# Types of Litmus Tests

# Types of litmus tests

1. **MP** - Message Passing
2. **S** - Store
3. **WRC** - Write to Read Causality
4. **ISA2**

5. **SB** - Store Buffer
6. **IRIW** - Independent Reads Independent Write
7. **RWC** - Read to Write Causality
8. **R** - Read Buffer
9. **2+2W** - two threads each with two writes

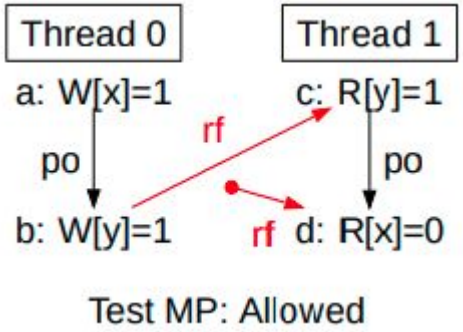
10. **LB** - Load Buffer
11. **PPOCA**
12. **PPO**

# Message passing (MP)

- Its family name is **WW+RR**
- The nodes in diagram are labelled as **a,b,c,d**
- Each node specify whether it is **read(R) or write(W)**,its location(here **x or y**) and value read or written in this execution (here **0 or 1**)
- The events are laid out in one column per thread and there is a **program order (po) edge** between two events of each thread.
- The rf edge between b and c indicates read c reads from the write b, rf edge from red dot to read d indicates the latter read from initial state

MP Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>y=1 | r1=y<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Allowed: 1:r1=1 ∧ 1:r2=0 | |

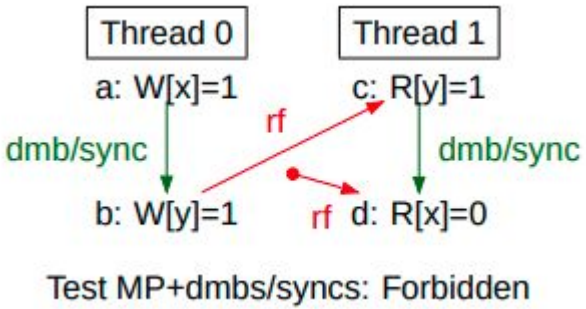


# Enforcing order with barriers/fences

- A strong memory barrier (or fence) instruction inserted between the two writes on Thread 0, and between the two reads on Thread 1, suffices.
- if they are inserted between every pair of memory accesses, they restore sequentially consistent behaviour

MP+dmb/syncs Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>dmb/sync<br>y=1 | r1=y<br>dmb/sync<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: 1:r1=1 ∧ 1:r2=0 | |



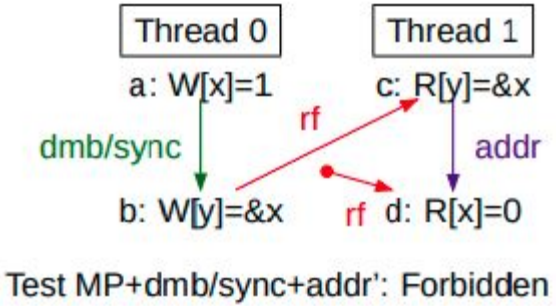
# Enforcing Order with Dependencies

- one can enforce enough ordering to prohibit the undesired outcome just by relying on various kinds of dependency in the code.

**Address dependency**

MP+dmb/sync+addr′ Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>dmb/sync<br>y=&x | r1=y<br><br>r2=*r1 |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: 1:r1=&x ∧ 1:r2=0 | |



# MP+barrier/fence+addr

- In the variation of MP, instead of writing a flag value of 1, the writer Thread 0 writes the address of location x, and the reader Thread 1 uses that address for its second read.
- The dependency is enough to keep the two reads satisfied in program order on Thread 1: the second read cannot get started until its address is known, so the second read cannot be satisfied until the first read is satisfied
- Combining that with the fence on Thread 0 (which keeps the write to x and the write to y in order) is enough to prevent Thread 1 reading 0 from x if it has read &x from y.

