Memory Consistency Models

CSE 451, Autumn 2015

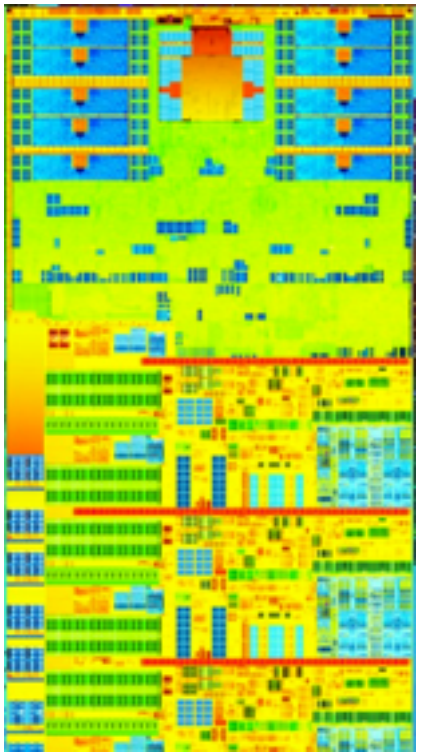
James Bornholt

With thanks to Sarita Adve, Luis Ceze, Brandon Lucia, Kayvon Fatahalian for slides

Memory consistency models

The short version:

- Multiprocessors reorder memory operations in unintuitive, scary ways
- This behavior is necessary for performance
- Application programmers rarely see this behavior
- But kernel developers see it all the time



Multithreaded programs

Initially A = B = 0

Thread 1

```
A = 1
if (B == 0)
    print “Hello”;
```

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

What can be printed?

- “Hello”?
- “World”?
- Nothing?
- “Hello World”?

Things that shouldn’t happen

This program should never print “Hello World”.

Thread 1

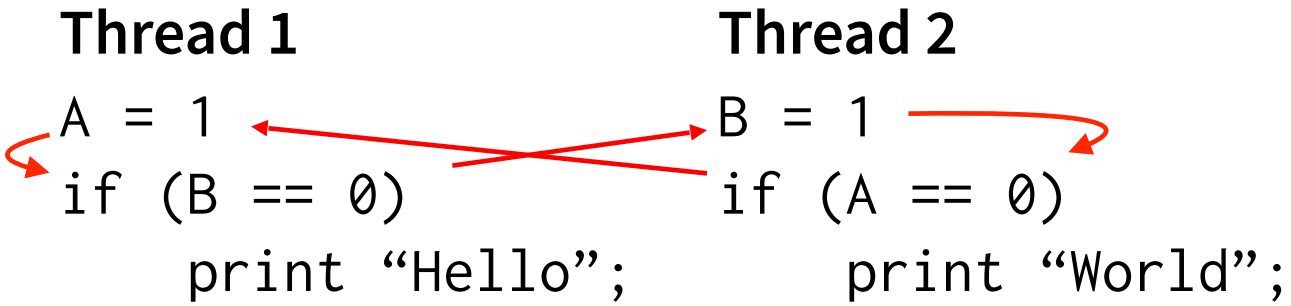
```
A = 1
if (B == 0)
    print “Hello”;
```

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

Things that shouldn't happen

This program should never print "Hello World".



A "happens-before" graph shows the order in which events must execute to get a desired outcome.

- If there's a cycle in the graph, an outcome is impossible—an event must happen before itself!

Sequential consistency

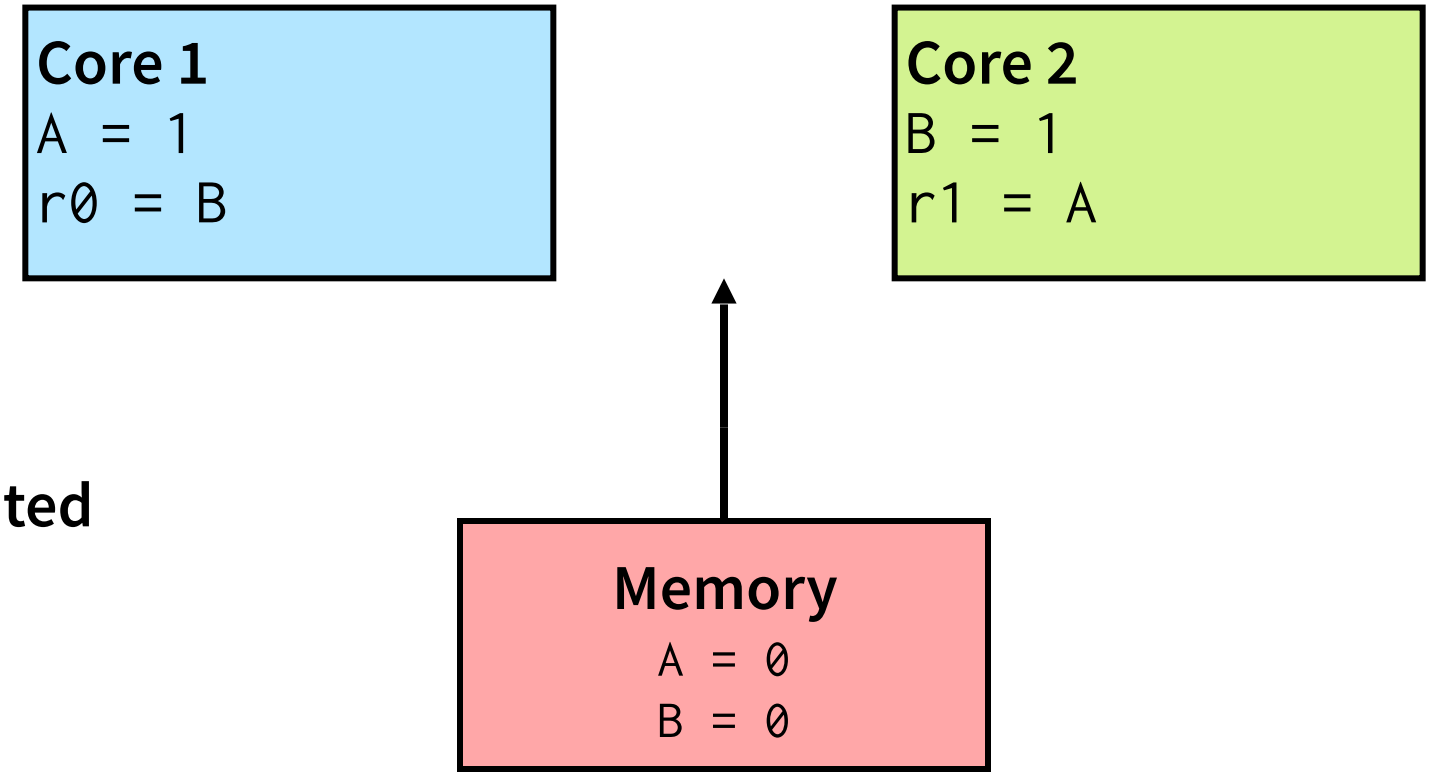
- All operations executed in some sequential order
  - As if they were manipulating a single shared memory
- Each thread's operations happen in program order

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

Not allowed: `r0 = 0` and `r1 = 0`

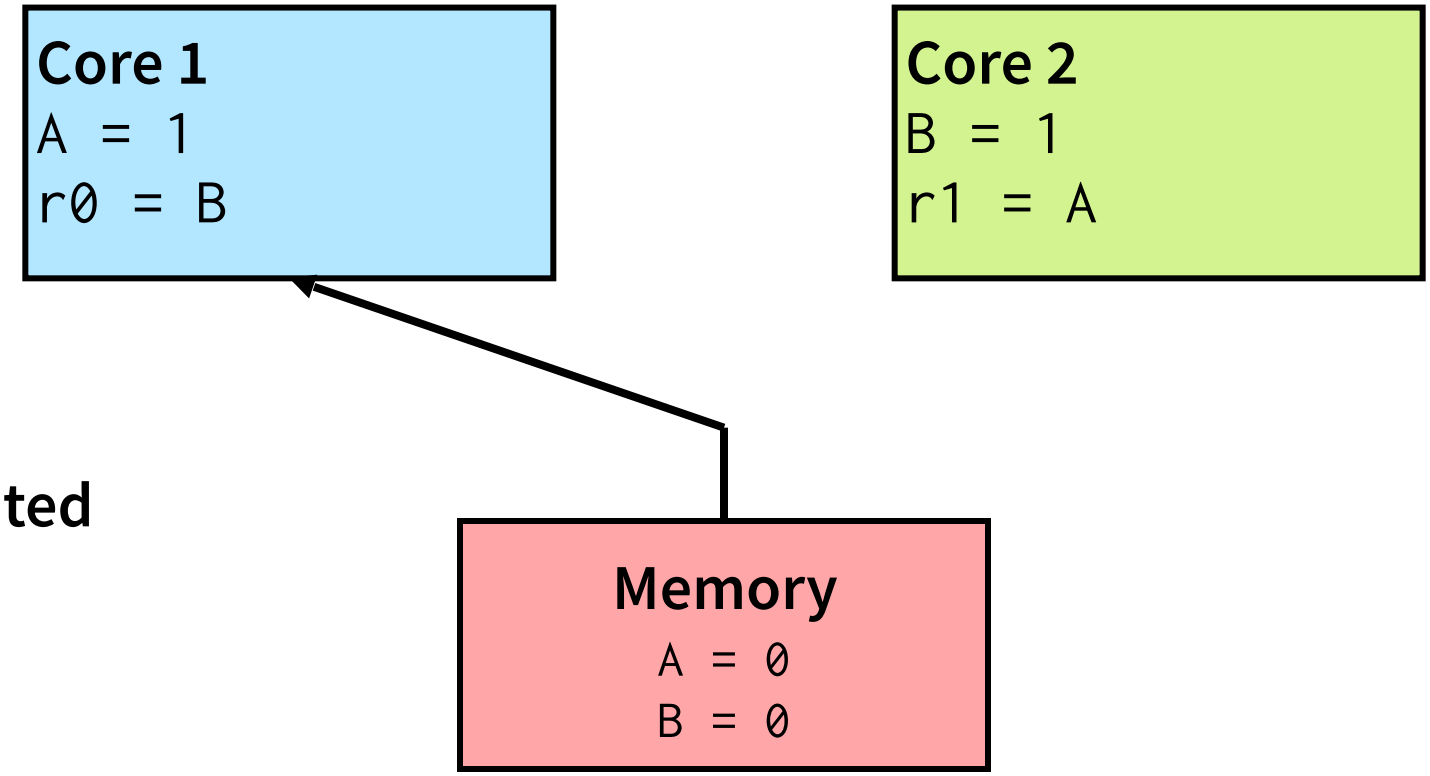
Sequential consistency

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



Sequential consistency

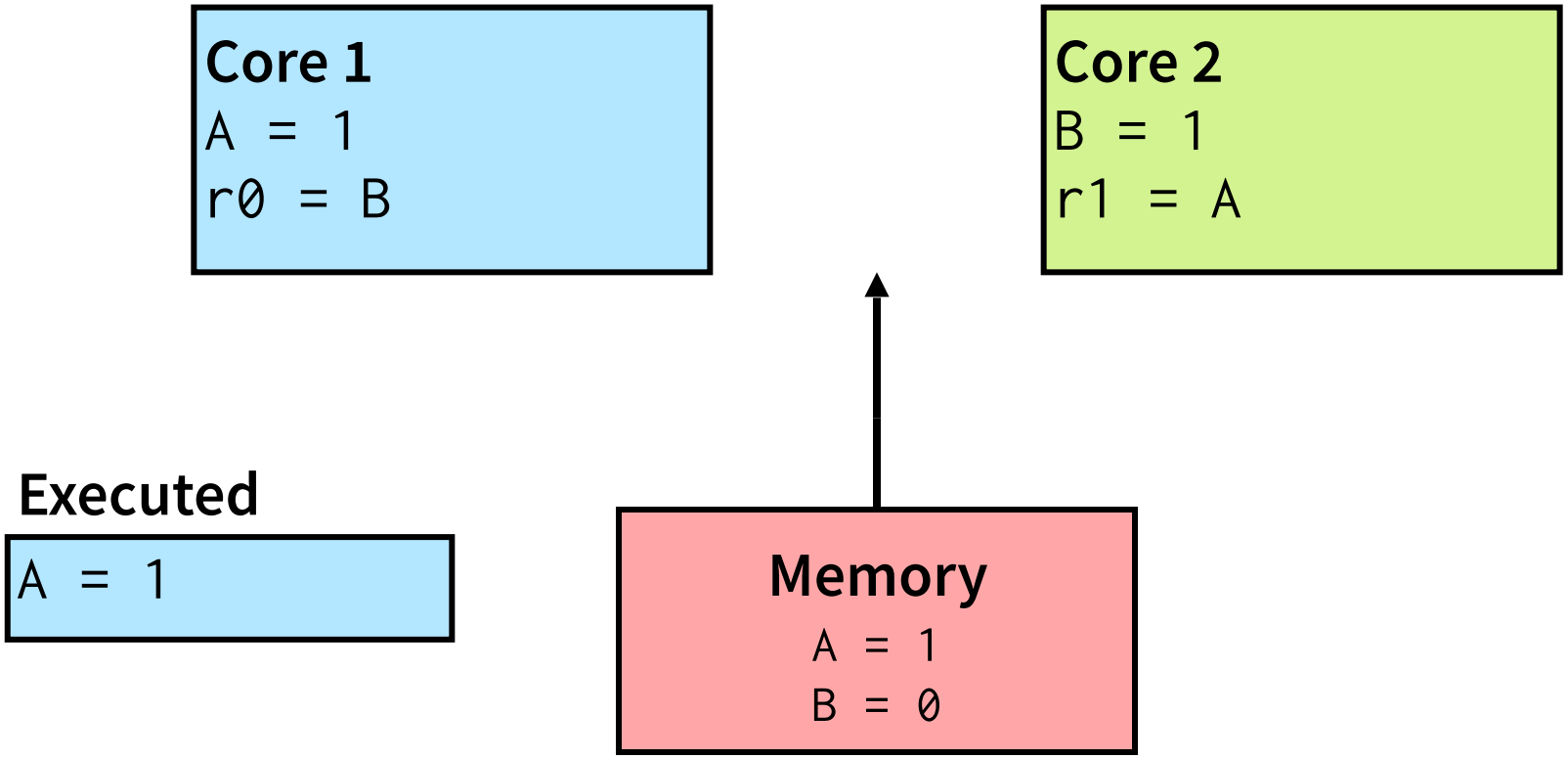
Can be seen as a “switch” running one instruction at a time



