

Memory Consistency

A Crash Course

Brandon Lucia
CSE 471

Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”

Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

Memory Consistency Model

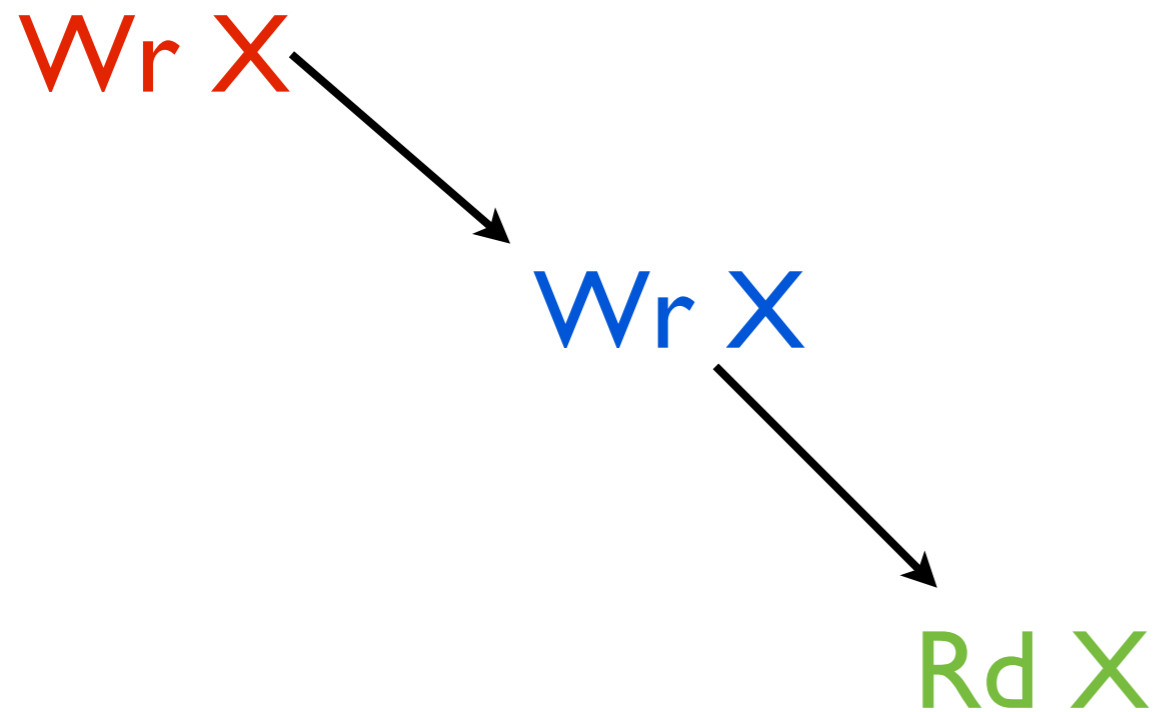
Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

“Defines which reorderings of memory operations are permitted”

Review: Coherence

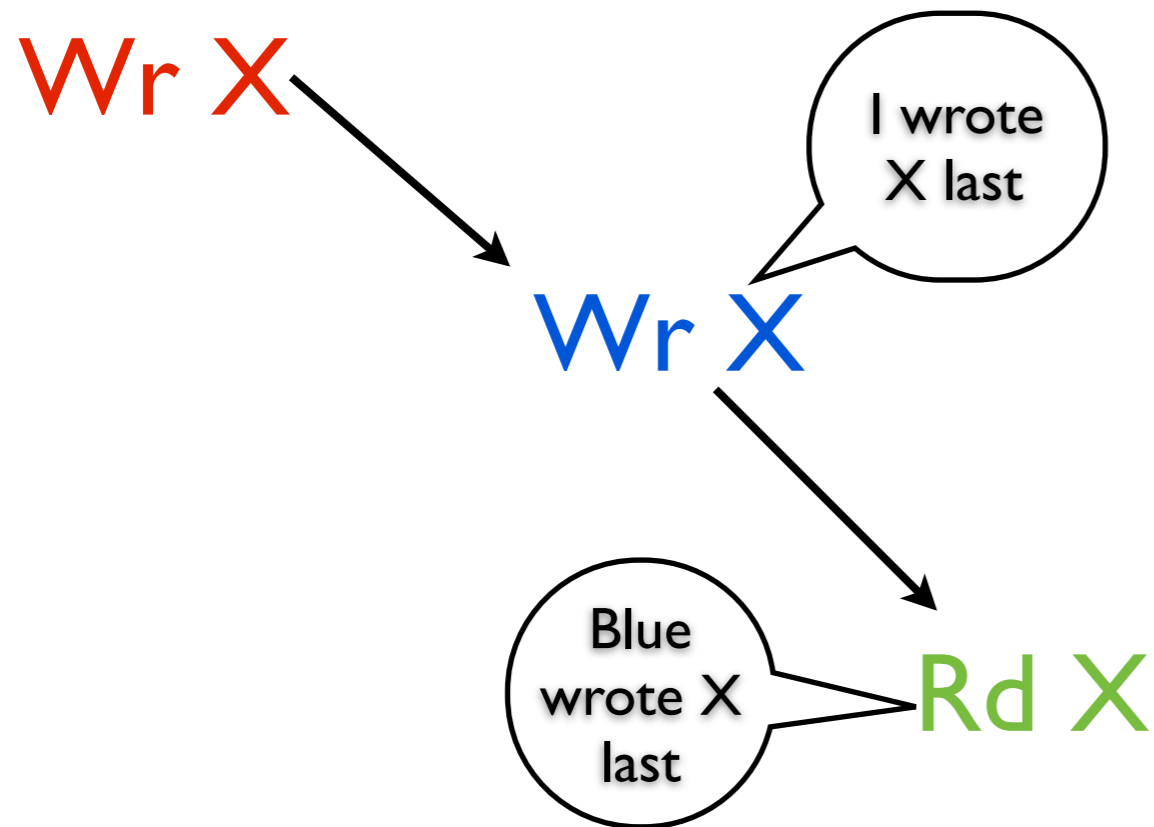


2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading X gets the value of the last write to X ”

Review: Coherence



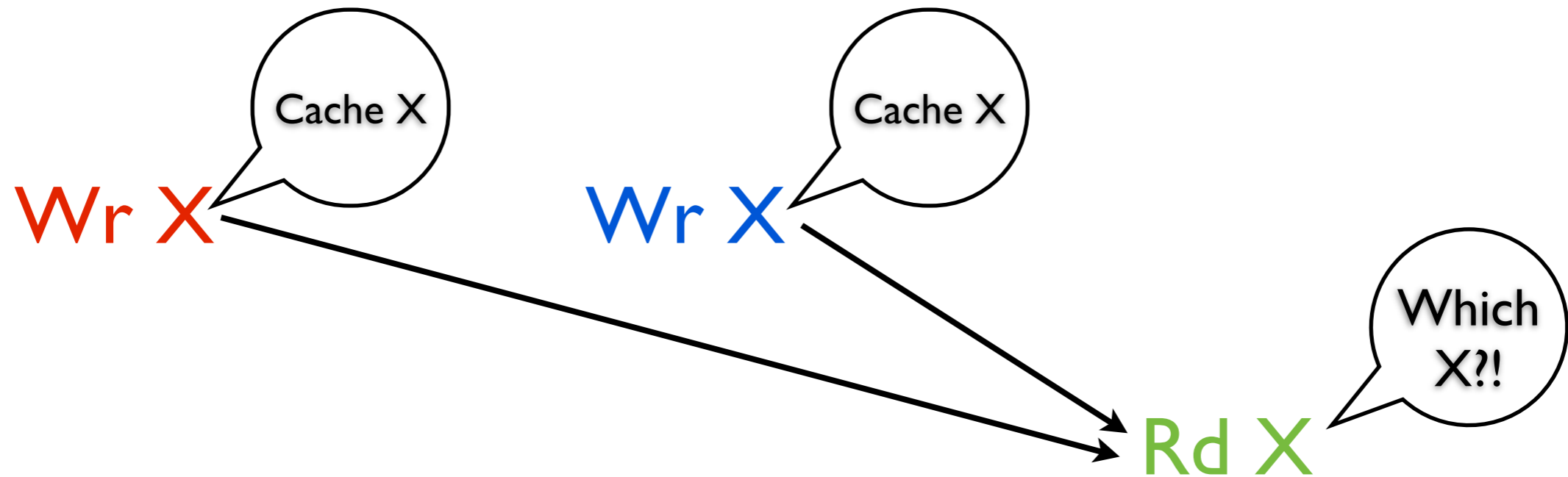
2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading X gets the value of the **last** write to X”

