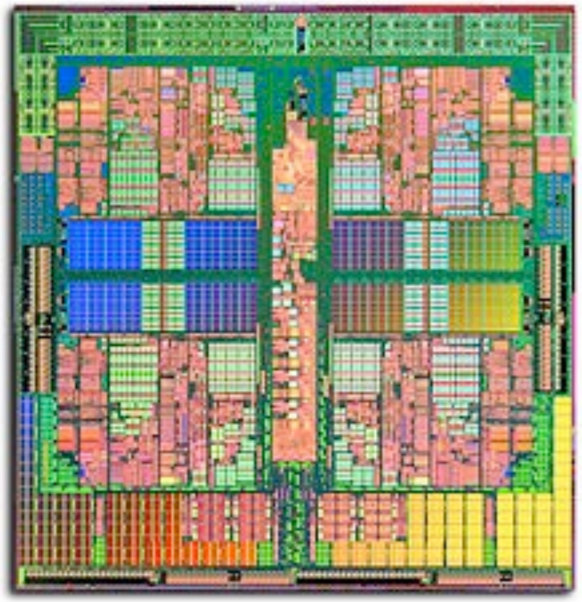


Semantics, languages and algorithms for multicore programming

Albert Cohen

Luc Maranget

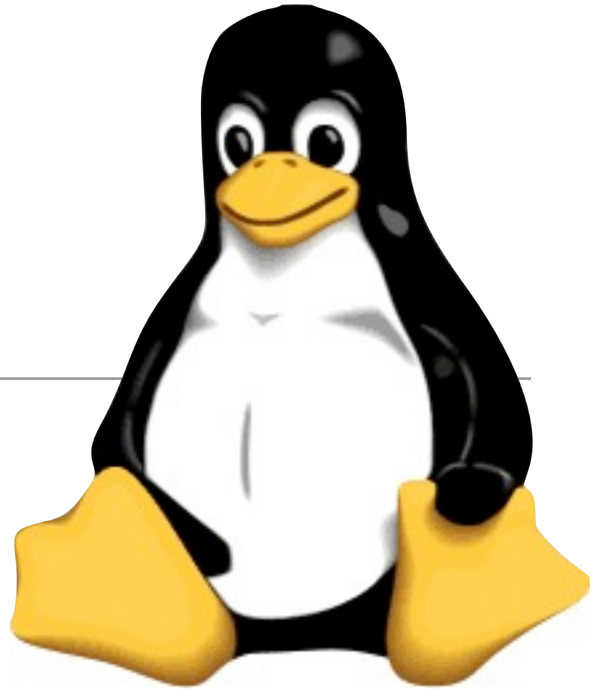
Francesco Zappa Nardelli

Vote: topics for my this lecture

1. Operational and axiomatic formalisation of x86-TSO (3)
2. The lwarx and stwcx Power instructions (2)
3. Fence optimisations for x86-TSO (6)
4. The Java memory model (3)
5. The C++11 memory model (10)
6. Static and dynamic techniques for data-race detection (5)
7. The Linux memory model (?!) (13)
8. Compiler correctness statements (compile non-determinism?) (5)

1. The Linux memory model (ahem, kinda)

The Linux memory model



Facts:

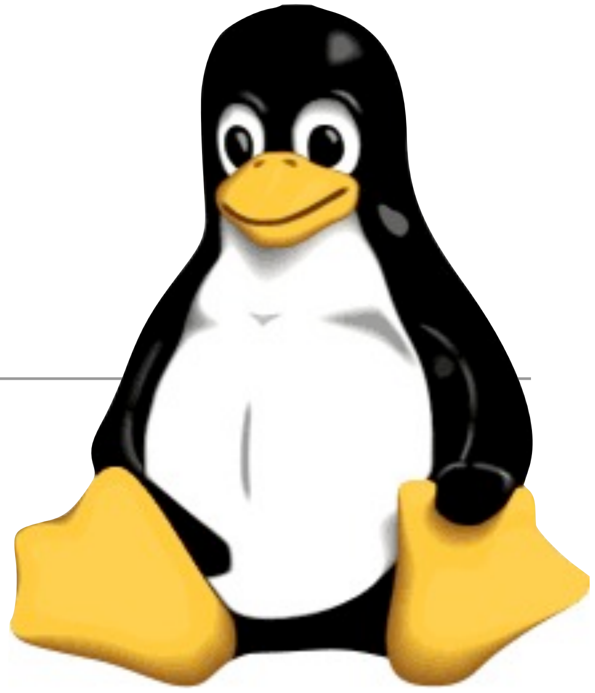
- abstraction layer over hardware and compilers