Executed

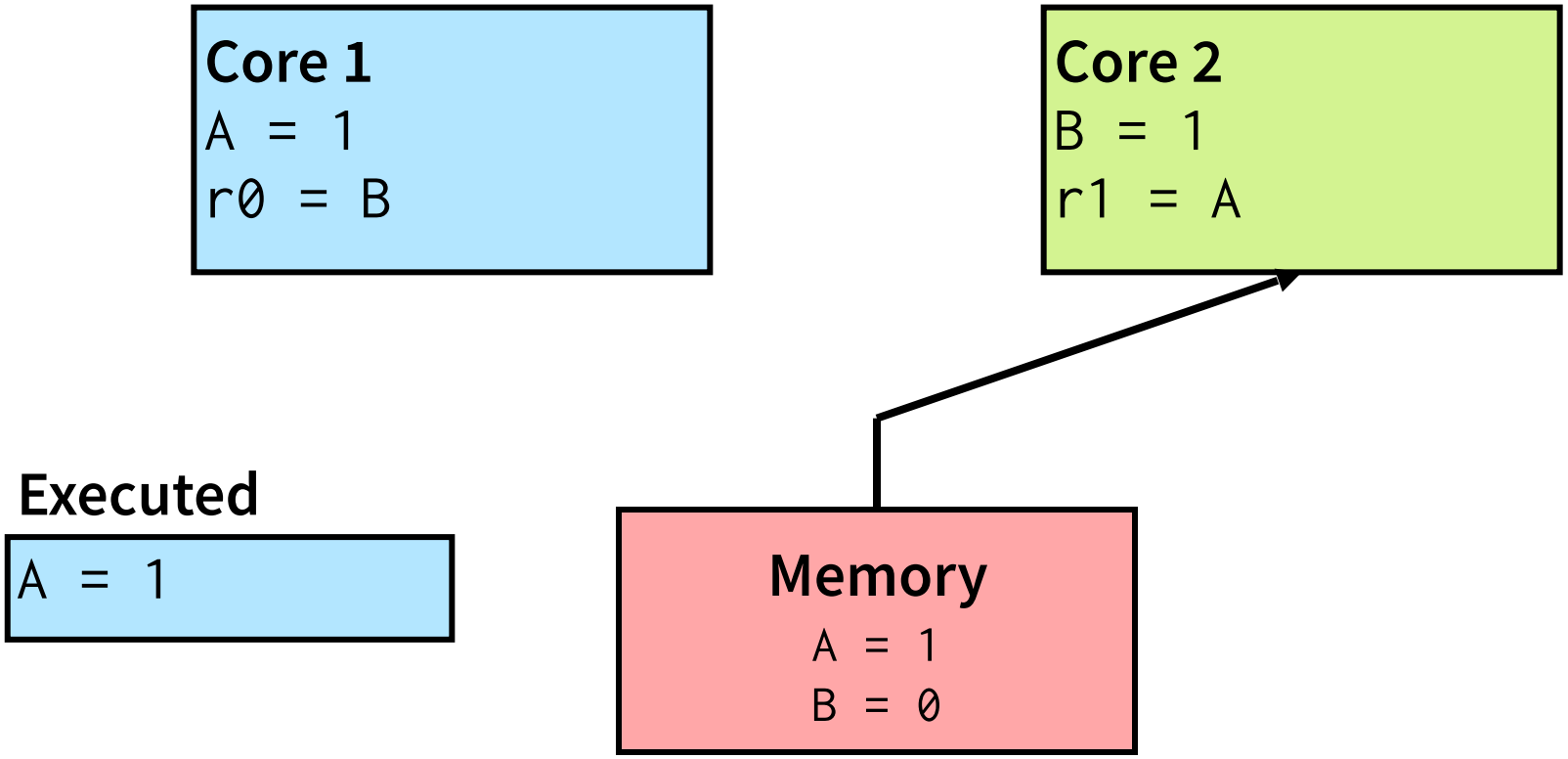
Sequential consistency

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



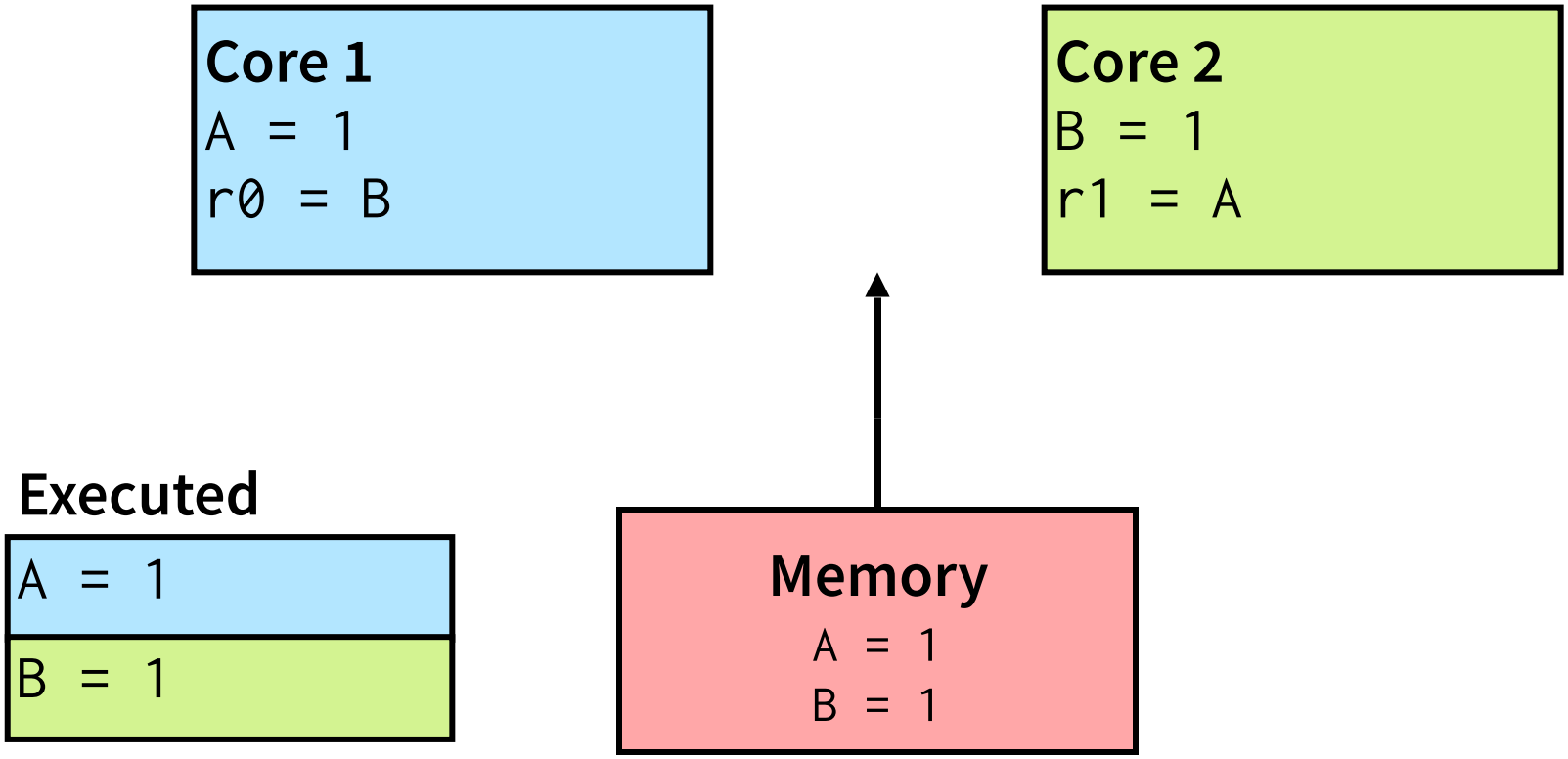
Sequential consistency

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



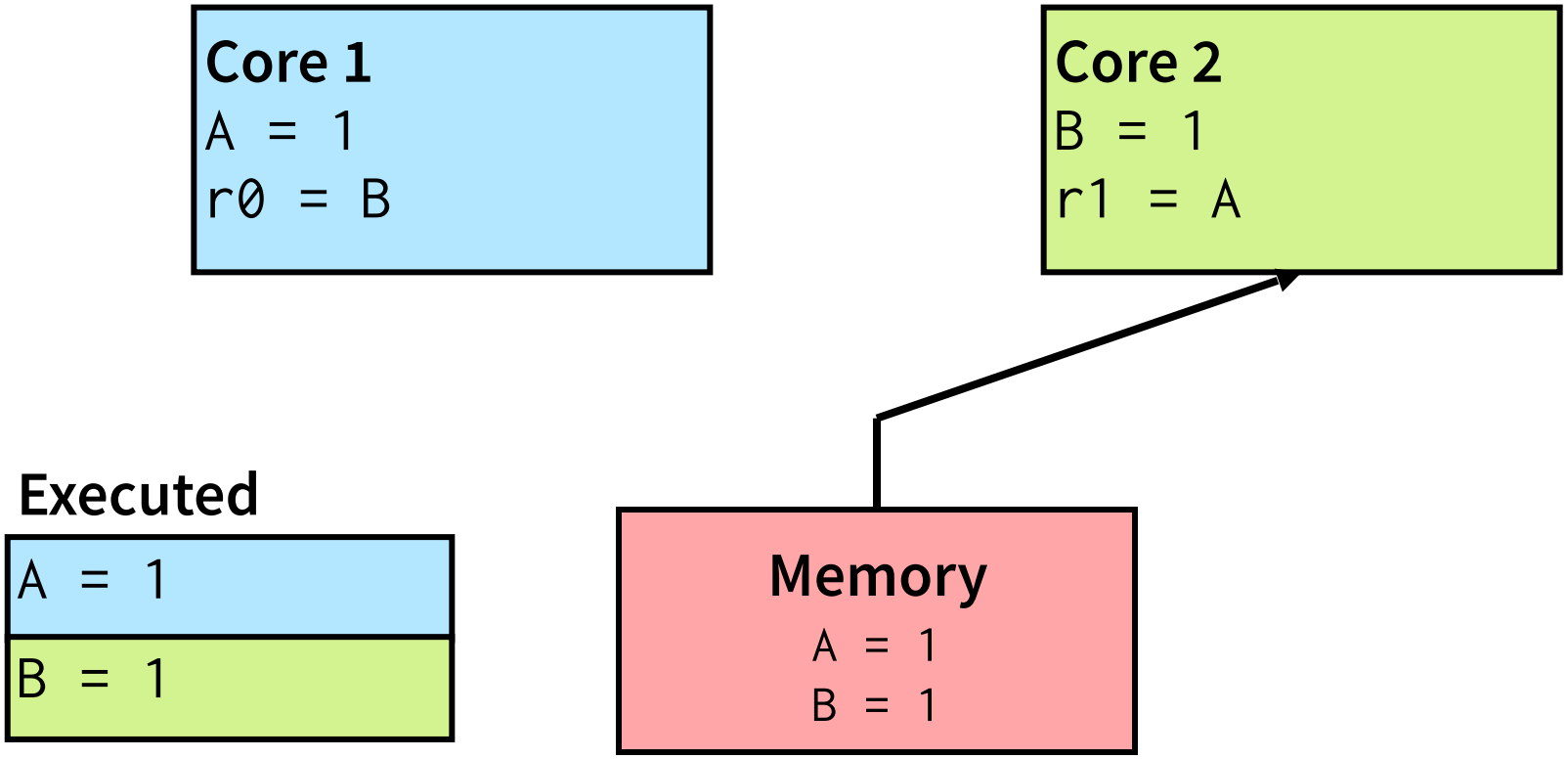
Sequential consistency

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



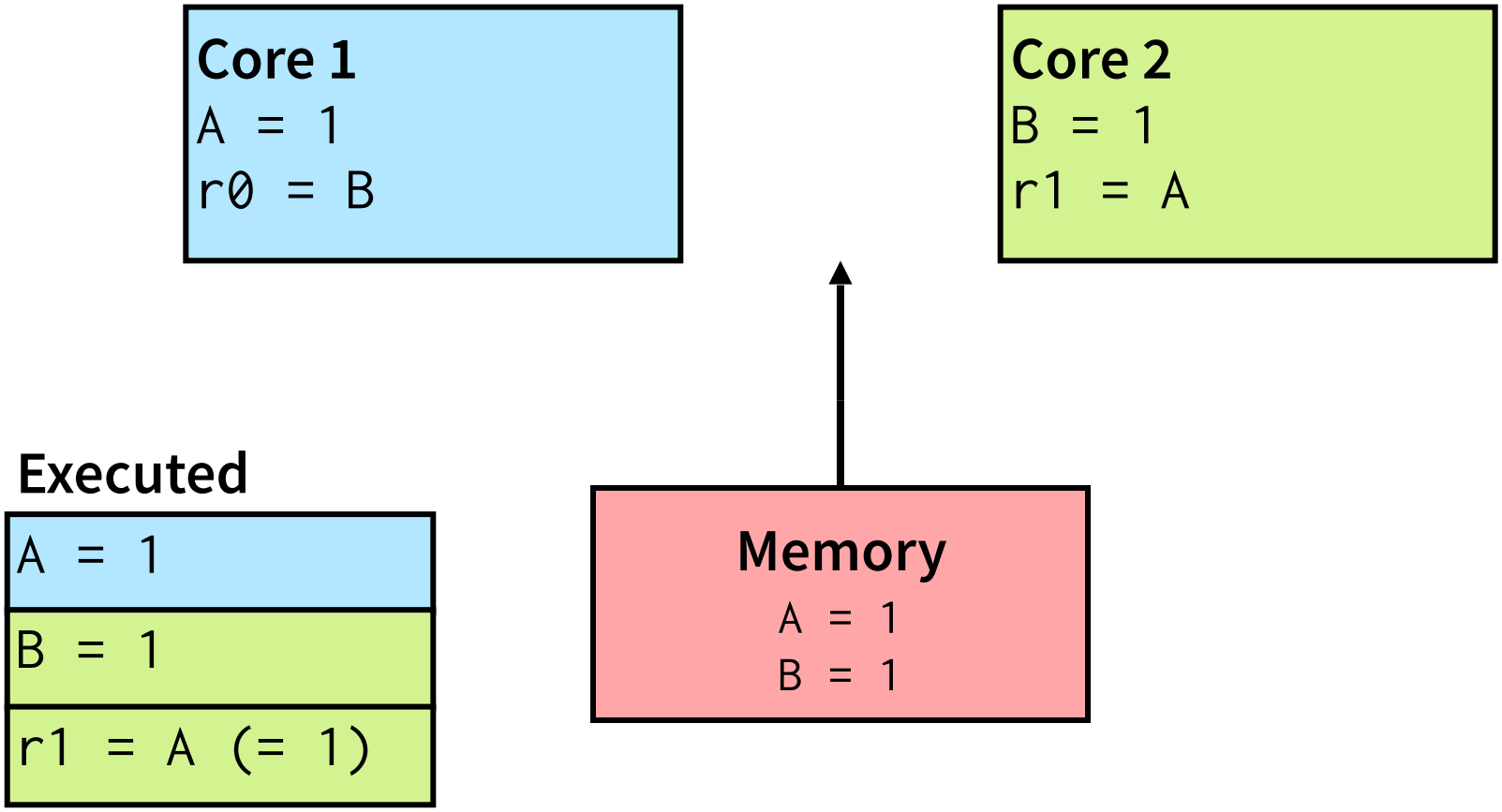
Sequential consistency

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



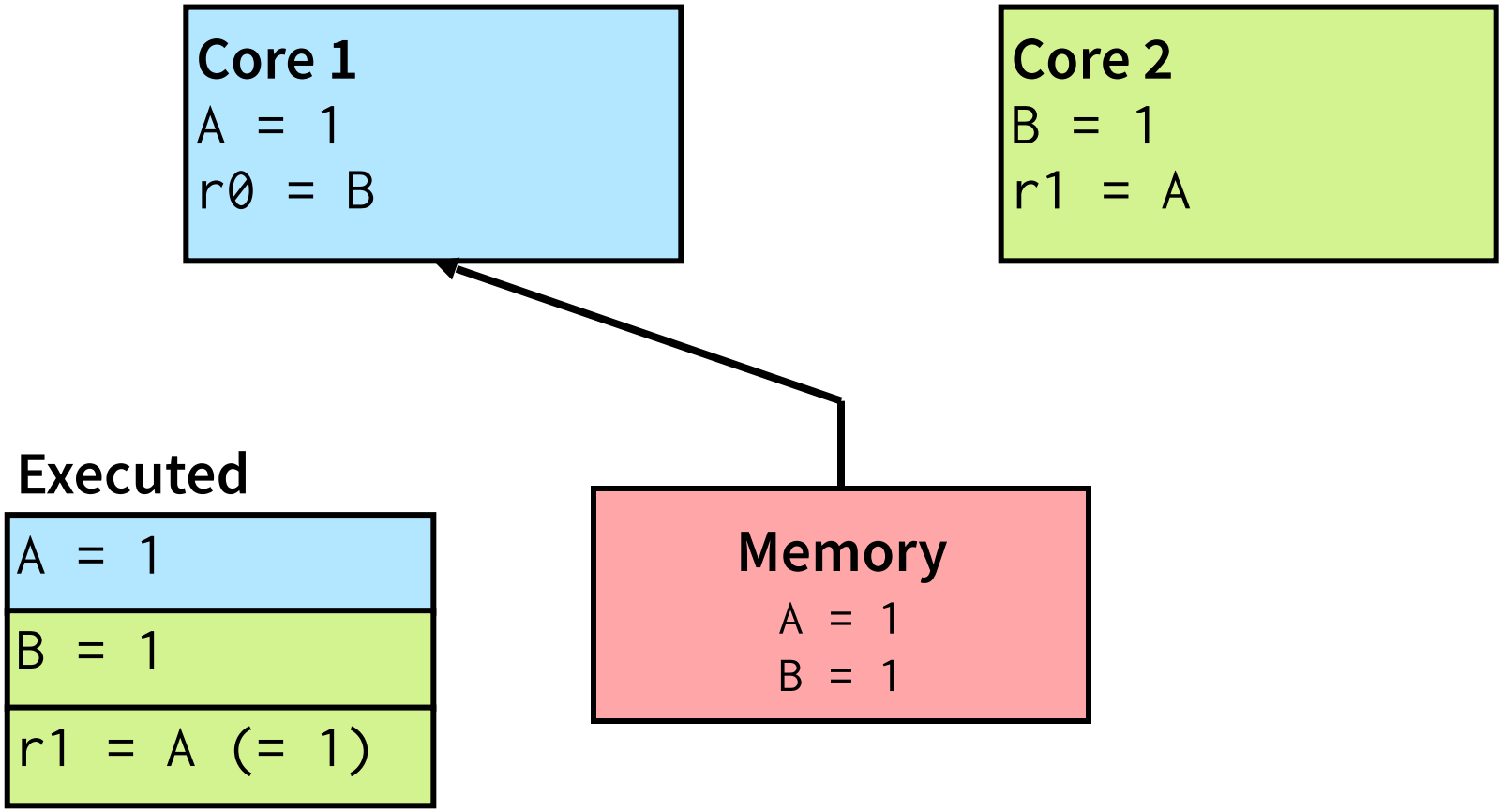
Sequential consistency

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



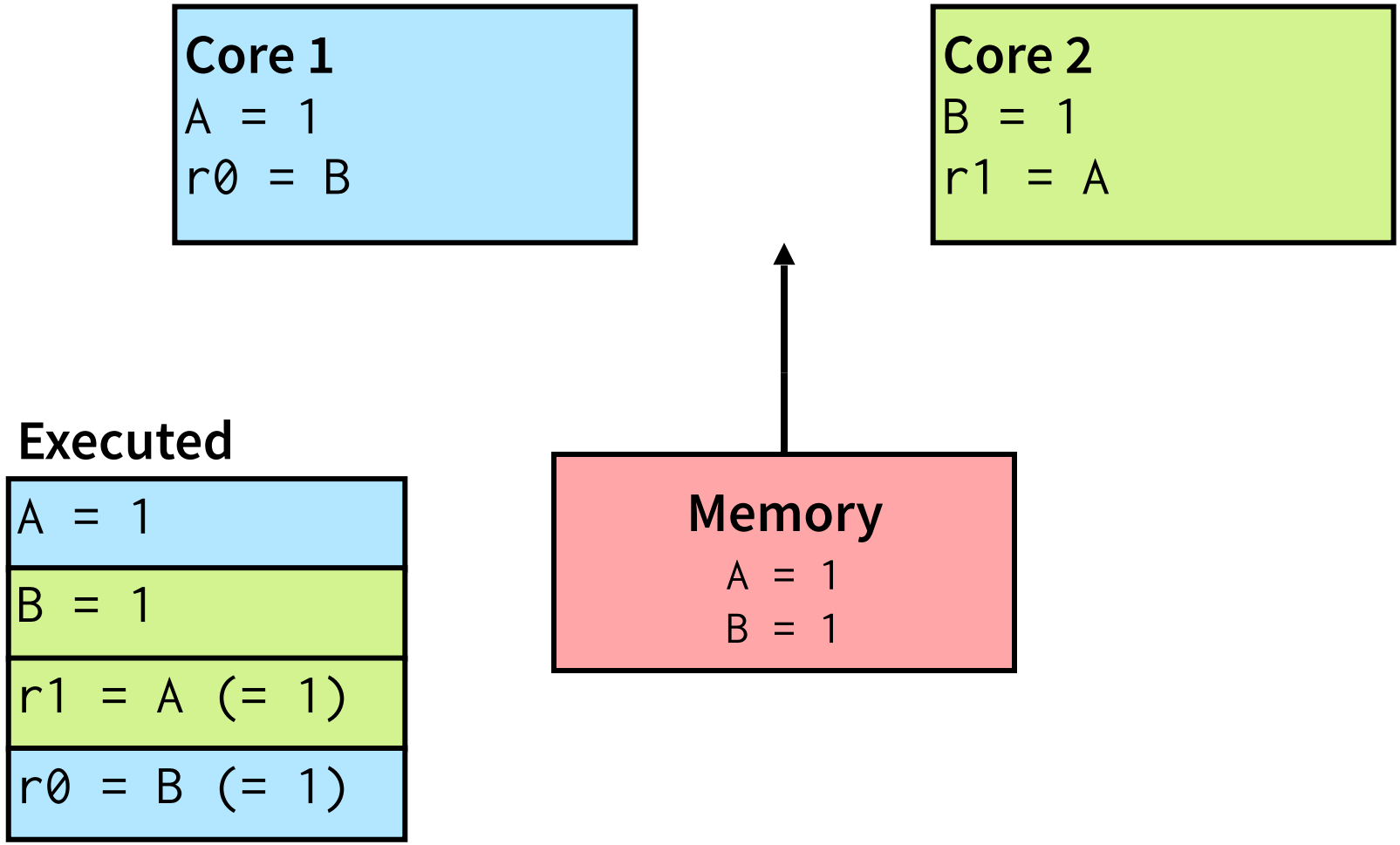
Sequential consistency

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

Two invariants:

- All operations executed in some sequential order
- Each thread's operations happen in program order

Says nothing about **which** order all operations happen in

- Any interleaving of threads is allowed

- Due to Leslie Lamport in 1979

Memory consistency models