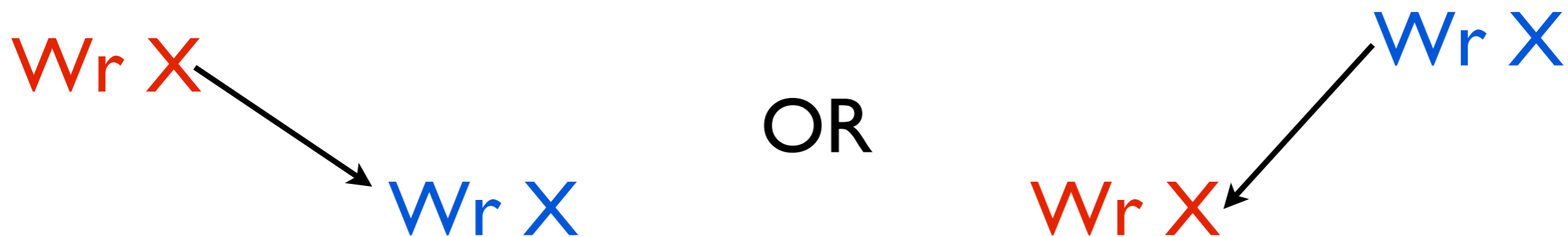
Without Coherence

(The coherence invariants prevent this from happening)



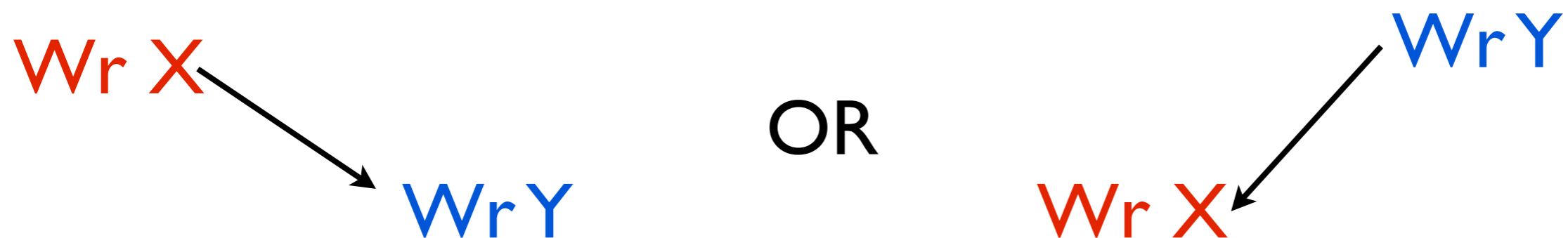
Processors can't decide who wrote last.
Green is hosed.

Coherence is Ordering



Coherence defines the set of legal orders of accesses to a **single** memory location

Consistency is Ordering



Consistency defines the set of legal orders of accesses to **multiple** memory locations

Sequential Consistency (SC)

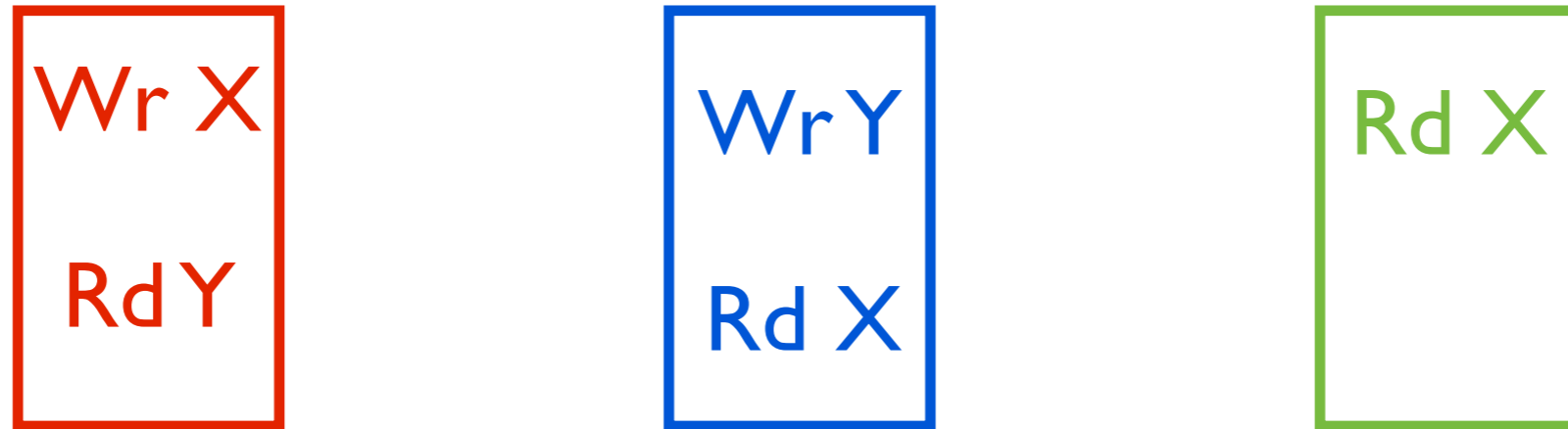
The simplest, most intuitive memory consistency model

Two Invariants to SC:

Instructions are
executed in program
order

All processors agree
on a total order of
executed instructions

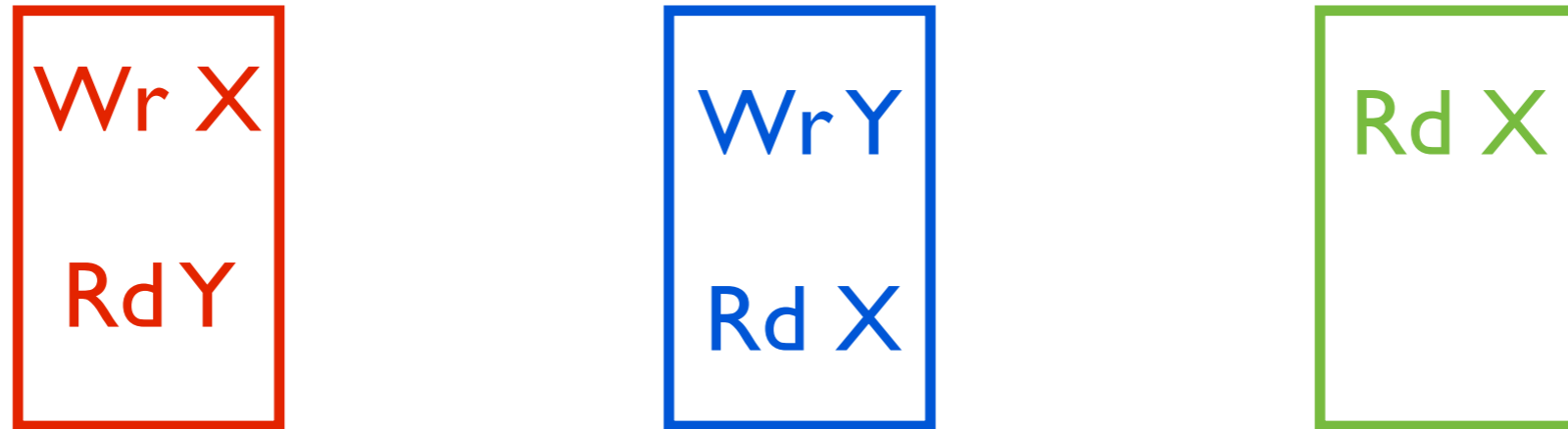
The SC “Switch”



Execution



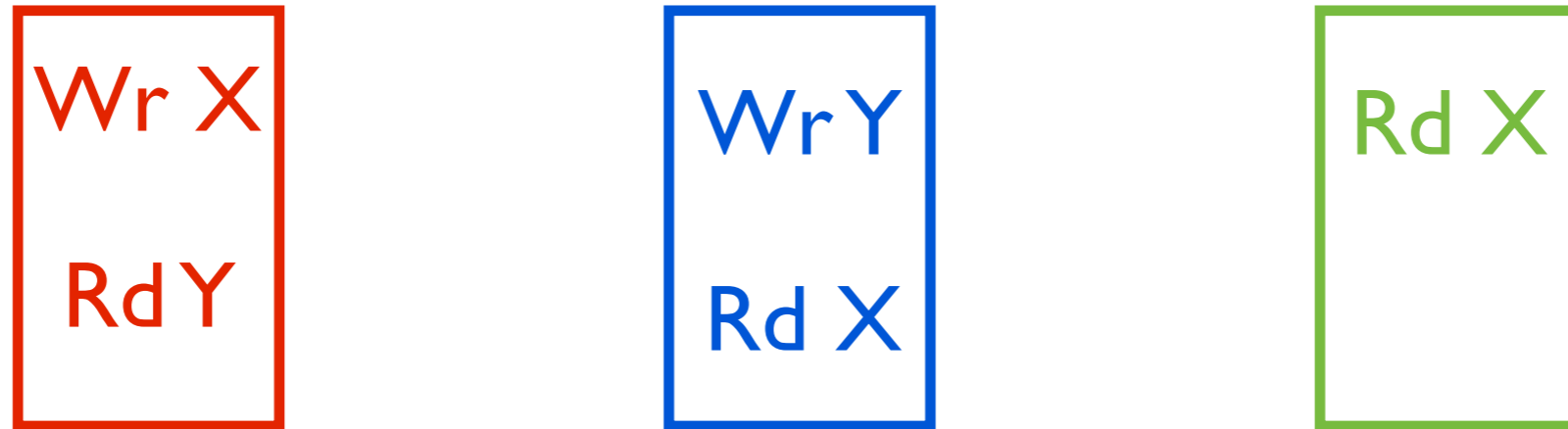
The SC “Switch”