- relied upon by kernel developers to write "portable kernel code"

- documented by a text file:

`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

The Linux memory model



Facts:

- abstraction layer over hardware and compilers

- relied upon by kernel developers to write "portable kernel code"

- documented by a text file:

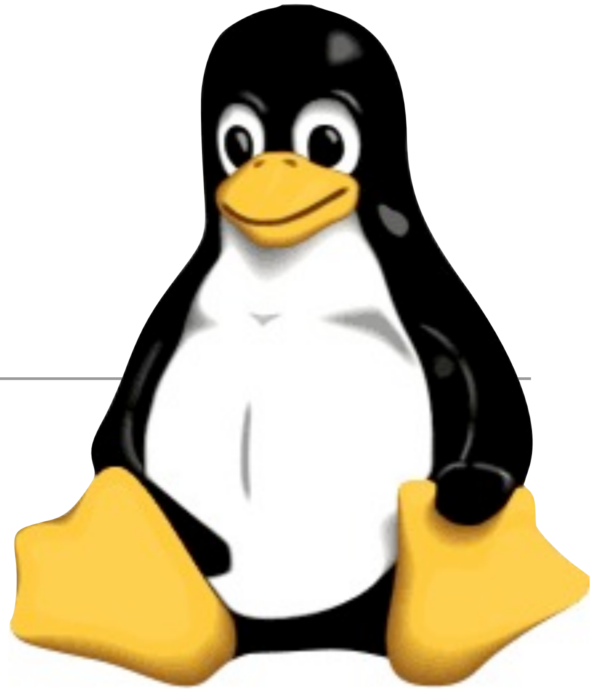
`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

More facts:

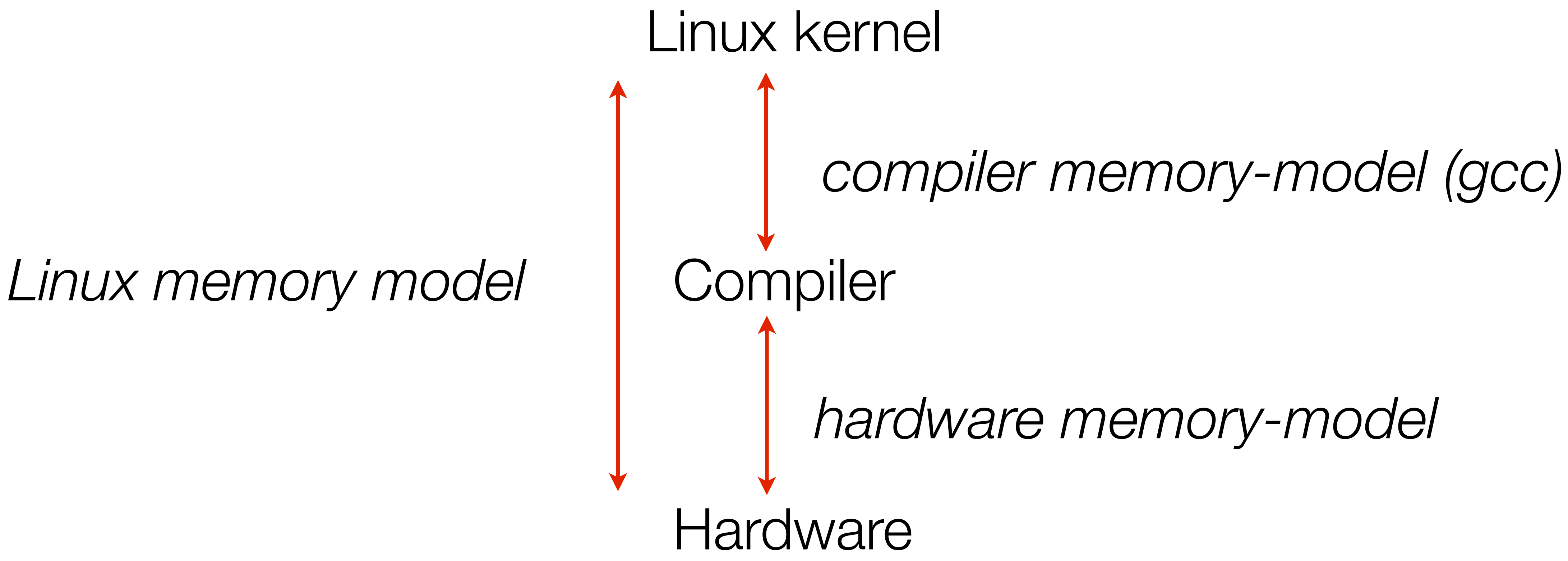
I attempted to understand the doc, and exchanged a few email with Paul Mc Kenney. However I don't understand much…

In the next hour, let's go over the documentation together and see if we can make sense of it...

