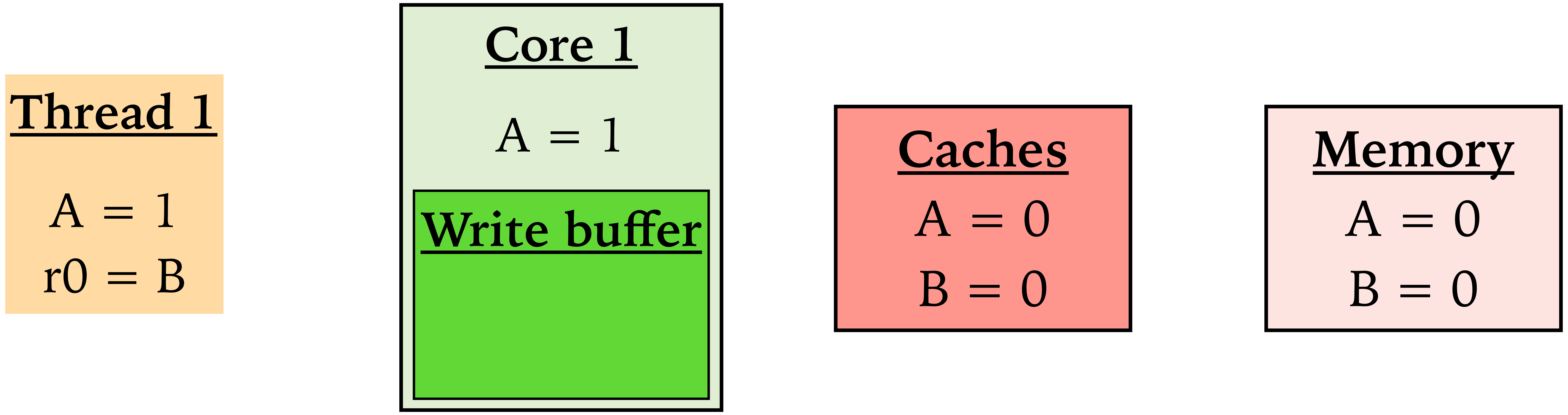
- Extremely slow to guarantee in hardware!
  - The “switch” model is very conservative and does not allow several valid reorderings.

The Problem with SC


Thread 1
A = 1
r0 = B
Thread 2
B = 1
r1 = A
These instructions don't conflict; we don't really need to wait for the first to finish.
Also, writing to memory takes a lot of time.
Executed
A = 1
Memory
A = 1
B = 0

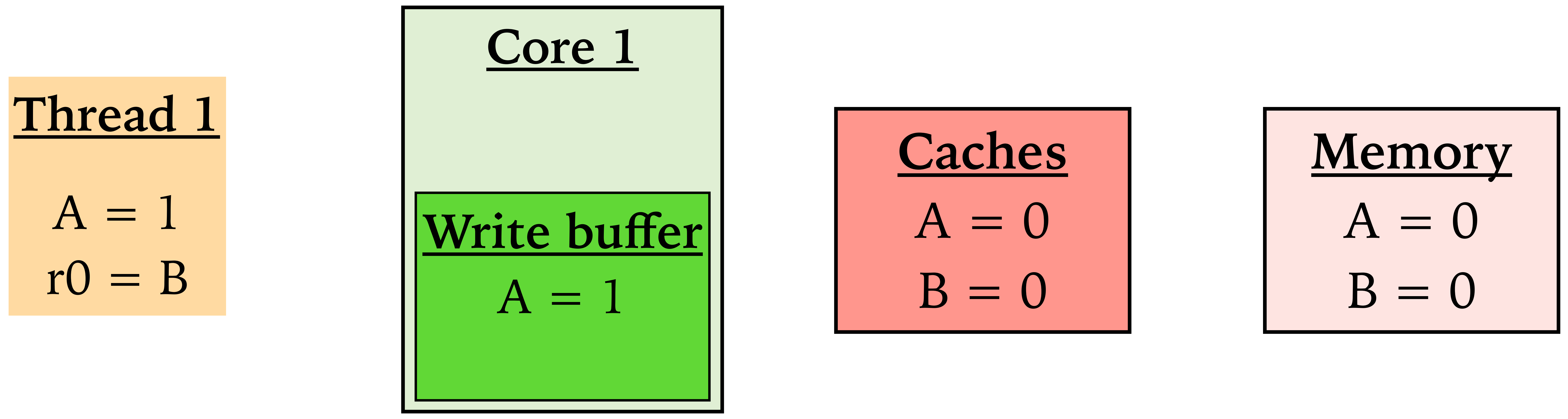
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it's ready.



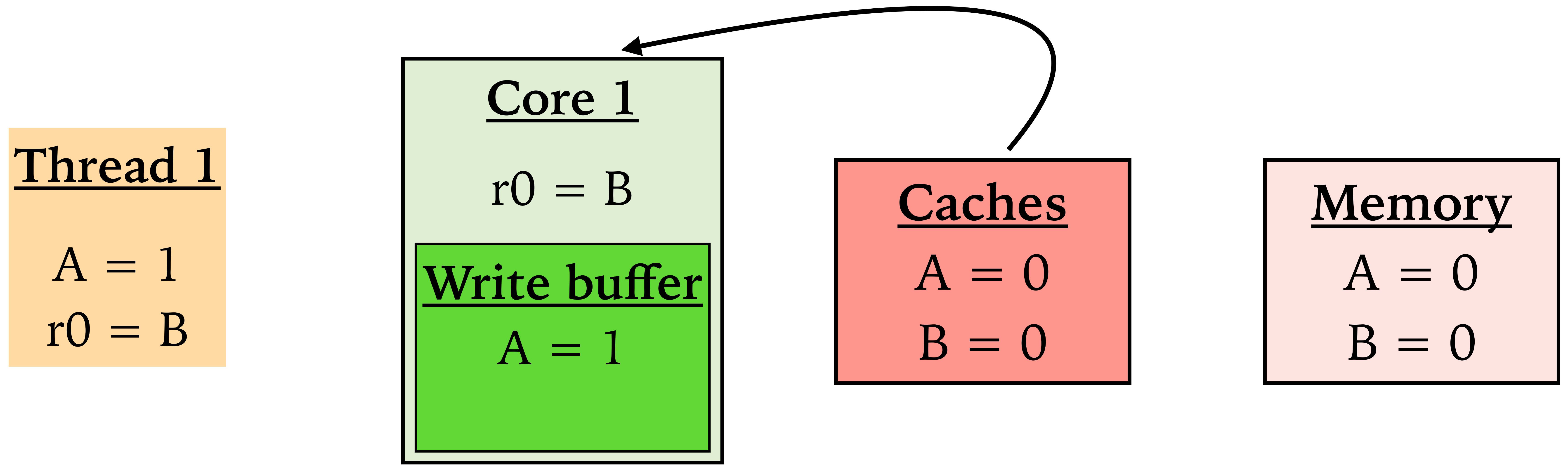
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it’s ready.