- A memory consistency 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 model: allows the fewest reorderings
  - A brief tangent on distributed systems…

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

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

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (1) → (2) → (3) → (4)

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (1) → (2) → (3) → (4)

Can `r0 = 0` and `r1 = 1`? (

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (1) → (2) → (3) → (4)

Can `r0 = 0` and `r1 = 1`? (1) → (3) → (4) → (2)

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (1) → (2) → (3) → (4)

Can `r0 = 0` and `r1 = 1`? (1) → (3) → (4) → (2)

Can `r0 = 1` and `r1 = 0`? No...

Pop Quiz!

Assume sequential consistency, and all variables are initially 0.

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

Can `r0 = 0` and `r1 = 0`? (3) → (4) → (1) → (2)

Can `r0 = 1` and `r1 = 1`? (1) → (2) → (3) → (4)

Can `r0 = 0` and `r1 = 1`? (1) → (3) → (4) → (2)

Can `r0 = 1` and `r1 = 0`? No!

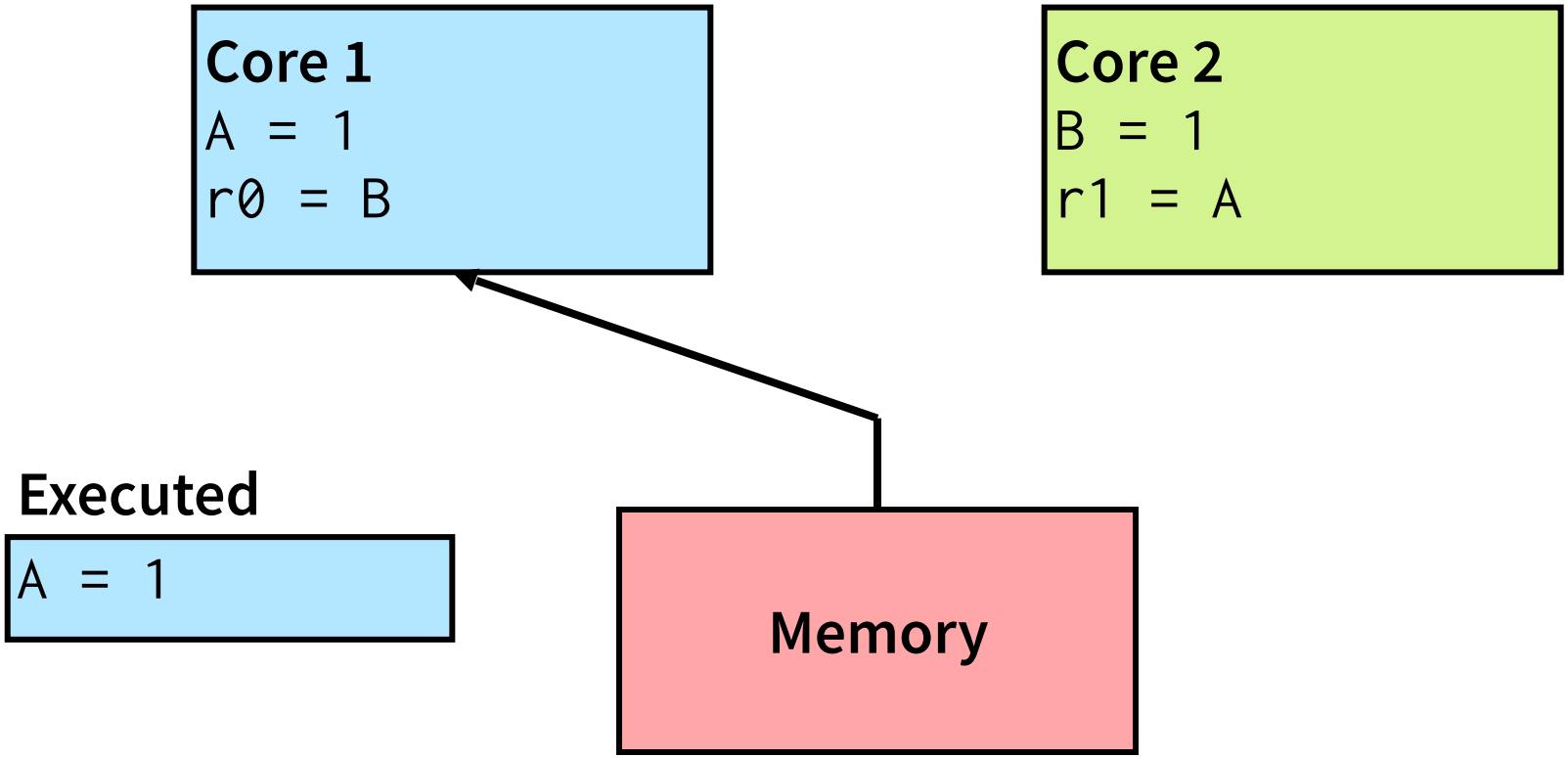
Why sequential consistency?

