

Course Description

Lecture 19: Consistency and Coherency

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia



“The **memory consistency model** of a shared-memory system specifies the **order in which memory operations will appear to execute to the programmer**. The memory consistency model affects the process of writing parallel programs and forms an integral part of the entire system, including the architecture, the compiler, and the programming language.”

Excerpt from “Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems”
Sarita Adve, et al, 1999

Memory Consistency

Memory Consistency Model

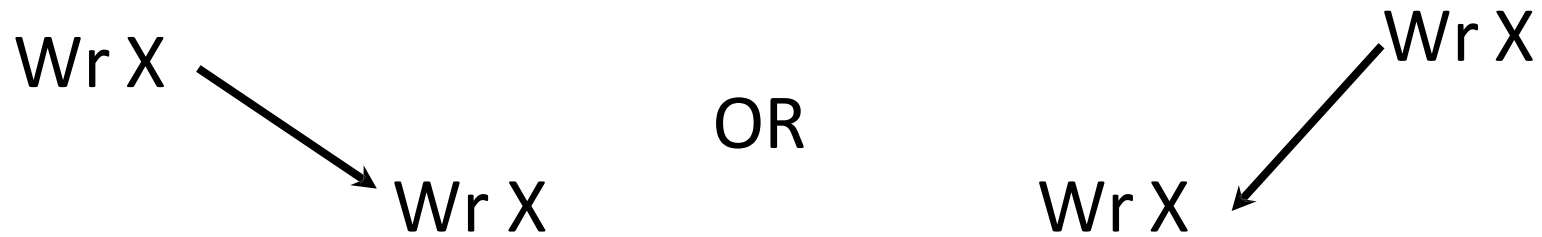
Informal Definition:

“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”

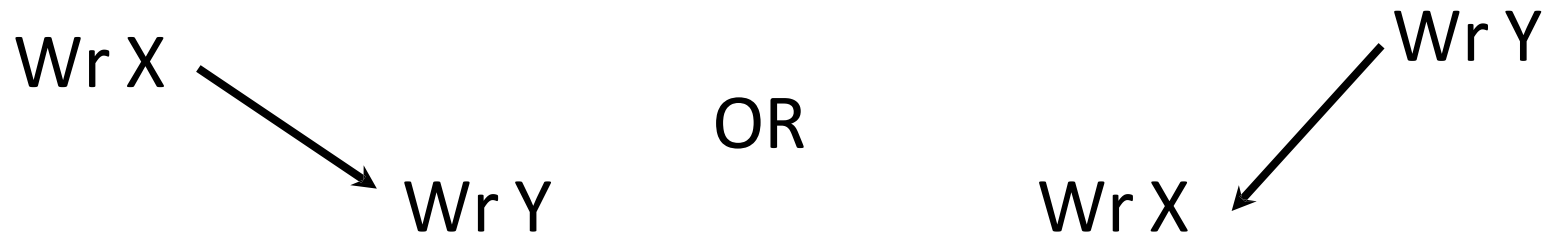
“Defines which reorderings of memory operations are permitted”

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:

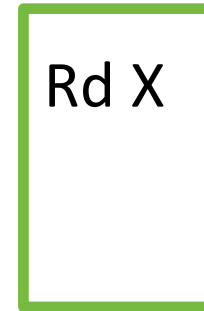
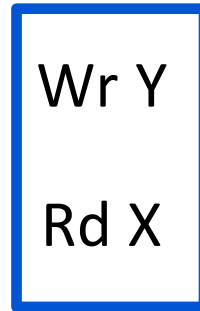
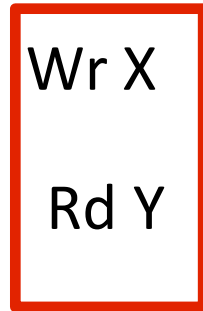
Invariant #1:

Instructions are
executed in program
order

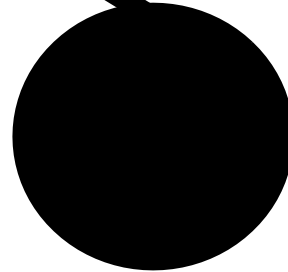
Invariant #2:

All processors agree
on a total order of
executed instructions

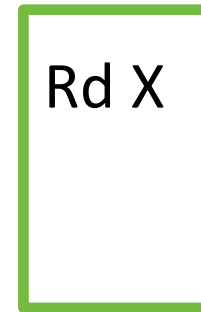
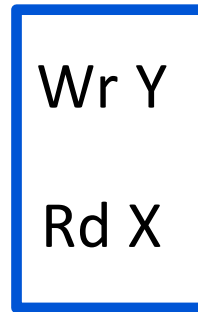
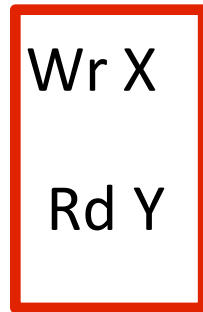
The SC "Switch"



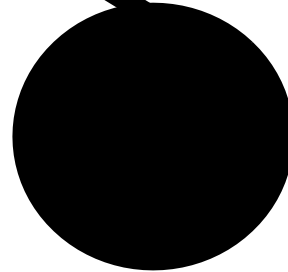
Execution



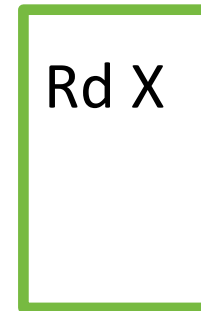
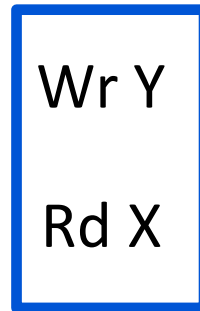
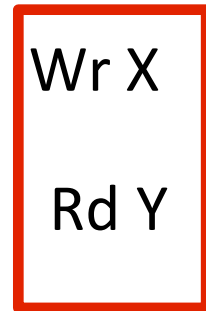
The SC "Switch"