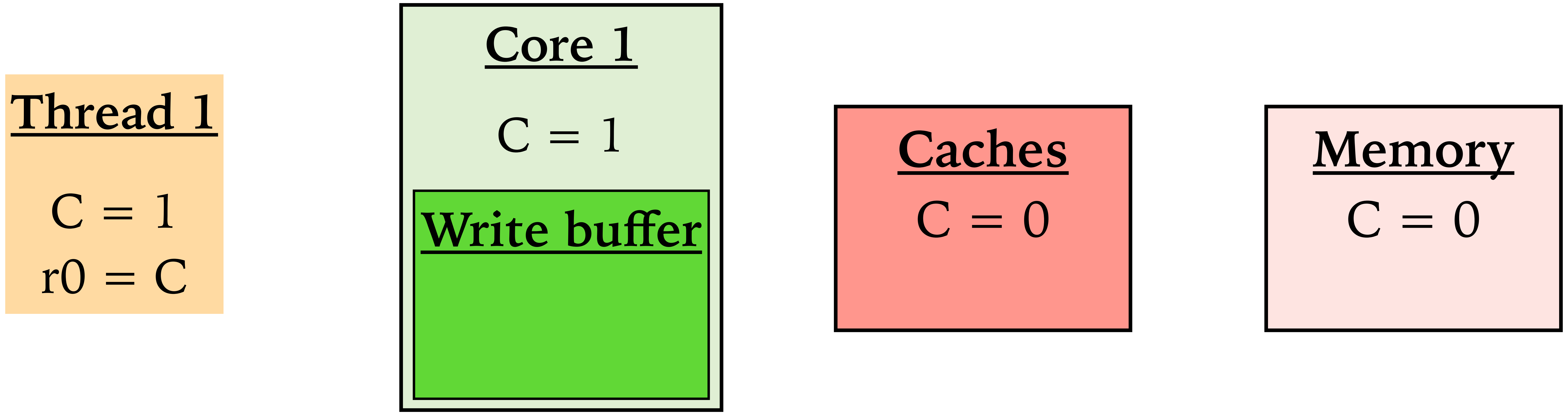
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it's ready.



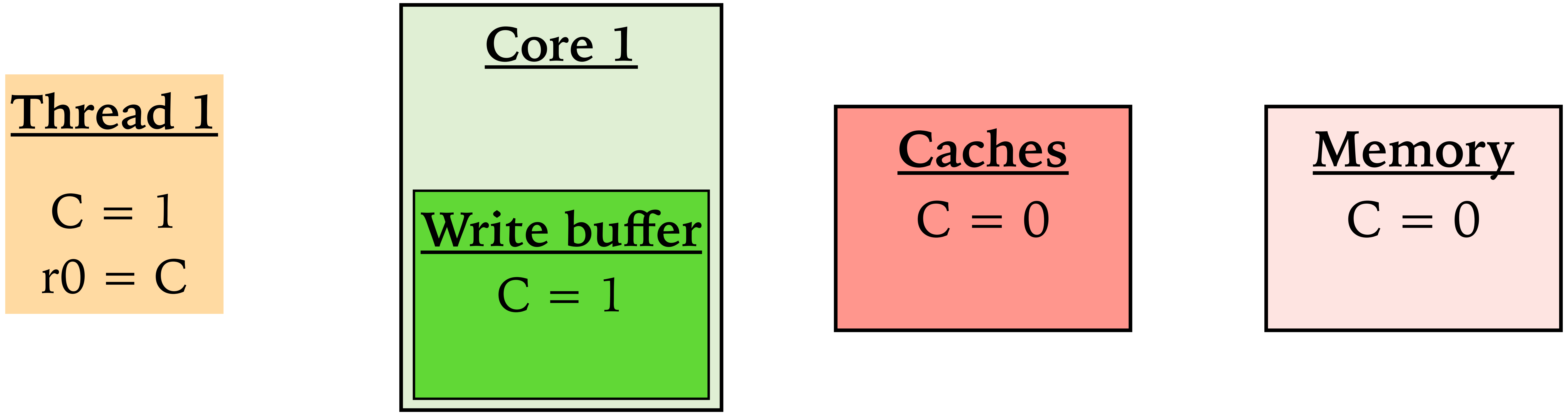
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it's ready.



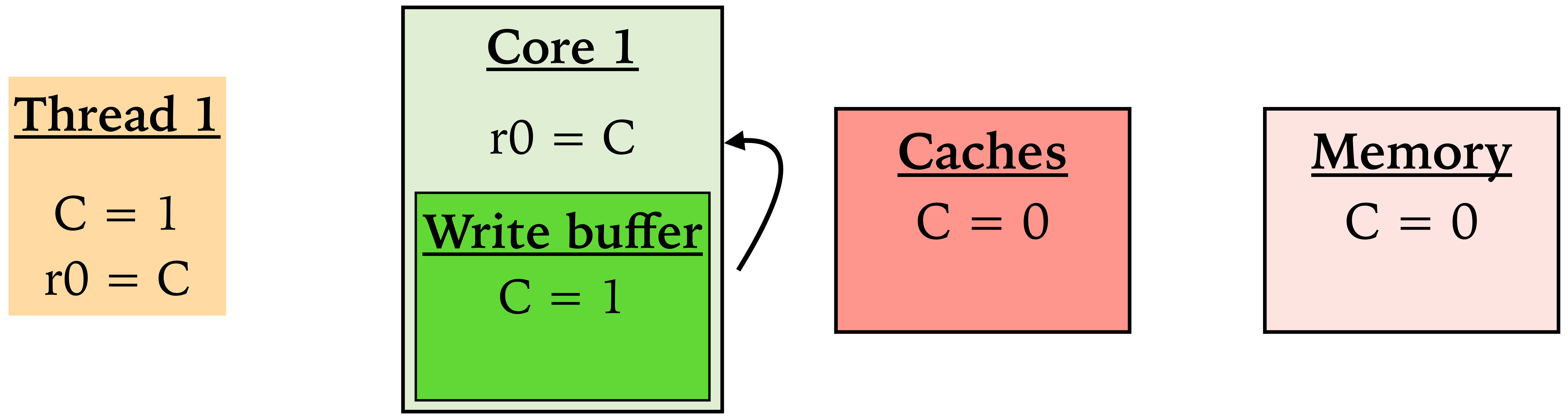
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it's ready.



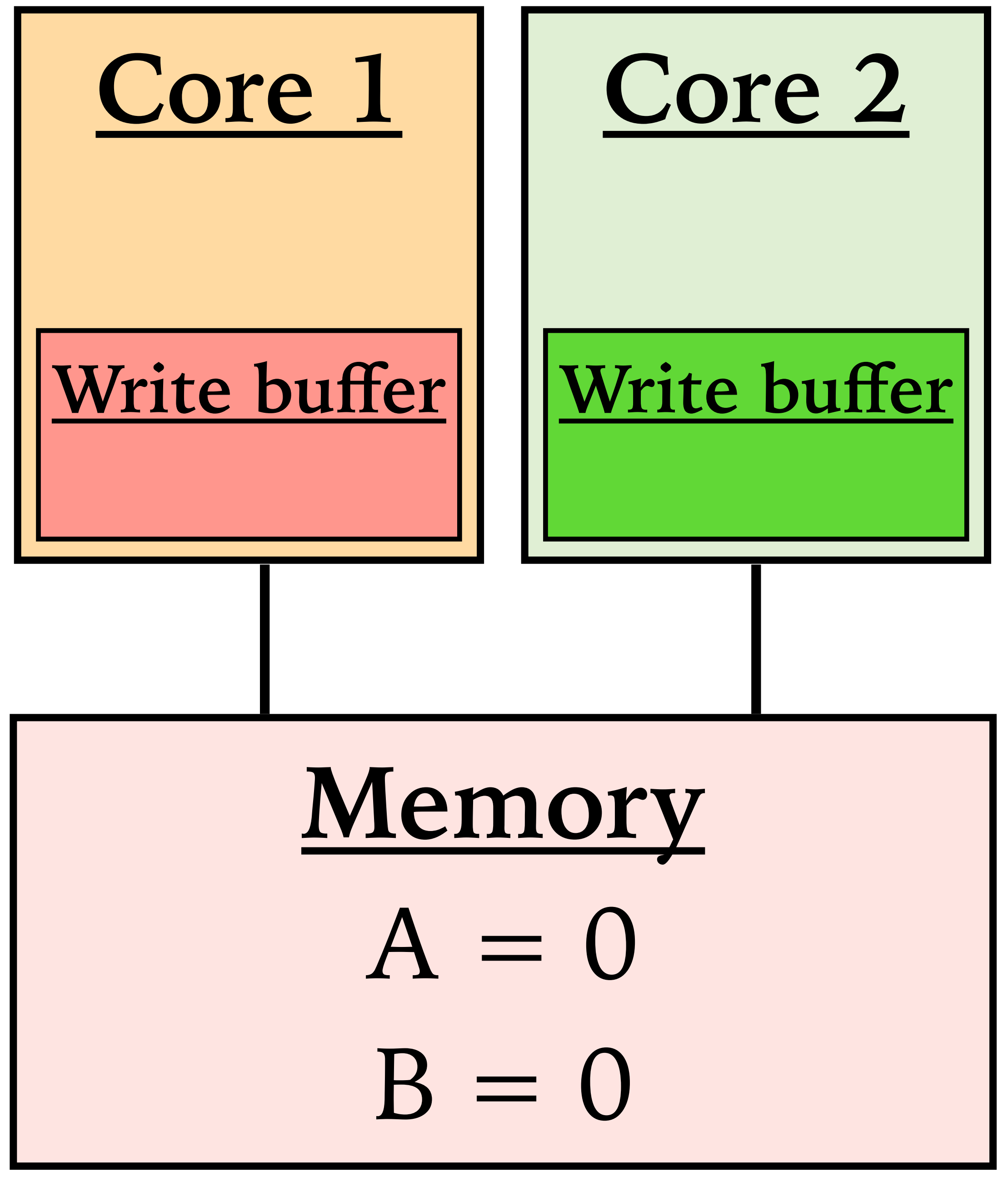
Hardware-Level Optimization: Write Buffers