Execution
Wr X



The SC “Switch”



Execution

Wr X
Rd Y



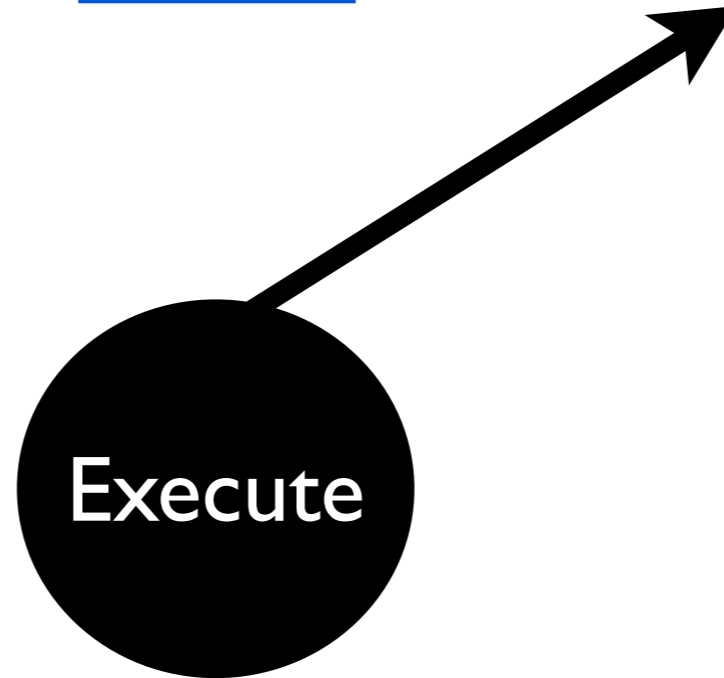
The SC “Switch”



Execution

Wr X
Rd Y
Wr Y

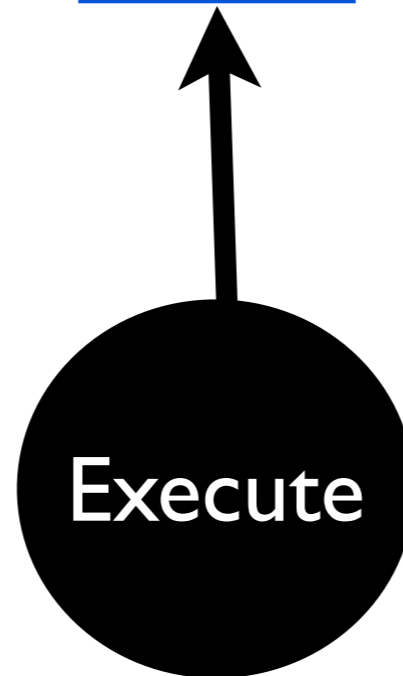
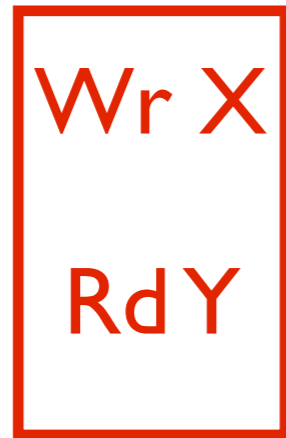
The SC “Switch”



Execution

Wr X
Rd Y
Wr Y
Rd X

The SC “Switch”



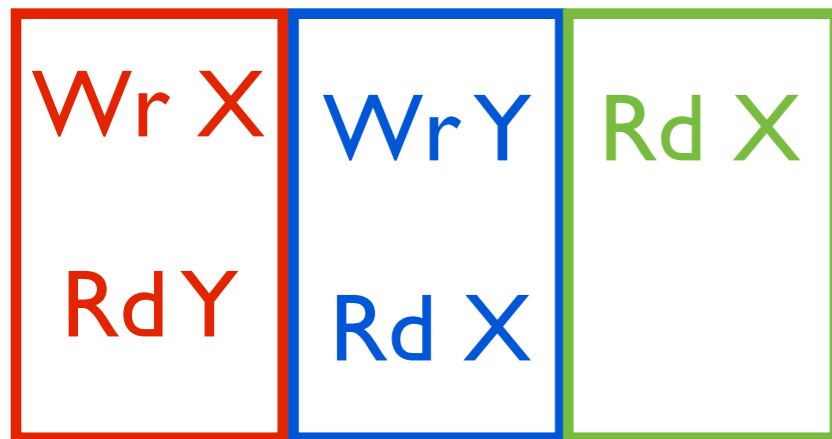
Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X

Why is SC Important?

Who cares?.... **You** care!

SC is how **programmers** think.



Intuitive (SC)

Wr X

Rd Y

Wr Y

Rd X

Rd X

Weird (not SC)

Rd Y

Wr X

Rd X

Rd X

Wr Y

SC prohibits **all** reordering of instructions (Invariant I)

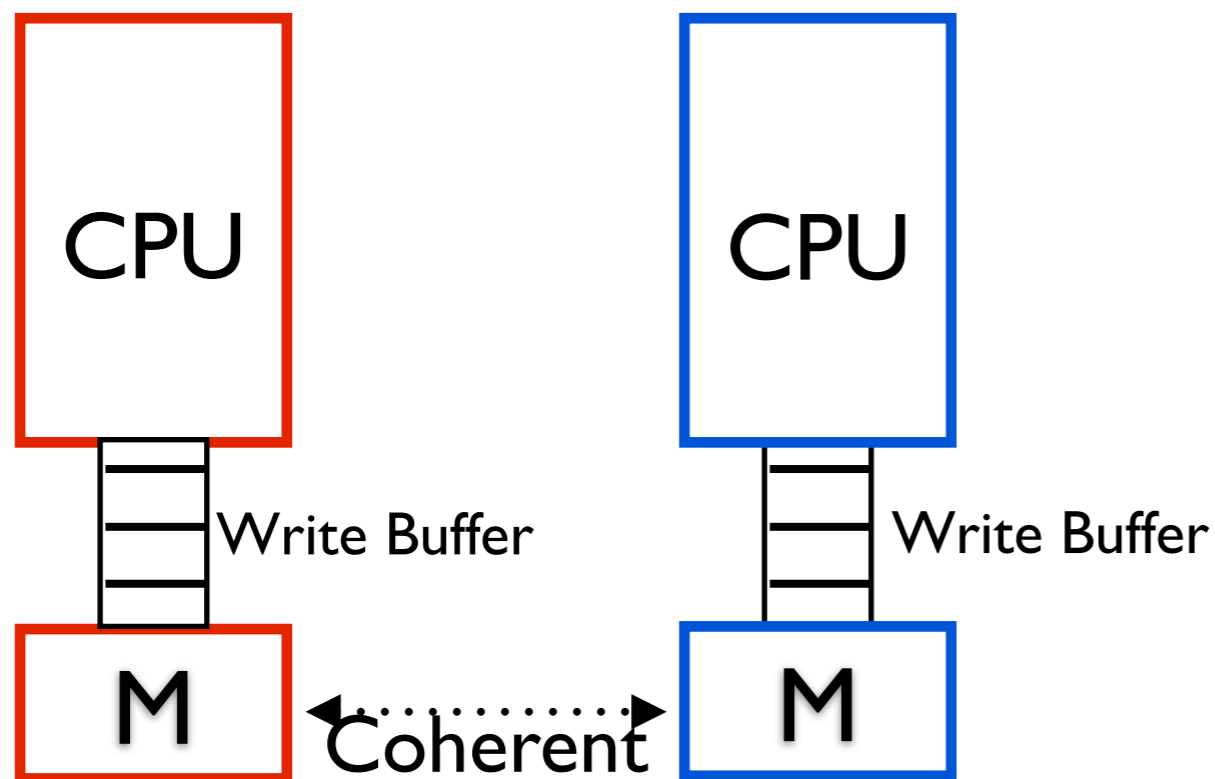
Why are Instructions Reordered?

And when does it matter anyway?

Why are Instructions Reordered?

Optimization.

