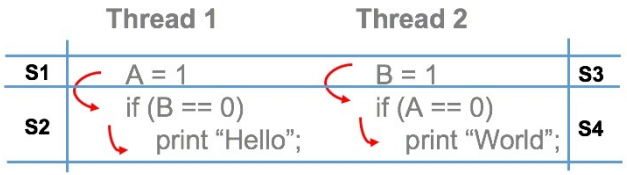
Lecture 09: Hardware Memory Consistency

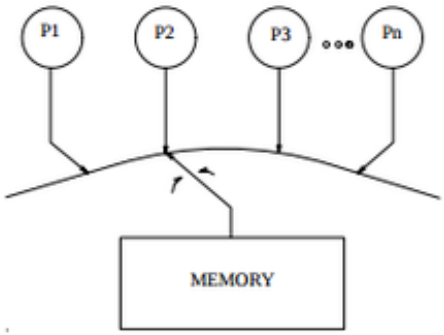
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture (Recap)

- Memory latency continues to limit the performance of multicore processors
  - Several optimizations inside processors for hiding the load/store latency
    - As a side effect of these optimizations, load/store inside a program could be reordered, and hence may not happen in the source code order as expected by programmer



- Memory consistency model defines a set of rules for valid set of reordering of **two different memory accesses**
  - Both compiler and processor can perform reordering

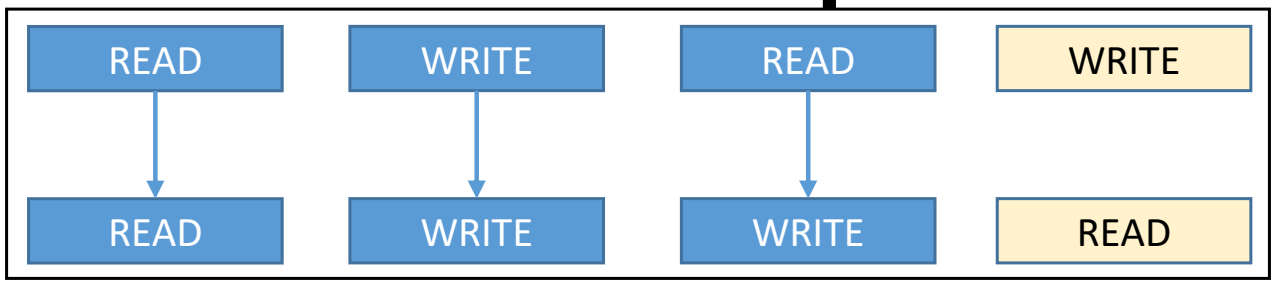


- Sequential consistency is the most primitive form of memory consistency that basically says memory access to any location always happens atomically, and the effect is visible to each and every core
  - Modern programming languages supports sequential consistency only for code block within a mutex lock/unlock operation (Data Race Free)

Today's Class

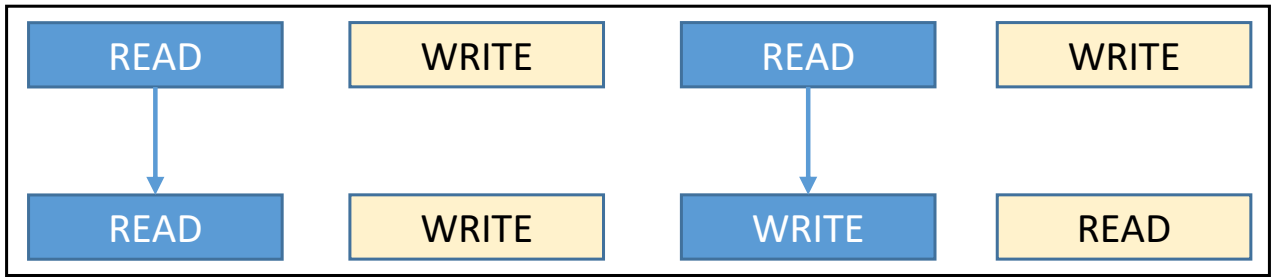
- x86-TSO memory consistency model
- Store buffer

Few Alternatives to Sequential Consistency

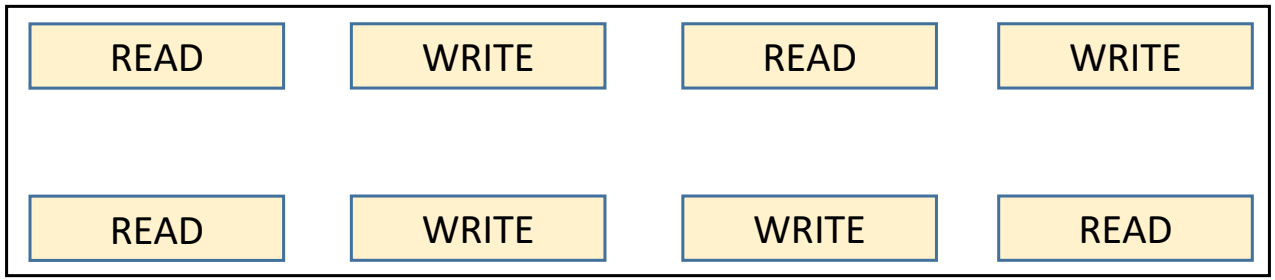


X86-TSO (Total Store Order)
Intel/AMD processors

Don't start the next operation until the previous completes

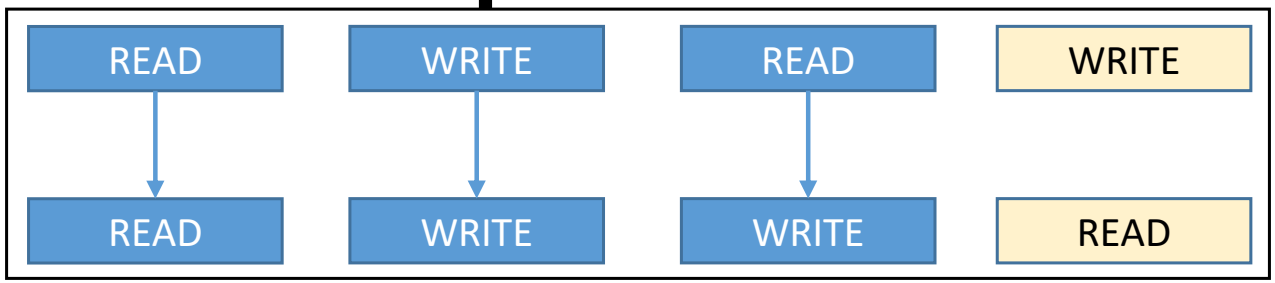


PSO (Partial Store Order)
SPARC processor



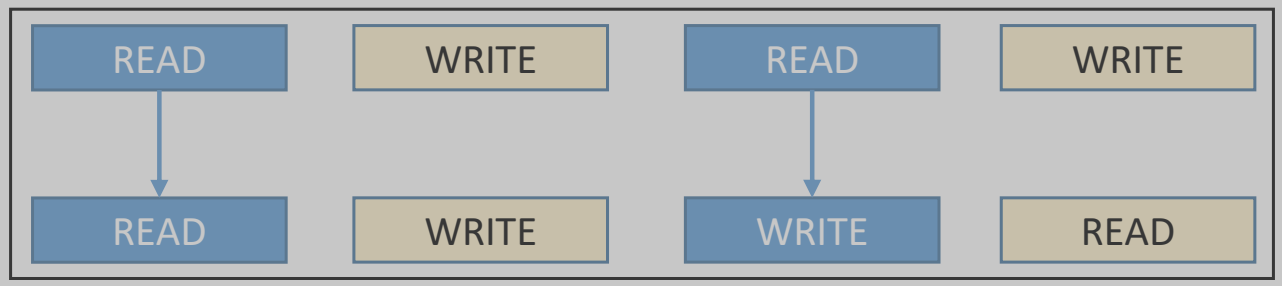
Relaxed Memory Ordering
IBM Power PC, ARM, etc.

