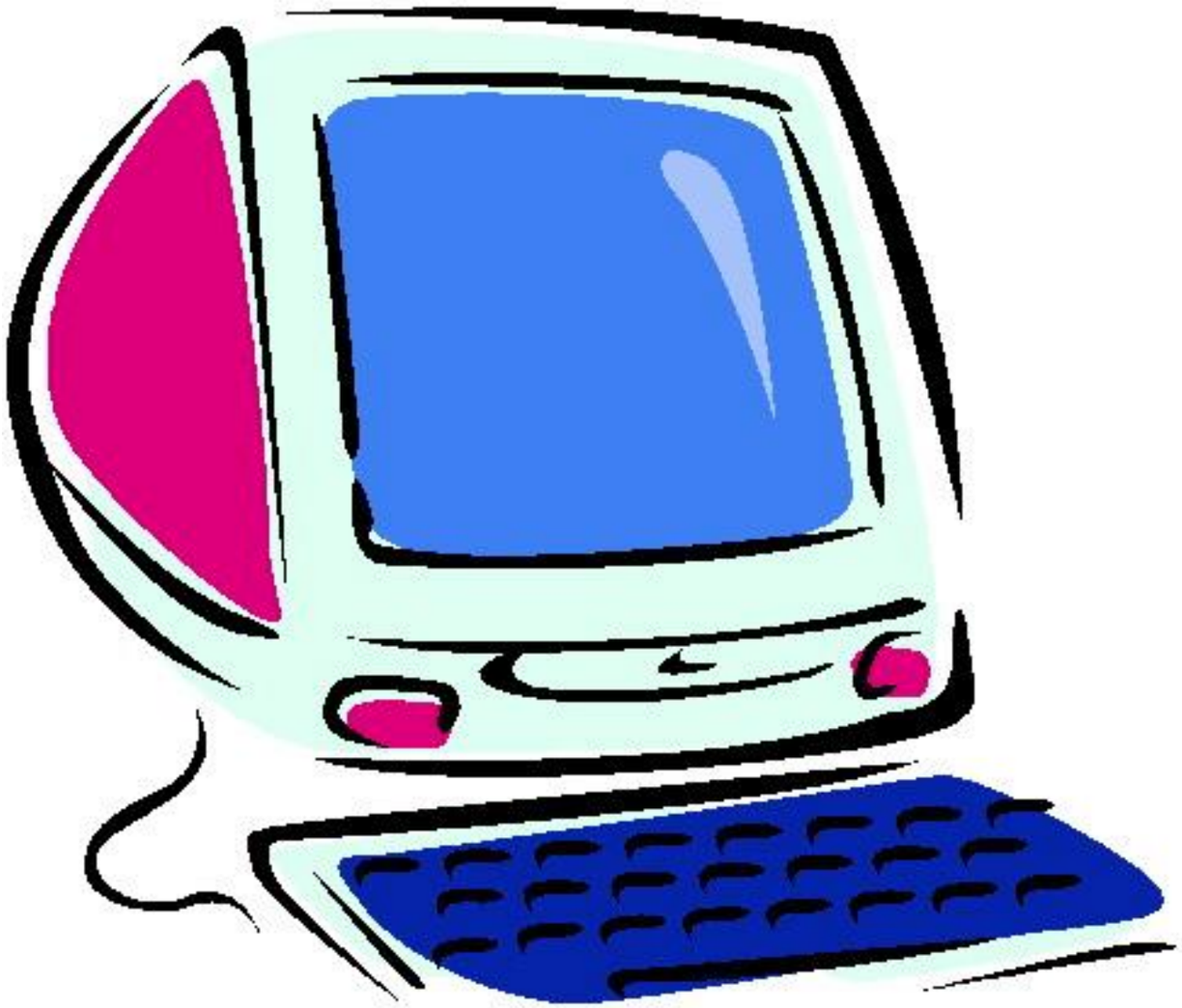
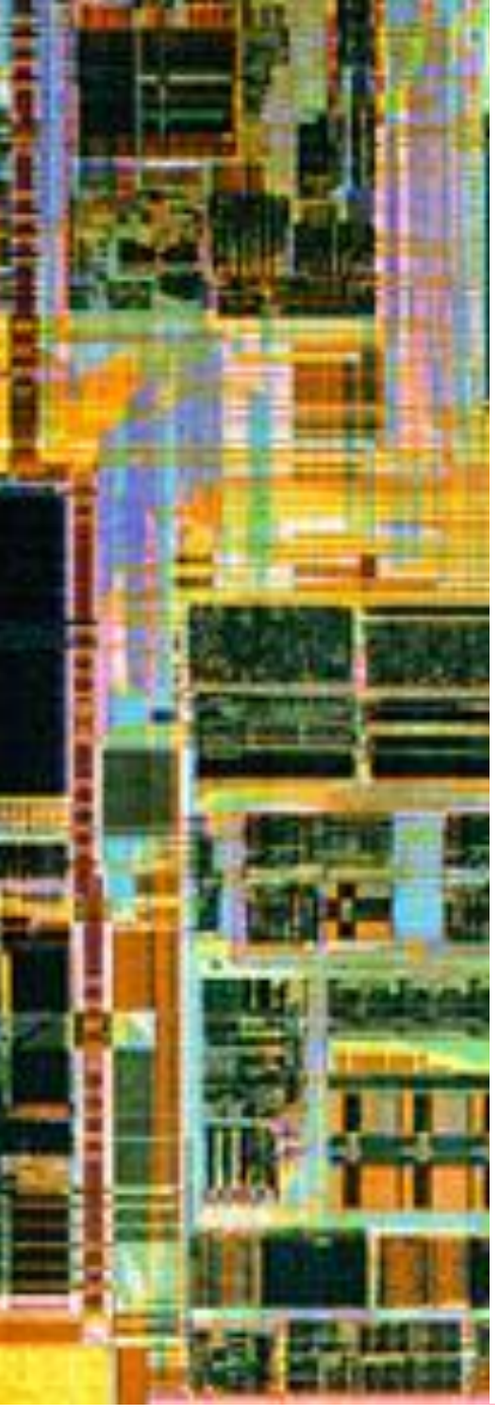
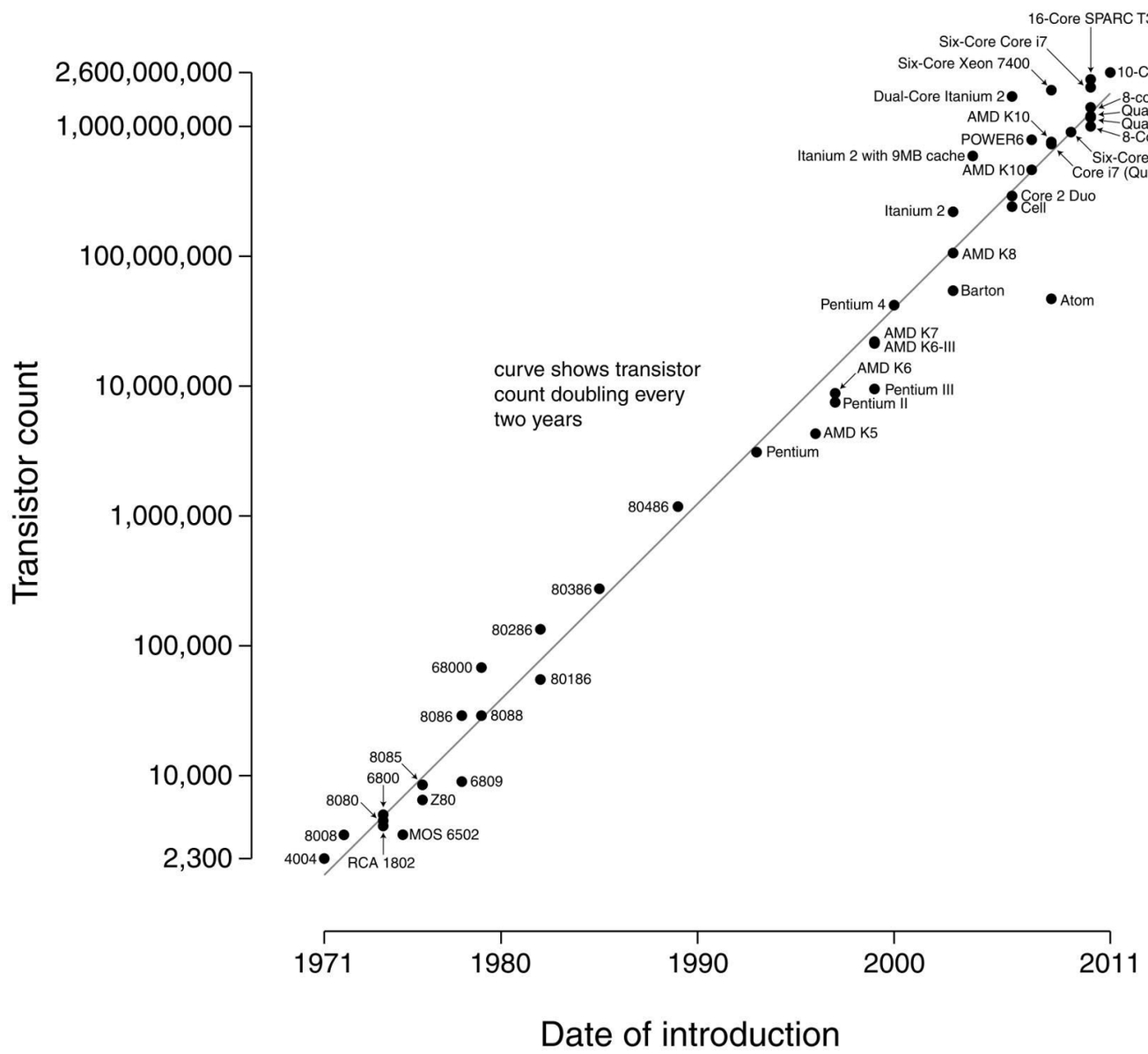
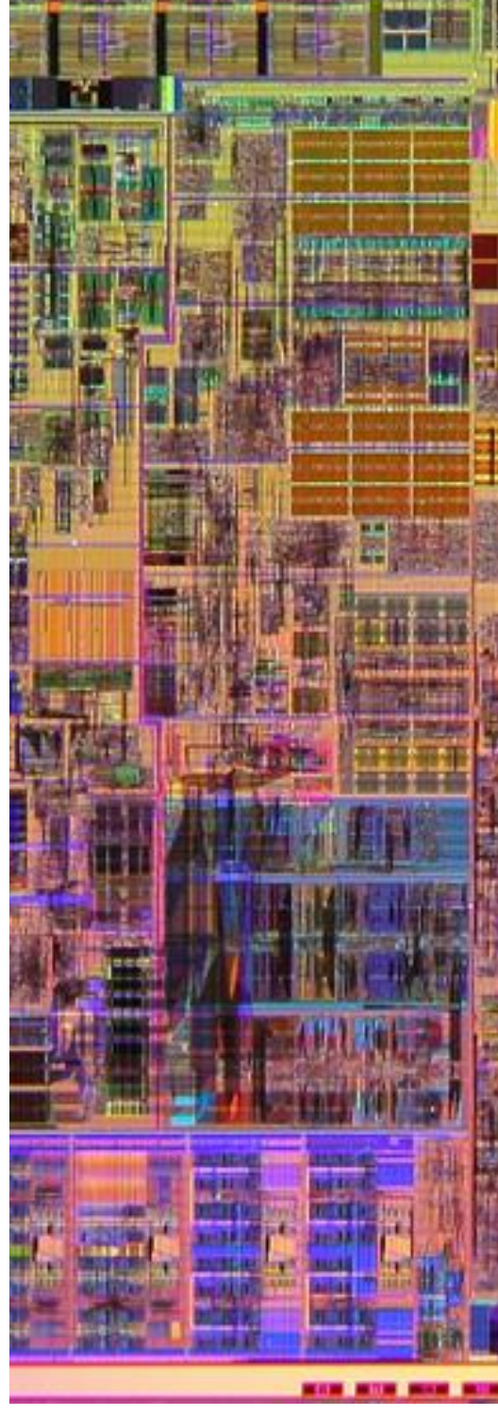


Multicore Computer Architecture Issues: From hardware to software

18-613, Spring 2019

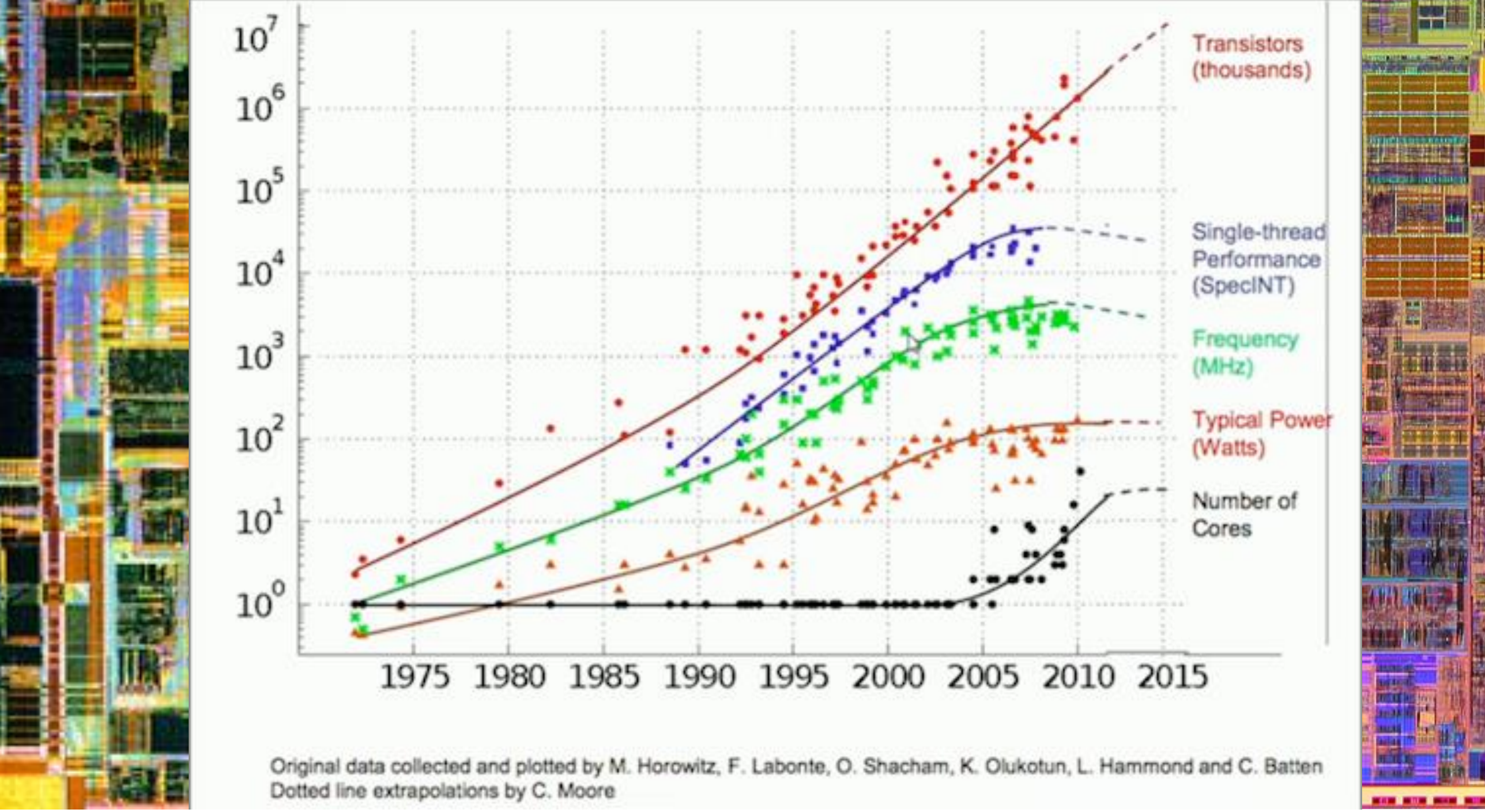






Technology Improves

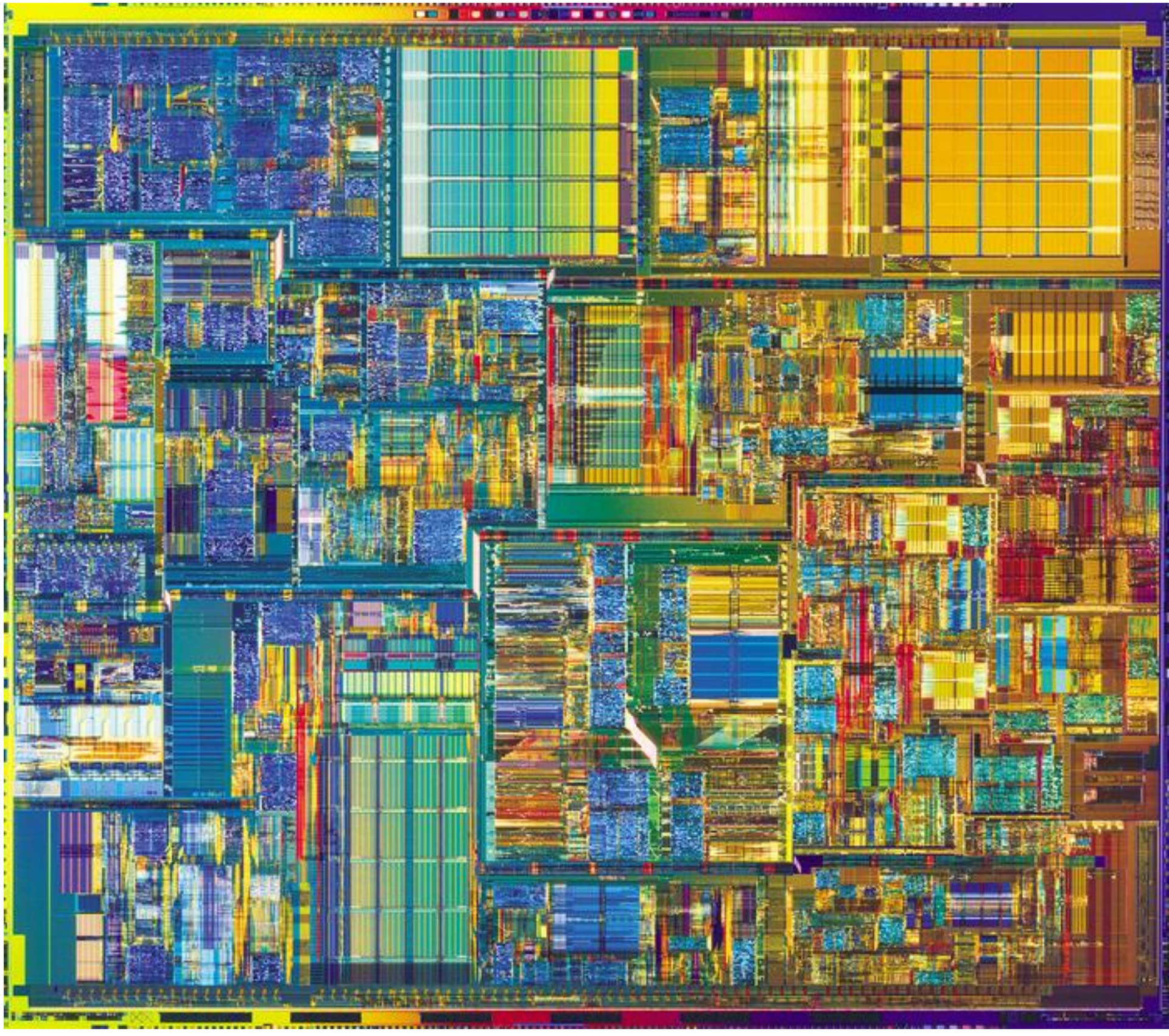
Transistors get smaller, clock speeds go up, power stays roughly constant. Party!



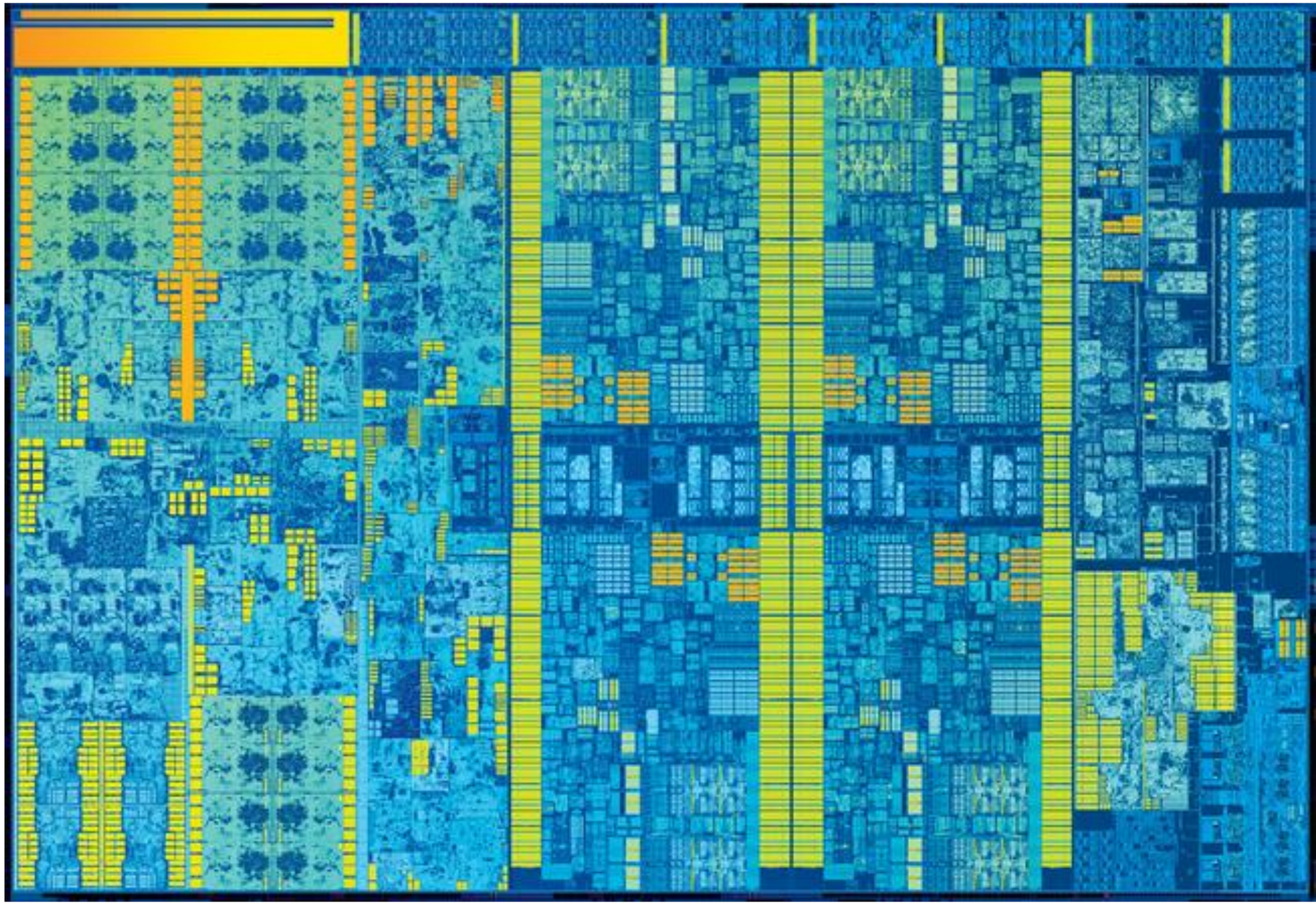
Performance Scaling Hit a Wall!

Transistors still doubling, performance tapers off.
Architects need to be creative!

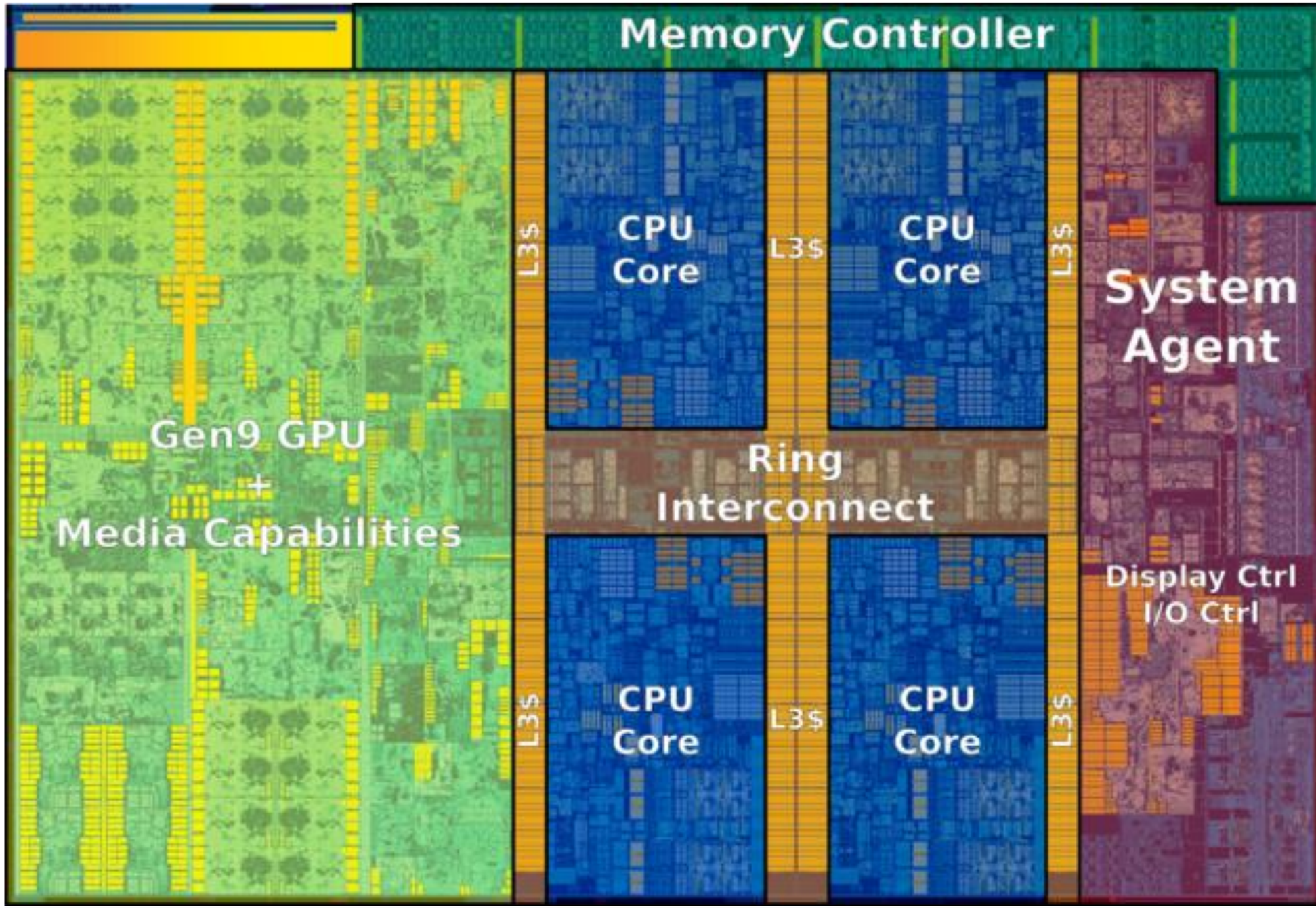
- Willamette core
- 180 nm process
- 217 mm² die size
- 42,000,000 transistors



- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs

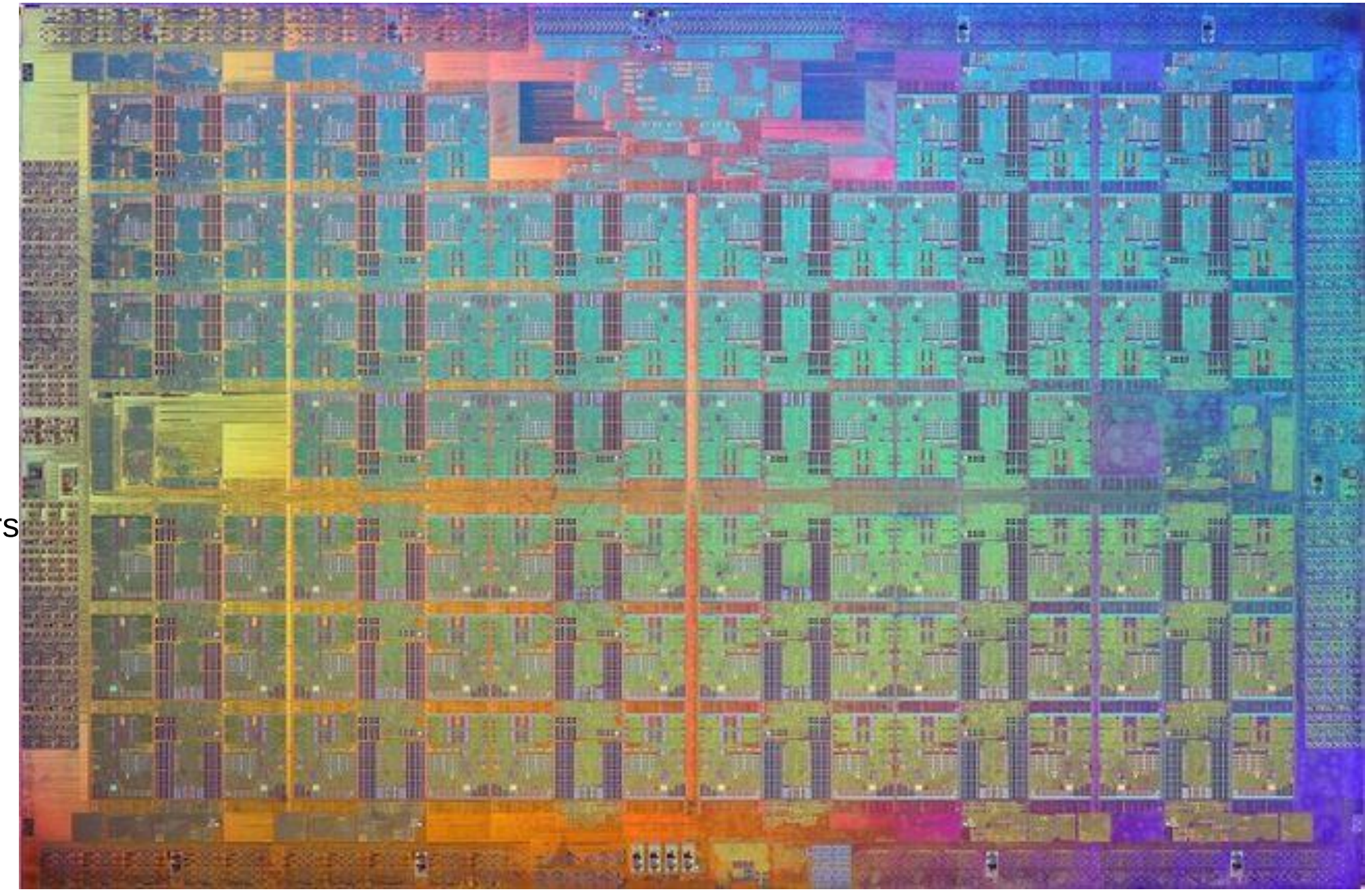


•14 nm process
•11 metal layers
•~1,750,000,000 transistors
•~9.19 mm x ~11.08 mm
•~101.83 mm² die size
•4 CPU cores + 24 GPU EUs