- Store writes in a processor-local buffer and proceed to next instruction immediately.
- The cache will pull out writes out of the write buffer when it's ready.



- Write buffers define a *new memory model*!

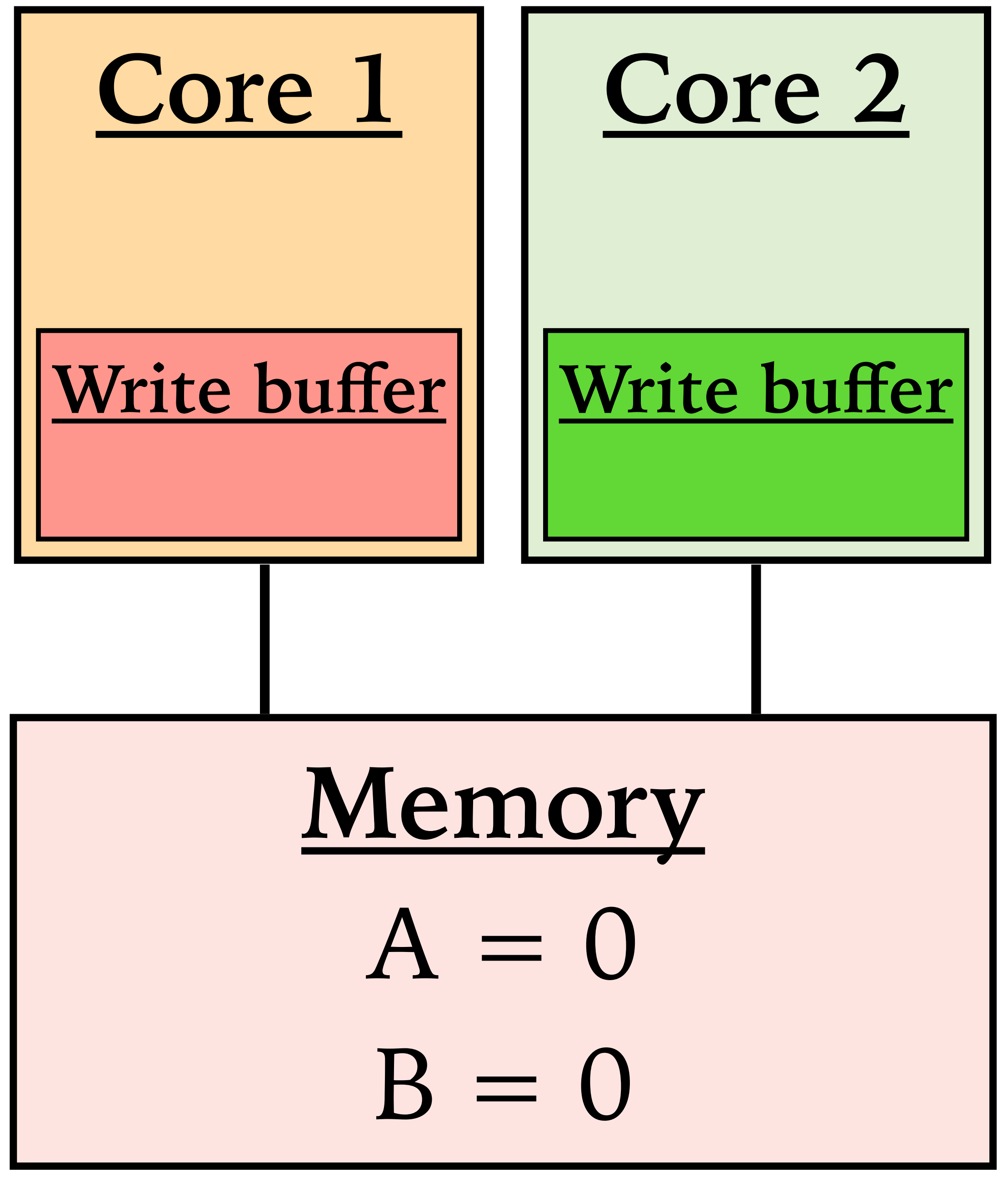
Write buffers change memory behaviour



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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour

