Memory Consistency Models

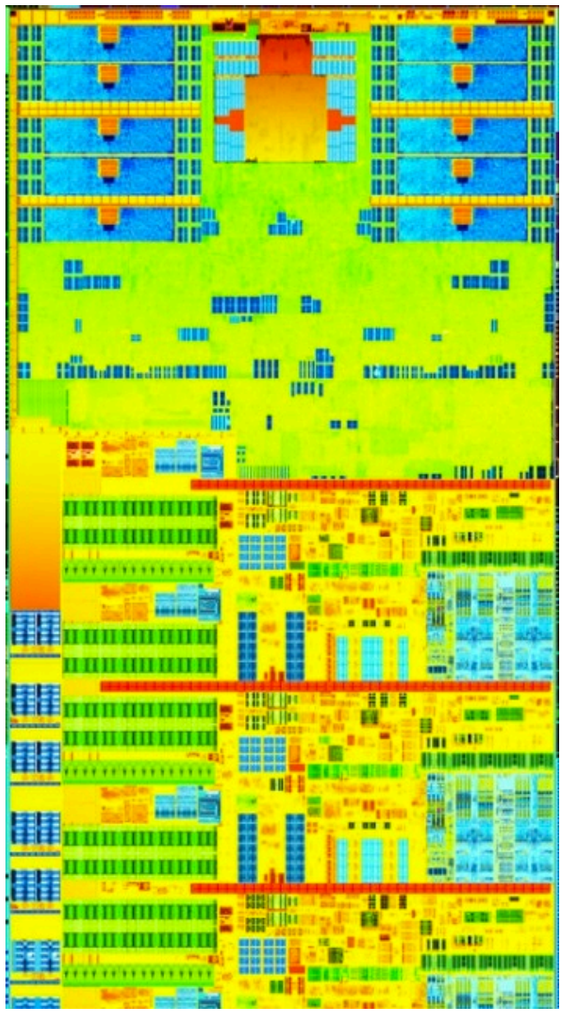
CSE 451

James Bornholt

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”?

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”?

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?

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”.

Thread 1

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

Thread 2

```
B = 1
if (A == 0)
  print “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!

Things that shouldn’t happen

This program should never print “Hello World”.

Thread 1

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

Thread 2

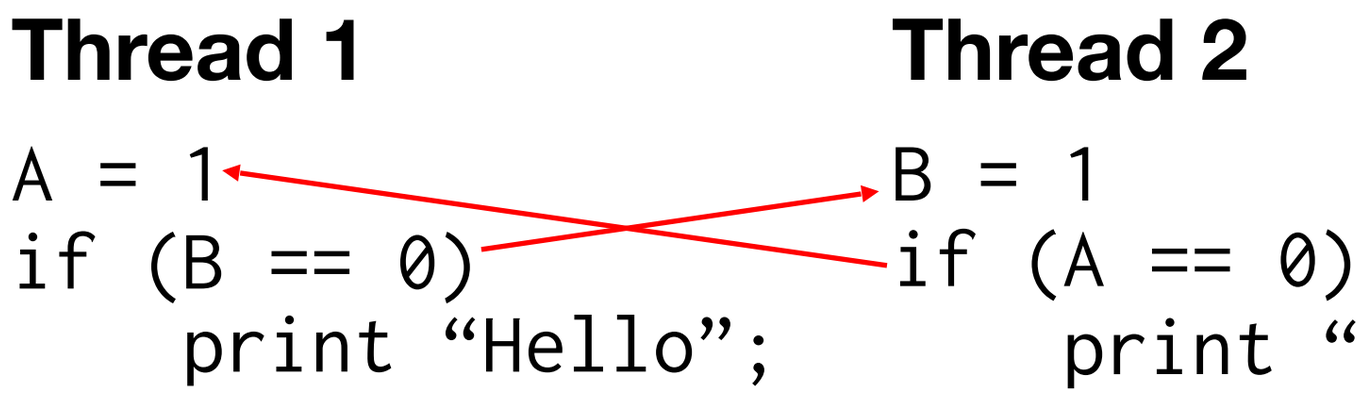
```
B = 1
if (A == 0)
  print “World”;
```

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!

Things that shouldn’t happen

This program should never print “Hello World”.

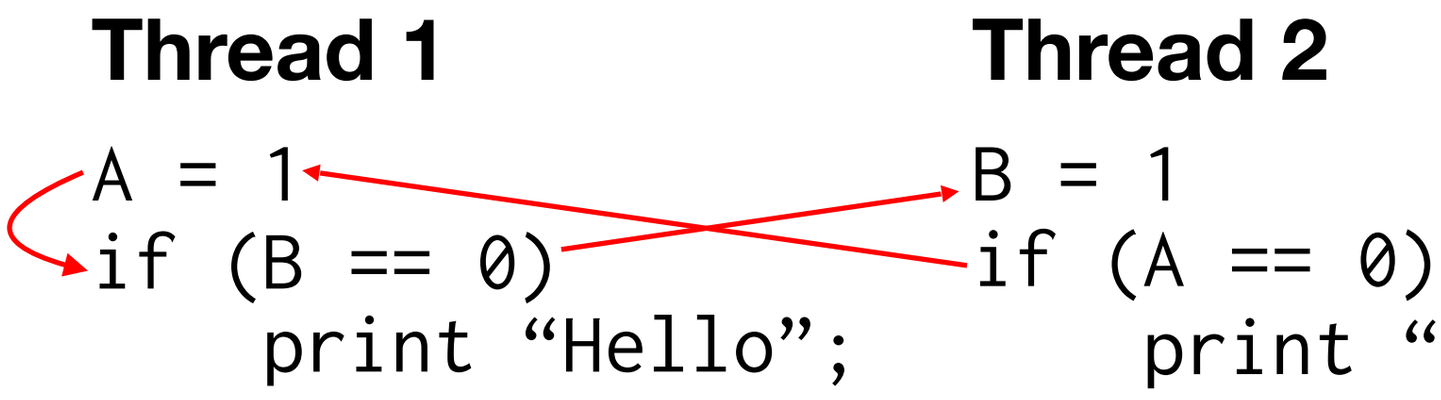


A “happens-before” graph shows the order in which events must execute to get a desired outcome.

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

Things that shouldn’t happen

This program should never print “Hello World”.

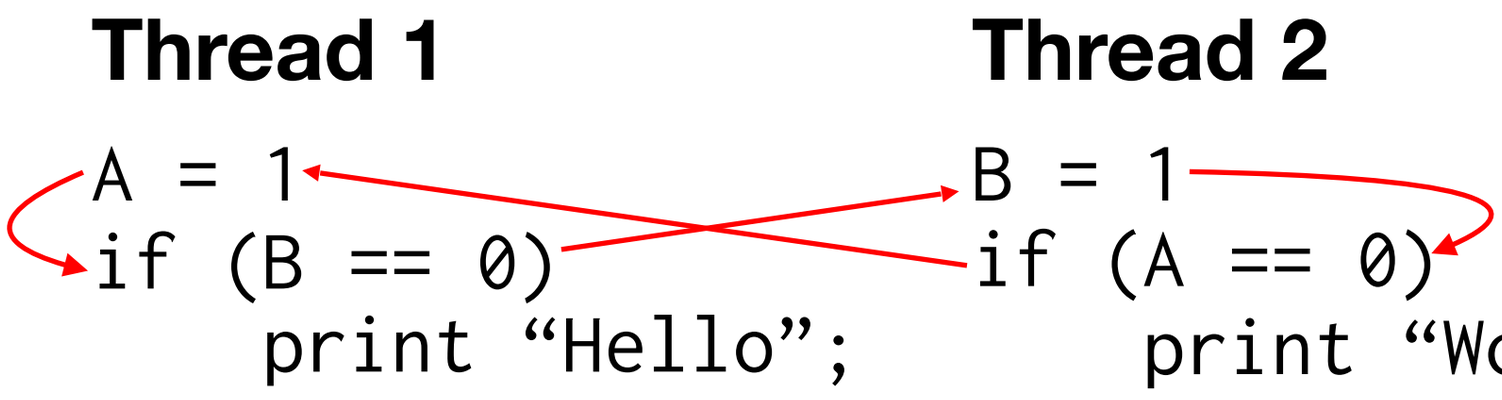


A “happens-before” graph shows the order in which events must execute to get a desired outcome.

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

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!

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”.

Thread 1