Let's take a deep dive in x86-TSO model

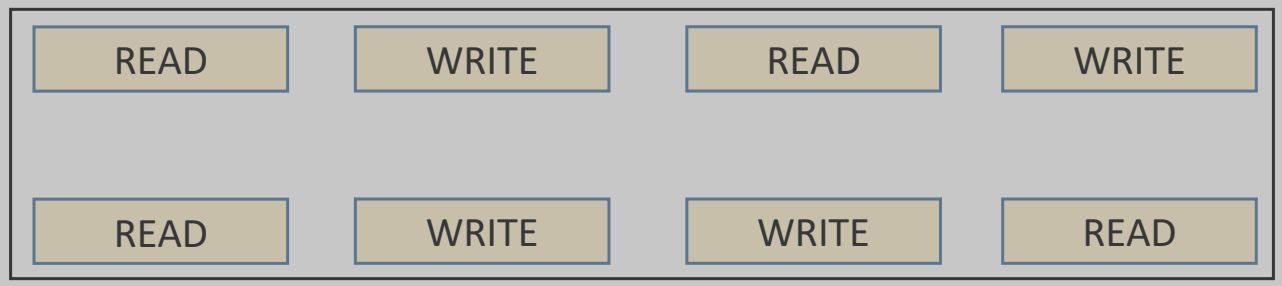


X86-TSO (Total Store Order)
Intel/AMD processors

Don't start the next operation until the previous completes



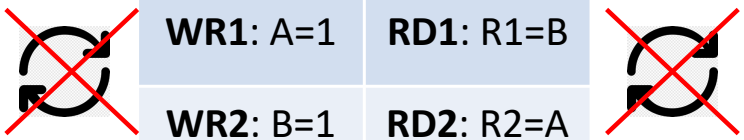
PSO (Partial Store Order)
SPARC processor



Relaxed Memory Ordering
IBM Power PC, ARM, etc.

x86-TSO Behaviors

- **Rule-1&2**: Read-Read & Write-Write reordering is NOT allowed (over same core)



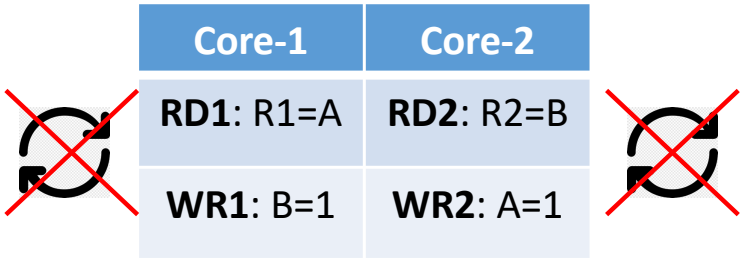
Core-1	Core-2
WR1: A=1	**RD1**: R1=B
WR2: B=1	**RD2**: R2=A

Question: What is an invalid set of result for R1 & R2? Assume initially A=B=0

Answer: R1=1 & R2= 0

x86-TSO Behaviors

- **Rule-3**: Writes are NOT reordered with earlier Reads (over same core)



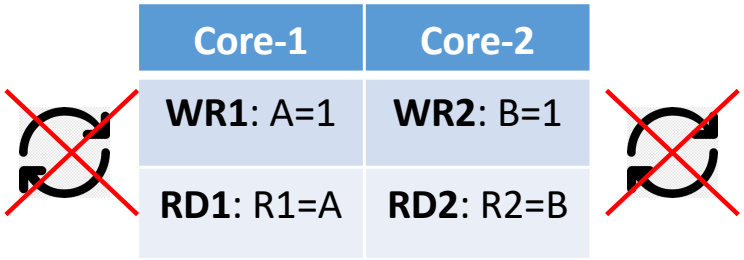
Core-1	Core-2
RD1: R1=A	**RD2**: R2=B
WR1: B=1	**WR2**: A=1

Question: What is an invalid set of result for R1 & R2? Assume initially A=B=0

Answer: R1=1 & R2= 1

x86-TSO Behaviors

- **Rule-4a**: Reads CANNOT be reordered with earlier Writes to **SAME** memory locations (over same core)



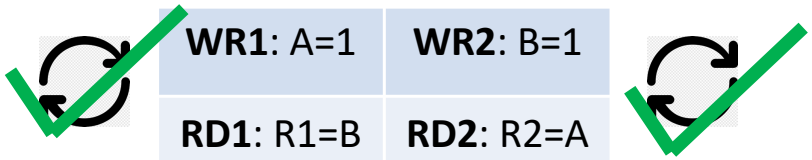
Core-1	Core-2
WR1: A=1	**WR2**: B=1
RD1: R1=A	**RD2**: R2=B

Question: What is an invalid set of result for R1 & R2? Assume initially A=B=0

Answer: R1=0 & R2= 0

x86-TSO Behaviors

- **Rule-4b**: Reads CAN be reordered with earlier Writes to **DIFFERENT** memory locations (over same core)



Core-1	Core-2
WR1: A=1	**WR2**: B=1
RD1: R1=B	**RD2**: R2=A

Question: What is an invalid set of result for R1 & R2? Assume initially A=B=0

Answer: None!

x86-TSO Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of the other

Core-1	Core-2
WR1: A=1	**WR2**: B=1
RD1a: R1=A	**RD2a**: R3=B
RD1b: R2=B	**RD2b**: R4=A

- Can we change the orderings below?
  - Core-1: WR1->RD1a->RD1b
  - Core-2: WR2->RD2a->RD2b
  - Not possible due to the Rules 1 and 4
- It could happen that R2 = R4 = 0 ✓
  - But why?