- Agrees with programmer intuition!

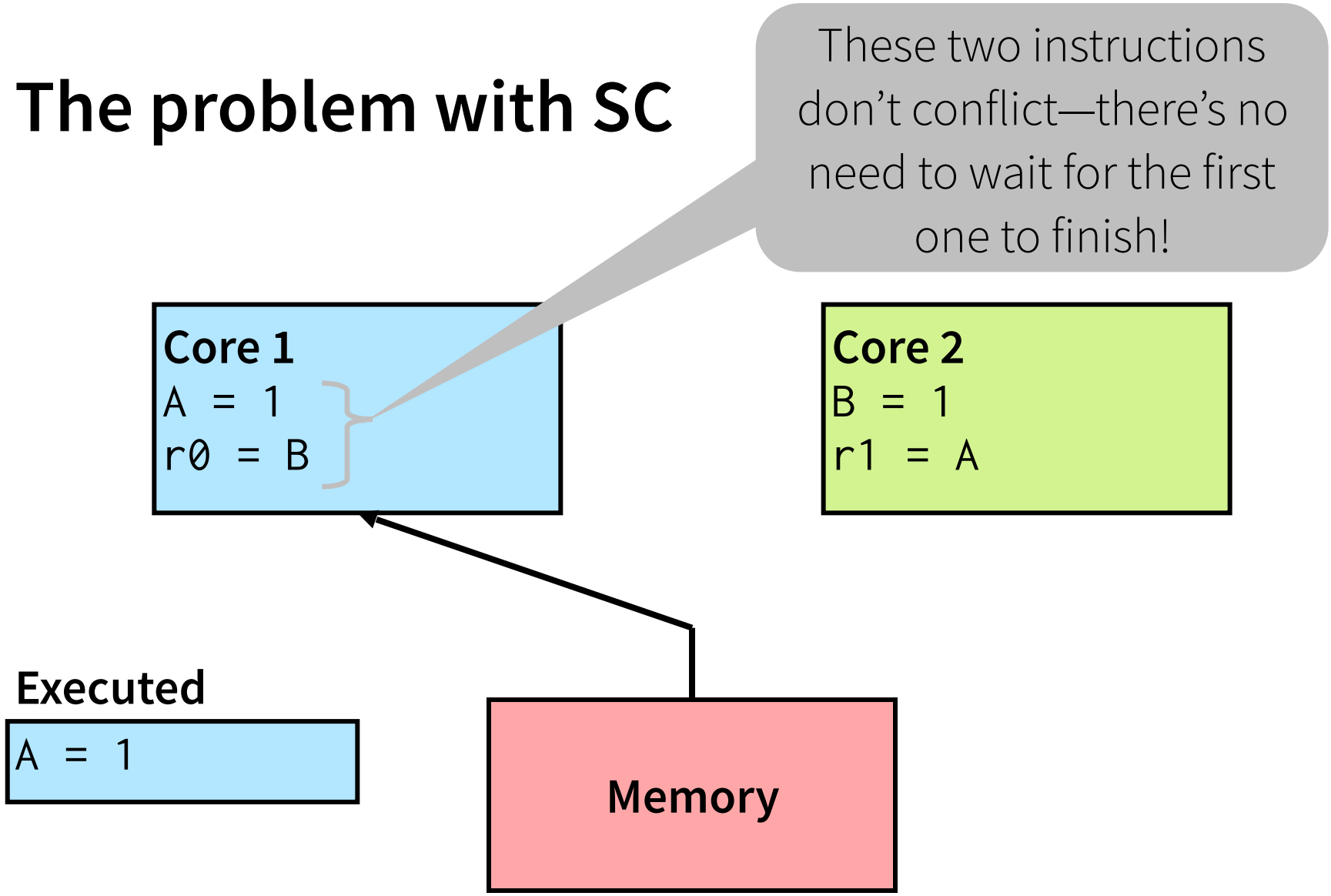
Why <u>not</u> sequential consistency?

- *Horribly slow* to guarantee in hardware
    - The “switch” model is overly conservative