```
A = 1
r0 = B
if (r0 == 0)
    print “Hello”;
```

Thread 2

```
B = 1
r1 = A
if (r1 == 0)
    print “World”;
```

Things that shouldn’t happen

This program should never print “Hello World”.

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

Things that shouldn’t happen

This program should never print “Hello World”.

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

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

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

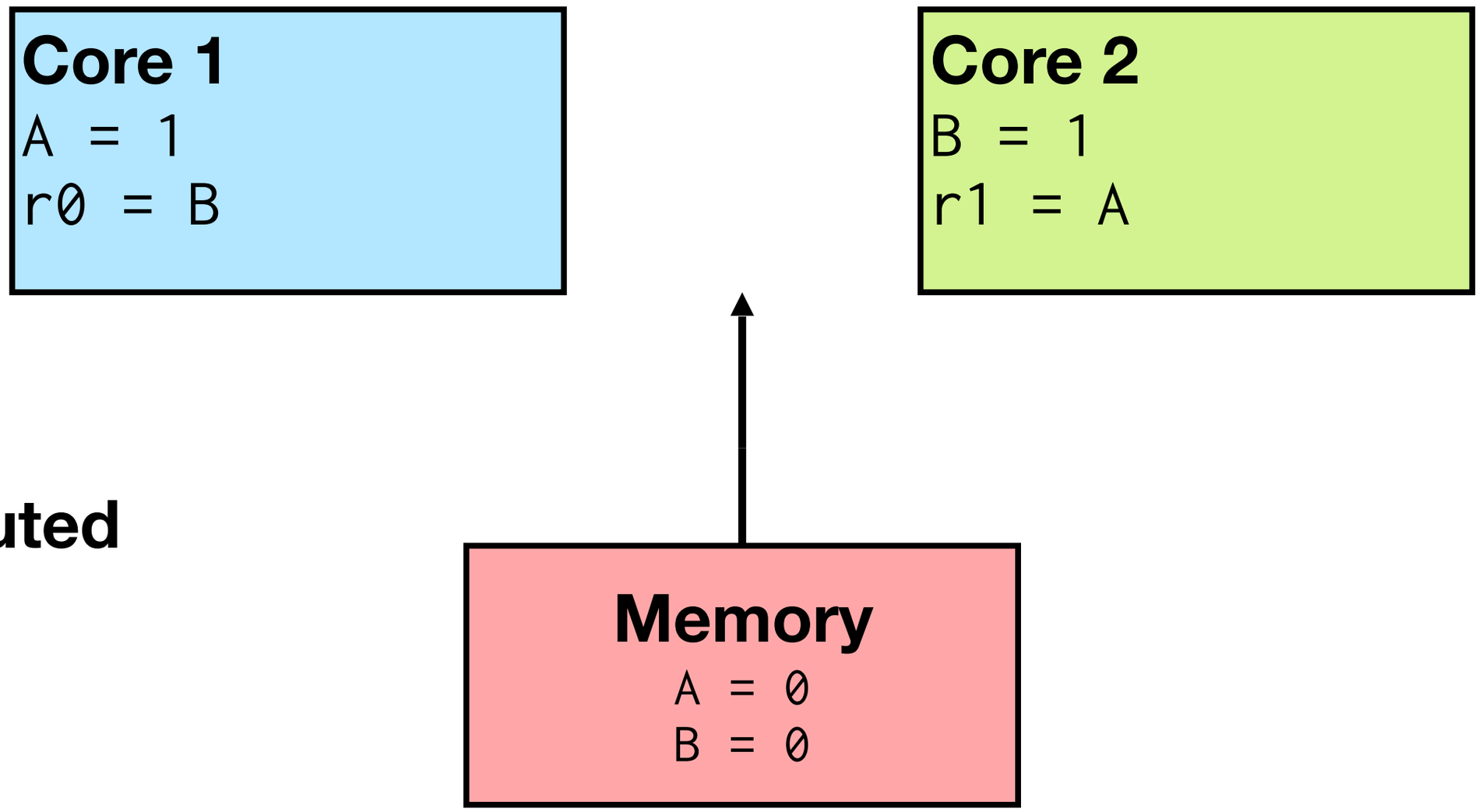
Thread 1	Thread 2
`A = 1`	`B = 1`
`r0 = B`	`r1 = A`

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

(This is the interleaving model you probably remember from 332)

Sequential consistency

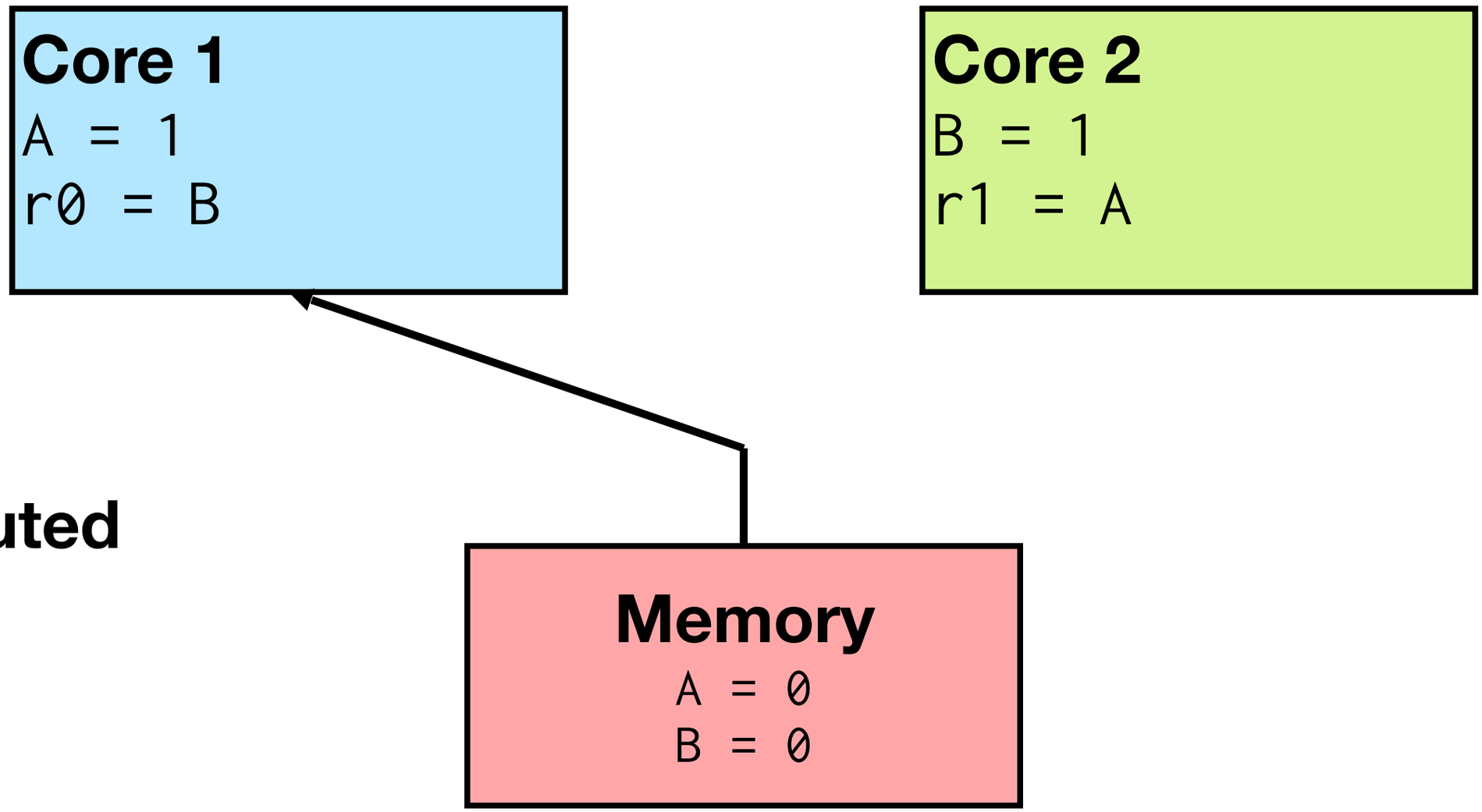
Can be seen as a “switch” running one instruction at a time



Executed

Sequential consistency

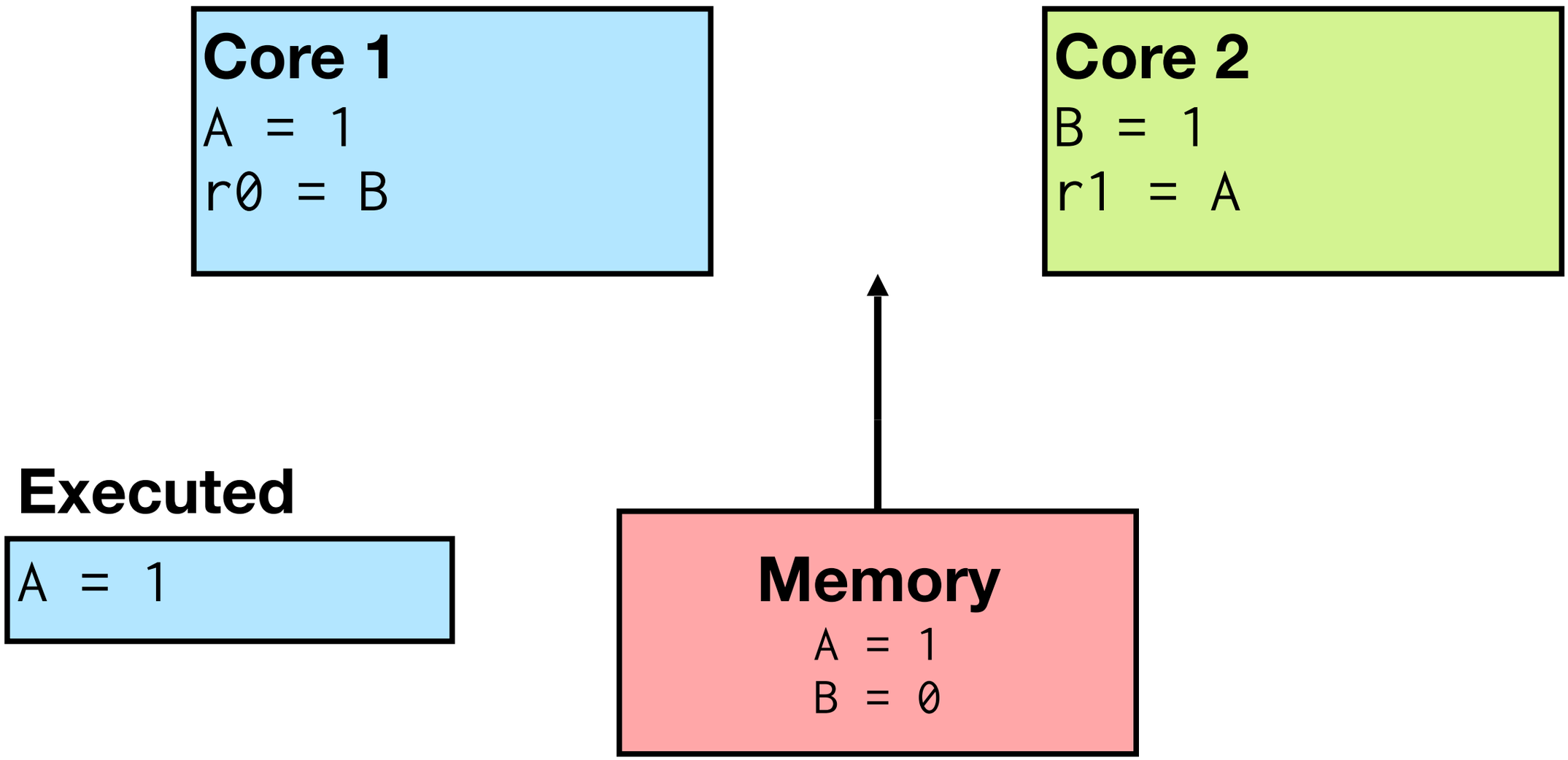
Can be seen as a “switch” running one instruction at a time



Executed

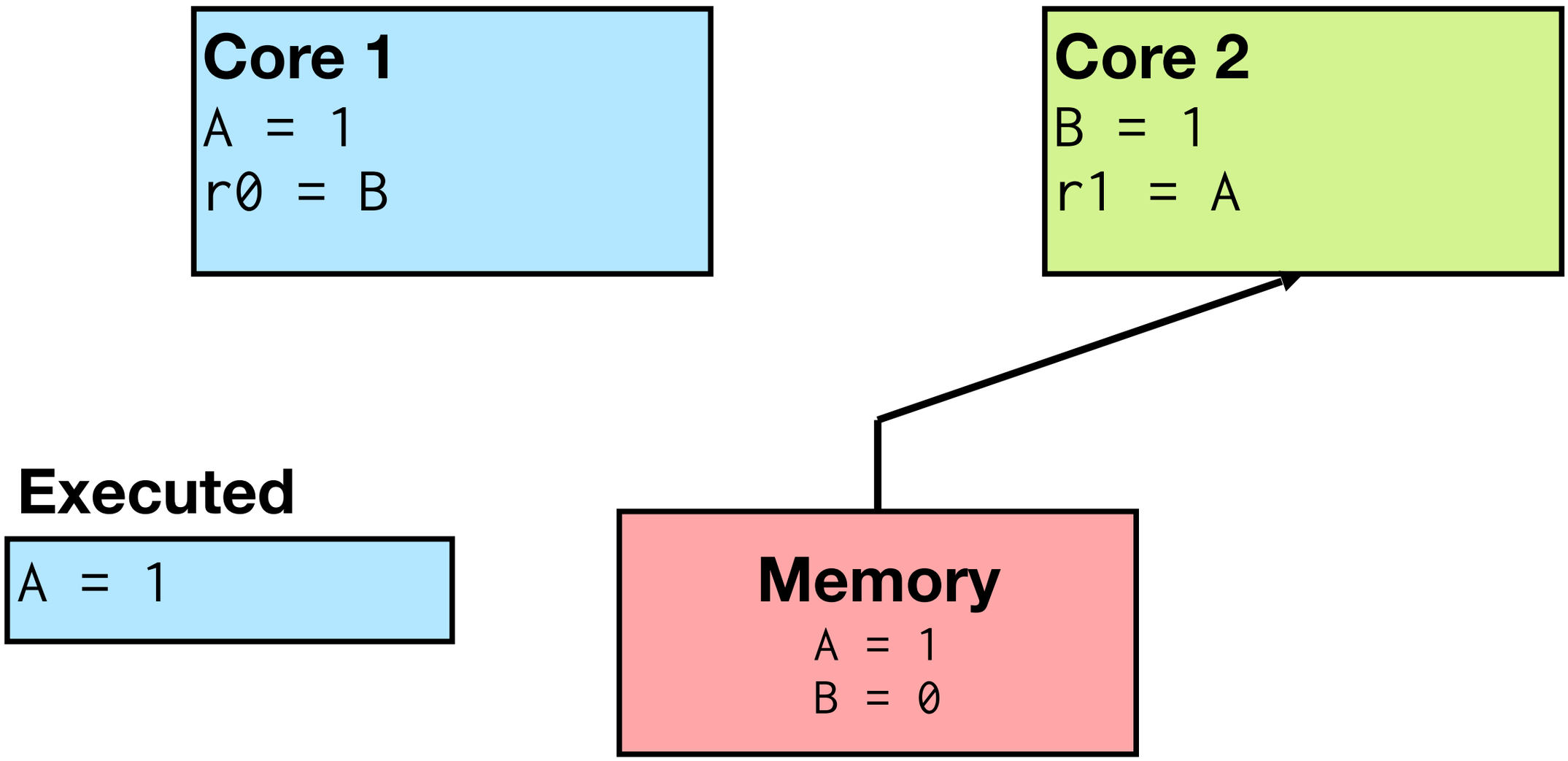
Sequential consistency

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



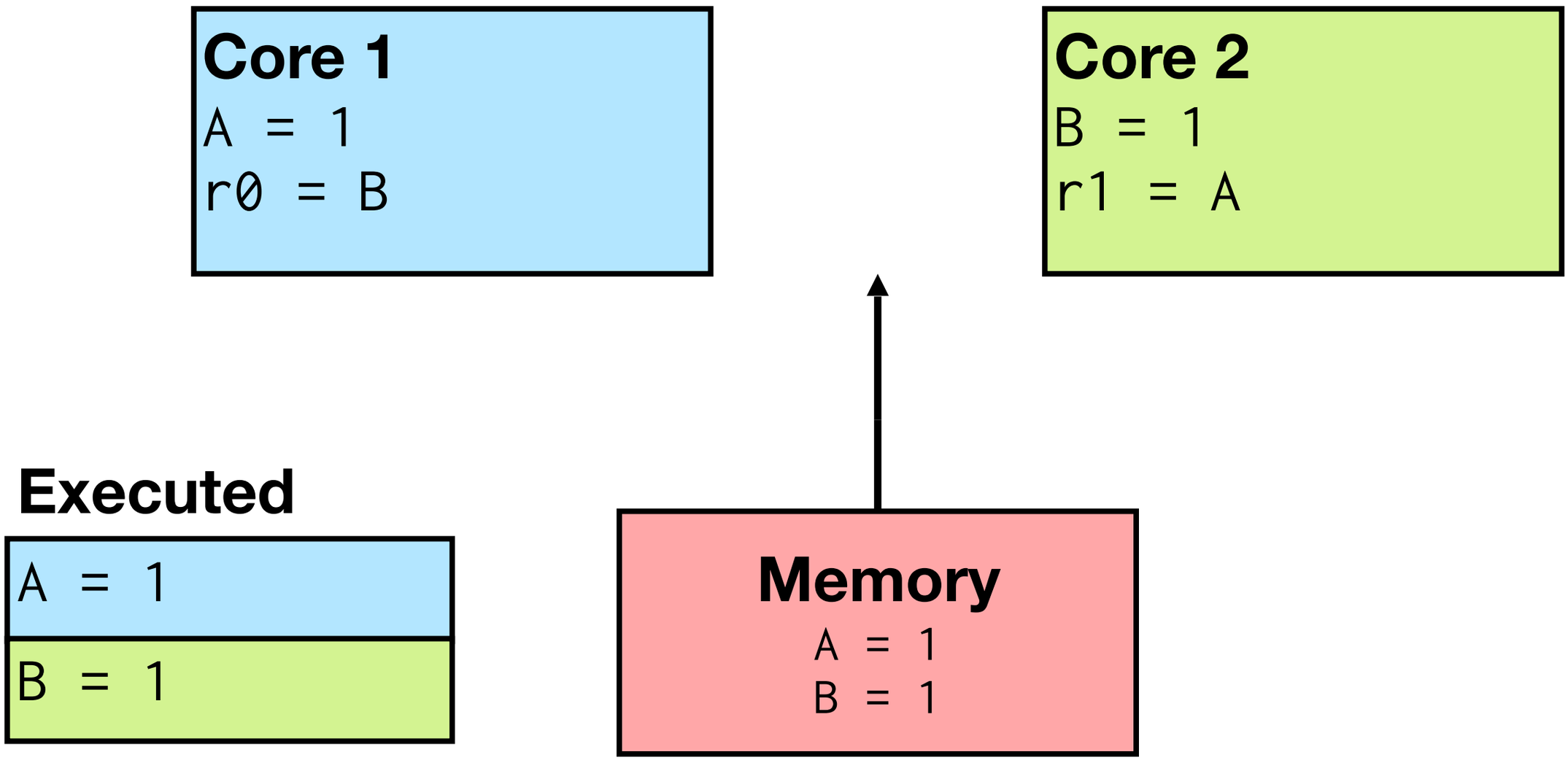
Sequential consistency

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

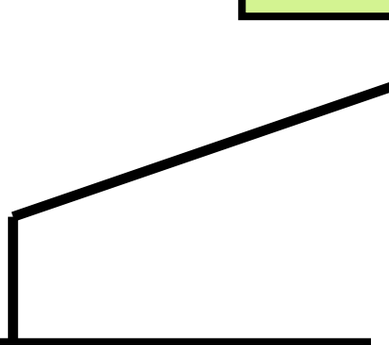
Can be seen as a “switch” running one instruction at a time

Core 1

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

Core 2