Execution
Wr X

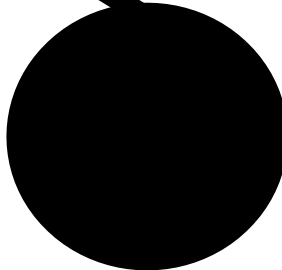


The SC "Switch"

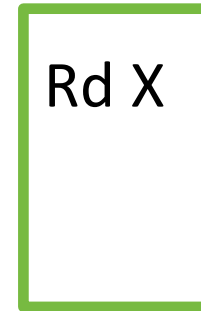
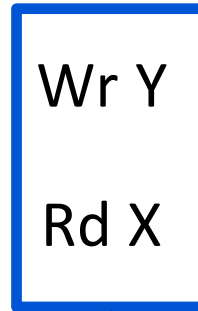
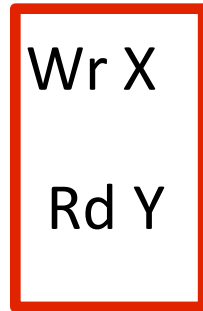


Execution

Wr X
Rd Y

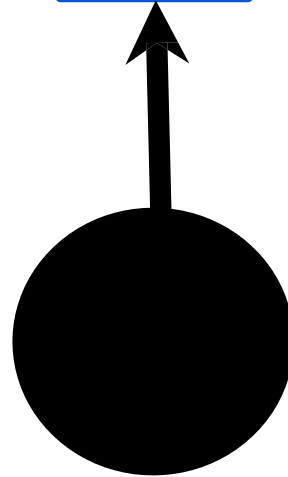


The SC "Switch"

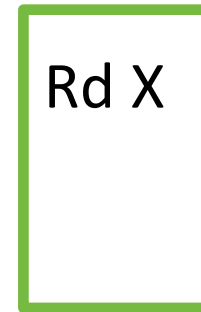
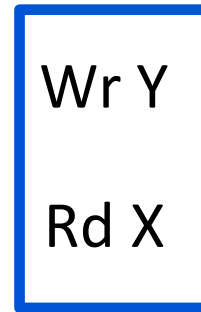
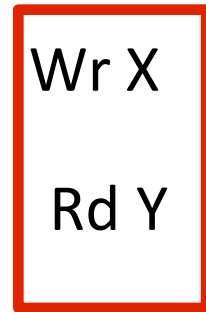


Execution

Wr X
Rd Y
Wr Y

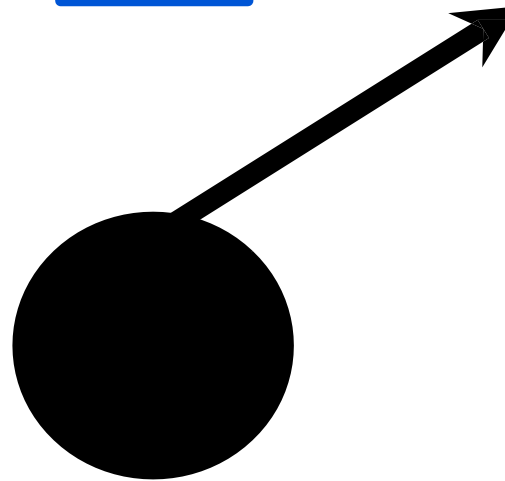


The SC "Switch"

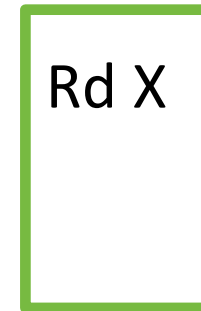
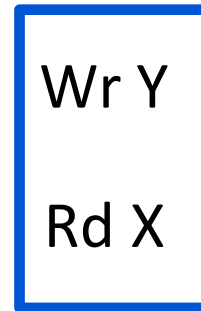
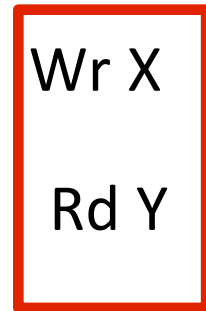


Execution

Wr X
Rd Y
Wr Y
Rd X

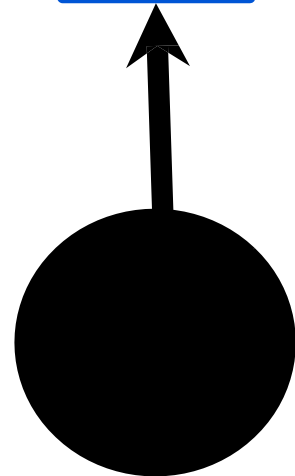


The SC "Switch"



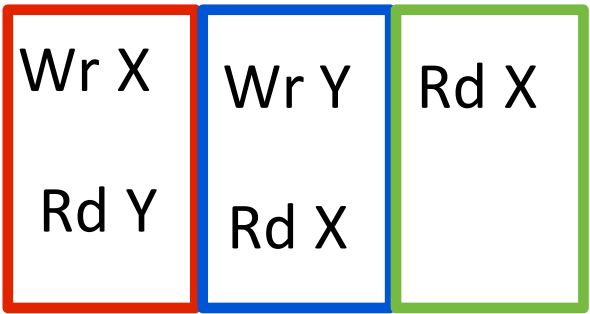
Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X



Why is SC Important?