Store Buffers

Store Buffer [edit]

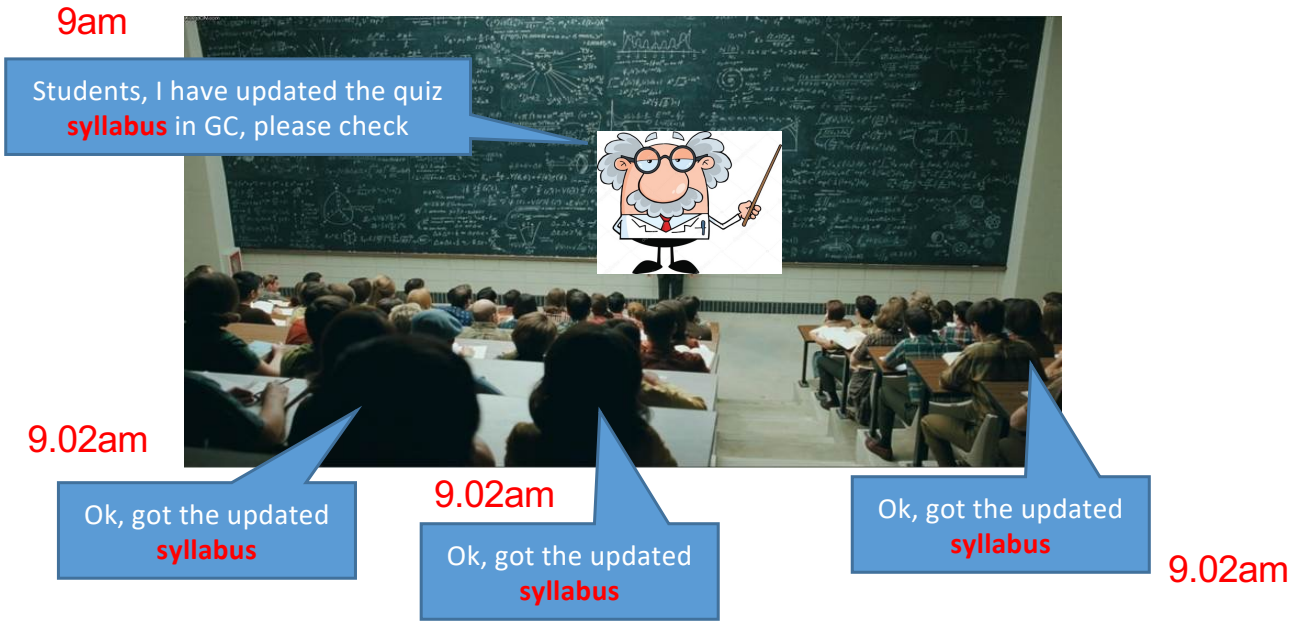
A store buffer is used when writing to an invalid cache line. Since the write will proceed anyway, the CPU issues a read-invalid message (hence the cache line in question and all other CPUs' cache lines that store that memory address are invalidated) and then pushes the write into the store buffer, to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