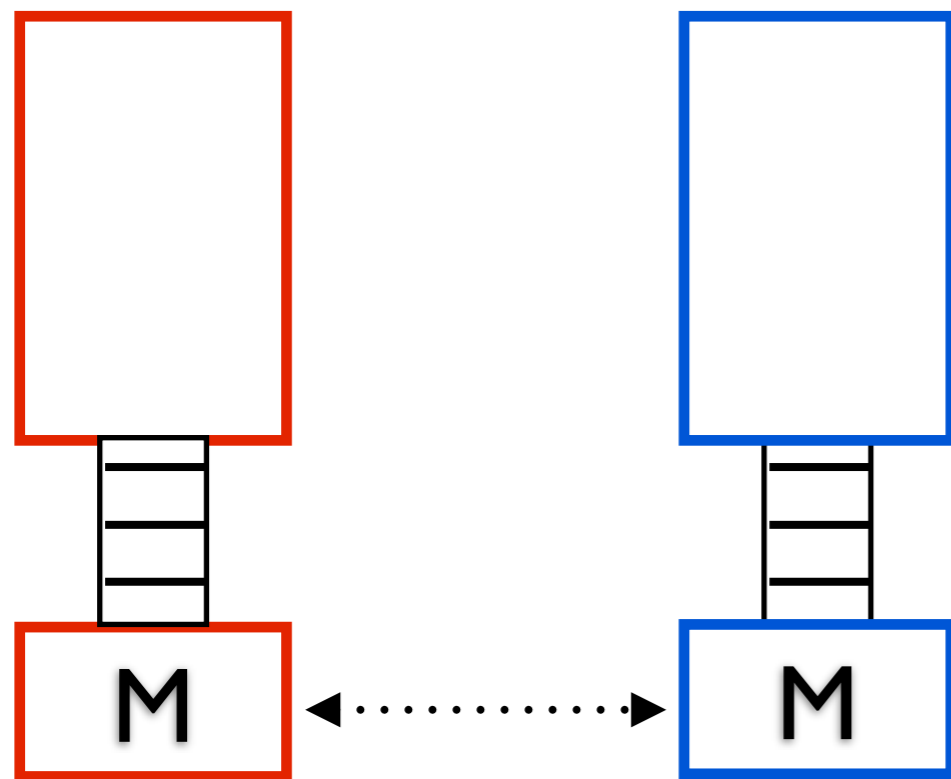
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

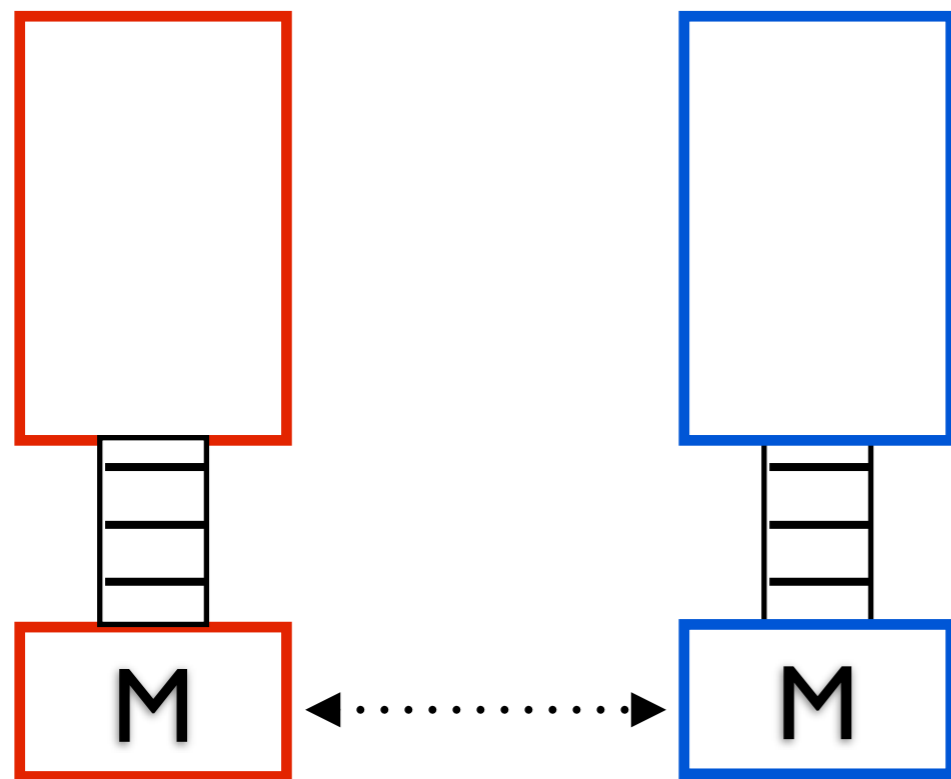
Initially $X == Y == 0$

$X=1$ $Y=1$

$r1=Y$ $r2=X$

Is $r1 == r2 == 0$
a valid result?

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

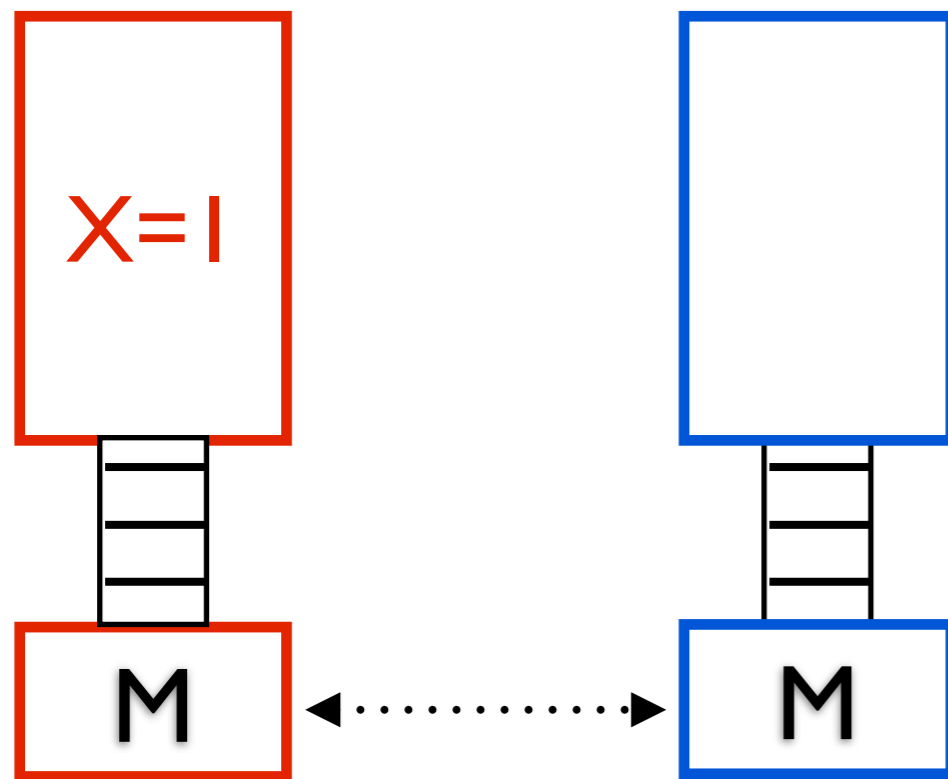
$X=1$ $Y=1$

$r1=Y$ $r2=X$

Is $r1 == r2 == 0$
a valid result?