SC is the most complex model that we can ask **programmers** to think about.

	<u>Intuitive (SC)</u>	<u>Weird (not SC)</u>
	Wr X	Rd Y
	Rd Y	Wr X
	Wr Y	Rd X
	Rd X	Rd X
	Rd X	Wr Y

SC prohibits **all** reordering of instructions (Invariant 1)

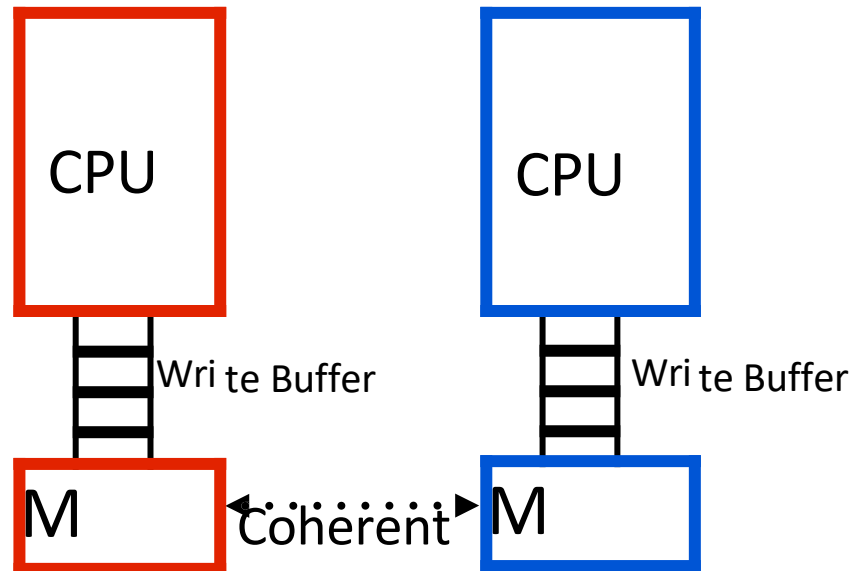
Real hardware does not enforce SC

The ARMv8 Memory Model:

The ARMv8 architecture employs a *weakly-ordered* model of memory. In general terms, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The processor is able to re-order memory read operations with respect to each other. Writes may also be re-ordered (for example, write combining). As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

<https://developer.arm.com/documentation/den0024/a/Memory-Ordering>

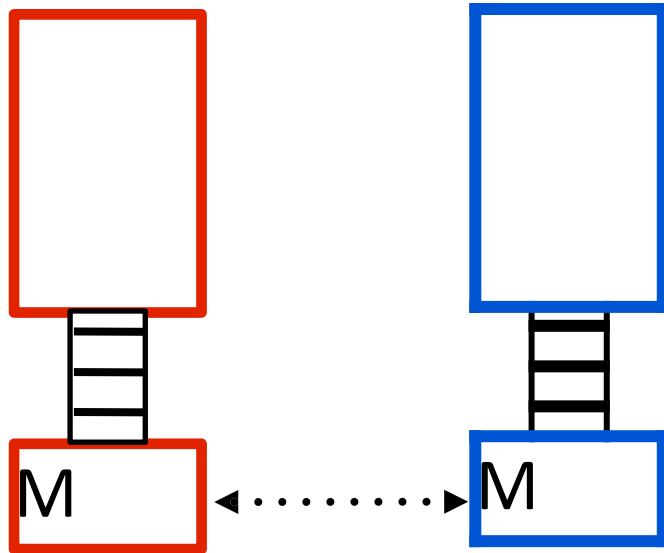
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

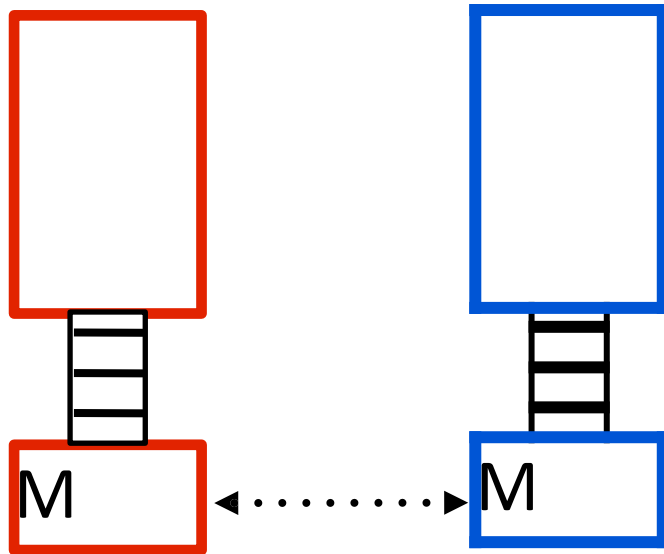
$X=1$ $Y=1$

$r1=Y$ $r2=X$

Is $r1==r2==0$

a valid result?