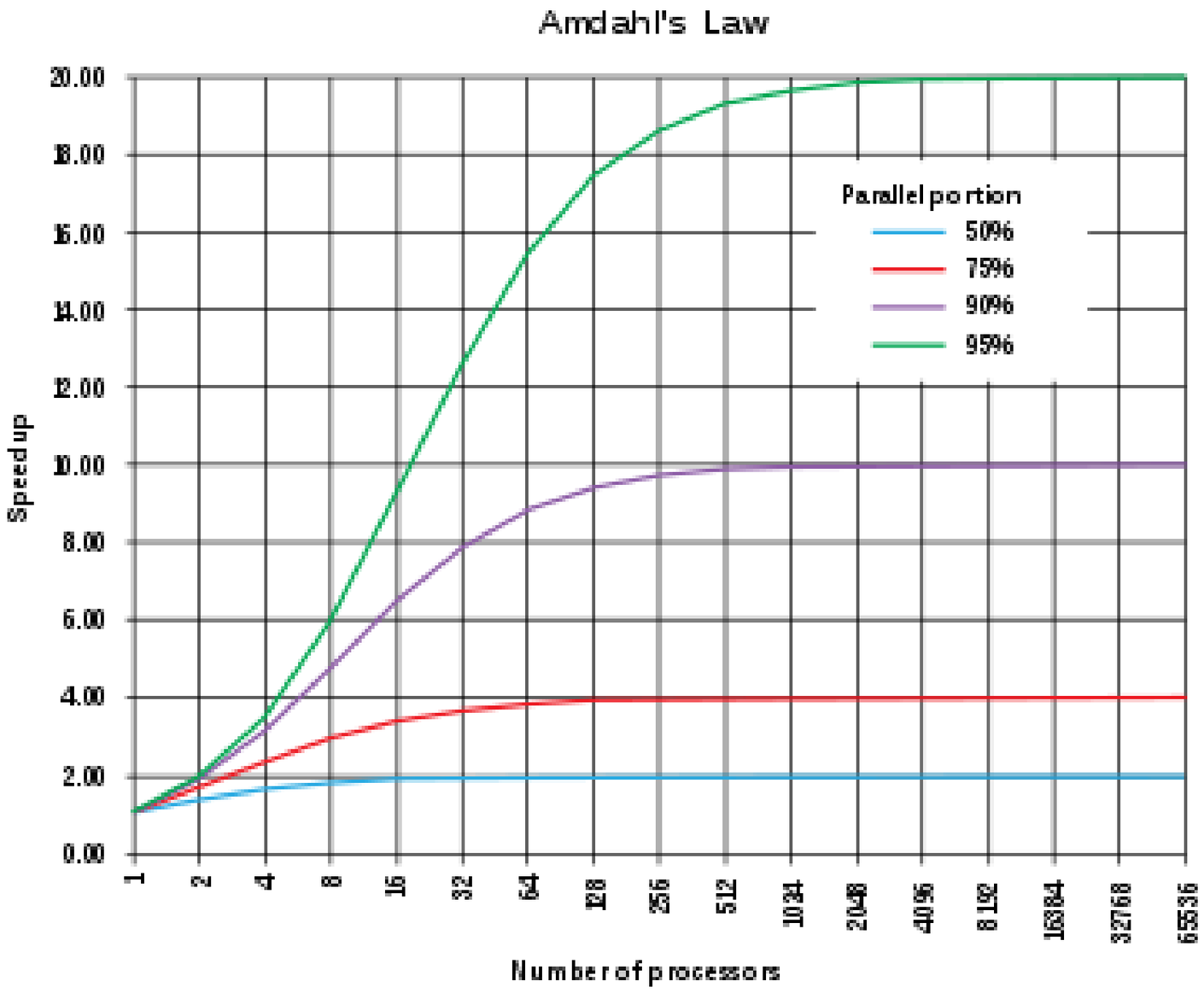
Shared memory multi-threading

- 14 nm process
- 682.6 mm² die size
- 76 CPU cores
- 7,100,000,000 transistors



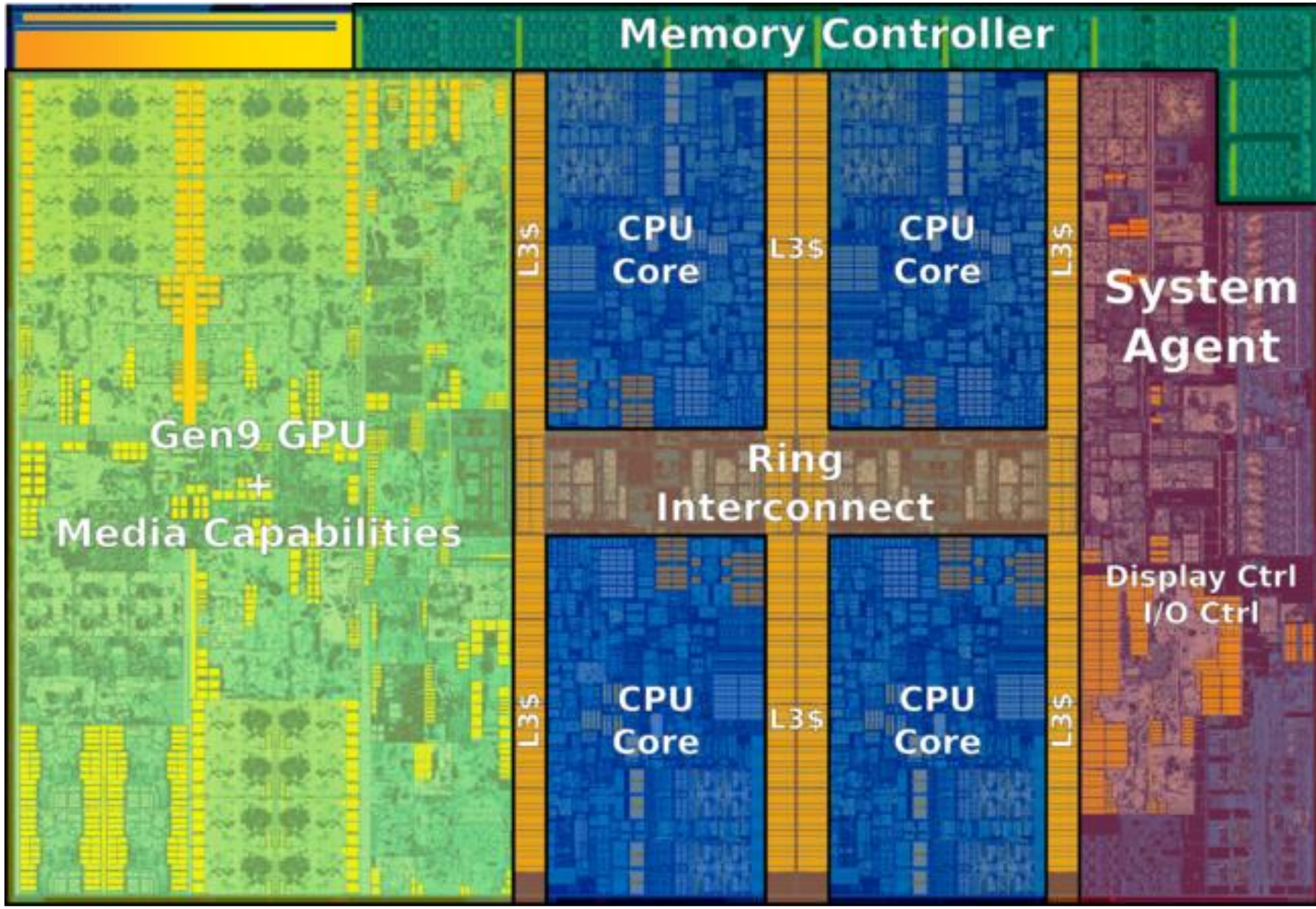
Amdahl’s Law

1/((1-p) + p/s)



Amdahl's Corollary:
Speedup is limited by fraction of the *program* that is parallelizable

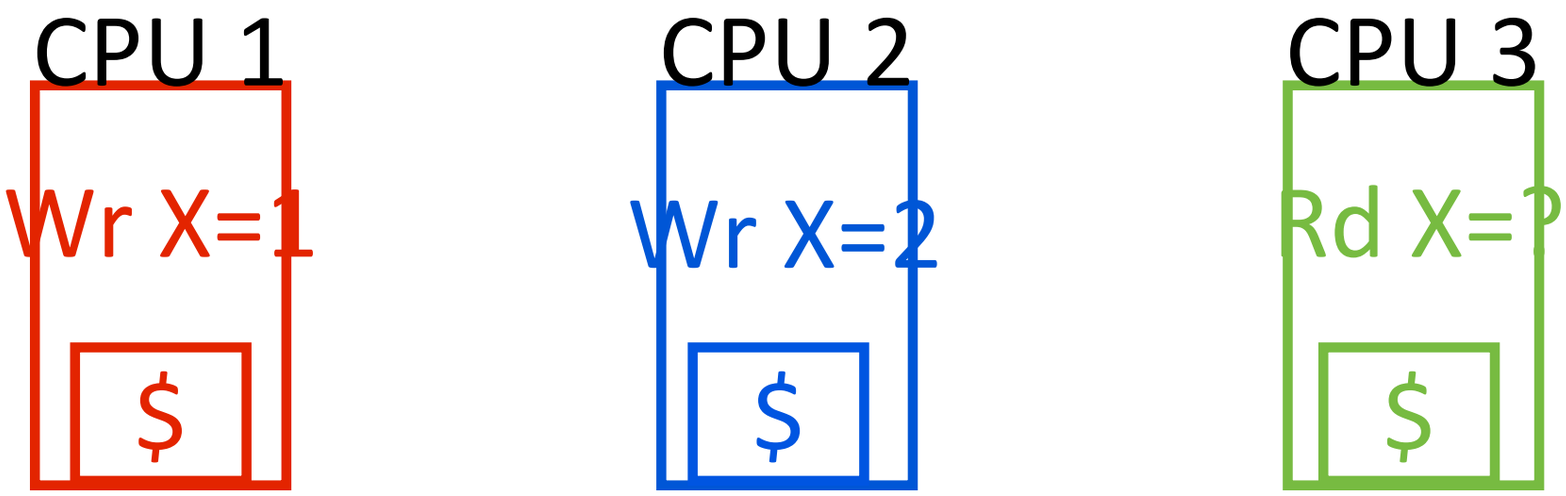
•14 nm process
•11 metal layers
•~1,750,000,000 transistors
•~9.19 mm x ~11.08 mm
•~101.83 mm² die size
•4 CPU cores + 24 GPU EUs



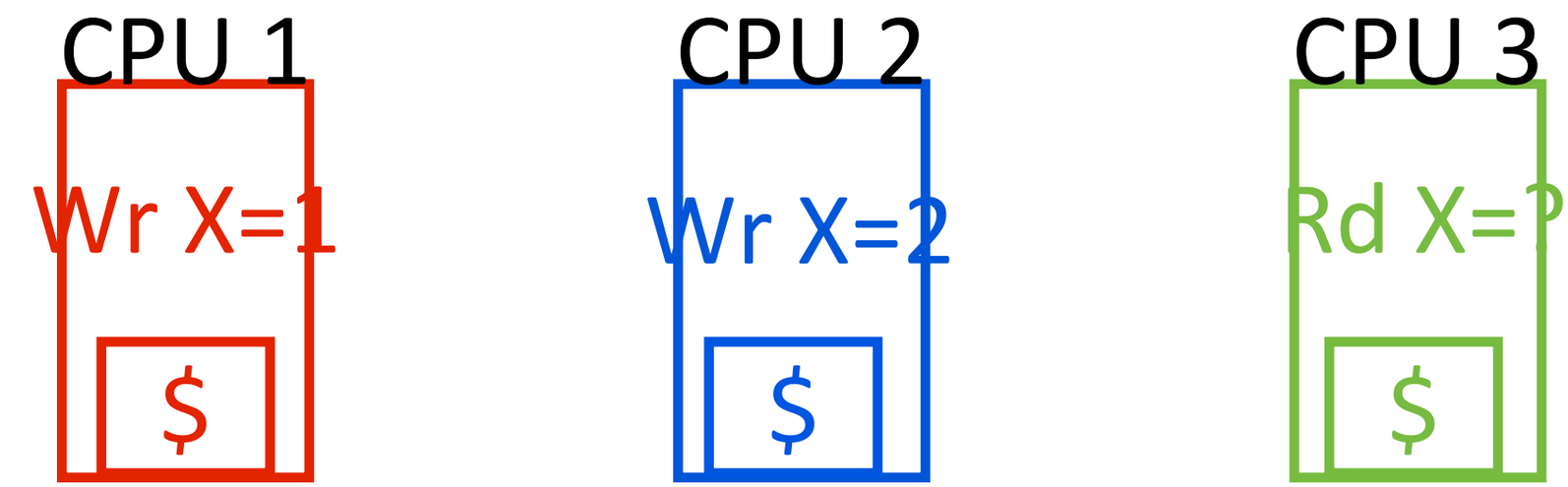
“Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system. Correct
coherence ensures that a programmer cannot determine whether and where a system has caches by
analyzing the results of loads and stores.”

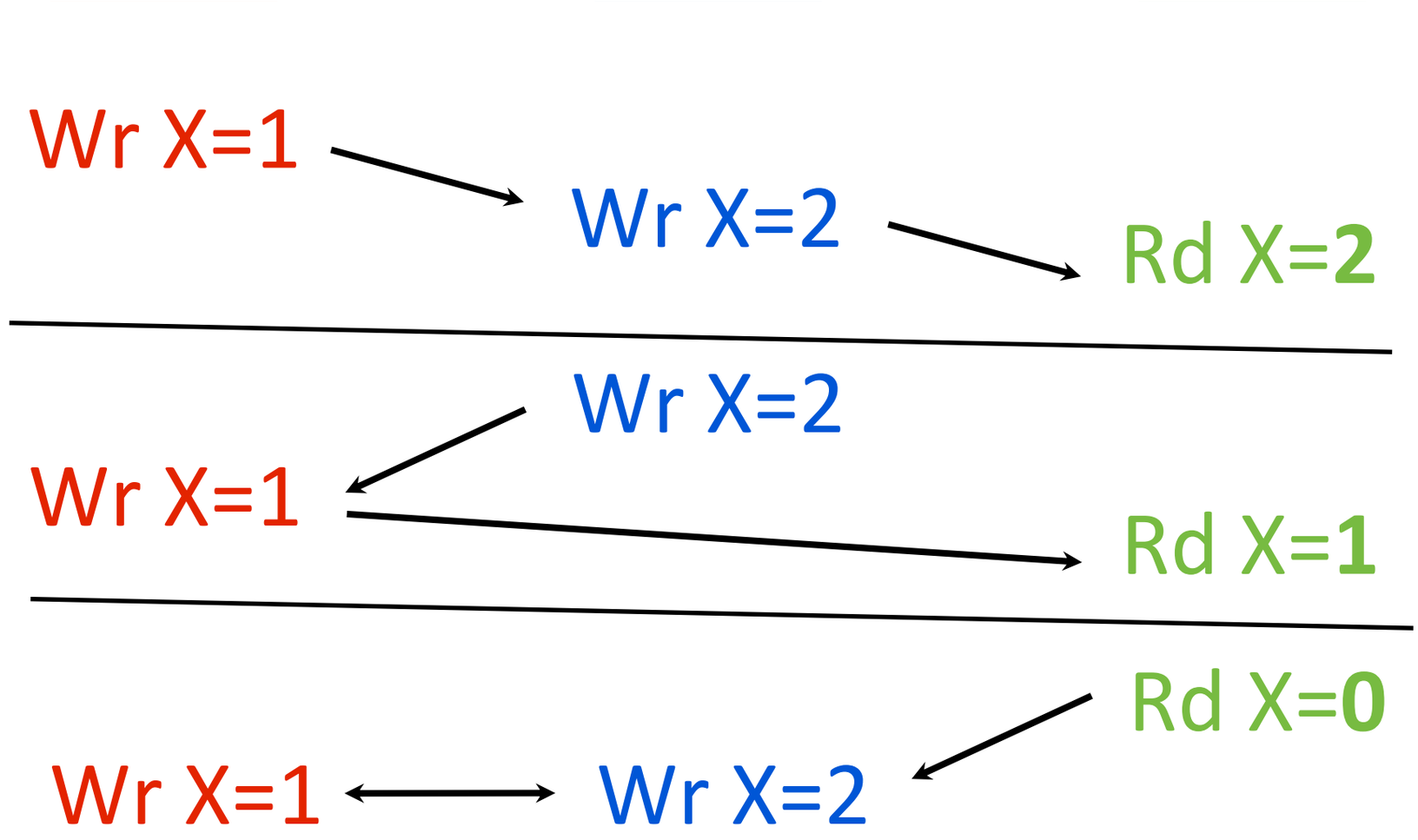
Excerpt from “Primer on Memory Consistency and Cache Coherence”
Mark Hill, 2011

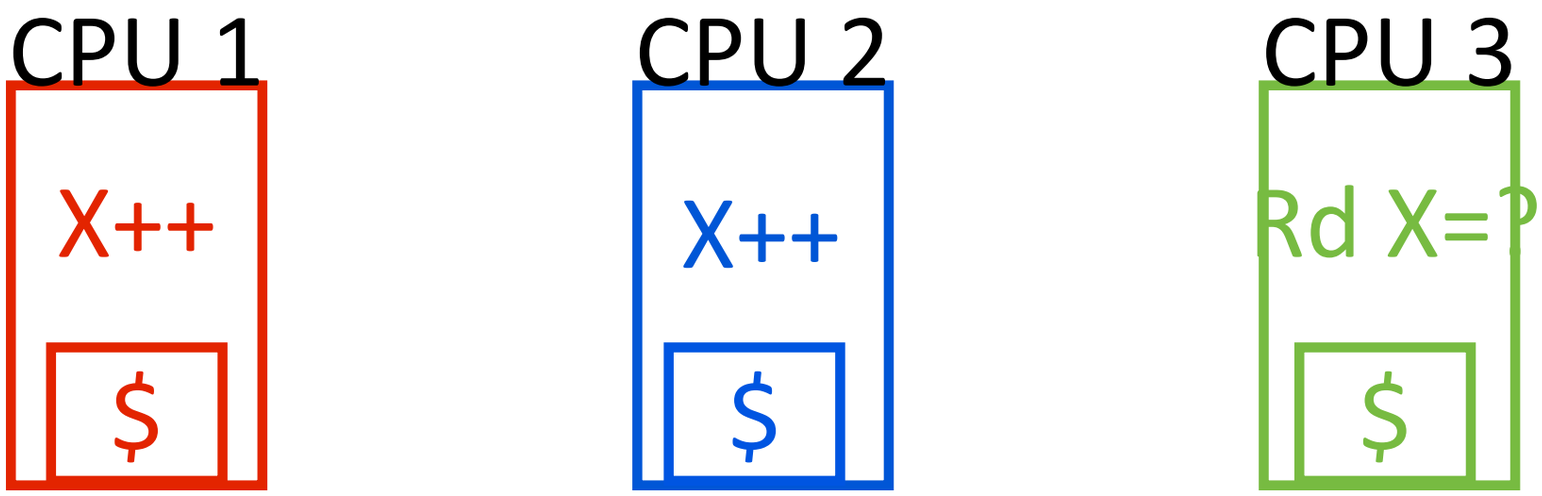
Cache Coherence



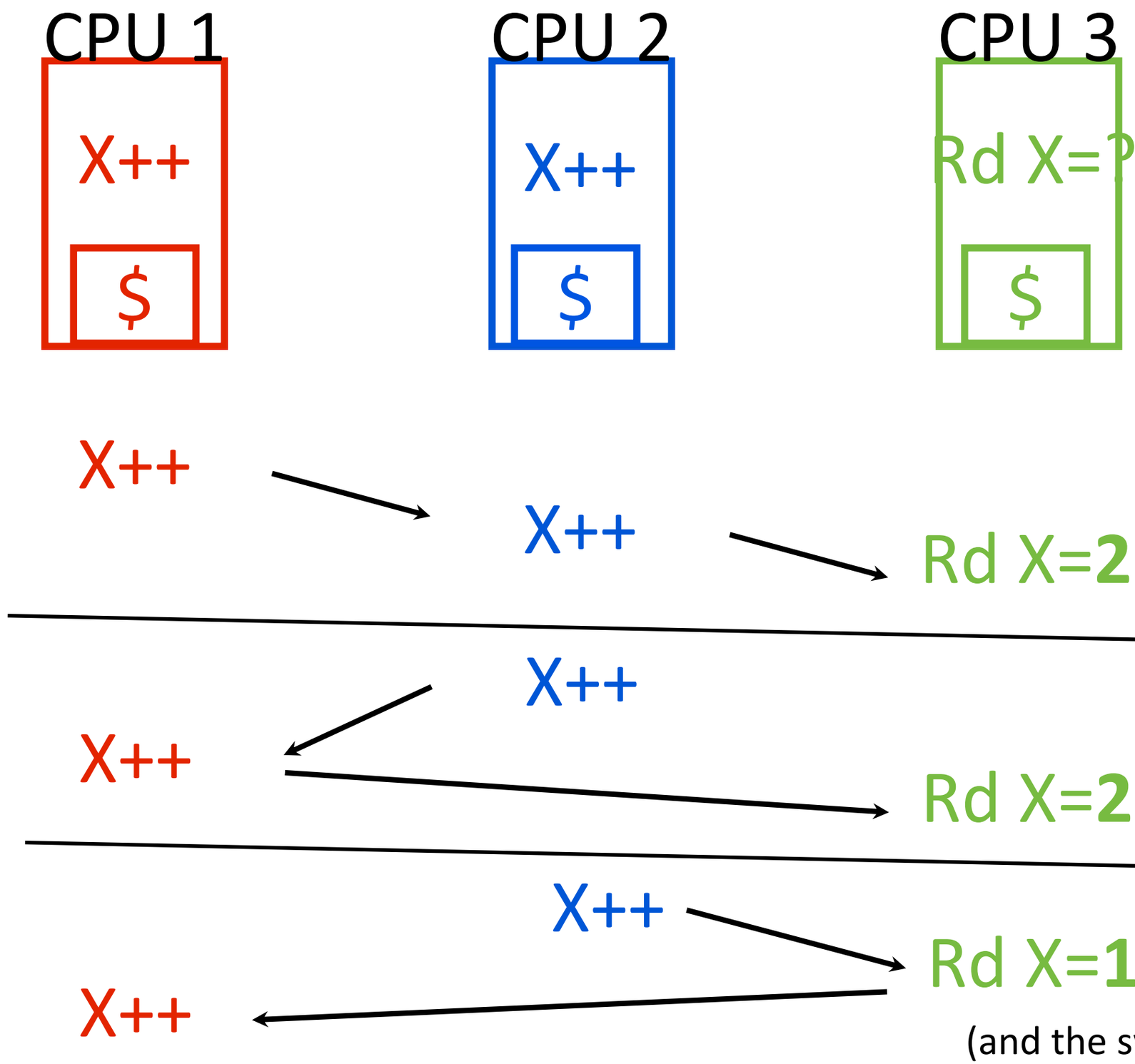
What is the behavior of this parallel program?
(X initially 0)


CPU 1
CPU 2
CPU 3
Wr X=1
Wr X=2
Rd X=?
$
$
$

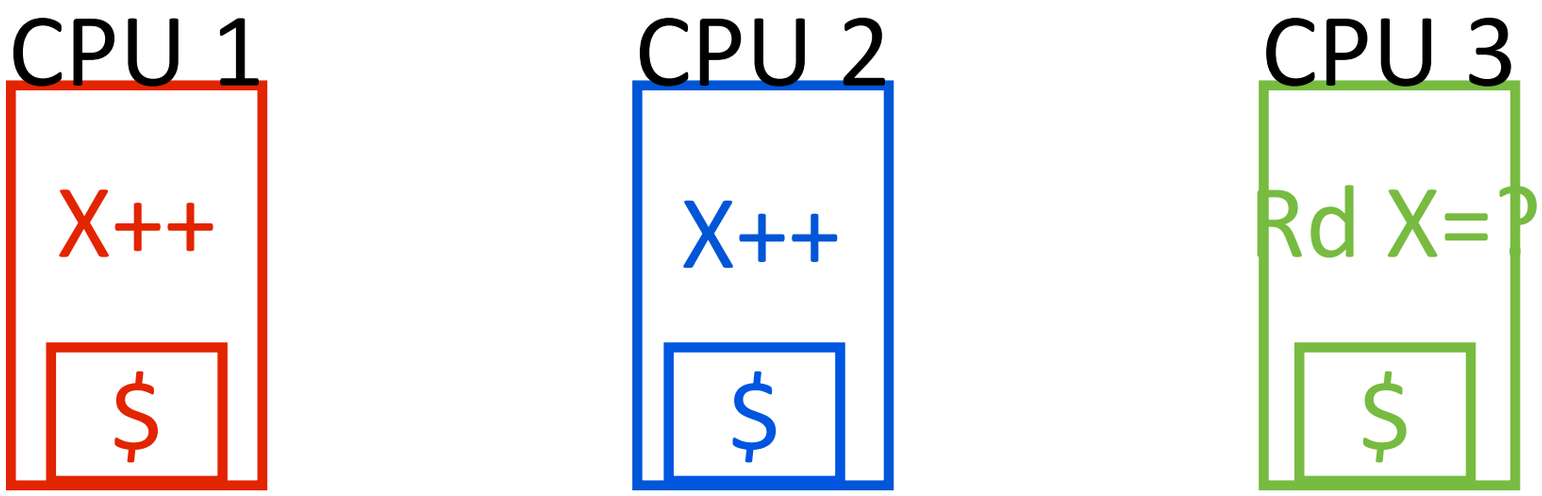

Wr X=1
Wr X=2
Rd X=2
Wr X=2
Wr X=1
Rd X=1
Rd X=0
Wr X=1
Wr X=2



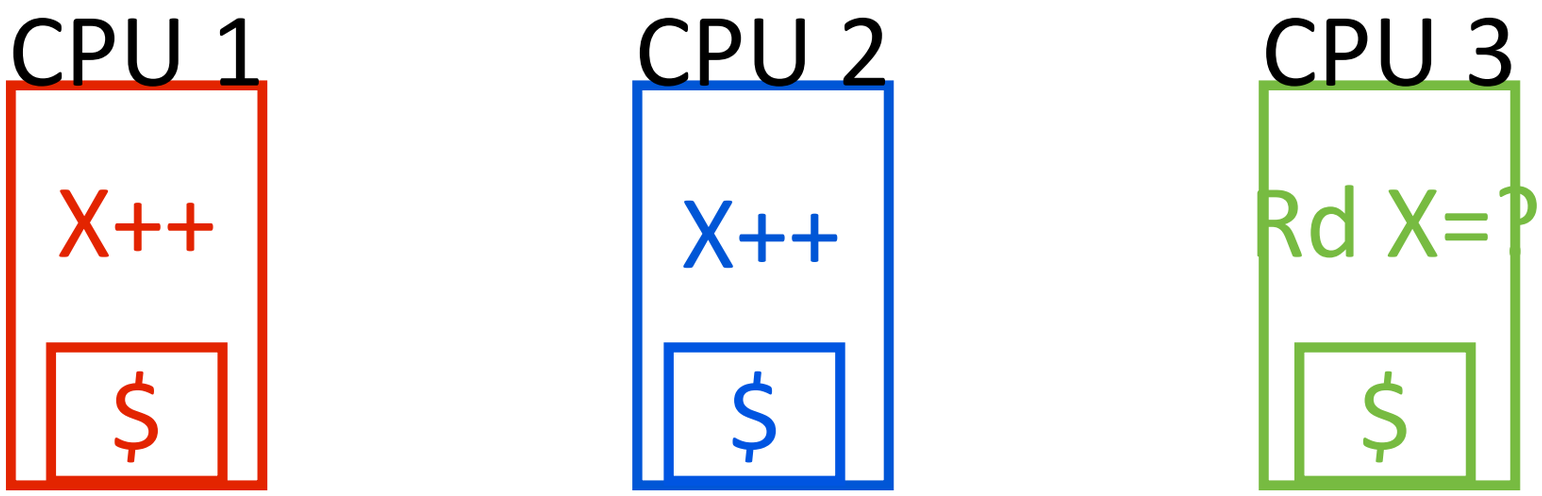
What about this example?
(X initially 0)



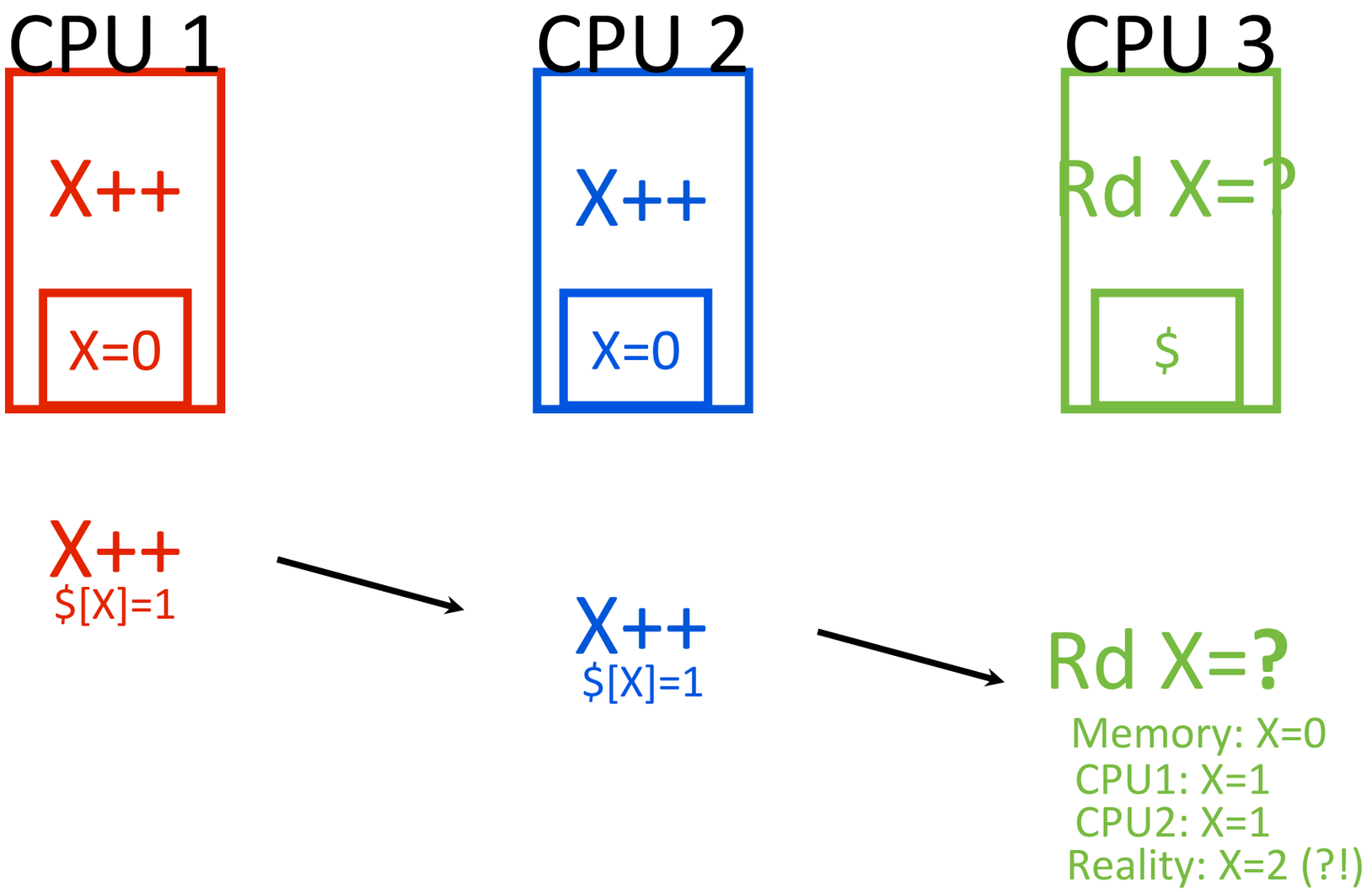
(and the symmetric case)



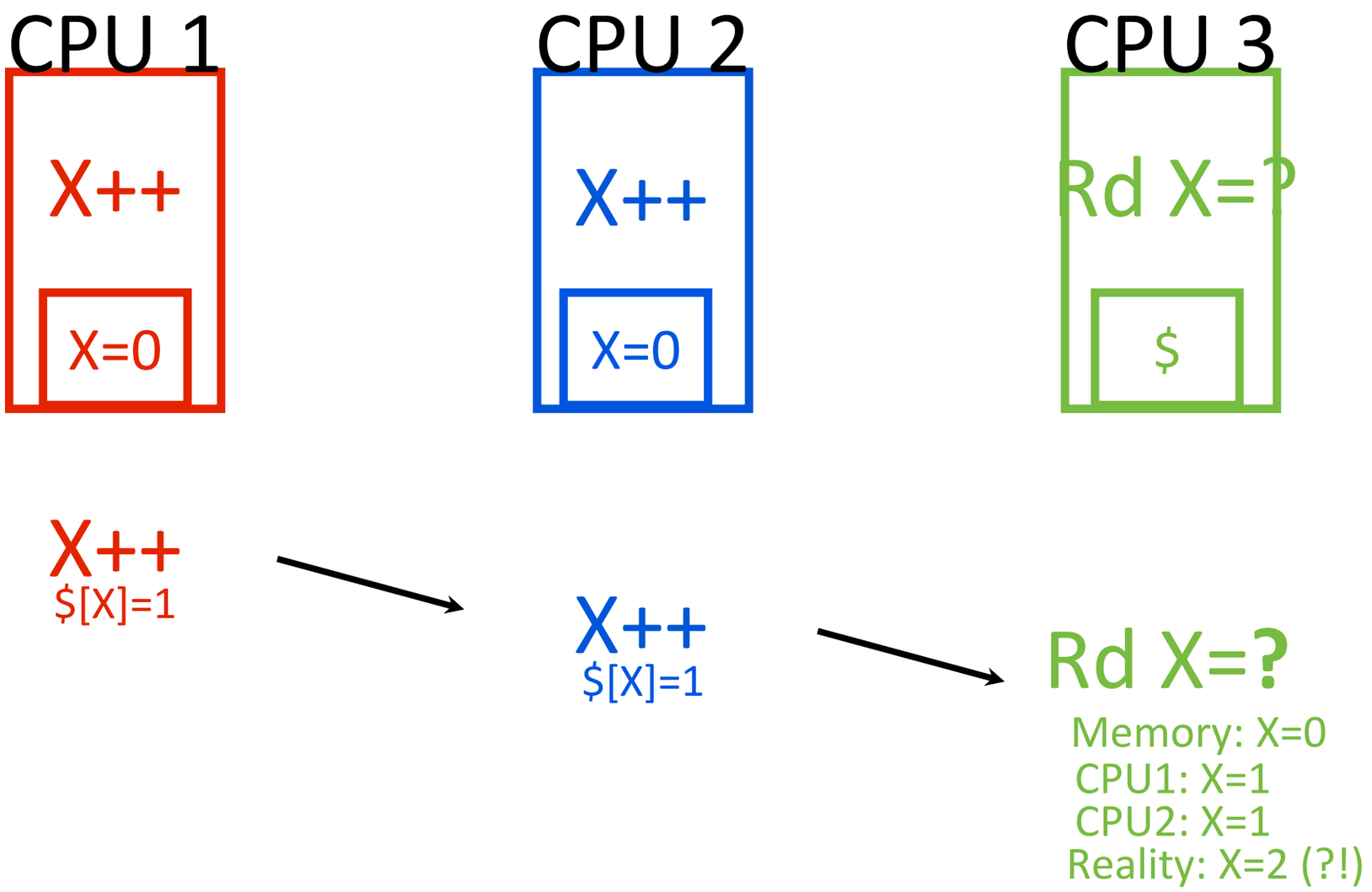
What assumptions are we making about the system to produce the results 0, 1, and 2?



We assume the updates see one anothers' results!
(Why wouldn't they?)



So what the heck do we do now?



