

04 Memory Consistency Models

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

Memory Consistency Models

- A memory consistency model is
 - A set of *rules* governing how memory operations from multiple threads are processed

Memory Consistency Models

- A memory consistency model is
 - A set of *rules* governing how memory operations from multiple threads are processed
- Affects the *order* in which memory operations will appear to execute
 - determines *what value a read should return*

Memory Consistency Models

- A memory consistency model is
 - A set of *rules* governing how memory operations from multiple threads are processed
- Affects the *order* in which memory operations will appear to execute
 - determines *what value a read should return*
- A *contract* between a programmer and a system
 - Determines what optimisations can be performed for correct programs

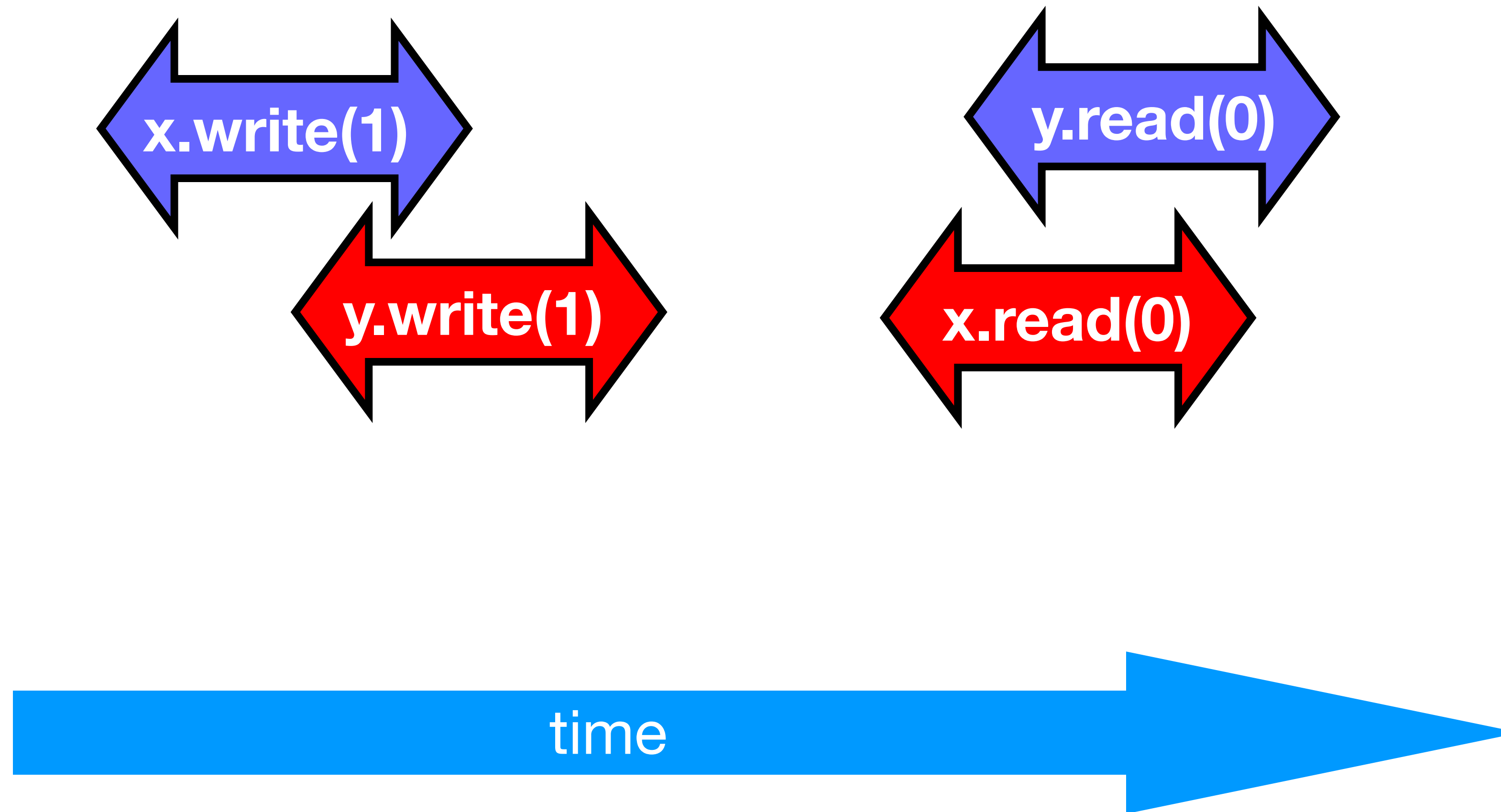
Memory Consistency Models

- A memory consistency model is
 - A set of *rules* governing how memory operations from multiple threads are processed
- Affects the *order* in which memory operations will appear to execute
 - determines *what value a read should return*
- A *contract* between a programmer and a system
 - Determines what optimisations can be performed for correct programs
- *Affects:* Ease of programming and performance

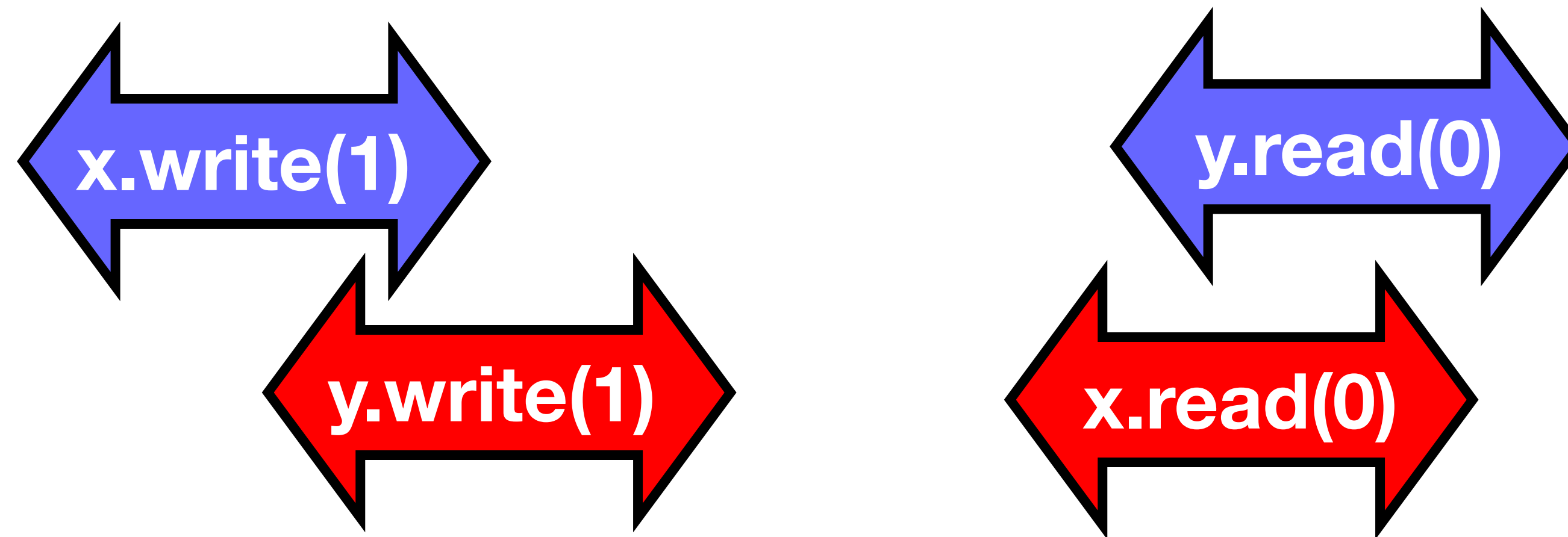
Fact

- Most hardware architectures don't support sequential consistency
- Because they think it's *too strong*
- Here's a story ...