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

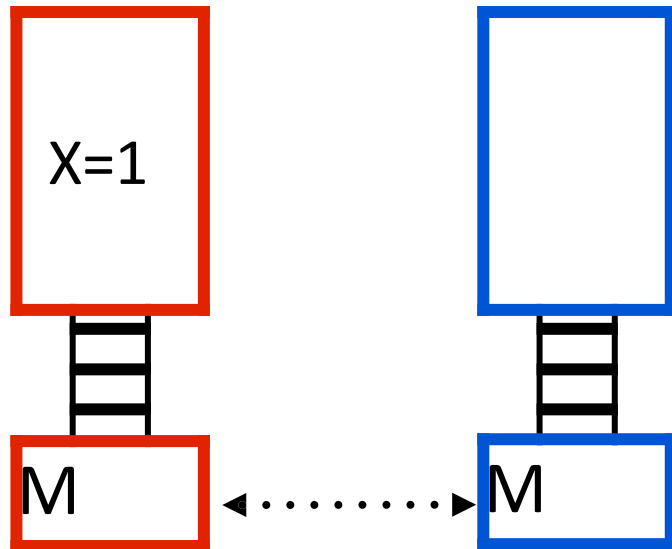
$X=1$ $Y=1$

$r1=Y$ $r2=X$

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 $X == Y == 0$

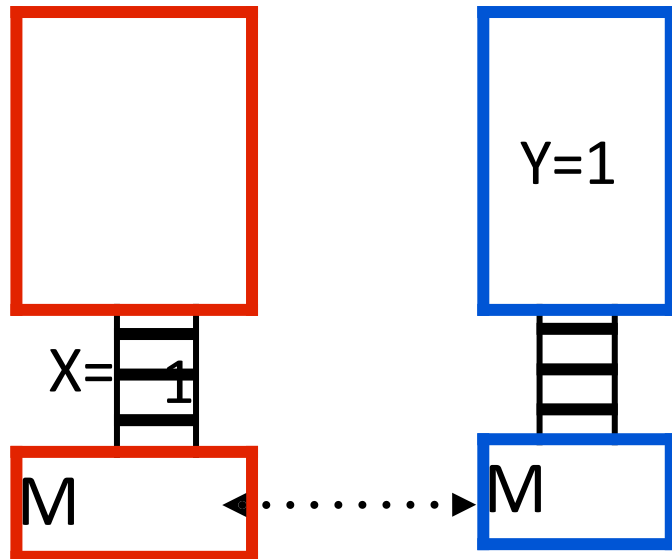
$Y = 1$

$r1 = Y$

$r2 = X$

Execution

Reordering #1: Write Buffers



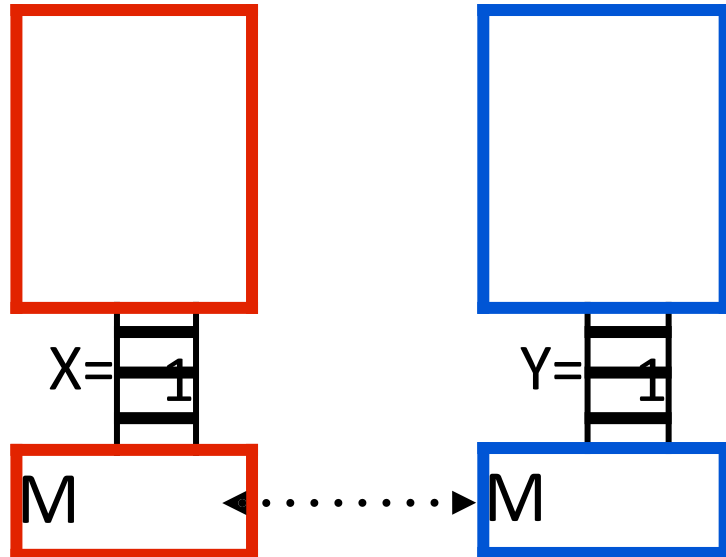
Program

Initially $X == Y == 0$

$r1=Y$ $r2=X$

Execution

Reordering #1: Write Buffers



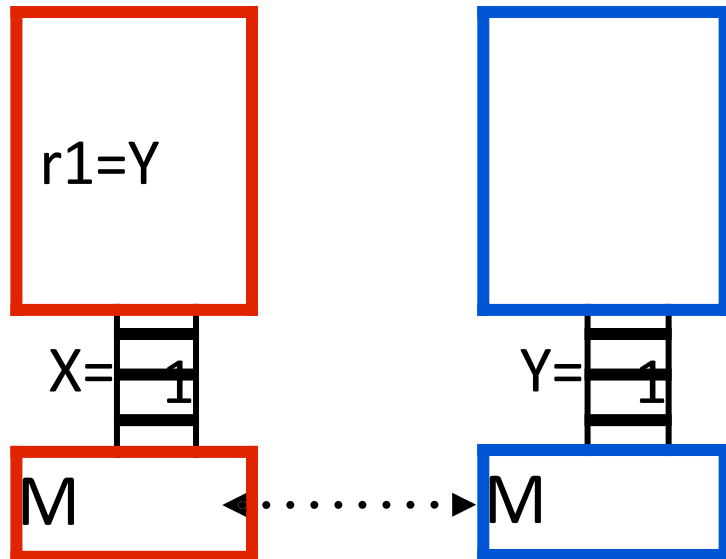
Program

Initially $X == Y == 0$

$r1=Y$ $r2=X$

Execution

Reordering #1: Write Buffers



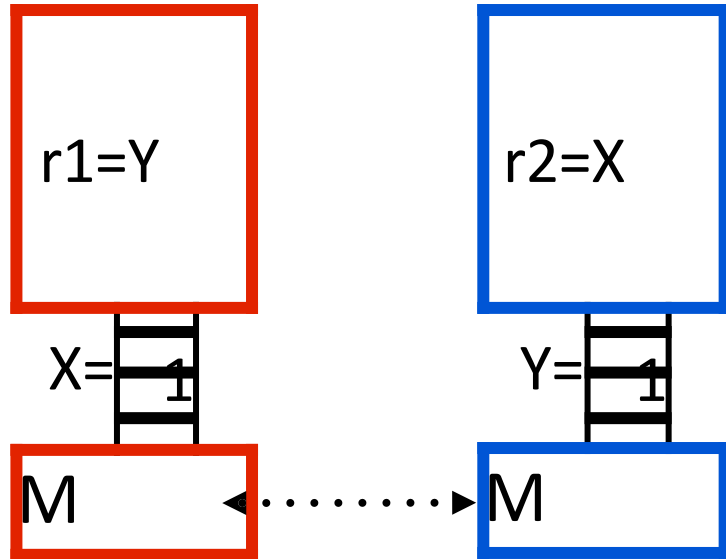