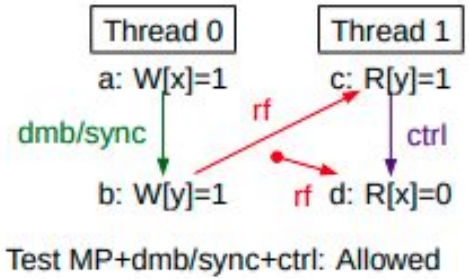
# MP+barrier/fence+ctrl

## Control dependency

- Here the value read by the first read is used to compute the condition of a conditional branch that is program-order-before the second read or write
- This example shows that it can allow the **non-sequentially consistent outcome** where Thread 1 reads 1 for y (from Thread 0's write) but still reads 0 for x (from the initial state).
- This outcome is allowed even though program order would suggest that the second read should happen after the first as control dependencies are not sufficient to guarantee memory ordering

MP+dmb/sync+ctrl Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>dmb/sync<br>y=1 | r1=y<br>if (r1 == r1) {}<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Allowed: 1:r1=1 ∧ 1:r2=0 | |



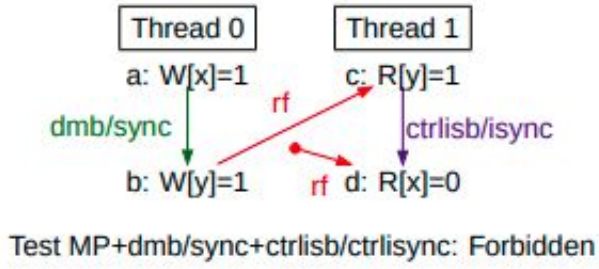
Test MP+dmb/sync+ctrl: Allowed

# MP+fence+ctrl-fence

- Add a fence instruction between the conditional branch and the second read, as in the example below.
- This prevents the second read from being satisfied until the conditional branch is committed, which cannot happen until the value of the first read is fixed (i.e., until that read is satisfied and committed); the two reads are thus kept in order and the specified outcome of the test is now forbidden.

MP+dmb/sync+ctrlisb/ctrlisync

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



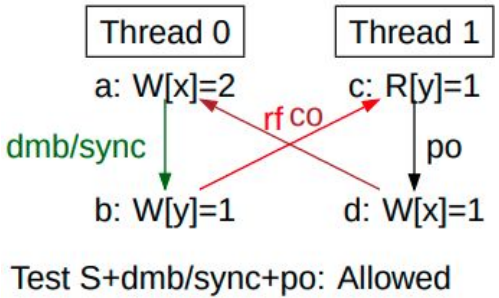

Test MP+dmb/sync+ctrlisb/ctrlisync: Forbidden

# Store (S)

- Its family name is **WW+RW**
- It is a variation of the MP test family, known as S, in which the second read on Thread 1 (of x) is replaced by a write of x.
- Here we check whether the Thread 1 write of x is guaranteed to be coherence-after the Thread 0 write of x
- Lets a take a case having fence on thread 0 (S+fence+po)

S+dmb/sync+po Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=2<br>dmb/sync<br>y=1 | r1=y<br>x=1 |
| Initial state: x=0 ∧ y=0 | |
| Allowed : 1:r1=1 ∧ x=2 | |

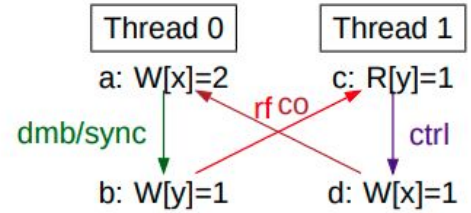



Test S+dmb/sync+po: Allowed

# S+fence+ctrl and S+fence+data

S+dmb/sync+ctrl Pseudocode

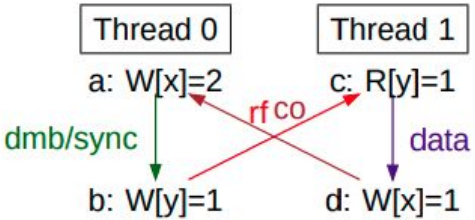
| Thread 0 | Thread 1 |
|---|---|
| x=2<br>dmb/sync<br>y=1 | r1=y<br>if (r1==r1) { }<br>x=1 |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: 1:r1=1 ∧ x=2 | |




Test S+dmb/sync+ctrl: Forbidden

S+dmb/sync+data Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=2<br>dmb/sync<br>y=1 | r1=y<br>r3 = (r1 xor r1)<br>x = 1 + r3<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: 1:r1=0 ∧ x=1 | |