Flag Example



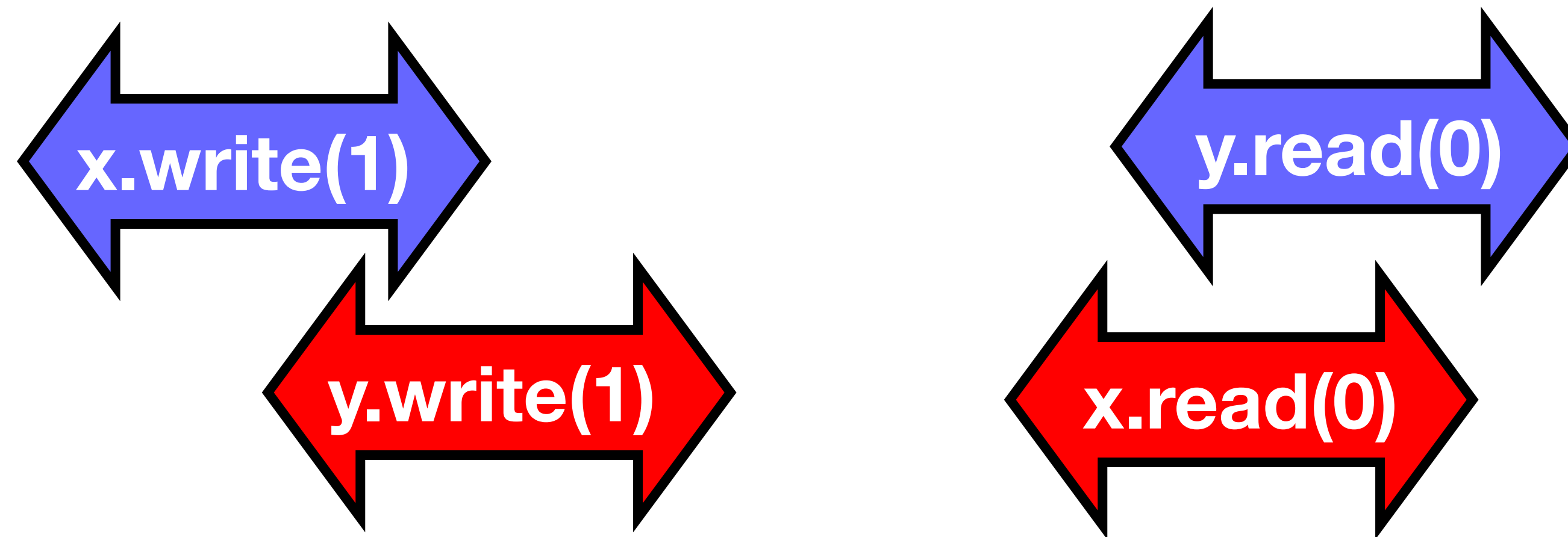
Flag Example



- Each thread's view is sequentially consistent
 - It went first



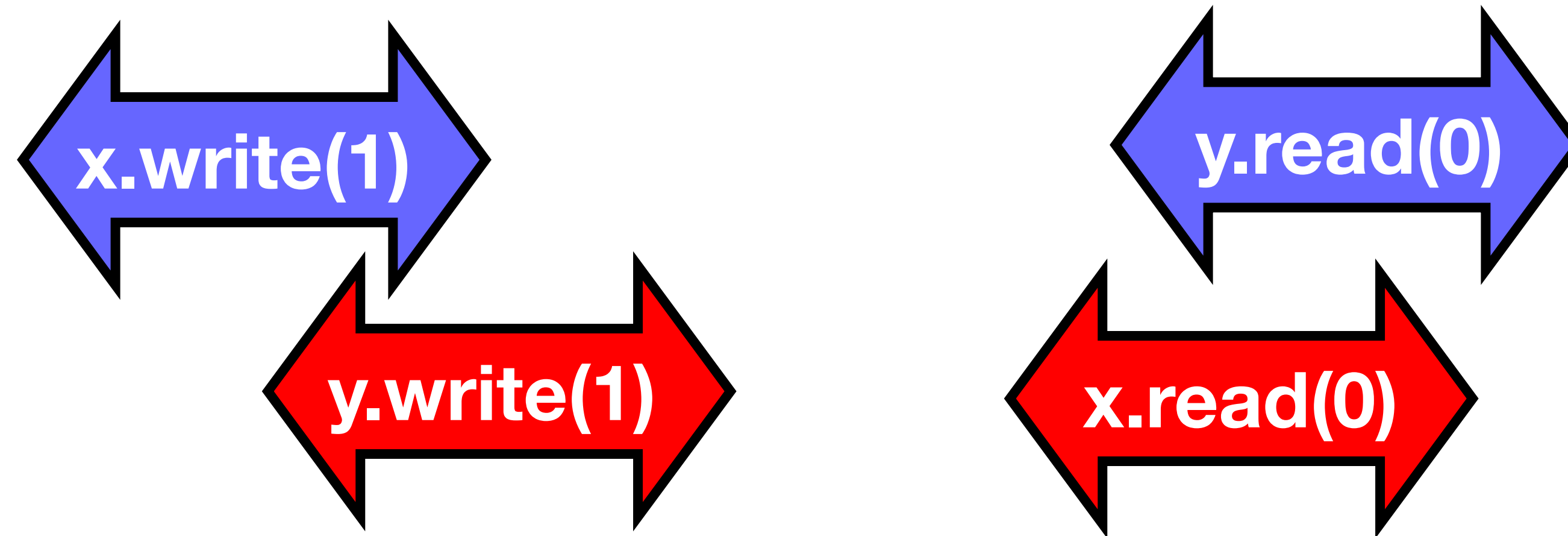
Flag Example



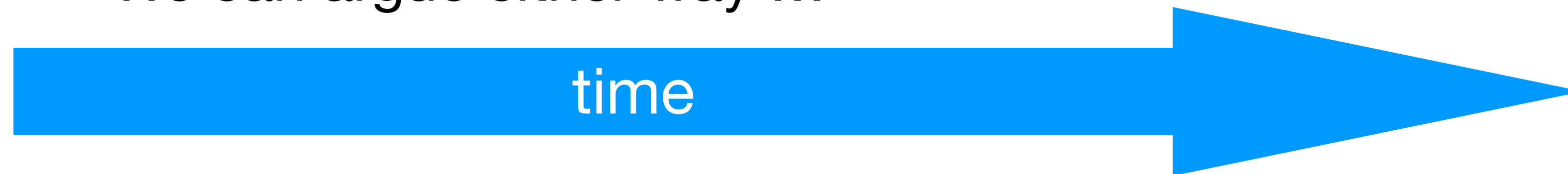
- Entire history isn't sequentially consistent
 - Can't both go first



Flag Example



- Is this behavior really so wrong?
 - We can argue either way ...



Opinion: It's Wrong

- This pattern
 - Write mine, read yours

Opinion: It's Wrong

- This pattern
 - Write mine, read yours
- Is exactly the flag principle
 - Beloved of Alice and Bob
 - Heart of mutual exclusion
 - Peterson
 - Bakery, etc.

Opinion: It's Wrong

- This pattern
 - Write mine, read yours
- Is exactly the flag principle
 - Beloved of Alice and Bob
 - Heart of mutual exclusion
 - Peterson
 - Bakery, etc.
- *It's non-negotiable!*

Peterson's Algorithm

- Combine ideas from LockOne and LockTwo

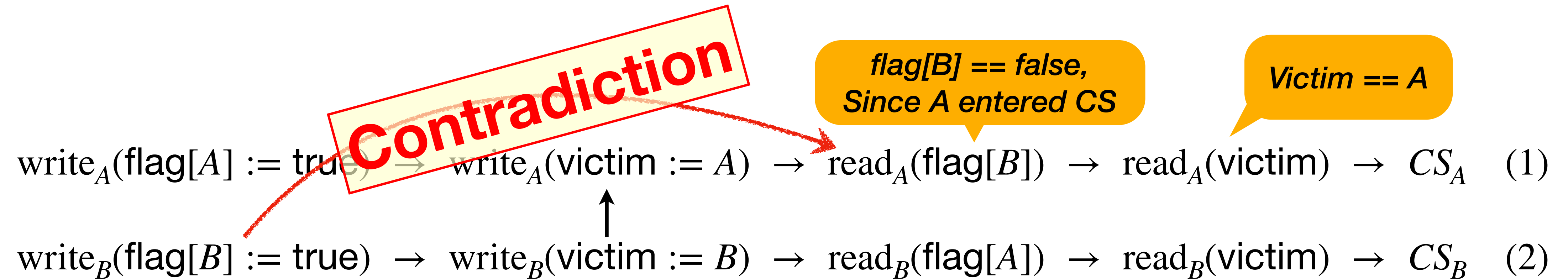
```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false

end
```

Crux of Peterson's proof



WLOG assume that A wrote to victim last

$$\text{write}_B(\text{victim} := B) \rightarrow \text{write}_A(\text{victim} := A) \quad (3)$$

Observation: proof relied on fact that if a location is stored, a later load by some thread will return this or a later stored value.

Unfortunately...

Unfortunately...

- Many hardware architects think that sequential consistency (SC) is too *strong*
- Too expensive to implement on modern hardware

Unfortunately...

- Many hardware architects think that sequential consistency (SC) is too *strong*
 - Too expensive to implement on modern hardware
- Many standard compiler optimisations are *incorrect* in the presence of multiple threads!
 - Too expensive not to optimise code

Unfortunately...

- Many hardware architects think that sequential consistency (SC) is too **strong**
 - Too expensive to implement on modern hardware
- Many standard compiler optimisations are **incorrect** in the presence of multiple threads!
 - Too expensive not to optimise code
- OK if flag principle
 - violated by **default**
 - Honoured by explicit **request**

Unfortunately...

- Many hardware architects think that sequential consistency (SC) is too **strong**
 - Too expensive to implement on modern hardware
- Many standard compiler optimisations are **incorrect** in the presence of multiple threads!
 - Too expensive not to optimise code
- OK if flag principle
 - violated by **default**
 - Honoured by explicit **request**
- **Relaxed memory models**
 - Models weaker than sequential consistency (SC)

Relaxed memory models

Relaxed memory models

- Models weaker than sequential consistency (SC)

Relaxed memory models

- Models weaker than sequential consistency (SC)
- Both hardware and programming languages go for weaker-than-SC by default
 - Still provide practical *recipes* for recovering SC

Relaxed memory models

- Models weaker than sequential consistency (SC)
- Both hardware and programming languages go for weaker-than-SC by default
 - Still provide practical *recipes* for recovering SC
- In this lecture,
 - Study one hardware (relaxed) memory model — *x86*
 - Study one programming language (relaxed) memory model — *OCaml*

Hardware Memory Consistency Models

Hardware Consistency

Initially, $a = b = 0$.

Processor 0

```
mov 1, a    ;Store  
mov b, %ebx ;Load
```

Processor 1

```
mov 1, b    ;Store  
mov a, %eax ;Load
```

Hardware Consistency

Initially, $a = b = 0$.

Processor 0

```
mov 1, a    ;Store  
mov b, %ebx ;Load
```

Processor 1

```
mov 1, b    ;Store  
mov a, %eax ;Load
```

What are the final possible values of **%*eax*** and **%*ebx*** after both processors have executed?

Hardware Consistency

Initially, $a = b = 0$.

Processor 0

```
mov 1, a    ;Store  
mov b, %ebx ;Load
```

Processor 1

```
mov 1, b    ;Store  
mov a, %eax ;Load
```

What are the final possible values of **%*eax*** and **%*ebx*** after both processors have executed?

Sequential consistency implies that no execution ends with **%*eax* = %*ebx* = 0**

Hardware Consistency

- **No** modern-day processor implements sequential consistency.
 - Hardware actively reorders instructions.

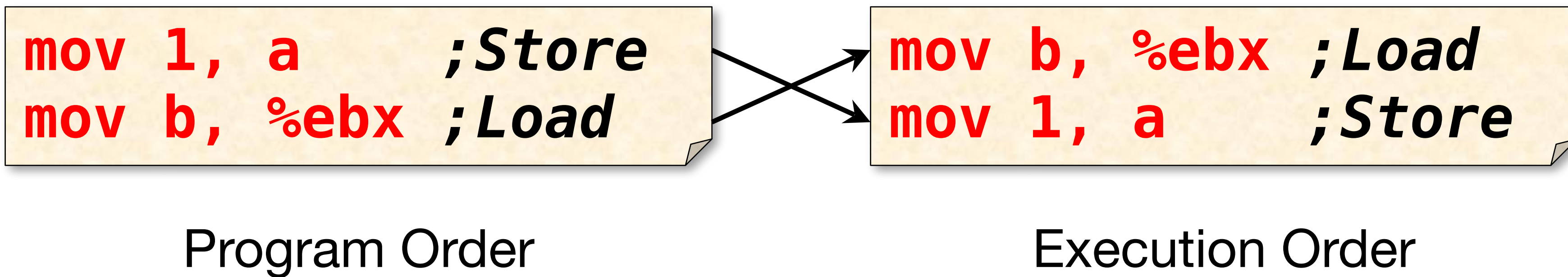
Hardware Consistency

- **No** modern-day processor implements sequential consistency.
 - Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
 - Why?