Program

Initially $X == Y == 0$

$r2 = X$

Execution

Reordering #1: Write Buffers

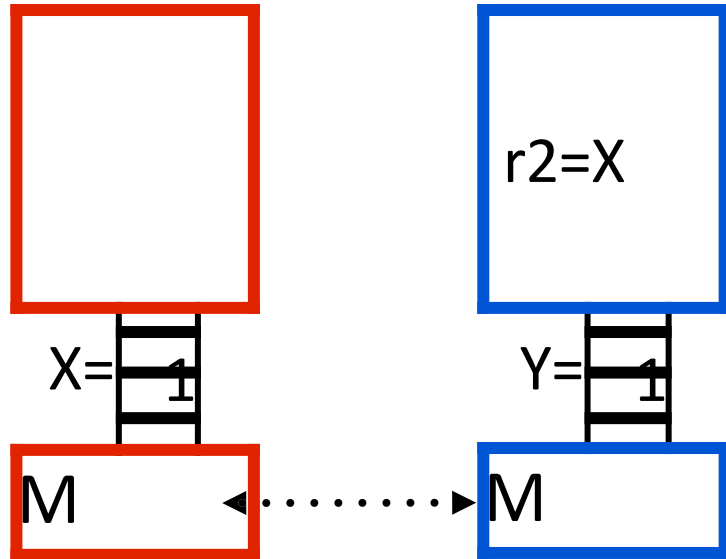


Program

Initially $X == Y == 0$

Execution

Reordering #1: Write Buffers



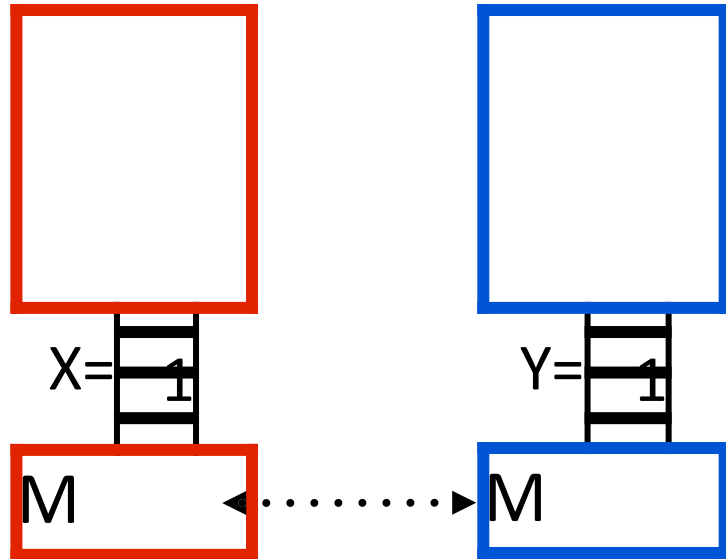
Program

Initially $X == Y == 0$

Execution

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

Reordering #1: Write Buffers



Program

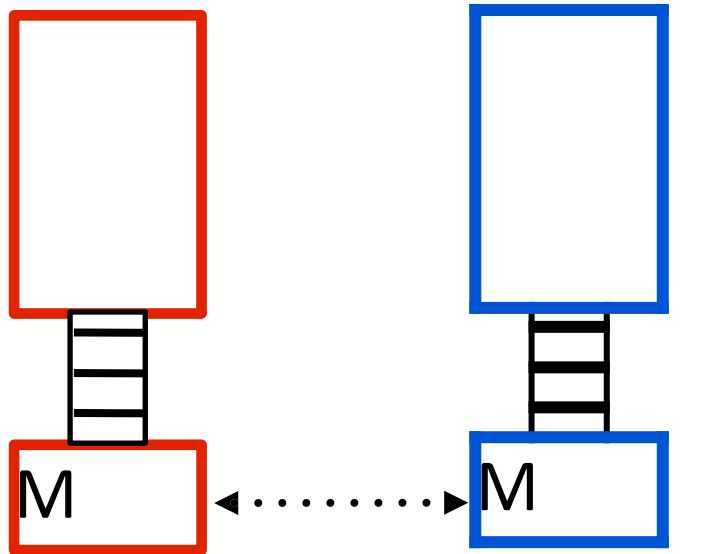
Initially $X == Y == 0$

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

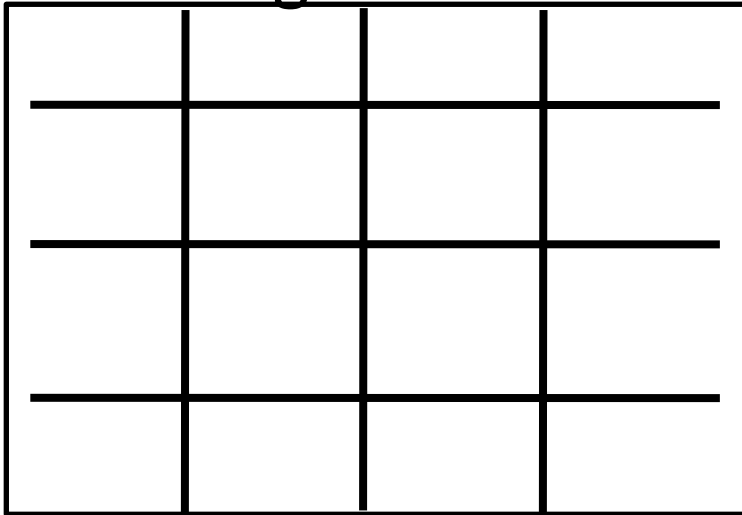
$r1=Y$ [$r1 \leftarrow 0$]