```
B = 1
r1 = A
```



Memory

A = 1

B = 1

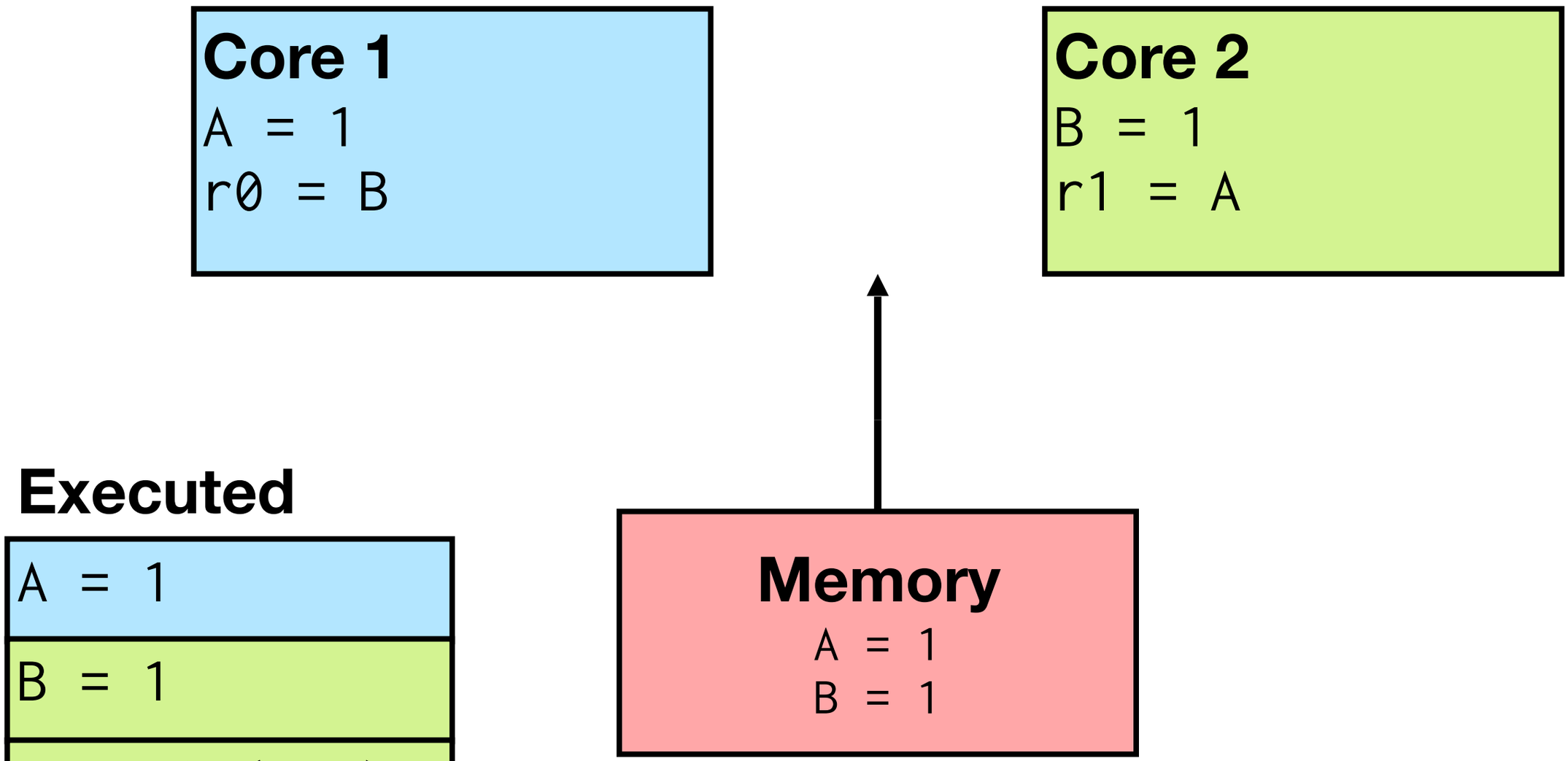
Executed

A = 1

B = 1

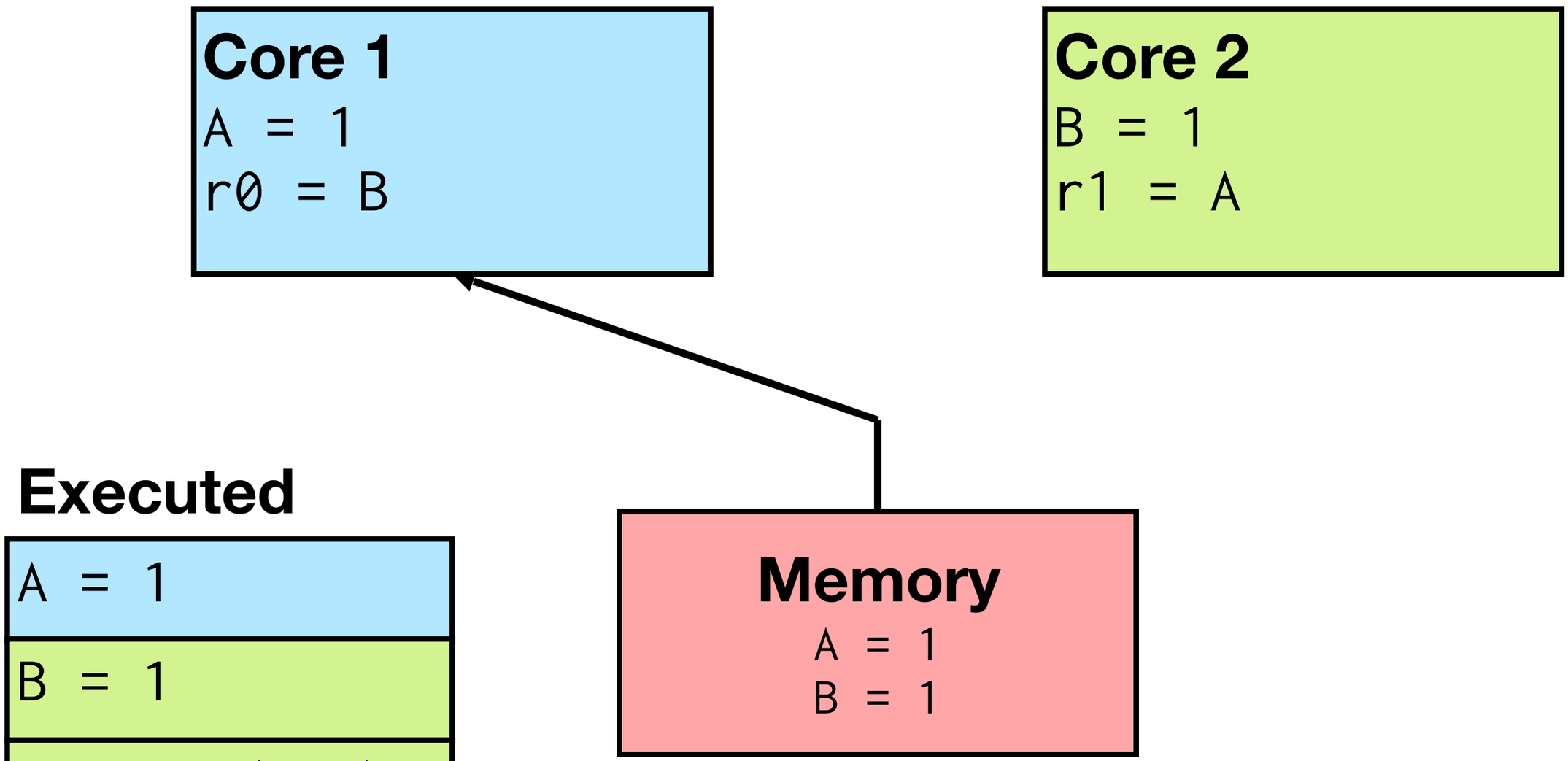
Sequential consistency

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



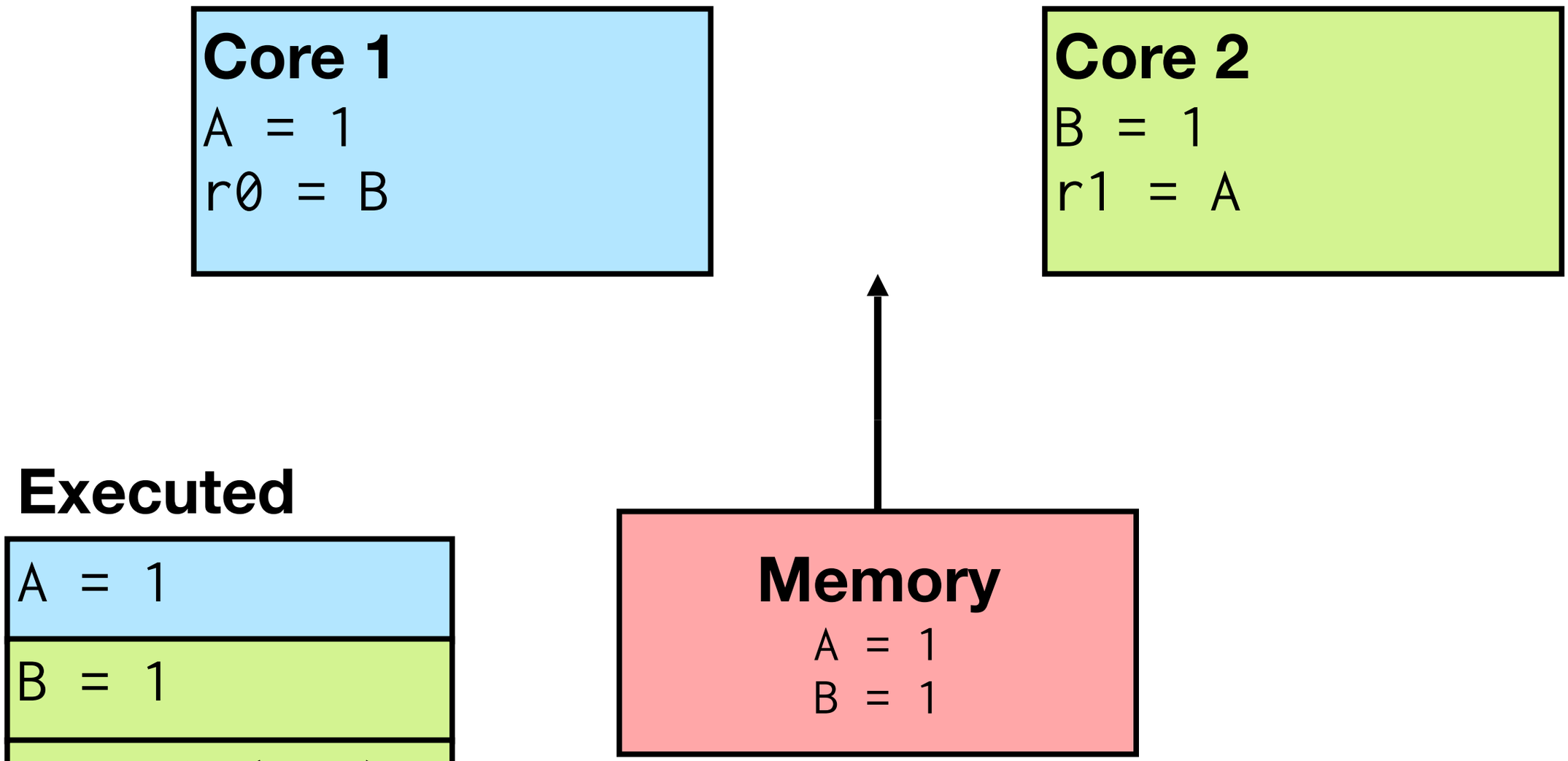
Sequential consistency

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

- Won the Turing award for this idea!

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/strange behaviors
  - (At least until you take 452!)

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`?

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!

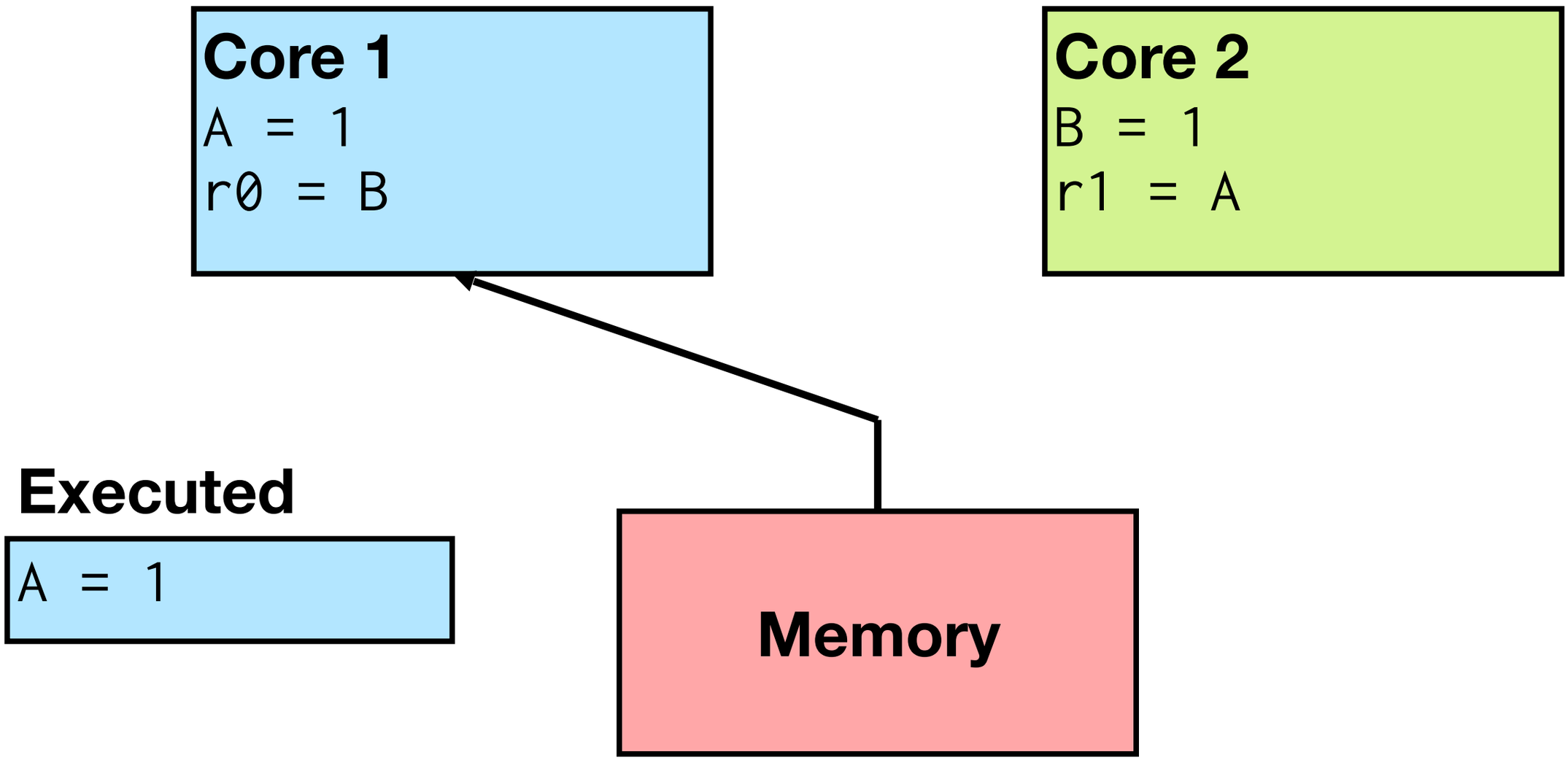
Why sequential consistency?

- Agrees with programmer intuition!

Why *not* sequential consistency?

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

The problem with SC



The problem with SC

These two instructions don't conflict—there's no need to wait for the first one to finish before executing the second.

Core 1

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

Core 2

```
B = 1
r1 = A
```

Executed

```
A = 1
```

Memory

The problem with SC

These two instructions don't conflict—there's no need to wait for the first one to finish before executing the second.

Core 1

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

Core 2

```
B = 1
r1 = A
```

Executed

```
A = 1
```

Memory

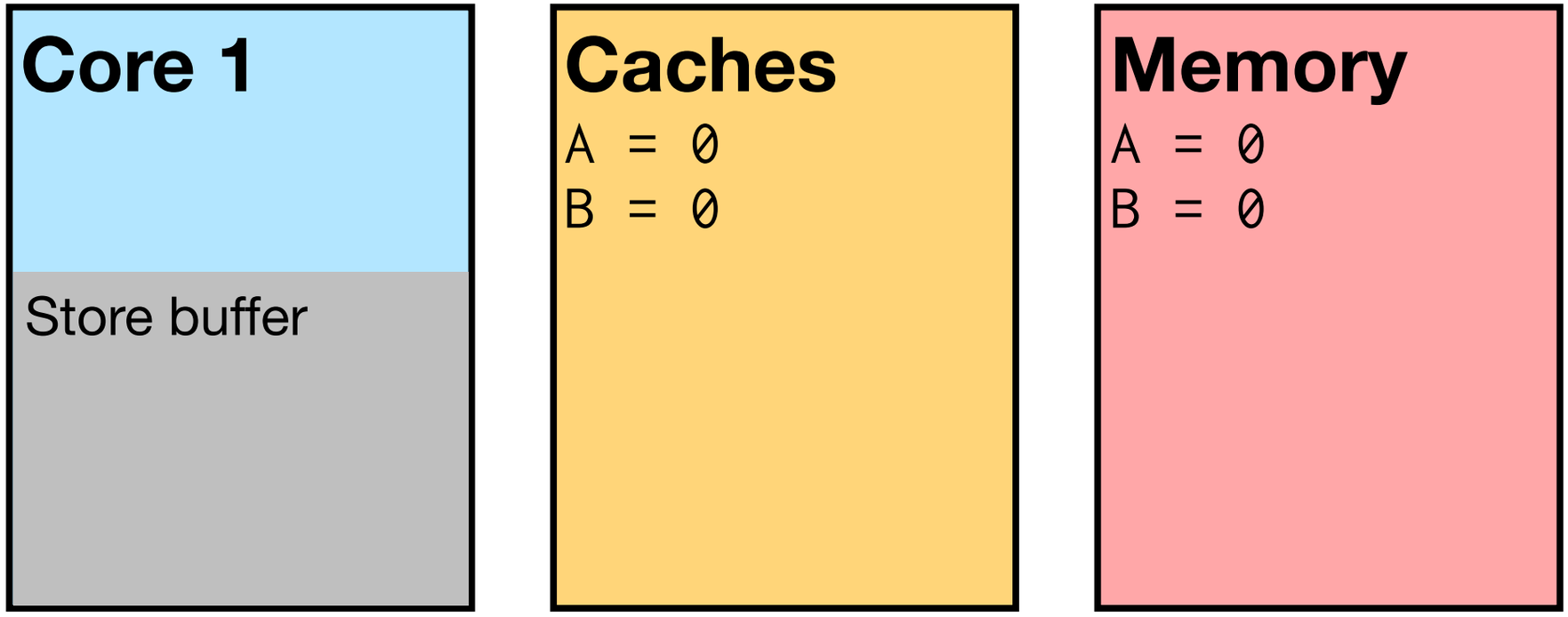
And writing to memory takes *forever*!
(about 100 cycles ≈ 30 ns)

Optimization: Store buffers

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

Thread 1

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

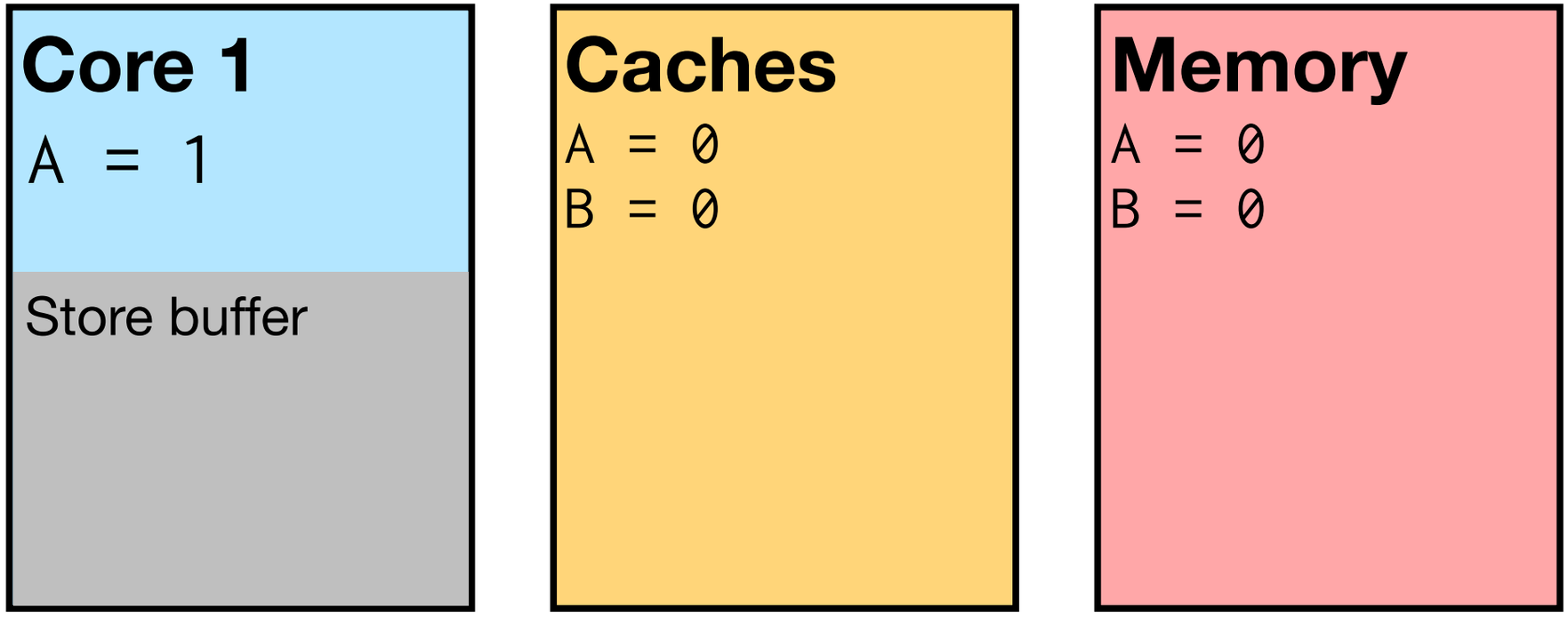


Optimization: Store buffers

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

Thread 1