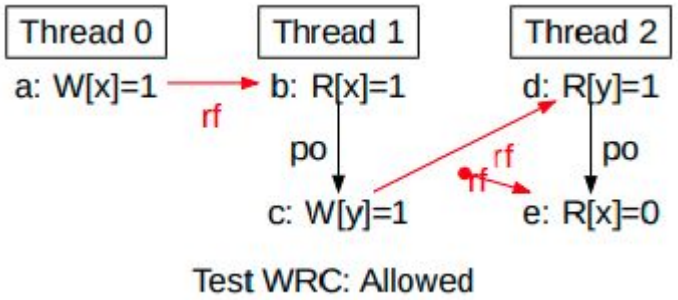

Test S+dmb/sync+data: Forbidden

# Write To Read Causality (WRC)

- The WRC test introduces a slight modification to the MP test by involving three threads instead of two. The key idea is to separate the two writes of Thread 0 (from the MP test) and move one of them to a new thread, which results in this three-thread pattern.

WRC Pseudocode

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1 | r1=x<br>y=1 | r2=y<br>r3=x |
| Initial state: x=0 ∧ y=0 | | |
| Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0 | | |

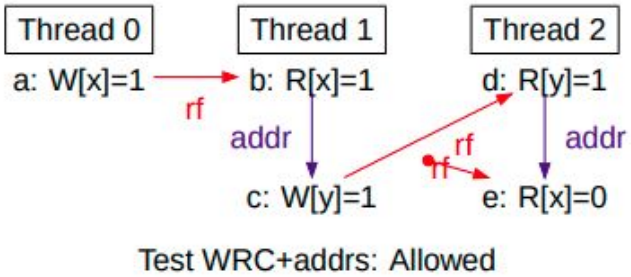


# WRC+addrs

- The WRC+addrs test is a variation of the WRC (Write-to-Read Causality) test, but with added artificial dependencies between the memory operations

WRC+addrs Pseudocode

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1 | r1=x<br>*(&y+r1-r1) = 1 | r2=y<br>r3 = *(&x + r2 - r2) |
| Initial state: x=0 ∧ y=0 | | |
| Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0 | | |




- On a **multiple-copy-atomic** architecture (like **x86**), where changes to memory are immediately visible to all threads, this behavior would be **forbidden**.

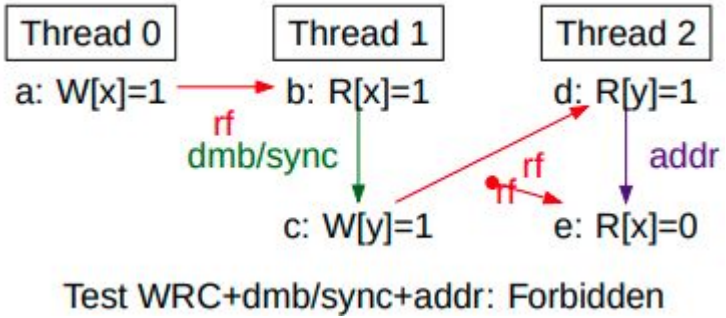
- However, on **ARM** ,**POWER and RISCV** architectures, which are **not multiple-copy atomic**, writes can propagate to different threads at different times. This means that **Thread 1** may see the previous state of x rather than the latest write to x in **Thread 0.**

# WRC with Barriers

- To prevent the unintended outcome of WRC, one can strengthen the Thread 1 address dependency above, replacing it by a DMB or sync barrier.
- The barrier ensures that any write that has propagated to Thread 1 before the barrier is propagated to any other thread before the Thread 1 writes after the barrier can propagate to that other thread.

WRC+dmb/sync+addr Pseudocode

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1 | r1=x<br>dmb/sync<br>y=1 | r2=y<br>r3 = *(&x + r2 - r2) |
| Initial state: x=0 ∧ y=0 | | |
| Forbidden: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0 | | |

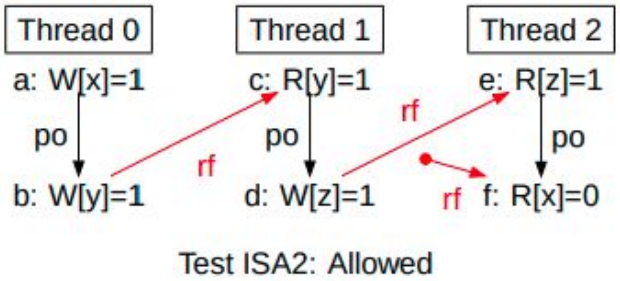


# ISA2

- The generalisation of message-passing to three threads we saw in Section 5, which needs a barrier only on the first thread
- It extends the basic Message Passing (MP) and Write-to-Read Causality (WRC) examples by introducing additional complexity with a third shared variable z, which introduces cumulative barriers to prevent improper reordering of memory operations across threads.
- In weak memory models like **ARM** and **POWER**, operations such as reads and writes to different memory addresses (in this case, x, y, and z) can be reordered across threads unless barriers are used. Specifically:
- **Thread 2** might see z = 1 (the write from **Thread 1**) without seeing x = 1 which leads to inconsistent behaviour