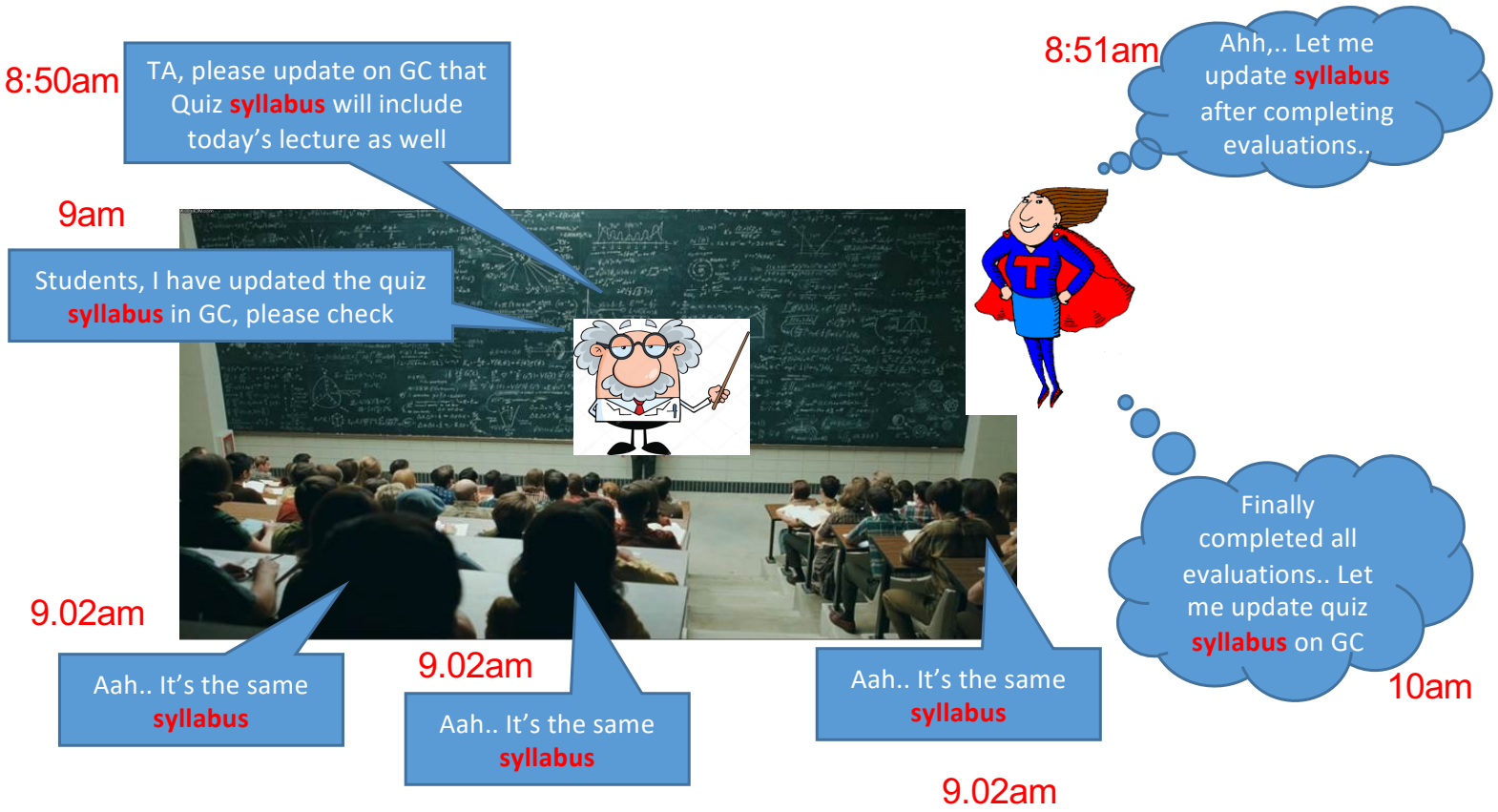
From Wikipedia

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

Coherence without Store Buffer



Coherence with Store Buffer

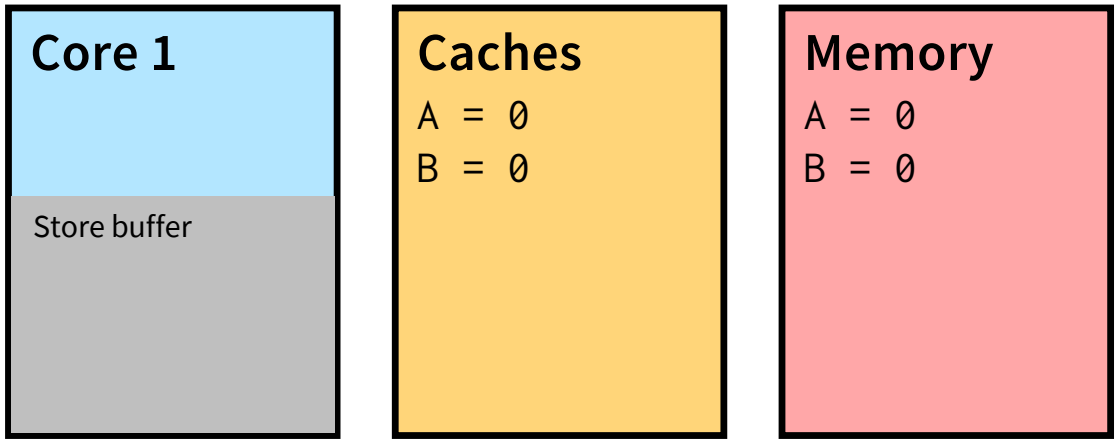


Store Buffers

- 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

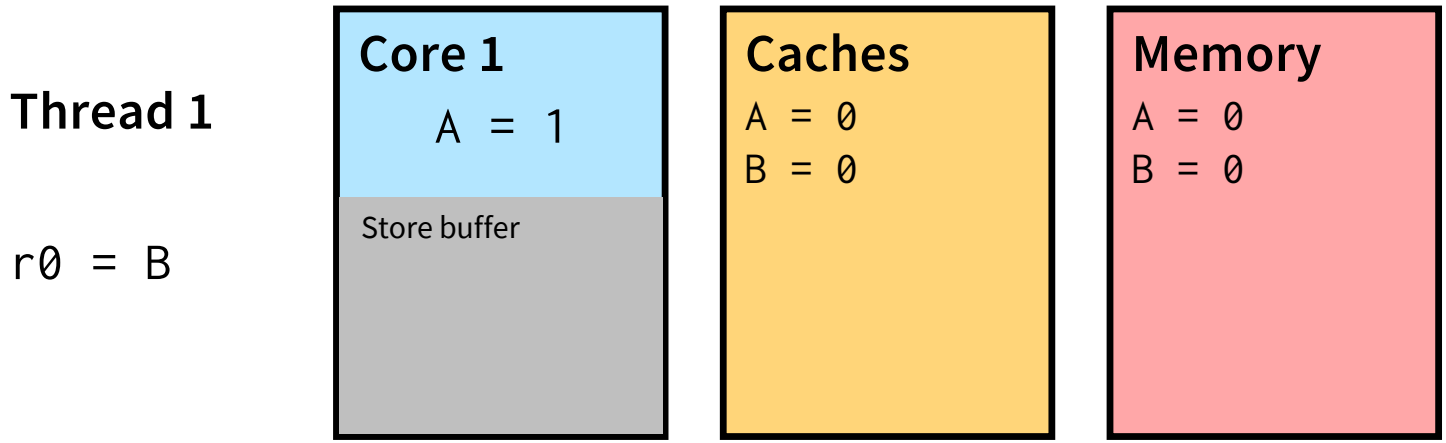
Thread 1

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



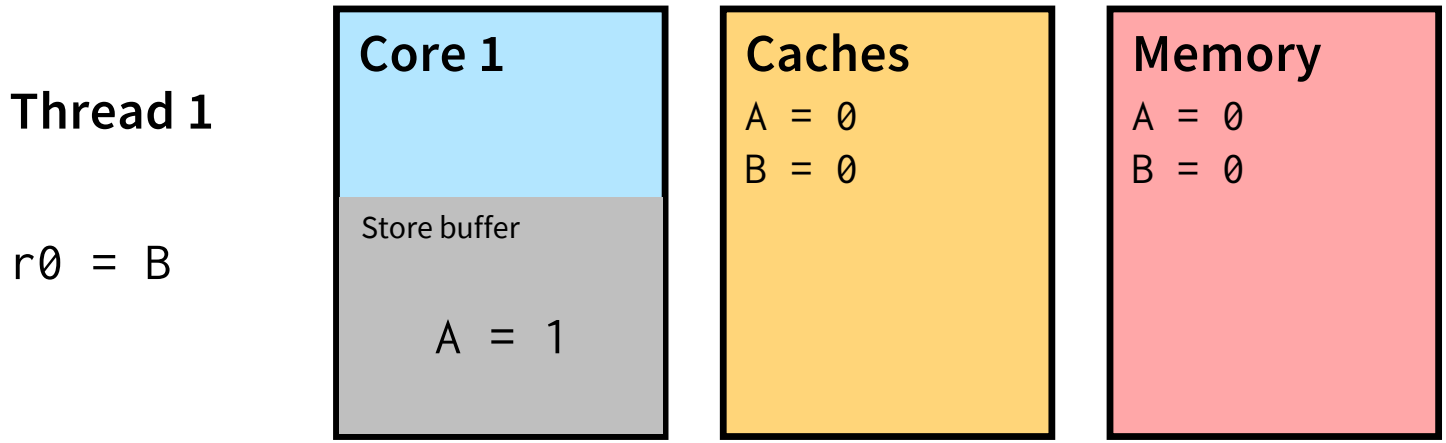
Store Buffers

- 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



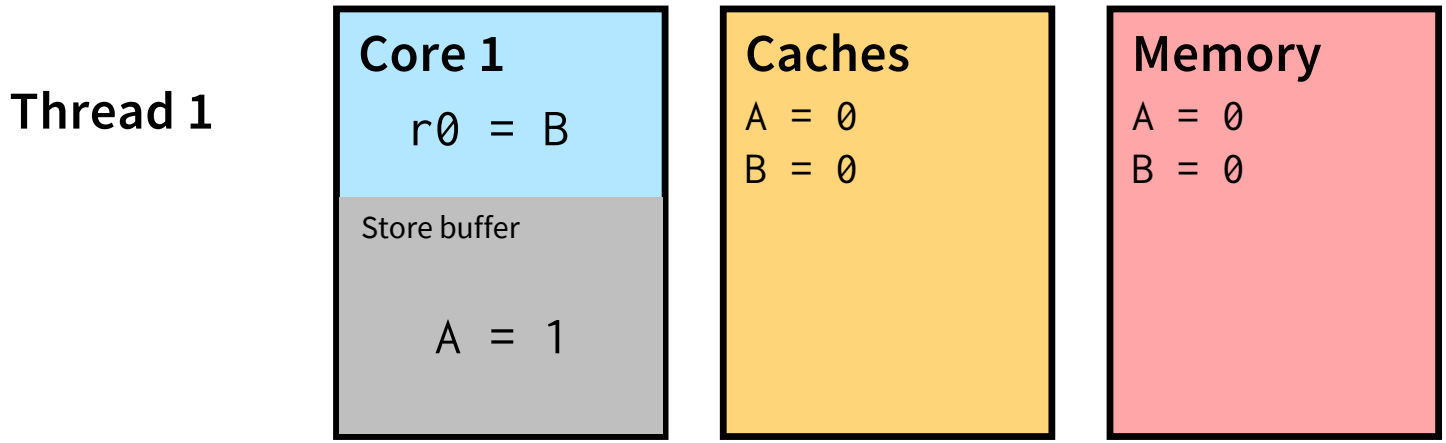
Store Buffers

