Memory Consistency

Computer Architecture: Design and Simulation
CMU 15-346, Spring 2022

Simple Example

- Can we print Hello World

X = 1	Y = 1
if (Y == 0) print “Hello”	if (X == 0) print “World”

- Lots of examples will have global variables
  - assume each variable is initialized to 0
  - assume each column is a separate thread

Simple Example

- Are the two code sequences equivalent?

```
X = 0
for (Y = 0; Y < N; Y++)
  X = 1
  print X
```

```
X = 1
for (Y = 0; Y < N; Y++)
  print X
```

- What if another thread does X = 2?

Where can we reorder?

- **Processor pipelines for ILP**
  - **r2 = r4 / r5; // expensive**
    ***(r1 + r2) = 1;**
    ***r3 = 1;**

- **Branch prediction**
  - **if (r2 > 0)**
    r3 = *(r1 + r2);

- **Interconnection Network**
  - ***A = 1; // remote**
    ***B = 1; // Local**

while (B == 0);
print A ;

What do we "expect"?

- **Intuitive model**
  - One instruction at a time, in order

- **"Sequential Consistency" (Lamport 79)**
  - Each processor / thread follows program order
  - Each processor / thread access can interleave

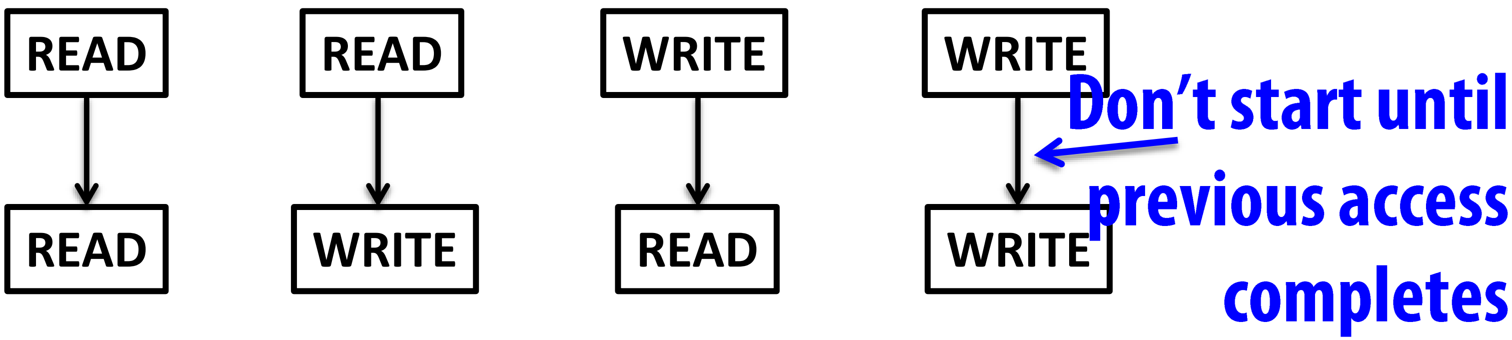
How might SC happen?

- Cache coherence is present
- Delay start of any memory access until the previous one is complete

- What is complete?
  - Read value “bound”
  - Write committed to a hypothetical serial order (HSO)