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

$X=1$
 $Y=1$ (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

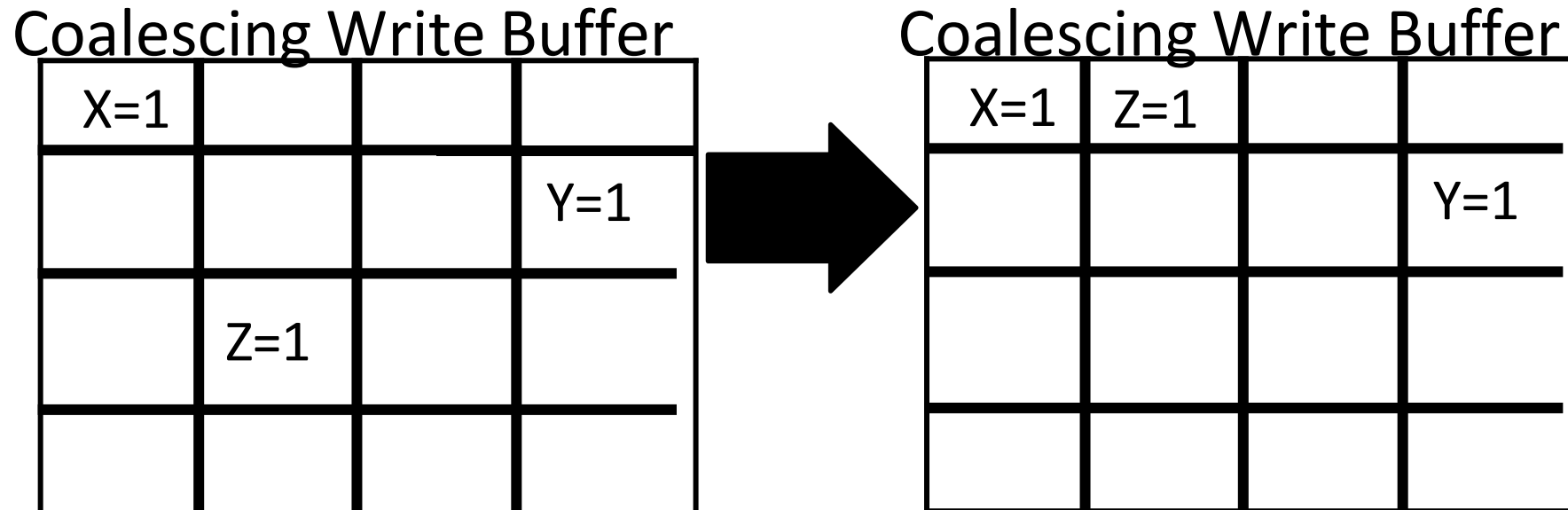
X,Z in same \$ line

X=1

Y=1

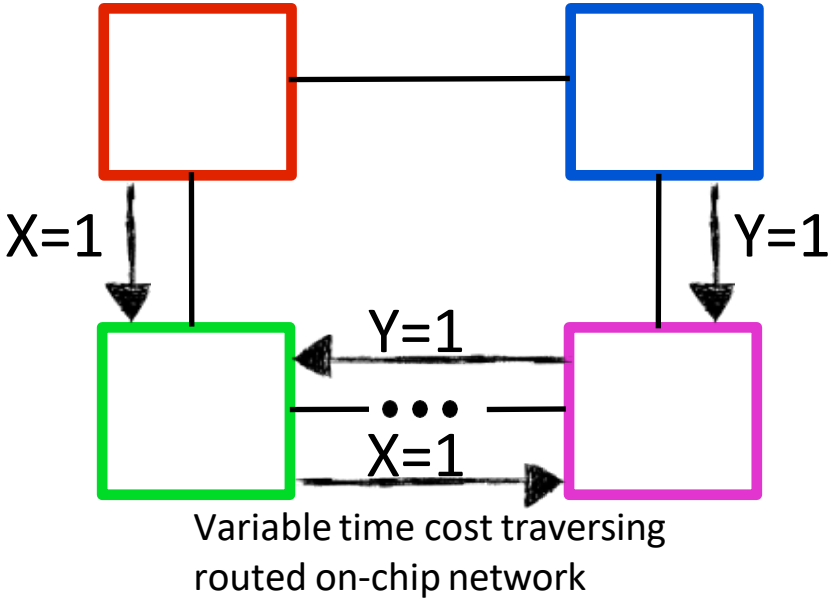
Z=1

Reordering #2: Write Combining



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

Reordering #3: Interconnect



X=1

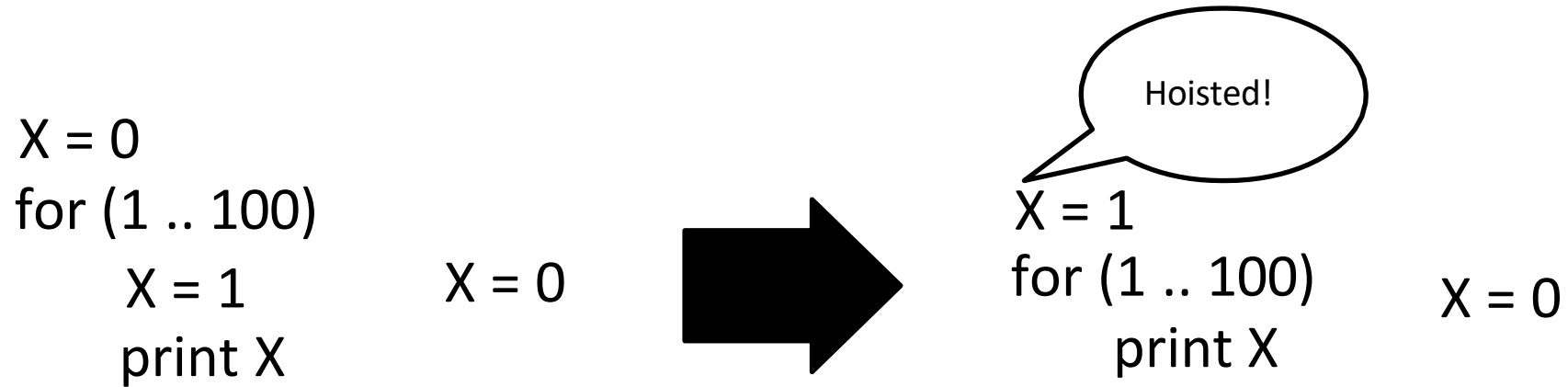
Program

r1=X Y=1 r3=Y
 r2=Y r4=X

Execution

X=1
 Y=1
 r1=X [r1 <- 1]
 r2=Y [r2 <- 0]
 r3=Y [r3 <- 1]
 r4=X [r4 <- 0]

Reordering #4: Compilers



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 an Execution Not SC?