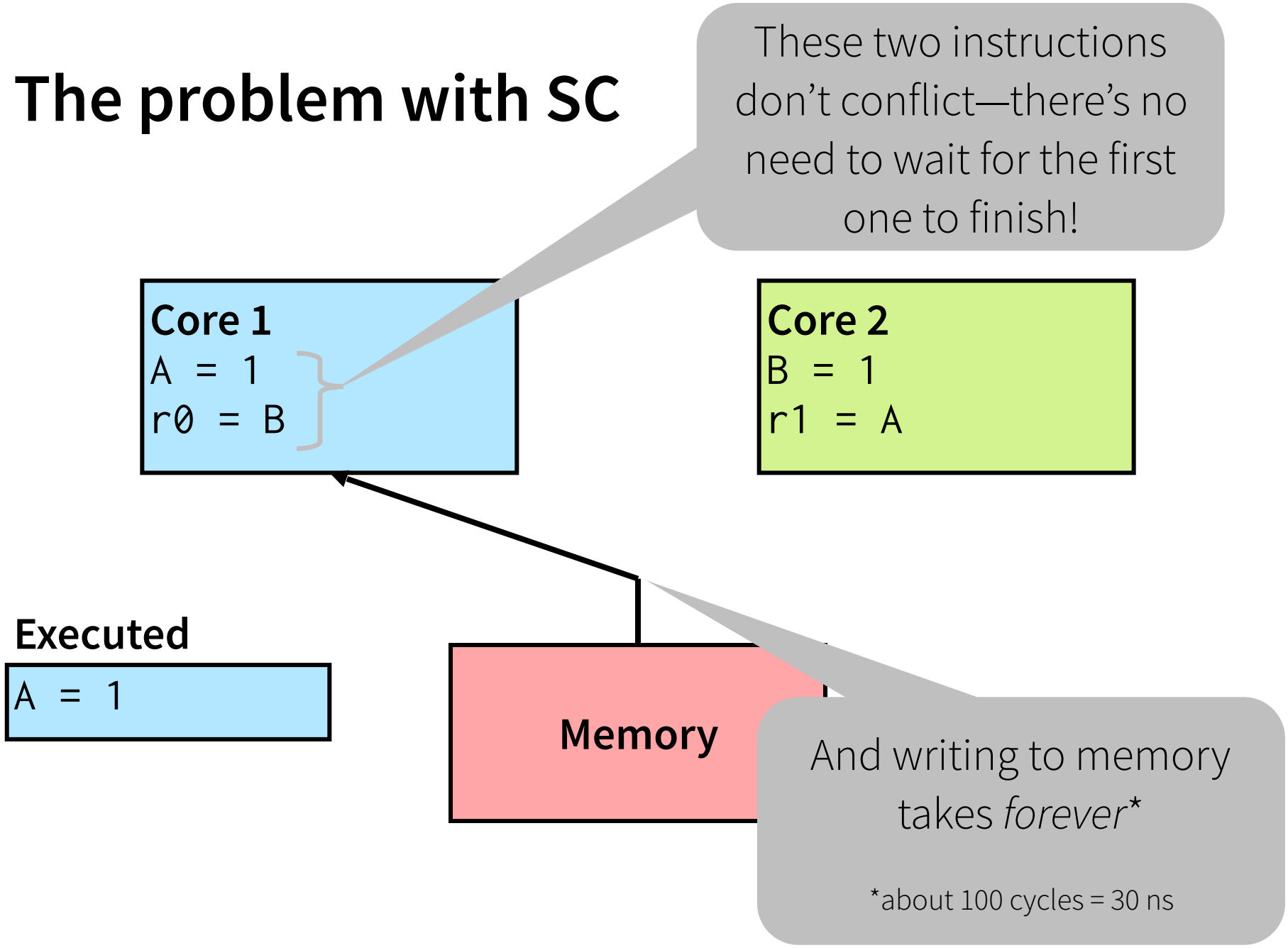
The problem with SC



The problem with SC



The problem with SC

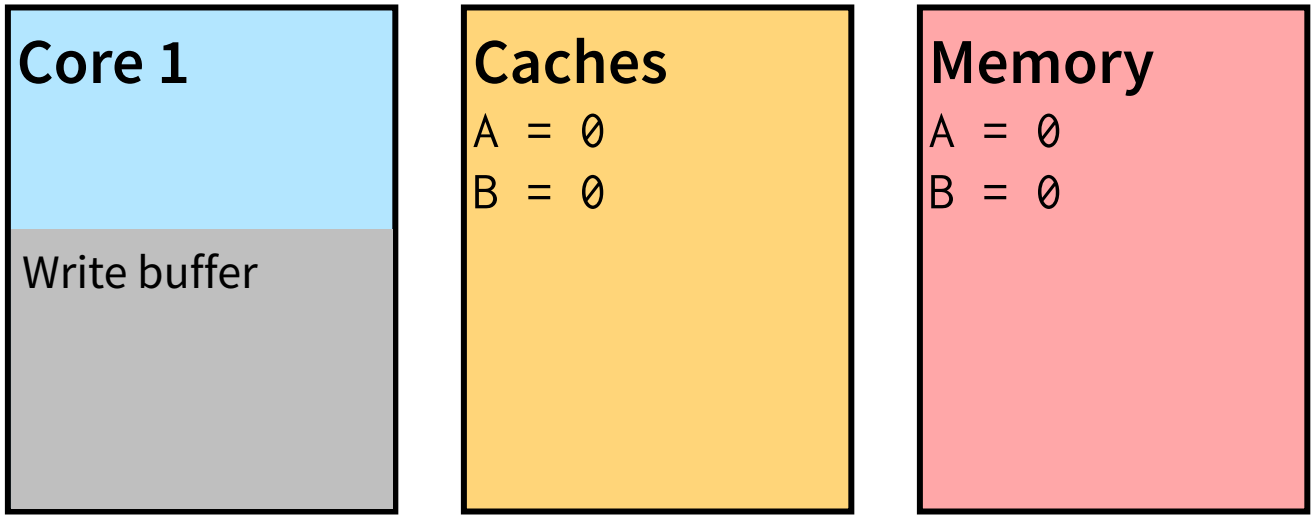


Optimization: Write buffers

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

Thread 1

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

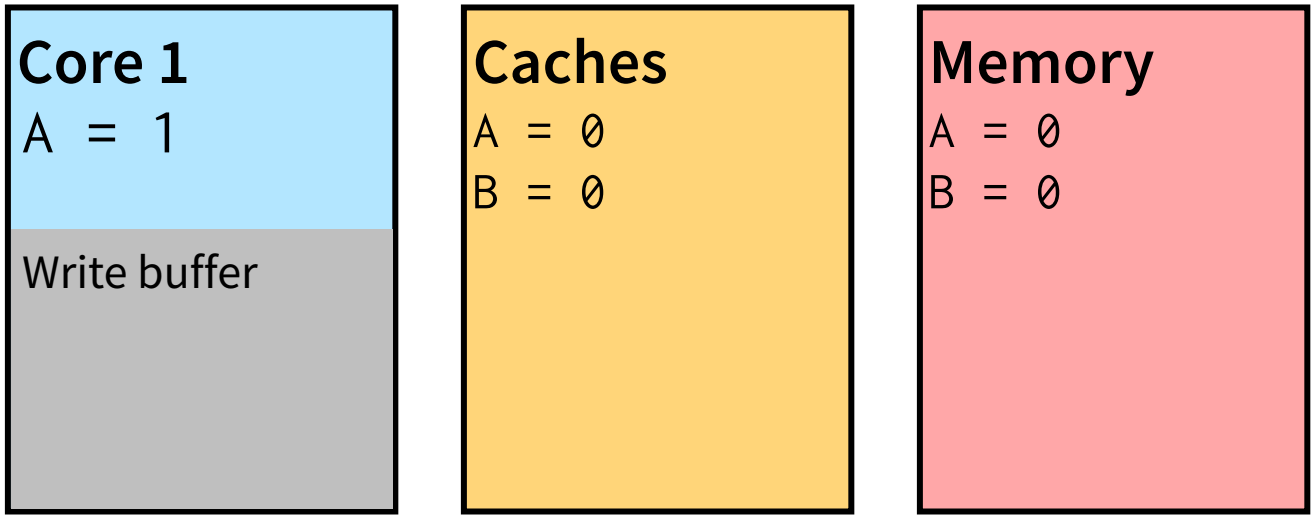


Optimization: Write buffers

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

Thread 1

`r0 = B`

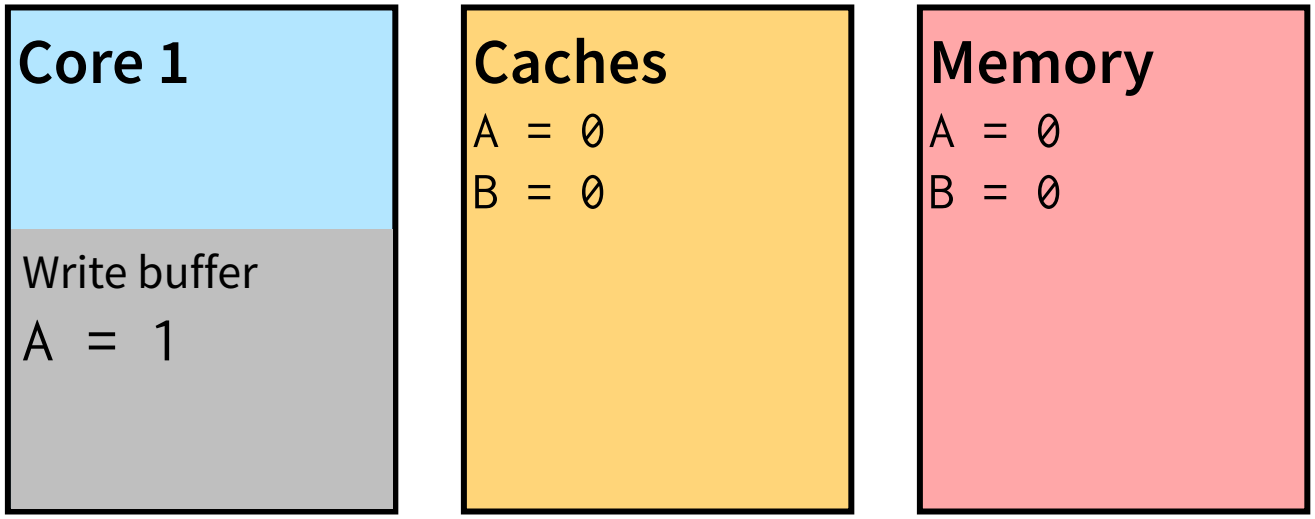


Optimization: Write buffers

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

Thread 1

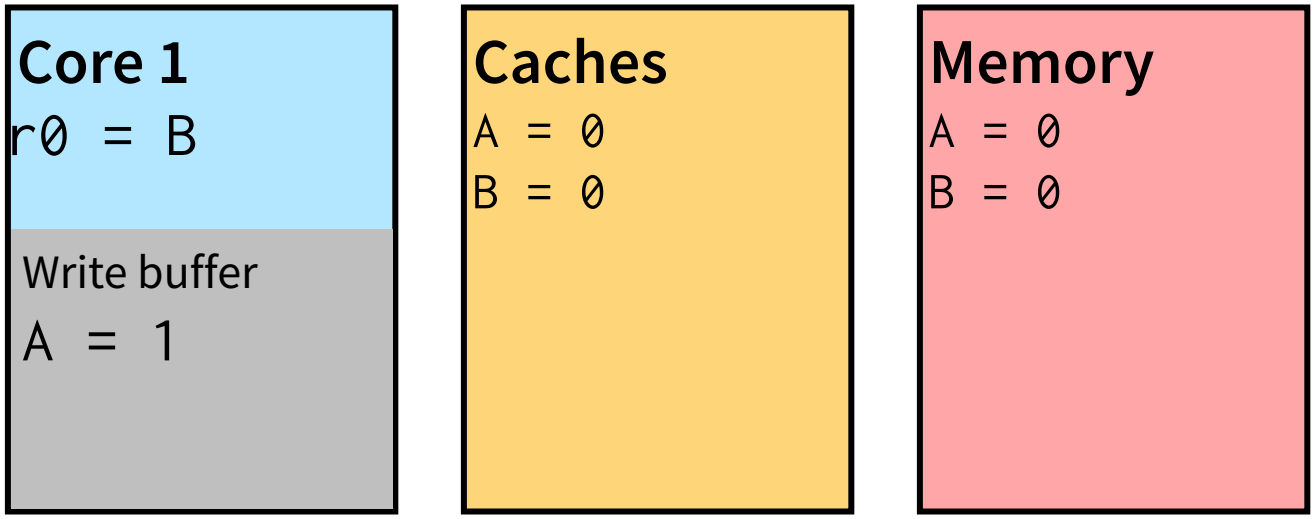
```
r0 = B
```



Optimization: Write buffers

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

Thread 1

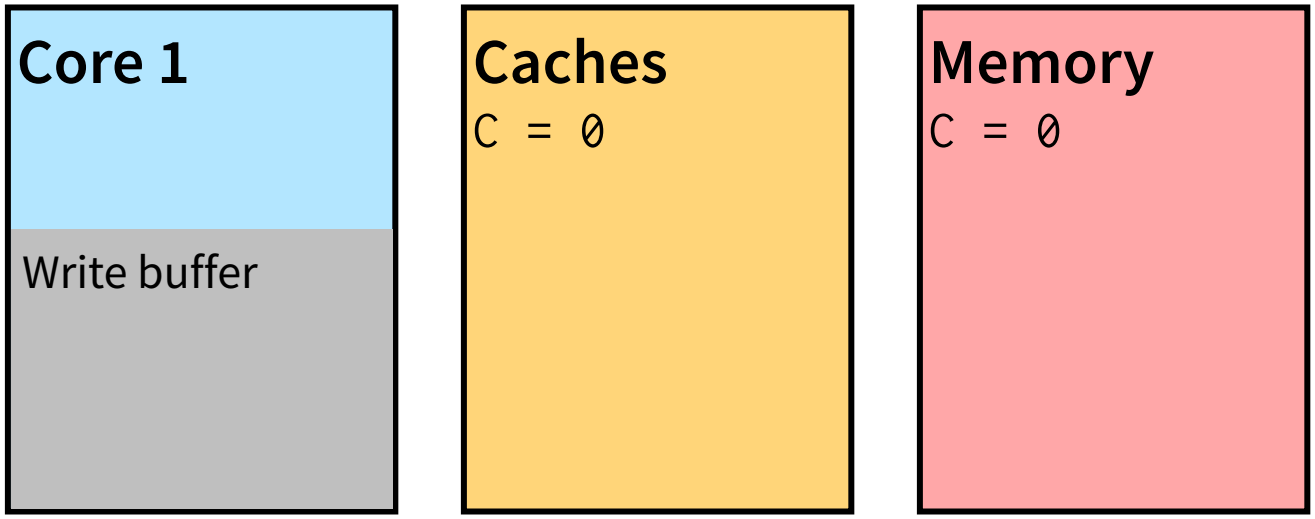


Optimization: Write buffers

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

Thread 1

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

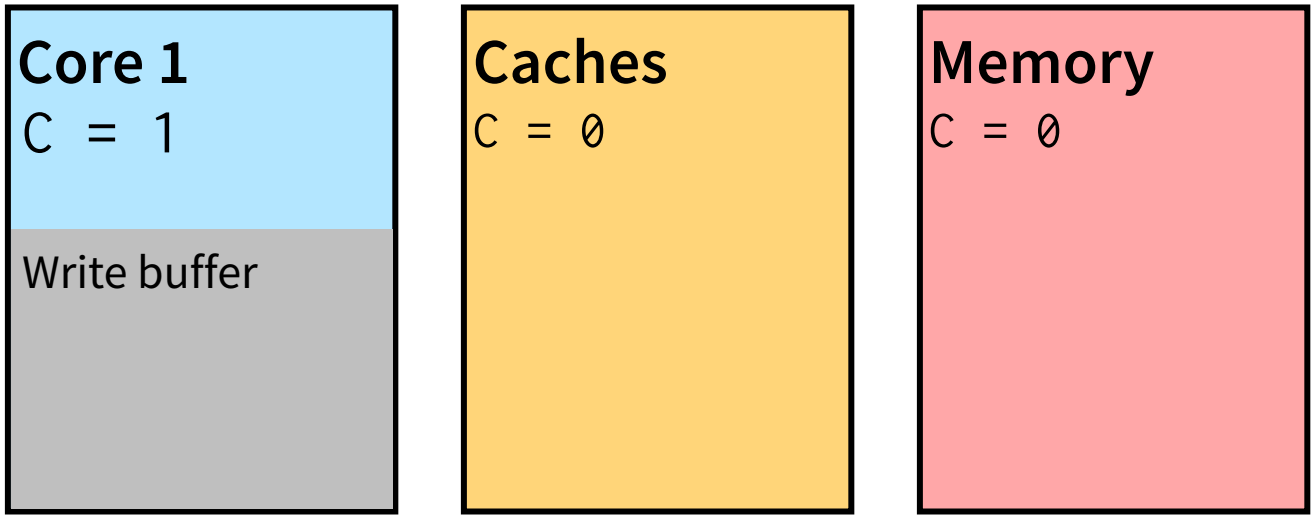


Optimization: Write buffers

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

Thread 1

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

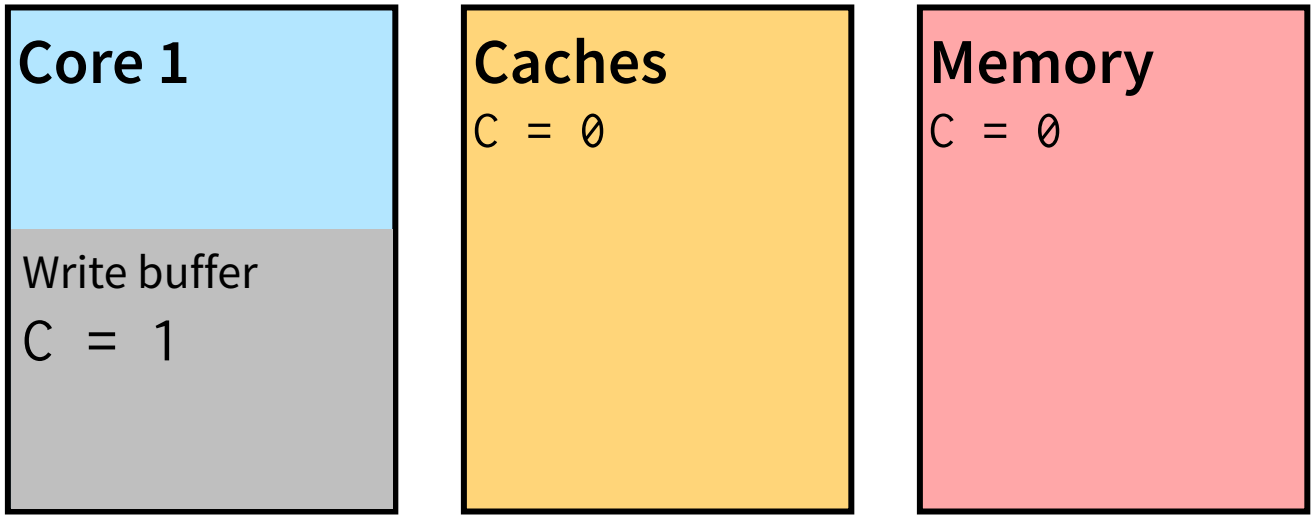


Optimization: Write buffers