```
r0 = B
```

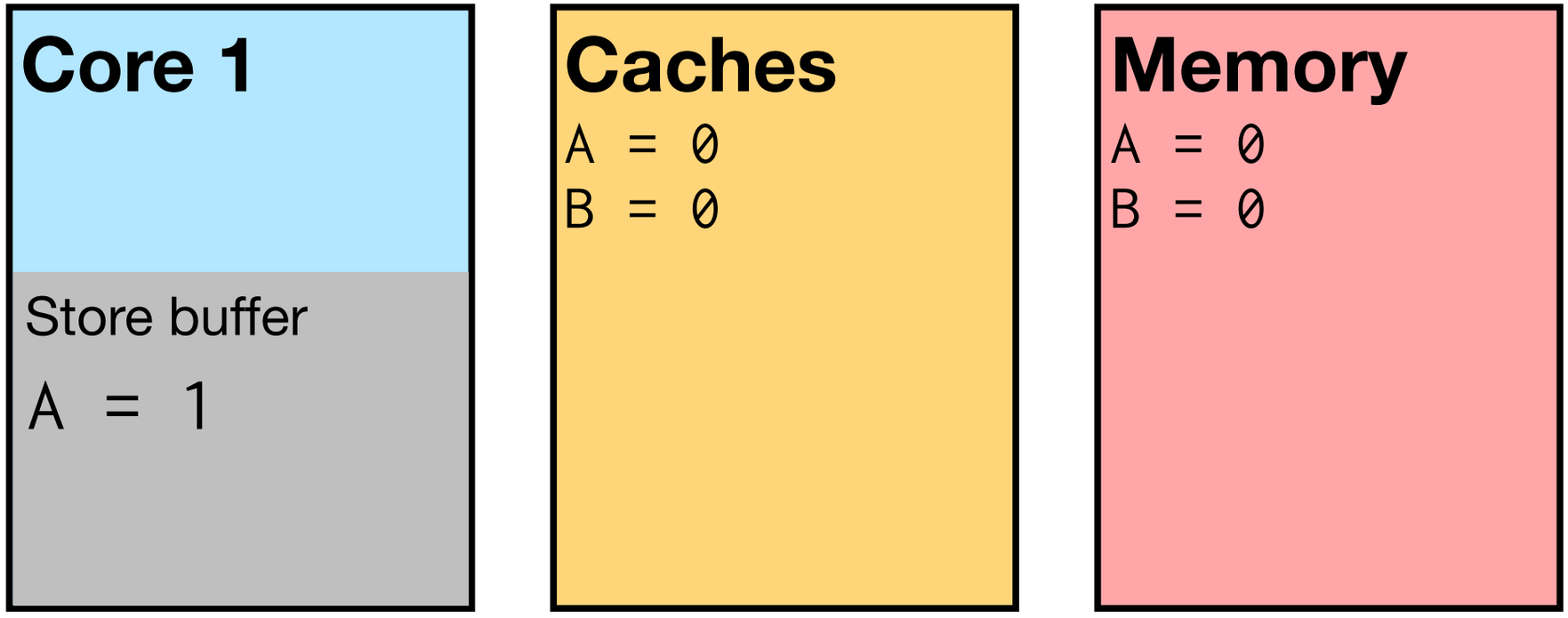


Optimization: Store buffers

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

Thread 1

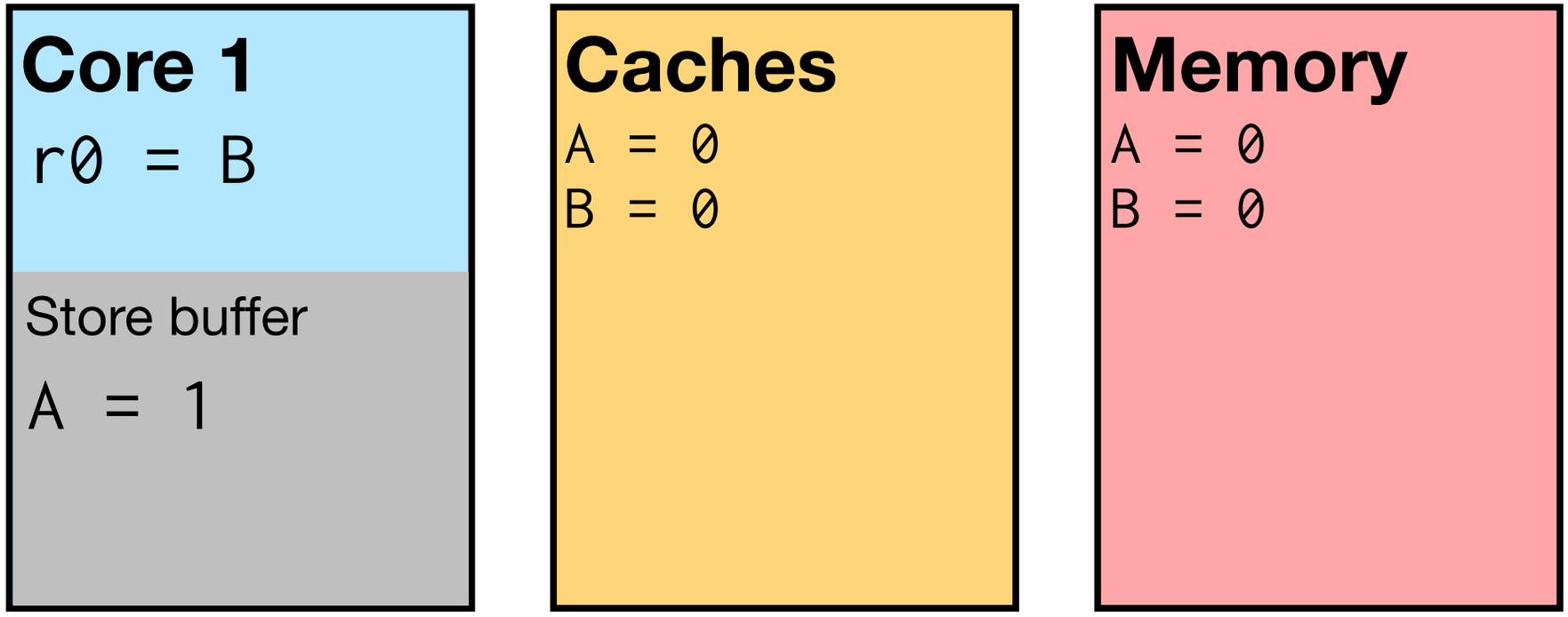
```
r0 = B
```



Optimization: Store buffers

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

Thread 1

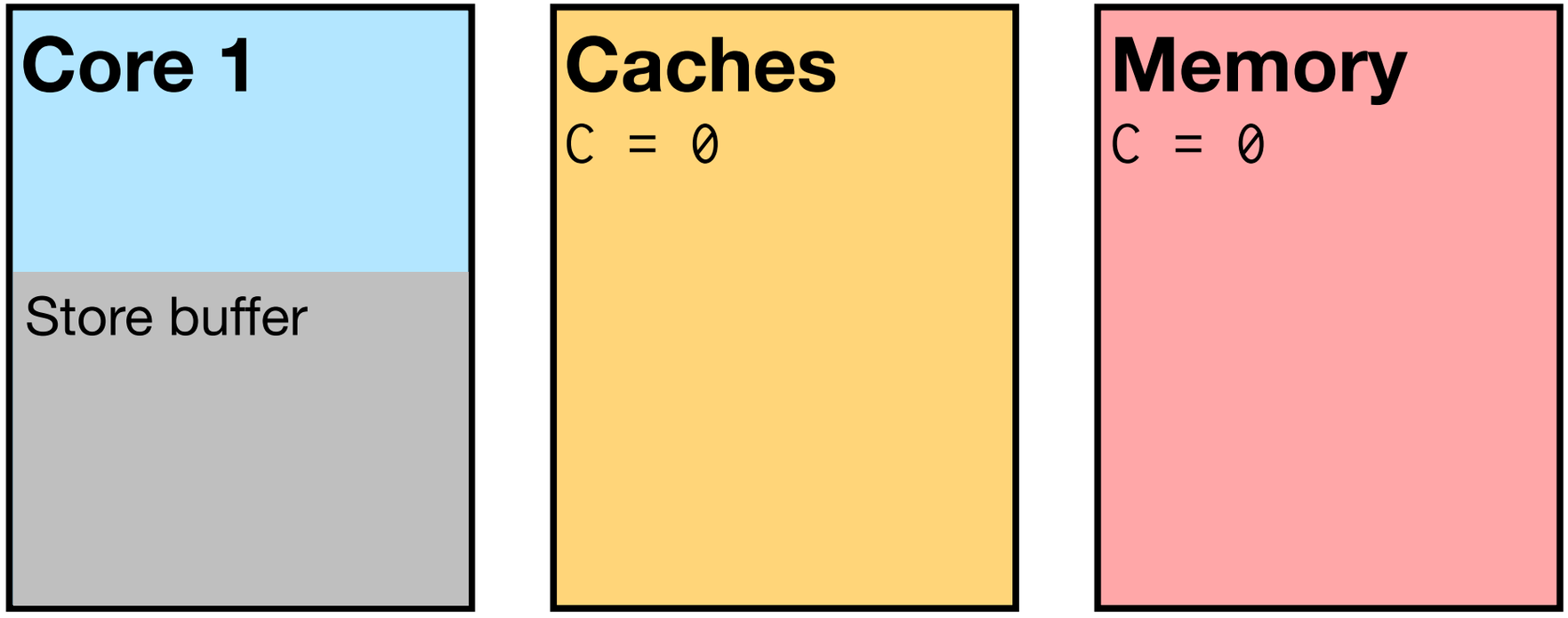


Optimization: Store buffers

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

Thread 1

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

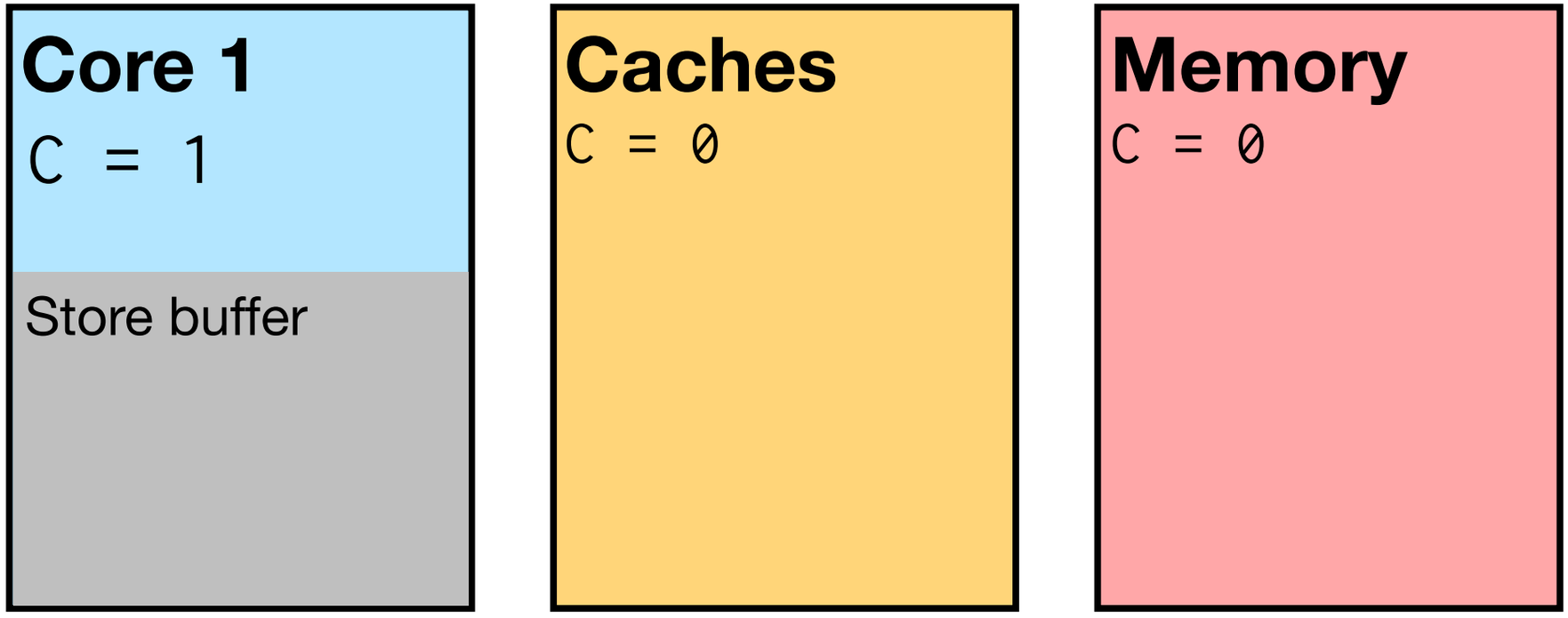


Optimization: Store buffers

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

Thread 1

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

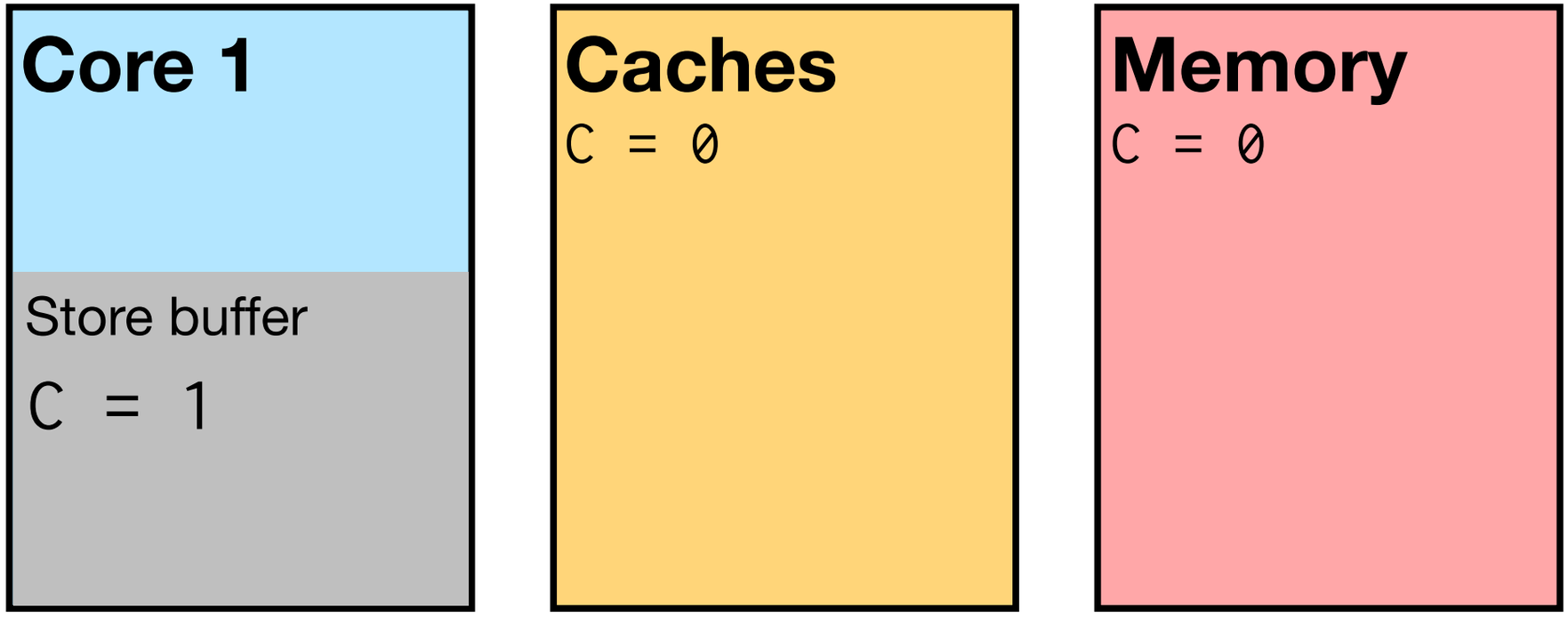


Optimization: Store buffers

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

Thread 1

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

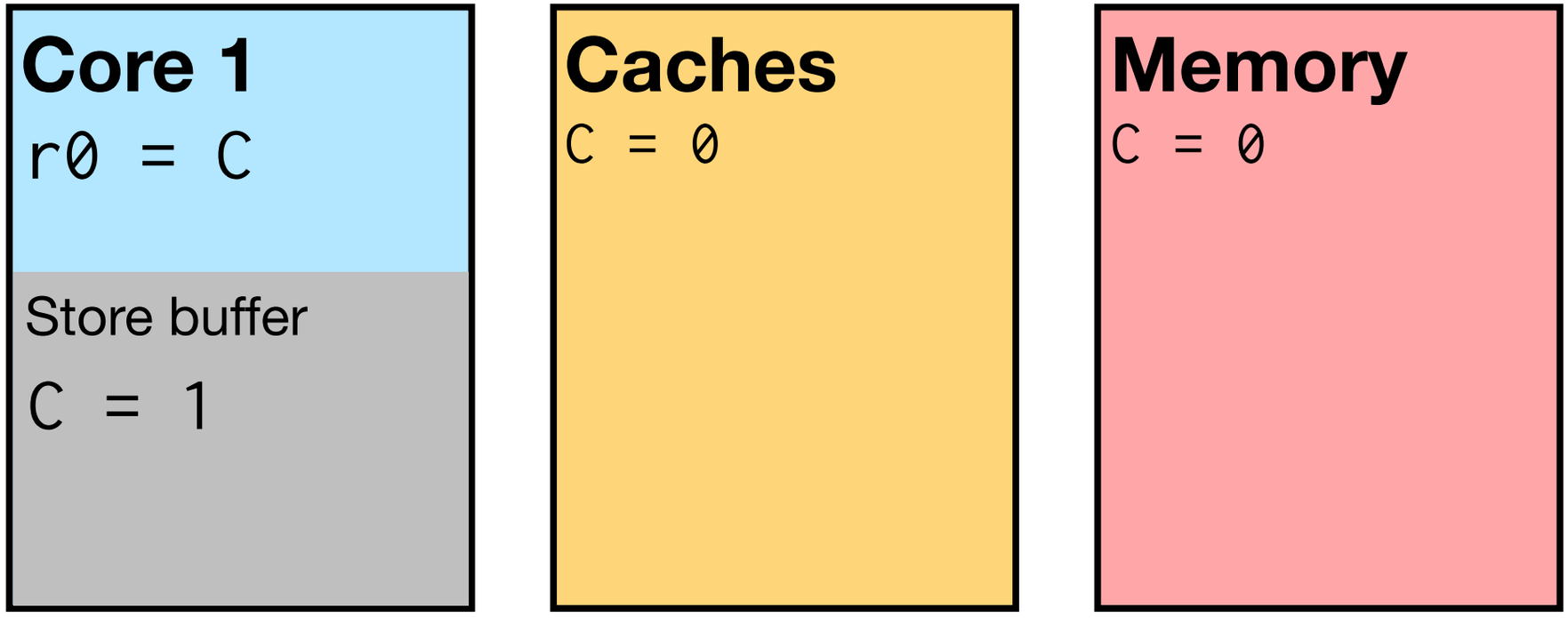


Optimization: Store buffers

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

Thread 1

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

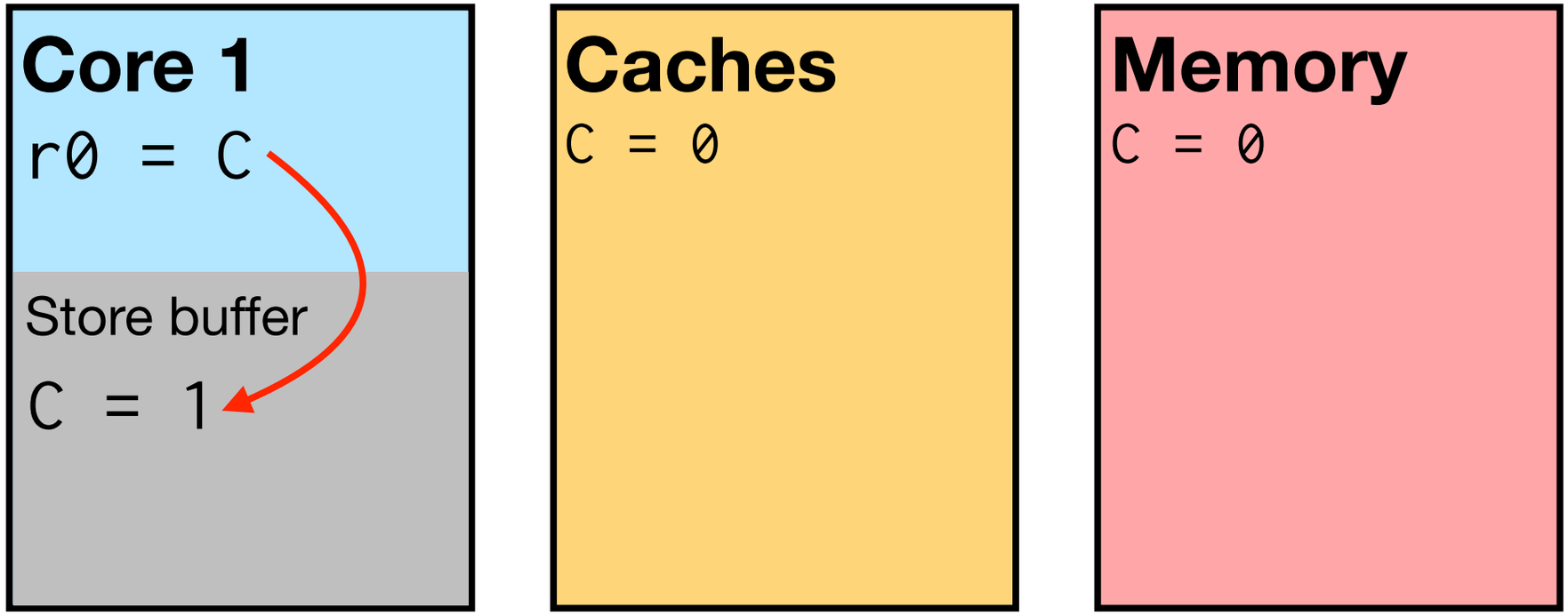


Optimization: Store buffers

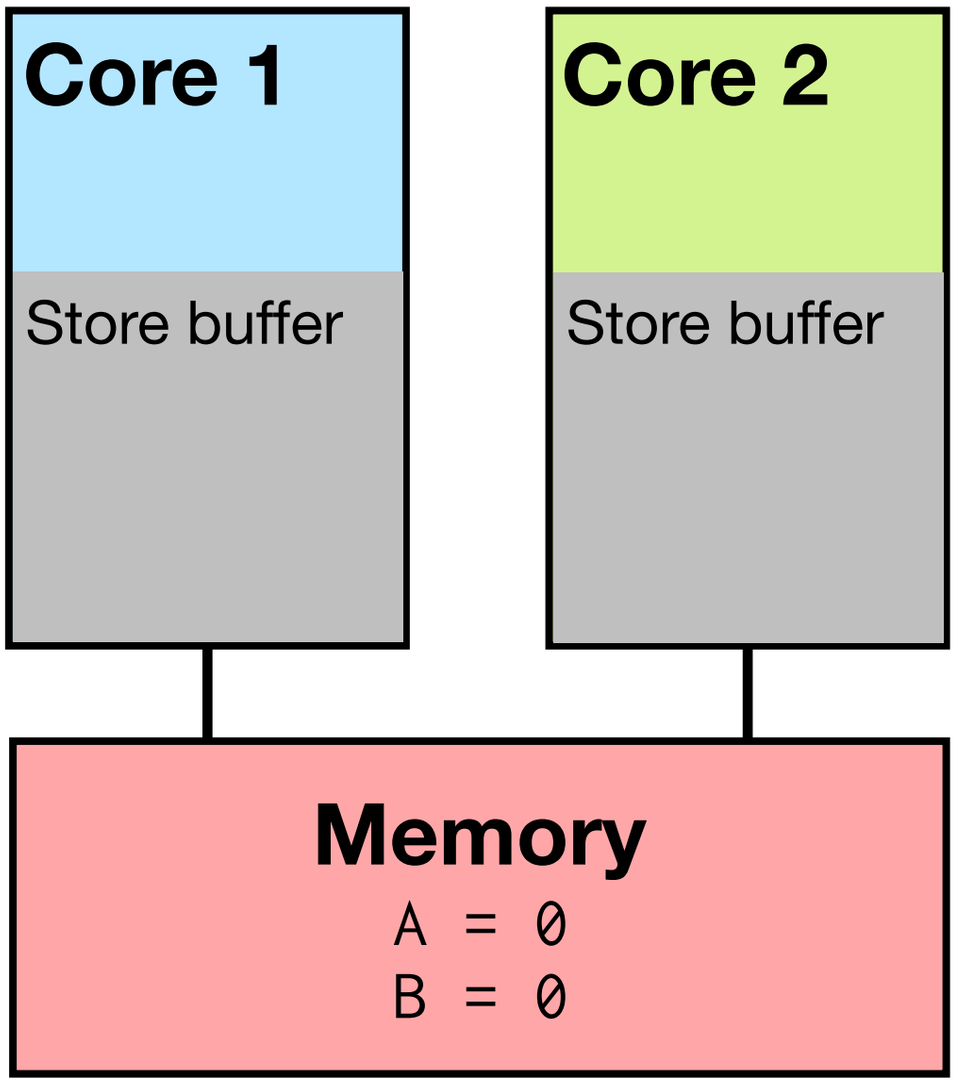