Describing Memory Consistency Constraints

- Maintain ordering between accesses in a processor

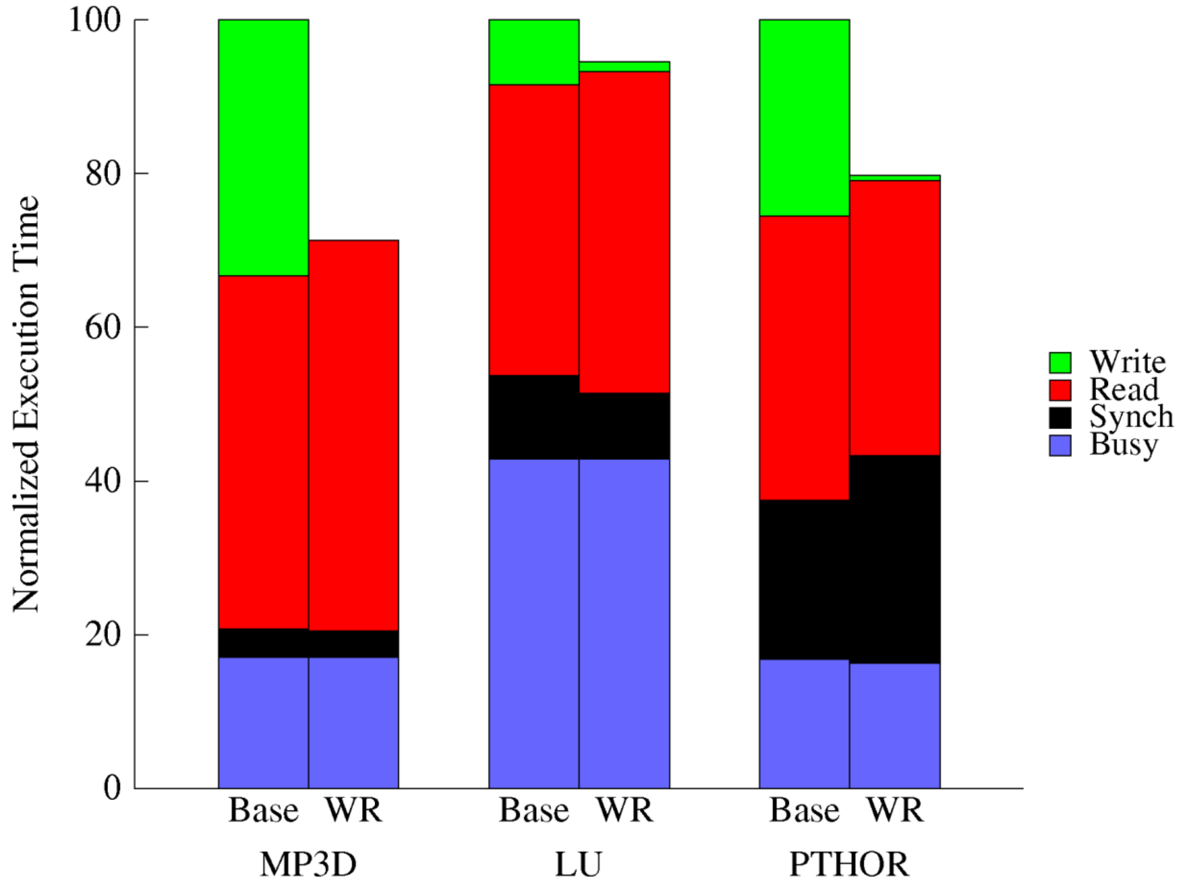


- Let's talk about two small relaxations

Write Buffering -> TSO

- Write buffers reduce write time
  - But this impacts our “hello world” example

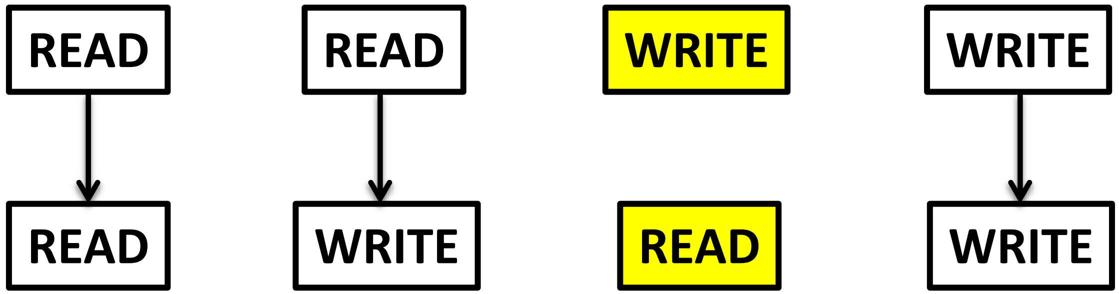
Write Buffering -> TSO



Write Coalescing -> PSO

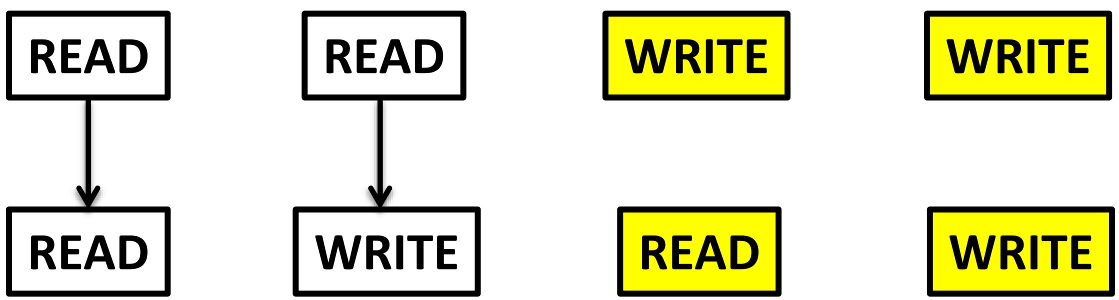