Hardware Consistency

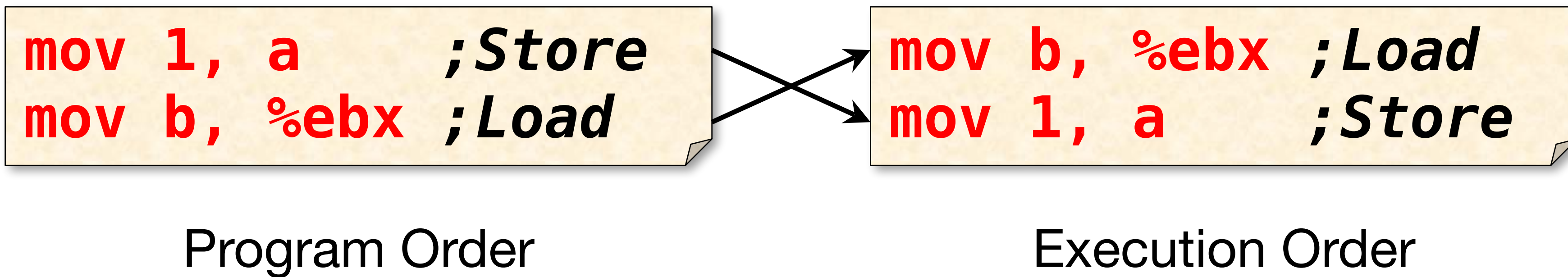
- **No** modern-day processor implements sequential consistency.
 - Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
 - Why?
- Most of the performance is derived from a single thread's unsynchronised execution of code.

Instruction Reordering



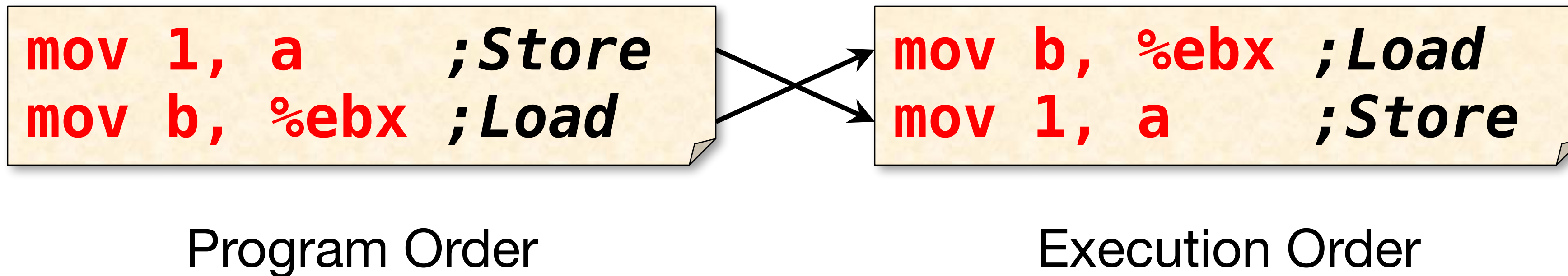
Q. Why might the hardware or compiler decide to reorder these instructions?

Instruction Reordering



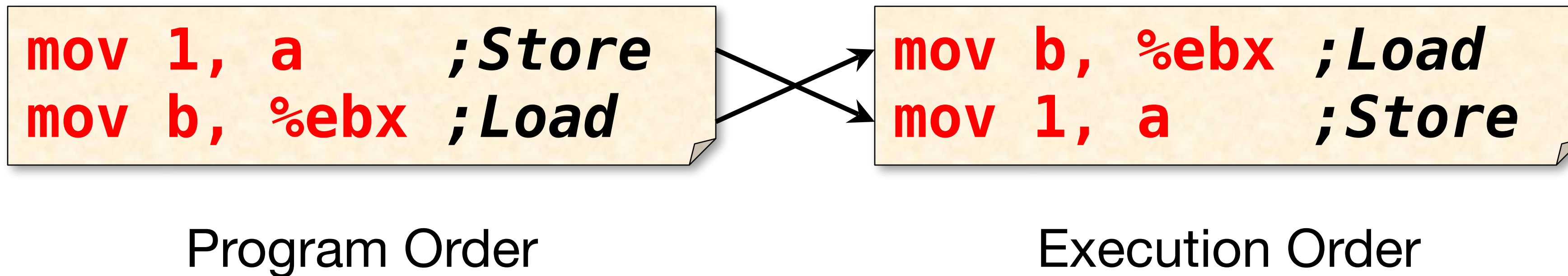
- Q. Why might the hardware or compiler decide to reorder these instructions?
- A. To obtain higher performance by covering load latency — ***instruction-level parallelism.***

Instruction Reordering



Q. When is it safe for the hardware or compiler to perform this reordering?

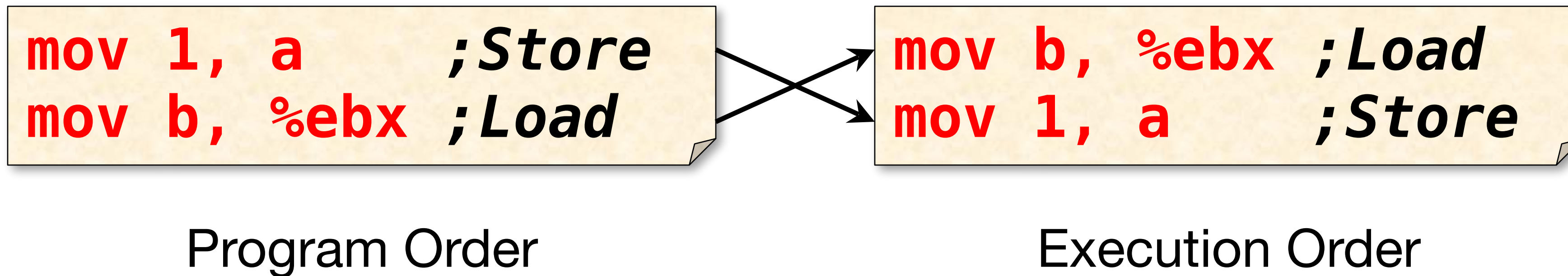
Instruction Reordering



Q. When is it safe for the hardware or compiler to perform this reordering?

A. When $a \neq b$.

Instruction Reordering

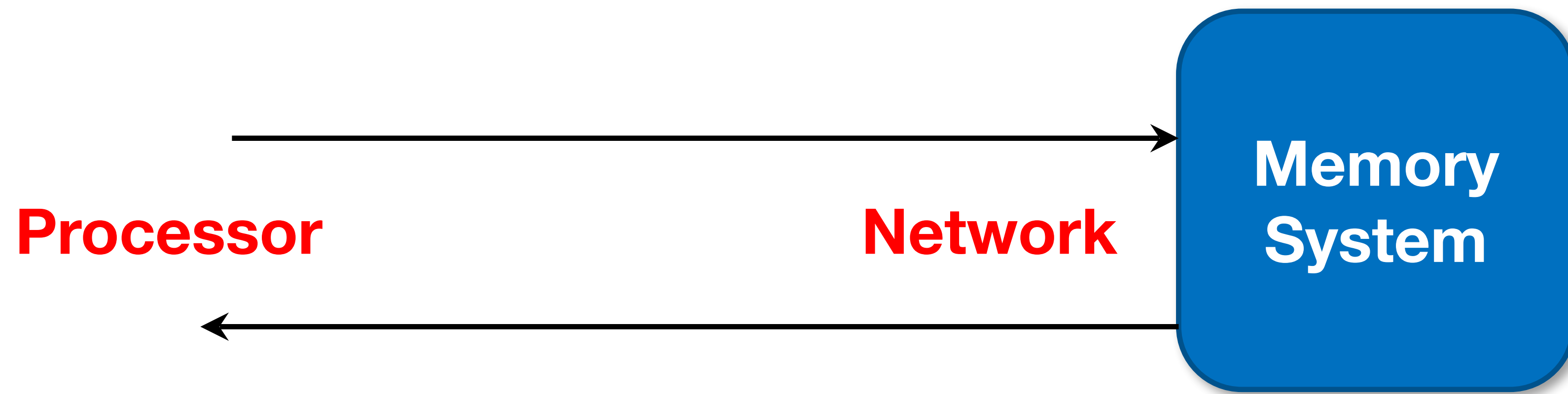


Q. When is it safe for the hardware or compiler to perform this reordering?

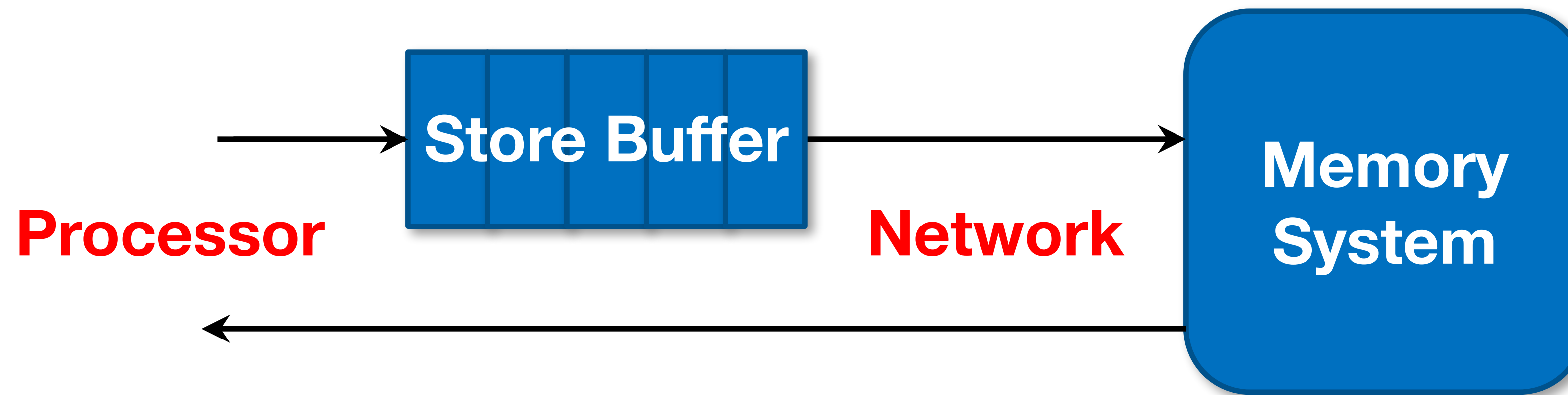
A. When $a \neq b$.