ISA2 Pseudocode

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1<br>y=1 | r1=y<br>z=1 | r2=z<br>r3=x |
| Initial state: x=0 ∧ y=0 ∧ z=0 | | |
| Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0 | | |



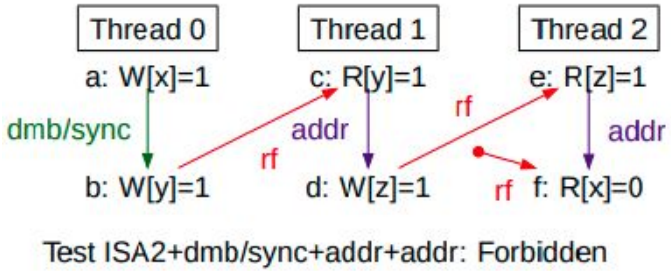
Test ISA2: Allowed

# ISA2+fence+addr+addr

- To make this work (i.e., to forbid the stated final state), it suffices to have a fence or barrier on Thread 0 and preserved dependencies on Threads 1 and 2 (an address, data or control dependency between the Thread 1 read/write pair, and an address or control-isb/isync dependency between the Thread 2 read/read pair).

ISA2+dmb/sync+addr+addr Pseudocode

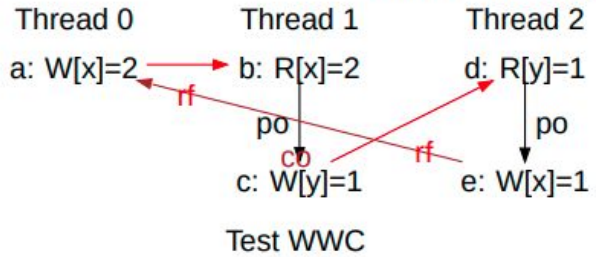
| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1<br>dmb/sync<br>y=1 | r1=y<br>*(&z+r1-r1)=1 | r2=z<br>r3 = *(&x +r2-r2) |
| Initial state: x=0 ∧ y=0 ∧ z=0 | | |
| Forbidden: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0 | | |