Never let this happen. Caches should be **coherent.**

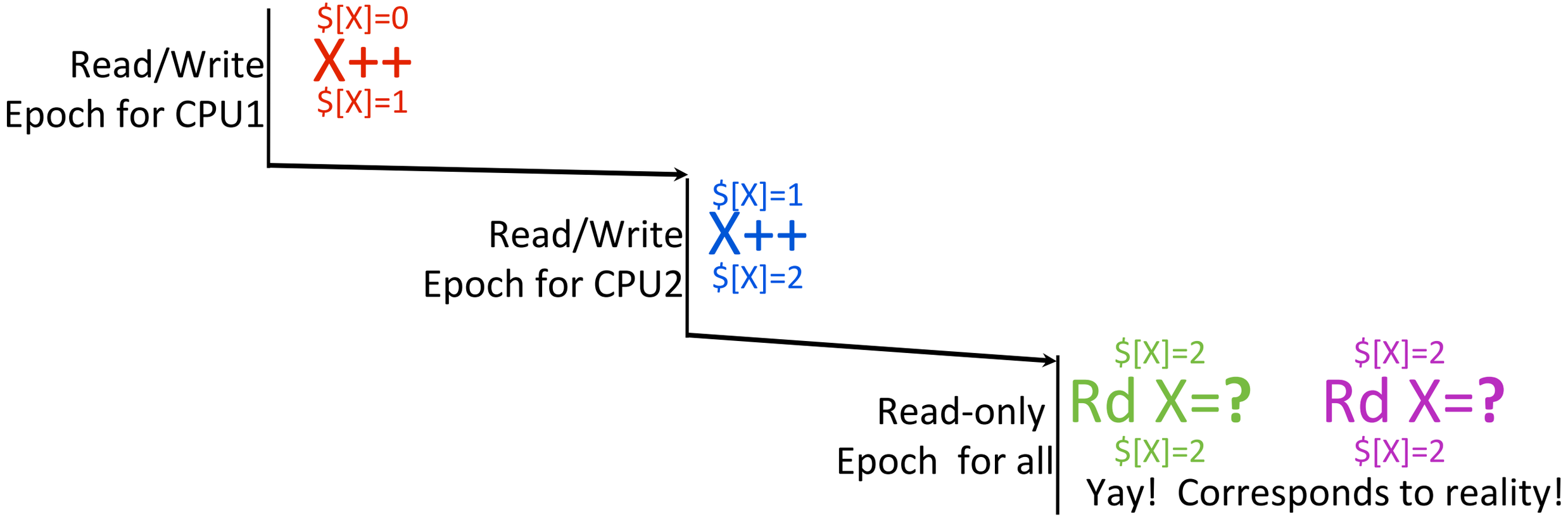
"coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores"

Informally Defining Coherence

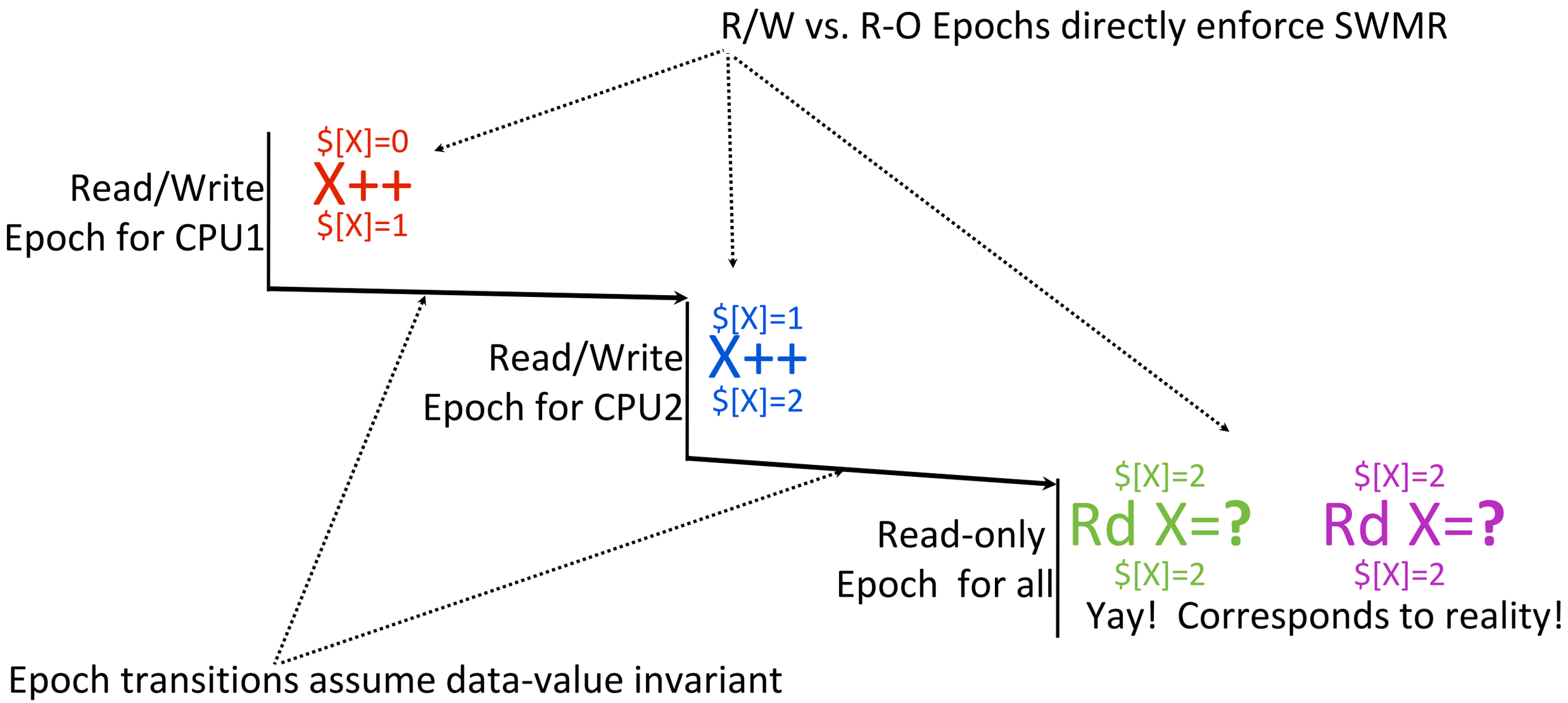
"Coherence serializes all reads with all updates to the same location by different CPUs/caches, so that each read sees the result of the most recent update by any other"

"Single Writer/Multiple Reader (SWMR) Invariant
+
Data-Value Invariant"

Epoch Model


$[X]=0
Read/Write
X++
$[X]=1
Epoch for CPU1
$[X]=1
Read/Write
X++
$[X]=2
Epoch for CPU2
Read-only
Epoch for all
$[X]=2
Rd X=?
$[X]=2
$[X]=2
Rd X=?
$[X]=2
Yay! Corresponds to reality!

Epoch Model


R/W vs. R-O Epochs directly enforce SWMR
Read/Write
Epoch for CPU1
$[X]=0
X++
$[X]=1
Read/Write
Epoch for CPU2
$[X]=1
X++
$[X]=2
Read-only
Epoch for all
$[X]=2
Rd X=?
$[X]=2
$[X]=2
Rd X=?
$[X]=2
Yay! Corresponds to reality!
Epoch transitions assume data-value invariant

What do we need to implement the Epoch Model?

Need to add concept of R/W epoch vs. R-O epoch

Need to add gadget that correctly moves data between epochs

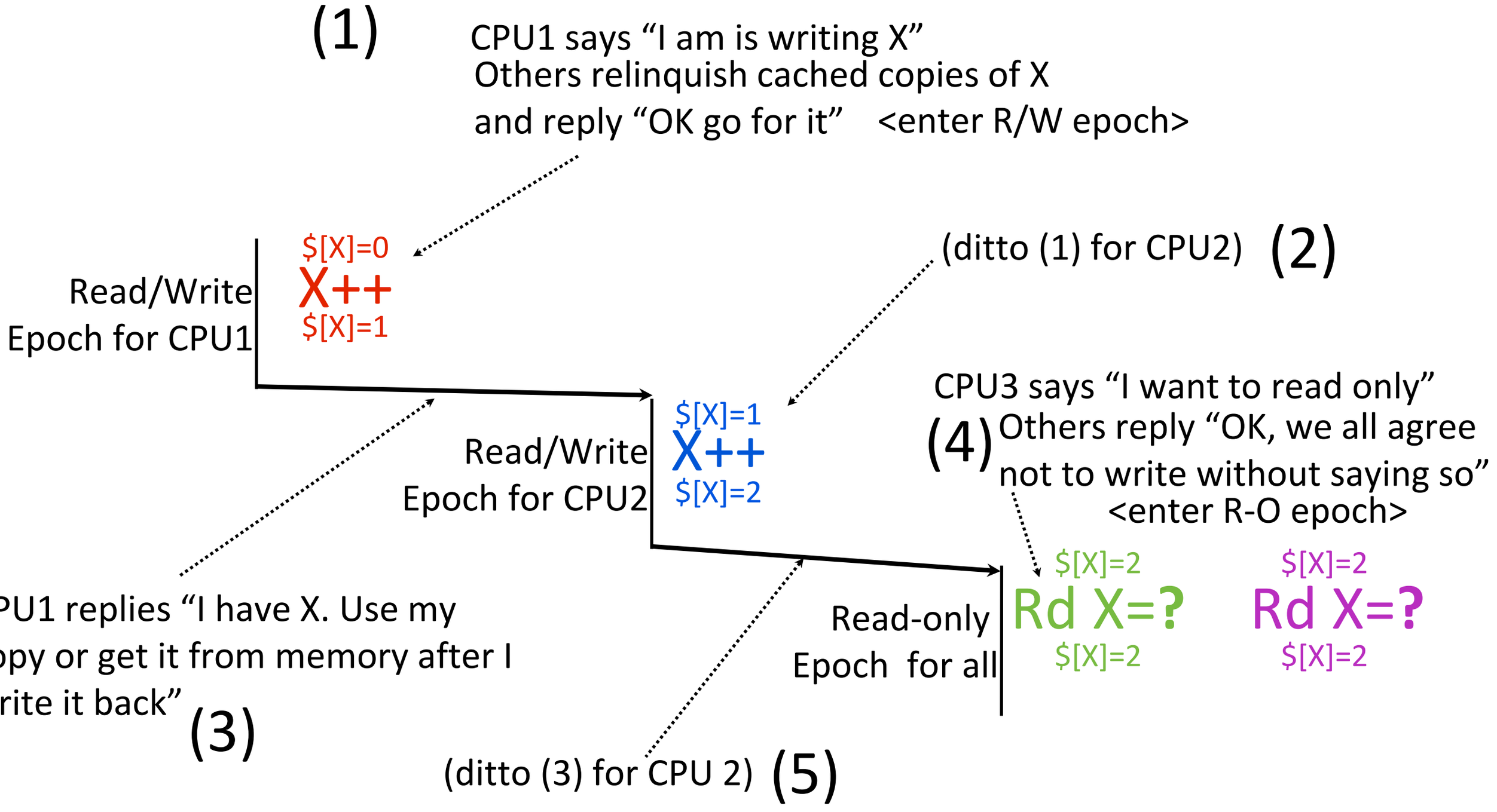
Cache Coherence Protocol

Add state to each cache line saying whether it is R-O or R/W

Add protocol actions to move lines from state to state based on (1)local memory operations; and (2)other CPUs' memory operations

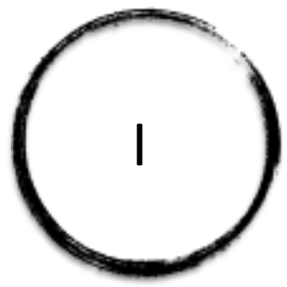
Add support to get data from (1)local cache; (2)a remote cache; or (3)main memory, depending on line's protocol state

High-level sketch of protocol in action



Cache Coherence Protocol

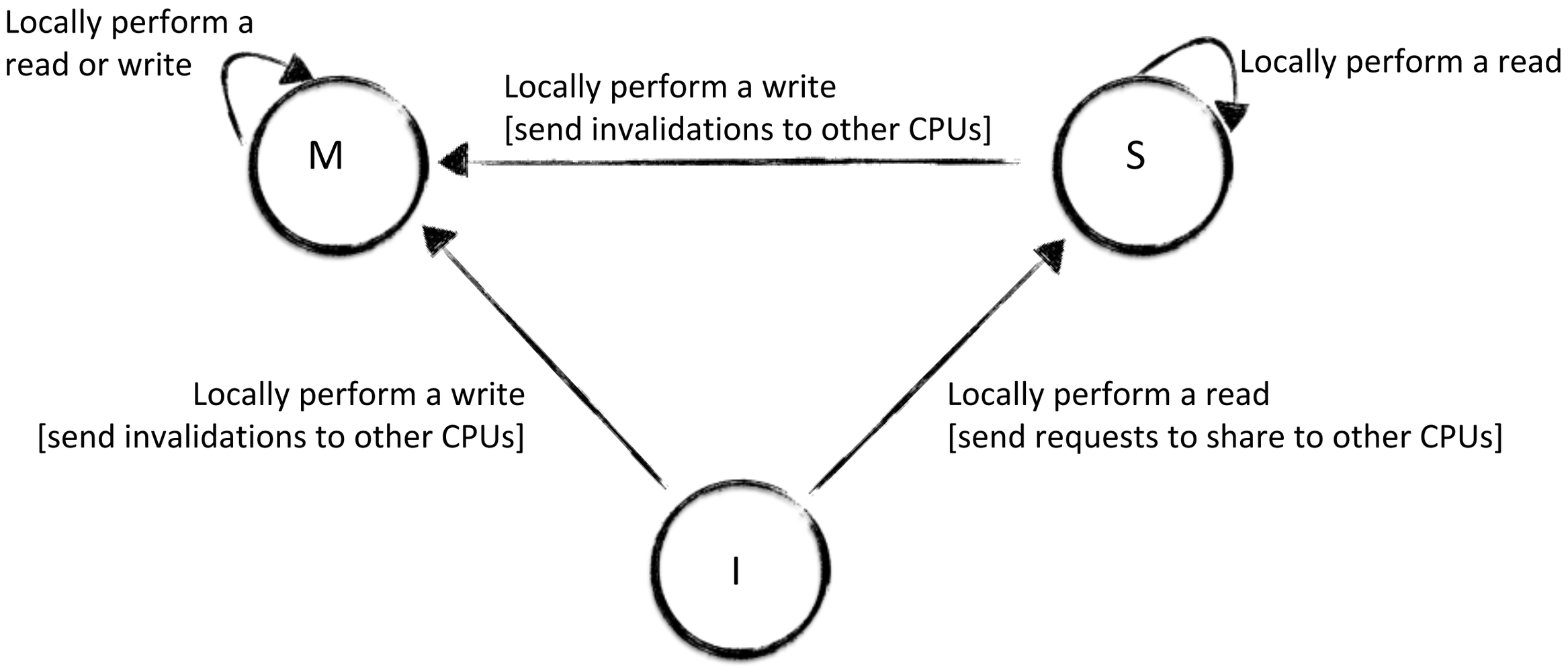
Per-line coherence states



Cache Coherence Protocol

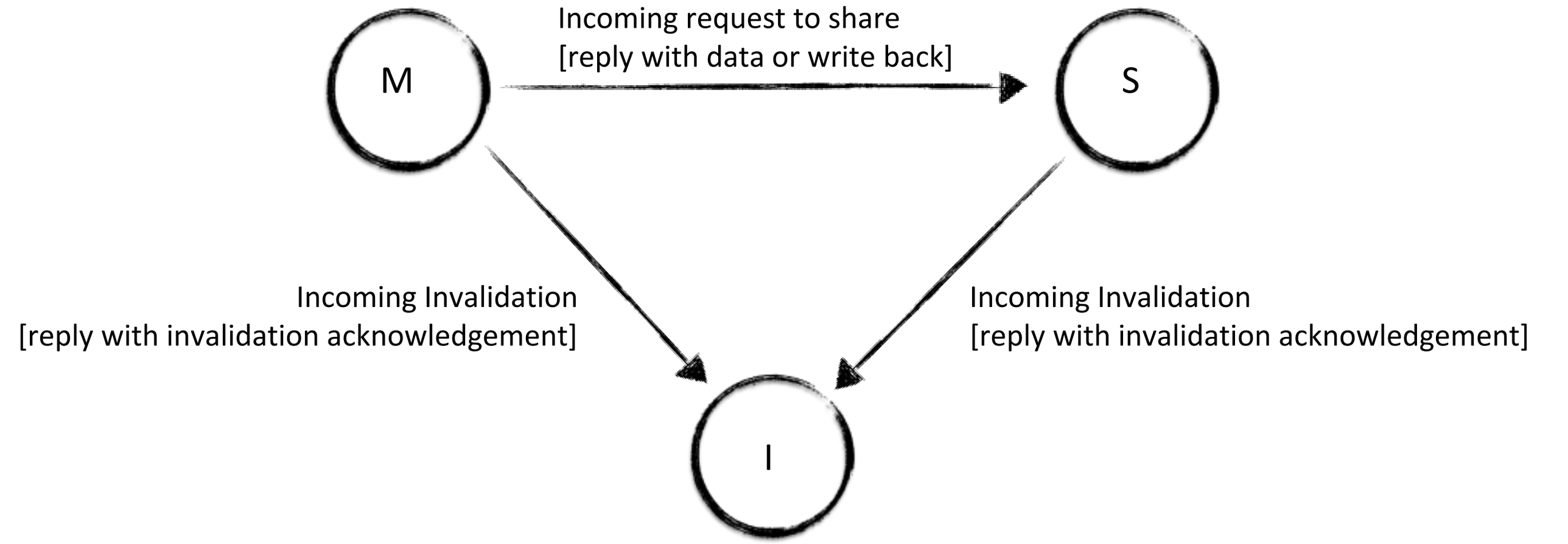
Cache Coherence Protocol

Local operations perspective

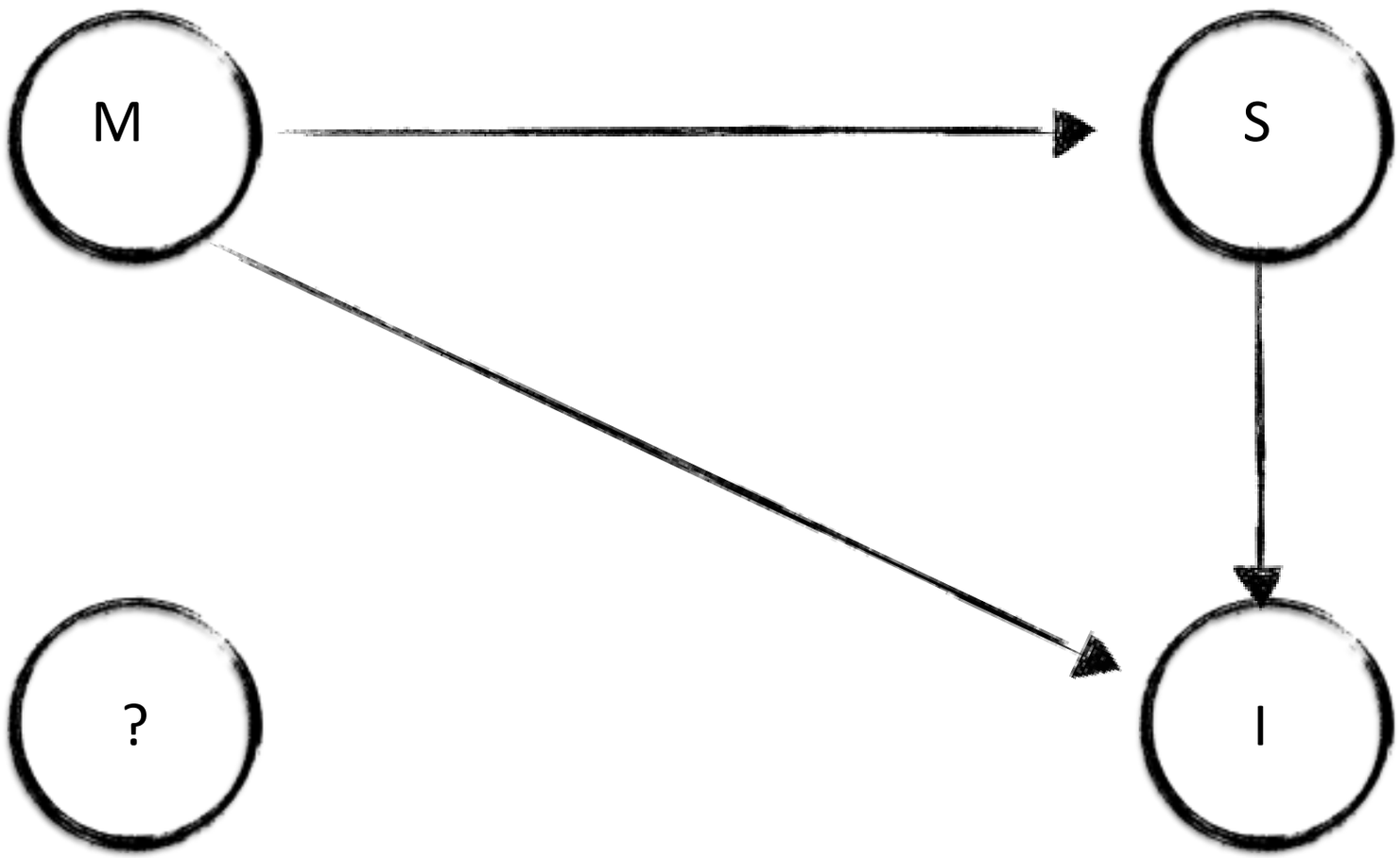


Cache Coherence Protocol

Remote operations perspective

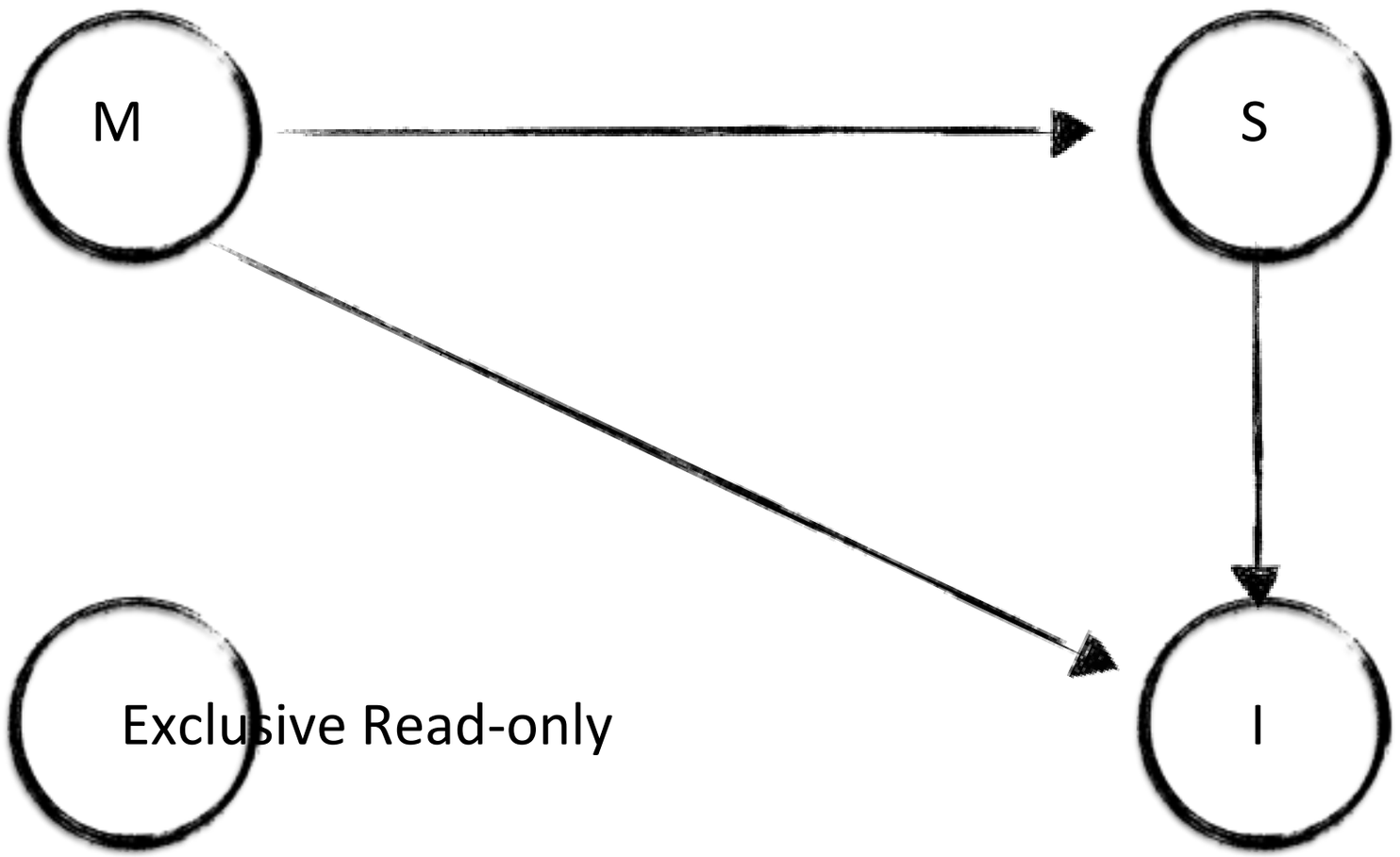


Can we design another state?



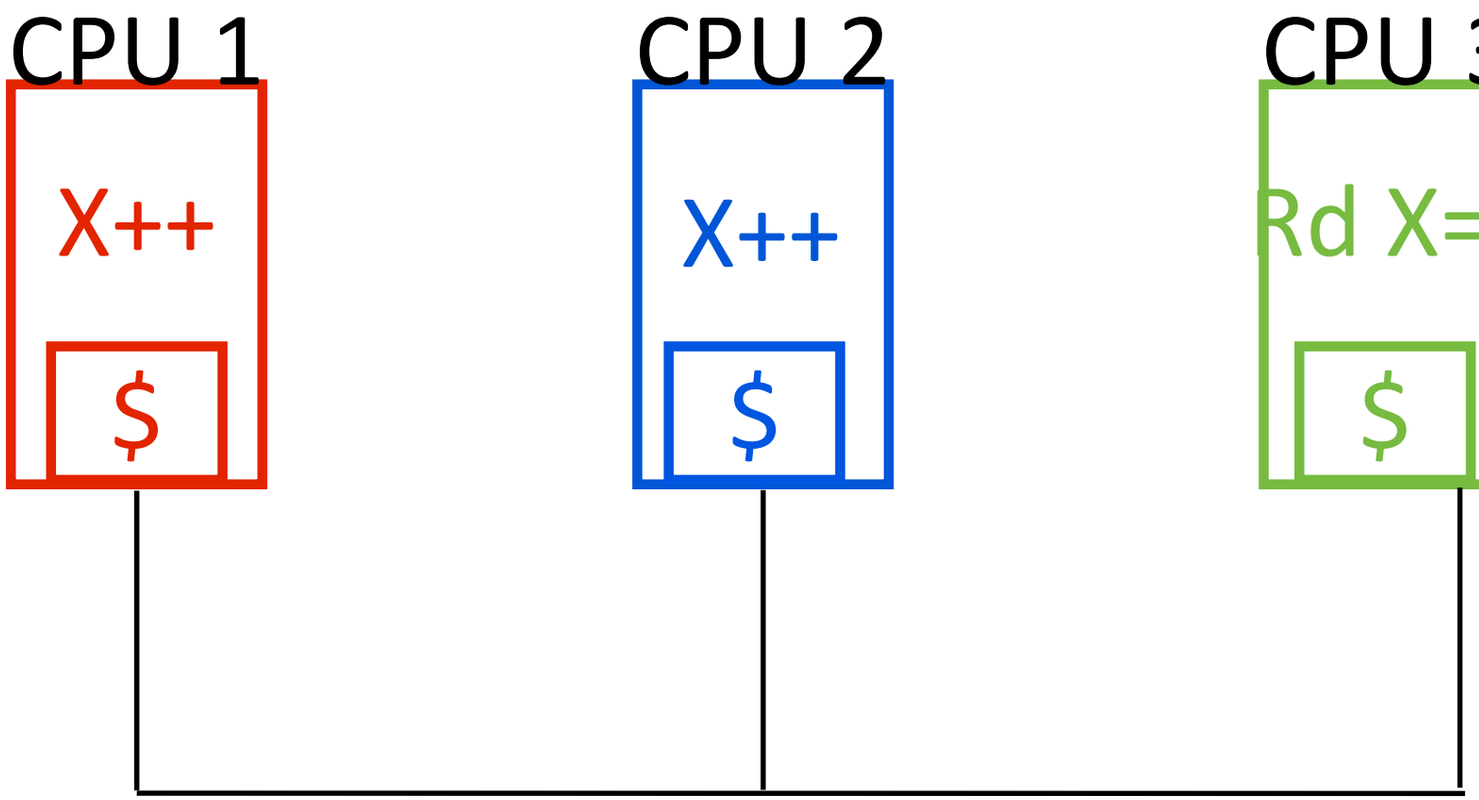
What should we optimize?

Can we design another state?



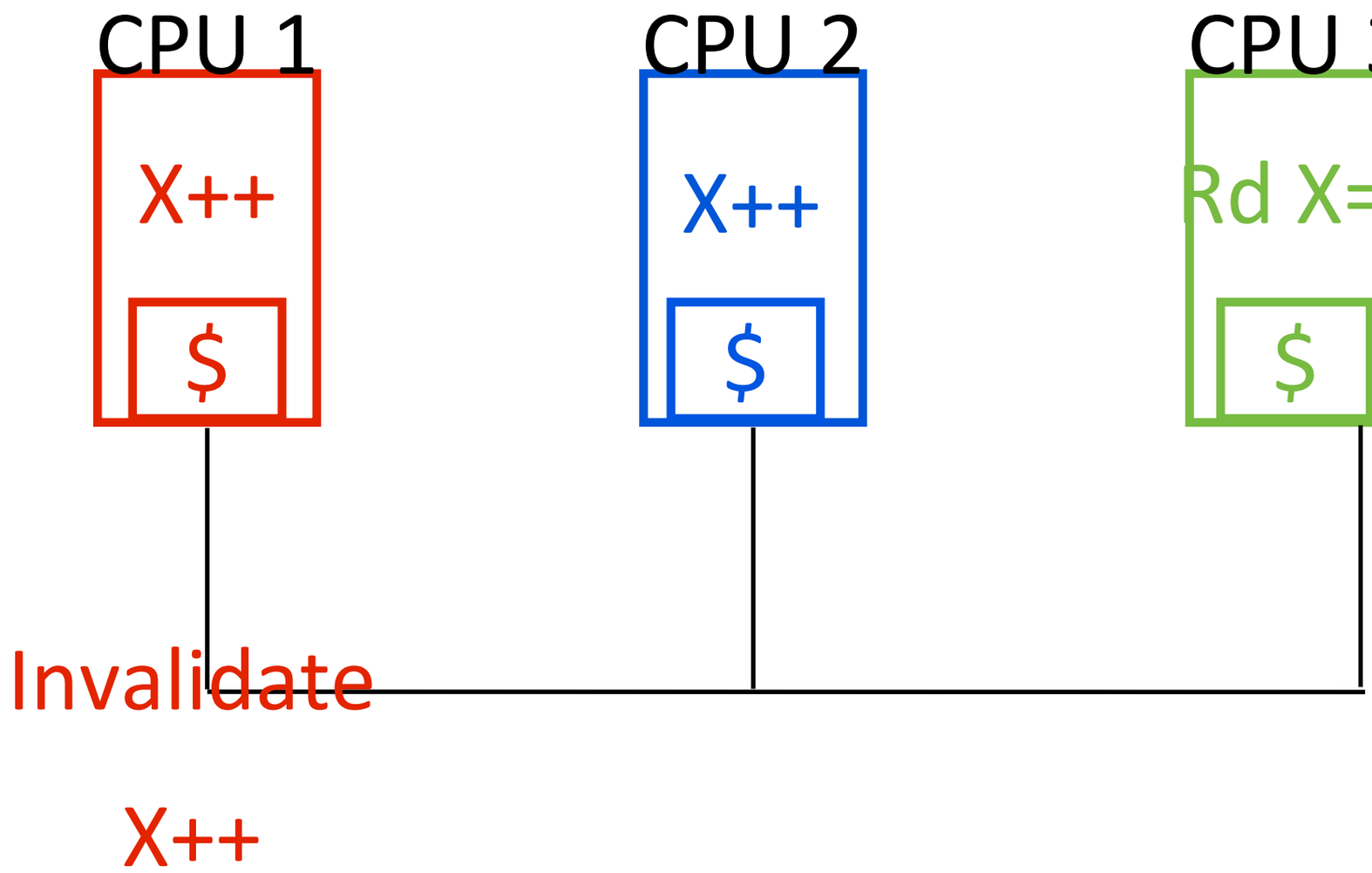
(Benefit: no invalidation required
to transition from E->M, like from S->M)