A'. And there's no concurrency.

Hardware Reordering

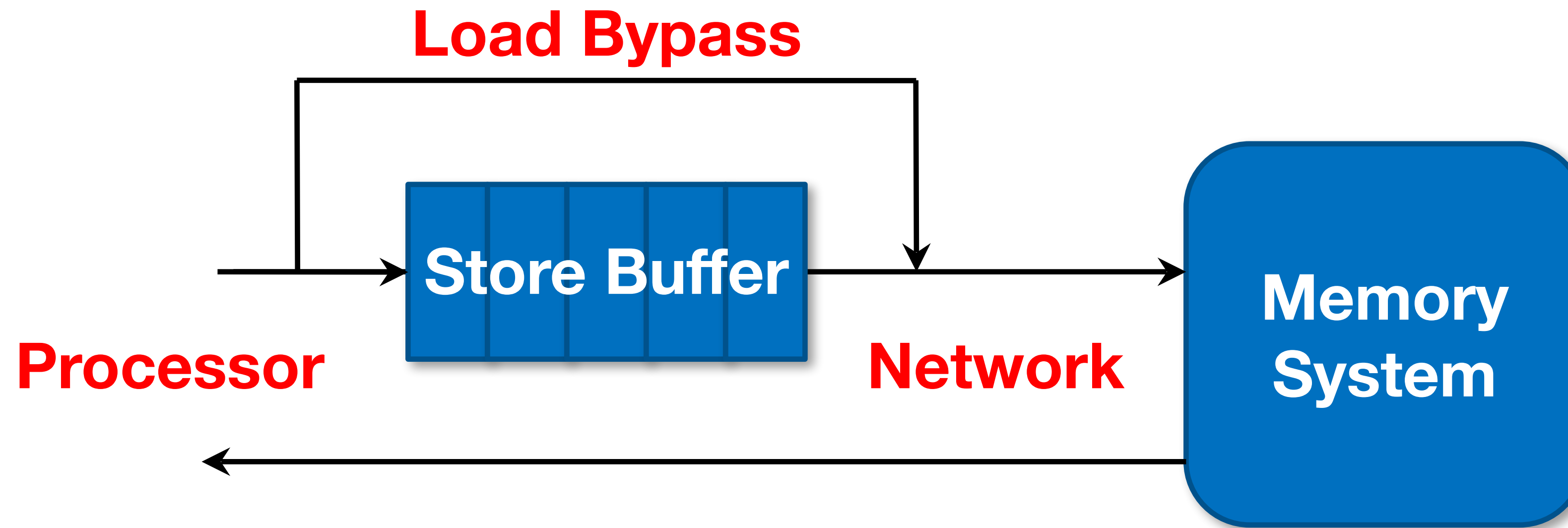


Hardware Reordering



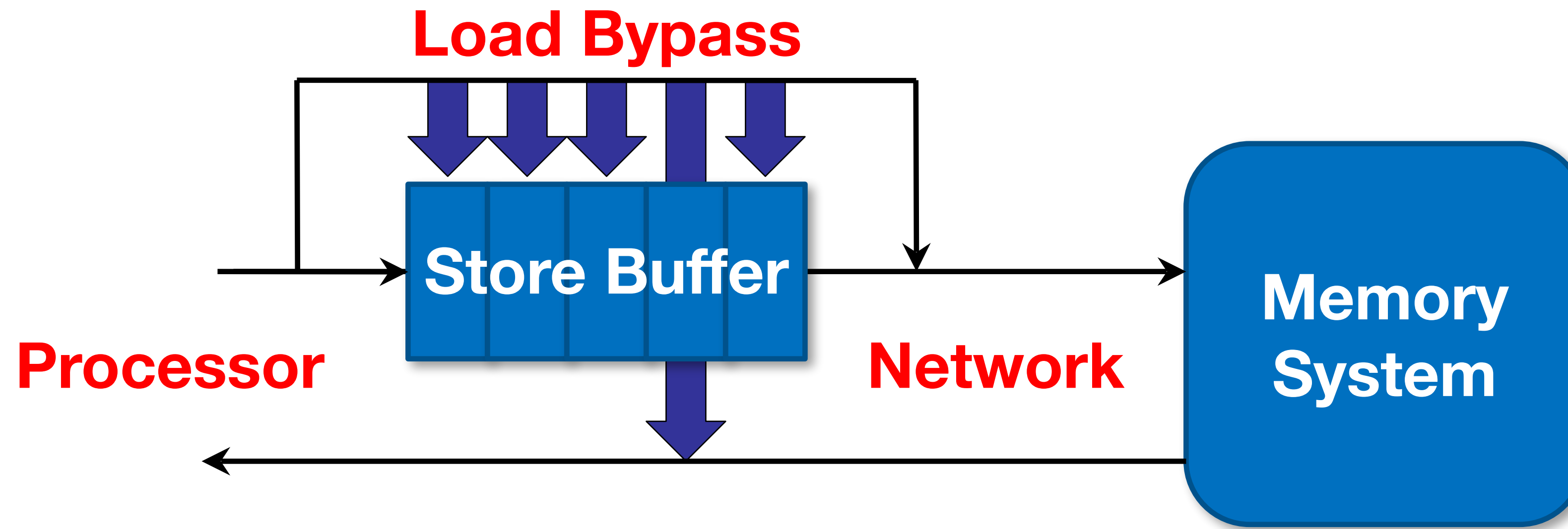
- Processor can issue stores faster than the network can handle them \Rightarrow store buffer

Hardware Reordering



- Processor can issue stores faster than the network can handle them \Rightarrow store buffer
- Loads take priority, bypassing the store buffer.

Hardware Reordering



- Processor can issue stores faster than the network can handle them \Rightarrow store buffer
- Loads take priority, bypassing the store buffer.
- Except if a load address matches an address in the store buffer, the store buffer returns the result.

X86 Memory Consistency

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

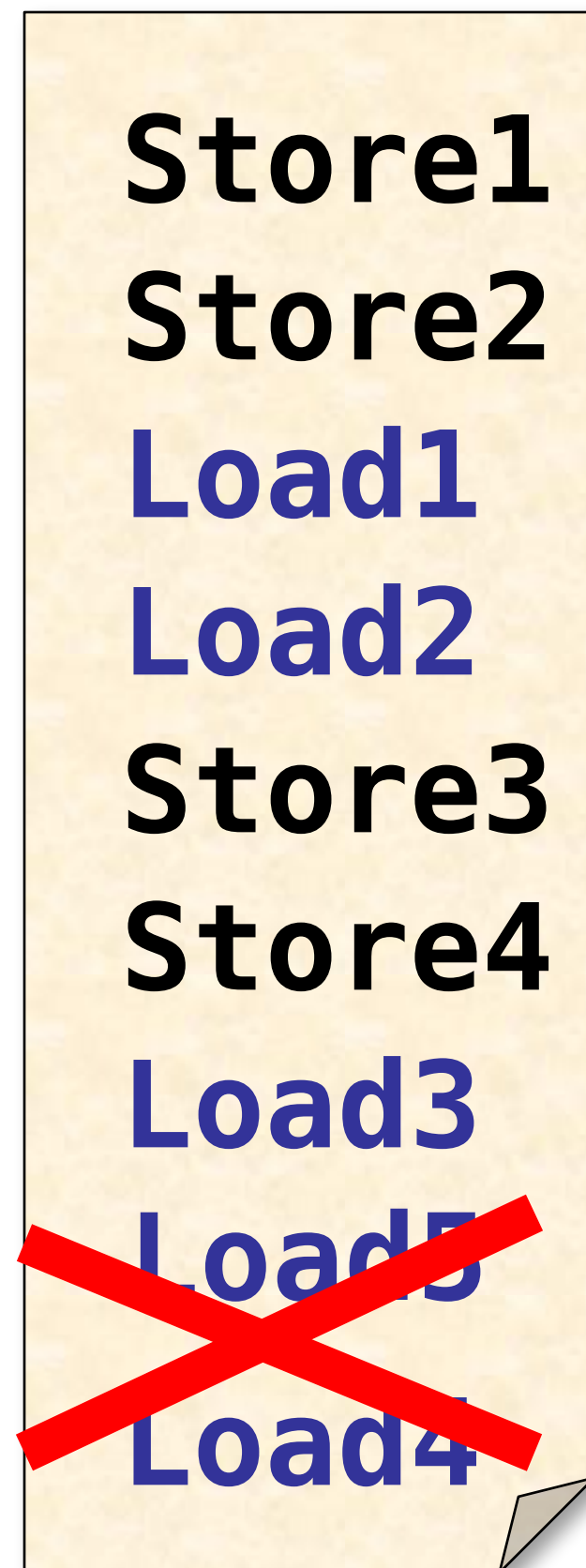
Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load5
Load4

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

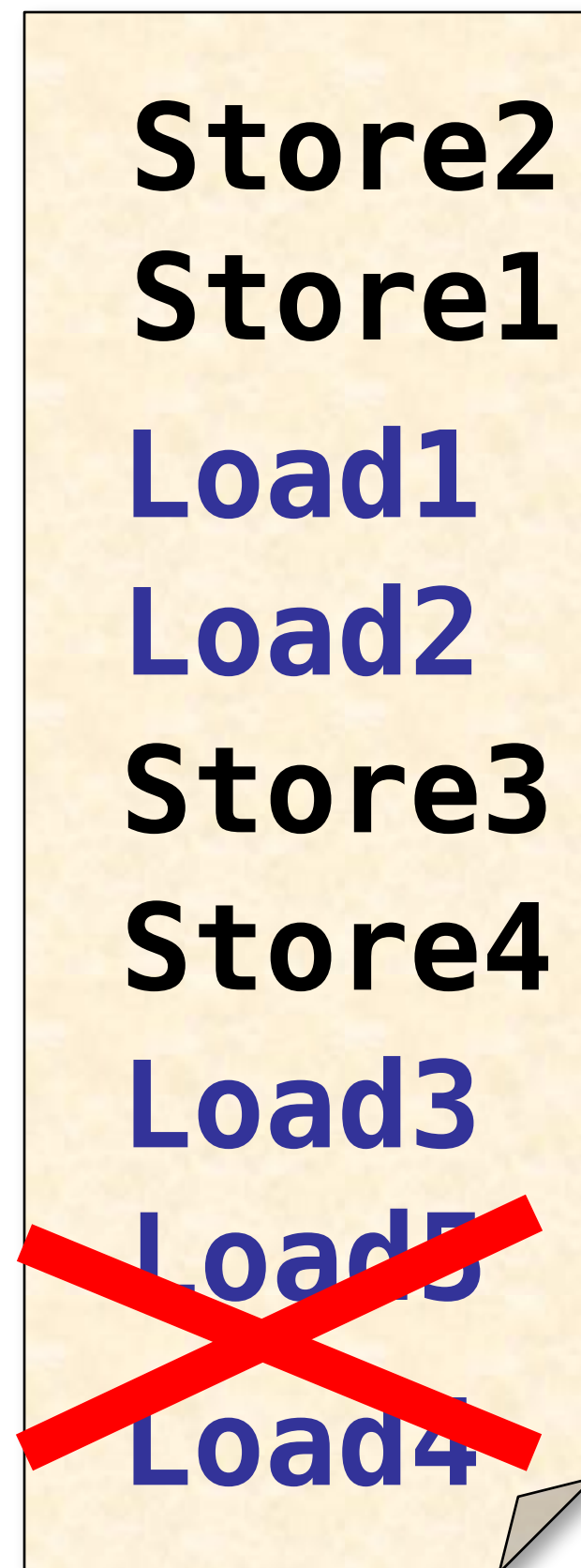


Store1
Store2
Load1
Load2
Store3
Store4
Load3
~~Load5~~
~~Load4~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