- Store writes in a local buffer and then proceed to next instruction immediately
- The cache will pull writes out of the store buffer when it's ready

Thread 1

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



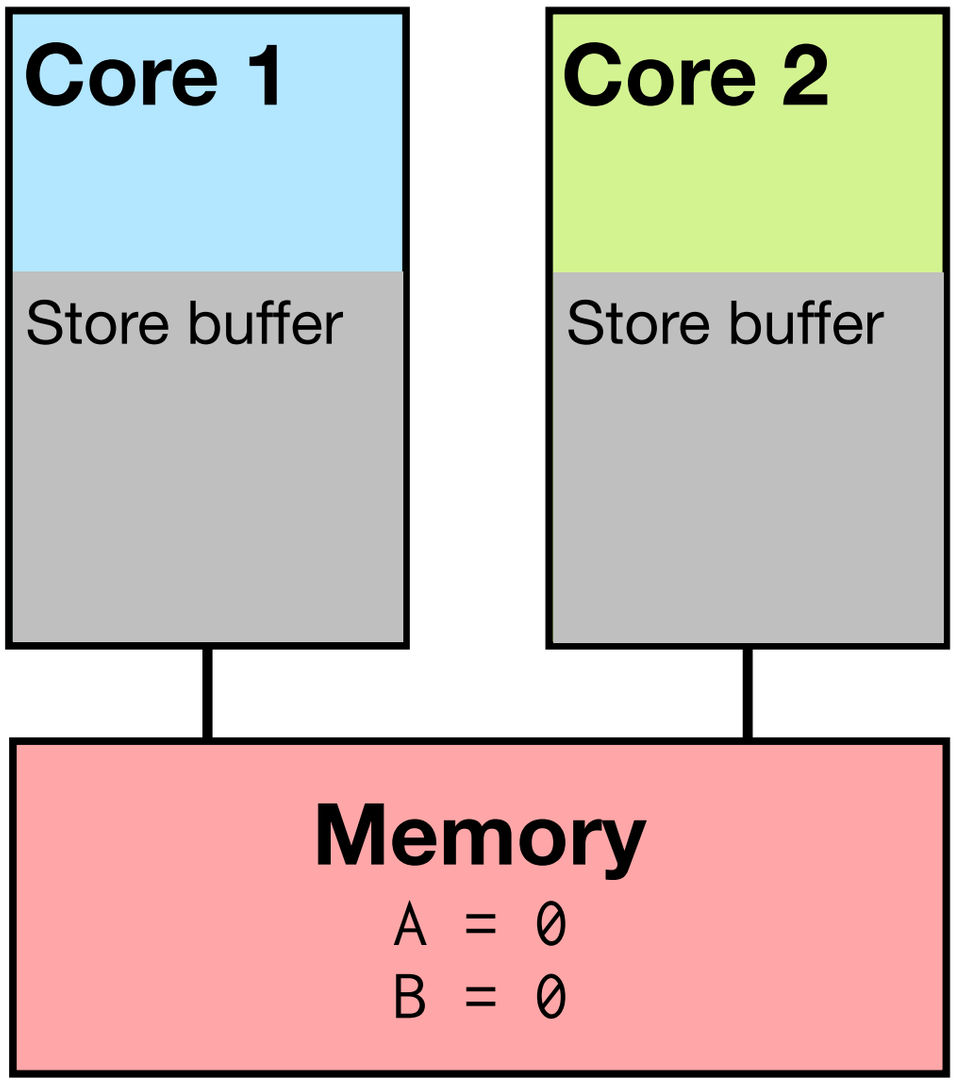
Store buffers change memory behavior

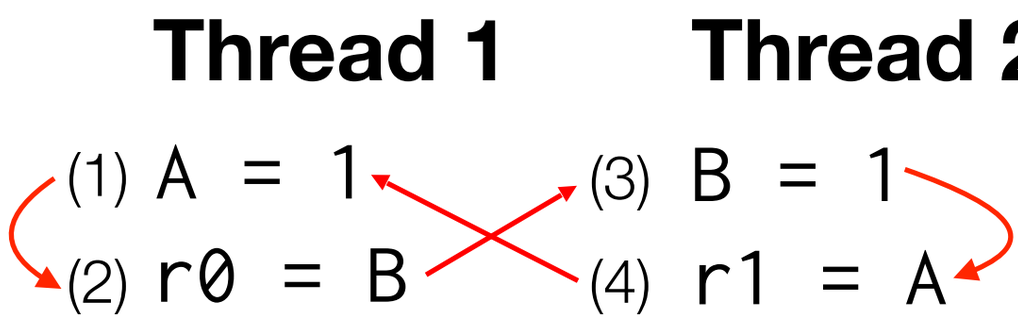


Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

Store buffers change memory behavior

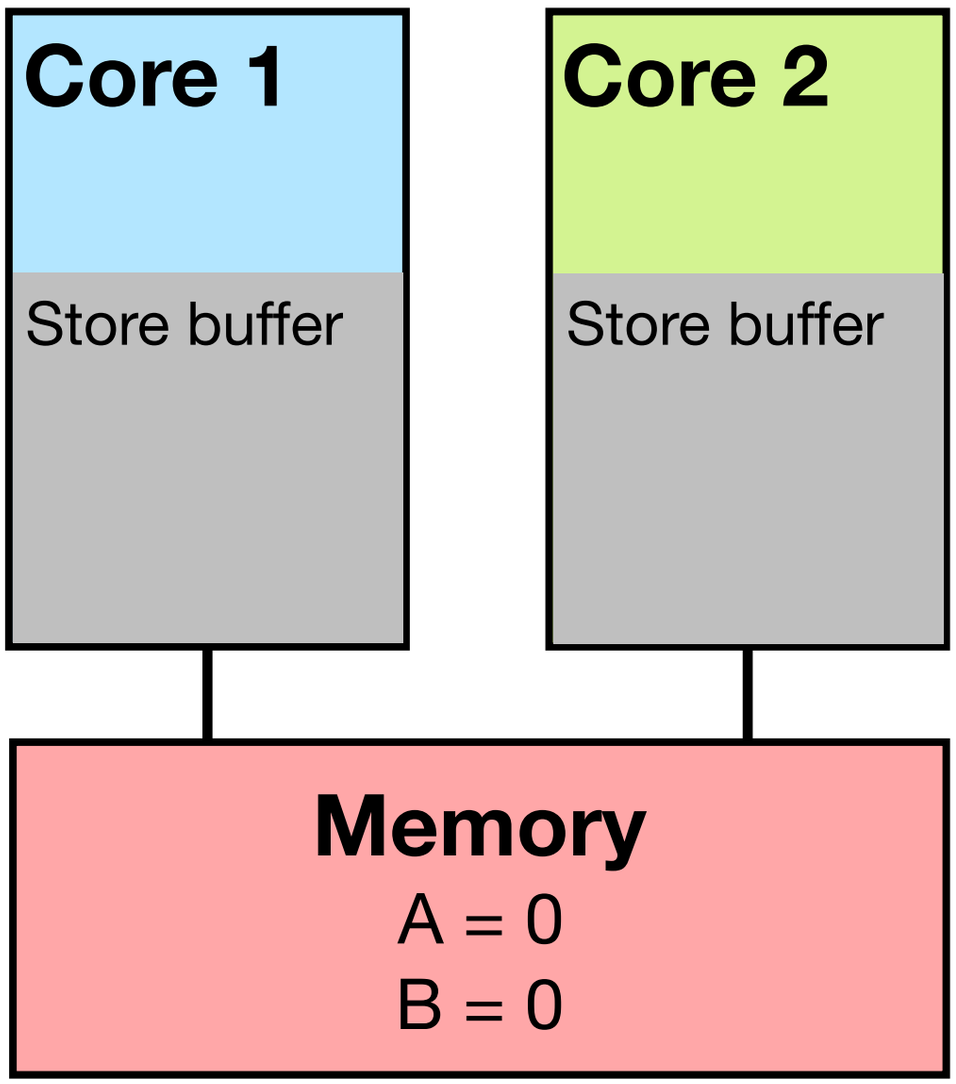




Can `r0 = 0` and `r1 = 0`?

SC: No!

Store buffers change memory behavior



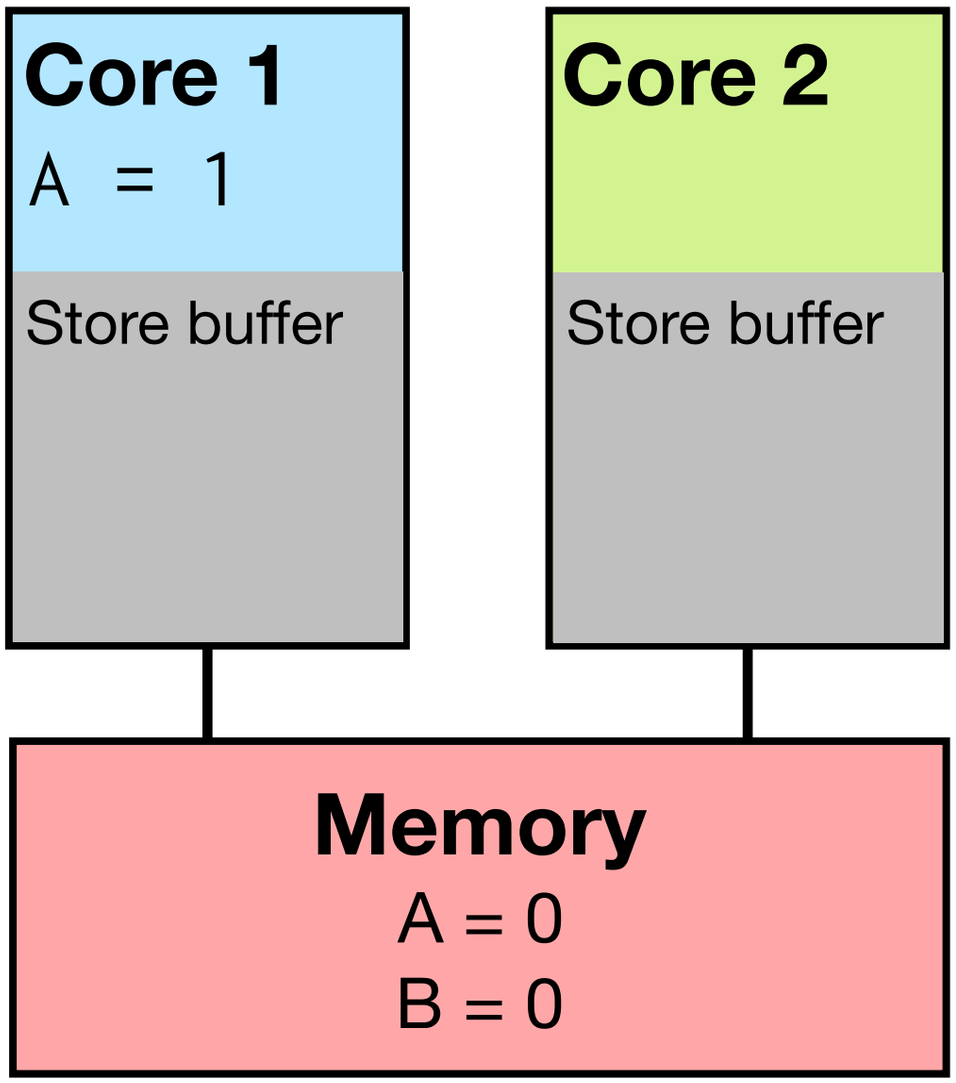
Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



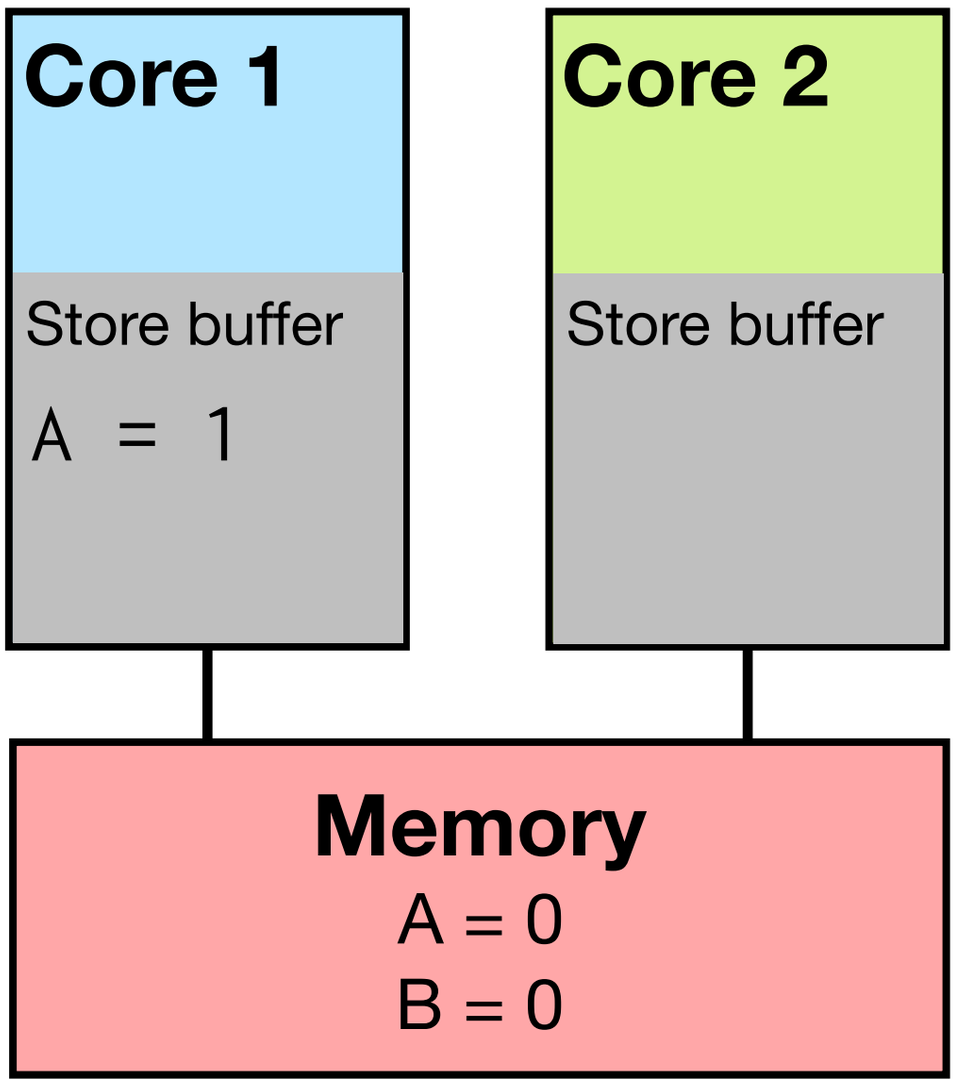
Thread 1	Thread 2
(1)	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



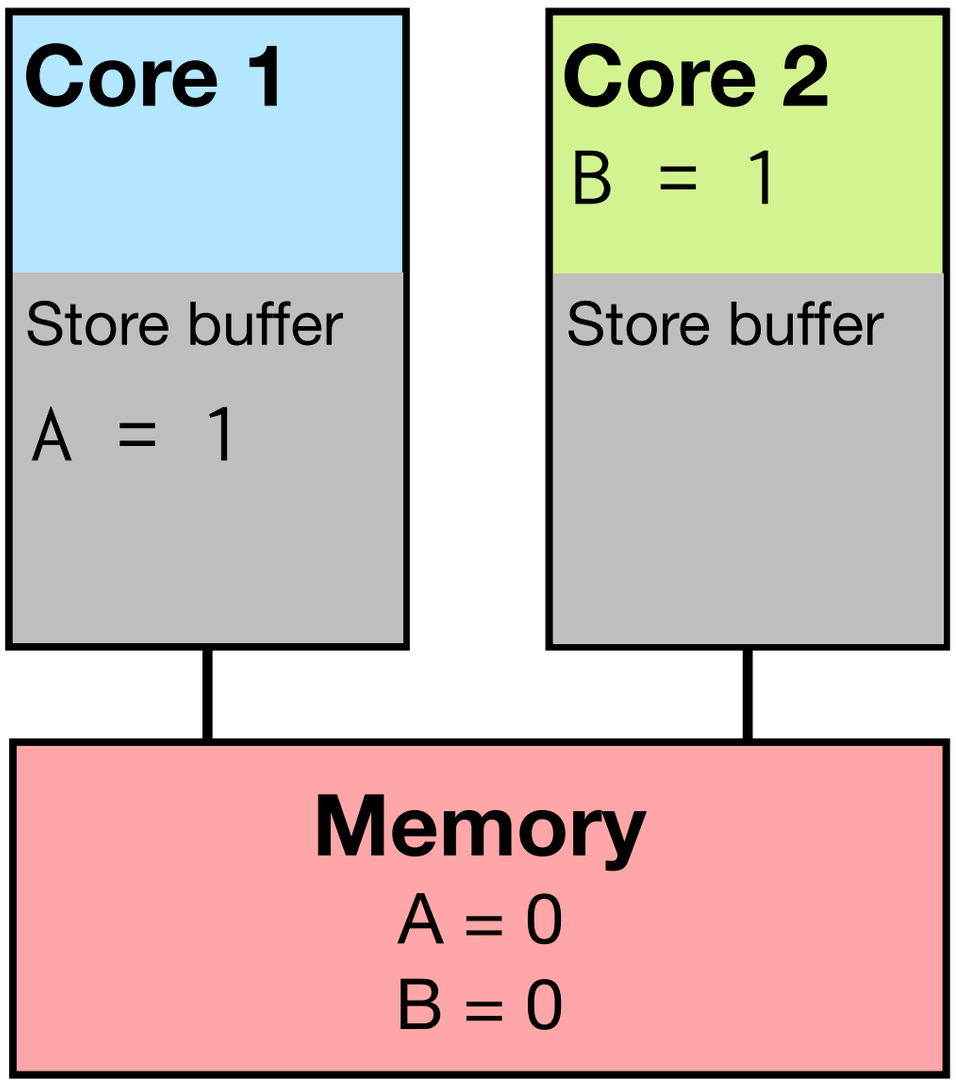
Thread 1	Thread 2
(1)	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



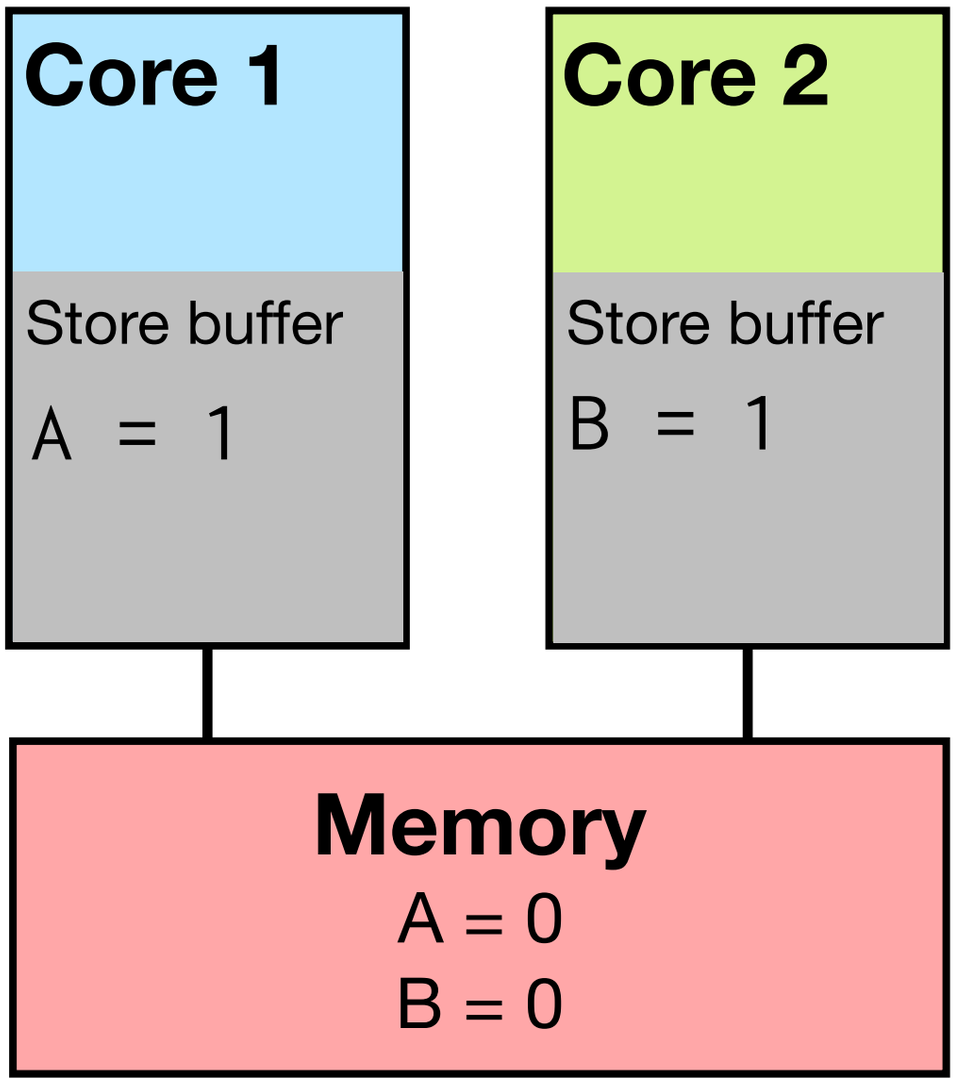
Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



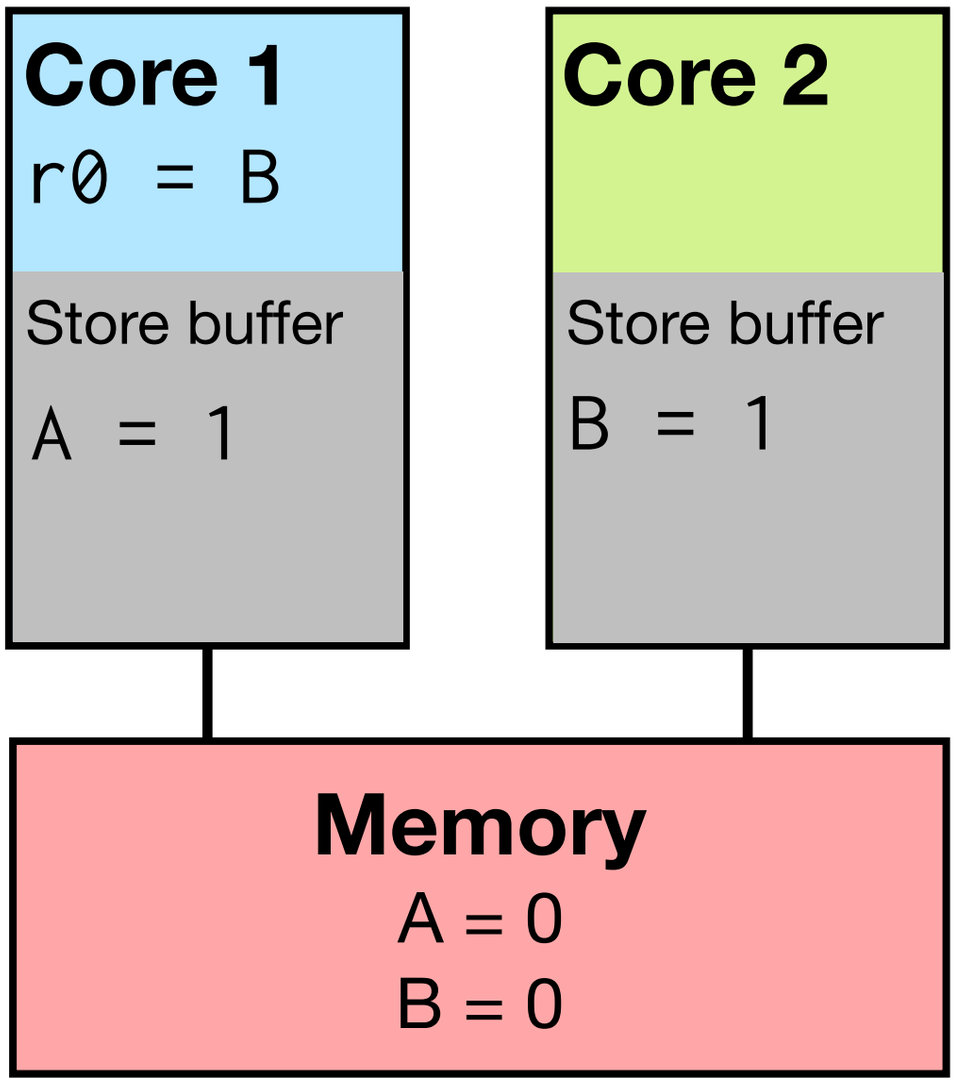
Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



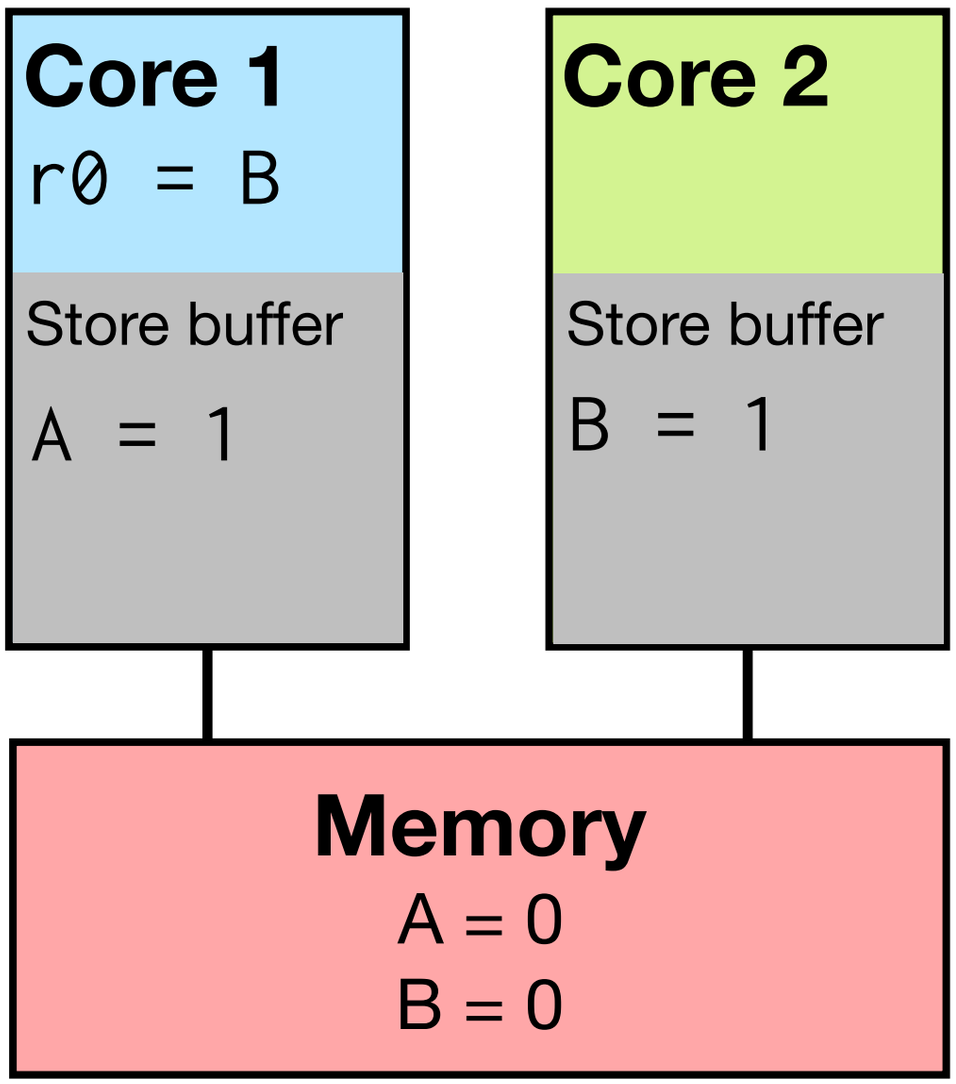
Thread 1	Thread 2