Implementing the Protocol

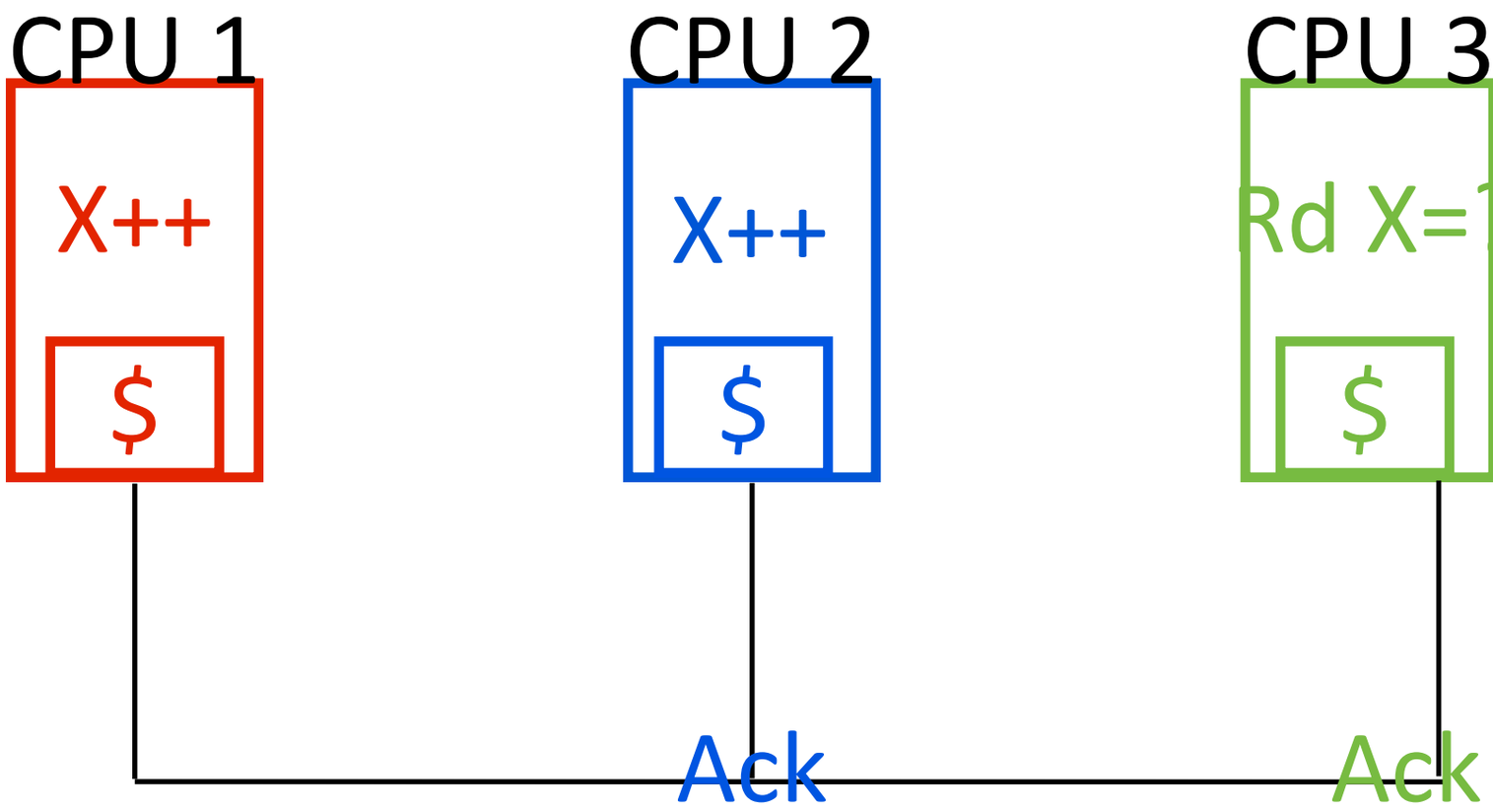


Snoopy Coherence

Implementing the Protocol

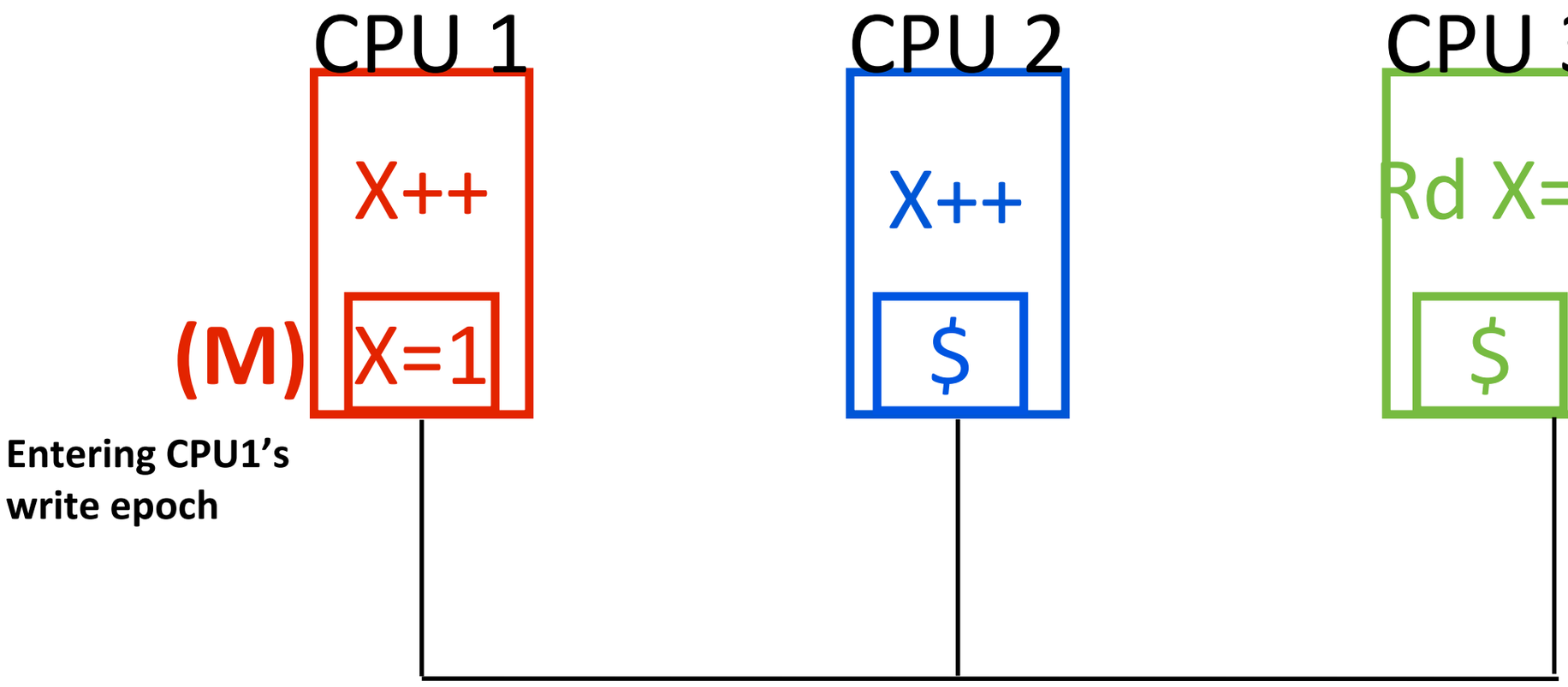


Implementing the Protocol



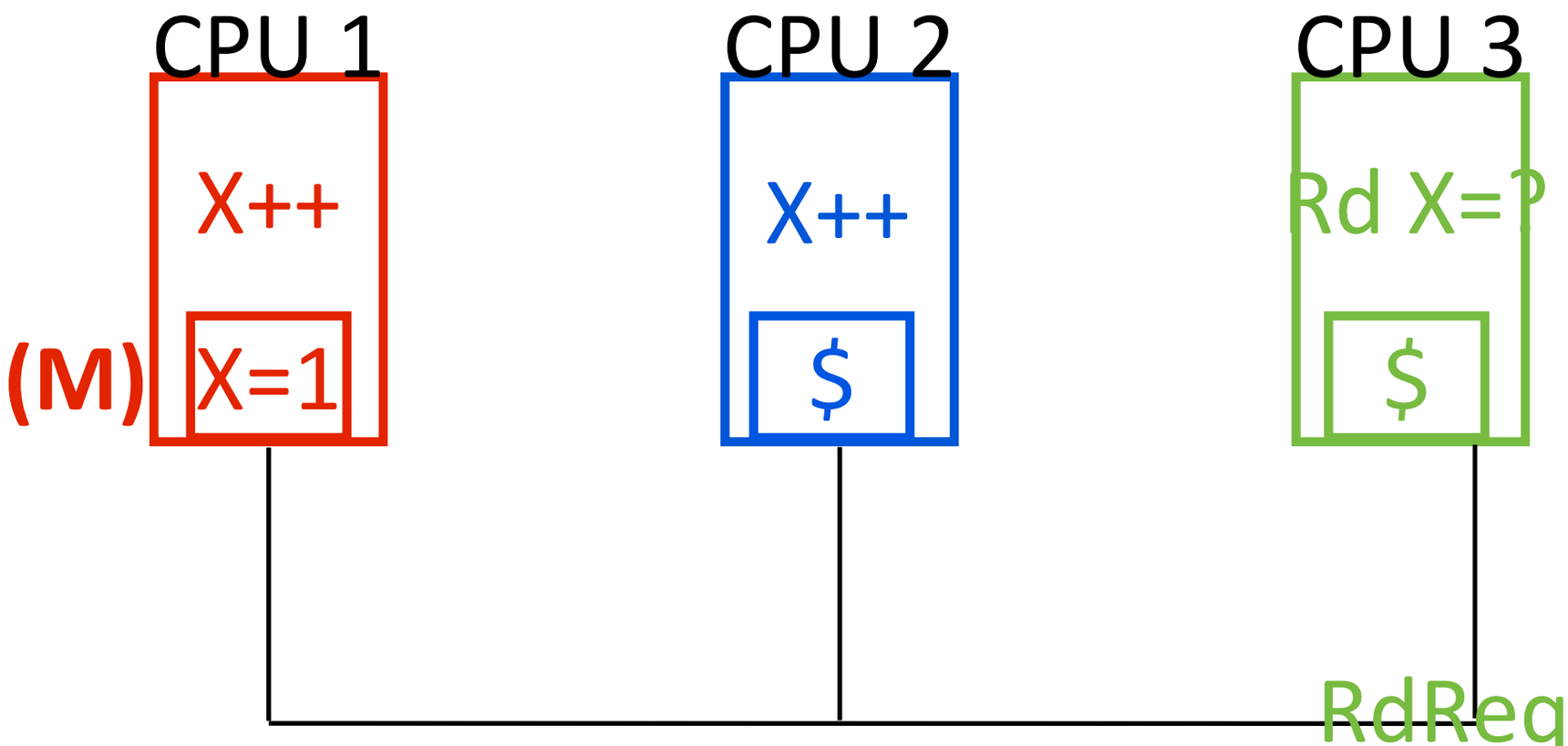
X++

Implementing the Protocol

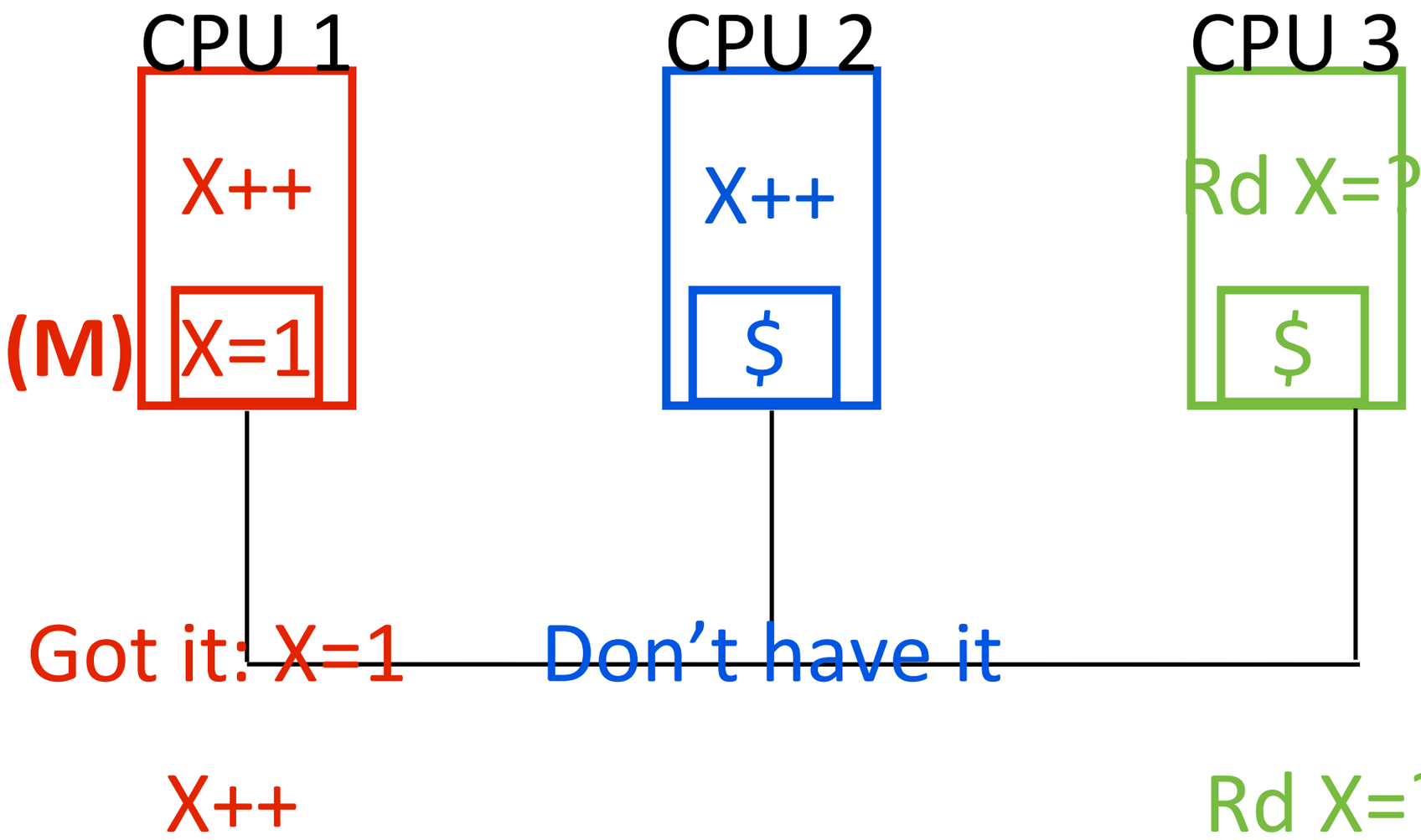


X++

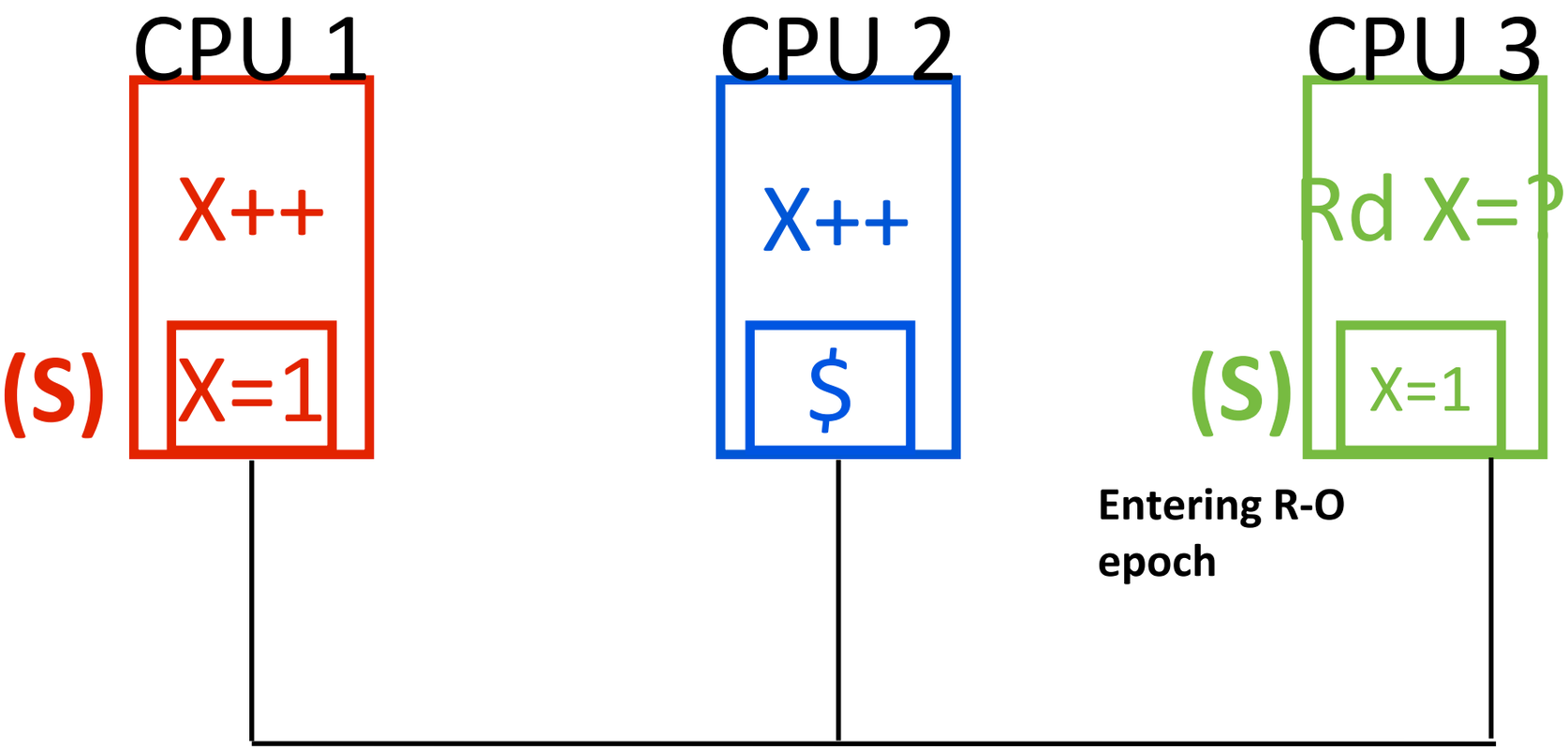
Implementing the Protocol



Implementing the Protocol



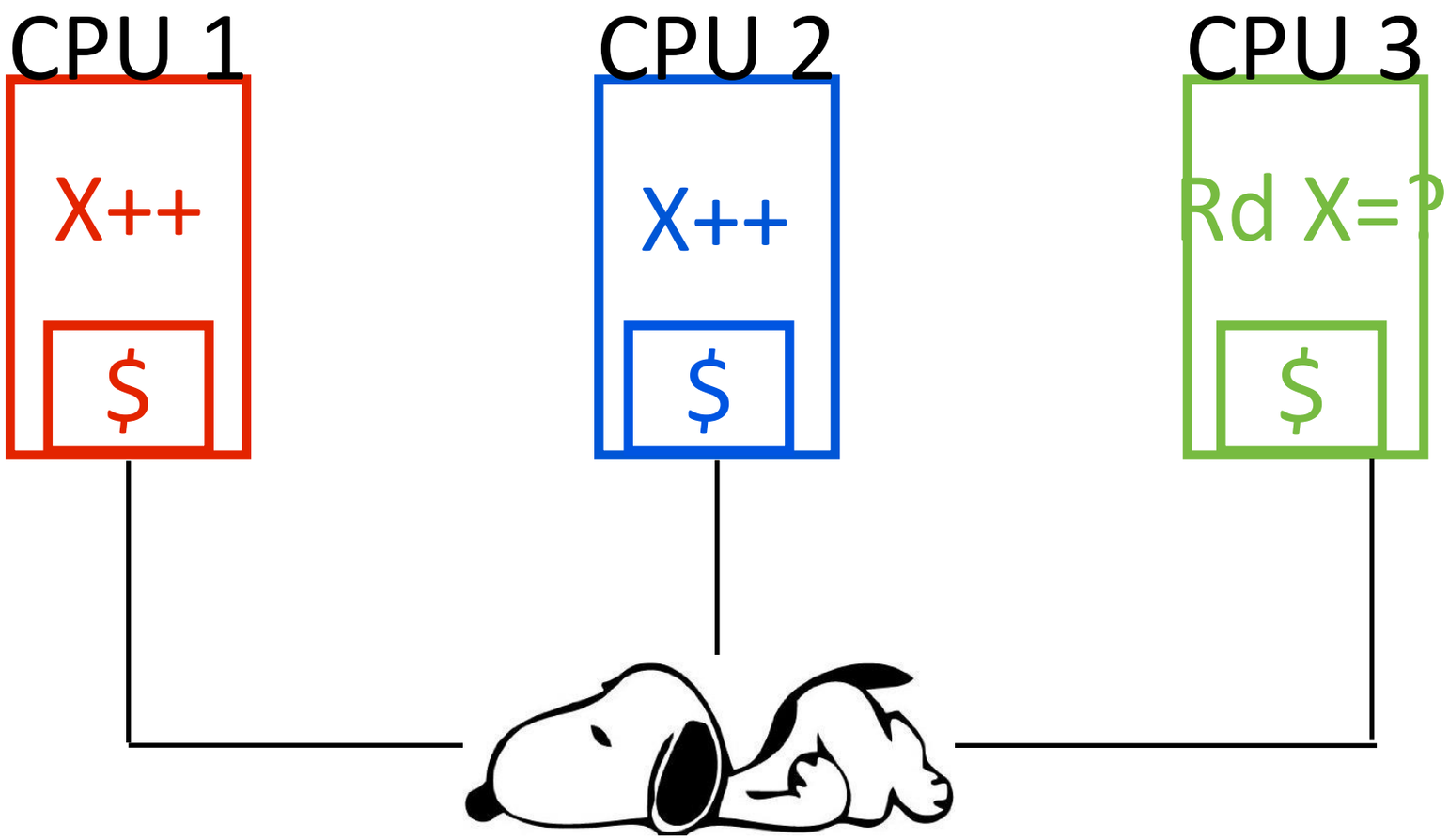
Implementing the Protocol



X++

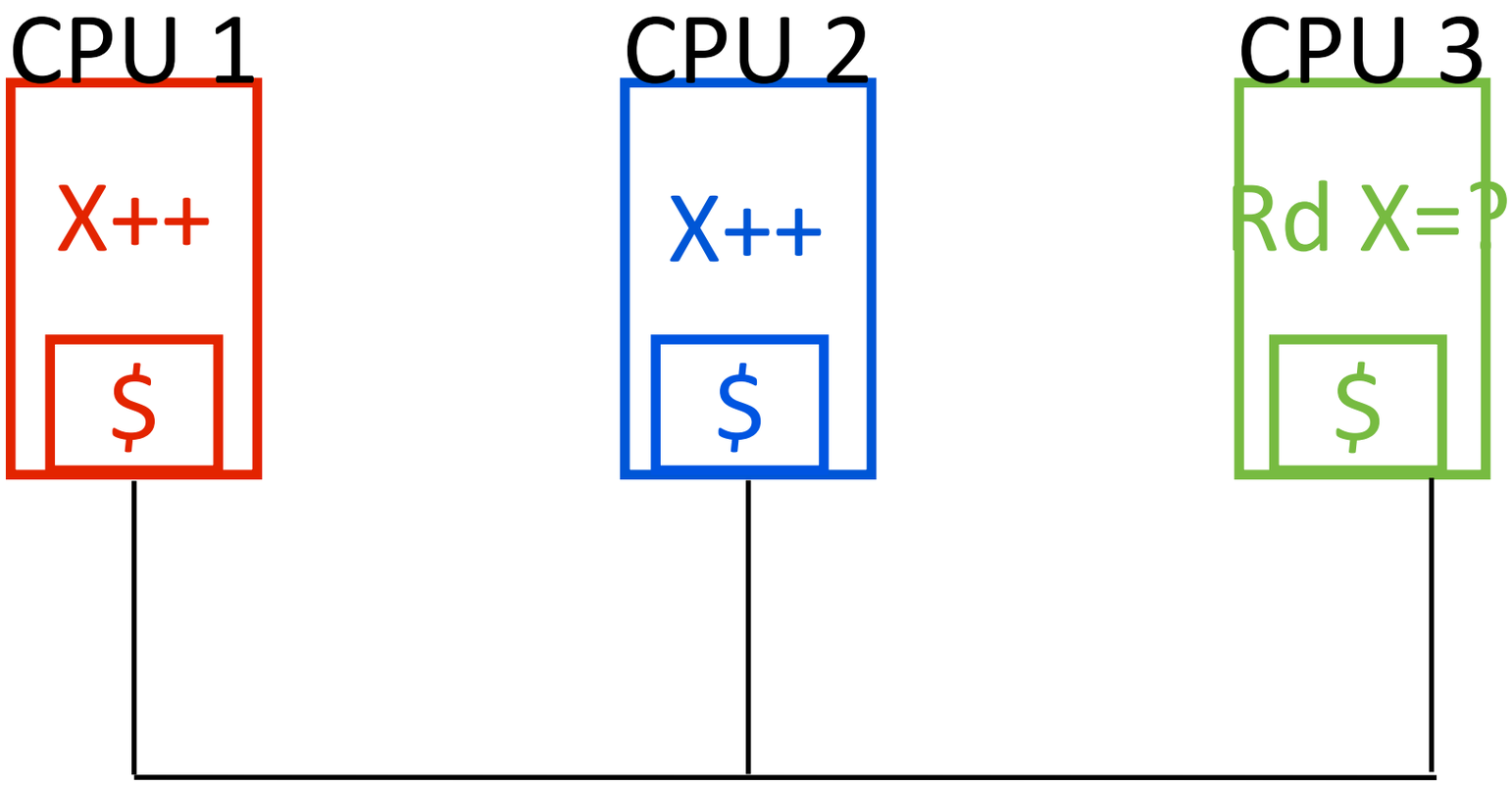
Rd X=?

Implementing the Protocol



What sucks about Snoopy?

Implementing the Protocol



Bus limits scalability due to congestion and complex message arbitration

Figure 1-1. Uncore Sub-system Block Diagram of Intel Xeon Processor E5-2600 Family

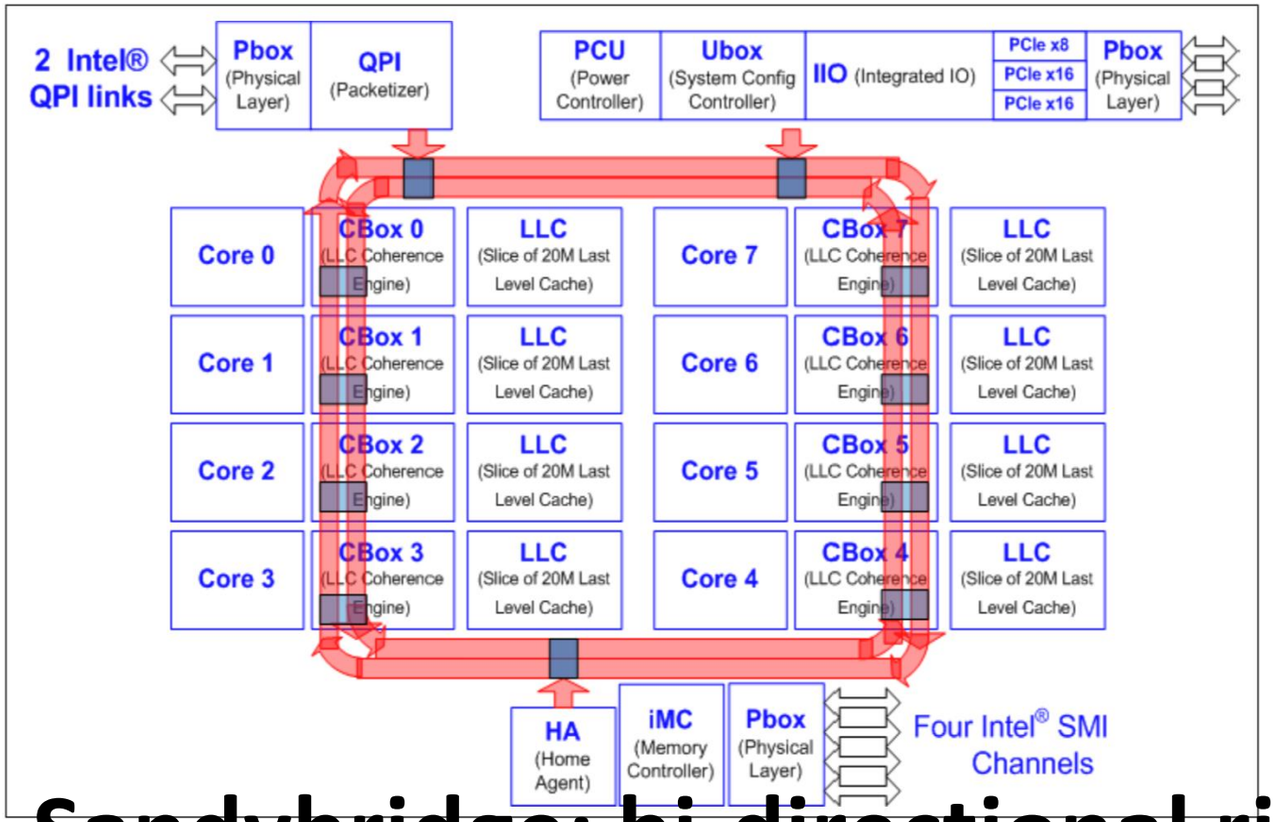
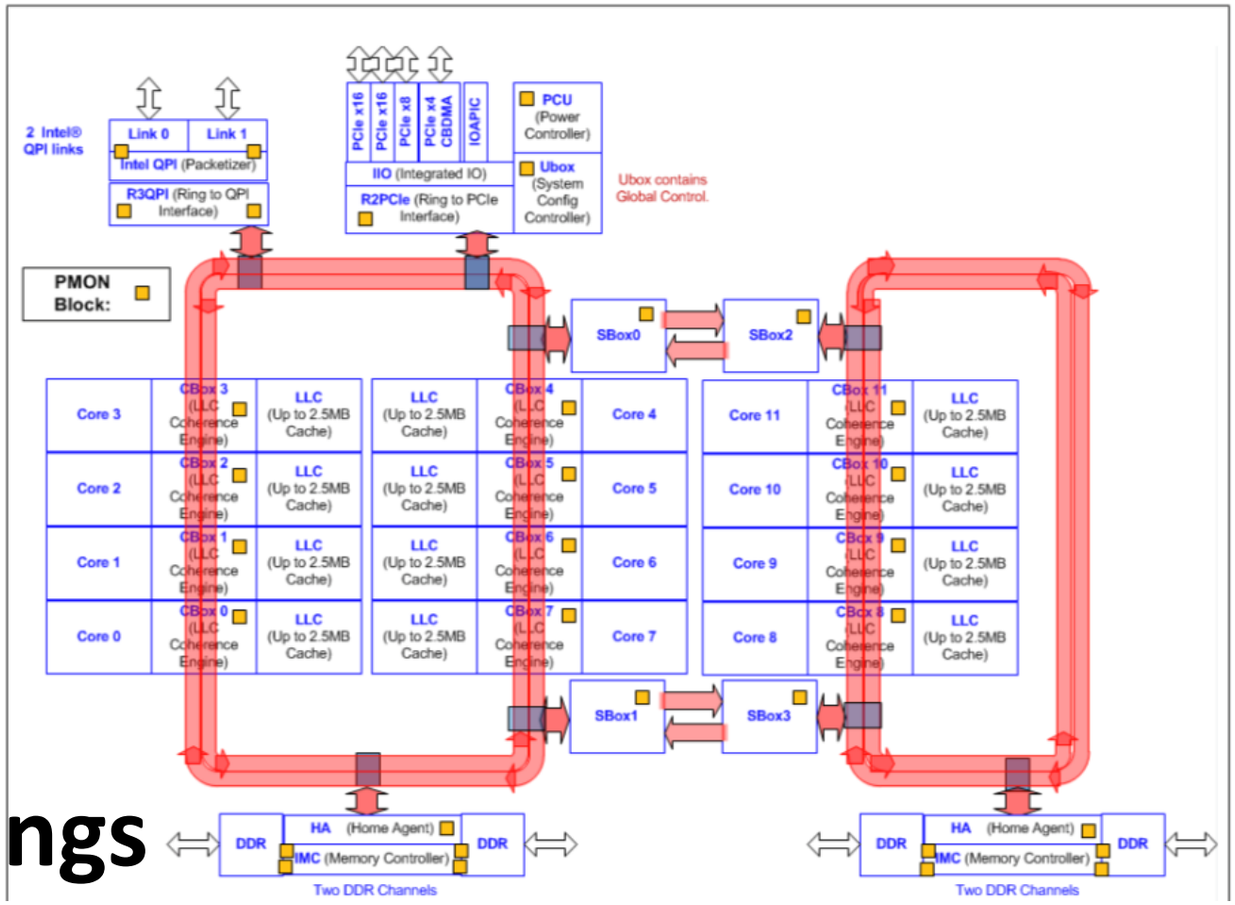


Figure 1-2. Intel® Xeon® Processor E5 v3-1600/2600/4600 Family -12C Block Diagram