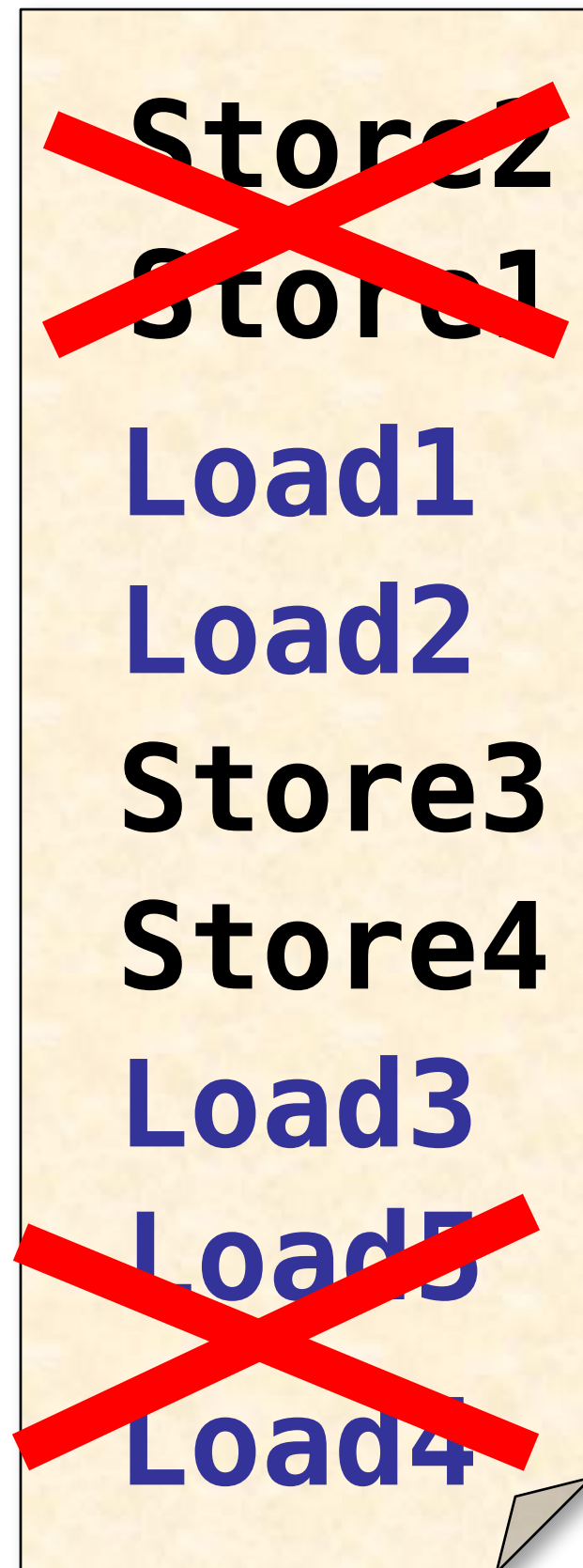


Store2
Store1
Load1
Load2
Store3
Store4
Load3
~~Load5~~
~~Load4~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

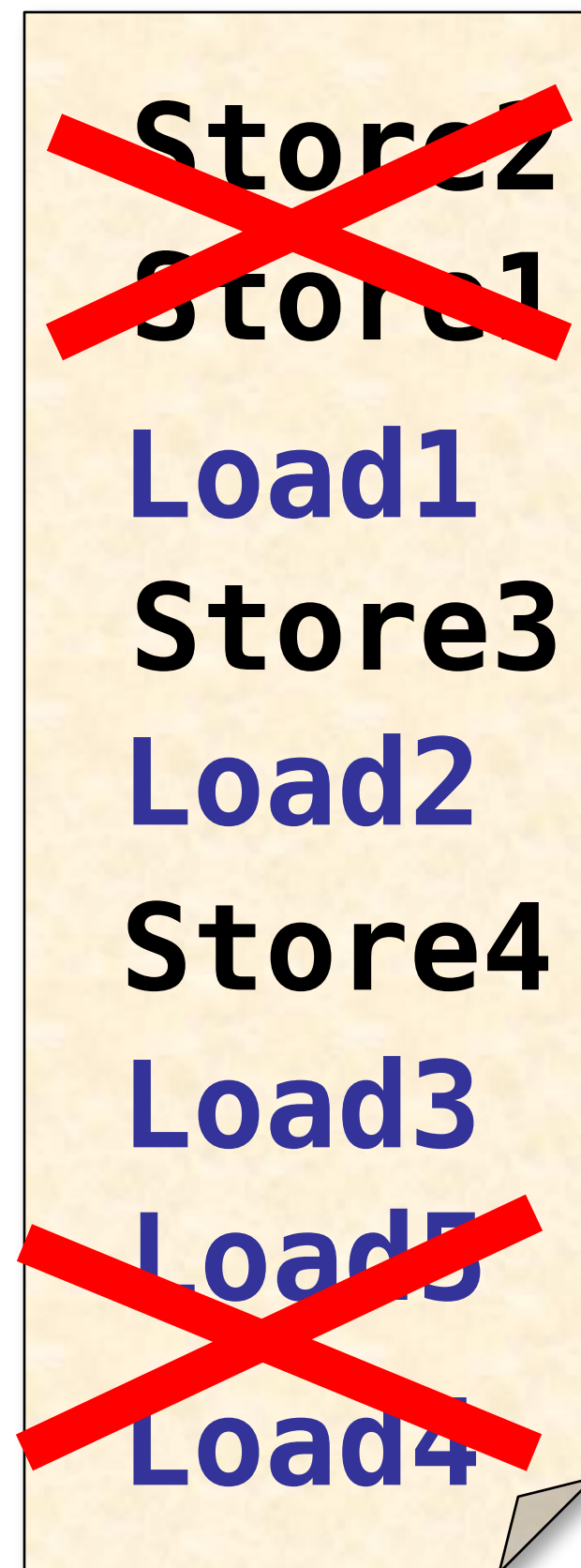


~~Store2~~
~~Store1~~
Load1
Load2
Store3
Store4
Load3
~~Load5~~
~~Load4~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

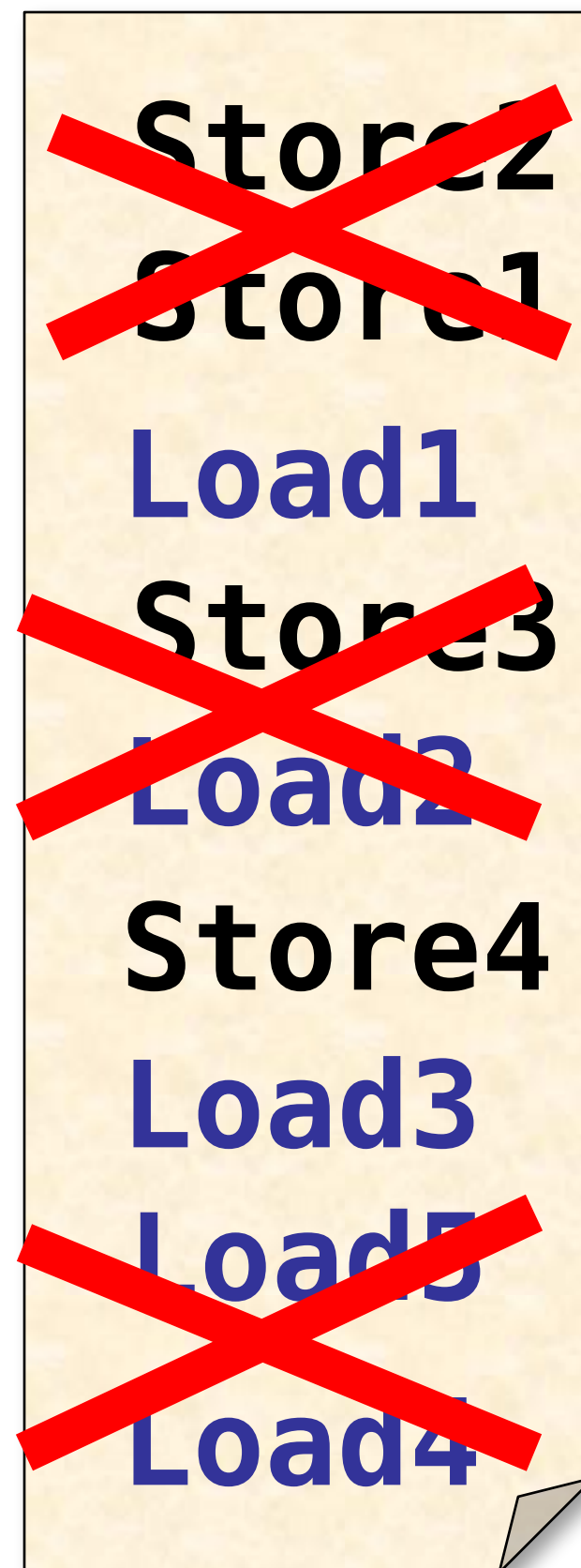


~~Store2~~
~~Store1~~
Load1
Store3
Load2
Store4
Load3
~~Load5~~
~~Load4~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code