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

Thread 1

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

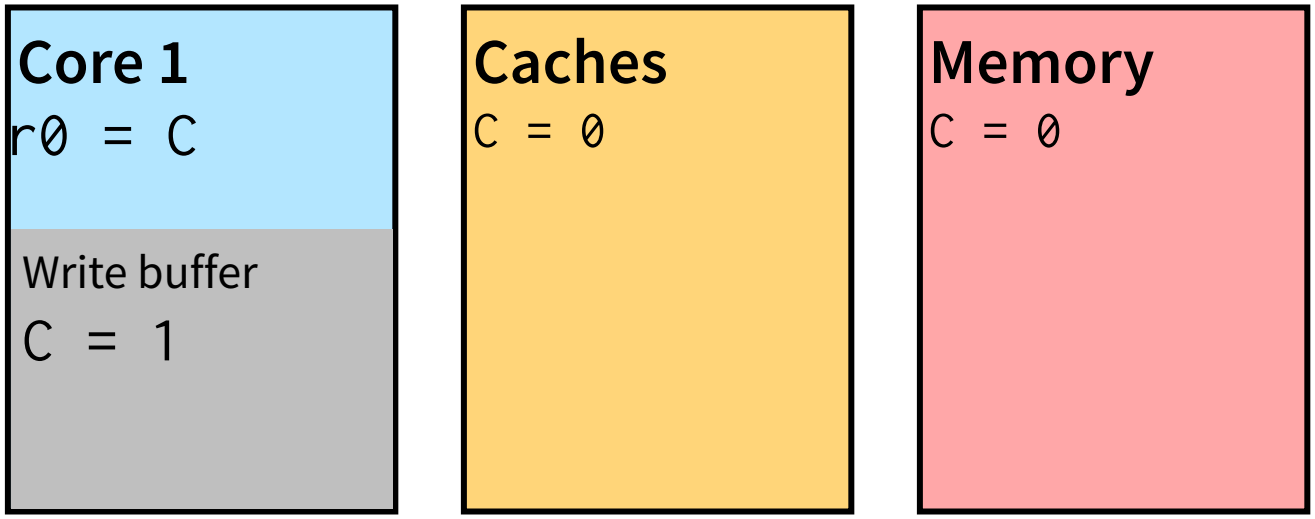


Optimization: Write buffers

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

Thread 1

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

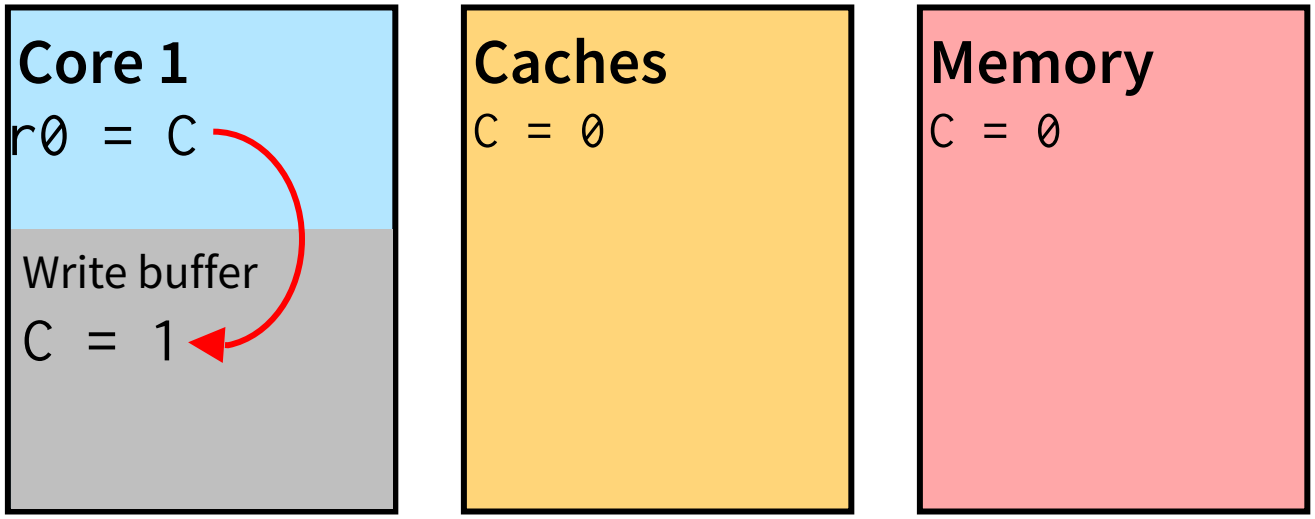


Optimization: Write buffers

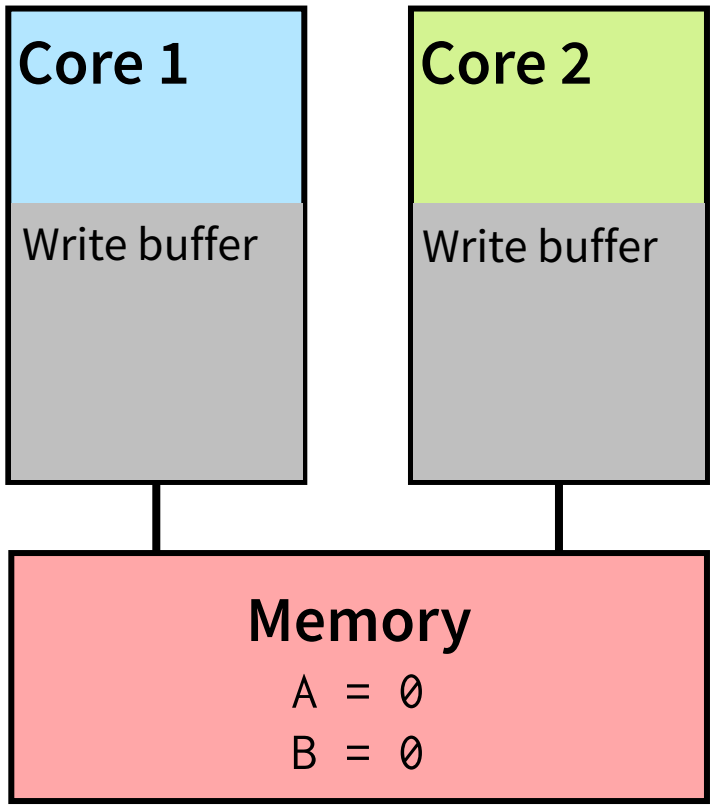
- Store writes in a local buffer and then proceed to next instruction immediately
- The cache will pull writes out of the write buffer when it's ready

Thread 1

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



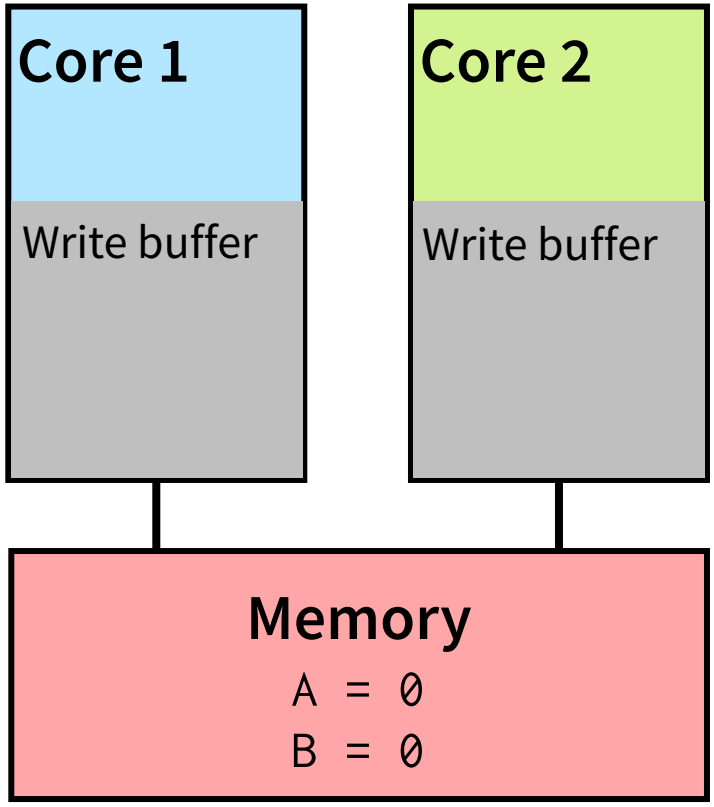
Write buffers change memory behavior

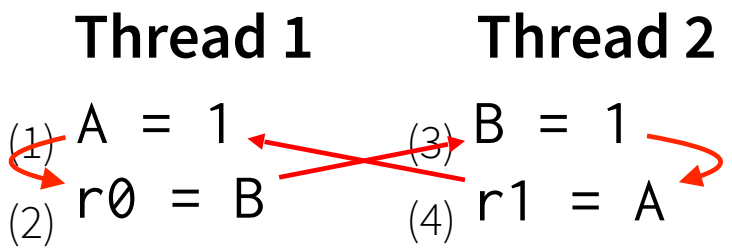


Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

Write buffers change memory behavior

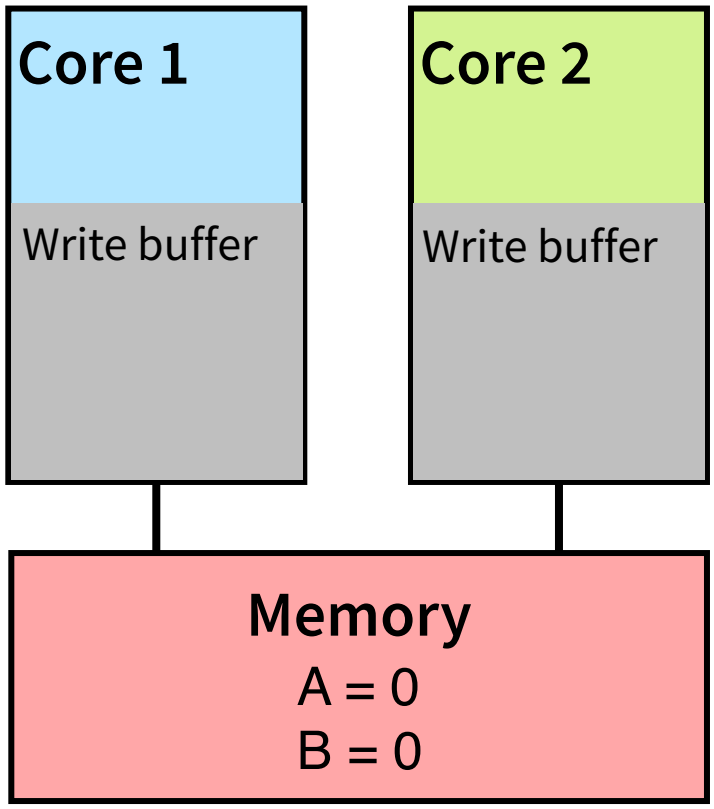




Can `r0 = 0` and `r1 = 0`?

SC: No!

Write buffers change memory behavior



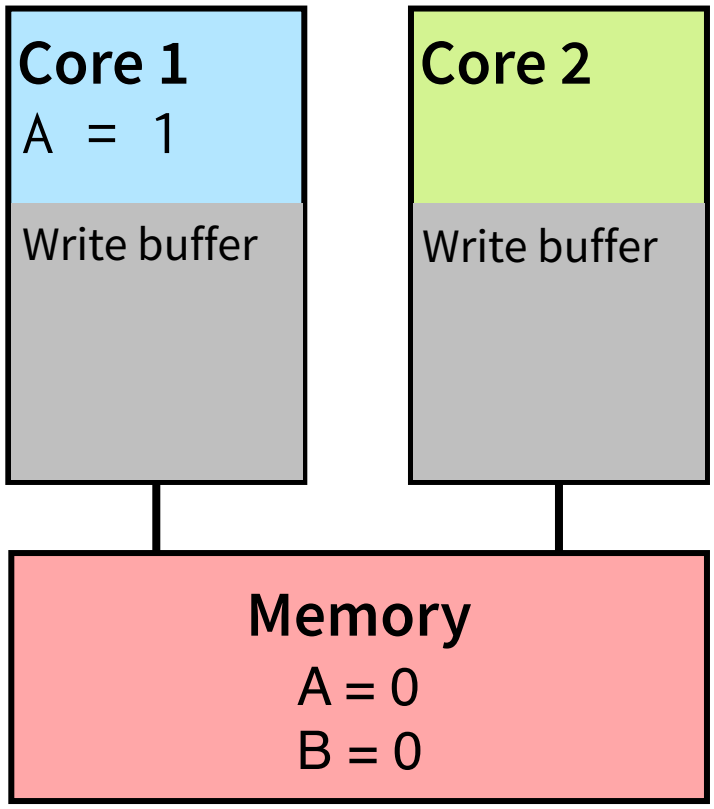
Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



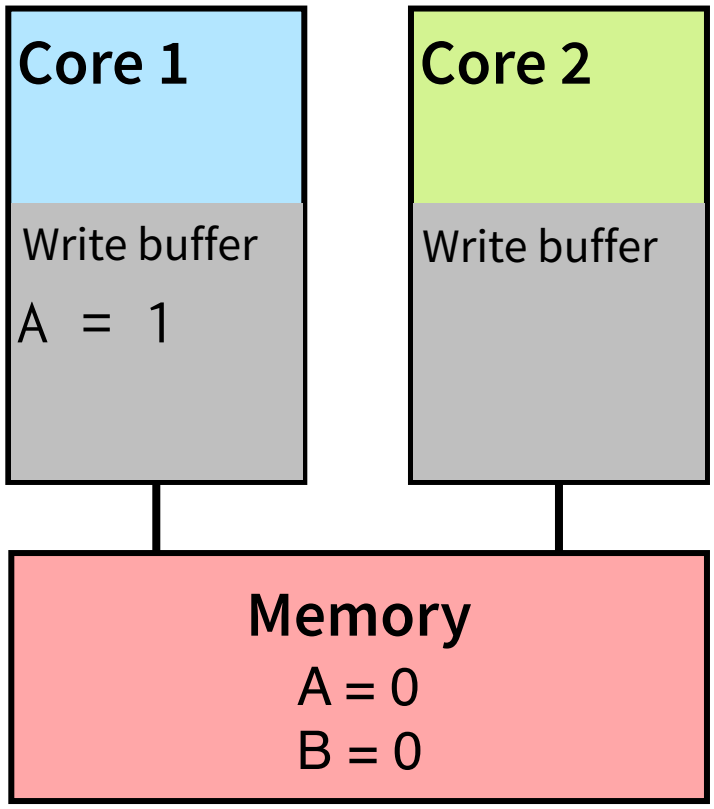
Thread 1	Thread 2
(1)	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