- Common writes to the same line could be merged

Alternatives to Sequential Consistency



Total Store Ordering (TSO) **(Similar to Intel)**

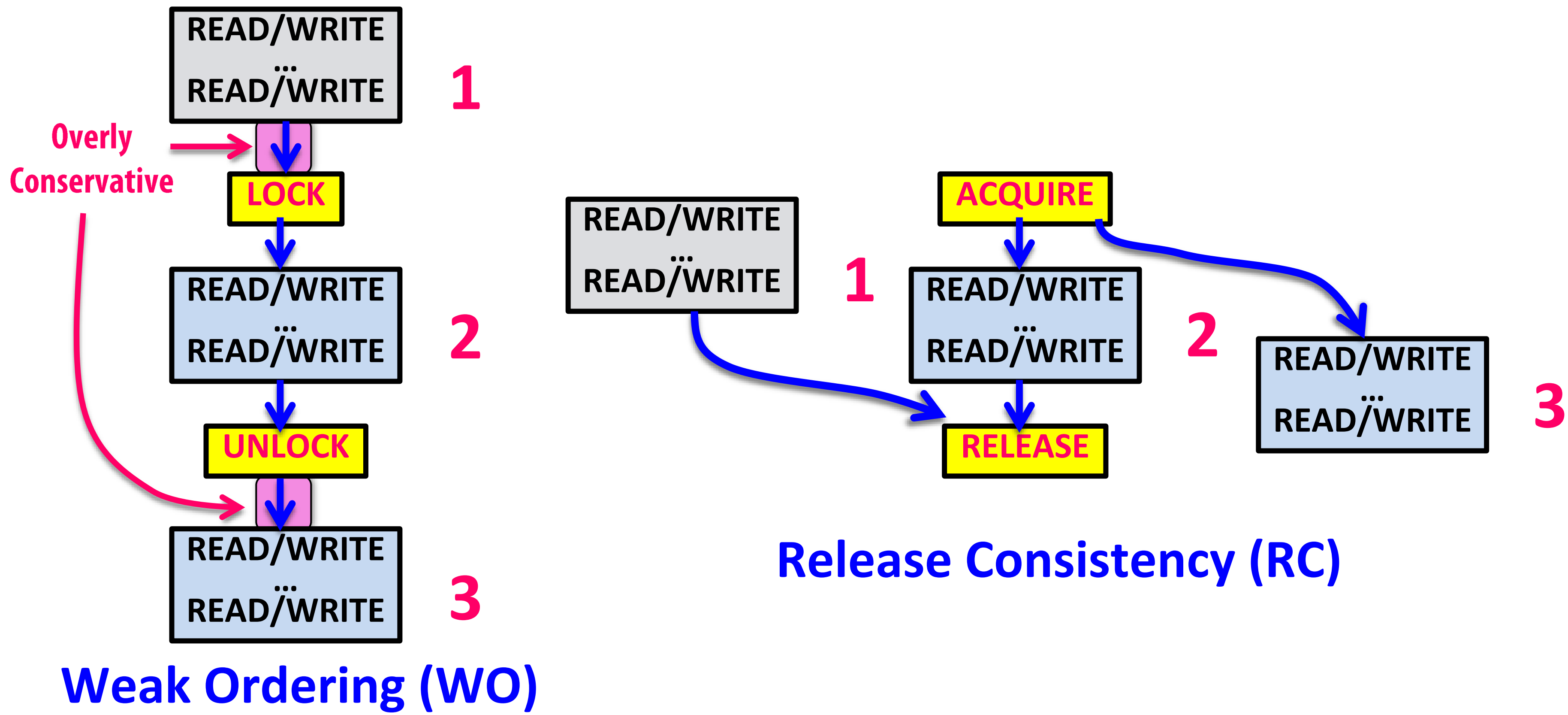
See Section 8.2 of "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1",
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf



Partial Store Ordering (PSO)

Relaxing further

- When we write parallel code, how do we make it "safe"?