~~Store2~~
~~Store1~~
Load1
~~Store3~~
~~Load2~~
Store4
Load3
~~Load5~~
~~Load4~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

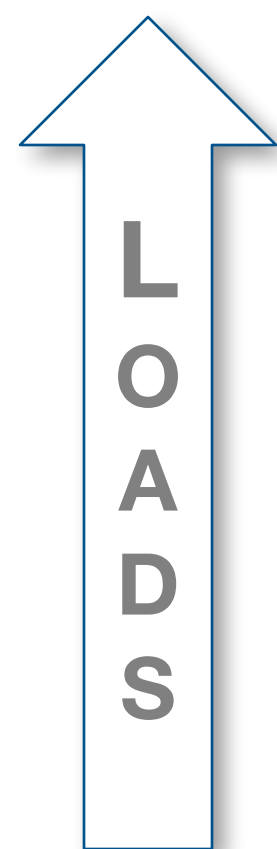
Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

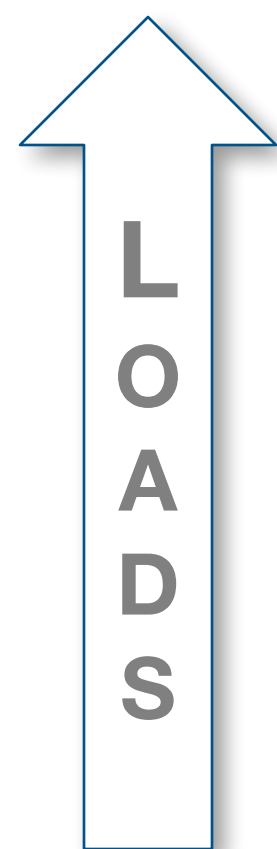


Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5

1. Loads **are not** reordered with loads.
2. Stores **are not** reordered with stores.
3. Stores **are not** reordered with prior loads.
4. A load **may** be reordered with a prior store to a different location **but not** with a prior store to the same location.
5. Stores to the same location **respect a global total order**.

X86 Memory Consistency

Thread's Code

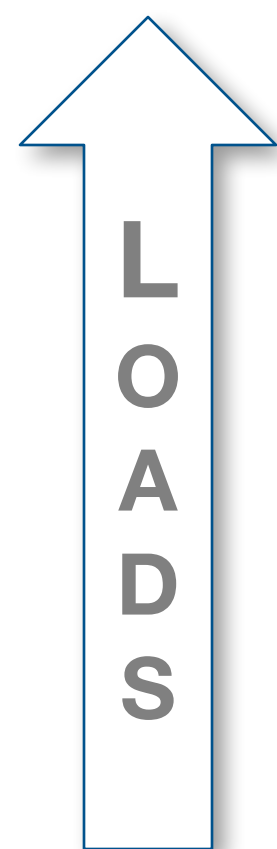


Store1
Store2
Load1
Load2
Load3
Store3
Load4
Store4
Load5

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code



Store1
Store2
Load1
Load2
Load3
Store3
Load4
Store4
Load5

OK!

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

X86 Memory Consistency

Thread's Code

L
O
A
D
S

Store1
Store2
Load1
Load2
Load3
Store3
Load4
Store4
Load5

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. **Total Store Ordering (TSO)...weaker than sequential consistency**
4. Stores to different locations *are not* reordered with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

OK!

Memory Barriers (Fences)

Memory Barriers (Fences)

- *A memory barrier (or memory fence)*
 - is a hardware action that
 - enforces an ordering constraint between the instructions before and after the fence.

Memory Barriers (Fences)

- *A memory barrier (or memory fence)*
 - is a hardware action that
 - enforces an ordering constraint between the instructions before and after the fence.
- A memory barrier can be issued explicitly as an instruction
 - (x86: mfence)

Memory Barriers (Fences)

- *A memory barrier (or memory fence)*
 - is a hardware action that
 - enforces an ordering constraint between the instructions before and after the fence.
- A memory barrier can be issued explicitly as an instruction
 - (x86: mfence)
- The typical cost of a memory fence is comparable to that of an L2-cache access.

X86 Memory Consistency

Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5

1. Loads **are not** reordered with loads.

2. Sto

3. Sto

load

4. A lo

store to

with a prior store to the same location.

5. Stores to the same location **respect a global total order.**

**Total Store Ordering +
properly placed memory
barriers = sequential
consistency**

X86 Memory Consistency

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5

1. Loads **are not** reordered with loads.

2. Sto

3. Sto

load

4. A lo

store to

with a prior store to the same location.

5. Stores to the same location **respect a global total order.**

**Total Store Ordering +
properly placed memory
barriers = sequential
consistency**

Memory Barriers

- Explicit Synchronization
- Memory *barrier* will
 - Flush write buffer
 - Bring caches up to date
- Compilers often do this for you
 - Entering and leaving critical sections

Summary: Hardware Consistency

Summary: Hardware Consistency

- Hardware memory models are weaker than sequential consistency
 - They show up as reordering of instruction sequences on a single thread

Summary: Hardware Consistency

- Hardware memory models are weaker than sequential consistency
 - They show up as reordering of instruction sequences on a single thread
- X86 uses Total Store Order (TSO)
 - ARM, RISC-V and Power uses ***even weaker models!***

Summary: Hardware Consistency

- Hardware memory models are weaker than sequential consistency
 - They show up as reordering of instruction sequences on a single thread
- X86 uses Total Store Order (TSO)
 - ARM, RISC-V and Power uses ***even weaker models!***
- Instruction Set Architectures (ISAs) provide fences/memory barriers to restrict reordering

Summary: Hardware Consistency

- Hardware memory models are weaker than sequential consistency
 - They show up as reordering of instruction sequences on a single thread
- X86 uses Total Store Order (TSO)
- ARM, RISC-V and Power uses ***even weaker models!***
- Instruction Set Architectures (ISAs) provide fences/memory barriers to restrict reordering

What does it mean for concurrent programming languages?

PL Memory Consistency Models

<https://ocaml.org/manual/5.4/memorymodel.html>

PL Consistency

- We saw that hardware architectures do not provide SC
 - Programming languages compile to hardware
 - Hence, provide weaker than SC model
- In addition, ***compiler optimisations also violate SC!***

Example

Initially $!a = !b = 1$

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

Example

Initially !a = !b = 1

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

- What are the possible outcomes for r1, r2 and r3?

Example

Initially $!a = !b = 1$

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

- What are the possible outcomes for $r1$, $r2$ and $r3$?
- $r1 = 2$, $r2 = 1$, $r3 = 2$
 - *if the $t1$ reads b before the $t2$ writes 0 to it*

Example

Initially $!a = !b = 1$

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

- What are the possible outcomes for $r1$, $r2$ and $r3$?
- $r1 = 2$, $r2 = 1$, $r3 = 2$
 - *if the $t1$ reads b before the $t2$ writes 0 to it*
- $r1 = 2$, $r2 = 0$, $r3 = 2$
 - *if the $t1$ reads b after the $t2$ writes 0 to it*

Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```


Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

- What are the possible outcomes for r1, r2 and r3?

Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

- What are the possible outcomes for r1, r2 and r3?
- r1 = 2, r2 = 1, r3 = 2
 - All reads happen before the write (ab = 1 throughout)

Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

- What are the possible outcomes for r1, r2 and r3?
- r1 = 2, r2 = 1, r3 = 2
 - All reads happen before the write (ab = 1 throughout)
- r1 = 2, r2 = 0, r3 = 0
 - First read before write, then write happens, and remaining reads after write

Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

- What are the possible outcomes for r1, r2 and r3?
- r1 = 2, r2 = 1, r3 = 2
 - All reads happen before the write (ab = 1 throughout)
- r1 = 2, r2 = 0, r3 = 0
 - First read before write, then write happens, and remaining reads after write
- r1 = 0, r2 = 0, r3 = 0
 - All reads happen after the write (ab = 0 throughout)

Enter aliasing

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

*Will this assertion
ever fail?*

- What are the possible outcomes for r1, r2 and r3?
- r1 = 2, r2 = 1, r3 = 2
 - All reads happen before the write (ab = 1 throughout)
- r1 = 2, r2 = 0, r3 = 0
 - First read before write, then write happens, and remaining reads after write
- r1 = 0, r2 = 0, r3 = 0
 - All reads happen after the write (ab = 0 throughout)

Common Subexpression Elimination (CSE)

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = r1 in (* CSE: !a * 2 ==> r1 *)  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

*a and b are
aliases*

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*Will this assertion
ever fail?*

Common Subexpression Elimination (CSE)

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = r1 in (* CSE: !a * 2 ==> r1 *)  
  (r1, r2, r3)
```

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

*a and b are
aliases*

*Will this assertion
ever fail?*

- $r1 = 2, r2 = 0, r3 = 2$
- First read before write, then write happens, and the remaining read after write

Common Subexpression Elimination (CSE)

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r2 = !b in  
  let r3 = !a * 2 in  
(r1, r2, r3)
```

```
L104: ldr x2, [x0, #0] ; Load !a (ONCE)  
L106: add x4, x3, x2, lsl #1 ; r1 = !a * 2  
L107: ldr x5, [x1, #0] ; Load !b (r2)  
  
L114: str x4, [x0, #0] ; Store r1  
L115: str x5, [x0, #8] ; Store r2  
L116: str x4, [x0, #16] ; Store r3 (SAME x4!)
```

```
; CSE: !a loaded once, x4 reused for r1 and r3  
; No second load or computation for r3
```

OCaml

Assembly

Explaining this behaviour

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r3 = !a * 2 in  
  let r2 = !b in  
  (r1, r2, r3)
```

Reordering

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

Explaining this behaviour

```
let t1 a b =  
  let r1 = !a * 2 in  
  let r3 = !a * 2 in  
  let r2 = !b in  
  (r1, r2, r3)
```

Reordering

```
let t2 b = b := 0
```

```
let main () =  
  let ab = ref 1 in  
  let h = Domain.spawn (fun _ ->  
    let r1, r2, r3 = t1 ab ab in  
    assert (not (r1 = 2 && r2 = 0 && r3 = 2)))  
  in  
  t2 ab;  
  Domain.join h
```

- $r1 = 2, r2 = 0, r3 = 2$
- Write happens after the first two reads

OCaml Relaxed Memory Model

OCaml Relaxed Memory Model

- Precisely describes what behaviours can be *observed*
 - Analogous to x86 TSO, but for the source language

OCaml Relaxed Memory Model

- Precisely describes what behaviours can be *observed*
 - Analogous to x86 TSO, but for the source language
- Compiler and hardware can *optimise* given they respect OCaml memory model

OCaml Relaxed Memory Model

- Precisely describes what behaviours can be *observed*
 - Analogous to x86 TSO, but for the source language
- Compiler and hardware can *optimise* given they respect OCaml memory model
- OCaml also provides *language constructs* to enforce SC

Atomic locations

Atomic locations

- *Non-atomic* locations are default
 - Reference cells, array fields, mutable record fields, ...
 - Immutable locations only have an *initialising write* and only reads afterwards

Atomic locations

- **Non-atomic** locations are default
 - Reference cells, array fields, mutable record fields, ...
 - Immutable locations only have an *initialising write* and only reads afterwards
- **Atomic** locations are created using the `Atomic` module

Module `Atomic`

```
module Atomic: sig .. end
```

Atomic references.

See [the examples](#) below. See 'Memory model: The hard bits' chapter in the manual.

Since 4.12

```
type 'a t
```

An atomic (mutable) reference to a value of type `'a`.