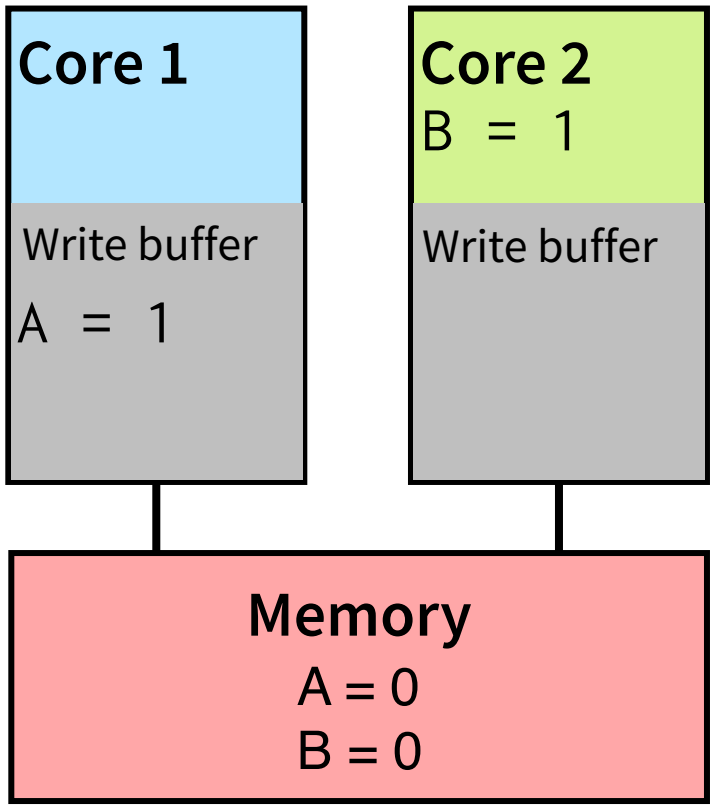
Thread 1	Thread 2
(1)	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



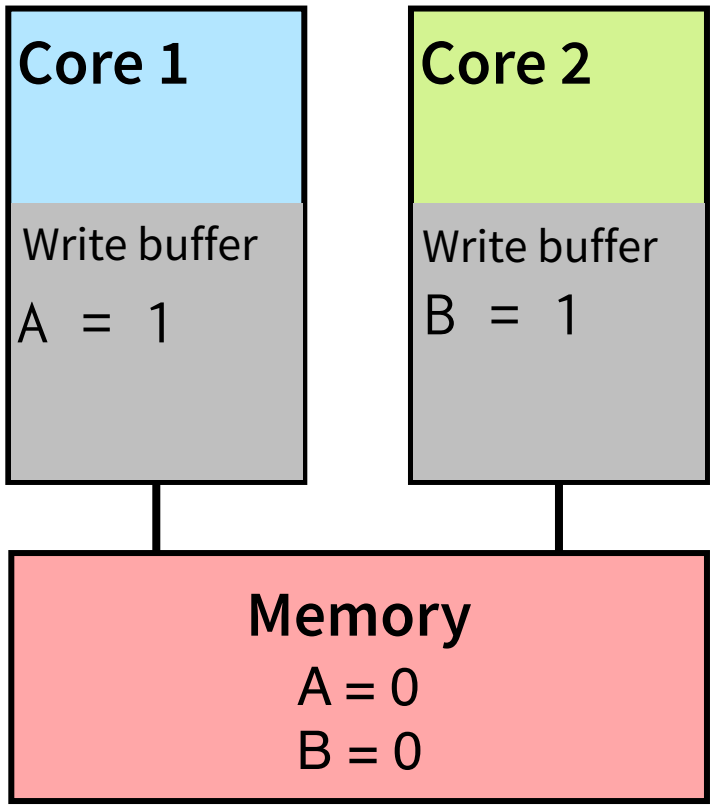
Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



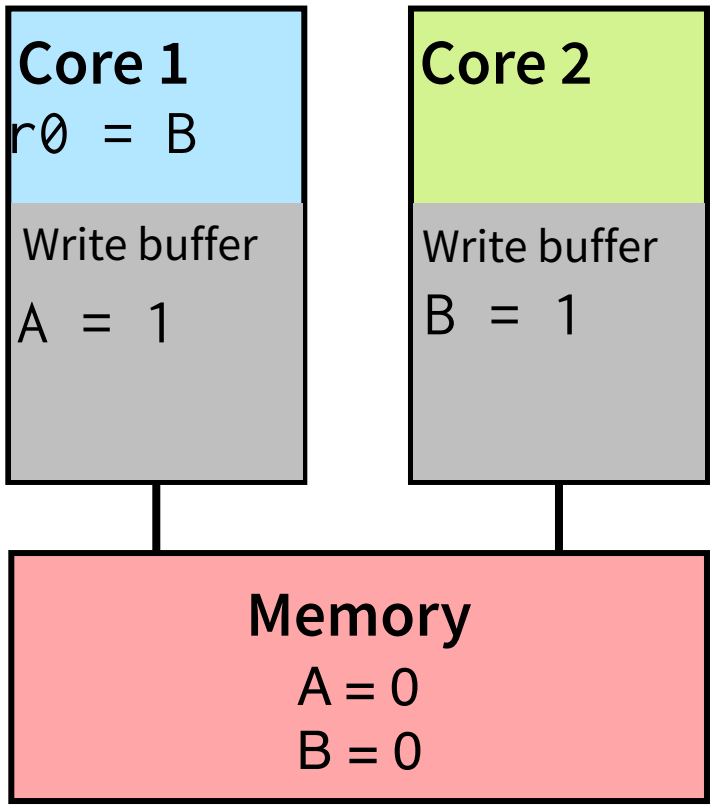
Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



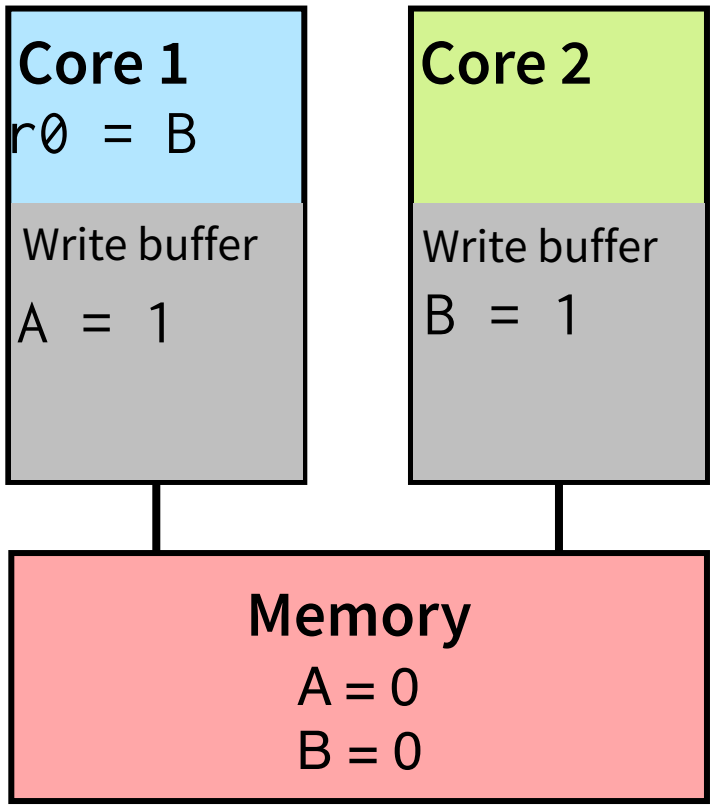
Thread 1	Thread 2
(1)	(3)
(2)	(4) `r1 = A`

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

SC: No!

Executed

Write buffers change memory behavior



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

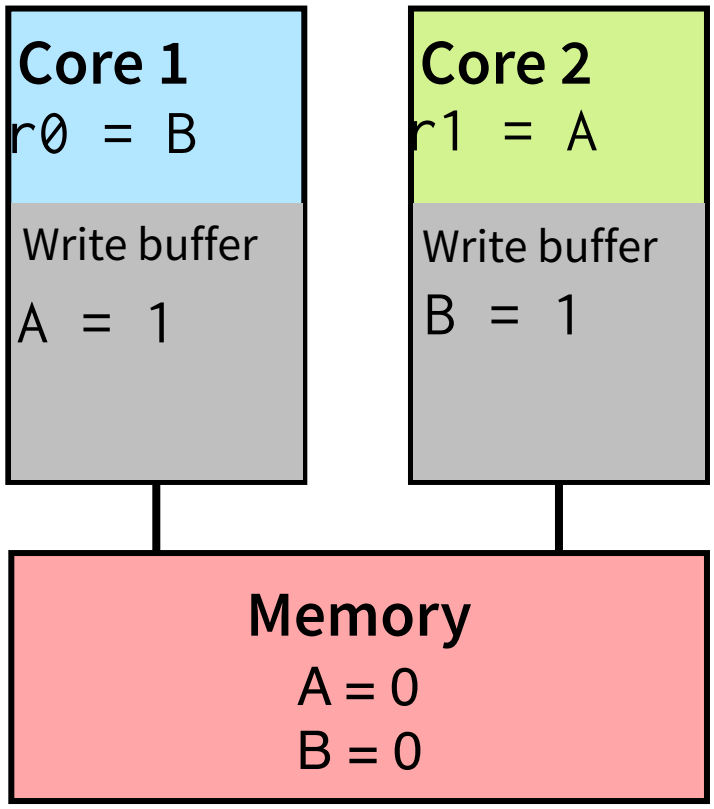
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Write buffers change memory behavior



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

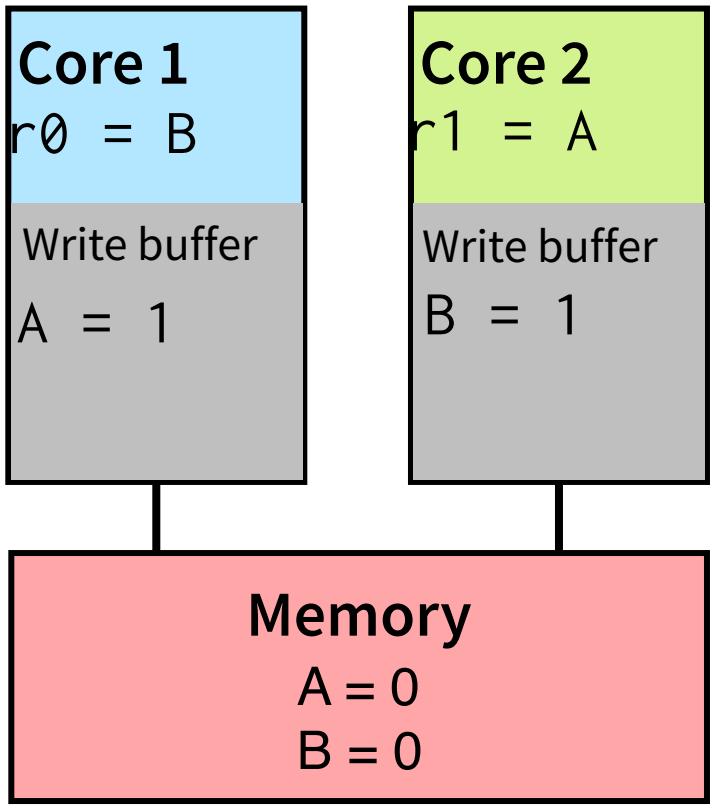
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Write buffers change memory behavior



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

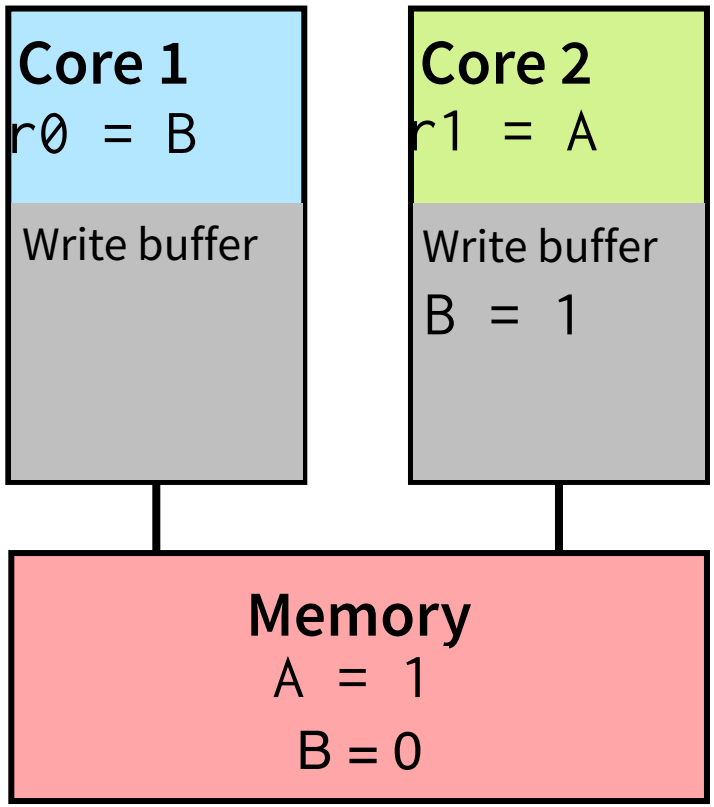
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Write buffers change memory behavior



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

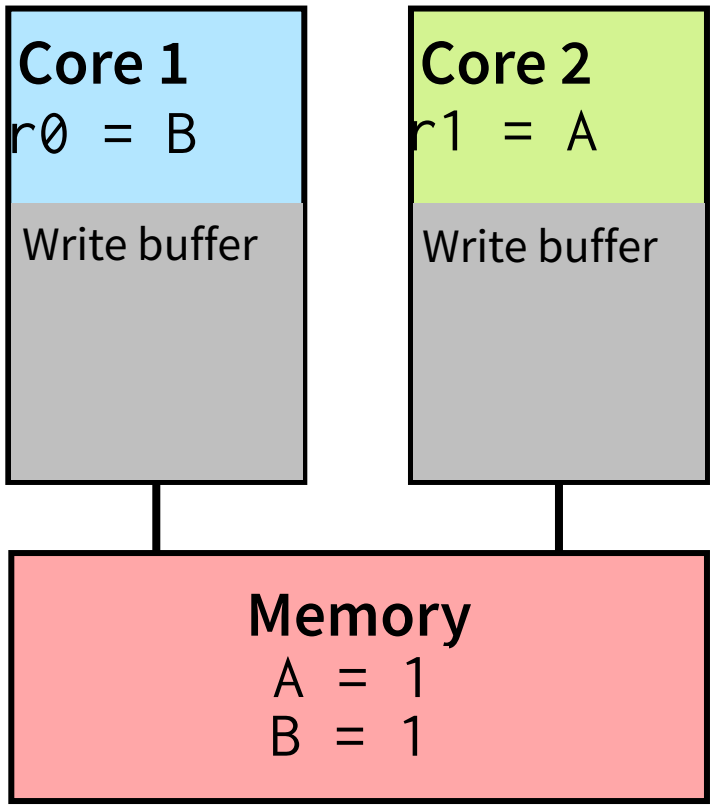
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Write buffers change memory behavior