$r1 == r2 == 0$ is **not** SC, but it can happen with write buffers

Reordering #1: Write Buffers



Program
Initially $X == Y == 0$

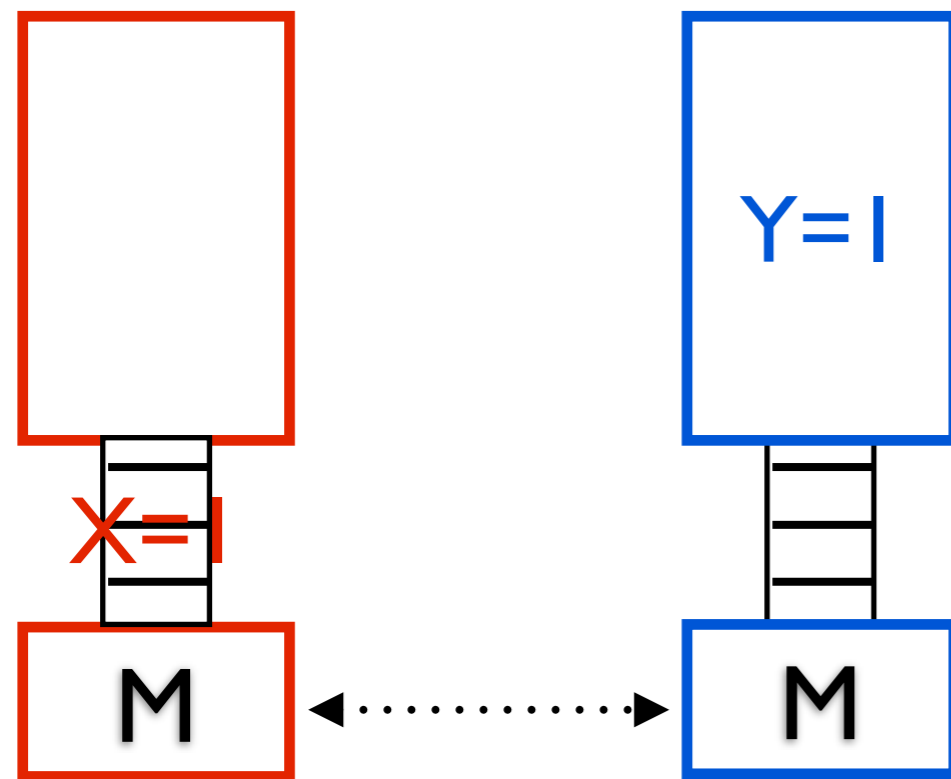
$Y=1$

$r1=Y$

$r2=X$

Execution

Reordering #1: Write Buffers



Program

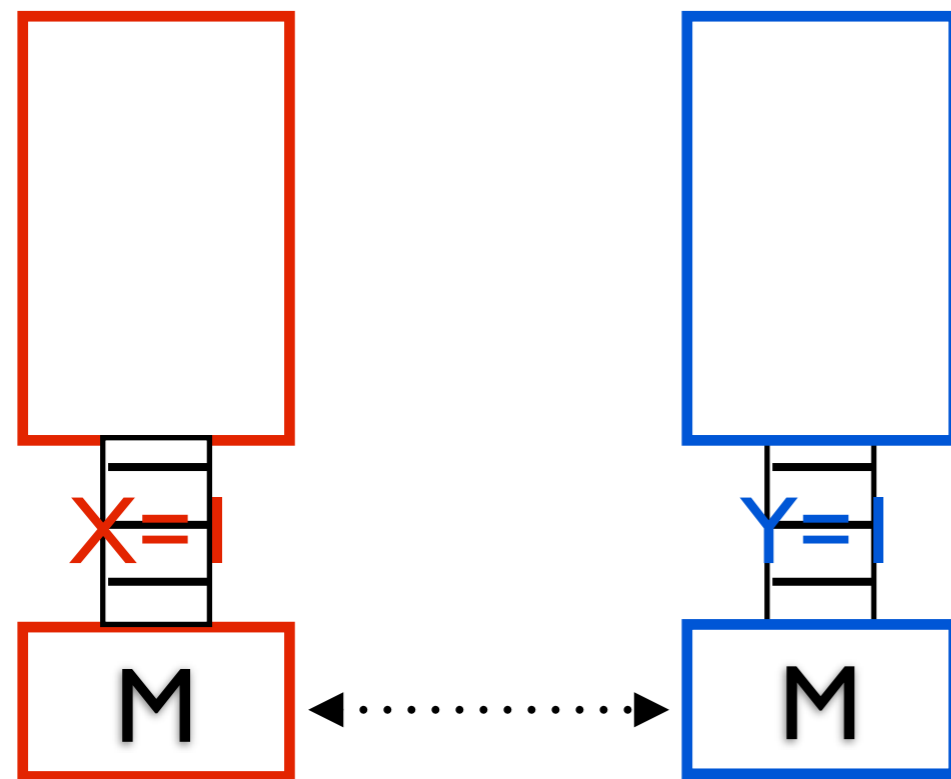
Initially $X == Y == 0$

$r1=Y$

$r2=X$

Execution

Reordering #1: Write Buffers

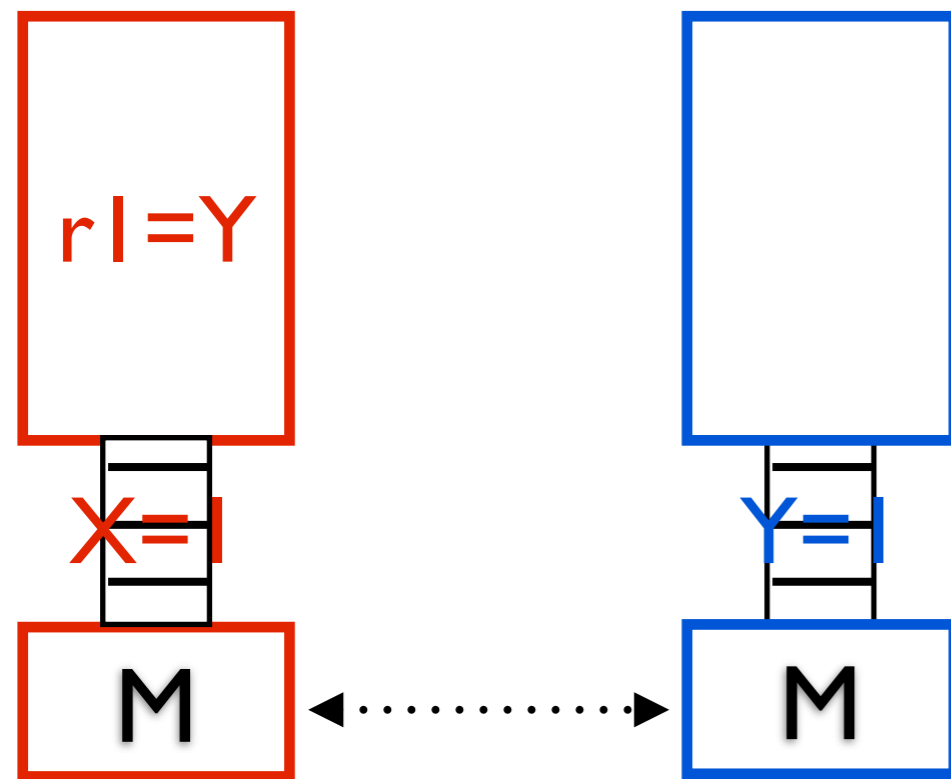


Program
Initially $X == Y == 0$

$r1=Y$ $r2=X$

Execution

Reordering #1: Write Buffers

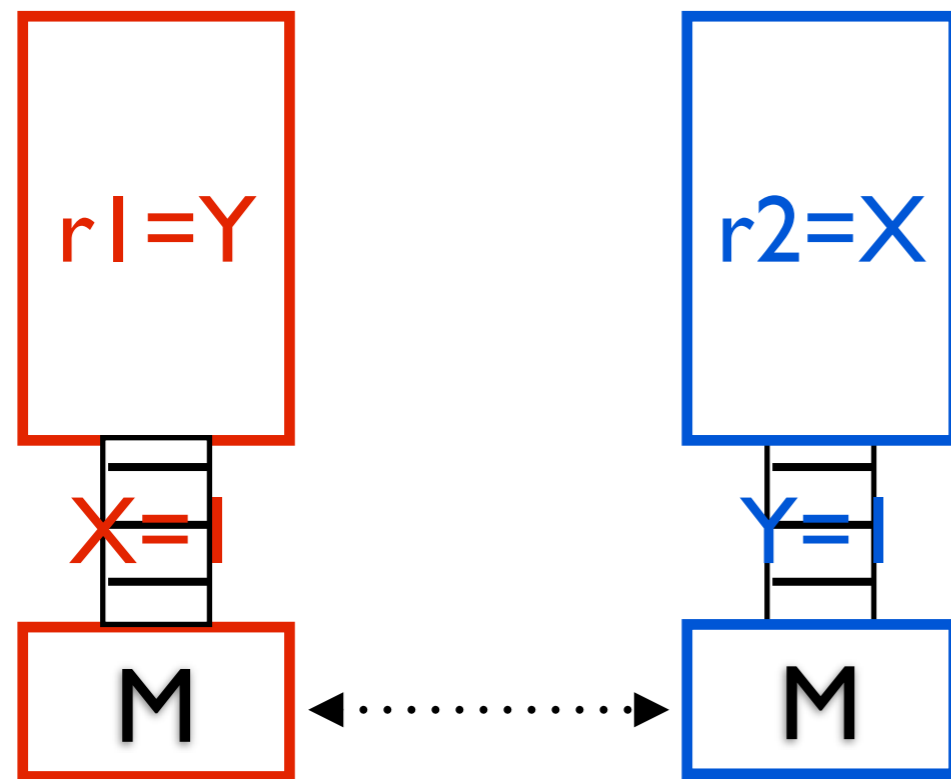


Program
Initially $X == Y == 0$

$r2 = X$

Execution

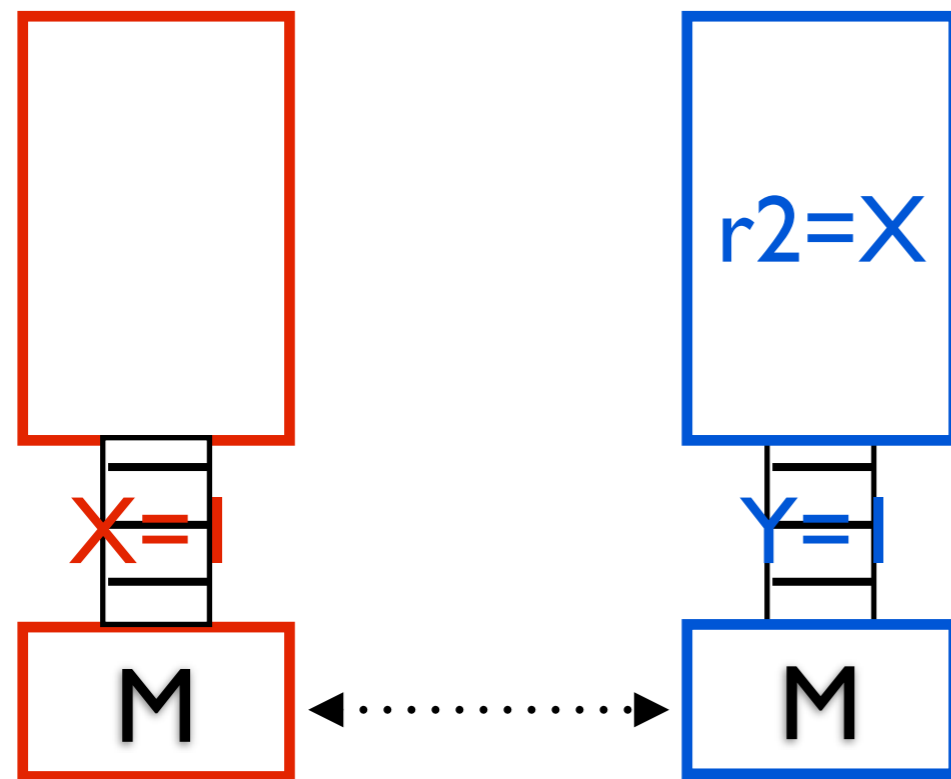