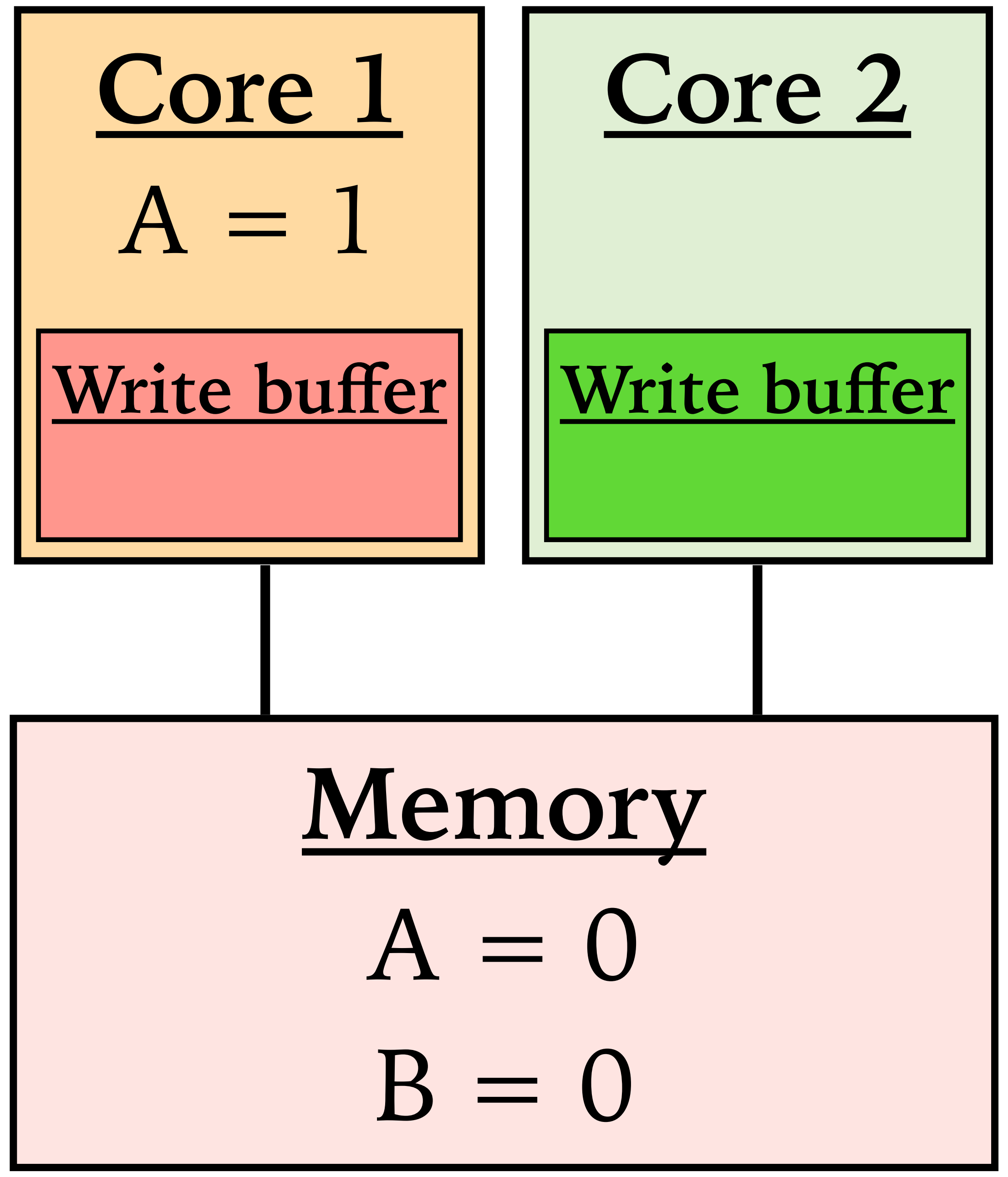


Executed

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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour

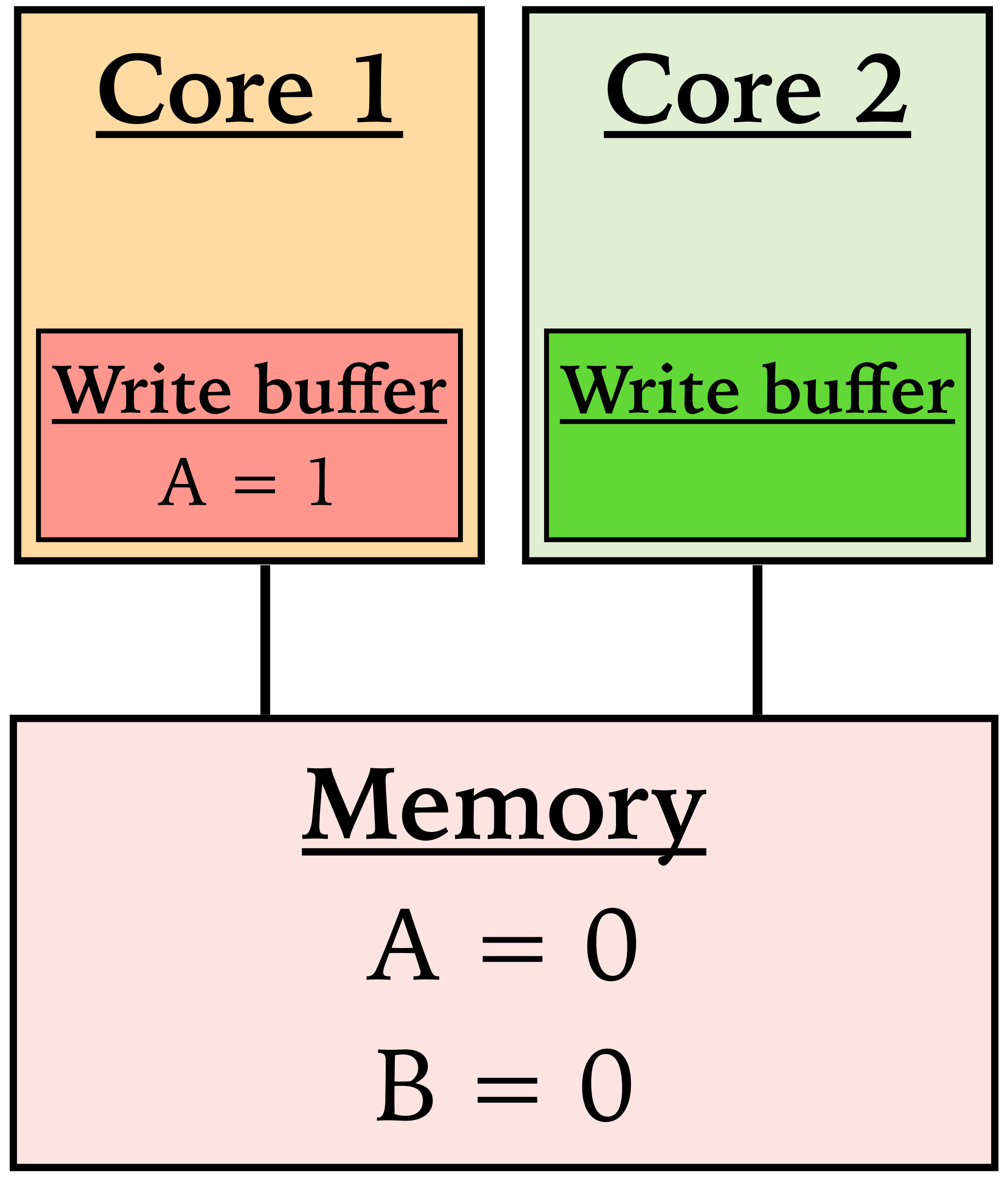


Executed

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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour



Thread 1

(1) A = 1
(2) r0 = B

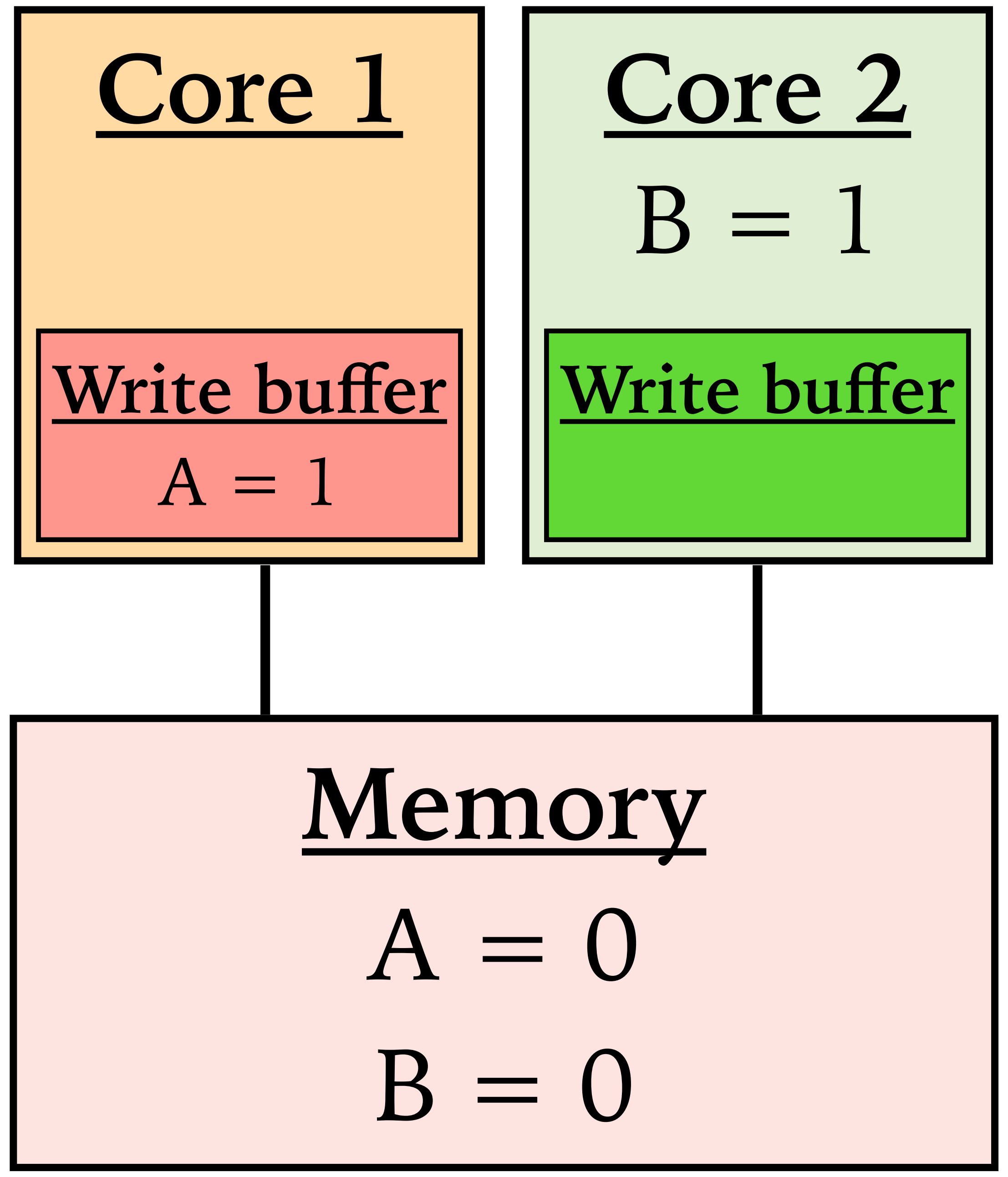
Thread 2

(3) B = 1
(4) r1 = A

Executed

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour

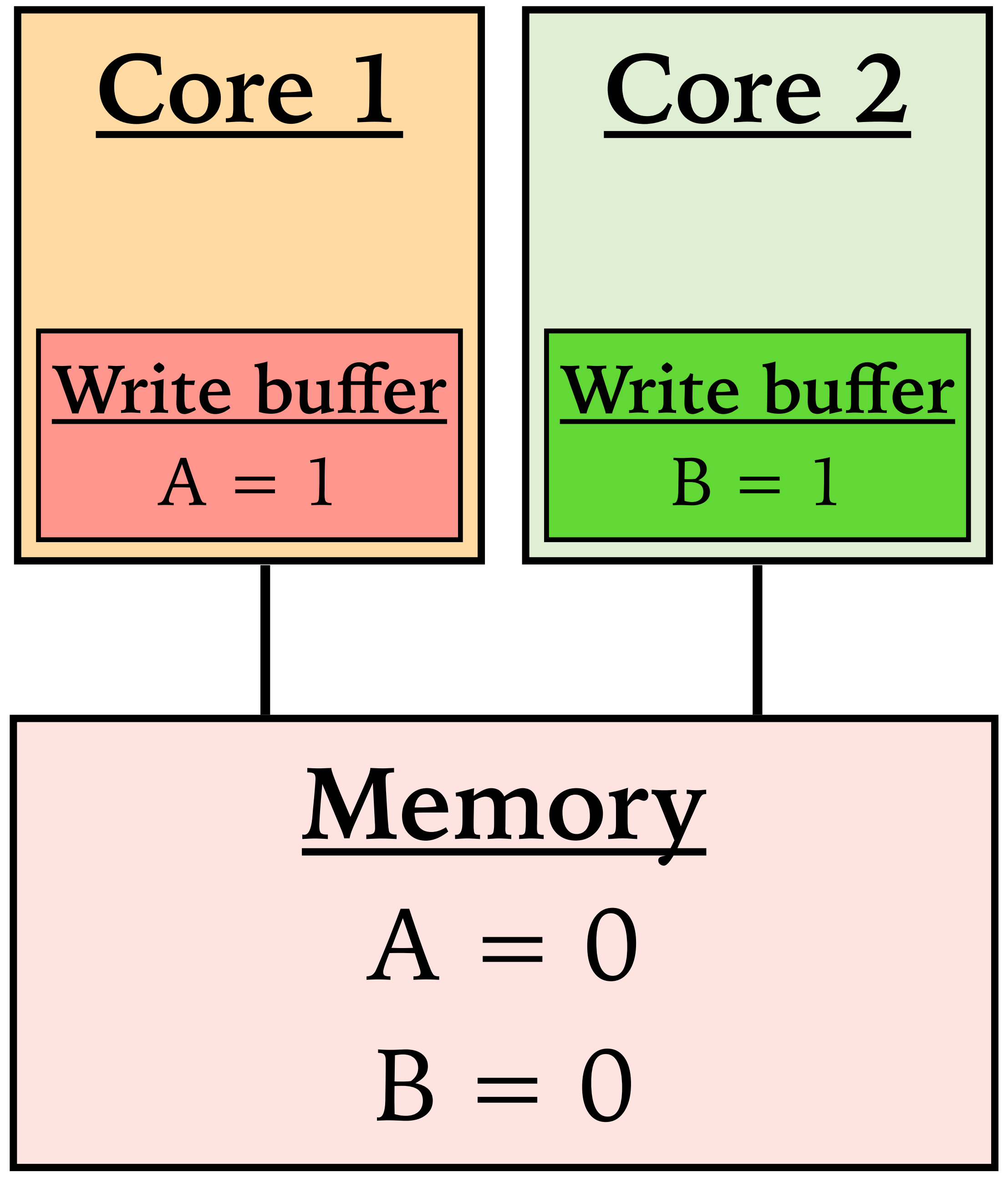


Executed

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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour



Thread 1

(1) A = 1
(2) r0 = B

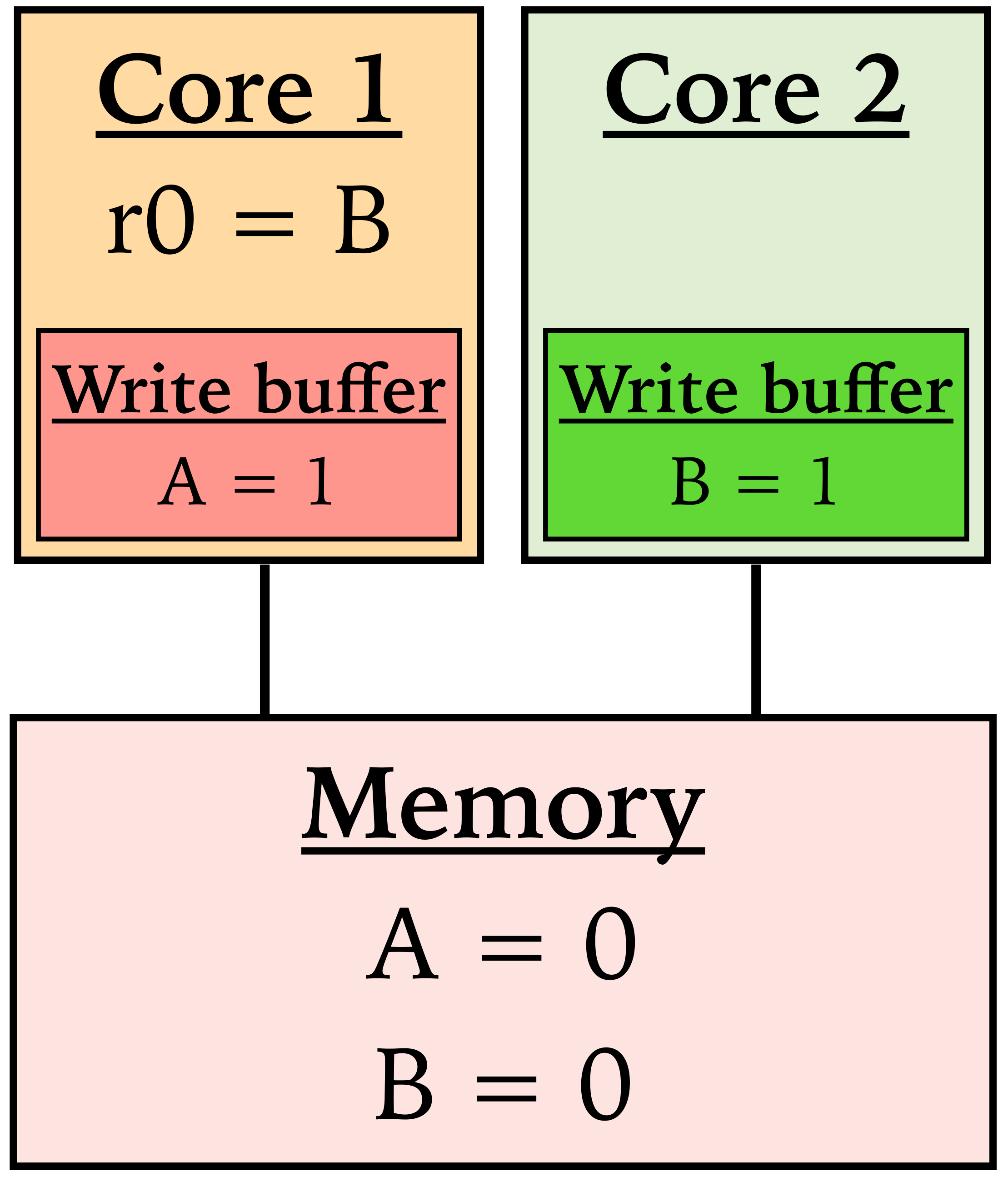
Thread 2

(3) B = 1
(4) r1 = A

Executed

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour



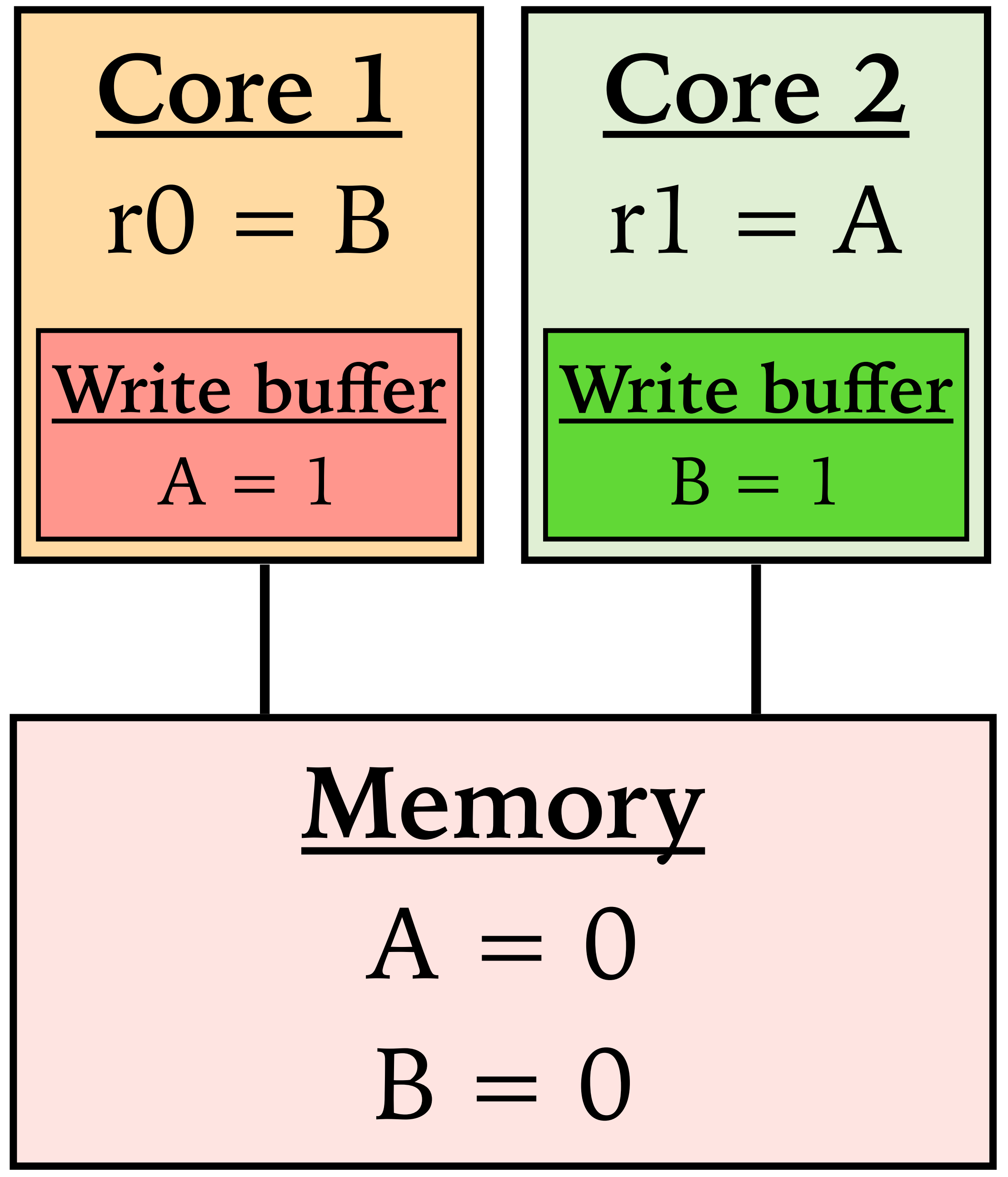
Thread 1	**Thread 2**
(1) A = 1	(3) B = 1
(2) r0 = B	(4) r1 = A

- r0 = 0? and r1 = 0?
  - SC? No.

Executed

r0 = B (=0)