Proving Memory Consistency

- Given a set of instructions and initial state, is there a valid ordering to reach a final state
  - What are the ordering constraints?
    - Barriers / fences
      - Explicit ordering of some or all instructions
    - Dependencies
      - Addr / data
    - Control dependencies
      - Should a branch matter?

Initial Example

- Two threads
  - Thread 0 writes to locations X, Y
  - Thread 1 reads from locations X, Y

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: $x=0 \wedge y=0$	

- What values can be observed? Can r1 = 1 and r2 = 0?
  - SC?
    - NO
  - TSO?
    - NO
  - Release Consistency?
    - Yes?!

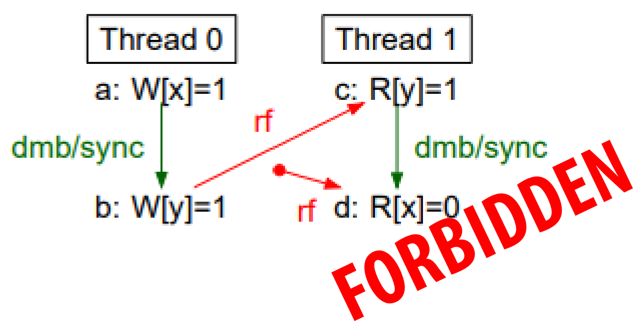
https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

Barriers

- **Placing a full barrier between the operations in threads 0 and 1**
  - **All examples are release consistency now**

MP+dmb/syncs Pseudocode

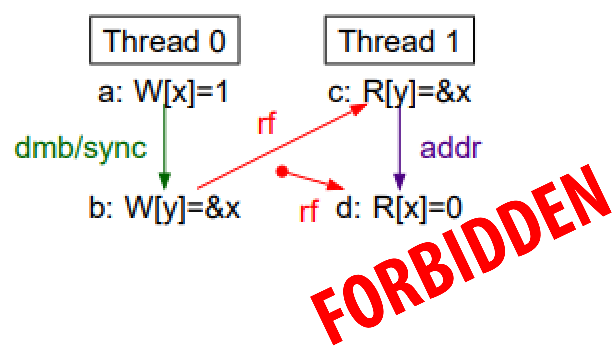
Thread 0	Thread 1
x=1	r1=y
dmb/sync	dmb/sync
y=1	r2=x
Initial state: $x=0 \wedge y=0$	



Adding a dependency between reads

- Replacing the barrier with an address dependency
  - Can thread 1 read *r1 and get 0?

MP+dmb/sync+addr′	Pseudocode
Thread 0	Thread 1
x=1 dmb/sync y=&x	r1=y r2=*r1
Initial state: $x=0 \wedge y=0$	

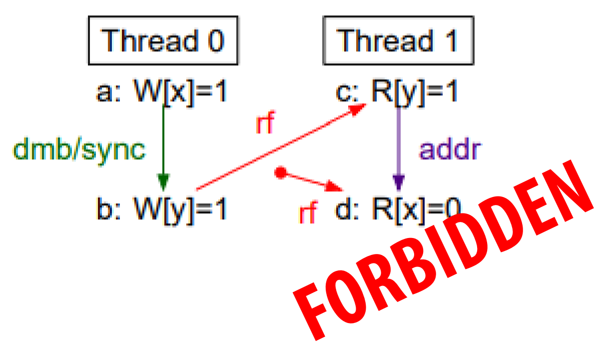


Extending dependencies

- **Programmers can force the value to still be a dependency without *using* it**
  - **The read of x “depends” on the value of y**

MP+dmb/sync+addr Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y r3=(r1 xor r1) r2=*(&x + r3)
Initial state: $x=0 \wedge y=0$	