Reordering #1: Write Buffers



Program
Initially $X == Y == 0$

Execution

Reordering #1: Write Buffers



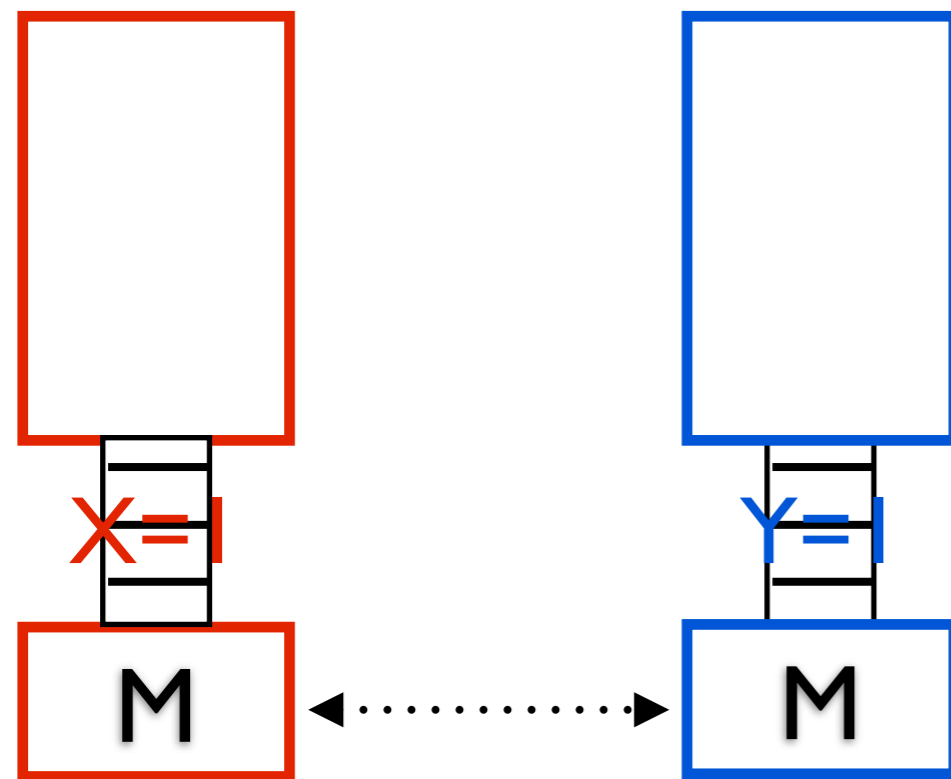
Program

Initially $X == Y == 0$

Execution

$r1 = Y$ [$r1 \leftarrow 0$]

Reordering #1: Write Buffers



Program

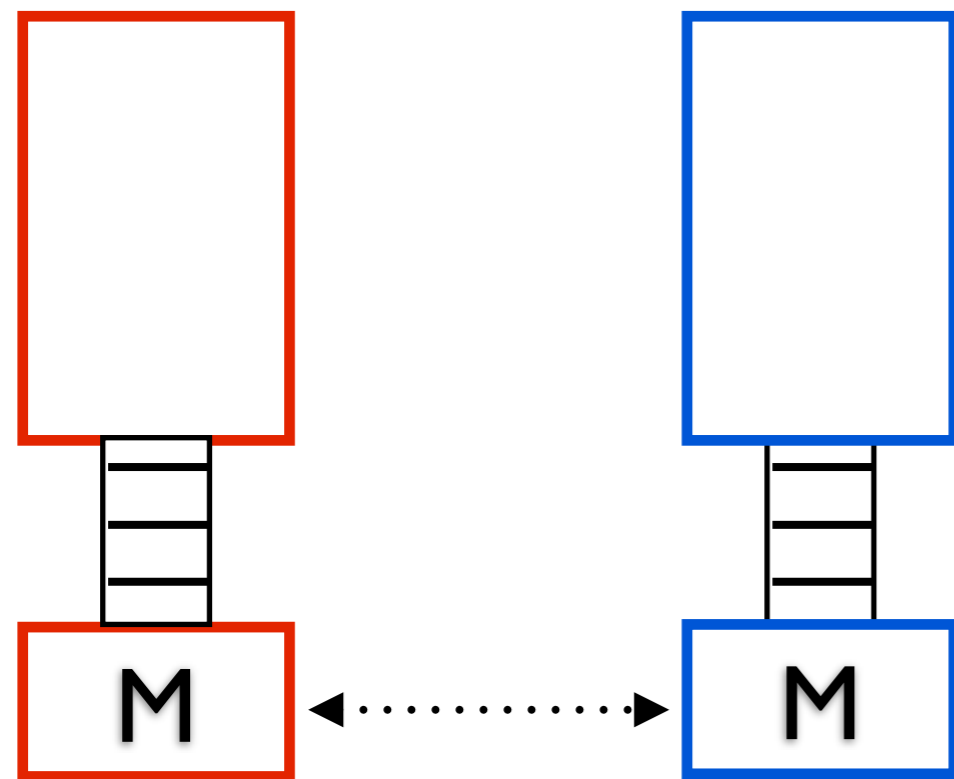
Initially $X == Y == 0$

Execution

$r1 = Y$ [$r1 \leftarrow 0$]

$r2 = X$ [$r2 \leftarrow 0$]

Reordering #1: Write Buffers



WBs let reads finish
before older writes

Program

Initially $X == Y == 0$

Execution

$r1 = Y$ [$r1 \leftarrow 0$]

$r2 = X$ [$r2 \leftarrow 0$]

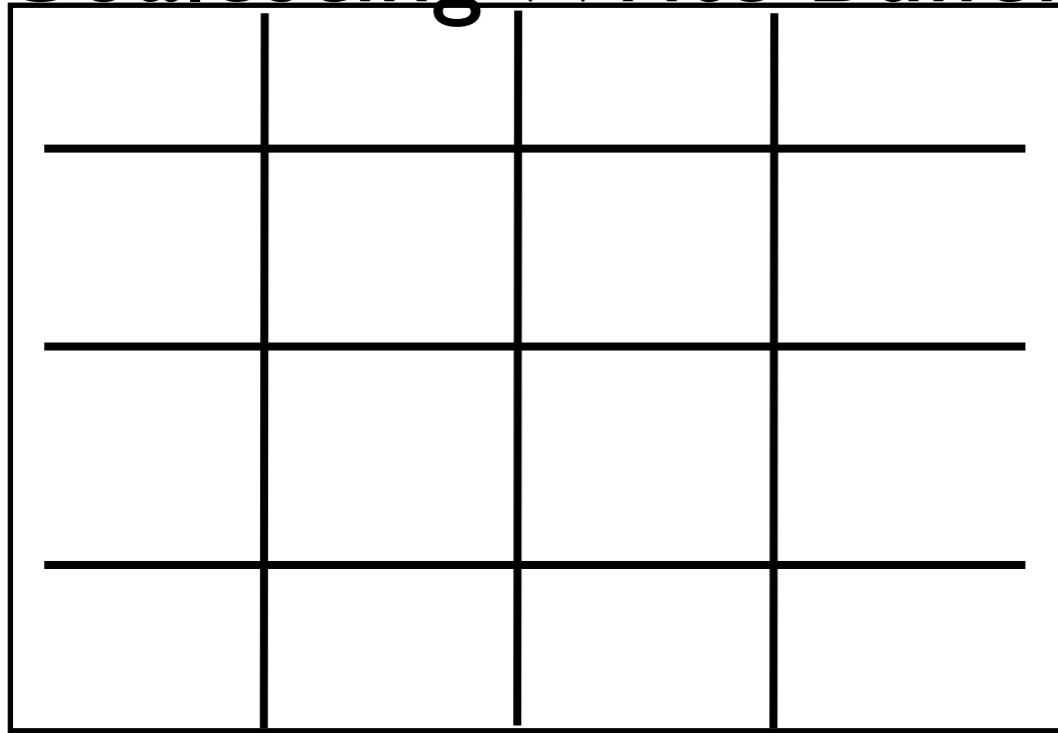
$X = 1$

$Y = 1$

(Not SC!)