(1)	(3)
(2)	(4) `r1 = A`

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

SC: No!

Executed

Store buffers change memory behavior



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

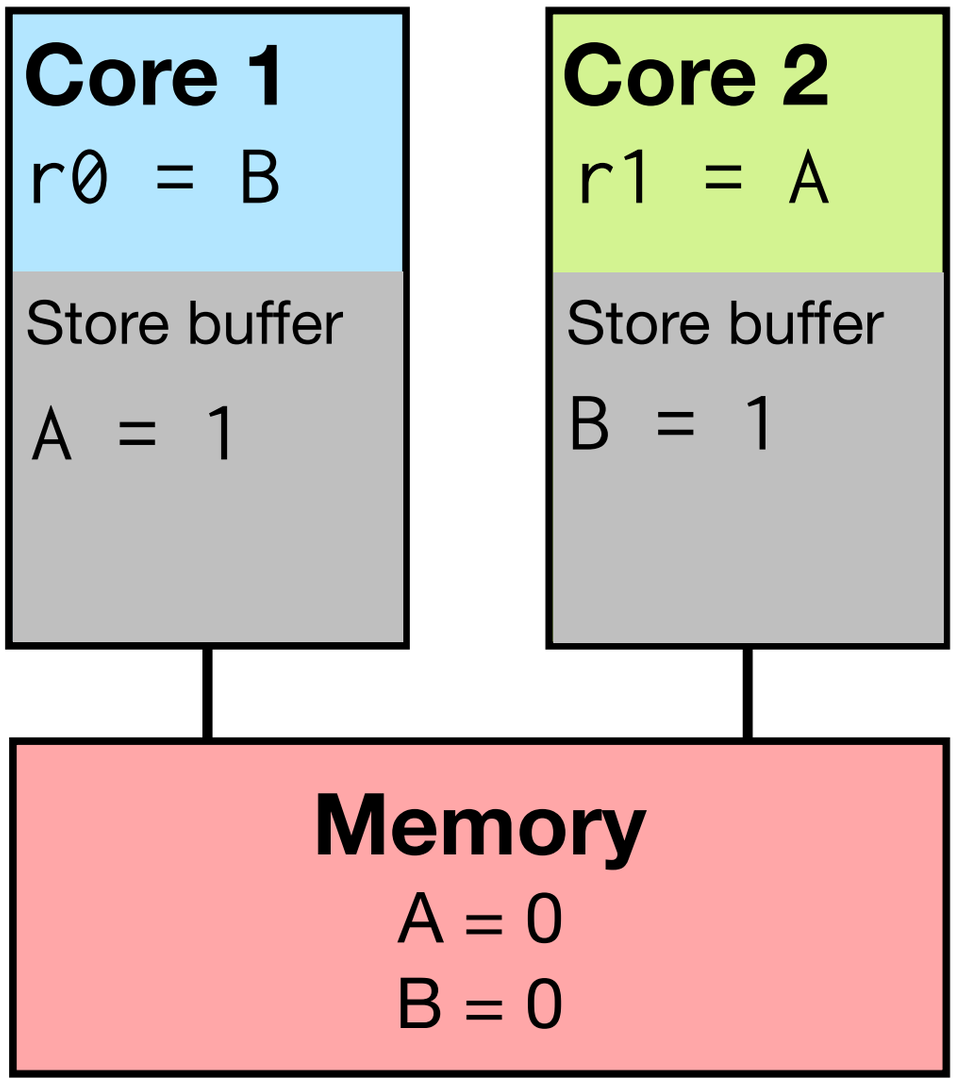
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Store buffers change memory behavior



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

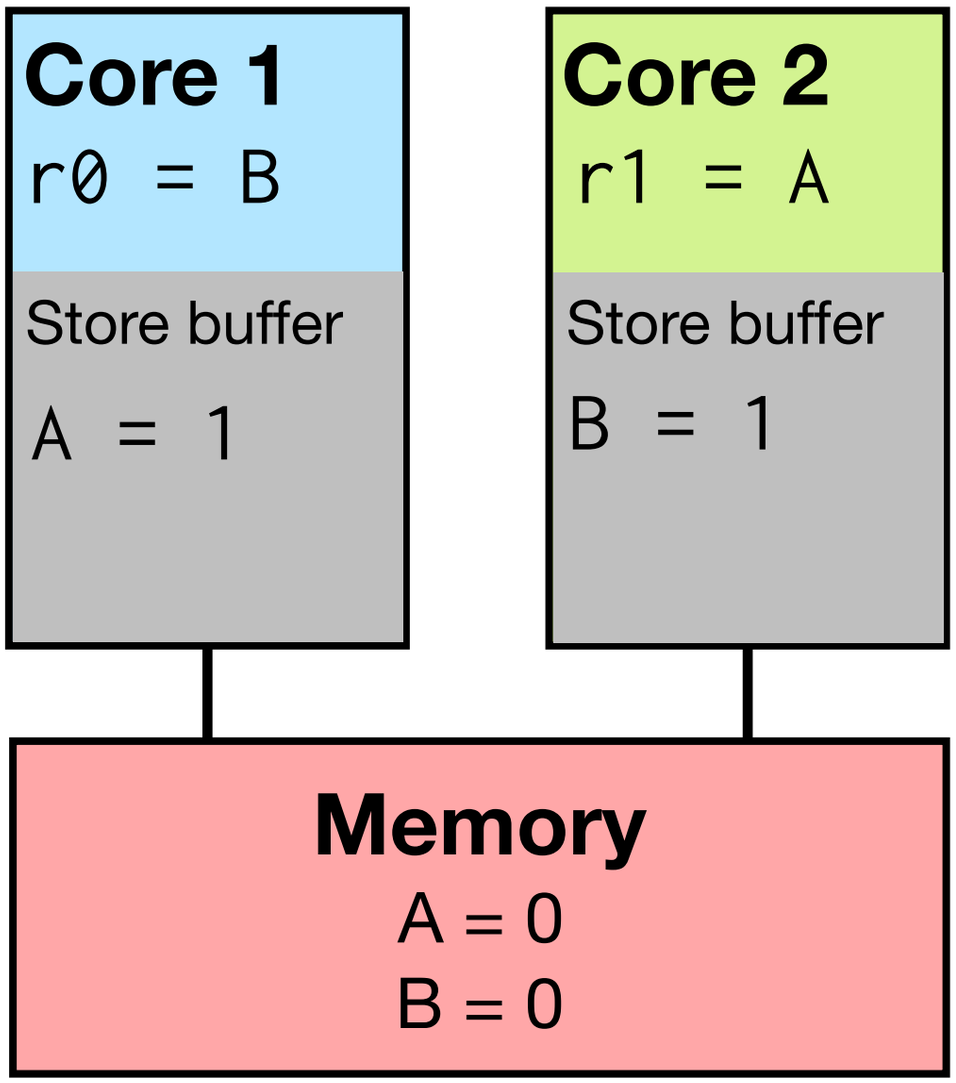
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Store buffers change memory behavior



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

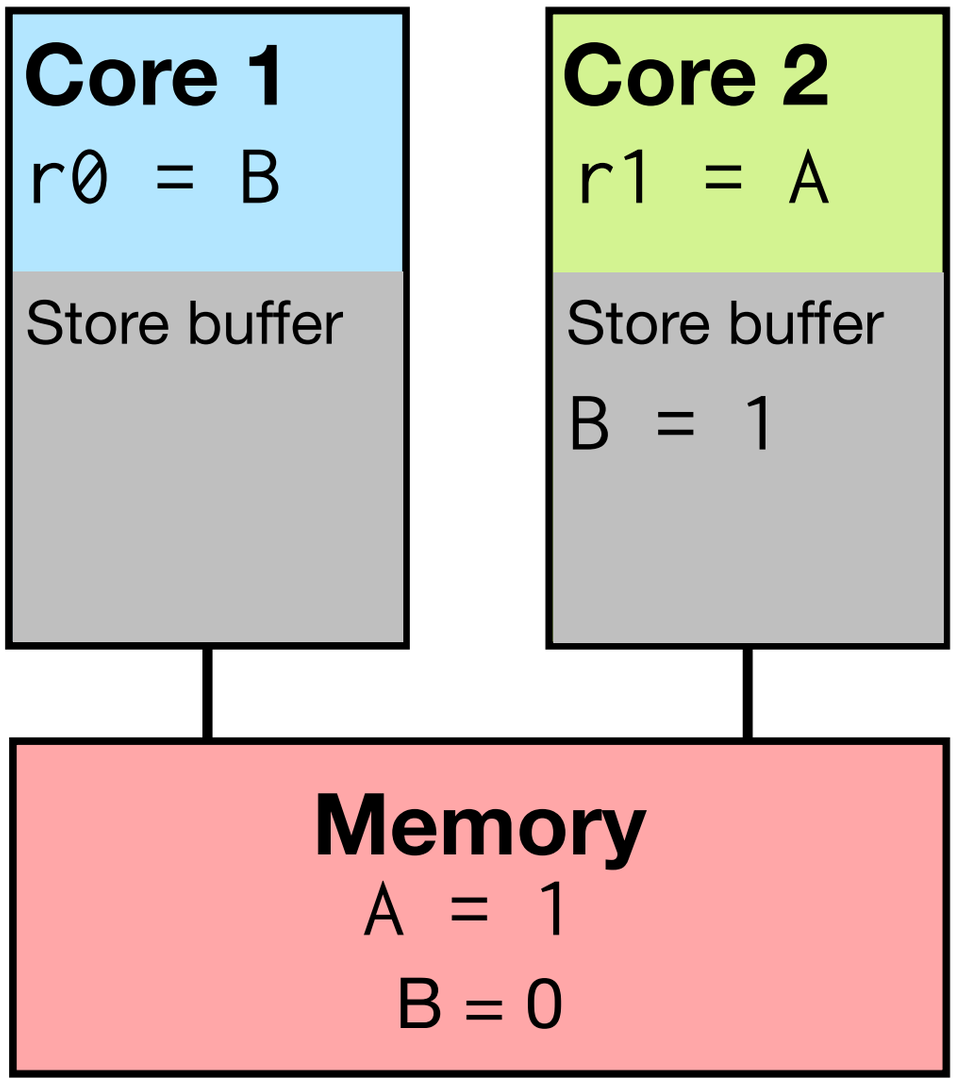
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Store buffers change memory behavior



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

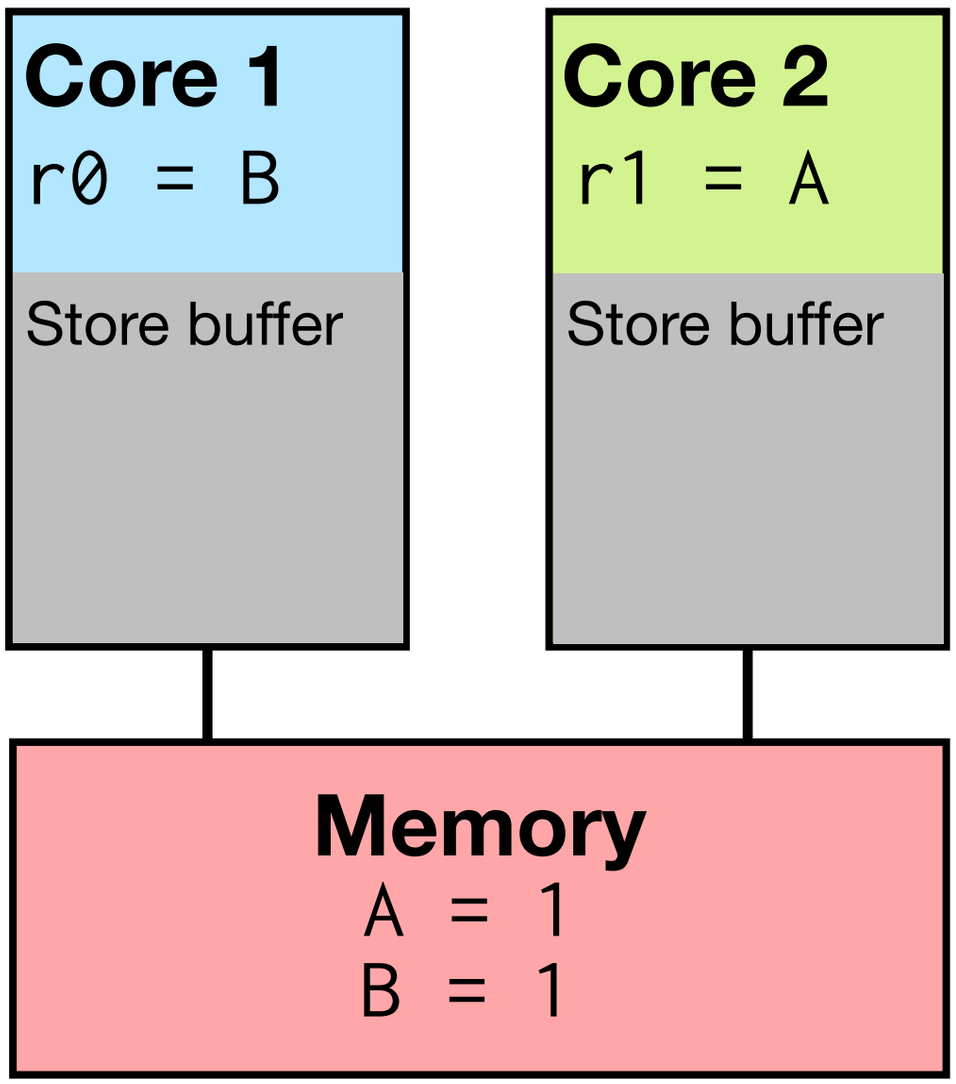
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Store buffers change memory behavior



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

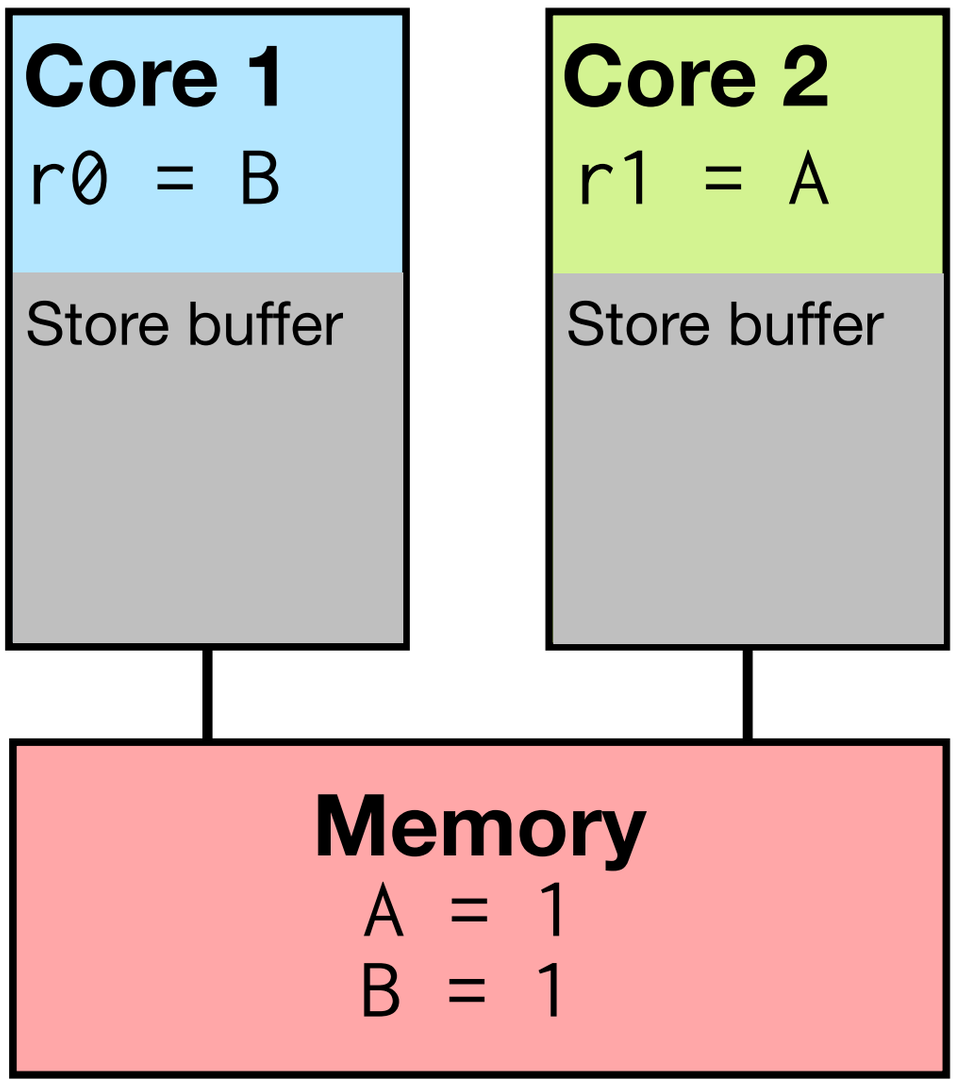
Can `r0 = 0` and `r1 = 0`?

SC: No!

Executed

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

Store buffers change memory behavior



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