Sandybridge: bi-directional rings

Knight's Landing 2017 2D mesh

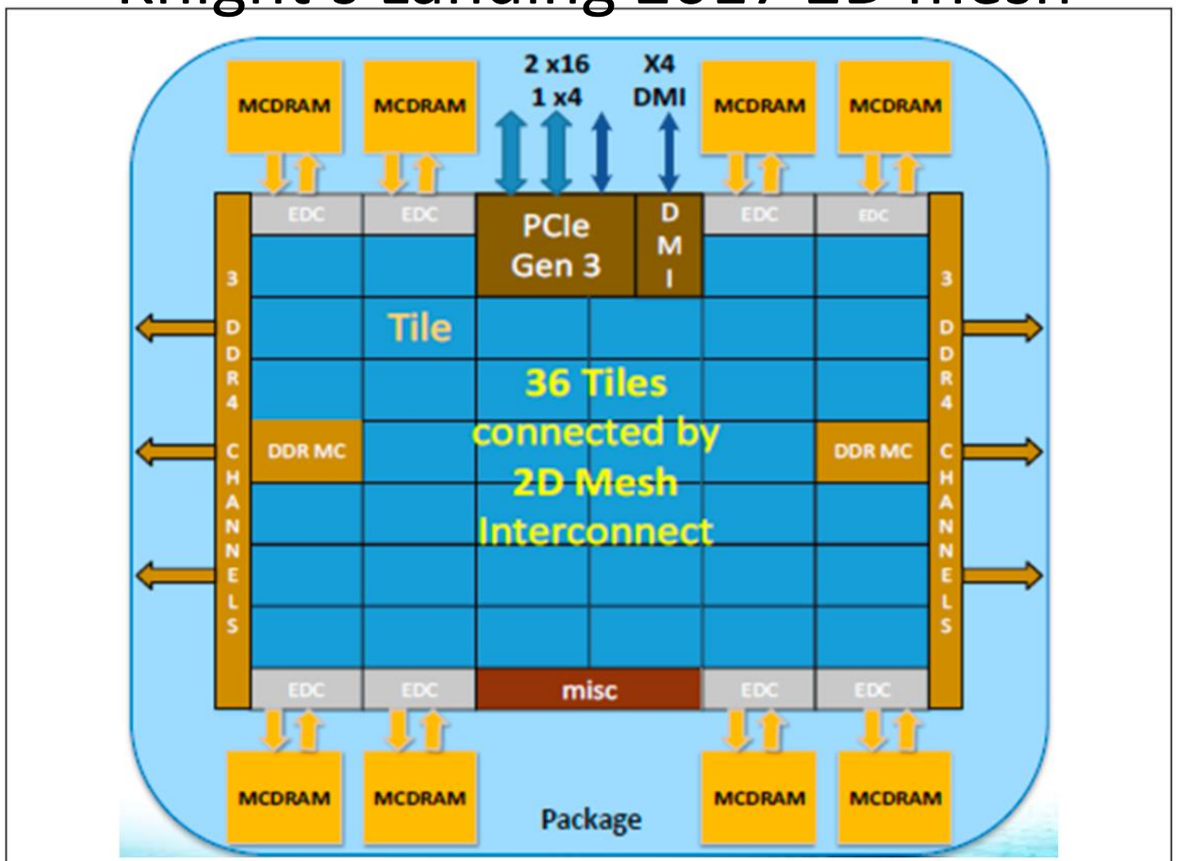
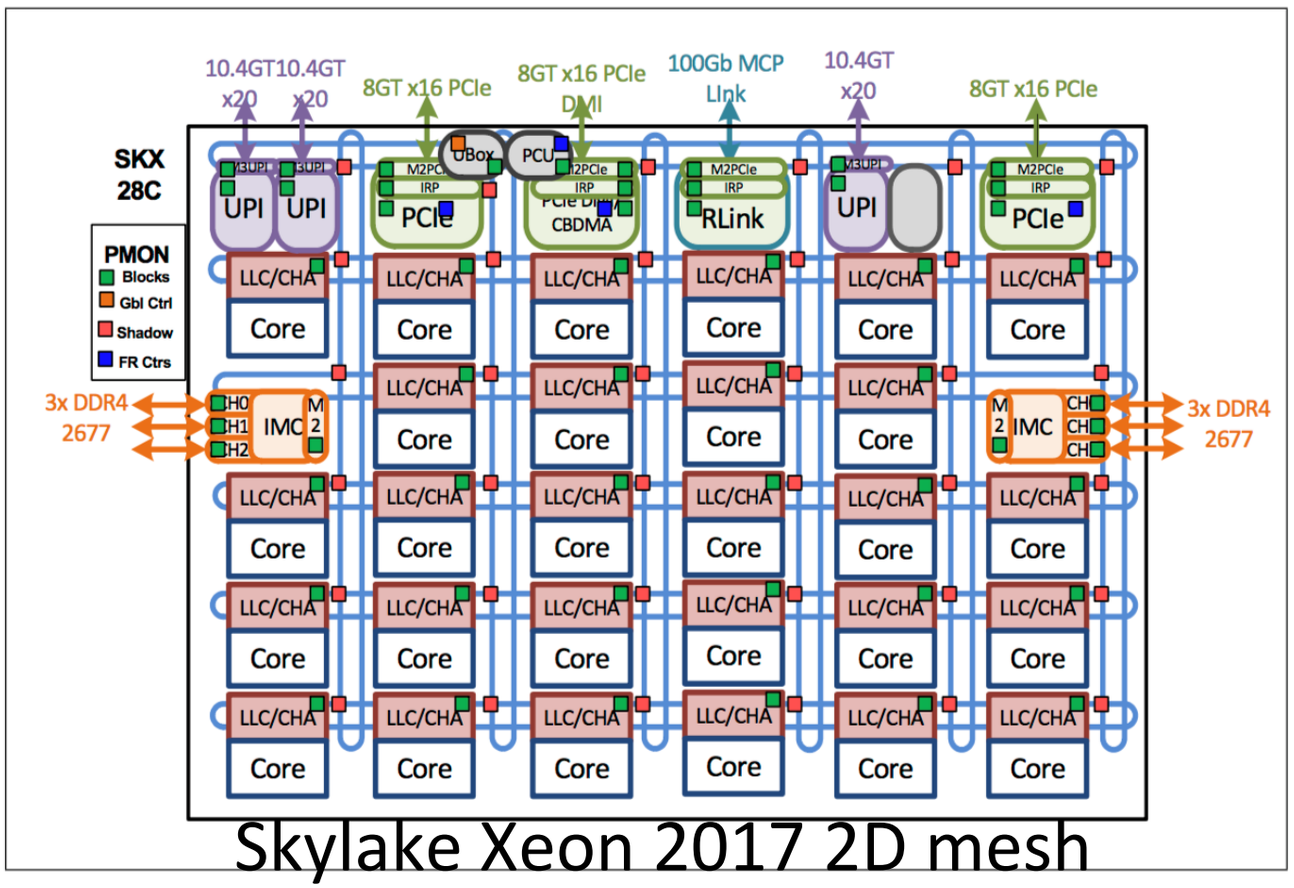
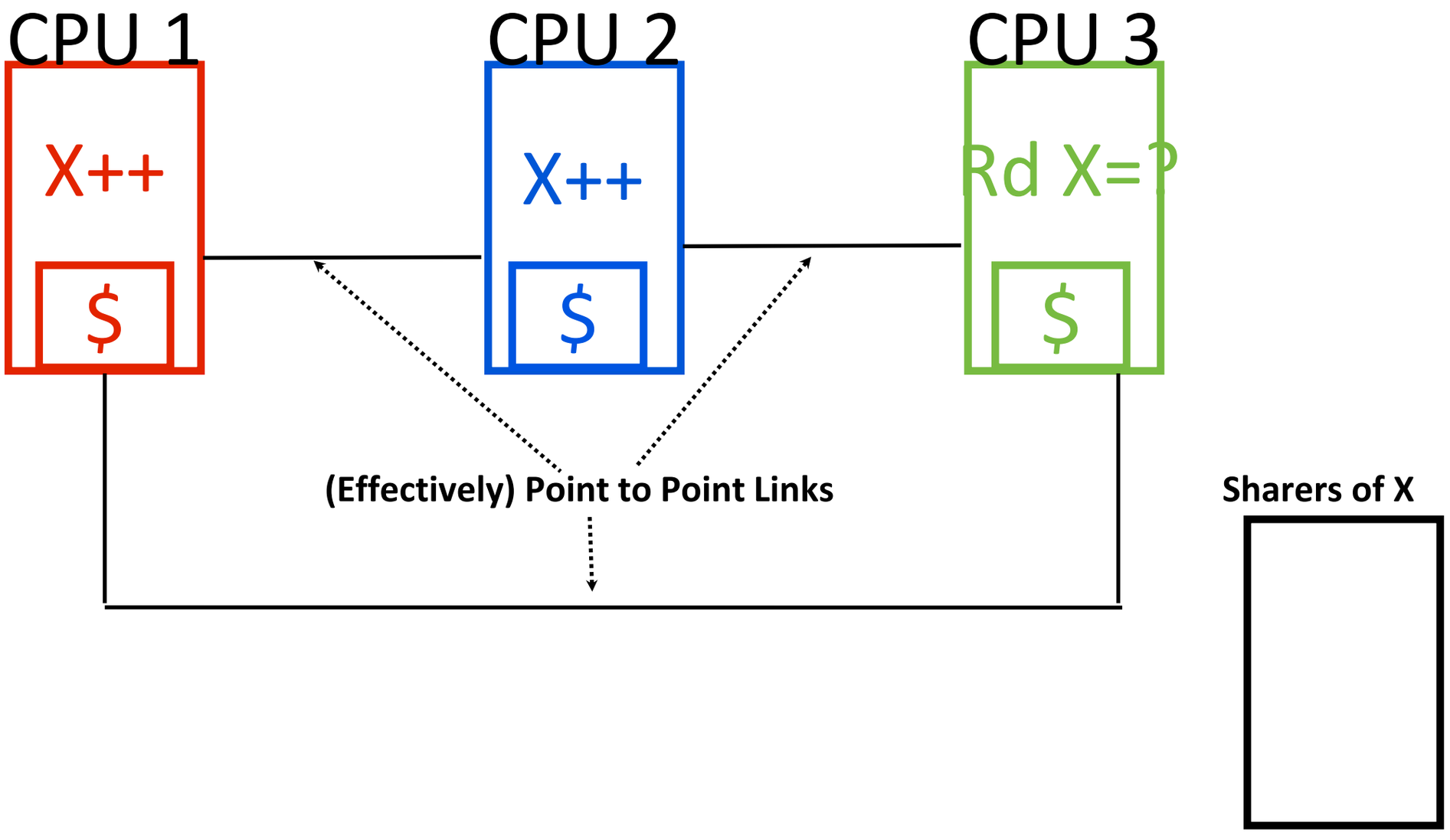


Figure 1-1. Intel® Xeon® Processor Scalable Memory Family - Block diagram for a 28C part



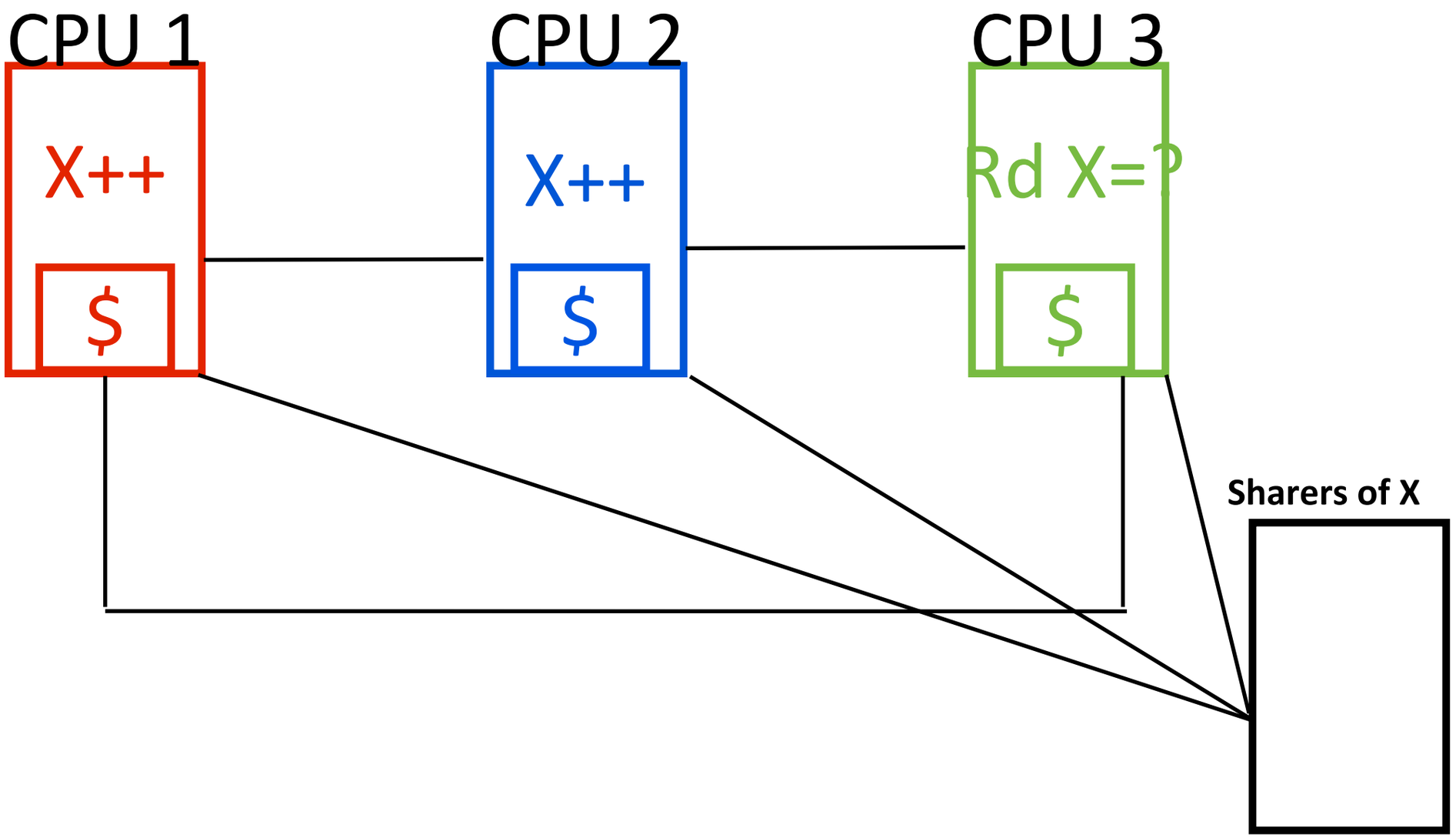
Skylake Xeon 2017 2D mesh

Implementing the Protocol



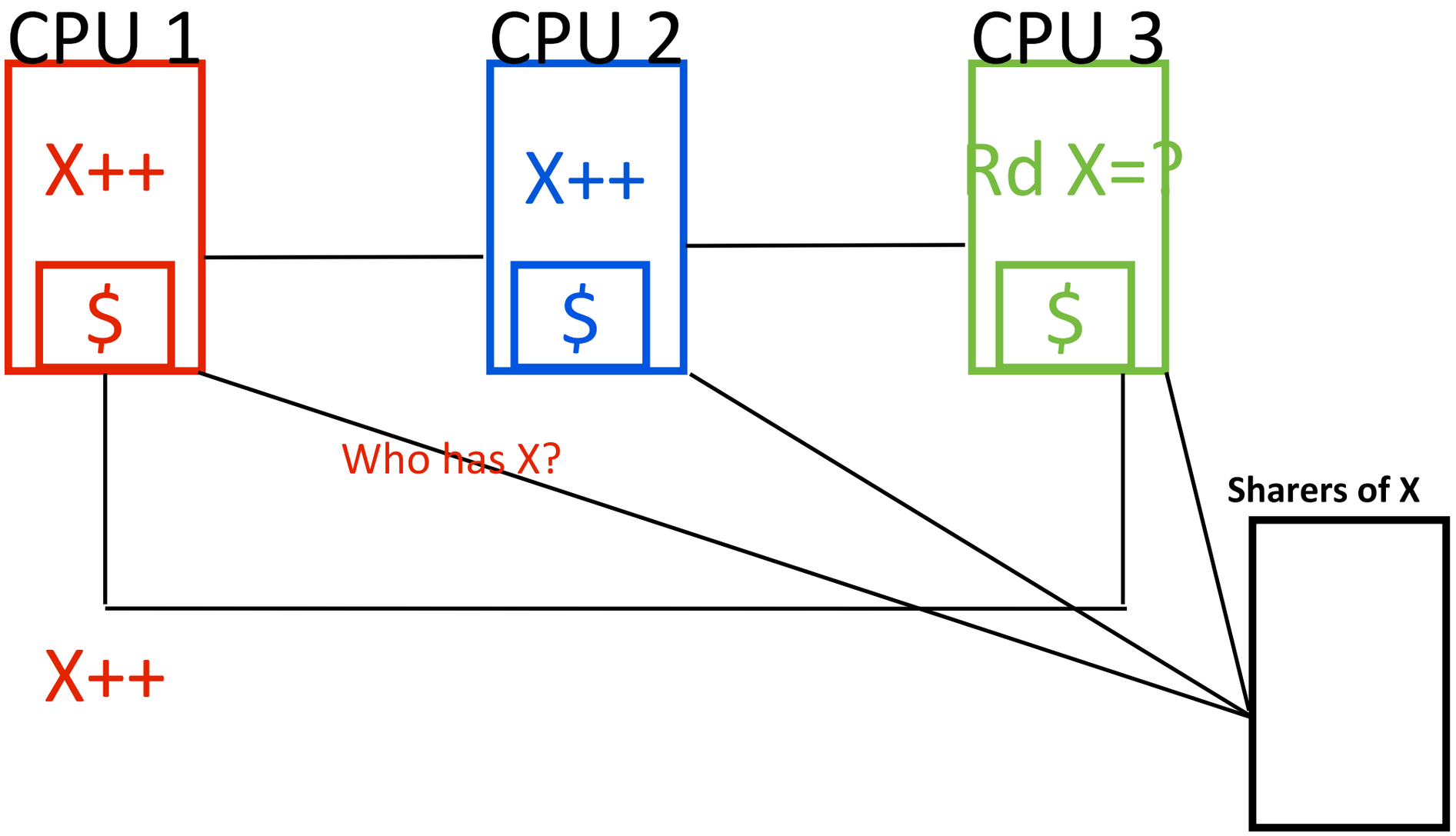
Directory-based

Implementing the Protocol



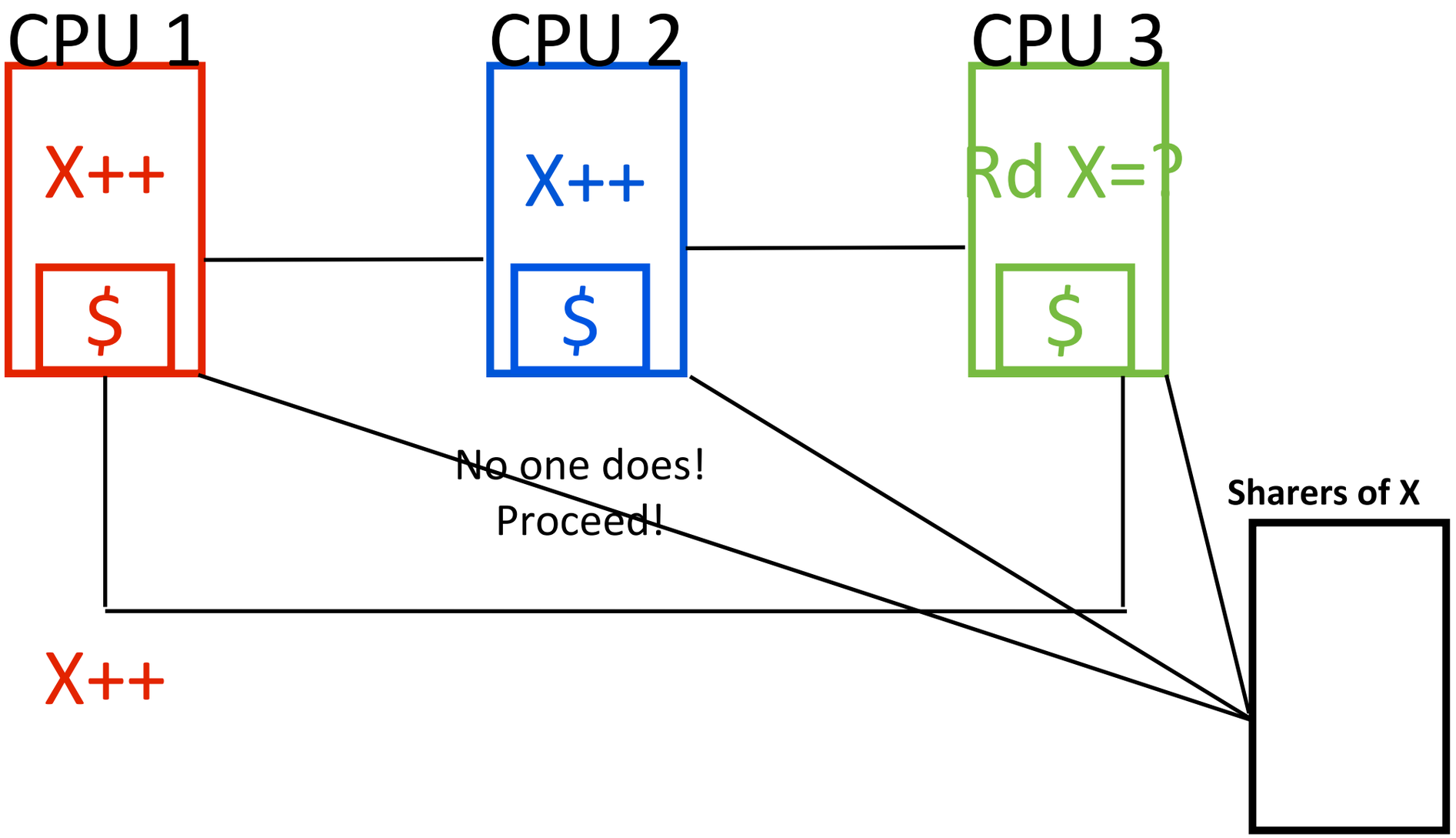
Directory-based

Implementing the Protocol



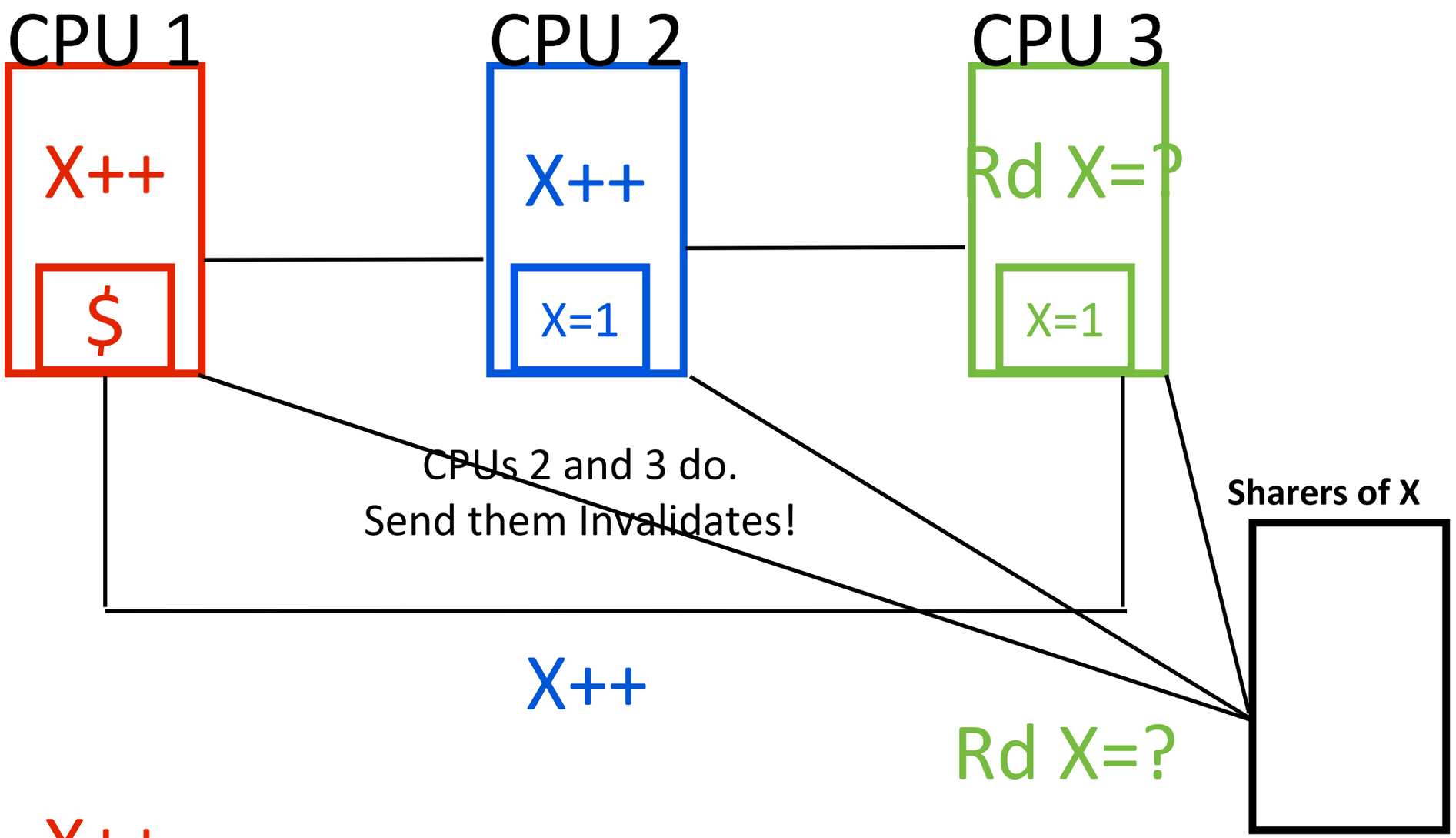
Directory-based

Implementing the Protocol



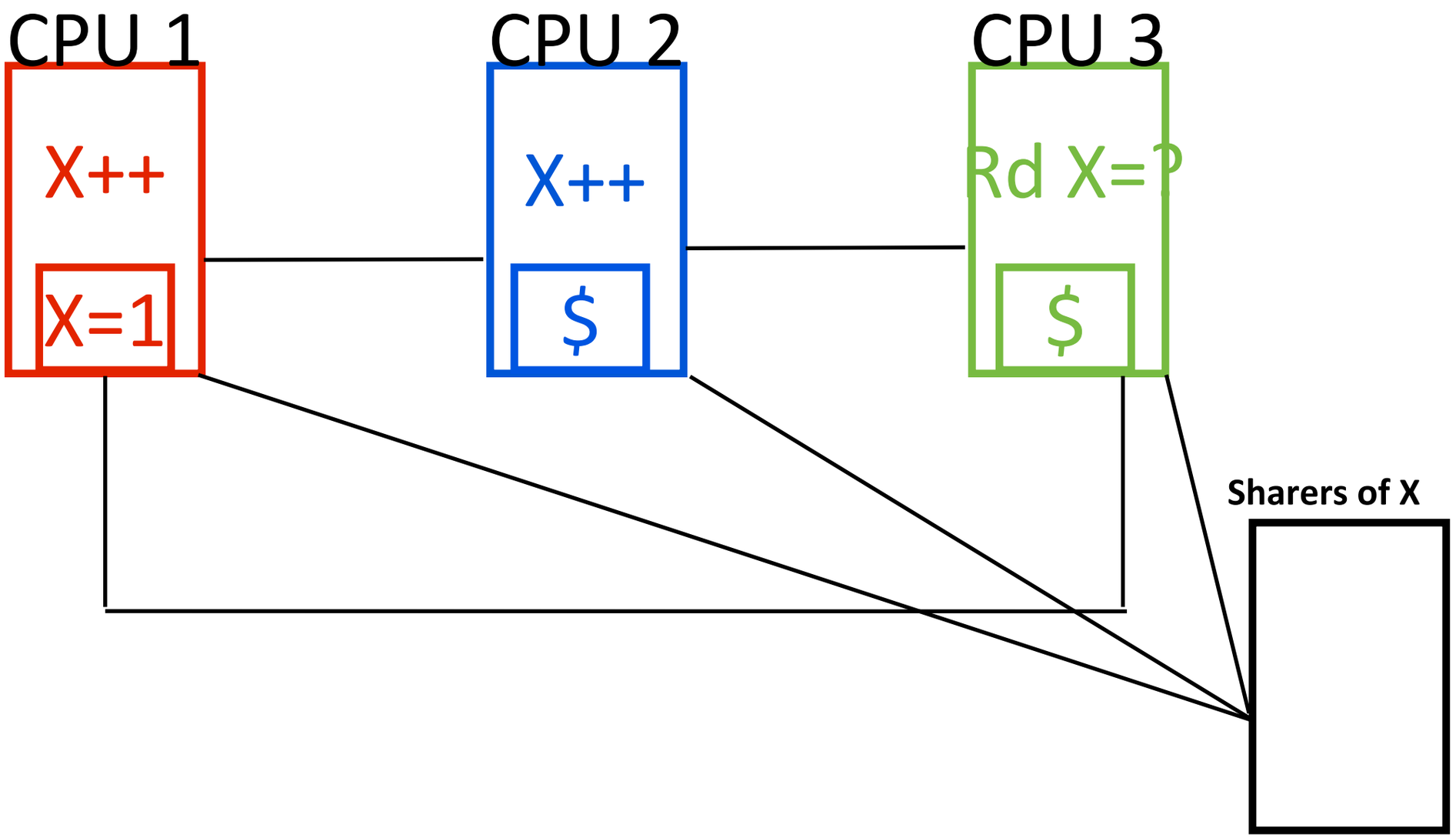
Directory-based

Implementing the Protocol



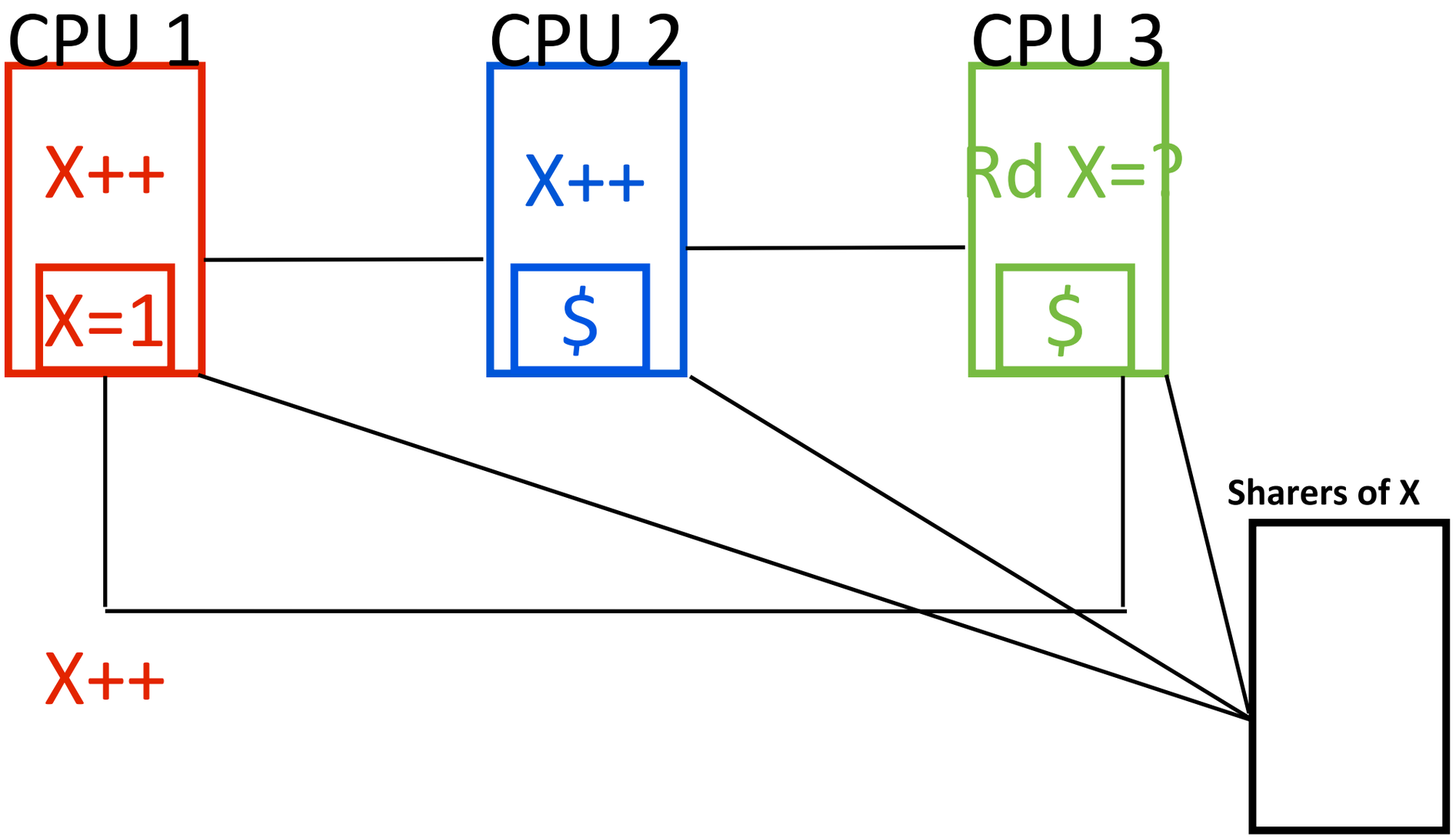
Directory-based

Implementing the Protocol



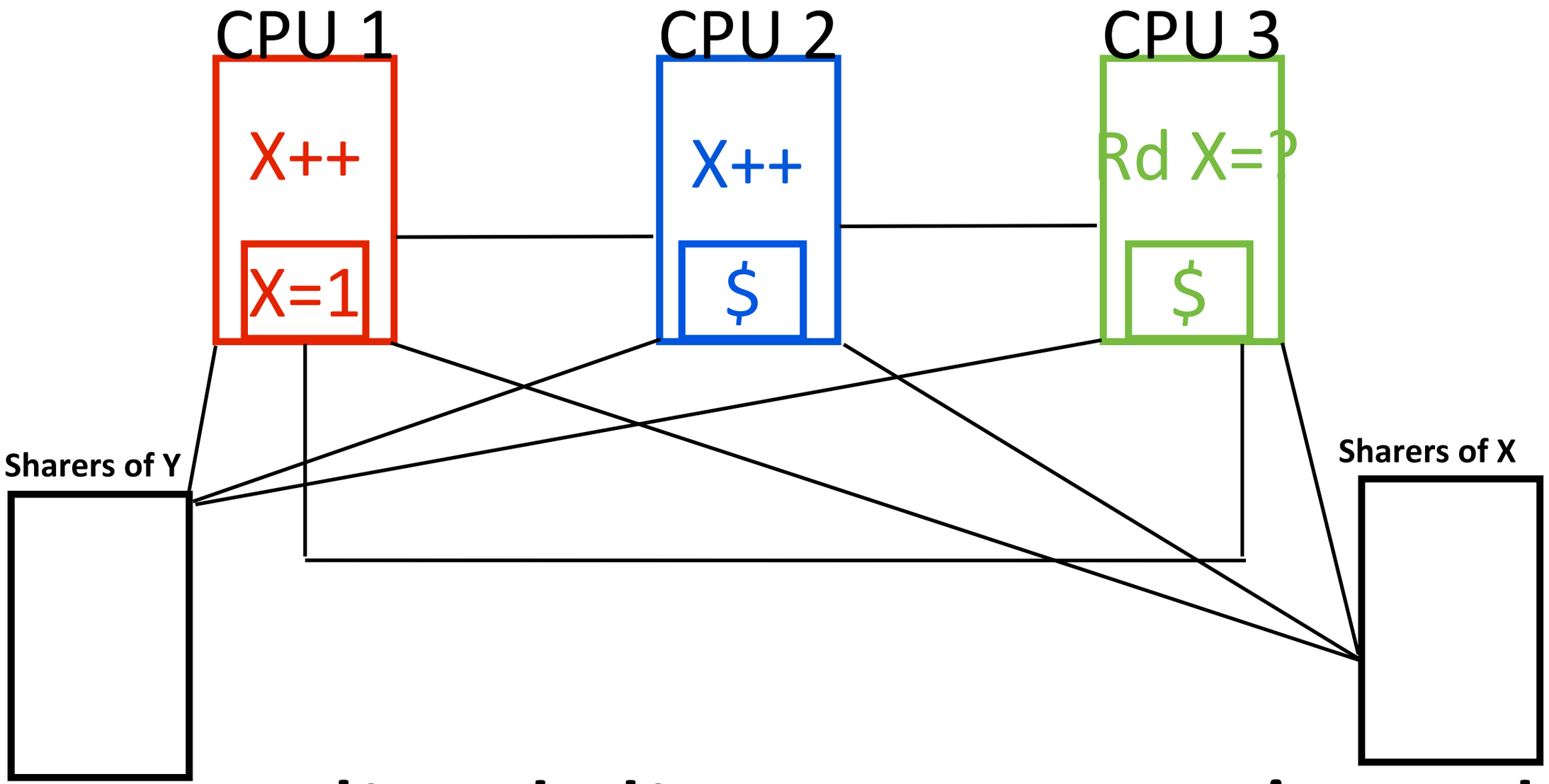
Benefit: No broadcast on shared bus

Implementing the Protocol



Drawbacks?