- 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

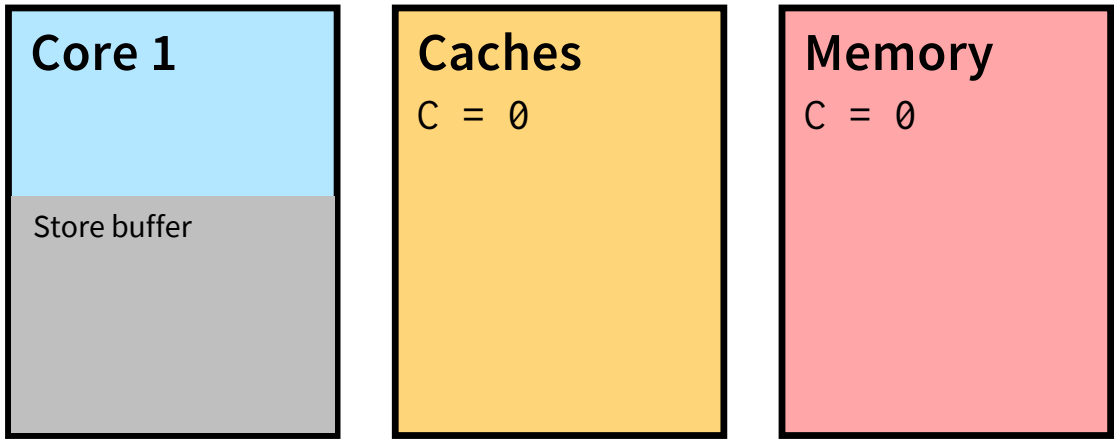


Store Buffers

- 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

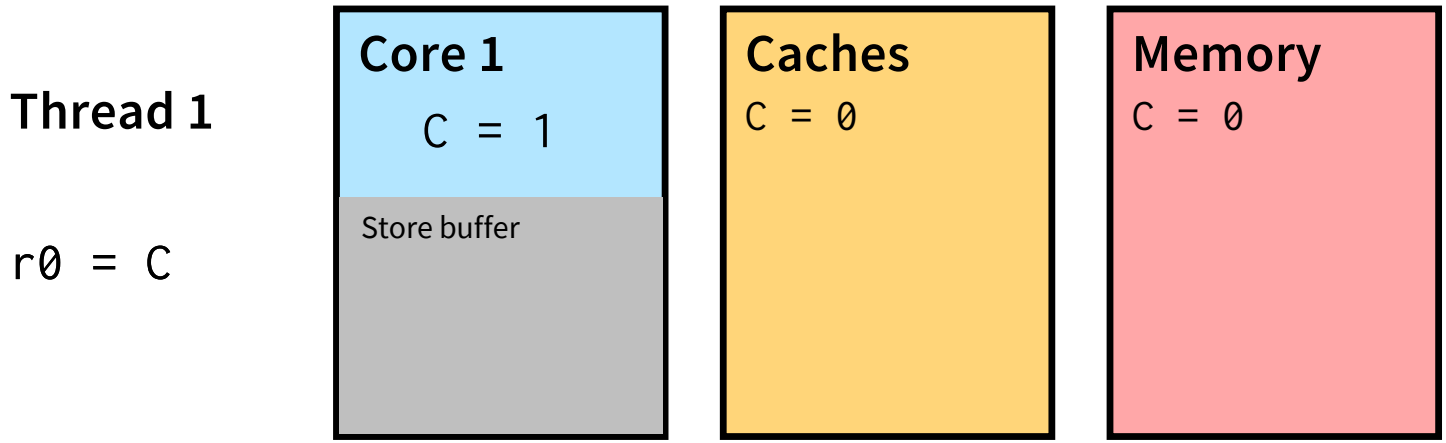
Thread 1

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



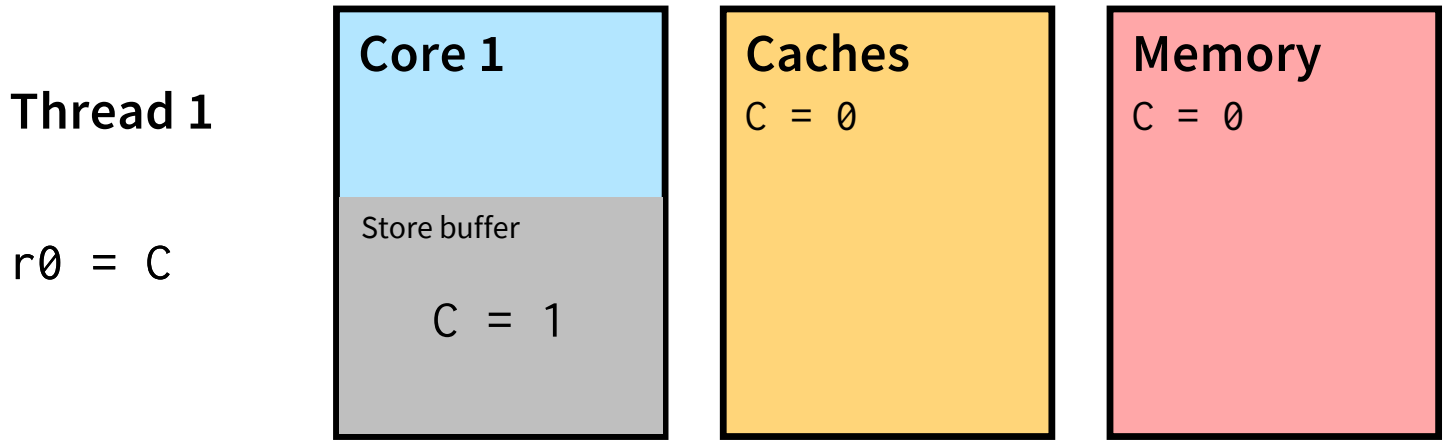
Store Buffers

- 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



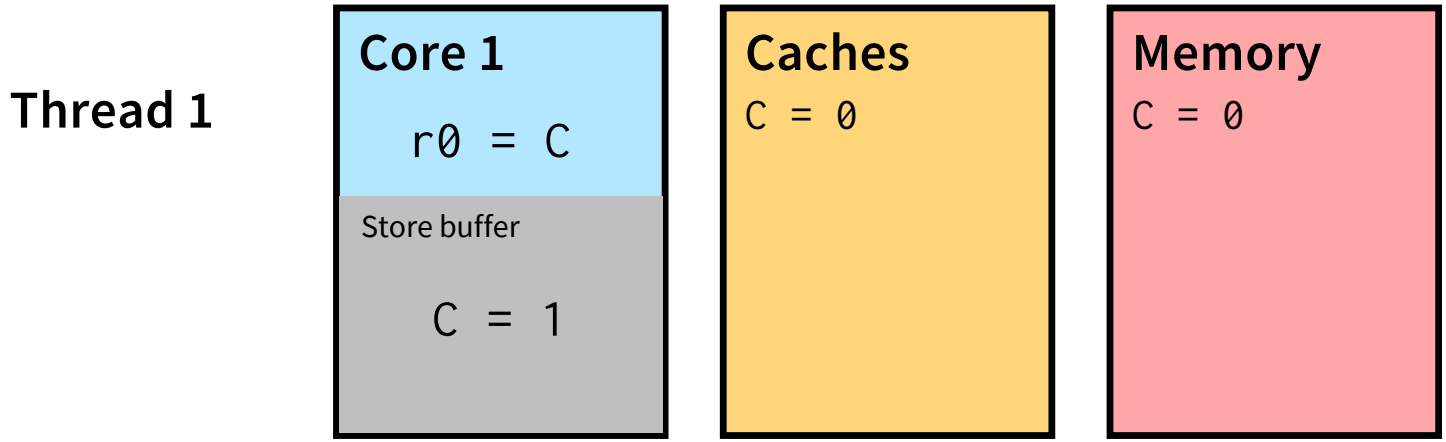
Store Buffers

- 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



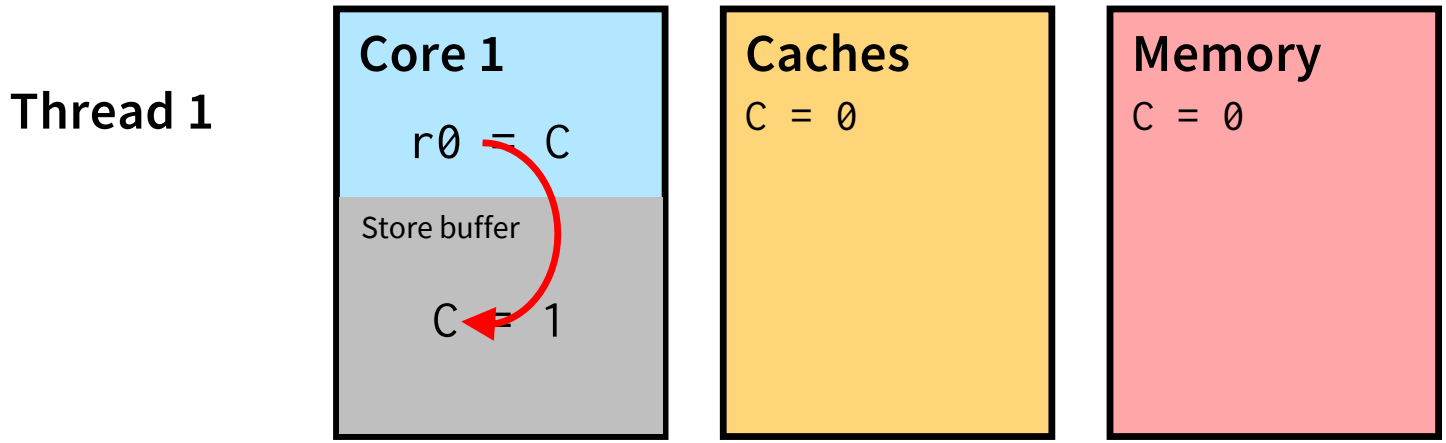
Store Buffers