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

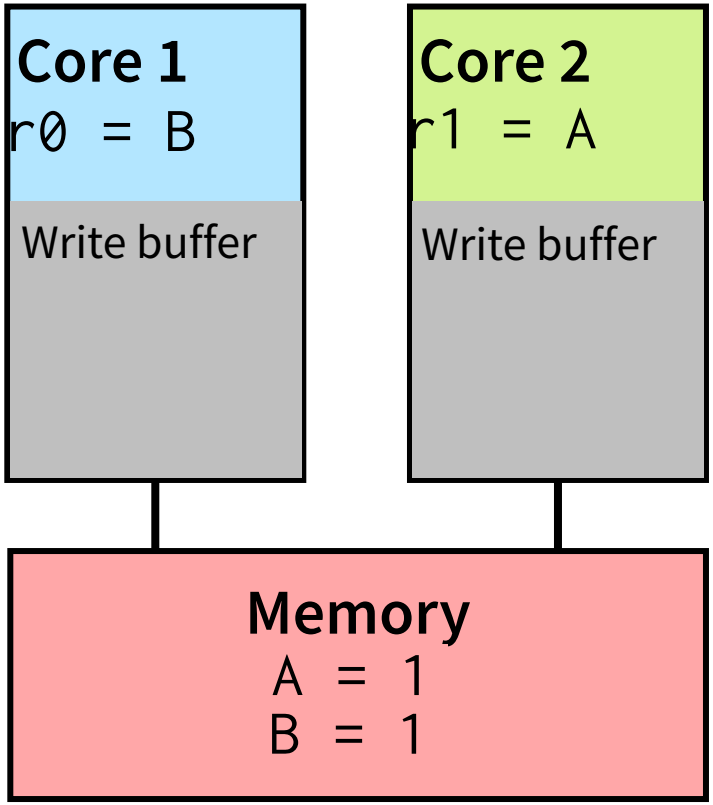
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Write buffers change memory behavior



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

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

SC: No! Write buffers: Yes!

Executed

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

So, who uses write buffers?

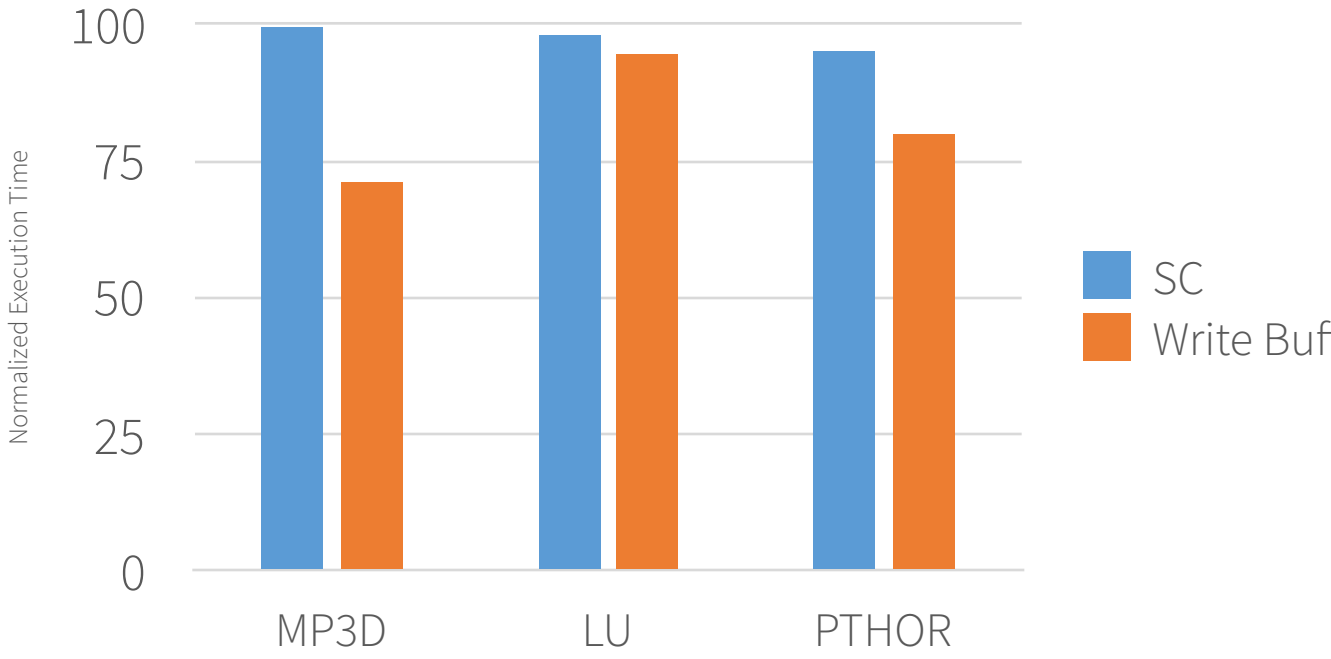
Every modern CPU!

- x86
- ARM
- PowerPC
- …

So, who uses write buffers?

Every modern CPU!

- x86
- ARM
- PowerPC
- …



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

Total Store Ordering (TSO)

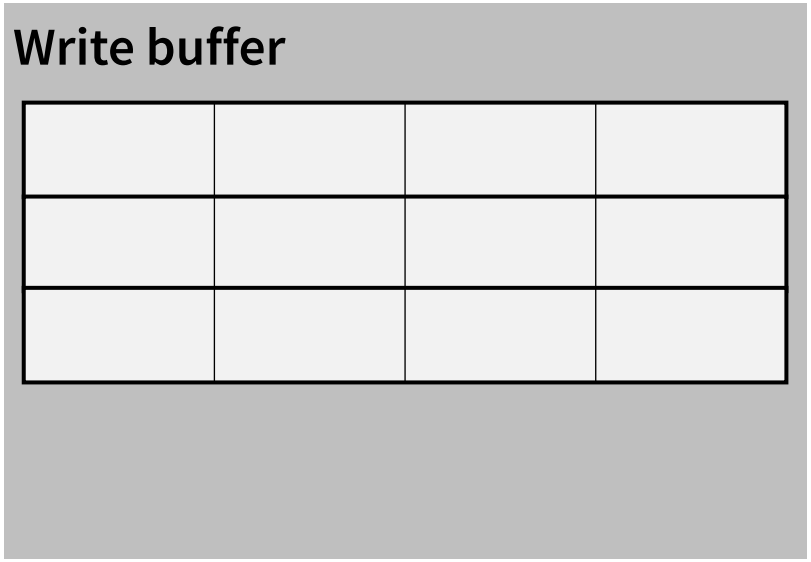
- Sequential consistency plus write buffers
- Allows more behaviors than SC
  - Harder to program!
- x86 specifies TSO as its memory model

More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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

More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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



Thread 1

X = 1
Y = 1
Z = 1

Assume X and Z are on the same cache line

Executed

More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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

Write buffer

X = 1			
	Y = 1		
		Z = 1	

Thread 1

X = 1
Y = 1
Z = 1

Assume X and Z are on the same cache line

Executed

More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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

Write buffer

X = 1		Z = 1	
	Y = 1		

Thread 1

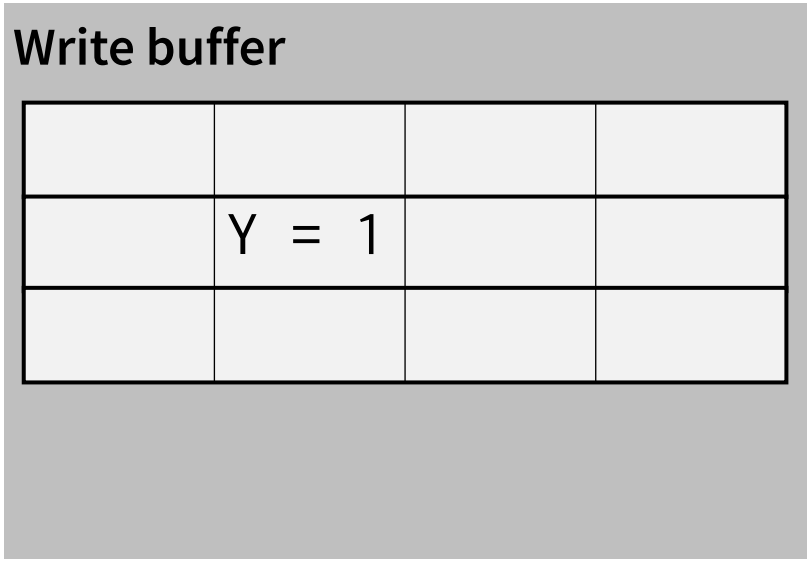
X = 1
Y = 1
Z = 1

Assume X and Z are on the same cache line

Executed

More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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

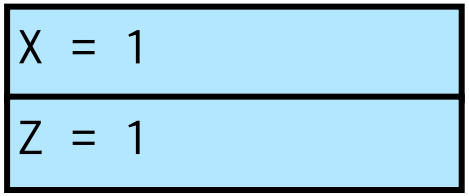


Thread 1

X = 1
Y = 1
Z = 1

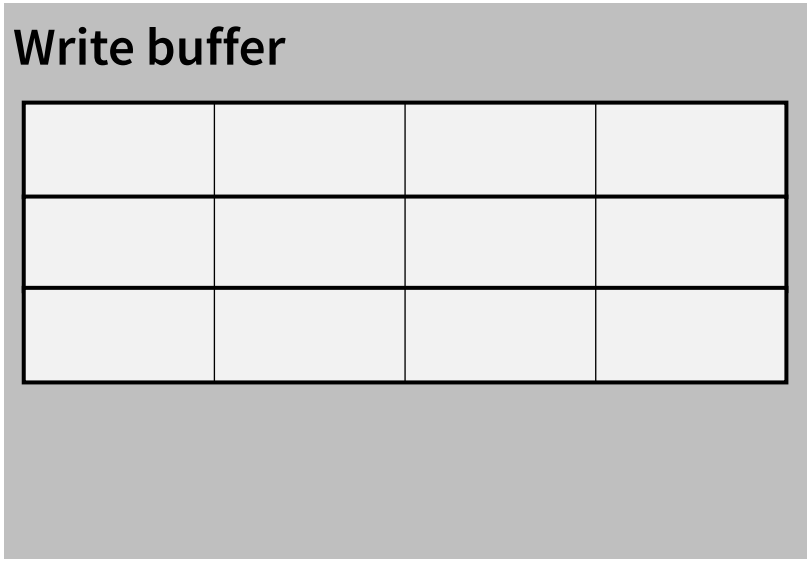
Assume X and Z are on the same cache line

Executed



More esoteric memory models

- Partial Store Ordering (used by SPARC)
  - 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



Thread 1

X = 1
Y = 1
Z = 1

Assume X and Z are on the same cache line

Executed

X = 1
Z = 1
Y = 1

More esoteric memory models

- Weak ordering (ARM, PowerPC)
  - No guarantees about operations on data!
  - **Everything** can be reordered

This seems like a nightmare!

- Every architecture provides synchronization primitives to make memory ordering stricter
  - Fence instructions prevent reorderings, but are expensive
  - Other synchronization primitives: read-modify-write/compare-and-swap, transactional memory, …

But it’s not just hardware…

Thread 1

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

But it’s not just hardware…

Thread 1

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

Thread 1

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

But it’s not just hardware…

Thread 1

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

Thread 1

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

But it’s not just hardware…

Thread 1

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

Thread 2

```
X = 0
```

Thread 1

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

Thread 2

```
X = 0
```

But it’s not just hardware…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

But it's not just hardware…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

11111000000…

But it's not just hardware…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

11111011111…

Thread 1

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

Thread 2

```
X = 0
```

11111111111…

11111000000…

Are computers broken?

- Every example so far has involved a **data race**
  - Two accesses to the same memory location
  - At least one is a write
  - Unordered by **synchronization operations**

- If there are no data races, reordering behavior doesn't matter
  - Accesses are ordered by synchronization, and synchronization forces sequential consistency
  - Note this is **not the same as determinism**

Memory models in the real world

- Modern (C11, C++11) and not-so-modern (Java 5) languages guarantee **sequential consistency for data-race-free programs** ("SC for DRF")
  - Compilers will insert the necessary synchronization to cope with the hardware memory model

- No guarantees if your program contains data races!
  - The intuition is that most programmers would consider a racing program to be buggy
  - Use a synchronization library!

“Reordering” in computer architecture

- Today: **memory consistency models**
  - Ordering of memory accesses to different locations
  - Visible to programmers!
- **Cache coherence protocols**
  - Ordering of memory accesses to the same location
  - Not visible to programmers
- **Out-of-order execution**
  - Ordering of execution of a single thread’s instructions
  - Significant performance gains from dynamically scheduling
  - Not visible to programmers

Memory consistency models

- Define the allowed reorderings of memory operations by hardware and compilers
- A **contract** between hardware/compiler and software
- Necessary for good performance?
  - Is 20% worth all this trouble?