```
val make : 'a -> 'a t
```

Create an atomic reference.

```
val get : 'a t -> 'a
```

Get the current value of the atomic reference.

```
val set : 'a t -> 'a -> unit
```

Set a new value for the atomic reference.

Atomic locations

- **Non-atomic** locations are default
 - Reference cells, array fields, mutable record fields, ...
 - Immutable locations only have an *initialising write* and only reads afterwards
- **Atomic** locations are created using the `Atomic` module
- They're like reference cells in OCaml, but something more...
 - Include memory fences to enforce ordering against hardware memory model
 - Disables unsafe compiler optimisations that break SC

Module **Atomic**

```
module Atomic: sig .. end
```

Atomic references.

See [the examples](#) below. See 'Memory model: The hard bits' chapter in the manual.

Since 4.12

```
type !'a t
```

An atomic (mutable) reference to a value of type `'a`.

```
val make : 'a -> 'a t
```

Create an atomic reference.

```
val get : 'a t -> 'a
```

Get the current value of the atomic reference.

```
val set : 'a t -> 'a -> unit
```

Set a new value for the atomic reference.

Reasoning about OCaml memory model

- How do we understand the behaviour of OCaml programs under the relaxed memory model?
- **Events**
 - Reads and writes of atomic and non-atomic locations
 - Spawn and join of domains
 - Operations on mutexes

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain

1: `let t1 a b =`

2: `let r1 = !a * 2 in`

3: `let r2 = !b in`

4: `let r3 = !a * 2 in`

5: `(r1, r2, r3)`

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain

```
1: let t1 a b =  
2:   let r1 = !a * 2 in  
3:   let r2 = !b in  
4:   let r3 = !a * 2 in  
5:   (r1, r2, r3)
```

- $R(a) @ 2 \xrightarrow{hb} R(b) @ 3$

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain

1: `let t1 a b =`

2: `let r1 = !a * 2 in`

3: `let r2 = !b in`

4: `let r3 = !a * 2 in`

5: `(r1, r2, r3)`

- $R(a) @ 2 \xrightarrow{hb} R(b) @ 3$
- $R(b) @ 3 \xrightarrow{hb} R(a) @ 4$

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain

1: `let t1 a b =`

2: `let r1 = !a * 2 in`

3: `let r2 = !b in`

4: `let r3 = !a * 2 in`

5: `(r1, r2, r3)`

- $R(a) @ 2 \xrightarrow{hb} R(b) @ 3$
- $R(b) @ 3 \xrightarrow{hb} R(a) @ 4$
- $R(a) @ 2 \xrightarrow{hb} R(a) @ 4$

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain
 - On an atomic location, a write and a subsequent
 - Read that sees the written value or
 - Write that overwrites the original write

```
let a = Atomic.make 1
```

```
let t1 () = Atomic.set a 2
```

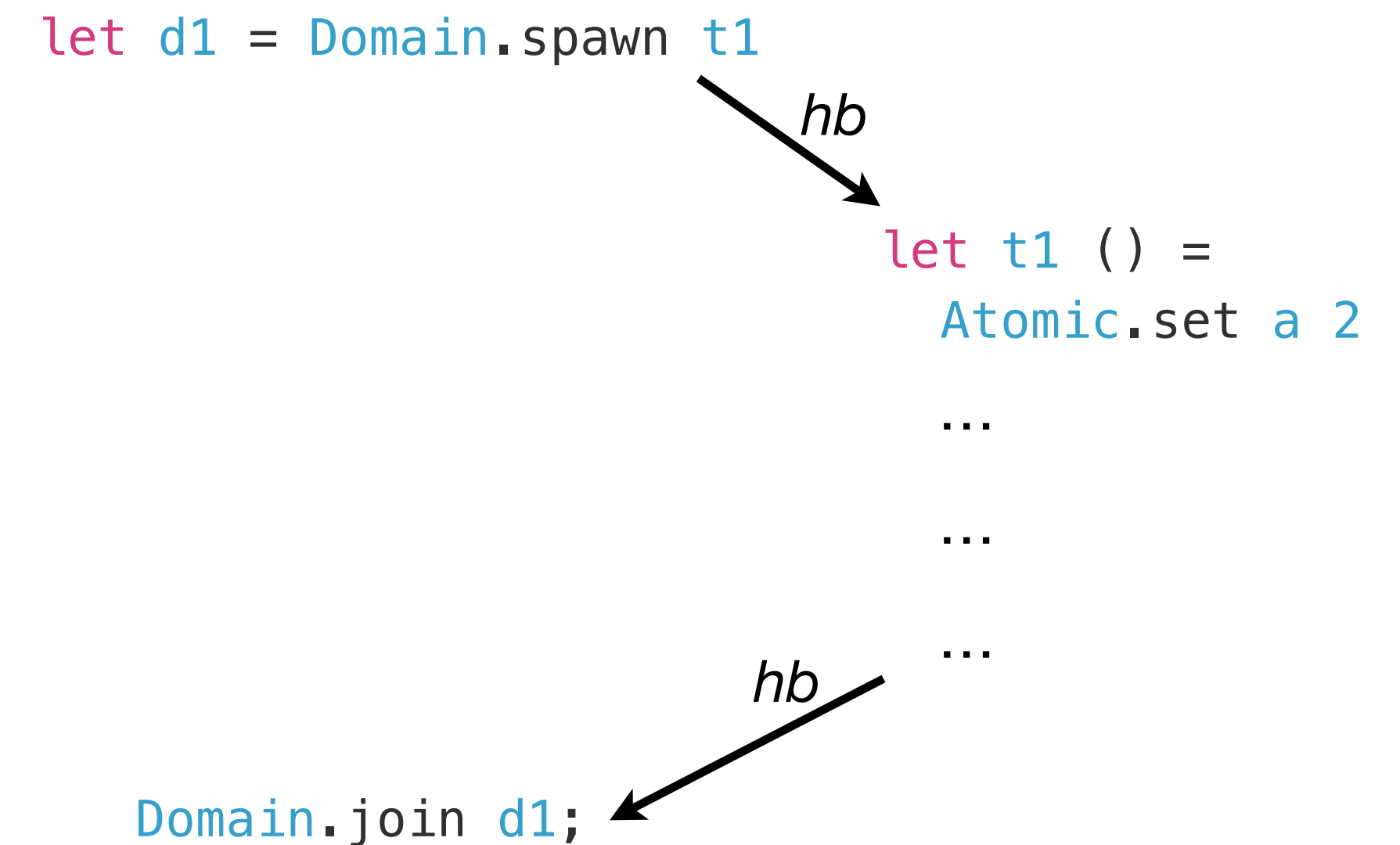
```
let t2 () = Atomic.get a
```

```
let t3 () = Atomic.get a
```

- $R(a) @ t1 \xrightarrow{hb} R(a) @ t2$
- $R(a) @ t1 \xrightarrow{hb} R(a) @ t3$

Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain
 - On an atomic location, a write and a subsequent
 - Read that sees the written value or
 - Write that overwrites the original write
 - `Domain.spawn` and the first event of the newly spawned domain
 - Last event of domain and `Domain.join`



Happens-before relation

- *Happens-before relation* is a *reflexive, transitive closure* of
 - *Program order* — order in which events appear in a given domain
 - On an atomic location, a write and a subsequent
 - Read that sees the written value or
 - Write that overwrites the original write
 - `Domain.spawn` and the first event of the newly spawned domain
 - Last event of domain and `Domain.join`
 - `Unlock` of a mutex and subsequent `Lock`

```
let t1 m =  
  Mutex.lock m;  
  (* .... *)  
  (* .... *)  
  Mutex.unlock m;  
                                     hb  
                                     ↘  
let t2 m =  
  Mutex.lock m;  
  (* .... *)  
  (* .... *)  
  Mutex.unlock m;
```

Data race

Data race

- In a given execution, two actions are said to be **conflicting** if
 - At least one of them is a **write**
 - Neither is an **initialising write**
- An execution is said to have a **data race**
 - There is a pair of **conflicting** actions
 - There is no **happens-before** relationship between them
- Conflicting memory accesses without happens-before are said to be **unsynchronised**
- A program is said to have a data race if there exists at least one execution which has a data race
 - A program is said to be **correctly synchronised** if it doesn't have a data race

Why does this matter?

Why does this matter?

- **Programs with data races**
 - In C/C++, programs with data races have *undefined behaviour*
 - Can do anything at all — crash, leak secrets, launch missiles, explode, ...
 - In OCaml, programs with data races have *well-defined behaviour*
 - Type-safety preserved
 - Semantics are more complicated; you can see stale writes

Why does this matter?

- **Programs with data races**

- In C/C++, programs with data races have *undefined behaviour*
 - Can do anything at all — crash, leak secrets, launch missiles, explode, ...
- In OCaml, programs with data races have *well-defined behaviour*
 - Type-safety preserved
 - Semantics are more complicated; you can see stale writes

- **Programs without data races**

- In C/C++, Java, OCaml, etc, programs without data races are sequentially consistent.
- Data-race-free sequential consistency (DRF-SC) model
- *No need to reason about reorderings!*

Does this have a data race?

- Consider that the functions d^* run in parallel

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
type t = {
  mutable a : int;
  mutable b : int
}
let r = {a = 0; b = 1}
let d1 () = r.a <- 42
let d2 () = r.b <- 42
```

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

No data race

```
type t = {
  mutable a : int;
  mutable b : int
}
let r = {a = 0; b = 1}
let d1 () = r.a <- 42
let d2 () = r.b <- 42
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
type t = {
  mutable a : int;
  mutable b : int
}
let r = {a = 0; b = 1}
let d1 () = r.a <- 42
let d2 () = r.b <- 42
```

No data race

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

No data race

```
let r = Atomic.make 0
let d1 () = Atomic.set r 1
let d2 () = Atomic.get r
```


Does this have a data race?