- 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



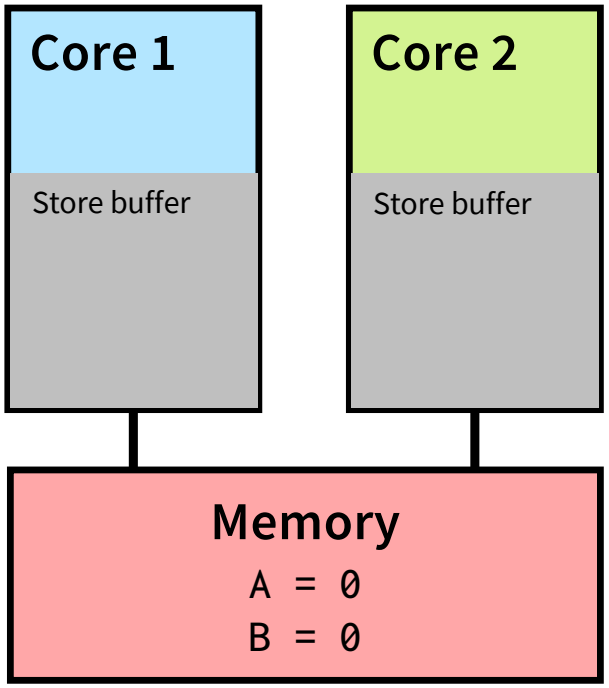
Store Buffers

- 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 Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

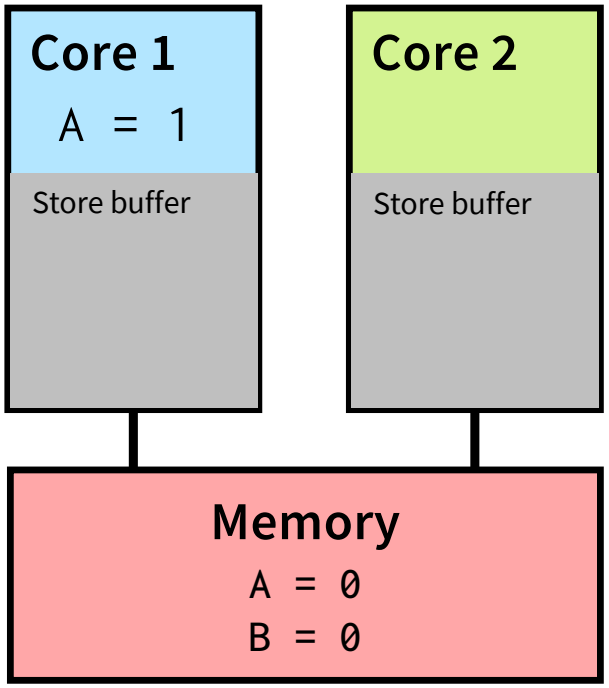


Thread 1	Thread 2
(1) `A = 1`	(3) `B = 1`
(2) `r0 = B`	(4) `r1 = A`

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

x86-TSO Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

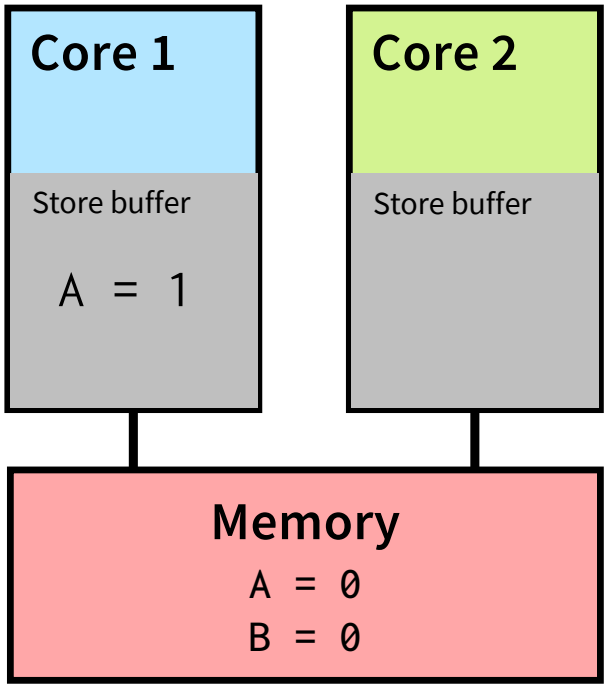


Thread 1	Thread 2
(1)	(3) B = 1
(2) r0 = B	(4) r1 = A

Can r0 = 0 and r1 = 0?