Test ISA2+dmb/sync+addr+addr: Forbidden

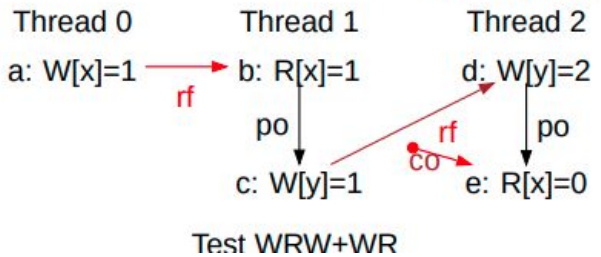
# Other Three Threads Tests

**WWC: rf,rf,co** needs lwsync+RWdep or dmb+RWdep



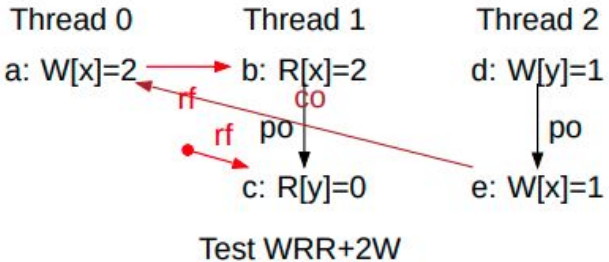

Test WWC

**WRW+WR: rf,co,fr** needs sync+sync or dmb+dmb



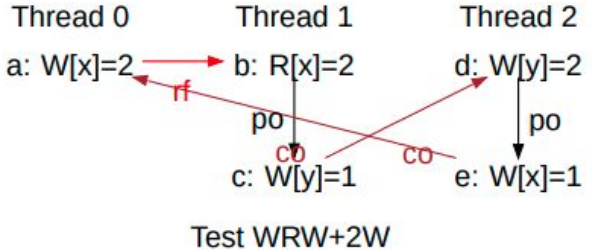

Test WRW+WR

**WRR+2W: rf,fr,co** needs sync+sync or dmb+dmb




Test WRR+2W

**WRW+2W: rf,co,co** needs lwsync+lwsync or dmb+dmb



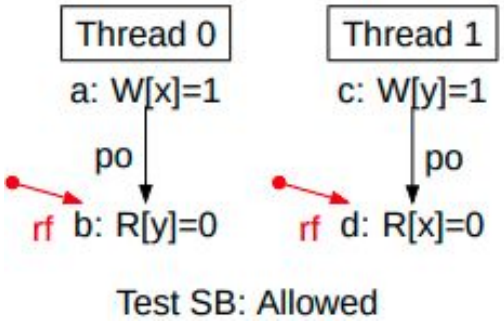

Test WRW+2W

# Store Buffering (SB)

- Its family name is WR+WR.
- It is just like MP, but now each thread writes one location then reads from the other
- Without any barriers or dependencies, that outcome is allowed,due to weak memory model (reordering)

SB Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>r1=y | y=1<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Allowed: 0:r1=0 ∧ 1:r2=0 | |

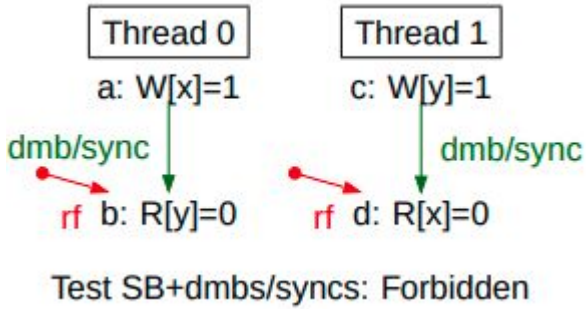


# SB+fence

- The only possible strengthening of the code of SB is to insert barriers. Adding a fence on both threads suffices to rule out the unintended outcome
- Here the dmb or sync barriers ensure that the program-order-previous writes must have propagated to all threads before the reads are satisfied

SB+dmbs/syncs Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>dmb/sync<br>r1=y | y=1<br>dmb/sync<br>r2=x |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: 0:r1=0 ∧ 1:r2=0 | |

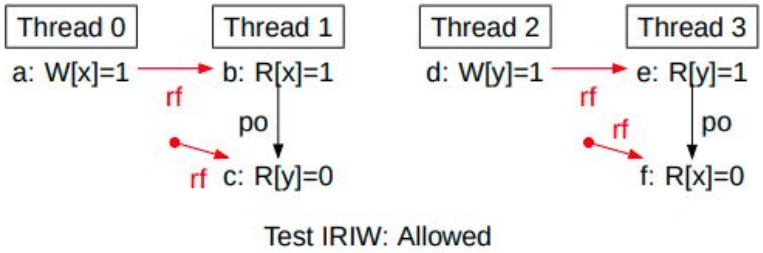


# Independent Read and Independent Write (IRIW)

- The IRIW (Independent Reads of Independent Writes) test is a memory consistency test designed to show how independent writes to different memory locations can be observed out of order by different threads in weak memory models
- The IRIW test involves four threads interacting with two shared variables, x and y. The two key operations are independent writes to x and y, followed by reads of these variables in different orders by two other threads.

IRIW Pseudocode

| Thread 0 | Thread 1 | Thread 2 | Thread 3 |
|---|---|---|---|
| x=1 | r1=x<br>r2=y | y=1 | r3=y<br>r4=x |
| Initial state: x=0 ∧ y=0 | | | |
| Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0 | | | |

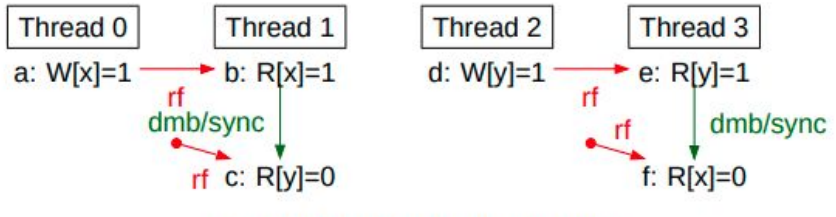


# IRIW+fence

- In above example, due to  weak memory models (like RISCV ,ARM)  IRIW shows that Thread 1 can still observe x = 1 but not y = 1, while Thread 3 can observe y = 1 but not x = 1.
- To prevent this behavior, you may add memory barriers in Thread 1 and Thread 3 after their first read and before their second read. But due yo lack of multiple-copy atomicity the barriers ensures that reads and writes within each thread are ordered, but it doesn't guarantee that other threads will see those writes immediately.