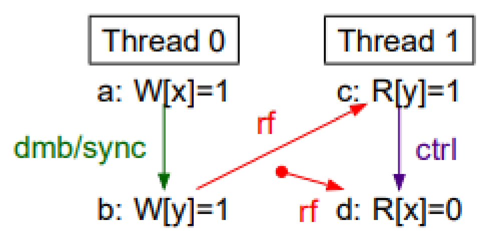


Control Dependencies

- **Rather than using the value in the address, can we control the order using branches?**
  - **If there is a branch or loop between the two operations, does that order them?**

MP+dmb/sync+ctrl Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y if (r1 == r1) {} r2=x
Initial state: $x{=}0 \wedge y{=}0$	



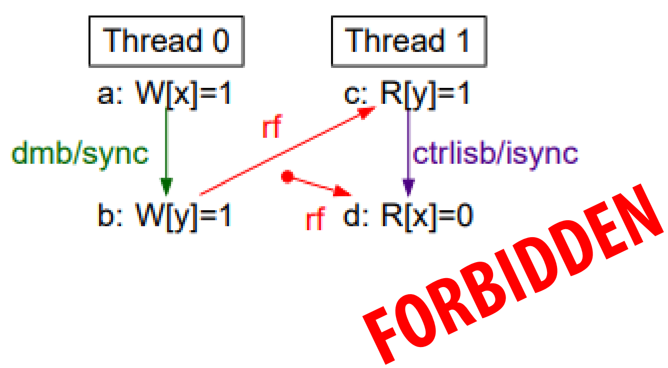
ALLOWED

Less Control Dependence

- **isb – instruction barrier**

MP+dmb/sync+ctrlisb/ctrlisync

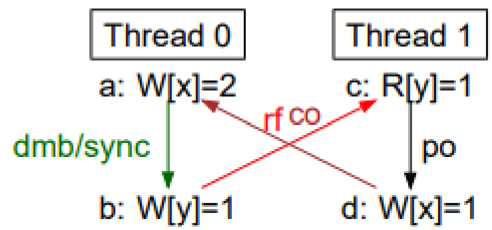
Thread 0	Thread 1
x=1	r1=y
dmb/sync	if (r1 == r1) {}
	isb/isync
y=1	r2=x
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	



Coherence Ordering of Writes

- **Before adding the branch…**

S+dmb/sync+po	Pseudocode
Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y x=1
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ x=2	



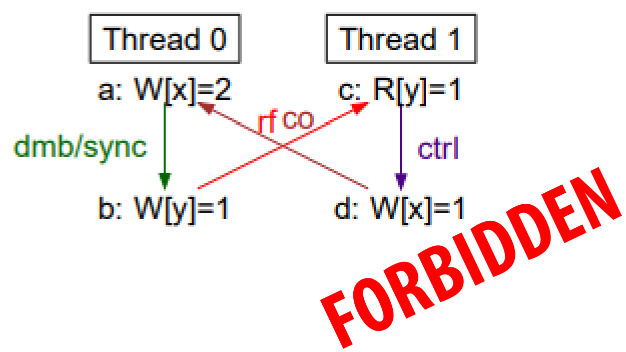
ALLOWED

Control Dependencies Matter

- Writes respect the branch
  - Without the branch, thread 1 can write before its read
  - With the branch (or a direct value dependence), the write ordering is forced.

S+dmb/sync+ctrl Pseudocode

Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y if (r1==r1) { } x=1
Initial state: x=0 ∧ y=0	



Are these orderings common?

- **Testing on ARM verifies that the orderings occur as expected.**

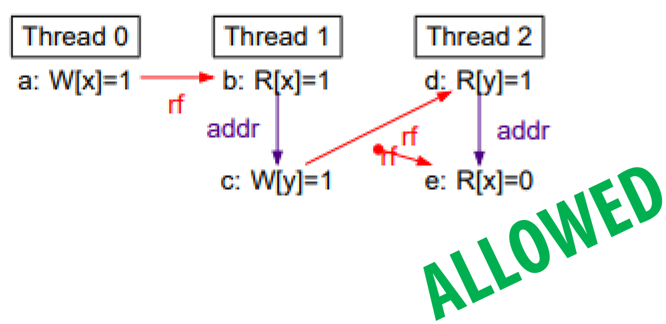
		ARM			
	Kind	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	40M/3.8G	138k/16M	61k/552M	437k/185M
MP+dmb/sync+po	Allow	3.1M/3.9G	50/28M	69k/743M	249k/195M
MP+dmb/sync+addr	Forbid	0/29G	0/39G	0/26G	0/2.2G
MP+dmb/sync+ctrl	Allow	5.7M/3.9G	1.5k/53M	556/748M	1.5M/207M
MP+dmb/sync+ctrlsib/isync	Forbid	0/29G	0/39G	0/26G	0/2.2G
S+dmb/sync+po	Allow	271k/4.0G	84/58M	357/1.8G	211k/202M
S+dmb/sync+ctrl	Forbid	0/24G	0/39G	0/26G	0/2.2G
S+dmb/sync+data	Forbid	0/24G	0/39G	0/26G	0/2.2G