Write buffers change memory behaviour



Thread 1

(1) A = 1
(2) r0 = B

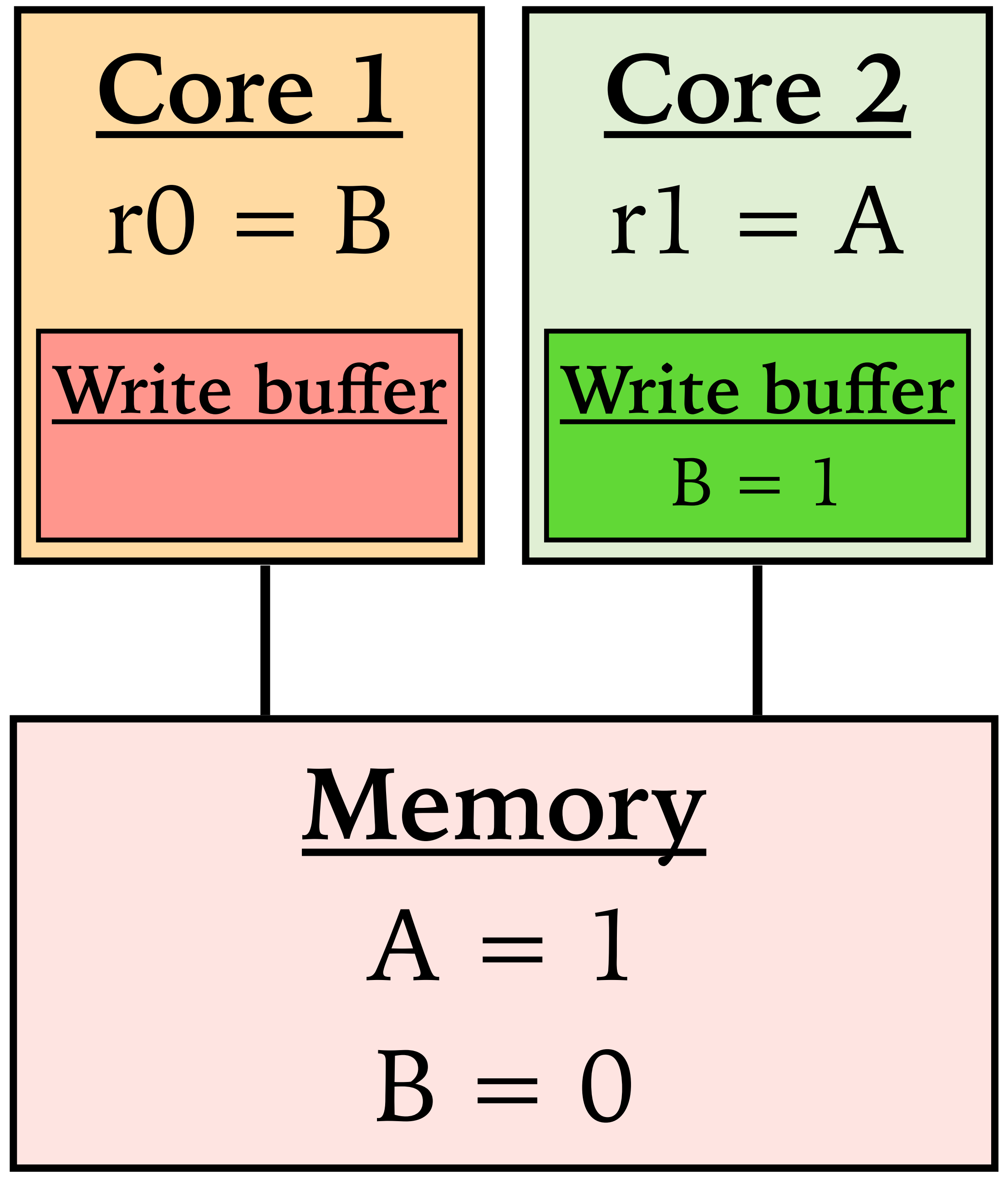
Thread 2

(3) B = 1
(4) r1 = A

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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour

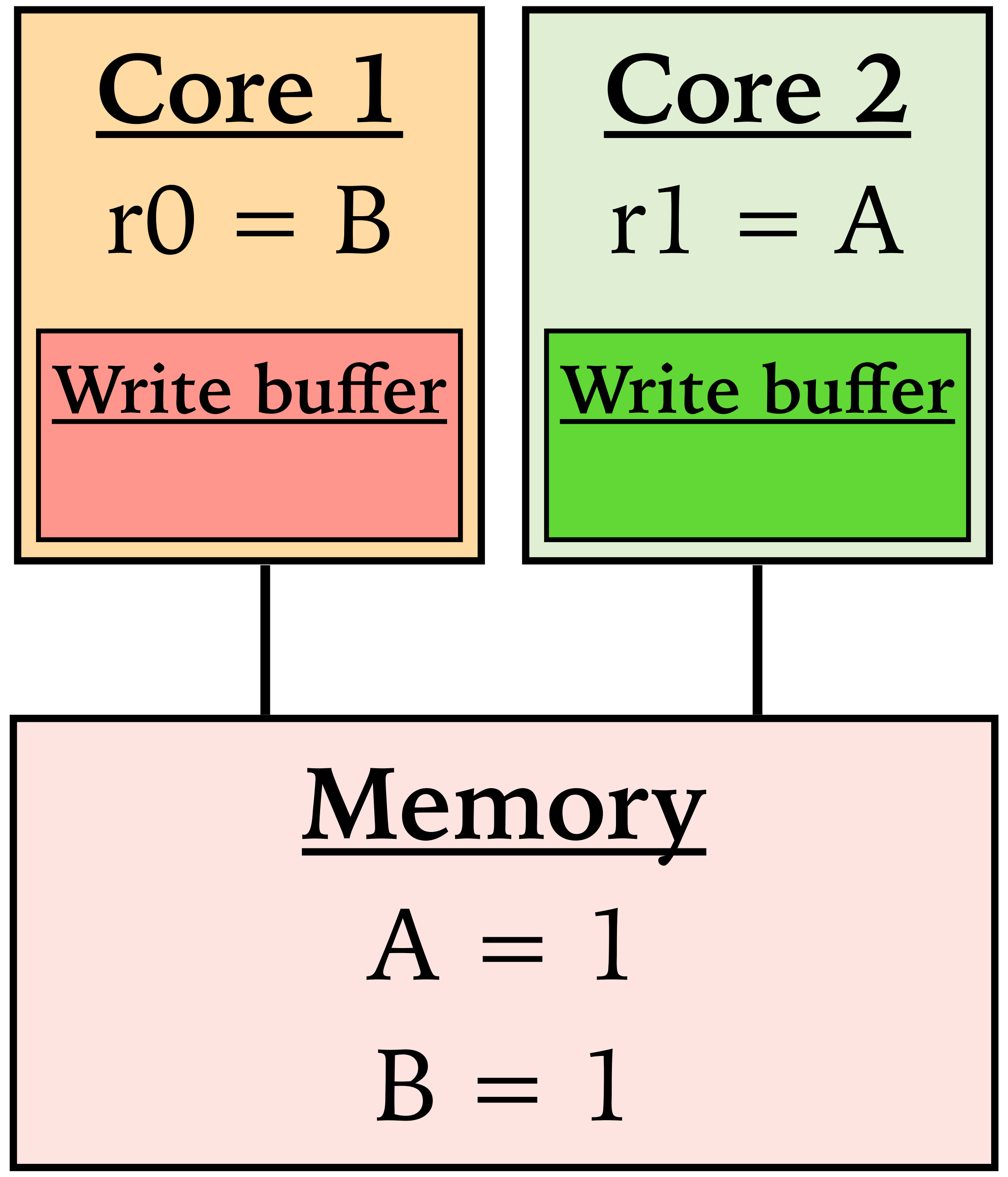


Thread 1	**Thread 2**
(1) A = 1	(3) B = 1
(2) r0 = B	(4) r1 = A

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

- r0 = 0? and r1 = 0?
  - SC? No.