IRIW+dmbs/syncs Pseudocode

| Thread 0 | Thread 1 | Thread 2 | Thread 3 |
|---|---|---|---|
| x=1 | r1=x<br>dmb/sync<br>r2=y | y=1 | r3=y<br>dmb/sync<br>r4=x |
| Initial state: x=0 ∧ y=0 | | | |
| Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0 | | | |

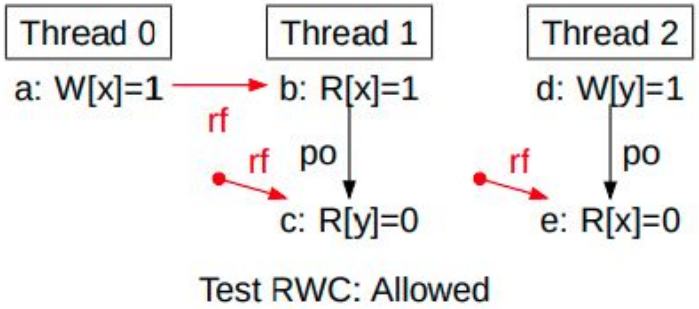


# Read to Write Causality (RWC)

- It is variant of SB involving three threads , where one of the SB write is pulled out to a new thread creating three thread pattern.

RWC Pseudocode

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| x=1 | r1=x<br>r2=y | y=1<br>r4=x |
| Initial state: x=0 ∧ y=0 ∧ z=0 | | |
| Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 2:r4=0 | | |

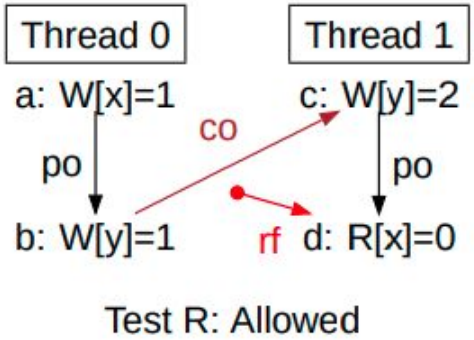


# Read (R)

- Variation R of SB involves modifying the store buffer model by replacing one of the reads with write, to observe how such changes impact memory ordering and consistency.
- To forbid this unintended outcome use fences in both threads

R Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>y=1 | y=2<br>r1=x |
| Initial state: x=0 ∧ y=0 | |
| Allowed: y=2 ∧ 1:r1=0 | |

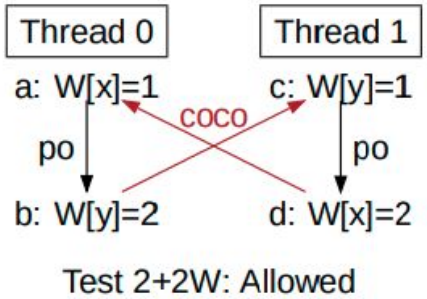


# 2+2W

- Variation 2+2W extends the R variation by involving two threads performing two writes each. This test explores how weak memory models handle multiple writes and their interaction with reads.
- To forbid this unintended outcome use fences in both threads

2+2W Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| x=1<br>y=2 | y=1<br>x=2 |
| Initial state: x=0 ∧ y=0 | |
| Allowed: x=1 ∧ y=1 | |



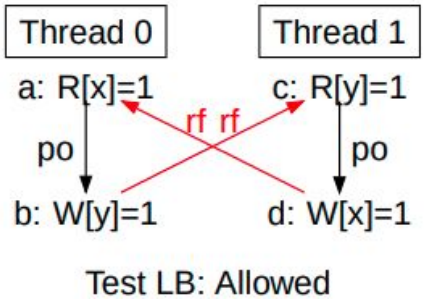
Test 2+2W: Allowed

# Load Buffering (LB)

- The opposite of store buffering is called Load Buffering (LB)
- Here, two threads first read from two shared locations respectively and then write to the other locations
- The outcome in which the reads both read from the write of the other thread
- Here both writes may be committed before reads due weak memory model and reordering of operations. So, the final condition is allowed.