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			
	Y=1		

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			
			Y=1
	Z=1		

Program

X,Z in same \$ line

X=1

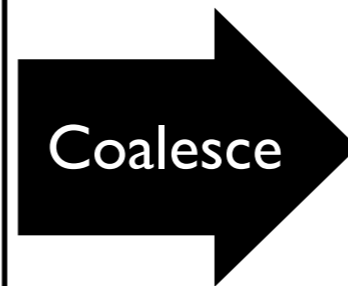
Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			
			Y=1
	Z=1		



Coalescing Write Buffer

X=1	Z=1		
			Y=1

Combining the write to X & Z saves bandwidth,
but **reorders** Z=1 and Y=1

Reordering #3: Compilers

```
X = 0  
for (i .. 100)  
  X = 1  
  print x
```

X = 0

Compiler

```
X = 1  
for (i .. 100)  
  print x
```

X = 0

Been hoisted!

The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0..”)

When is Reordering a Problem?

When is Reordering a Problem?

When Executions Aren't SC

When is an Execution Not SC?

When a memory operation happens before itself

Execution

$r1=Y$ [$r1 \leftarrow 0$]

$r2=X$ [$r2 \leftarrow 0$]

$X=1$

$Y=1$

Happens-Before Graph

$X=1$

$Y=1$

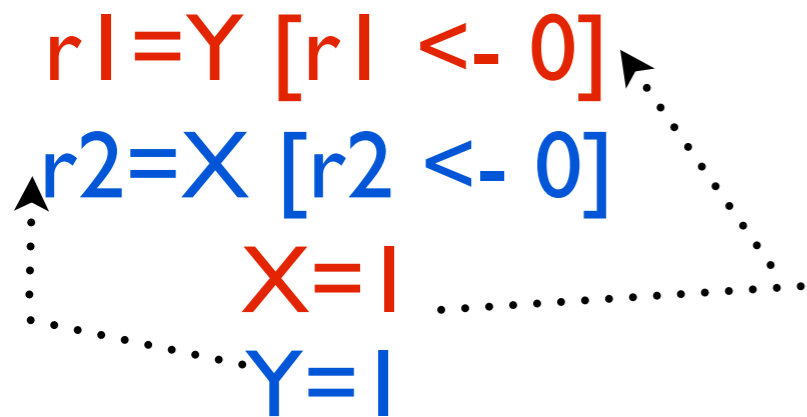
$r1=Y$

$r2=X$

When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph

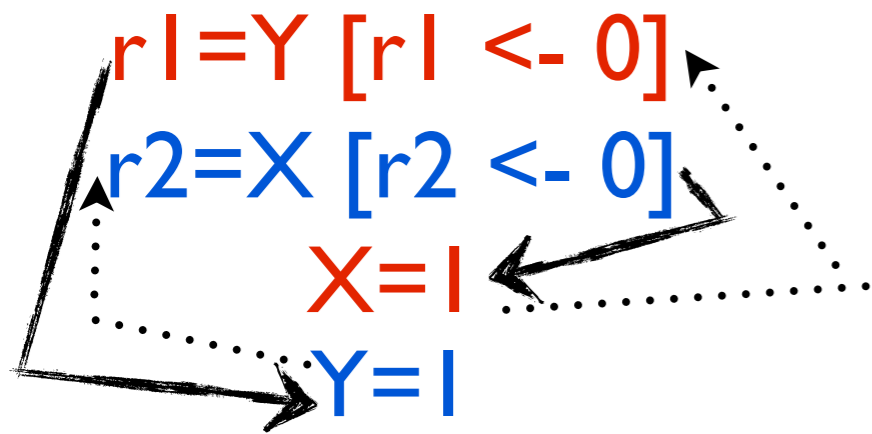


⋮ Program Order HB Edge

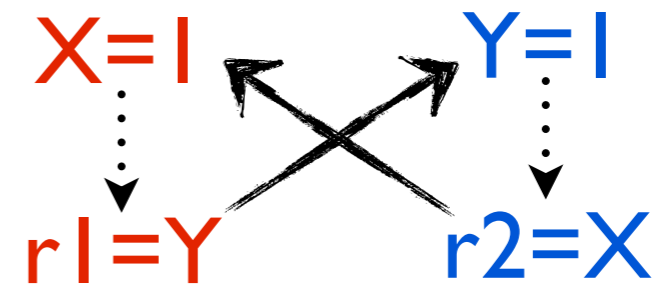
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph



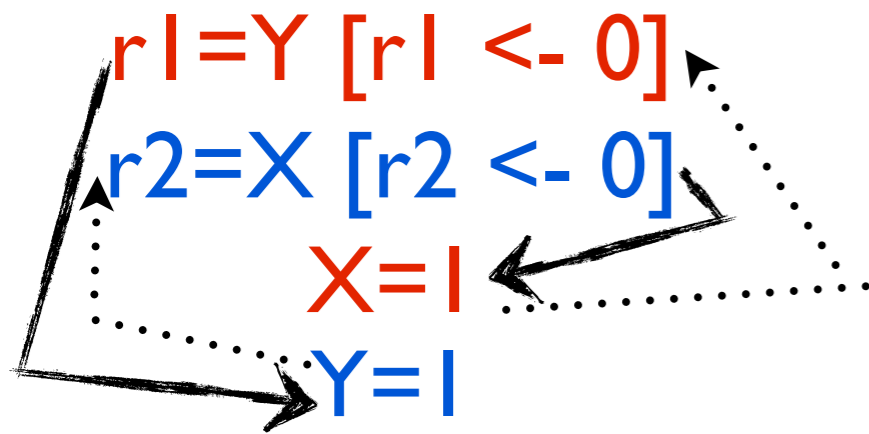
⋮ Program Order HB Edge

↓ Causal Order HB Edge

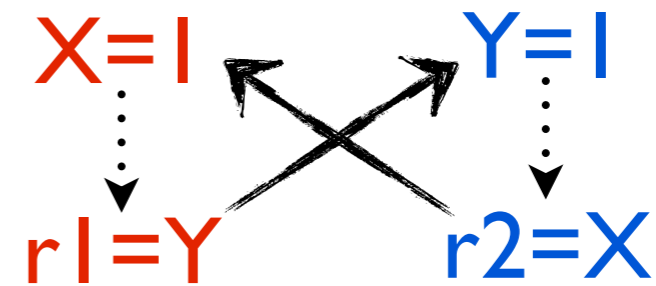
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph



If there is a cycle in the happens-before graph, the execution is not SC

So... are Computers Wrong?!

SC is how **programmers** think.

SC prohibits **all** reordering of instructions

WBs let reads finish before older writes

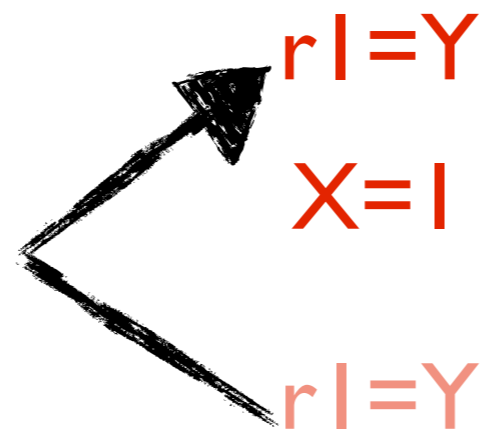
Combining writes saves bandwidth but reorders writes

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

“The Write Buffer Memory Model”

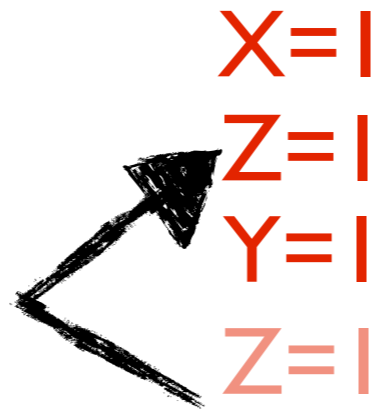


Relaxes W->R
order

Total Store Order - loads may complete before older stores to different locations complete.