When a memory operation happens before itself

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

Y=1

Happens-Before Graph

X=1

Y=1

r1=Y

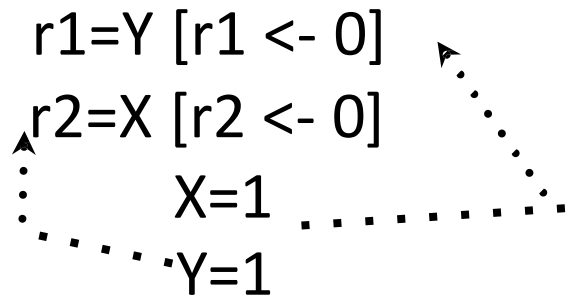
r2=X

When is an Execution Not SC?

When a memory operation *happens before* itself

Execution

r1=Y [r1 <- 0]
r2=X [r2 <- 0]
X=1
Y=1



Happens-Before Graph

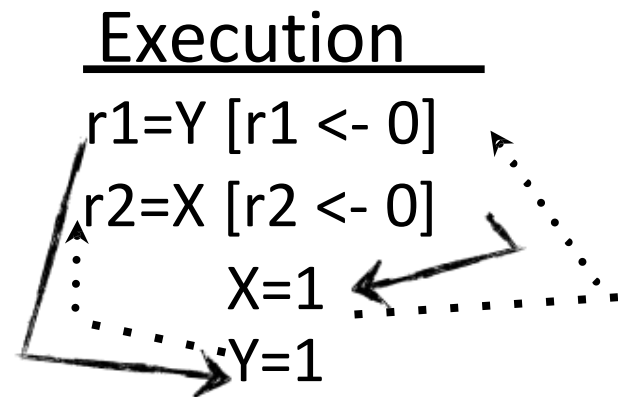
X=1	Y=1
⋮	⋮
↓	↓
r1=Y	r2=X

Program Order HB Edge



When is an Execution Not SC?

When a memory operation happens before itself



Happens-Before Graph

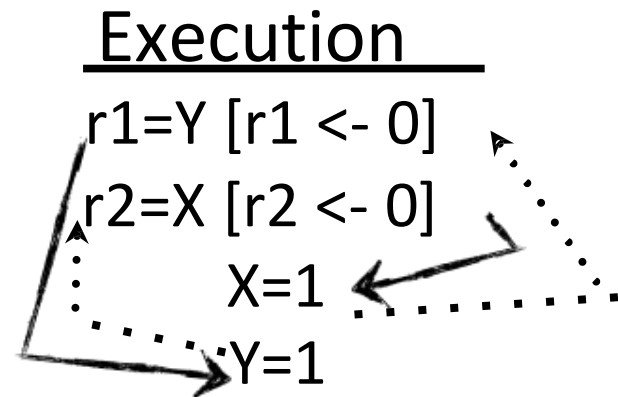


⋮ Program Order HB Edge

↓ Causal Order HB Edge

When is an Execution Not SC?