LB Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| r1=x<br>y=1 | r2=y<br>x=1 |
| Initial state: x=0 ∧ y=0 | |
| Allowed: r1=1 ∧ r2=1 | |



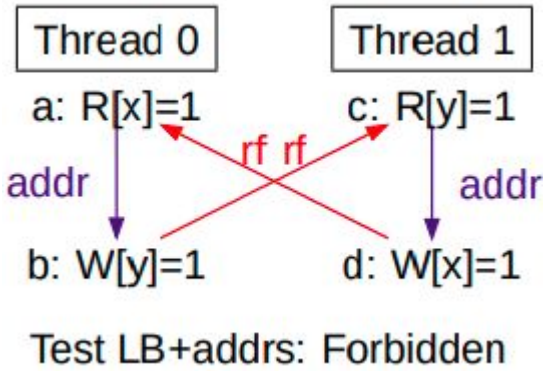
Test LB: Allowed

# LB+data

- To forbid the above condition add any read to write dependencies like data , address or control dependency to both threads

LB+addrs Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| r1=x<br>*(&y+r1-r1)=1 | r2=y<br>*(&x+r2-r2)=1 |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: r1=1 ∧ r2=1 | |



# LB+data and LB+ctrl

- Here these dependencies ensure that both writes cannot be committed until their program order preceding reads have been satisfied and committed
- There no possibility of getting “1” in the outcome in any of the register r1 , r2

LB+datas Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| r1=x<br>y=r1 | r2=y<br>x=r2 |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: r1=$m$ ∧ r2=$n$ for any $m, n \neq 0$ | |

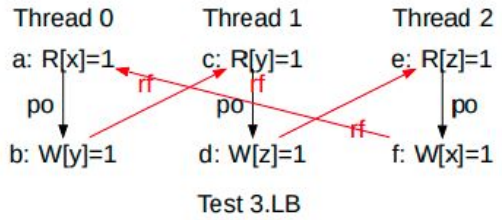
LB+ctrls Pseudocode

| Thread 0 | Thread 1 |
|---|---|
| r1=x<br>if (r1==1) y=1 | r2=y<br>if (r2==1) x=1 |
| Initial state: x=0 ∧ y=0 | |
| Forbidden: r1=$m$ ∧ r2=$n$ for any $m, n \neq 0$ | |

# 3.SB,3.2W and 3.LB

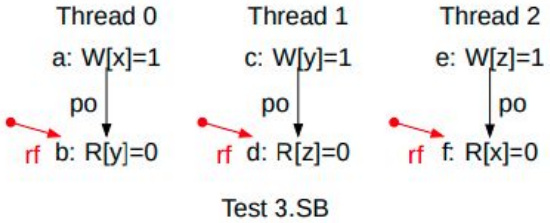
- These are the generalisations of SB, 2+2W, and LB to three threads, which need just the same as the two-thread variants.

**3.LB: rf,rf,rf** needs RWdep+RWdep+RWdep



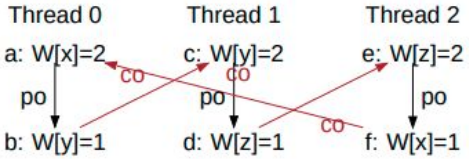

Test 3.LB

**3.SB: fr,fr,fr** needs sync+sync+sync



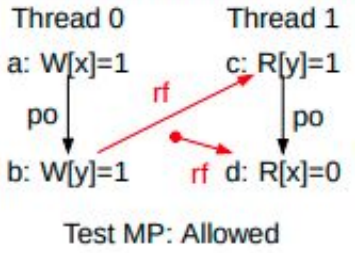
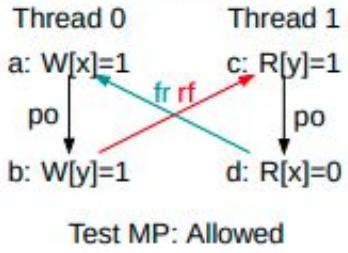
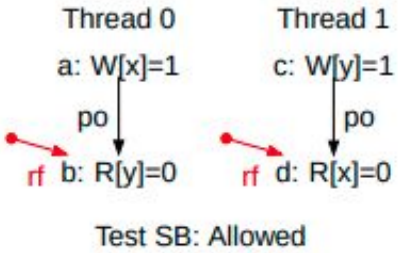
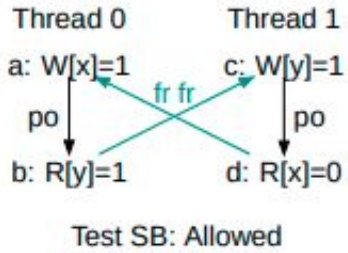