Three threads interact

- How do written values propagate to multiple threads?
  - Is it possible for r3 = 0, while r1 and r2 are 1?
  - Adding a barrier in thread 1 makes this result forbidden

WRC+addrs Pseudocode

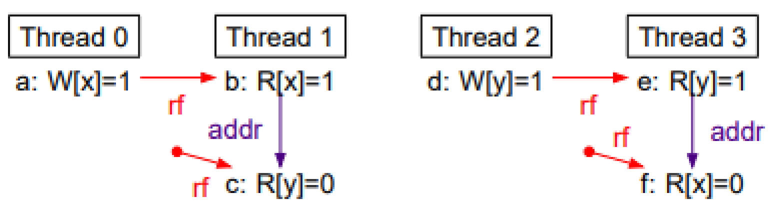
Thread 0	Thread 1	Thread 2
x=1	r1=x *(&y+r1-r1) = 1	r2=y r3 = *(&x + r2 - r2)
Initial state: $x=0 \wedge y=0$		



Threads propagate differently

IRIW+addrs Pseudocode

Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*(&y+r1-r1)	y=1	r3=y r4=*(&x+r3-r3)
Initial state: x=0 ∧ y=0 ∧ z=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			



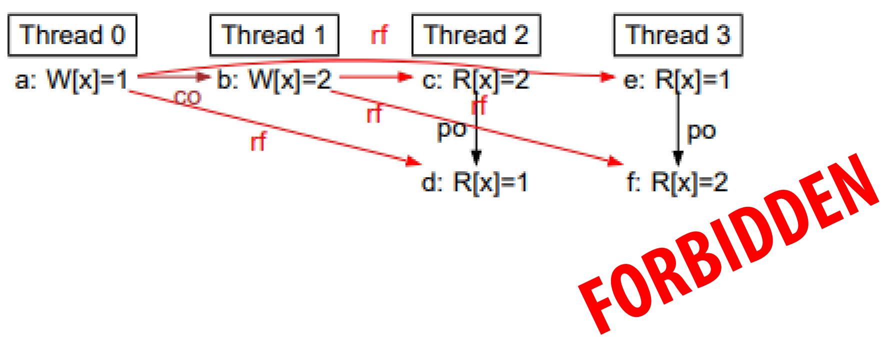
Test IRIW+addrs: Allowed

ALLOWED

Only experimentally observed on Power, not on ARM

Is there anything left?

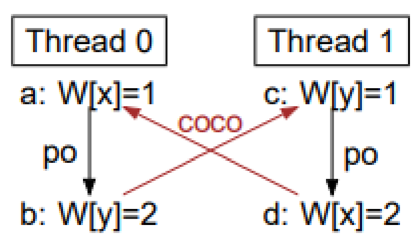
- Yes! Cache coherence must still work.
- Threads must agree on an order of writes.
- And remember that coherence is per-block