When a memory operation happens before itself



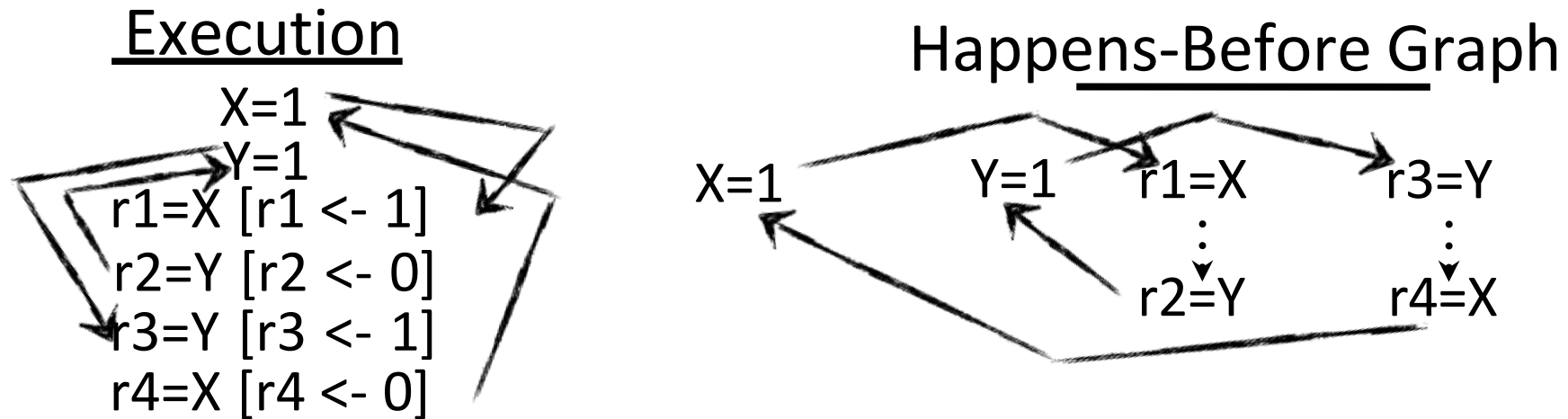
Happens-Before Graph



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

When is an Execution Not SC?

When a memory operation happens before itself



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

Two Design Constraints at Odds

SC is how **programmers** think, but restricts all reordering

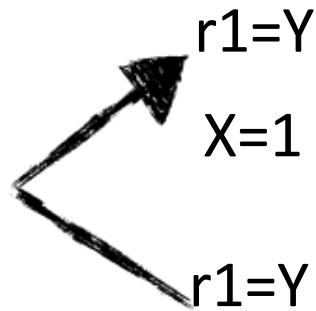
Reordering allows **optimization**, but leads to unintuitive non-SC behavior.

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

“The Write Buffer Memory Model”

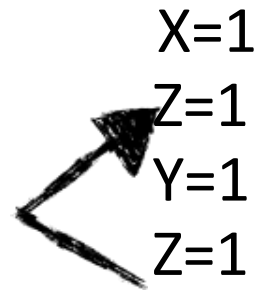


Relaxes W->R
order

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

PSO_(SPARC)

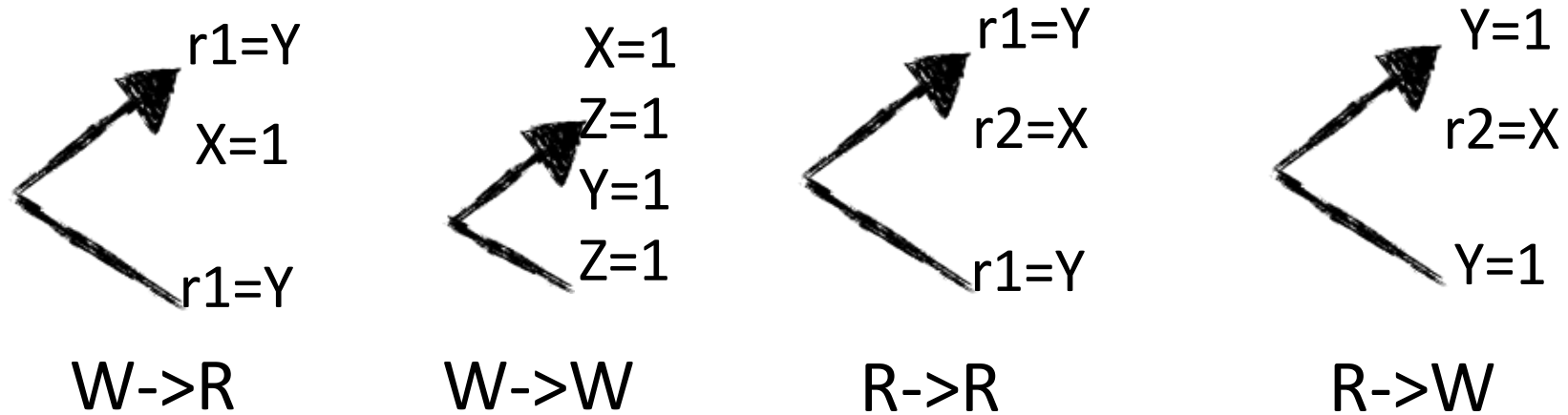
“The Write Combining Memory Model”



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

Implementing Synchronization for Weak Memory Models

- What does synchronization have to do to prevent SC violations?
 - Flush WB, prevent coalescing/bypassing, impose ordering in network, prevent compiler reorderings
- What does synchronization have to do to prevent other kinds of problems?
 - Enforce mutually exclusive execution by different threads of critical region, force threads to wait at barriers, enforce wait/notify discipline

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; MIPS memory model is **SC**

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

What did we just learn?

- Coherence and consistency are both memory ordering principles
- Understanding the memory model is compelling to understanding the execution and correctness of the program