Write buffers change memory behaviour



Thread 1

(1) A = 1
(2) r0 = B

Thread 2

(3) B = 1
(4) r1 = A

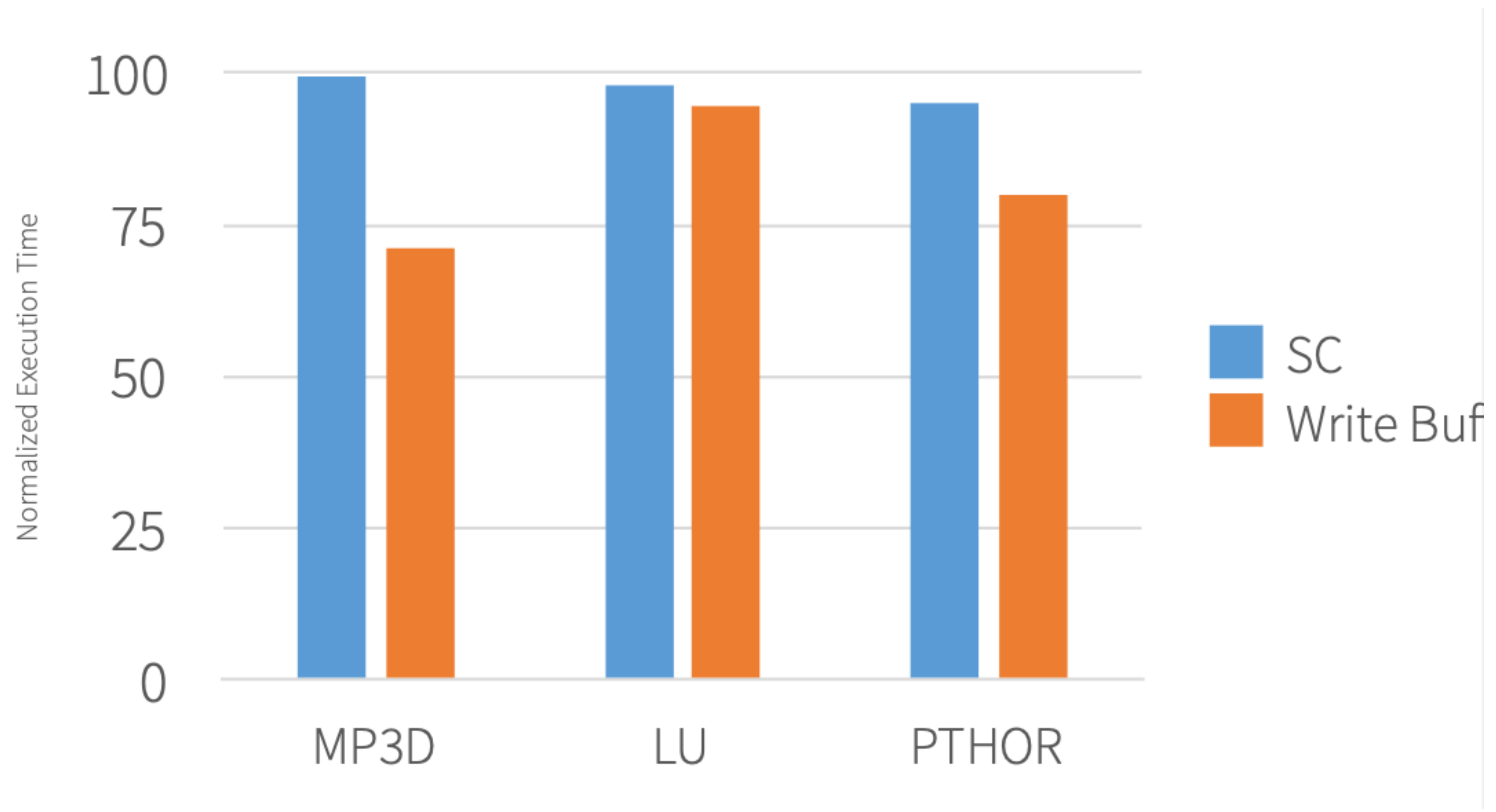
Executed

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

- r0 = 0? and r1 = 0?
  - SC? No.
  - **Write Buffers? Yes!**

But does anyone use write buffers?

- Every modern CPU!
  - x86
  - ARM
  - PowerPC
  - …



- Write buffers define a *new memory model*: **Total Store Ordering.**

Performance evaluation of memory consistency models for shared-memory multiprocessors. Gharachorloo, Gupta, Hennessy. ASPLOS 1991.

Total Store Ordering (TSO)

- TSO allows optimizations that can make the underlying program significantly faster.
- Essentially, SC + Write Buffers.
- But..
  - TSO allows more behaviours than SC.
  - Which means, programs are harder to reason about.
- Yet..
  - x86 specifies TSO as its memory model.

More "relaxed" memory models

- Partial Store Ordering (used by SPARC)
  - Supports ***write coalescing***: merge writes to the same cache line inside the write buffer to save memory bandwidth.
  - Allows writes to be reordered with other writes.
- Weak Ordering (ARM, PowerPC)
  - No guarantees about operations on data!
  - (Almost) **Everything** can be reordered.
- My head is spinning now!
  - **How do we manage predictable outputs?**
  - **Most hardware provide fences, barriers, atomic instructions, and so on.**

So compilers can use fences while generating code

- But compilers too reorder instructions, heavily!!
- SC is too restrictive even for compiler optimizations.

1. Loop-Invariant Code Motion

Thread 1
```
X = 0
for i = 0 to 100:
   X = 1
   print X
```

Thread 2
```
X = 0
```

The set of valid program outputs has changed!

Valid outputs:
```
1111111111…
1111101111…
```

Thread 1
```
X = 1
for i = 0 to 100:
   print X
```

Thread 2
```
X = 0
```

So compiler can't perform LICM??

Valid outputs:
```
1111111111…
1111100000…
```

So compilers can use fences while generating code

- And we know that compilers heavily reorder instructions!!
- SC is too restrictive even for compiler optimizations.

2. Common Subexpression Elimination

Thread 1

```
A = 6
FLAG = 1
```

Thread 2

```
C = A - 1
while (FLAG == 0) {}
B = A - 1
```

`(A - 1)` cannot be eliminated for assignment to B!

So compiler can't perform CSE??

So compilers can use fences while generating code

- And we know that compilers heavily reorder instructions!!
- SC is too restrictive even for compiler optimizations.

3. Register Allocation

FLAG cannot be allocated a register!

Thread 1

```
A = 6
FLAG = 1
```

Thread 2

```
while (FLAG == 0) {}
B = A
```

Basically, PLs need memory models of their own.

Memory load in each iteration??

Programming Language Memory Models

- Guess which was the first programming language to have a memory model?
  - Of course: Java
- And which language followed suit?
  - Of course: C++
- Java and C++ have the most extensive memory-model descriptions till date.
  - But Java has multithreading in-built, which is good.

Java Memory Model

- SC is guaranteed for programs that are **data-race free (DRF)**.
    - Observe that all confusing programs that we saw had data-races!
    - SC for DRF.
- If the program has data-races:
    - the user is supposed to know what they were doing;
    - the compiler can reorder anything that is not synchronized.
- *p.s.* C++ memory model is quite similar, except that it has undergone revisions more number of times than Java's, because:
    - multi-threading is a library feature (different stakeholders, more confusion).

JMM Guarantees

- Each action in a thread happens before every action in that thread that comes later in the program's order.
- An unlock on a monitor happens before every subsequent lock on that same monitor.
- A write to a volatile field happens before every subsequent read of that same volatile.
- A call to `start()` on a thread happens before any actions in the started thread.
- All actions in a thread happen before any other thread successfully returns from a `join()` on that thread.

TLDR: Synchronization and fork-join constructs, as well as writes to volatile fields, enforce our good-old friend: ***happens-before relationship*** :-)

Key Takeaways

- A memory model defines what valid behaviours can a programmer expect, with a multi-threaded program written in a language `L` and executed on a machine `M`.
- Simpler to understand memory models (e.g. SC) may disallow impactful reordering transformations, both at compiler and hardware levels.
- Relaxed memory models (e.g. TSO) allow performing some thread-local optimizations.
- Java assures SC on DRF programs; that is, programmer has to prevent data races using synchronization constructs.
    - Observe: SC on DRF is not the same as determinism (various thread-local reorderings and cross-thread interleavings still allowed).
- But programmers are stupid (okay, sometimes)!
- Consequence: Data-race detection is a *very* active research area.