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

SC: No! Store buffers: Yes!

Executed

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

So, who uses store buffers?

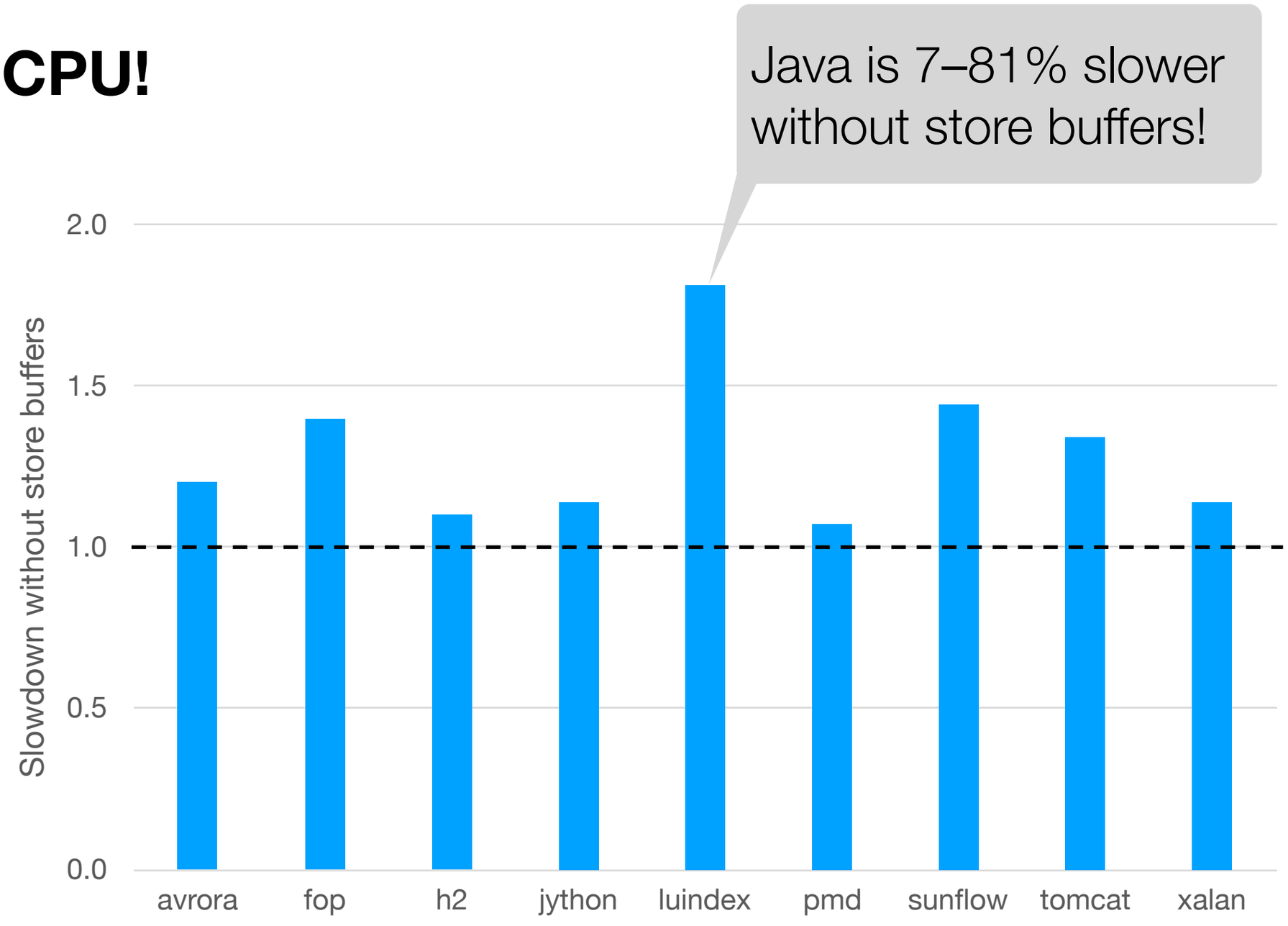
Every modern CPU!

- x86
- ARM
- PowerPC
- ...

So, who uses store buffers?

Every modern CPU!

- x86
- ARM
- PowerPC
- ...

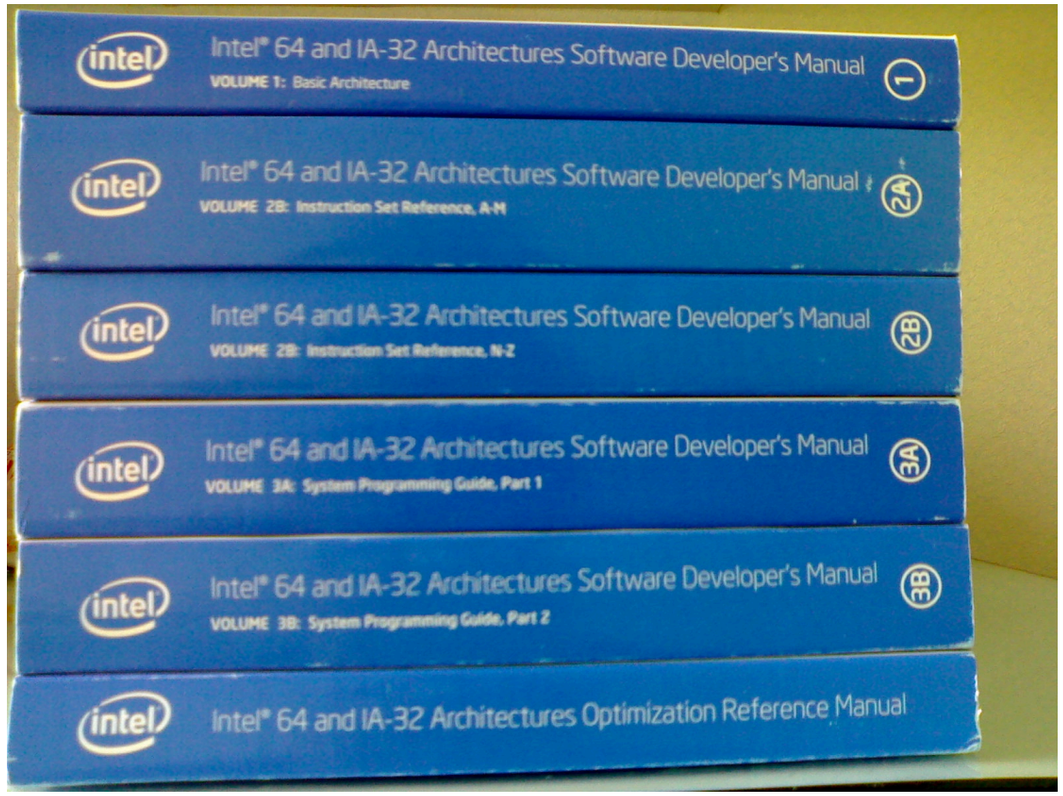


A Volatile-by-Default JVM for Server Applications. Liu, Millstein, Musuvathi. OOPSLA 2017.

Total Store Ordering (TSO)

- Sequential consistency plus store buffers
- Allows more behaviors than SC
  - Harder to program!

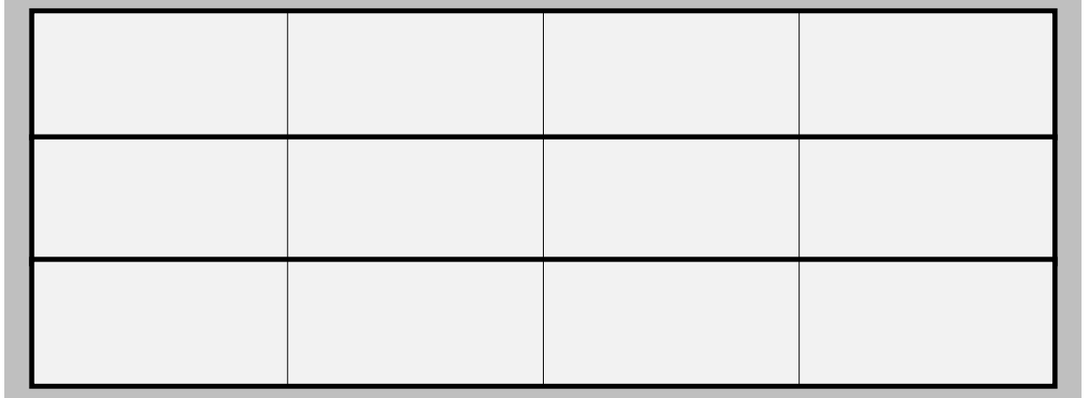
- x86 specifies TSO as its memory model



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

Write buffer



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

Write buffer

	Y = 1		

Thread 1

X = 1
Y = 1
Z = 1

Assume X and Z are on the same cache line

Executed

X = 1
Z = 1

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

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
  - **Almost everything** can be reordered! 😱
  - One exception: dependent operations are ordered