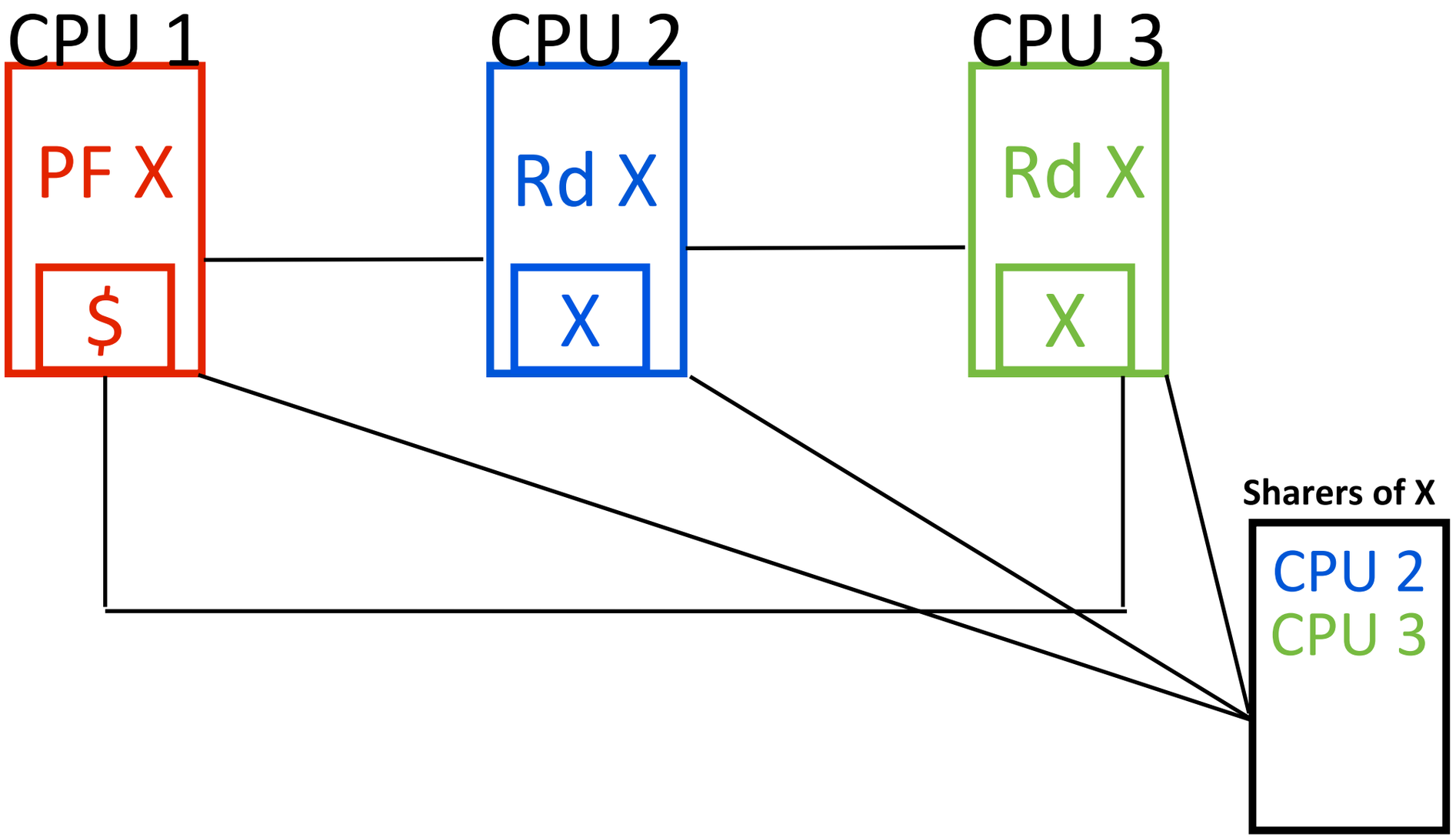
Implementing the Protocol



Centralized directory won’t scale

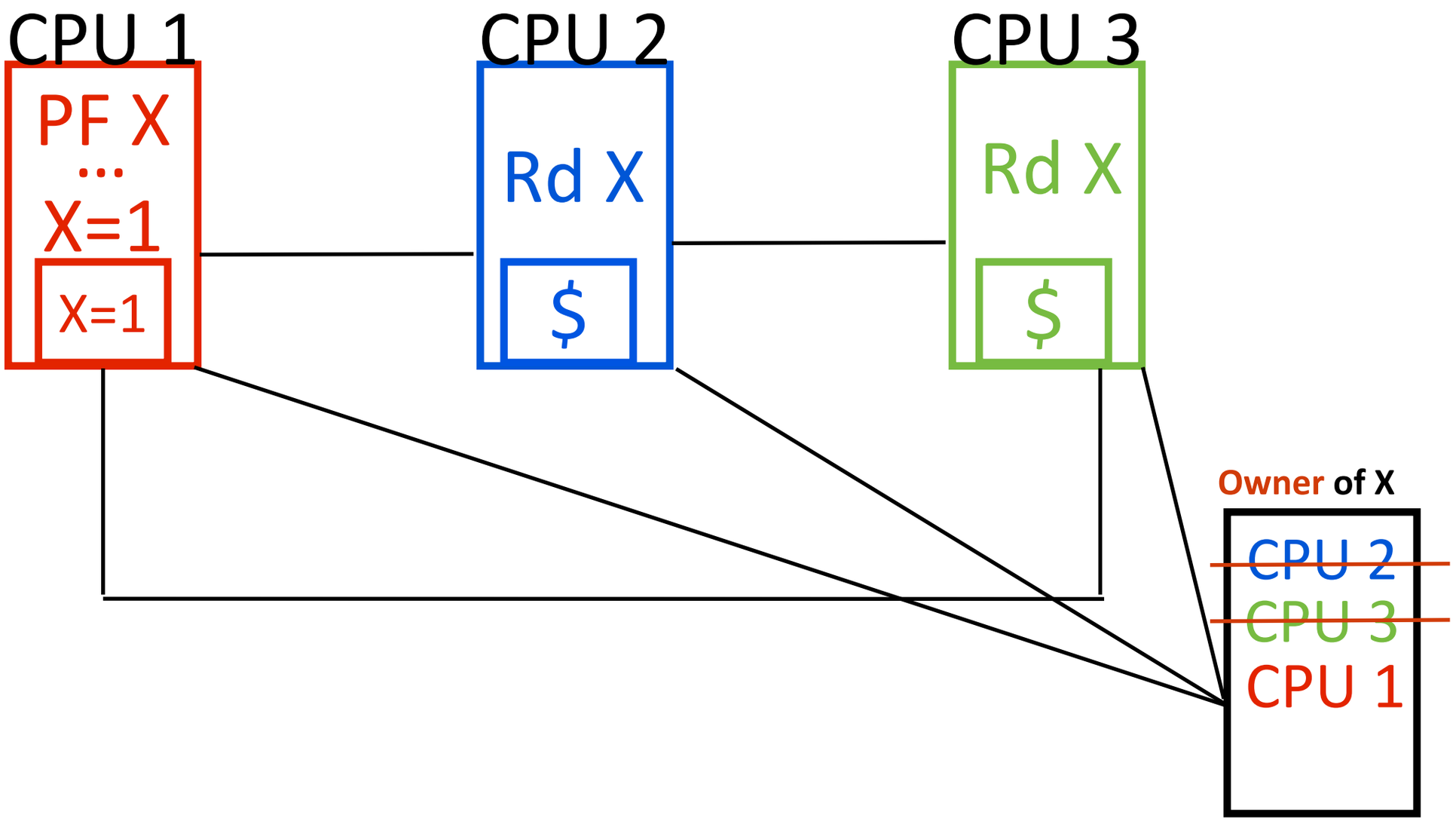
(In Practice: Distribute Directory)

Optimization: Non-binding Prefetch



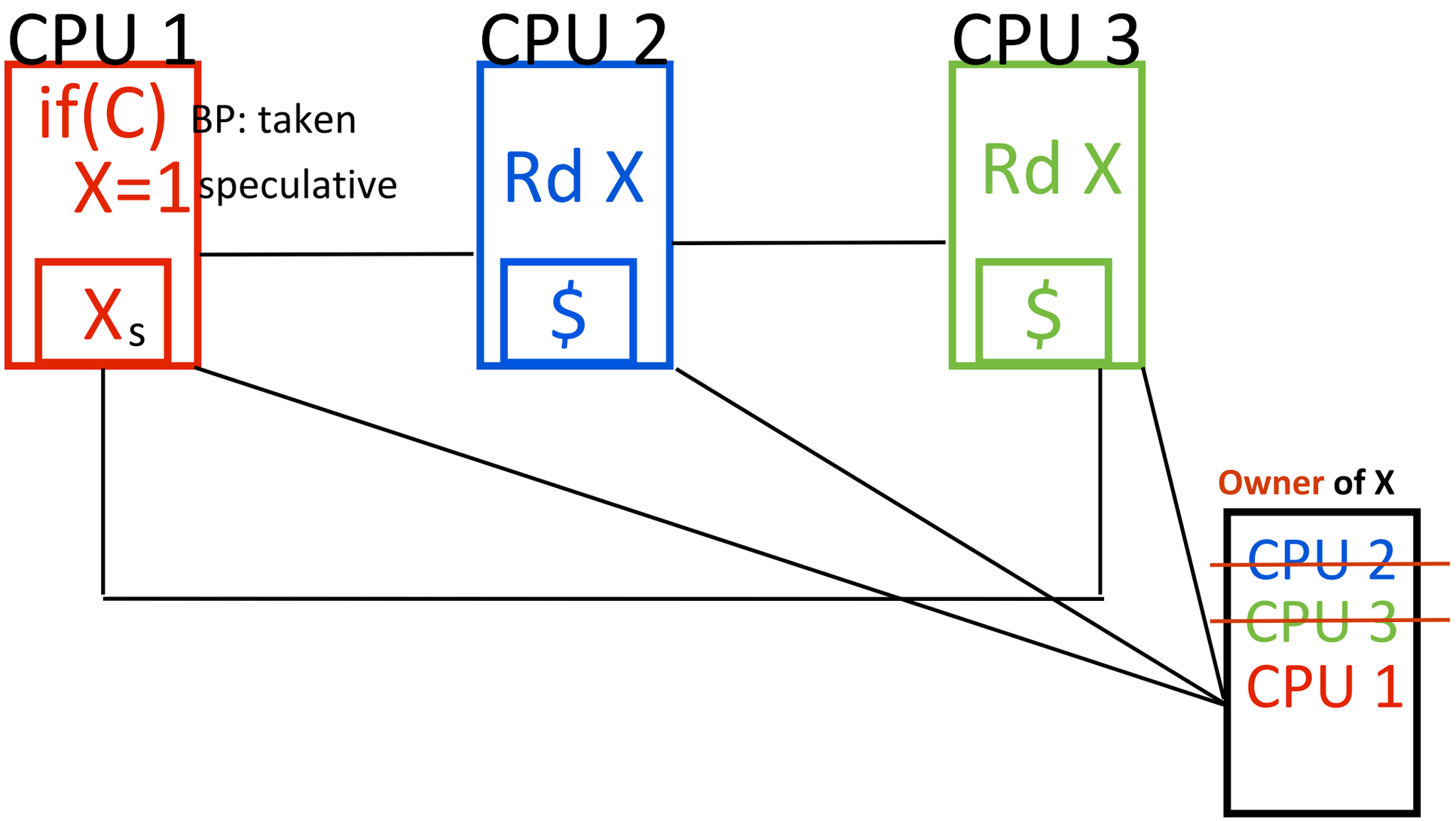
Prefetch instruction preemptively changes coherence state

Optimization: Non-binding Prefetch



Benefit?

Optimization: Speculation



Speculative operations that squash behave like non-binding pre-fetch

“computers execute operations in a **different order** than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor **does not guarantee the correct execution of the entire program**.”

Excerpt from “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program”
LESLIE LAMPORT, 1979

Memory Consistency

Memory Consistency Model

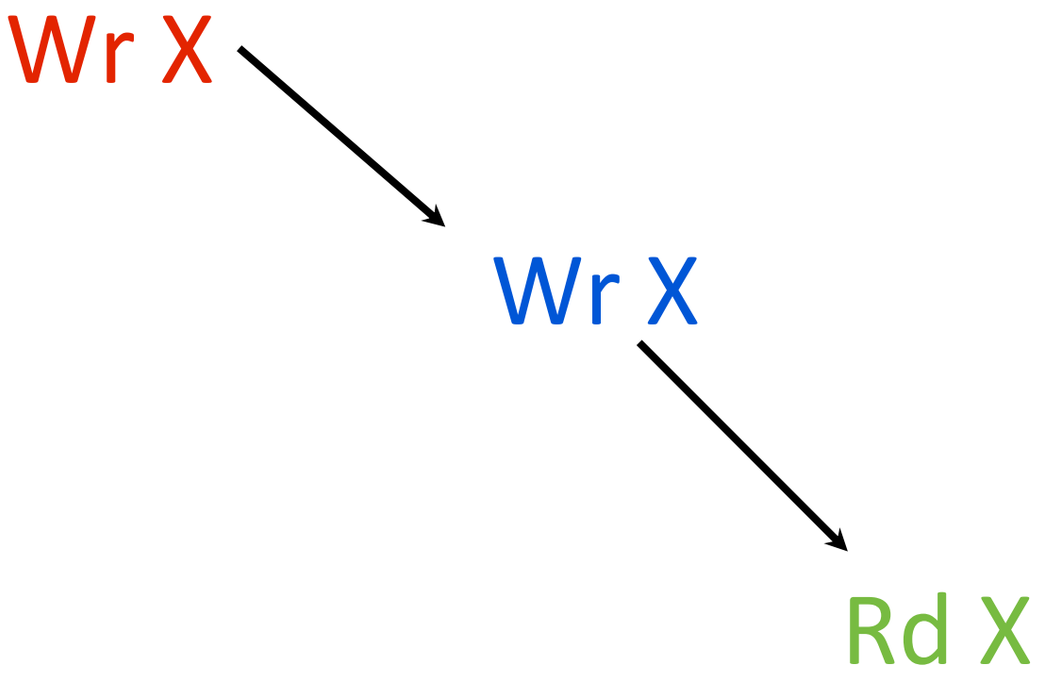
Informal Definition:

"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"

Review: Coherence

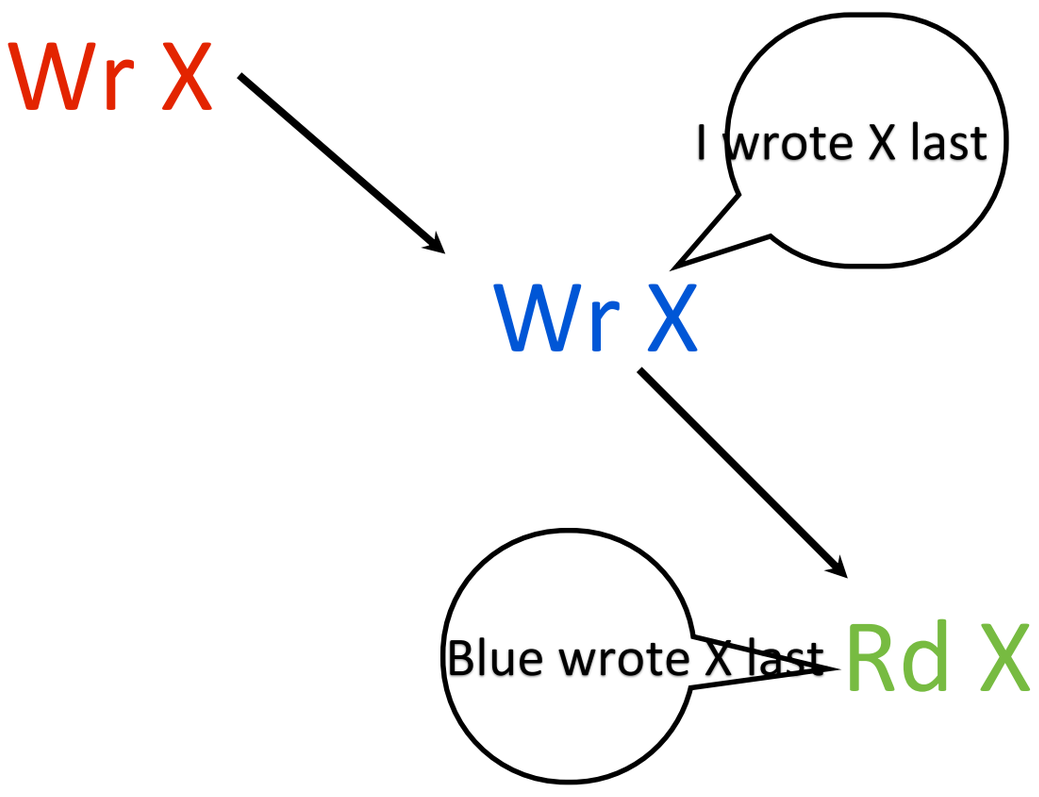


2 Invariants:

1) “One Writer or One or More Readers”

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

Review: Coherence



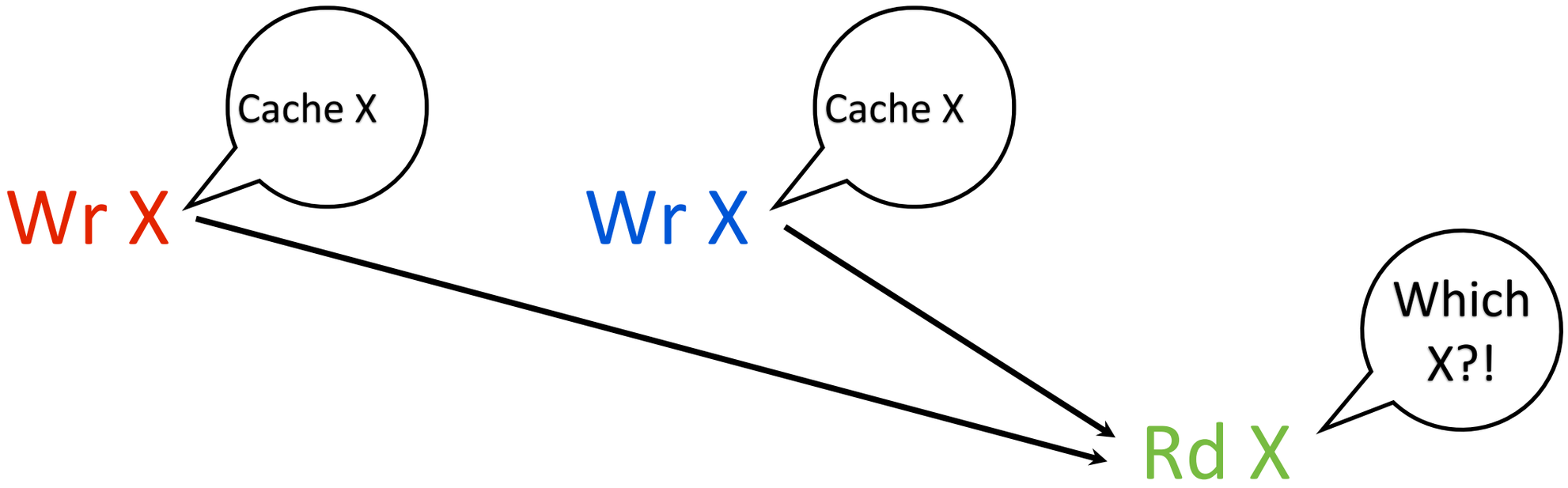
2 Invariants:

1) “One Writer or One or More Readers”

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

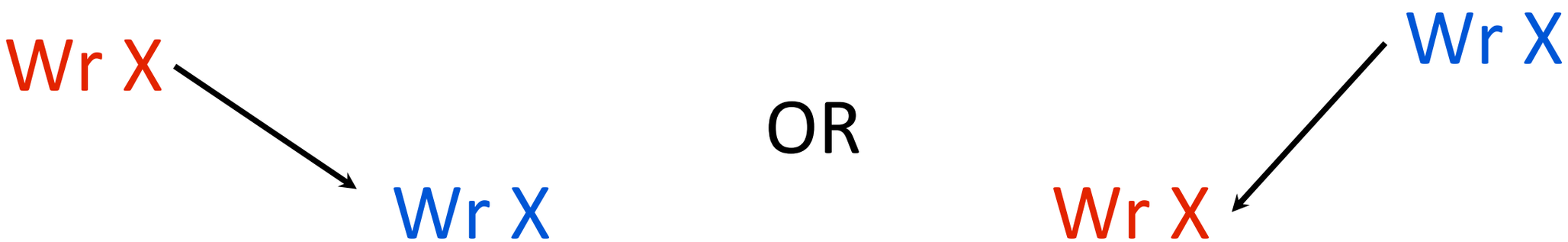
Without Coherence

(The coherence invariants prevent this from happening)



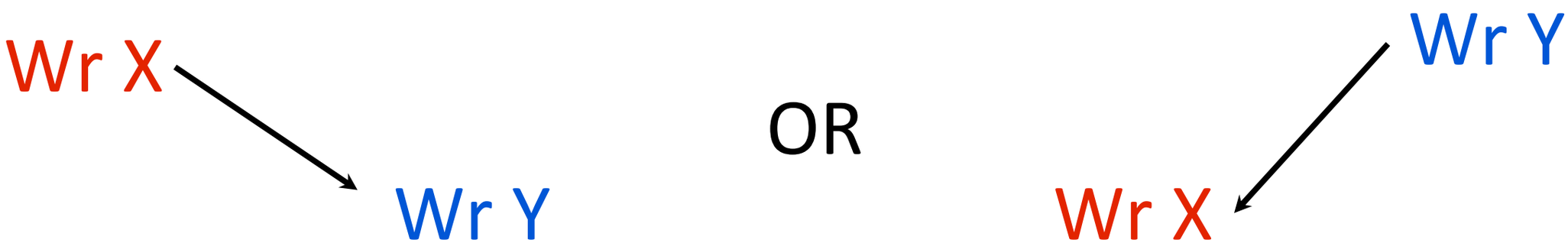
Processors can’t decide who wrote last.
Green is hosed.

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)

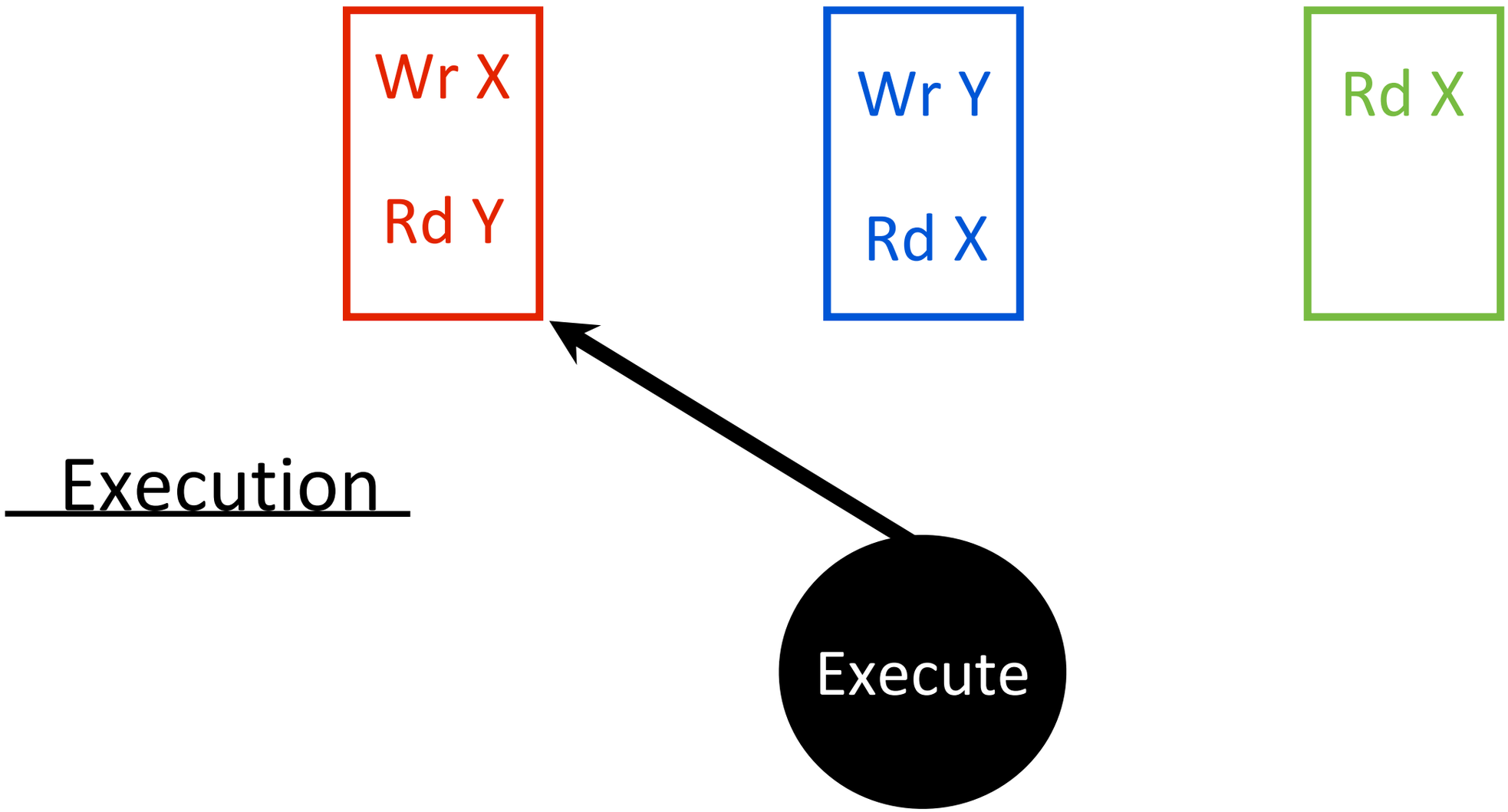
The simplest, most intuitive memory consistency model

Two Invariants to SC:

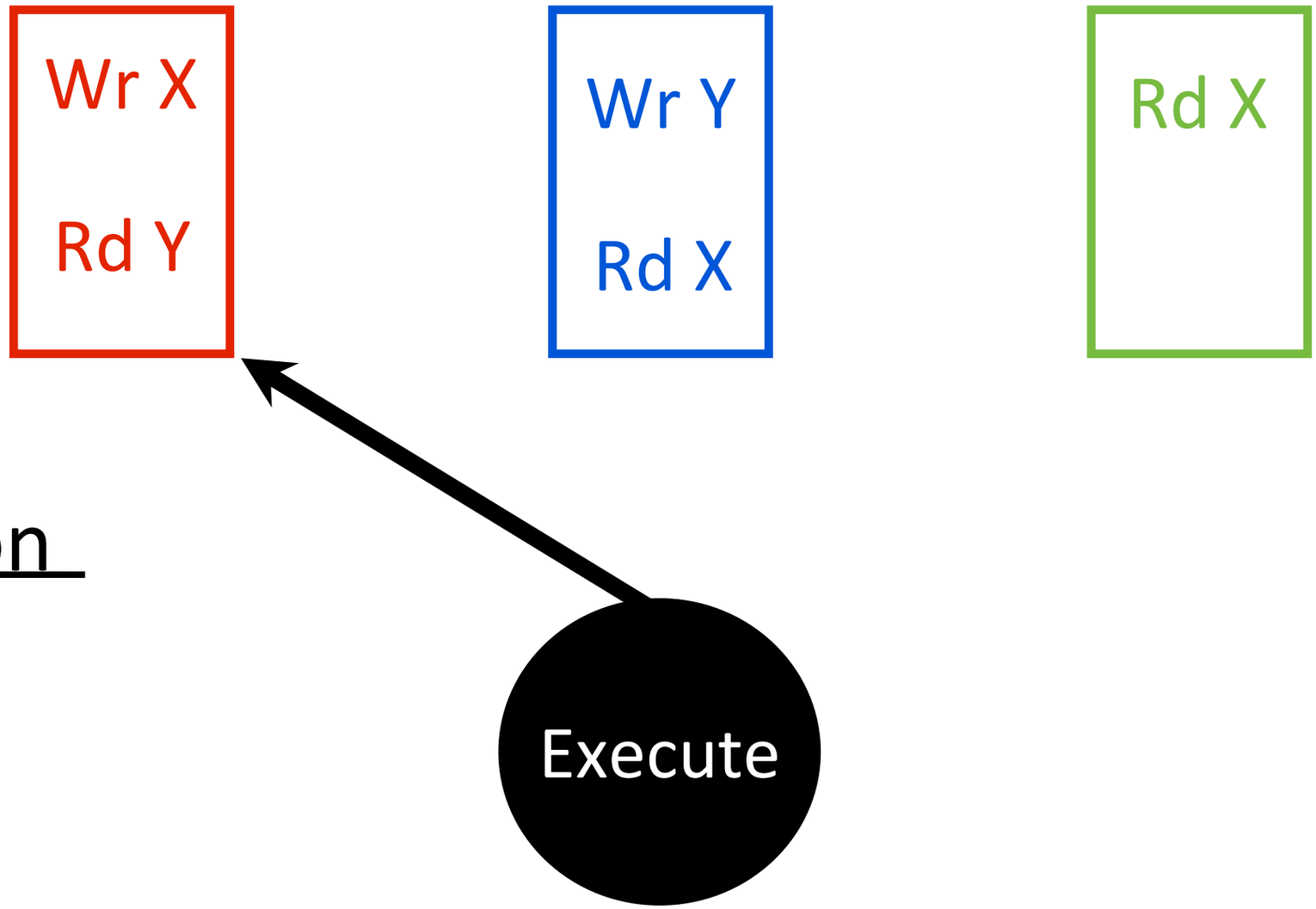
Instructions are executed in program order

All processors agree on a total order of executed instructions

The SC “Switch”



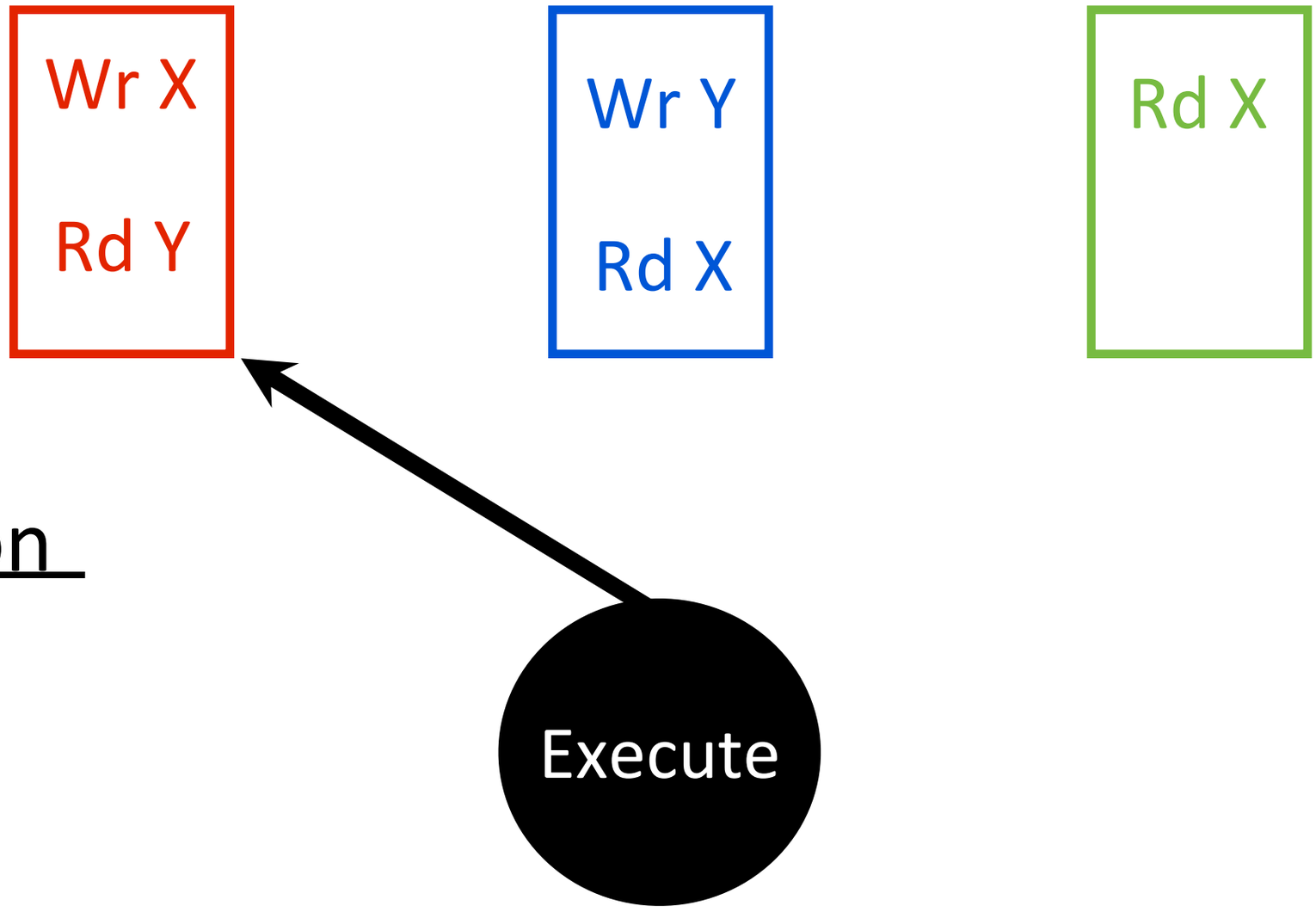
The SC “Switch”



Execution

Wr X

The SC “Switch”