Coherence is per-block

2+2W Pseudocode

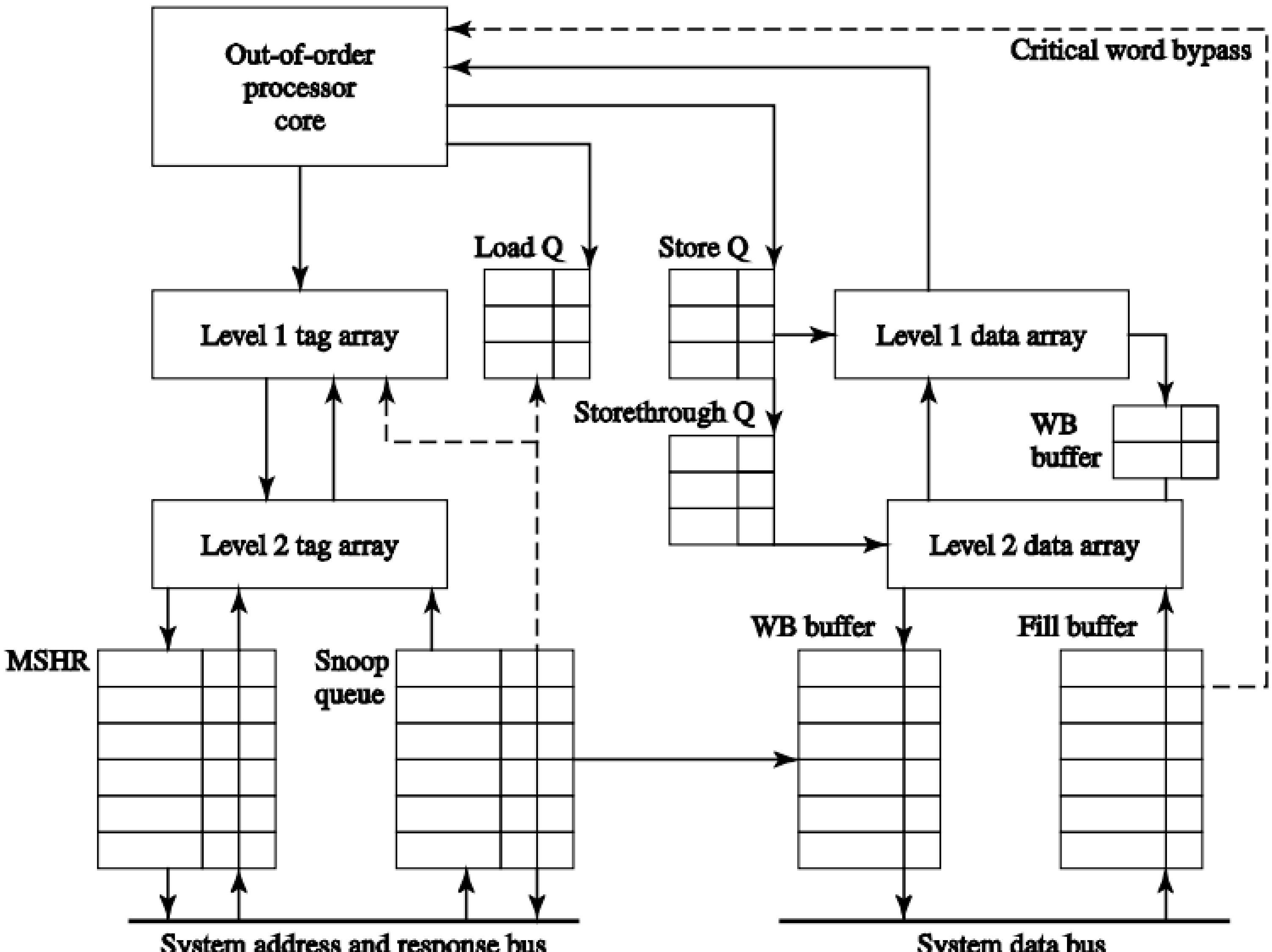
Thread 0	Thread 1
x=1 y=2	y=1 x=2
Initial state: $x=0 \wedge y=0$	
Allowed: $x=1 \wedge y=1$	



Test 2+2W: Allowed

ALLOWED

Coherent Memory Interface



https://ece752.ece.wisc.edu/lect11-advanced-caches.pdf

Coherent Memory Interface

- Load Queue – Tracks inflight loads for aliasing, coherence
- Store Queue – Defers stores until commit, tracks aliasing
- Storethrough Queue or Write Buffer or Store Buffer – Defers stores, coalesces writes, must handle RAW
- MSHR – Tracks outstanding misses, enables lockup-free caches [Kroft ISCA 91]
- Snoop Queue – Buffers, tracks incoming requests from coherent I/O, other processors
- Fill Buffer – Works with MSHR to hold incoming partial lines
- Writeback Buffer – Defers writeback of evicted line (demand miss handled first)