```
ldr r0, #y                  int** r0 = y;  // y stored in r0
ldr r1, [r0]      ⟺        int* r1 = *y;
ldr r2, [r1]                int* r2 = *r1;
```

Even more esoteric memory models

- DEC Alpha
  - A successor to VAX…
  - Killed in 2001

- *Dependent operations* can be reordered!

- Lowest common denominator for the Linux kernel

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/atomics, transactional memory, …

```
movl $1,%[x]          movl $1,%[y]
movl %[y],%eax        movl %[x],%ebx
```

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/atomics, transactional memory, …

```
movl $1, %ecx
xchg %ecx, %[x]
movl %[y],%eax
```

```
movl $1, %ecx
xchg %ecx, %[y]
movl %[x],%ebx
```

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/atomics, transactional memory, …

```
movl $1,%[x]          movl $1,%[y]
movl %[y],%eax        movl %[x],%ebx
```

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/atomics, transactional memory, …

```
movl $1,%[x]
mfence
movl %[y],%eax
```

```
movl $1,%[y]
mfence
movl %[x],%eax
```

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
```

compiler

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
```

11111111111…

11111011111…

Thread 2

```
X = 0
```

Thread 1

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

11111111111…

11111000000…

Thread 2

```
X = 0
```

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 (**undefined behavior**) if your program contains even a single data race!
  - The intuition is that most programmers would consider a racy program to be buggy
  - Use a synchronization library!

- *Incredibly* difficult to get right in the compiler and kernel
  - Countless bugs and mailing list arguments

Memory models in the Linux kernel

Manfred Spraul
spin_unlock optimization(i386)

the current spin_unlock asm code is
`lock; btrl $0,%0`
it takes ~ 22 ticks on my PII/350.
I think it's possible to replace that with
`movl $0,%0`
which would be a simple, pairable single-tick instruction.

Memory models in the Linux kernel

Manfred Spraul
spin_unlock optimization(i386)

the curr
`lock; b`
it takes
I think it
`movl $0`
which w
single-t

Linus Torvalds
Re: spin_unlock optimization(i386)

It does NOT WORK! Let the FreeBSD people use it, and then get faster timings. They will crash, eventually.
[…]
the above CAN return 1
[…]
I might be proven wrong, but I don’t think I am.

Memory models in the Linux kernel

Manfred Spraul
spin_unlock optimization(i386)

the curr
`lock; b`
it takes
I think it
`movl $0`
which w
single-t

Linus Torvalds
Re: spin_unlock optimization(i386)

It does NOT
it, and then
eventually.
[...]
the above C
[...]
I might be p

Erich Boleyn
Re: spin_unlock optimization(i386)

It will always return 0.
[...]
Erich Boleyn
PMD IA32 Architecture
Intel

Memory models in the Linux kernel

Manfred Spraul
spin_unlock optimization(i386)

the curr
`lock; b`
it takes
I think it
`movl $0`
which w
single-t

Linus Torvalds
Re: spin_unlock optimization(i386)

It does NOT
it, and then
eventually.
[…]
the above C
[…]
I might be p

Erich Boleyn
Re: spin_unlock optimization(i386)

It will always return 0.
[…]
Erich Boleyn
PMD IA32 Architecture
Intel

[119 emails later …]

Memory models in the Linux kernel

Manfred Spraul
spin_unlock optimization(i386)

the curr
`lock; b`
it takes
I think it
`movl $0`
which w
single-t

Linus Torvalds
Re: spin_unlock optimization(i386)

It does NOT
it, and then
eventually.
[…]
the above C
[…]
I might be p

Erich Boleyn
Re: spin_unlock optimization(i386)

It will always return 0.
[…]
Erich Boleyn
PMD IA32 Archite
Intel

[119 emails later …]

Linus Torvalds
Re: spin_unlock optimization(i386)

I'm happy.

Everybody has convinced me that yes, the Intel ordering rules are strong enough that all of this really is legal

Memory models in the Linux kernel

- New in 2018: a formal Linux kernel memory model
  - tools/memory-model/Documentation/explanation.txt
  - Only 12,000 words!

“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
    - Except through bugs — Spectre/Meltdown

Memory consistency models

- 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