x86-TSO Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

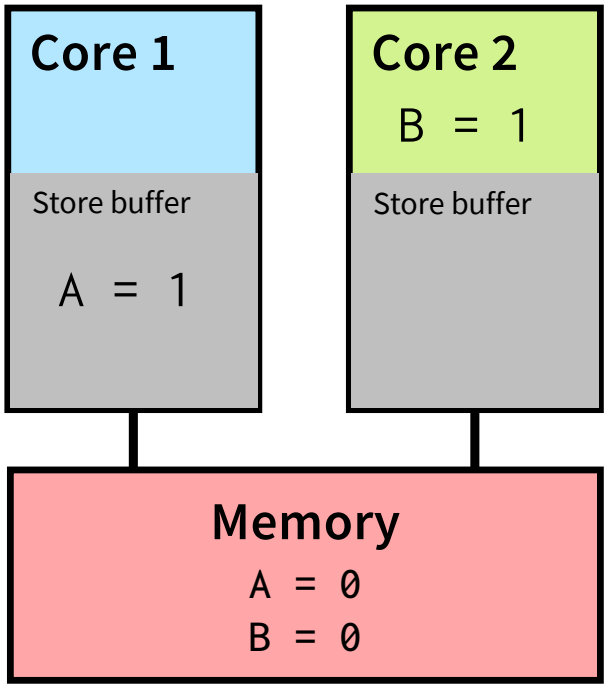


Thread 1	Thread 2
(1)	(3) B = 1
(2) r0 = B	(4) r1 = A

Can r0 = 0 and r1 = 0?

x86-TSO Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

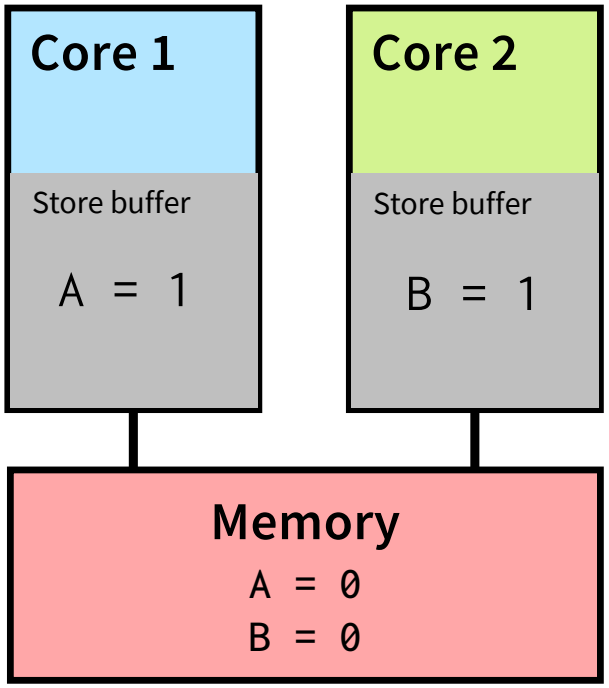


Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

x86-TSO Behaviors

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other

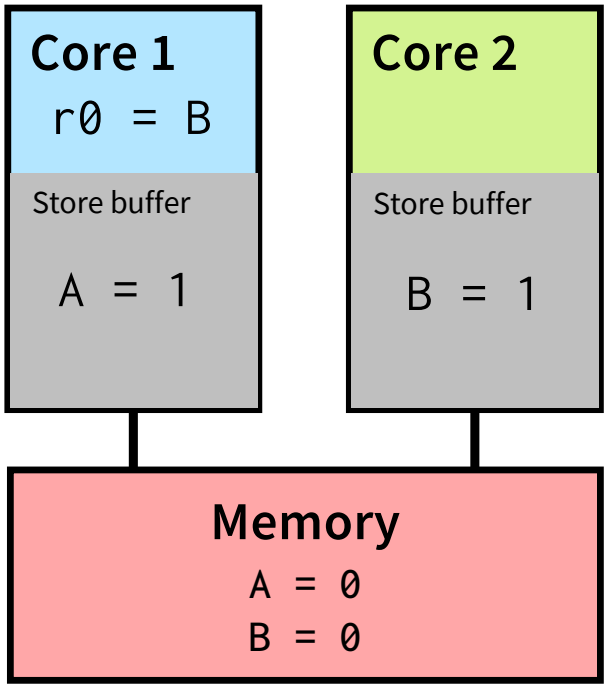


Thread 1	Thread 2
(1)	(3)
(2) `r0 = B`	(4) `r1 = A`

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

x86-TSO Behaviors

- **Rule-5**: 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`

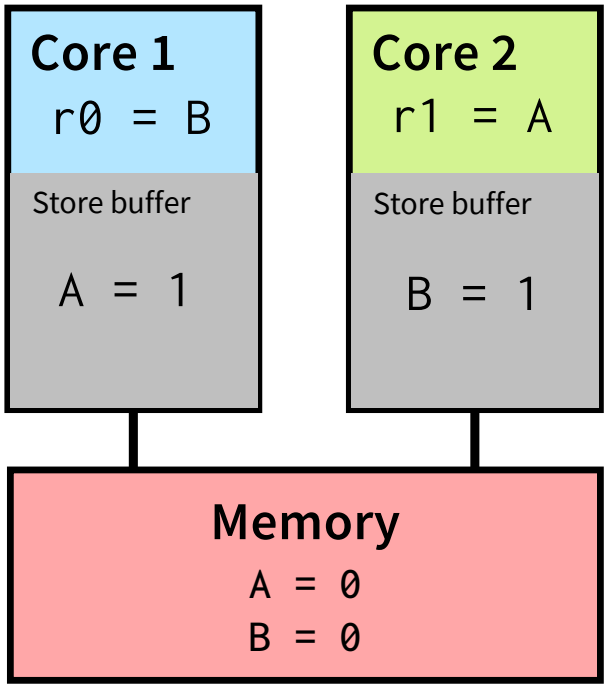
Can `r0 = 0` and `r1 = 0`?

Executed

`r0 = B (= 0)`

x86-TSO Behaviors

- **Rule-5**: 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)

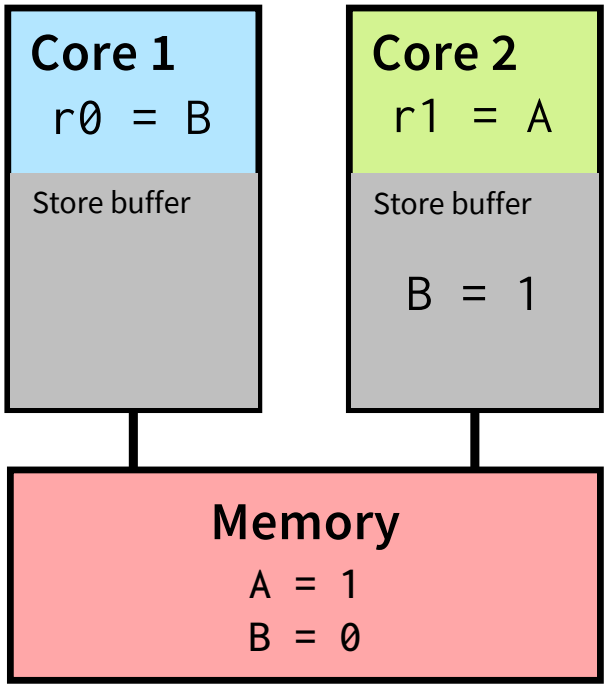
Can `r0 = 0` and `r1 = 0`?