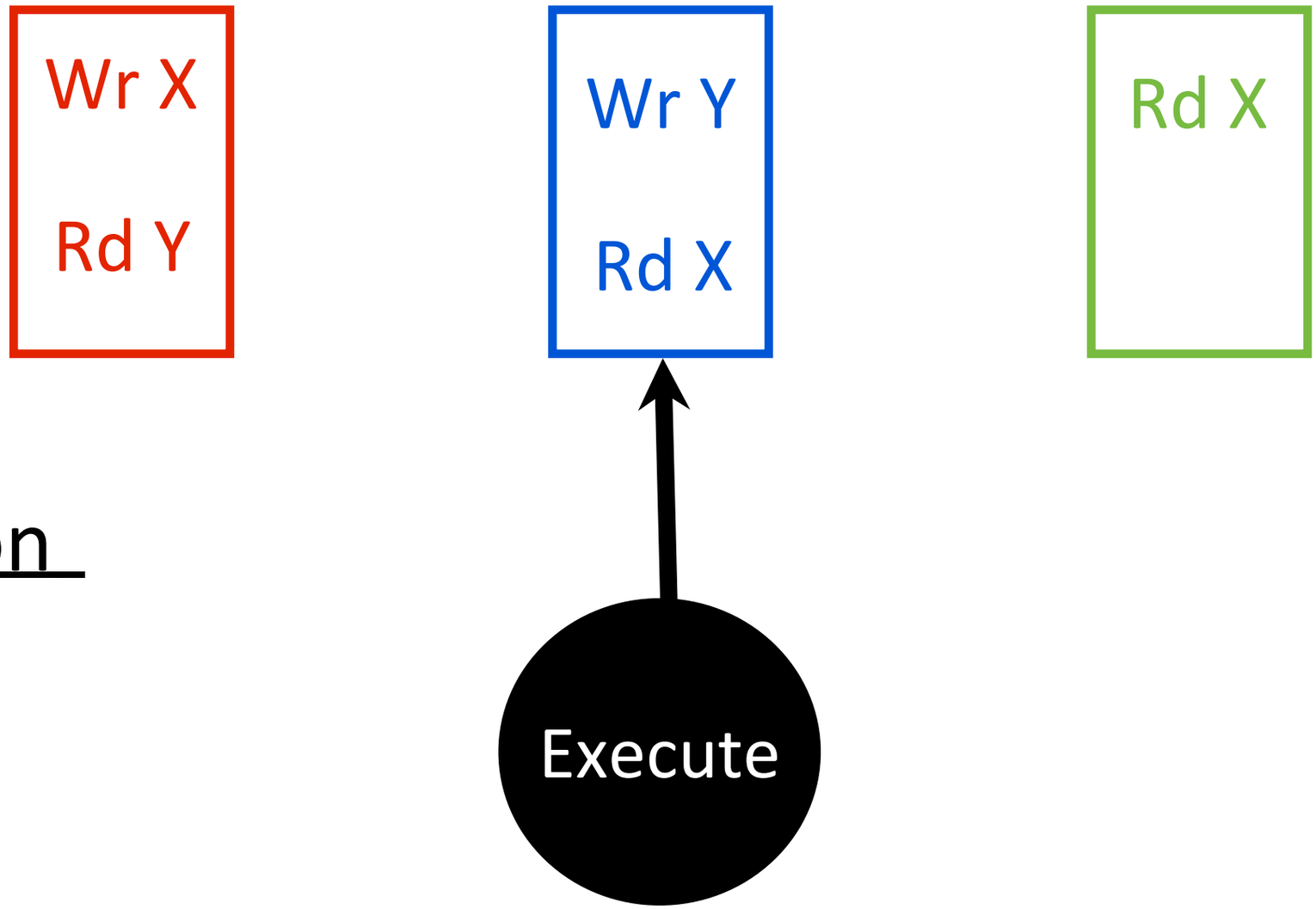


Execution

Wr X

Rd Y

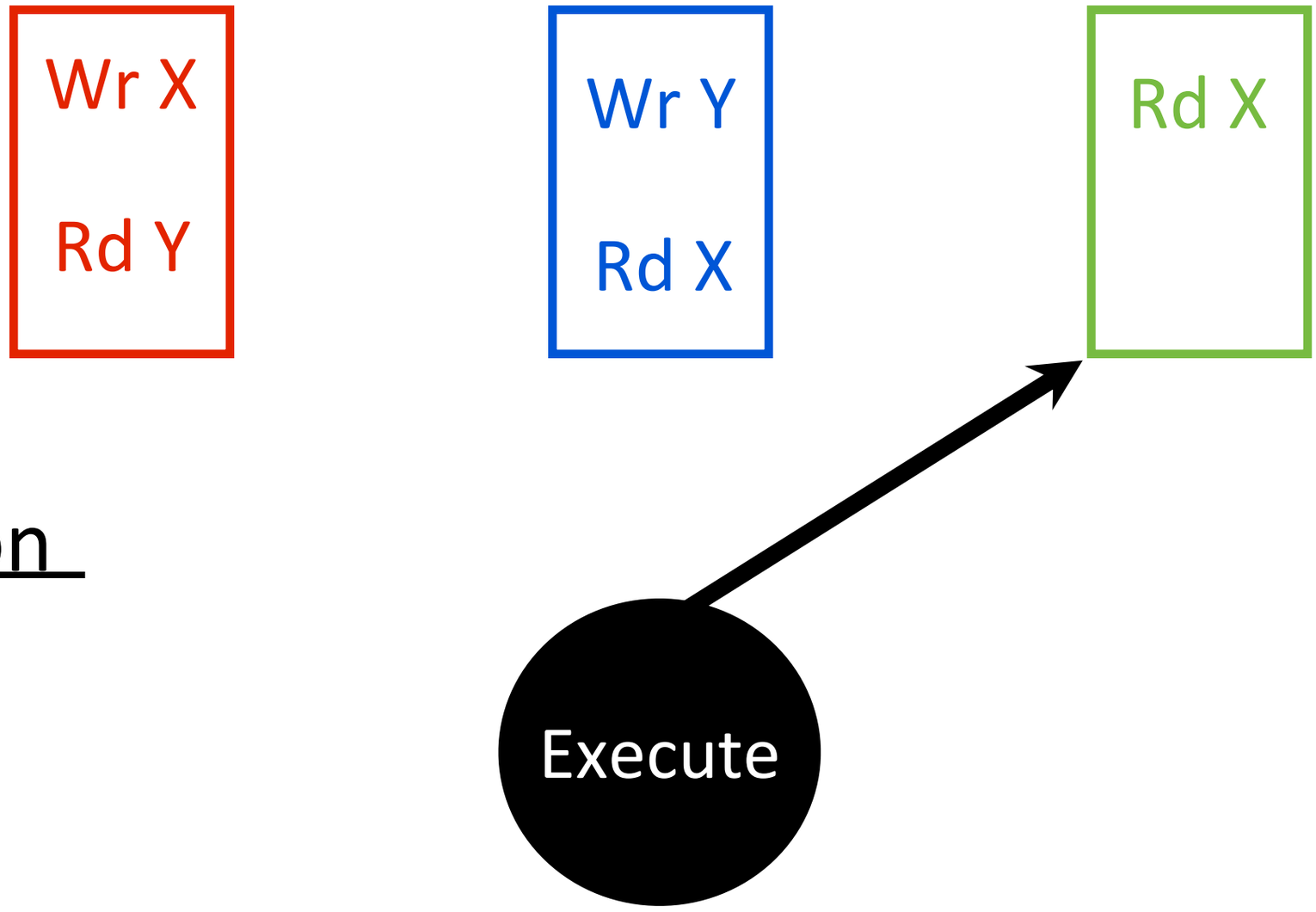
The SC “Switch”



Execution

Wr X
Rd Y
Wr Y

The SC “Switch”



Execution

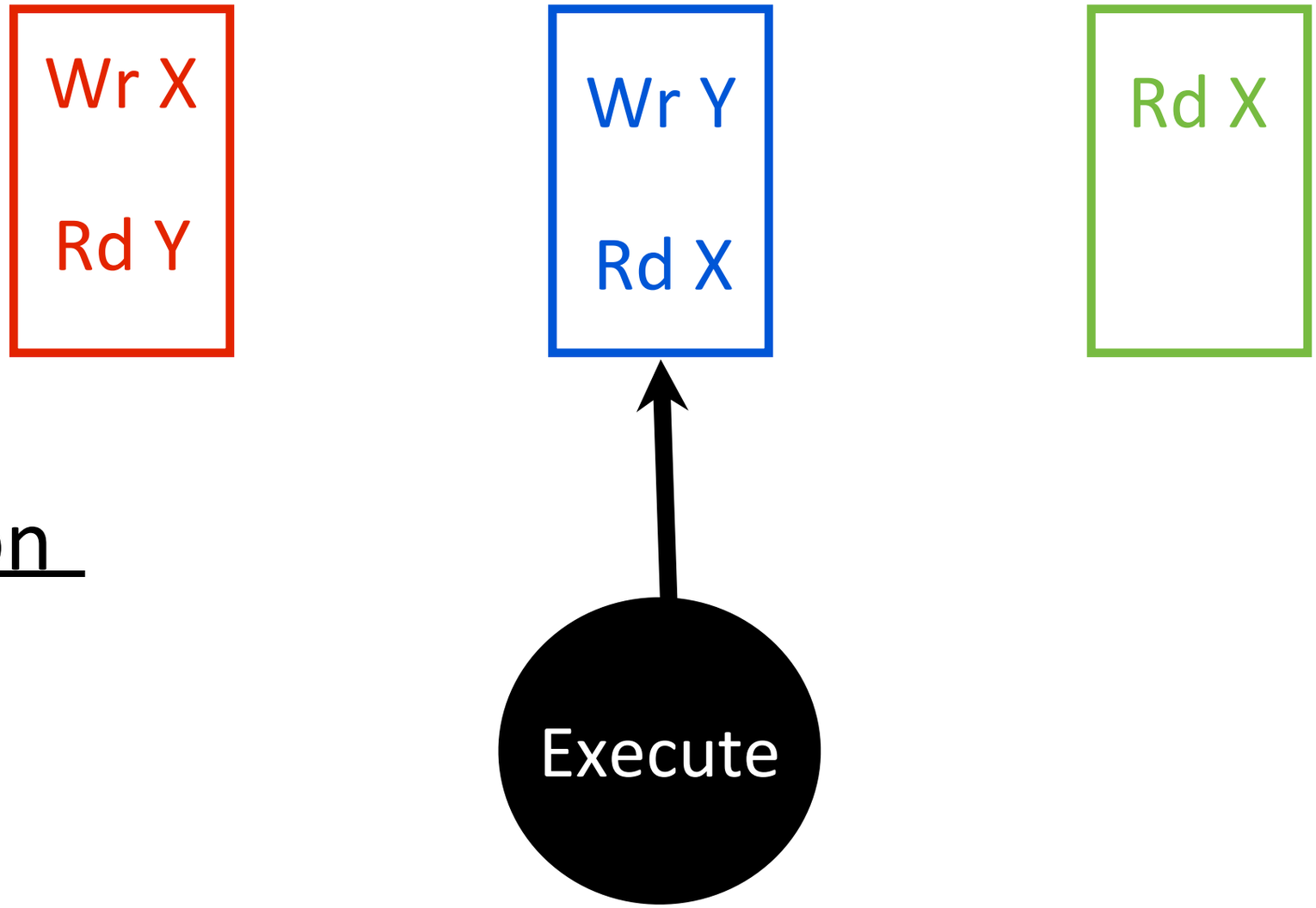
Wr X

Rd Y

Wr Y

Rd X

The SC “Switch”



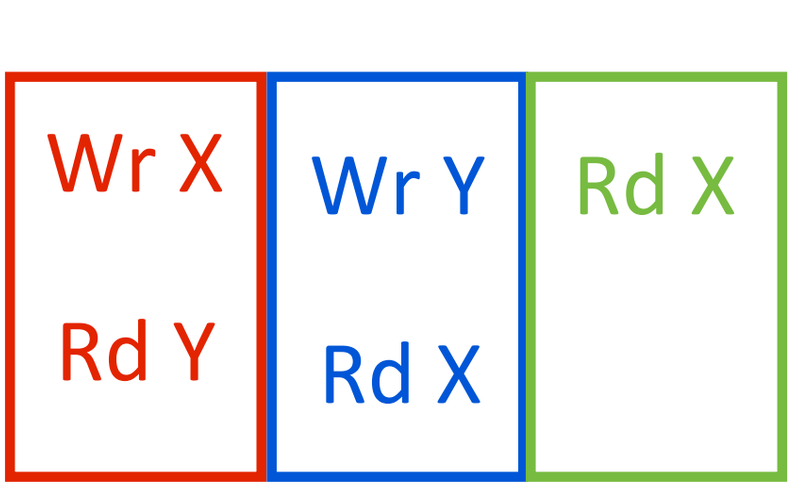
Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X

Why is SC Important?

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

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



Intuitive (SC)	Weird (not SC)
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)

Why are Instructions Reordered?

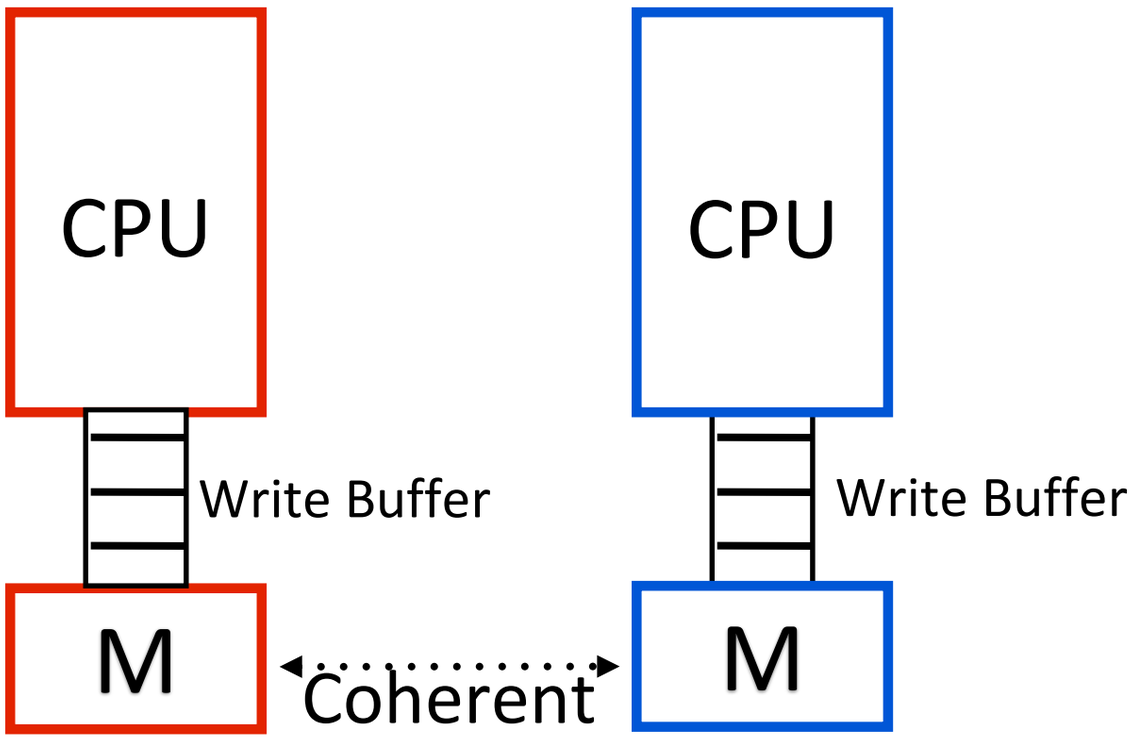
And when does it matter?

Why are Instructions Reordered?

Optimization.

Elsewhere?

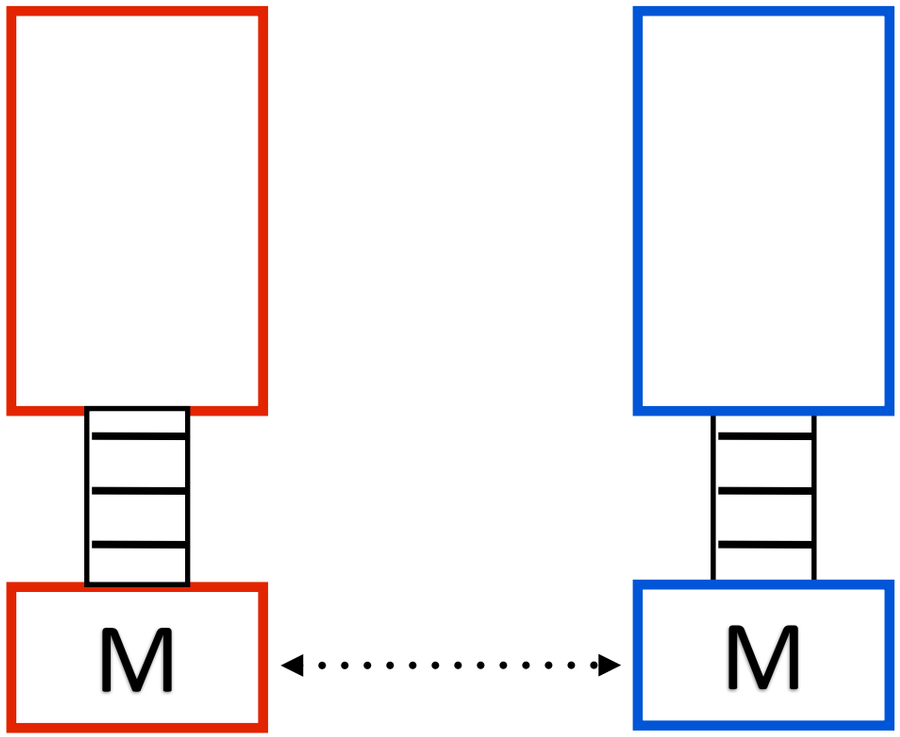
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

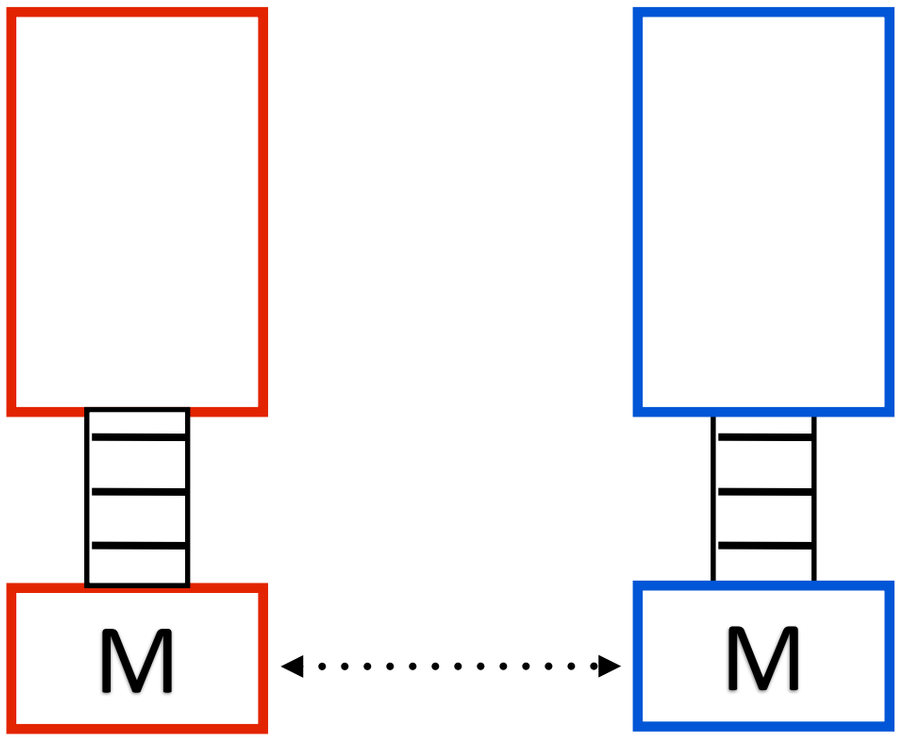
Initially X == Y == 0

X=1 Y=1

r1=Y r2=X

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

Reordering #1: Write Buffers



Program

Initially X == Y == 0

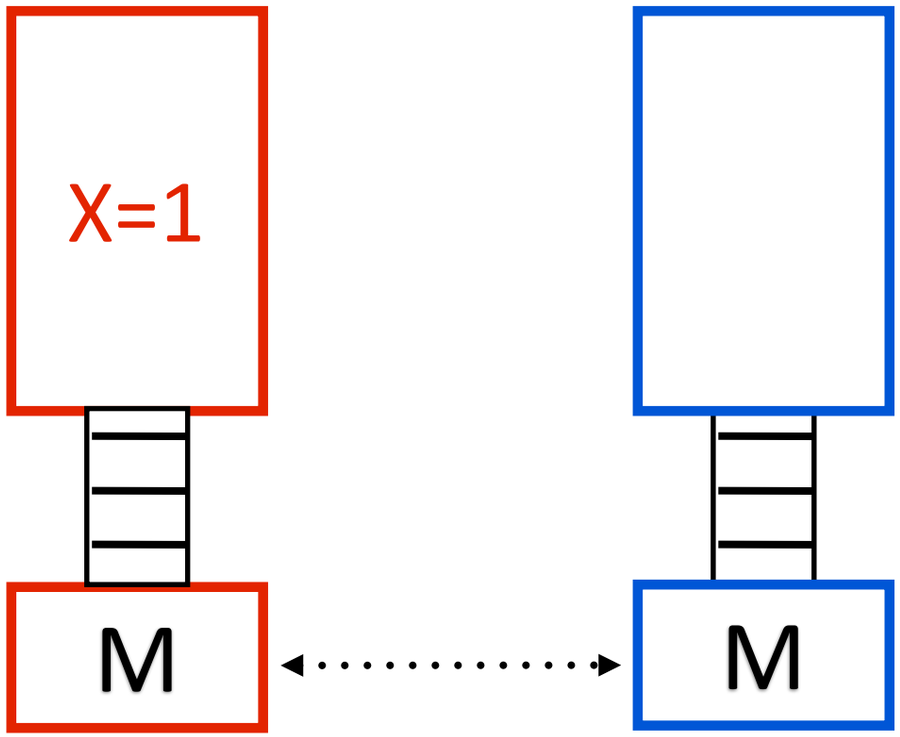
X=1 Y=1

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

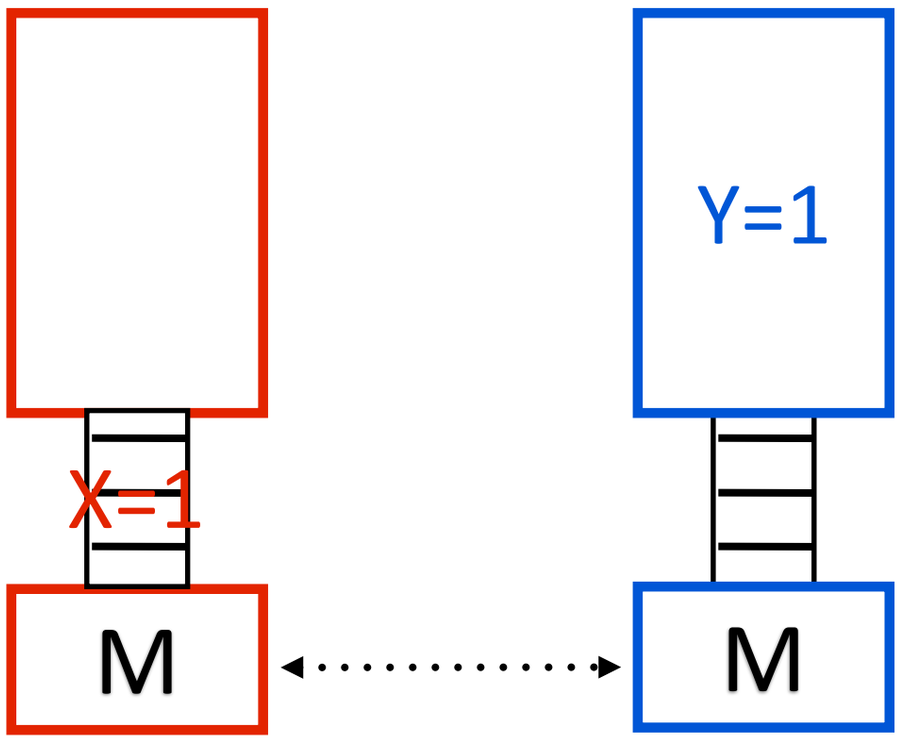
Y=1

r1=Y

r2=X

Execution

Reordering #1: Write Buffers



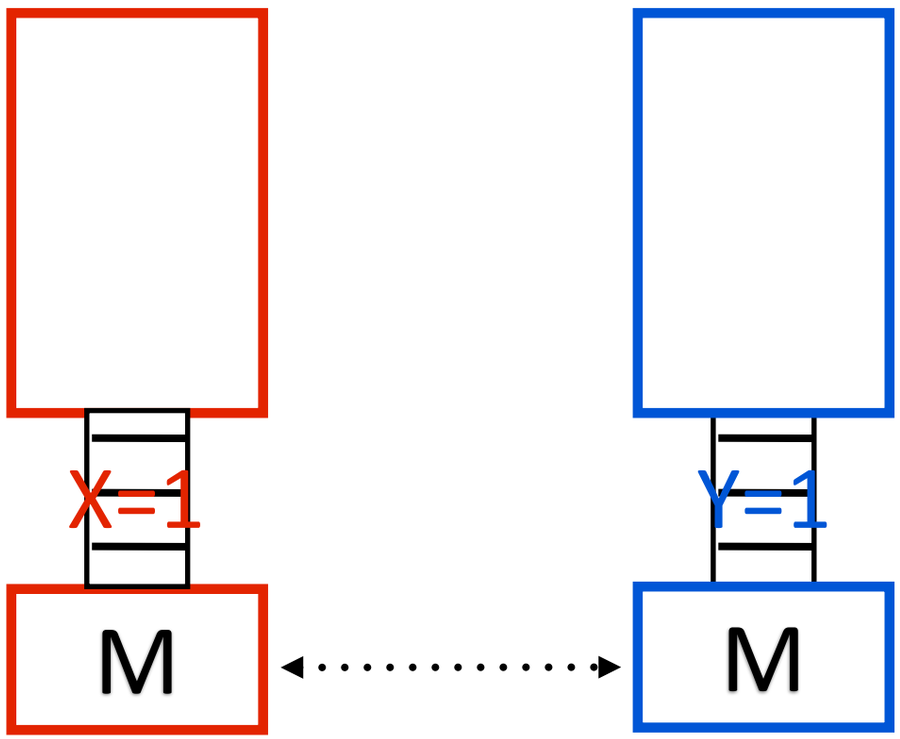
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



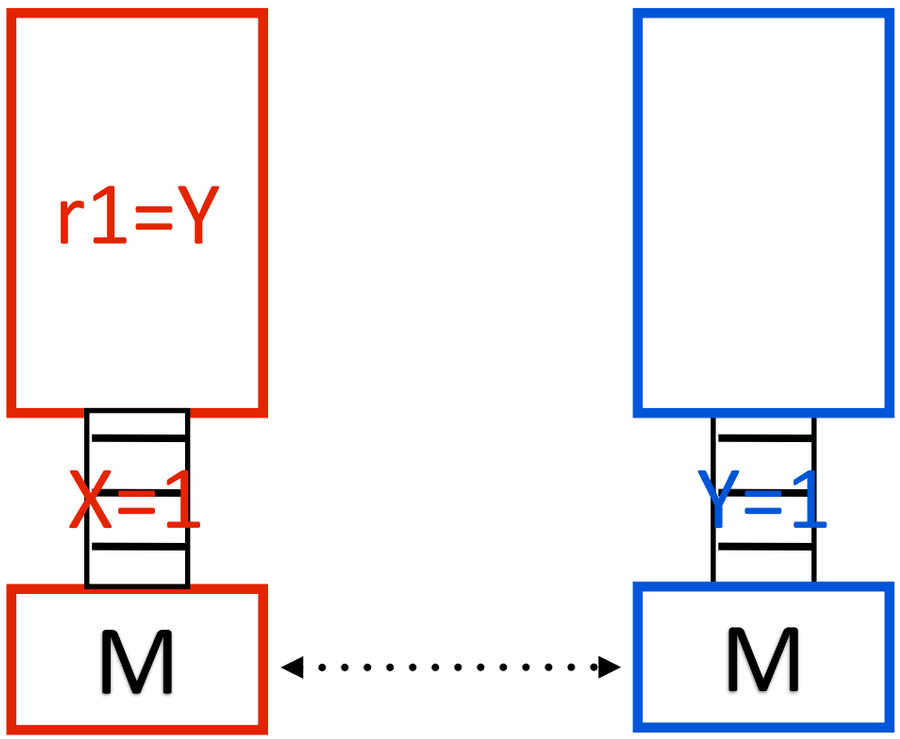
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



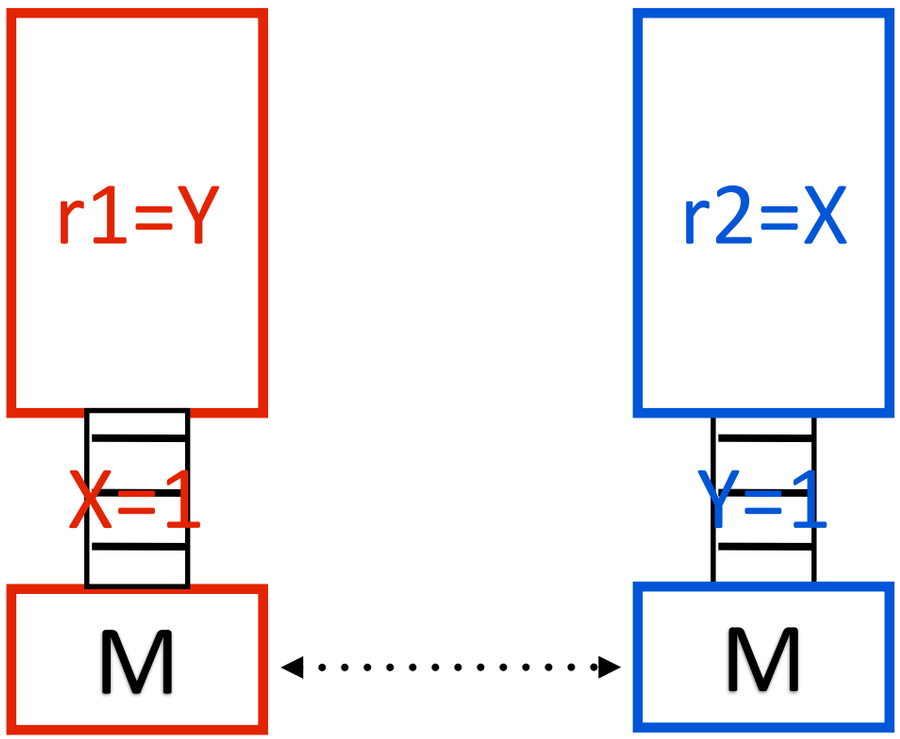
Program

Initially X == Y == 0

r2=X

Execution

Reordering #1: Write Buffers

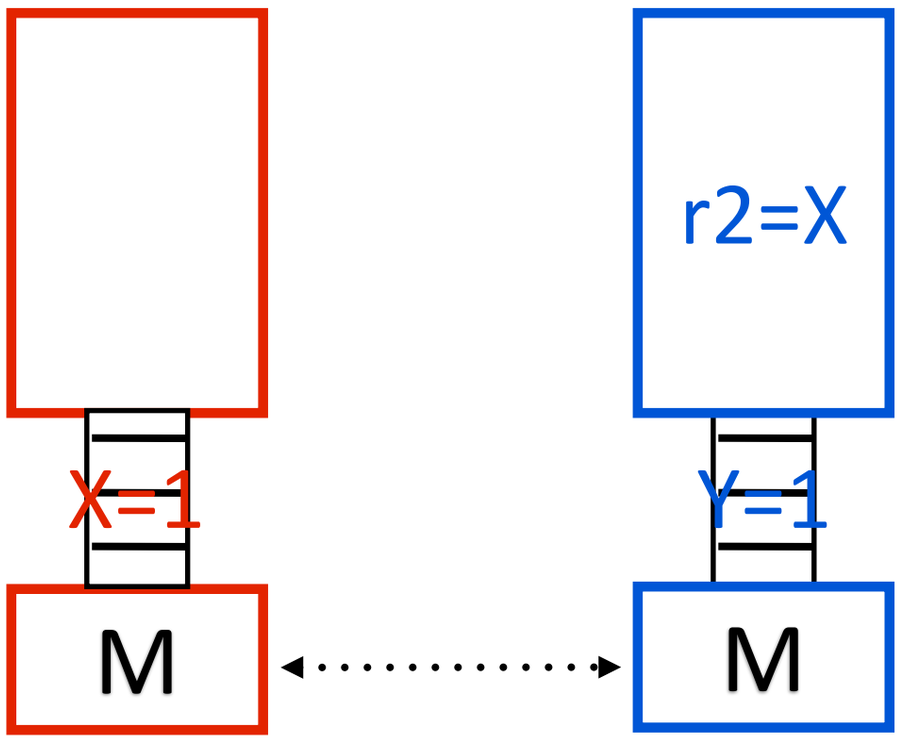


Program

Initially X == Y == 0

Execution

Reordering #1: Write Buffers



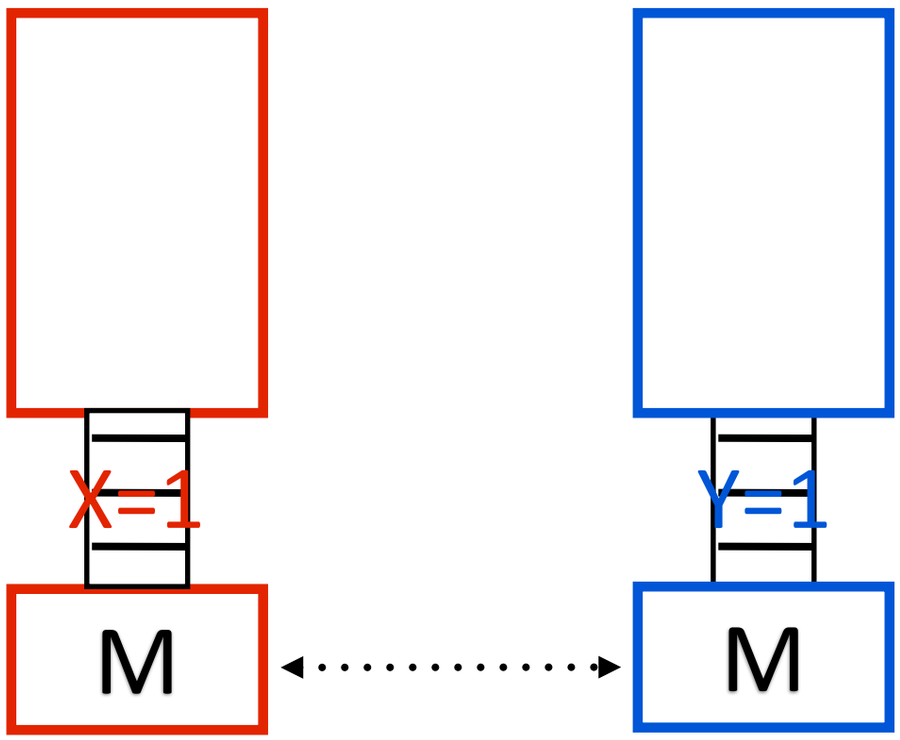
Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

Reordering #1: Write Buffers



Program

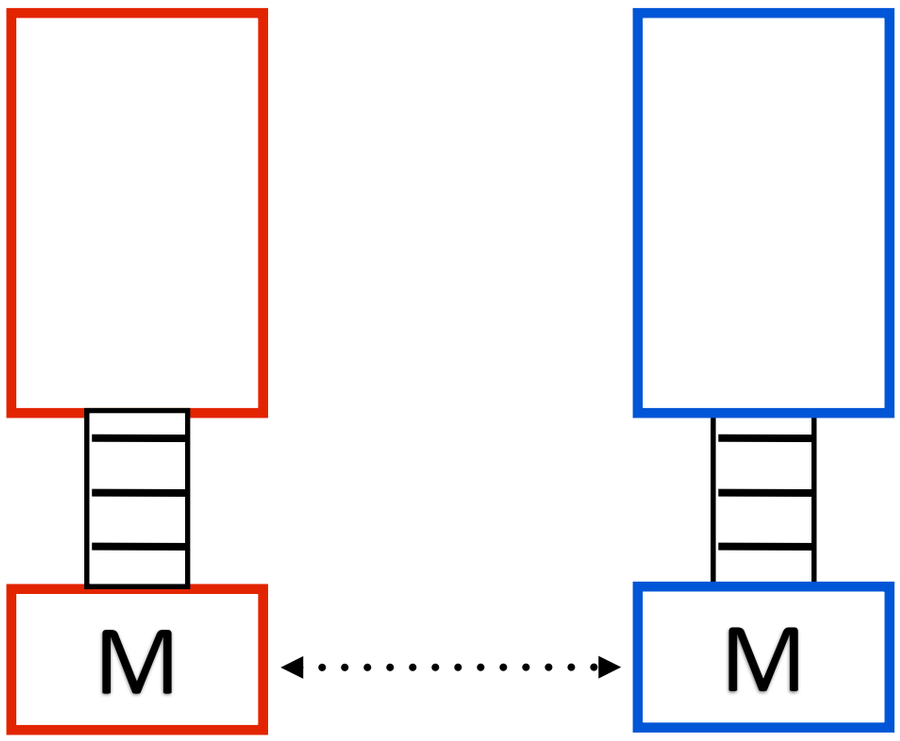
Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

Reordering #1: Write Buffers



WBs let reads finish before older writes

Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

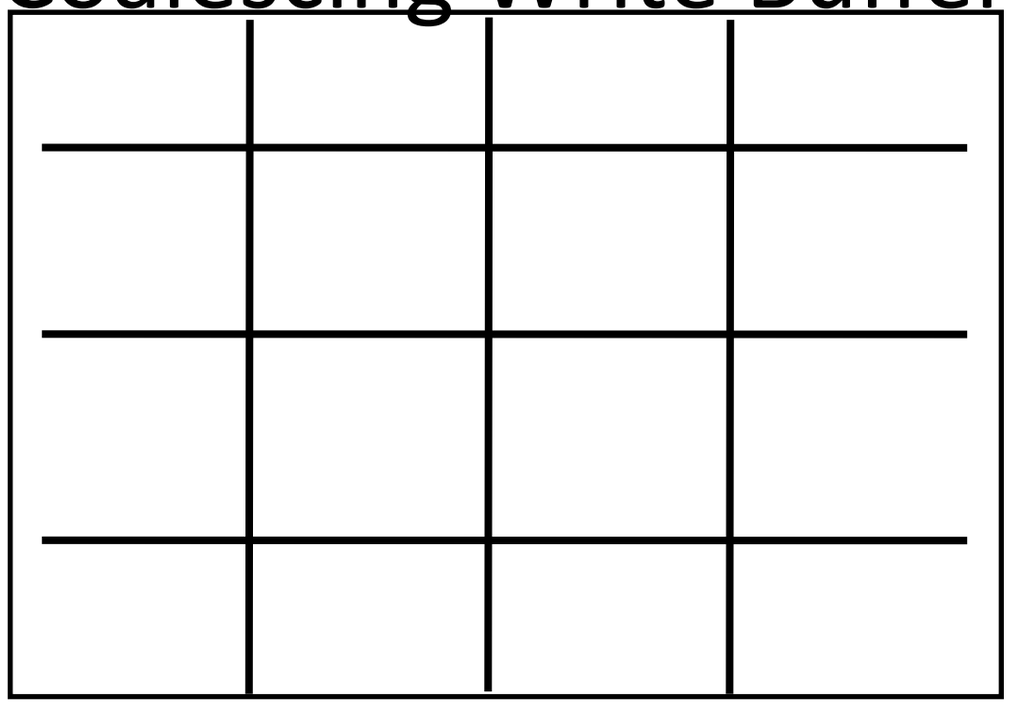
X=1

Y=1

(Not SC!)