PSO_(SPARC)

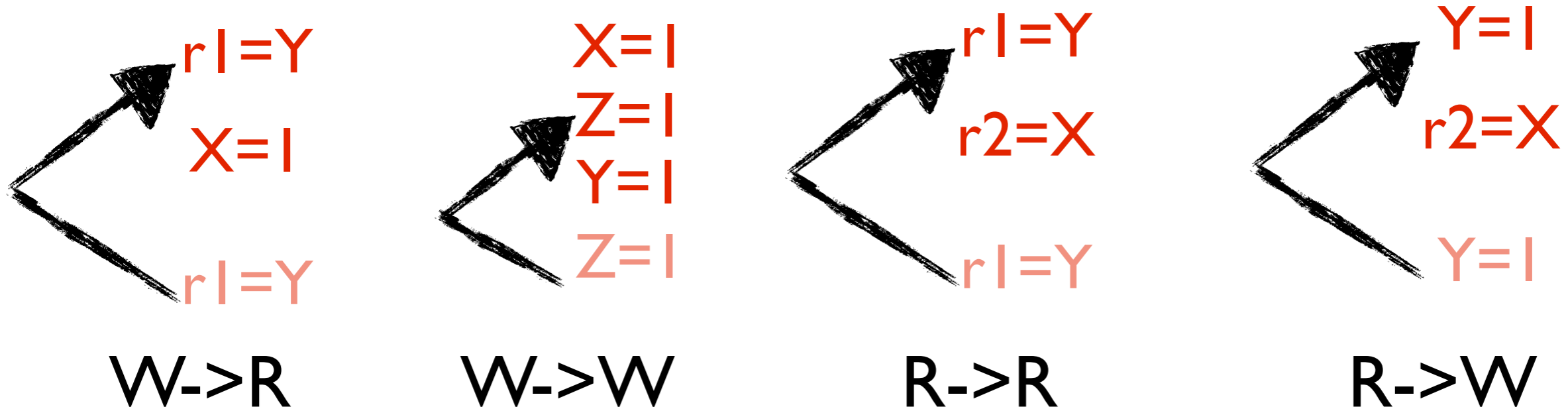
“The Write Combining Memory Model”



Relaxes W->W
order

Partial Store Order - loads and stores may complete before older stores to different locations complete.

In General



Starting with PSO and relaxing R->R and R->W yields
Weak Ordering or Release Consistency (alpha)

Depending on the implementation

SC and Relaxed Consistency

SC is required for correctness and programmer sanity

+

Reordering is required* for performance

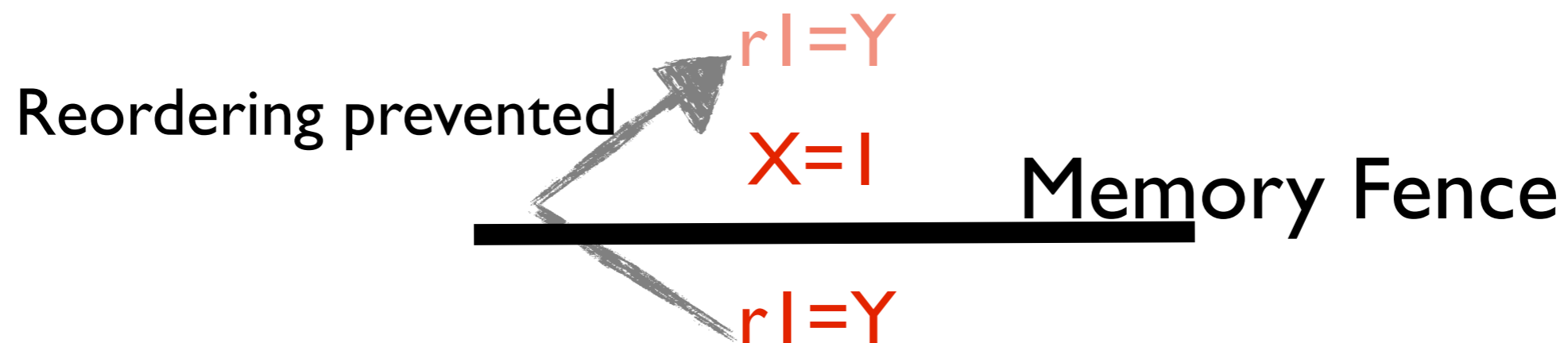
Goal: Ensure SC executions while permitting
Relaxed Consistency reorderings

*Usually; the MIPS memory model is **SC** (surprising!)

How to ensure SC, but permit reordering?

Synchronization Prevents Reordering

Memory fences are another type of synchronization

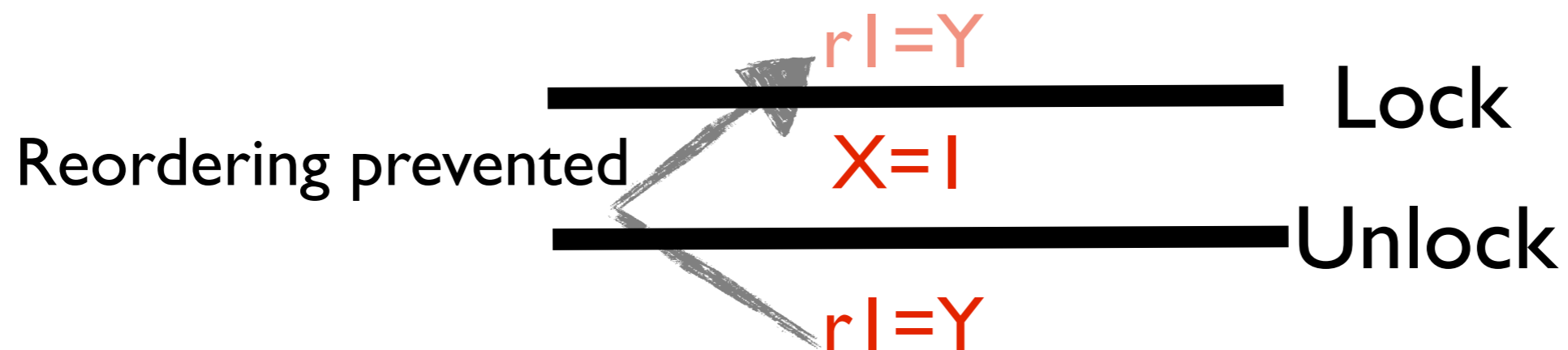


Fence implementation depends on reordering implementation

TSO: Stall reads until write buffer is empty

Synchronization For Real Programmers

Memory fences are wrapped up in locks, etc.

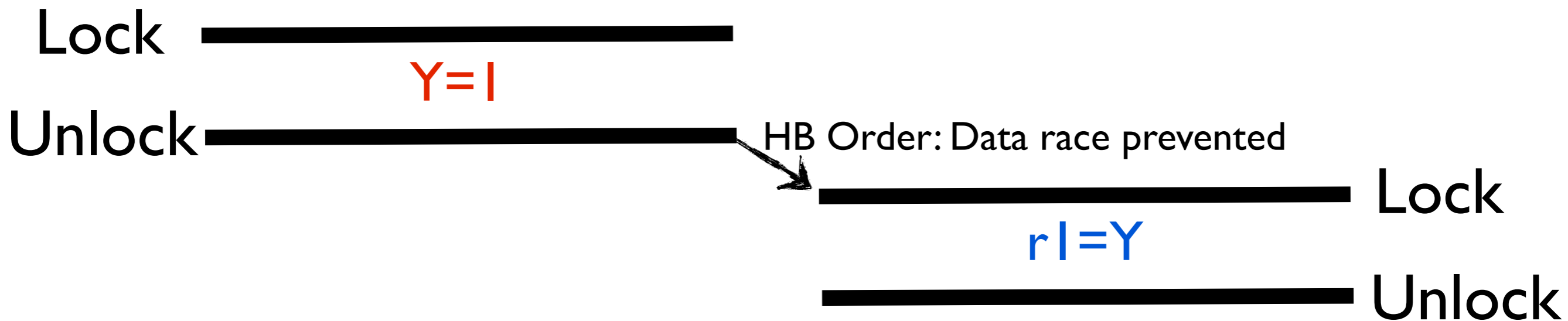


Direct use of fences possible, but inadvisable.

USE A SYNCHRONIZATION LIBRARY

Data Races

Synchronization imposes happens-before on otherwise unordered operations



Data Race: Unordered operations to the same memory location, at least one a write

Memory Models across the System Stack

Language

Java/C++: SC
for data-race-
free programs

Compiler

Conservative
with reordering
when d-r-f can't
be proved

Architecture

Usually very weak for
max optimization
(lots of reordering)

Note: fences from
“above” ensure SC