Executed

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

x86-TSO Behaviors

- **Rule-5**: 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)

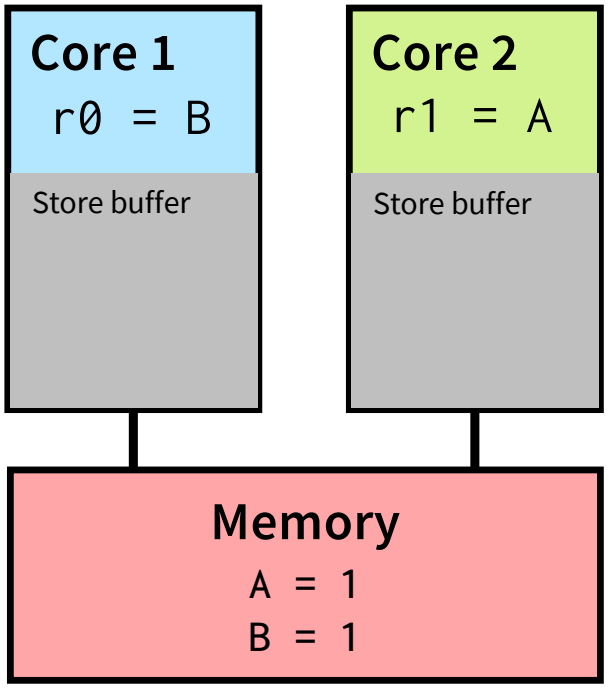
Can `r0 = 0` and `r1 = 0`?

Executed

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

x86-TSO Behaviors

- **Rule-5**: 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
```

x86-TSO Behaviors

- **Rule-6**: Writes that are causally related, appear to all processors to occur in an order consistent with the causality relation

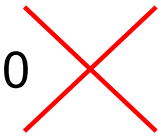
Core-1	Core-2	Core-3
WR1: A=1		
	RD2: R1=A	**RD3a**: R2=B
	WR2: B=1	**RD3b**: R3=A

- If R1 = R2 = 1, can R3 = 0 
  - R1=1
    - Implies WR1 → RD2
  - RD2 → WR2 (Rule-3)
  - RD3a → RD3b (Rule-1)
  - R2 = 1
    - Implies WR1→RD2→WR2→RD3a
    - Hence, R3 cannot be 0
      - **R3 = 1**

x86-TSO Behaviors

- **Rule-7**: Any two Writes must appear to execute in the same order to all cores other than those performing the writes

Core-1	Core-2	Core-3	Core-4
WR1: A=1	**WR2**: B=1	**RD3a**: R1=A	**RD4a**: R3=B
		RD3b: R2=B	**RD4b**: R4=A

- Is it possible
  - R1=1, R2=0, R3=1, R4=0 
- R1=1 & R2=0
  - Implies WR1 → WR2 w.r.t. Core-3
- R3=1, R4=0 implies:
  - Either RD3b → RD3a
    - Violates Rule-1
  - Or, WR2 → WR1 w.r.t. Core-2
    - Recall value propagation property of cache coherence
      - If A=1 has propagated to Core-3 before B=1 then A=1 must also have been propagated to Core-4 before B=1

x86-TSO Behaviors

- **Rule-8**: All cores agree on a single execution order of locked instructions
  - **I.E., Rule-7 hold true even with LOCK statements**

Core-1	Core-2	Core-3	Core-4
WR1: A=1	**WR2**: B=1	**RD3a**: R1=A	**RD4a**: R3=B
		RD3b: R2=B	**RD4b**: R4=A

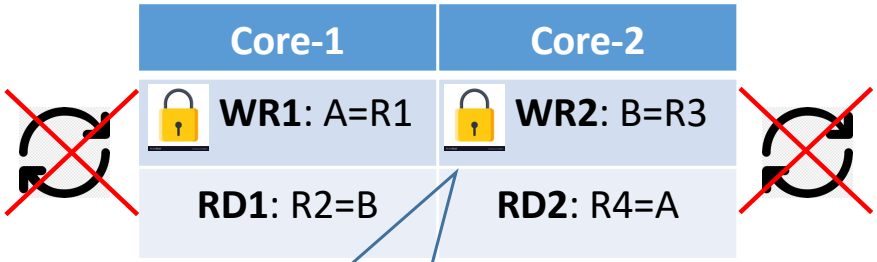
Same example as in previous slide (Rule #7) except the LOCK instruction

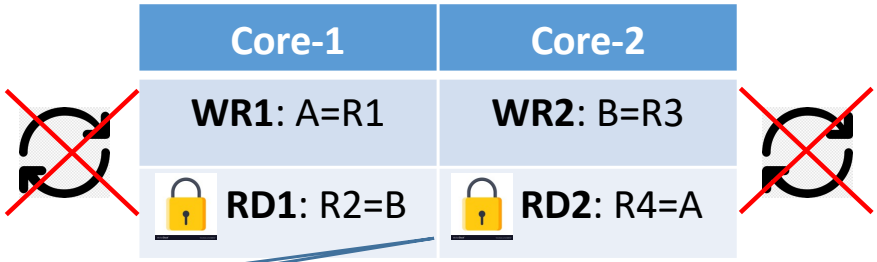
Same example as in previous slide (Rule #7). Its hold true even with LOCK instructions

- Is it possible
  - R1=1, R2=0, R3=1, R4=0
- R1=1 & R2=0
  - Implies WR1 → WR2 w.r.t. Core-3
- R3=1, R4=0 implies:
  - Either RD3b → RD3a
    - Violates Rule-1
  - Or, WR2 → WR1 w.r.t. Core-2
    - Recall value propagation property of cache coherence
      - If A=1 is propagated to Core-3 before B=1 then A=1 must also have been propagated to Core-4

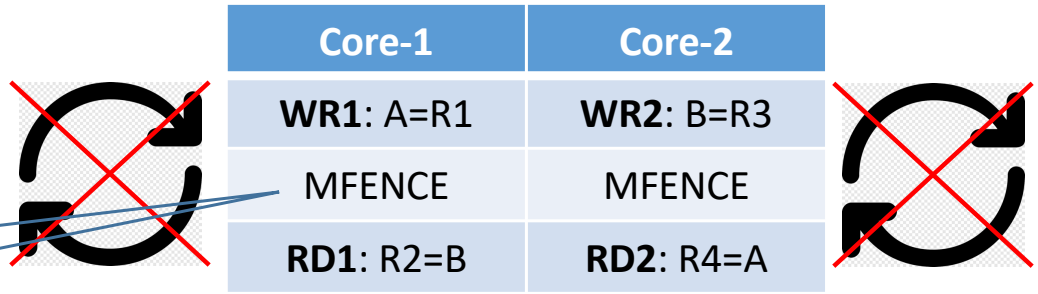
x86-TSO Behaviors

- **Rule-9**: Reads and Writes are NOT reordered with Locked instructions





Question: what would happen if we remove any of the LOCKED instructions?



Adding MFENCE will have the same effect as in above examples with LOCK

Reference Materials

- 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 (L #10)

- Language memory consistency model
- Next lecture tomorrow at 2.30pm