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same $ line

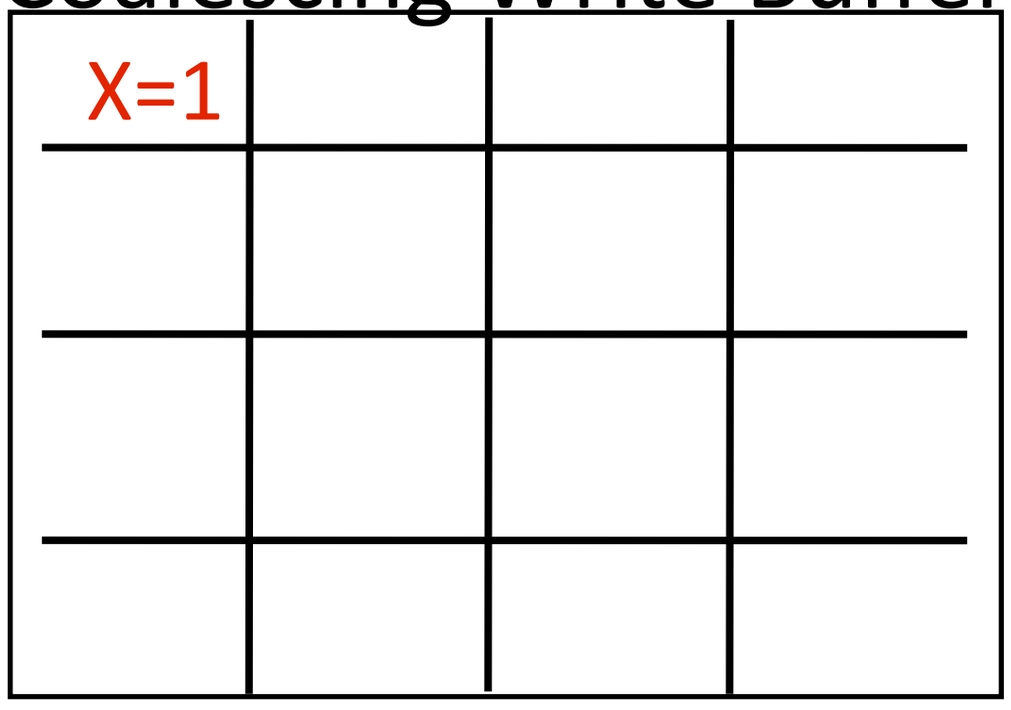
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

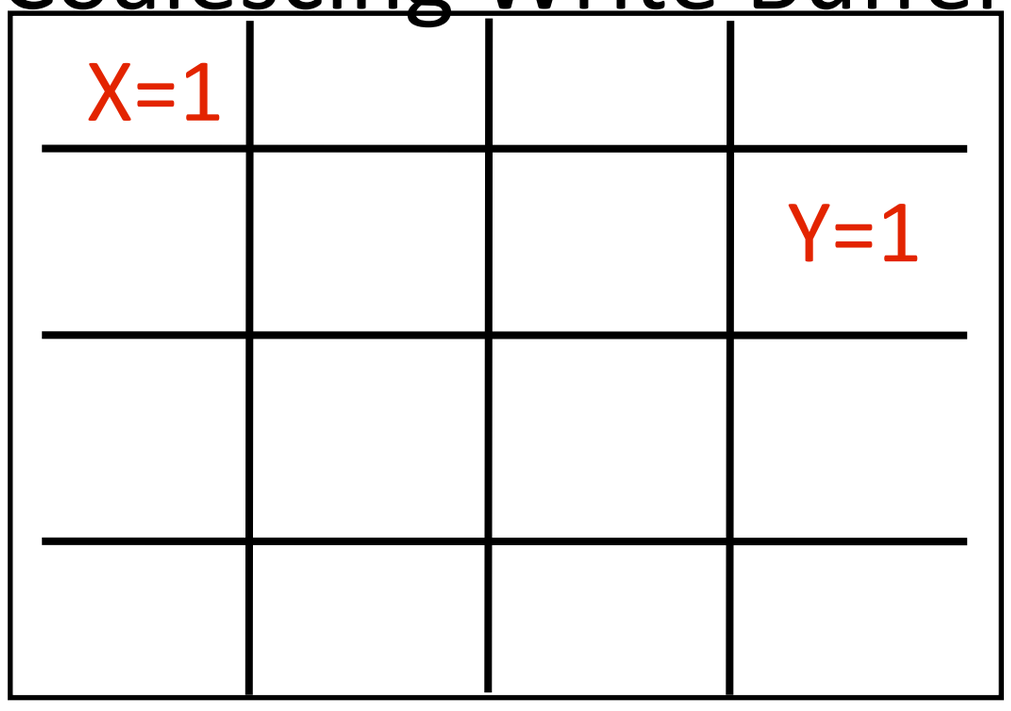
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

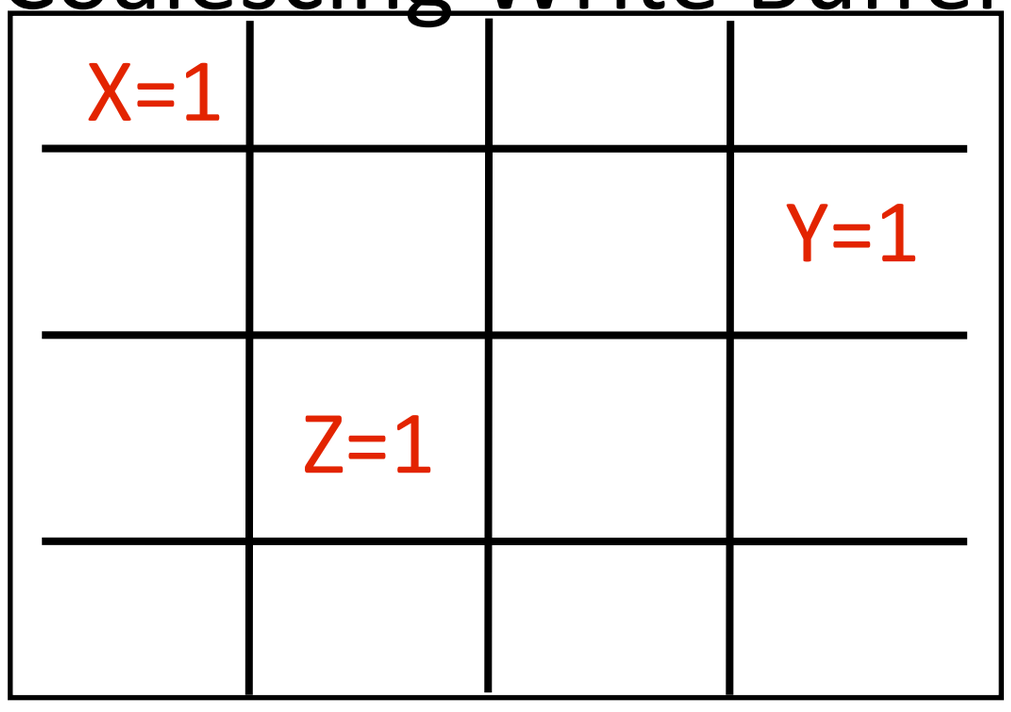
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

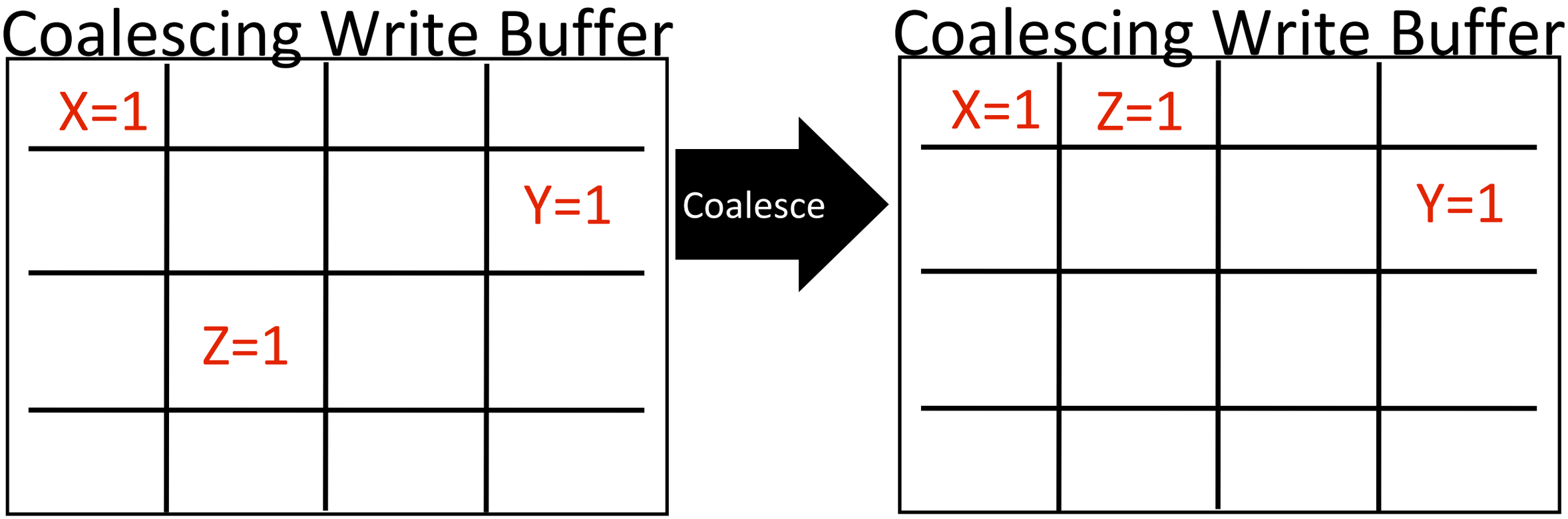
X,Z in same $ line

X=1

Y=1

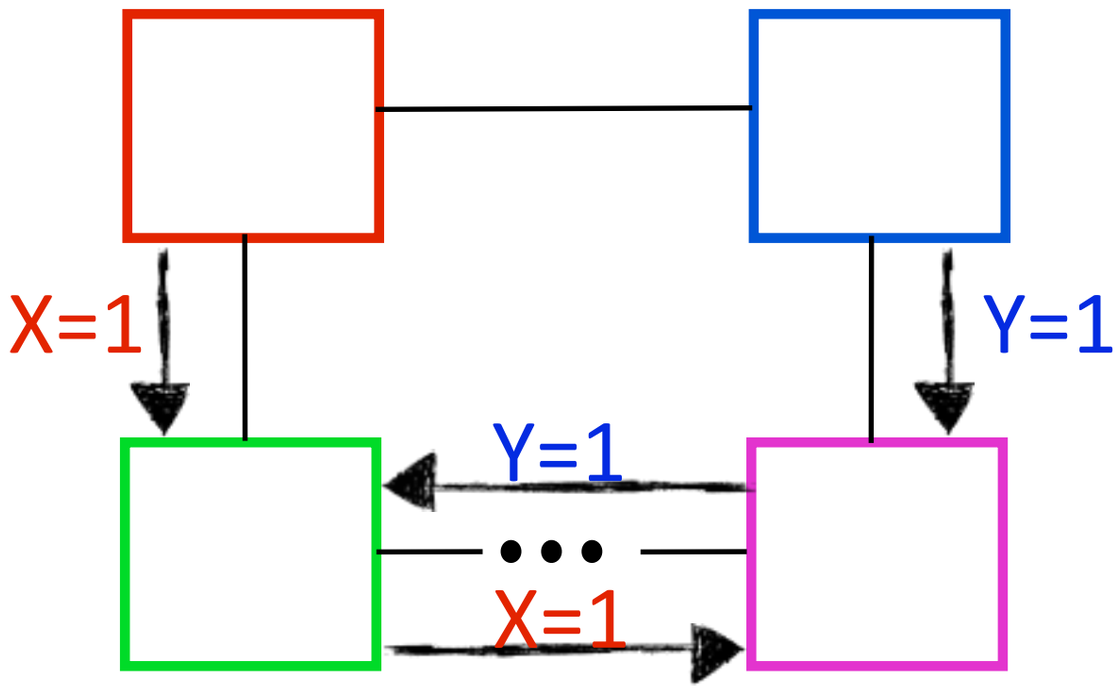
Z=1

Reordering #2: Write Combining



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

Reordering #3: Interconnect



Variable time cost traversing routed on-chip network

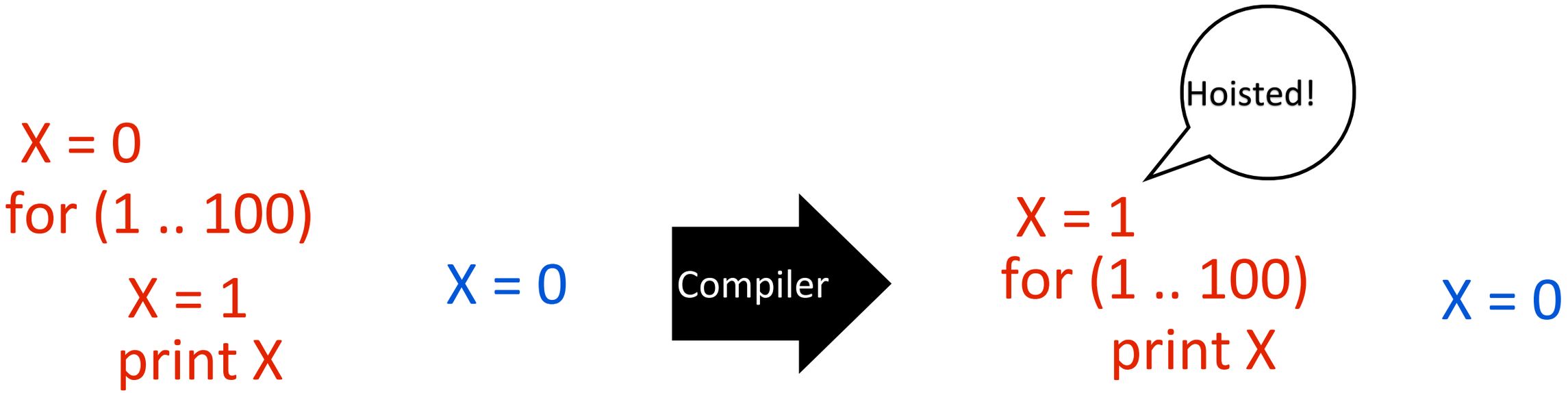
Program

X=1	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 0...”)

When is Reordering a Problem?

When Executions Aren’t SC

When is an Execution Not SC?

When a memory operation happens before itself

Execution

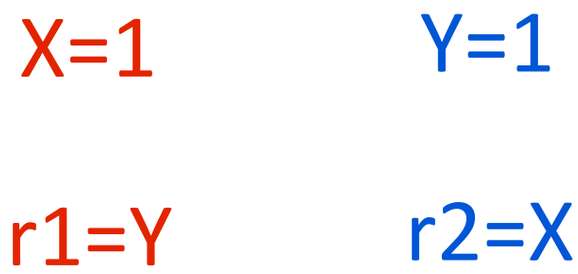
r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

Y=1

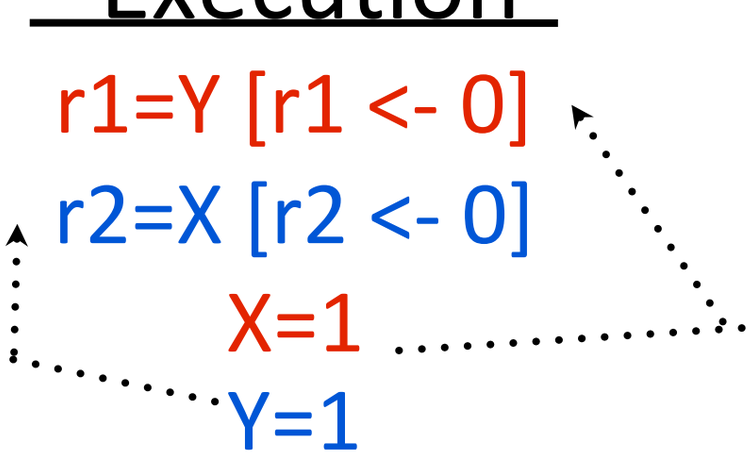
Happens-Before Graph



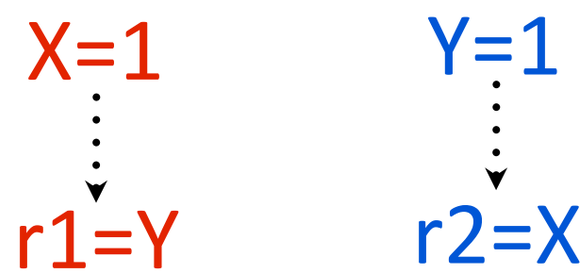
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph

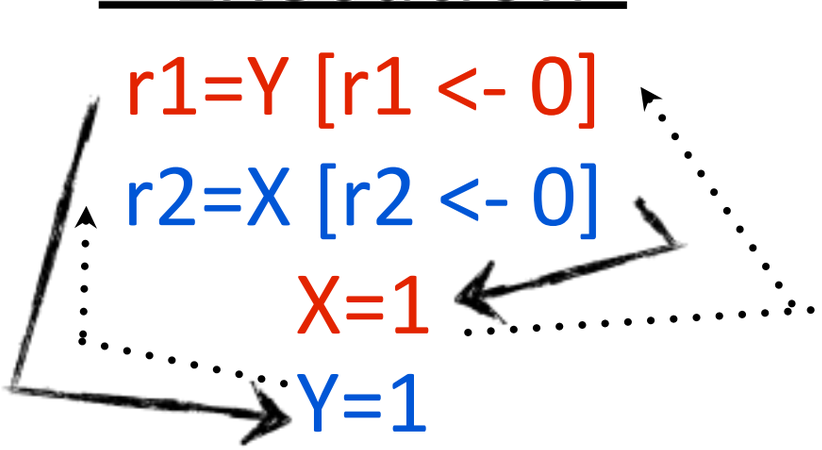


Program Order HB Edge

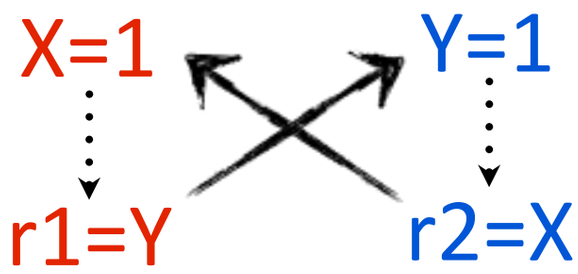
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph



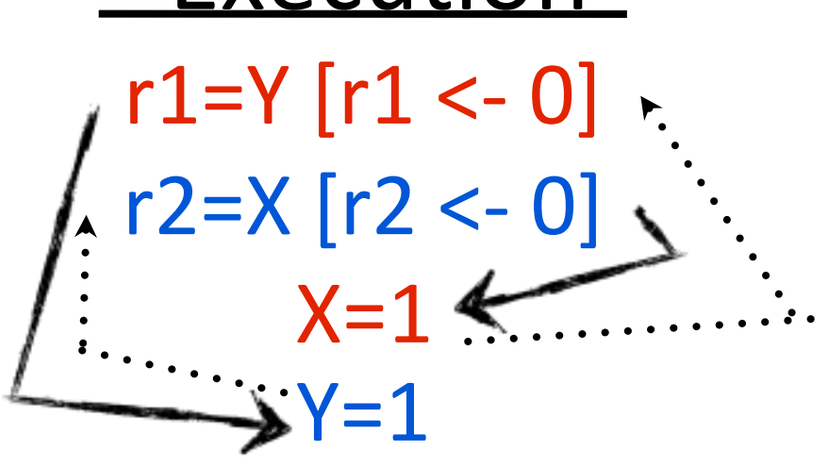
Program Order HB Edge

Causal Order HB Edge

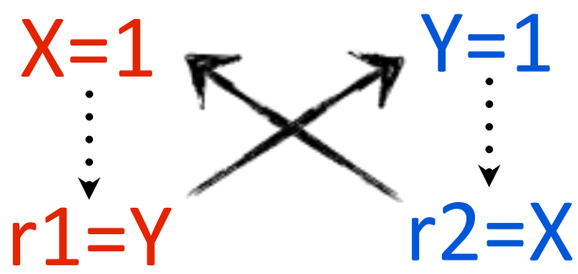
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph

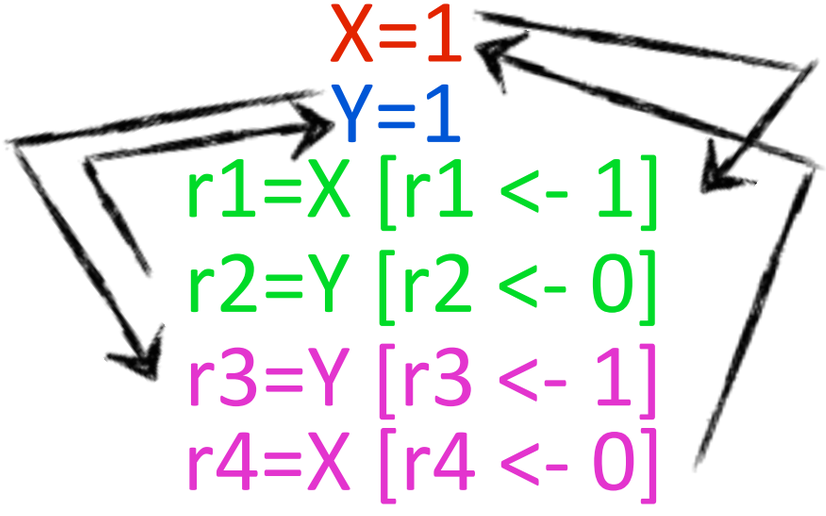


If there is a cycle in the happens-before graph, the execution is not SC

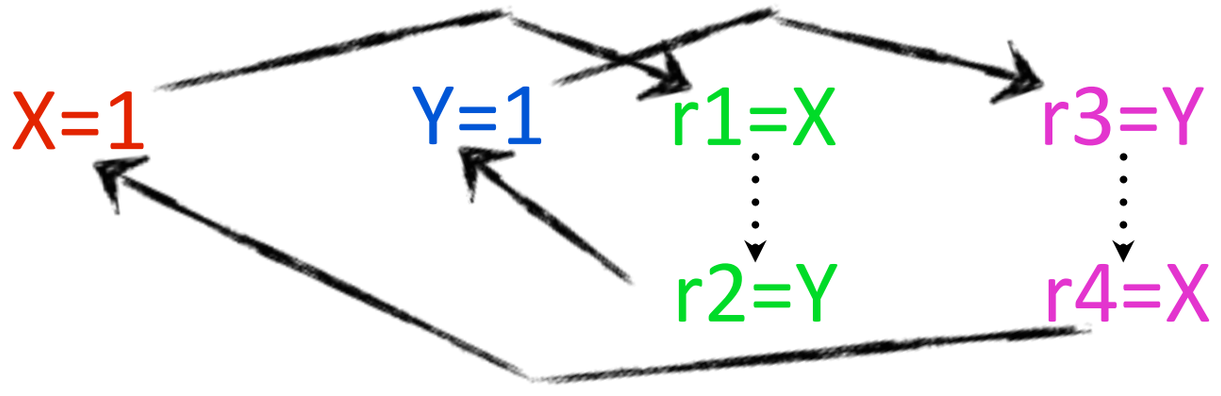
When is an Execution Not SC?

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

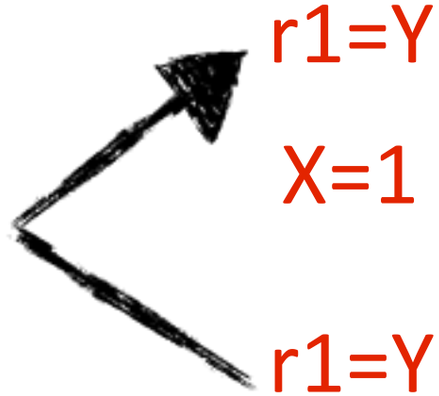
Combining writes saves bandwidth but reorders writes

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

"The Write Buffer Memory Model"

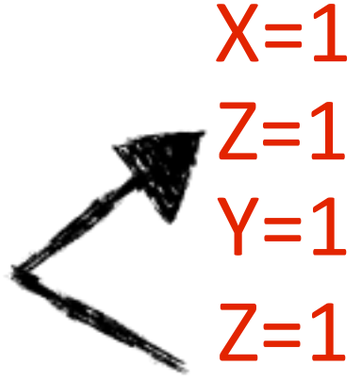


Relaxes W->R order

Total **S**tore **O**rder - loads may complete before older stores to different locations complete.

PSO(SPARC)

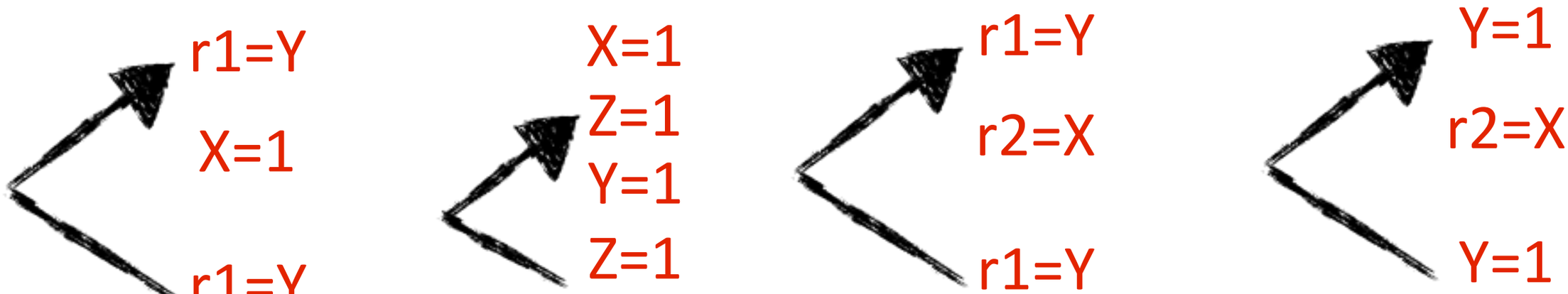
"The Write Combining Memory Model"



Relaxes W->W order

Partial **S**tore **O**rder - loads and stores may complete before older stores to different locations complete.

In General



W->R W->W R->R R->W

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

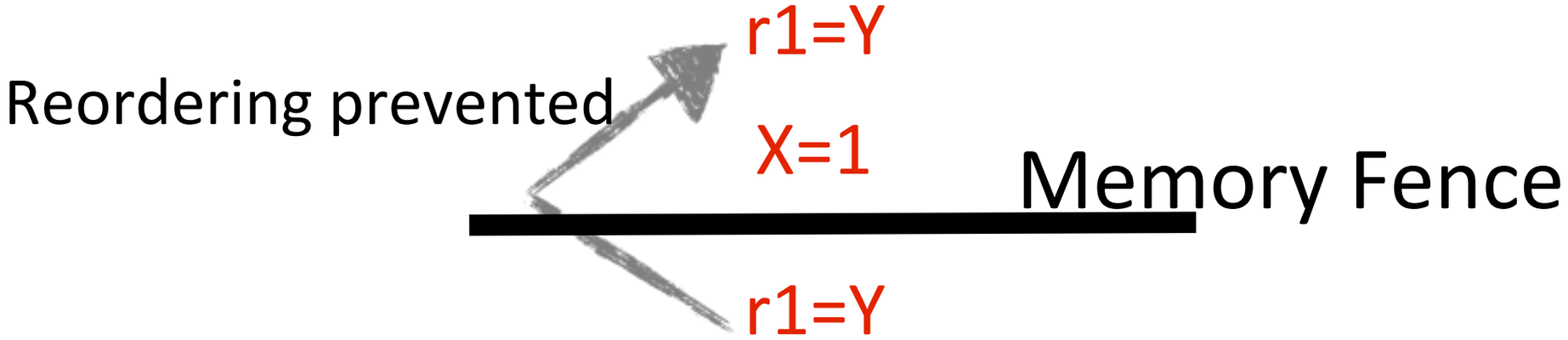
Goal: Ensure SC executions while permitting Relaxed Consistency reorderings

*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

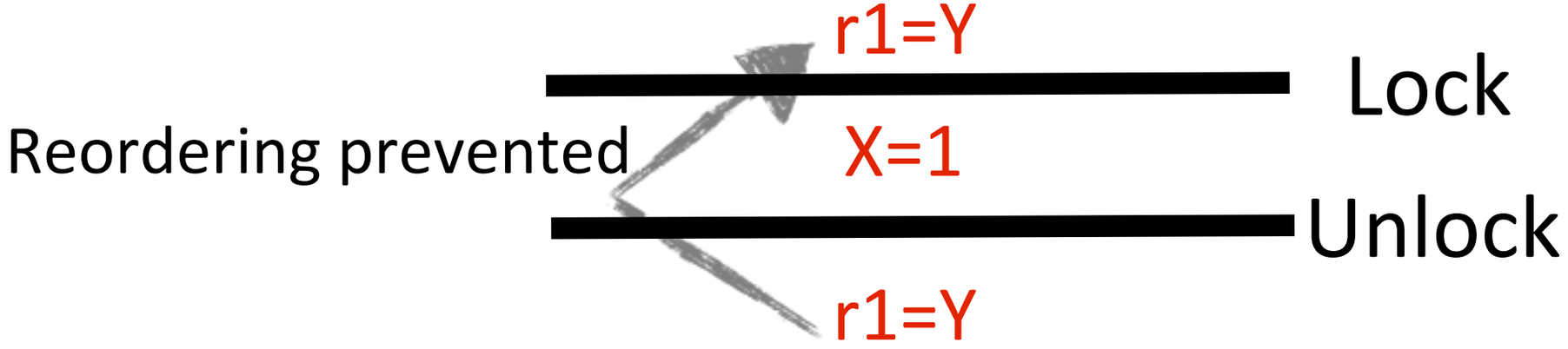


Fence implementation depends on reordering implementation

TSO: Stall reads until write buffer is empty

Synchronization For Real Programmers

Memory fences are wrapped up in locks, etc.

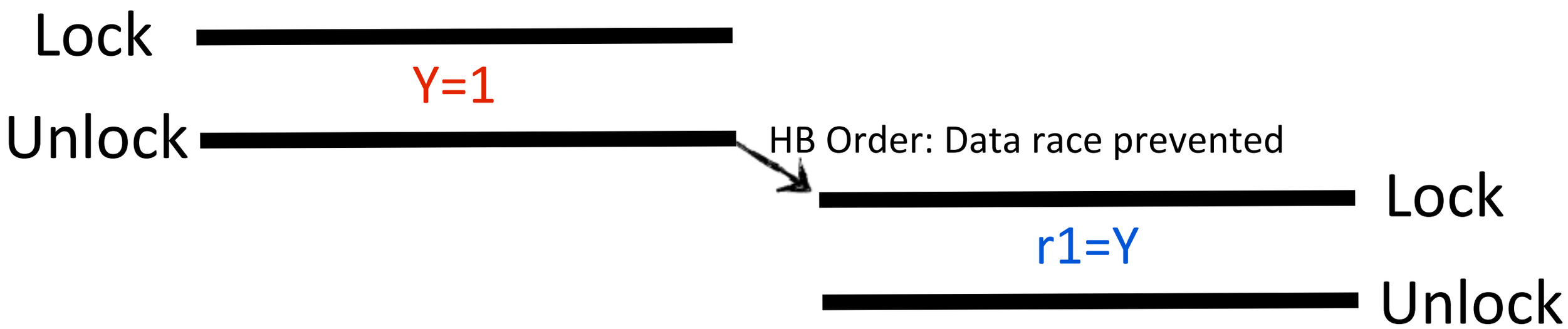


Direct use of fences possible, but inadvisable.

USE A 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	Compiler	Architecture
Java/C++: SC for data-race-free programs	Conservative with reordering when d-r-f can't be proved	Usually very weak for max optimization (lots of reordering) Note: fences from "above" ensure SC