Test 3.SB

**3.2W: co,co,co** needs lwsync+lwsync+lwsync




Test 3.2W

We can replace the reads-from edges from the initial state (to some read) by the from-reads edges from that read to the write(s) to the same address, without any loss of information. For example, for MP and SB, we have:

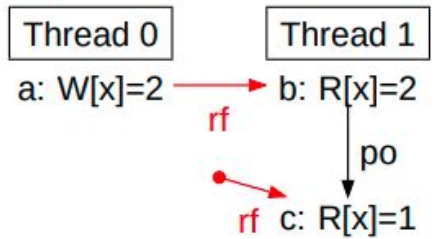
| | drawn with reads-from (rf) from initial state | drawn with from-reads (fr) |
|---|---|---|
| MP |  Test MP: Allowed |  Test MP: Allowed |
| SB |  Test SB: Allowed |  Test SB: Allowed |

# Coherence : CoRR

- Coherence means that, for any given memory location (e.g., x), all threads observe the writes to that location in the same global order.
- Lets consider coherence between two reads : CoRR
- In the given example of CoRR beside , if thread 1's first read (r1 = x) observes the value 2 (written by thread 0), but the second read (r2 = x) observes 1 (the initial value), this would violate coherence.
- Therefore the outcome where r1 = 2 and r2 = 1 is forbidden because it would imply that thread 1 is seeing the writes out of order

CoRR1 Pseudocode

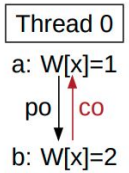
| Thread 0 | Thread 1 |
|---|---|
| x=2 | r1=x<br>r2=x |
| Initial state: x=1 | |
| Forbidden: 1:r1=2 ∧ 1:r2=1 | |




Test CoRR1: Forbidden

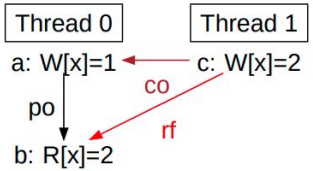
# CoWW , CoWR , CoRW , CoRW1

CoWW below shows that the coherence order must respect program order for a pair of writes by a single thread



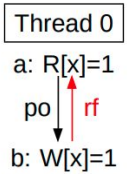

Test CoWW: Forbidden

CoWR shows that a read cannot read from a write that is coherence-hidden by another write that precedes the read on its own thread.



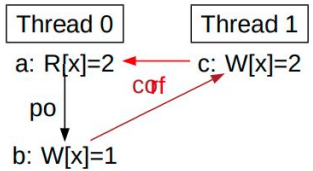

Test CoWR: Forbidden

CoRW1 shows that a read cannot read from a write that program-order follows it




Test CoRW1: Forbidden

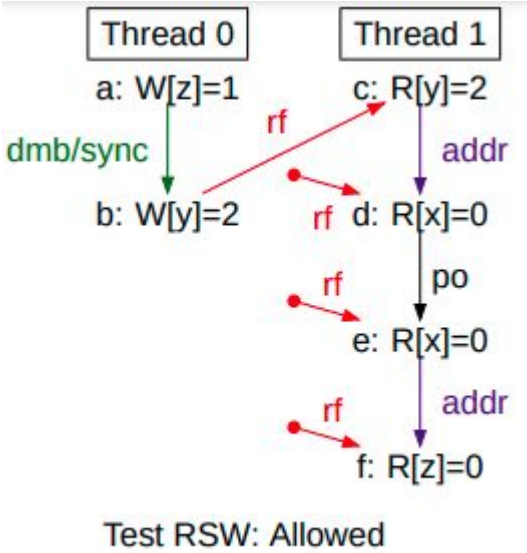
CoRW shows that a write cannot coherence-order-precede a write that a program-order-preceding read read from.




Test CoRW: Forbidden

# RSW (Read from Same Writes)

- It is a variant of MP
- The RSW variant addresses a situation where two reads (from the same thread) read from the same write.
- From beside example diagram :
    - the two reads of x, d and e, happen to read from the same write (the initial state)



# RDW (Read from DIfferent Writes)

- It is also a variant of MP
- Reads from Different Writes: In this scenario, multiple reads (d, e, etc.) depend on different preceding writes. Unlike in the Reads-from-Same-Write (RSW) case, where two reads can be satisfied from the same write (and may be reordered), here, each read has to wait for a specific write to be completed.
- From beside example diagram of RDW :
    - d and e read from different writes to x