- Consider that the functions d^* run in parallel

```
let r = ref 0
let d1 () = r := 1
let d2 () = !r
```

Data race

```
type t = {
  mutable a : int;
  mutable b : int
}
let r = {a = 0; b = 1}
let d1 () = r.a <- 42
let d2 () = r.b <- 42
```

No data race

```
let a = [| 0; 1 |]
let d1 () = a.(0) <- 42
let d2 () = a.(1) <- 42
```

No data race

```
let r = Atomic.make 0
let d1 () = Atomic.set r 1
let d2 () = Atomic.get r
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  if Atomic.get flag (* c *) then
    !msg (* d *)
  else 0
```

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  if Atomic.get flag (* c *) then
    !msg (* d *)
  else 0
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  if Atomic.get flag (* c *) then
    !msg (* d *)
  else 0
```

No data race

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  ignore (Atomic.get flag); (* c *)
  !msg (* d *)
```

Does this have a data race?

- Consider that the functions d* run in parallel

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  if Atomic.get flag (* c *) then
    !msg (* d *)
  else 0
```

No data race

```
let msg = ref 0
let flag = Atomic.make false
let d1 () =
  msg := 42; (* a *)
  Atomic.set flag true (* b *)
let d2 () =
  ignore (Atomic.get flag); (* c *)
  !msg (* d *)
```

Data race

Does this have a data race?

- Consider that the functions d^* run in parallel

Does this have a data race?

- Consider that the functions d^* run in parallel

```
(* A counter is just a reference *)
```

```
let create_counter initial_value =  
  ref initial_value
```

```
let get_and_increment counter =  
  let v = !counter in  
  counter := v + 1;  
  v
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```


Does this have a data race?

- Consider that the functions d^* run in parallel

```
(* A counter is just a reference *)
```

```
let create_counter initial_value =  
  ref initial_value
```

```
let get_and_increment counter =  
  let v = !counter in  
  counter := v + 1;  
  v
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```

Data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
(* A counter is just a reference *)
```

```
let create_counter initial_value =  
  ref initial_value
```

```
let get_and_increment counter =  
  let v = !counter in  
  counter := v + 1;  
  v
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```

```
type t = { counter : int ref; mutex : Mutex.t }
```

```
let create_counter () =  
  { counter = ref 0; mutex = Mutex.create () }
```

```
let get_and_increment t =  
  Mutex.lock t.mutex;  
  let old_value = !(t.counter) in  
  t.counter := old_value + 1;  
  Mutex.unlock t.mutex;  
  old_value
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```

Data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
(* A counter is just a reference *)
```

```
let create_counter initial_value =  
  ref initial_value
```

```
let get_and_increment counter =  
  let v = !counter in  
  counter := v + 1;  
  v
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```

Data race

```
type t = { counter : int ref; mutex : Mutex.t }
```

```
let create_counter () =  
  { counter = ref 0; mutex = Mutex.create () }
```

```
let get_and_increment t =  
  Mutex.lock t.mutex;  
  let old_value = !(t.counter) in  
  t.counter := old_value + 1;  
  Mutex.unlock t.mutex;  
  old_value
```

```
let c = create_counter 0  
let d1 () = get_and_increment c  
let d2 () = get_and_increment c
```

No data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Data race

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Data race

- Where are the races?**

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Data race

- Where are the races?**
 - There are multiple races!

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Data race

- Where are the races?**
 - There are multiple races!
- Shockingly, this Peterson's lock does not provide mutual exclusion! (**DEMO**)

Does this have a data race?

- Consider that the functions d^* run in parallel

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

```
let d1 () = Peterson.lock ()
let d2 () = Peterson.lock ()
```

Data race

- Where are the races?**
 - There are multiple races!
- Shockingly, this Peterson's lock does not provide mutual exclusion! (***DEMO***)
- Data races lead to unintuitive behaviour
 - Can we detect them automatically?*

ThreadSanitizer

ThreadSanitizer

- Runtime data race detector (dynamic analysis, not static!)
 - Initially developed for C++ by Google, now supported in
 - C, C++ with GCC and clang, Go, Swift and OCaml

ThreadSanitizer

- Runtime data race detector (dynamic analysis, not static!)
 - Initially developed for C++ by Google, now supported in
 - C, C++ with GCC and clang, Go, Swift and OCaml
- Battle-tested, already found:
 - 1200+ races in Google's codebase
 - ~100 in the Go stdlib
 - 100+ in Chromium
 - LLVM, GCC, OpenSSL, WebRTC, Firefox

ThreadSanitizer

ThreadSanitizer

- Requires compiling your program specially
 - Works by instrumenting your program and detecting unsynchronised accesses

ThreadSanitizer

- Requires compiling your program specially
 - Works by instrumenting your program and detecting unsynchronised accesses
- Limitations
 - Only detects data races that appear during execution
 - “Testing only shows the presence of bugs, never the absence of them” — Edsger W. Dijkstra
 - Bookkeeping overheads
 - A finite number of memory accesses is remembered for every memory location
 - Misses out data races separated by many synchronised accesses in between
 - 2x to 7x slower execution time of programs

ThreadSanitizer Demo

```
opam switch create 5.4.0+tsan ocaml-option-tsan  
opam install dune
```


ThreadSanitizer Demo

```
opam switch create 5.4.0+tsan ocaml-option-tsan  
opam install dune
```

- Need to install a separate switch with TSAN enabled

ThreadSanitizer Demo

```
opam switch create 5.4.0+tsan ocaml-option-tsan
opam install dune
```

- Need to install a separate switch with TSAN enabled
- Let's detect data races on Peterson's Lock

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

ThreadSanitizer Demo

```
opam switch create 5.4.0+tsan ocaml-option-tsan
opam install dune
```

- Need to install a separate switch with TSAN enabled
- Let's detect data races on Peterson's Lock
- macOS detects the race, but no location info :-(
 - Location info reported on Linux

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one
     victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I'm interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter
       AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

DRF Peterson's Lock

```
module Peterson = struct
  let flag = [| Atomic.make false; Atomic.make false |]
  let victim = Atomic.make 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    Atomic.set flag.(i) true;
    Atomic.set victim i;
    while Atomic.get flag.(j) && Atomic.get victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    Atomic.set flag.(i) false

end
```

```
module Peterson : LOCK = struct
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true;
    victim := i;
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false

end
```

DRF Peterson's Lock

```
module Peterson = struct
  let flag = [| Atomic.make false; Atomic.make false |]
  let victim = Atomic.make 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    Atomic.set flag.(i) true;
    Atomic.set victim i;
    while Atomic.get flag.(j) && Atomic.get victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    Atomic.set flag.(i) false

end
```

```
module Peterson : LOCK = struct
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true;
    victim := i;
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false

end
```

No data race

DRF Peterson's Lock

```
module Peterson = struct
  let flag = [| Atomic.make false; Atomic.make false |]
  let victim = Atomic.make 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    Atomic.set flag.(i) true;
    Atomic.set victim i;
    while Atomic.get flag.(j) && Atomic.get victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    Atomic.set flag.(i) false

end
```

No data race

```
module Peterson : LOCK = struct
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true;
    victim := i;
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false

end
```

Data race

Programs from previous lectures

Programs from previous lectures

- Most of the concurrent programs from the previous lecture have data races
 - LockOne, LockTwo, Peterson's, Bakery Lock, Wait-free SPSC queue, ...

Programs from previous lectures

- Most of the concurrent programs from the previous lecture have data races
 - LockOne, LockTwo, Peterson's, Bakery Lock, Wait-free SPSC queue, ...
- All of them assume sequential consistency

Programs from previous lectures

- Most of the concurrent programs from the previous lecture have data races
 - LockOne, LockTwo, Peterson's, Bakery Lock, Wait-free SPSC queue, ...
- All of them assume sequential consistency
- Real-world implementations need to use **Atomic** module
 - You can fix them like we did the Peterson's lock

Summary

Summary

- ***Relaxed memory models*** are those which are weaker than SC
 - Behaviours must be explained with ***reorderings*** and not just ***interleaving***

Summary

- ***Relaxed memory models*** are those which are weaker than SC
 - Behaviours must be explained with ***reorderings*** and not just ***interleaving***
- ***Hardware Models*** are usually weaker than SC
 - ***Hardware optimisations*** such as store buffers, out-of-order executions, speculation, cache-coherence protocols, ...

Summary

- ***Relaxed memory models*** are those which are weaker than SC
 - Behaviours must be explained with ***reorderings*** and not just ***interleaving***
- ***Hardware Models*** are usually weaker than SC
 - ***Hardware optimisations*** such as store buffers, out-of-order executions, speculation, cache-coherence protocols, ...
- ***PL Models*** are usually weaker than SC
 - ***Compiler optimisations***

Summary

- ***Relaxed memory models*** are those which are weaker than SC
 - Behaviours must be explained with ***reorderings*** and not just ***interleaving***
- ***Hardware Models*** are usually weaker than SC
 - ***Hardware optimisations*** such as store buffers, out-of-order executions, speculation, cache-coherence protocols, ...
- ***PL Models*** are usually weaker than SC
 - ***Compiler optimisations***
- Data races are unsynchronised, conflicting memory accesses
 - PL models offer SC for data-race-free programs (DRF SC)

Summary

- ***Relaxed memory models*** are those which are weaker than SC
 - Behaviours must be explained with ***reorderings*** and not just ***interleaving***
- ***Hardware Models*** are usually weaker than SC
 - ***Hardware optimisations*** such as store buffers, out-of-order executions, speculation, cache-coherence protocols, ...
- ***PL Models*** are usually weaker than SC
 - ***Compiler optimisations***
- Data races are unsynchronised, conflicting memory accesses
 - PL models offer SC for data-race-free programs (DRF SC)
- Data races complicate program reasoning
 - OCaml ThreadSanitizer (TSAN) can detect data races dynamically



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/2.5/).

- **You are free:**
 - **to Share** – to copy, distribute and transmit the work
 - **to Remix** – to adapt the work
- **Under the following conditions:**
 - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.