The Linux memory model



Expected to account for all supported combinations of compiler and hardware memory model...



Linux kernel

compiler memory-model (gcc)

Linux memory model

Compiler

hardware memory-model

Hardware

alpha: Weak ordering. No dependency ordering. "Time does not go backwards" gives guarantees similar to Power/ARM A-cumulativity. Possibly B-cumulativity as well. I am not aware of formalization of this architecture's memory ordering other than Gharachorloo's PhD.

arm: You know at least as much as I do about this one.

avr32: Uniprocessor-only, kernel build failure for SMP.

blackfin: Uniprocessor-only to the best of my knowledge. There are rumored to be some experimental SMP systems that lack cache coherence, and are thus outside of the Linux kernel's remit. See for example: https://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:smp-like The system.h file flushes cache when a memory barrier is encountered, which is consistent with an attempt to run the Linux kernel on a non-cache-coherent system…

cris: Uniprocessor-only to the best of my knowledge. Though there appears to be recent addition of some SMP support. Its system.h file is consistent with full sequential consistency. Or extreme optimism on the part of the cris developers.

frv: Uniprocessor-only to the best of my knowledge.

h8300: Uniprocessor-only to the best of my knowledge. There is code in system.h that appears to be intended for SMP, but it looks to me like a (harmless) copy-paste error. Either that or SMP h8300 systems are sequentially consistent.

ia64: Total order of all release operations, which include the "mf" (memory fence) instruction. Memory fences cannot restore sequential consistency.

m32r: Uniprocessor-only to the best of my knowledge. However, there does appear to be some recent multiprocessor support. This is quite strange -- atomic instructions flush cache, but memory barriers are no-ops. Looks quite experimental.

m68k: Uniprocessor-only to the best of my knowledge.

microblaze: Uniprocessor-only to the best of my knowledge. At least one SMP attempt: http://microblazesmp.blogspot.com/ Its system.h file looks uniprocessor-only.

mips: Multiprocessor. Old SGI MIPS systems were sequentially consistent. Newer systems used for network infrastructure are rumored to have weak memory models similar to Power and ARM. And its system.h file is consistent with a weak memory model.

mn10300: Recent SMP support which I know little about. The system.h file looks uniprocessor only, and contains comments on Intel, so copy-pasted from x86.

parisc: TSO, similar to x86.

powerpc: You know at least as much about this as I do.

s390: TSO, but with self-snooping of store buffer prohibited.

score: Uniprocessor-only to the best of my knowledge.

sh: Recent SMP support which I know little about. Its system.h file is consistent with weak memory ordering.

sparc: TSO, similar to x86. There is documentation about weaker memory models (PSO and RMO), but in practice the hardware is TSO.

tile: Recent SMP CPU which I know little about. Seems to be weakly ordered based on its system.h file.

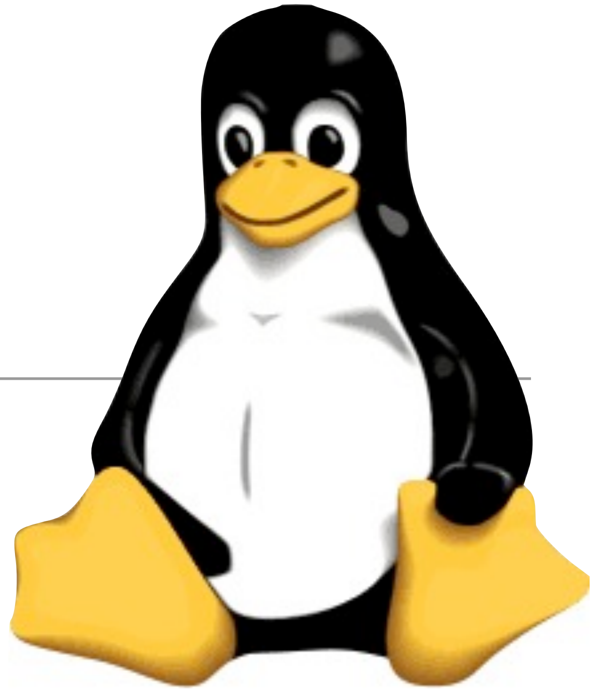
um: Looks like an x86 knockoff judging by the system.h file.

unicore32: Uniprocessor-only to the best of my knowledge.

x86: You know this one at least as well as do I.

xtensa: Uniprocessor-only -- kernel build failure otherwise.

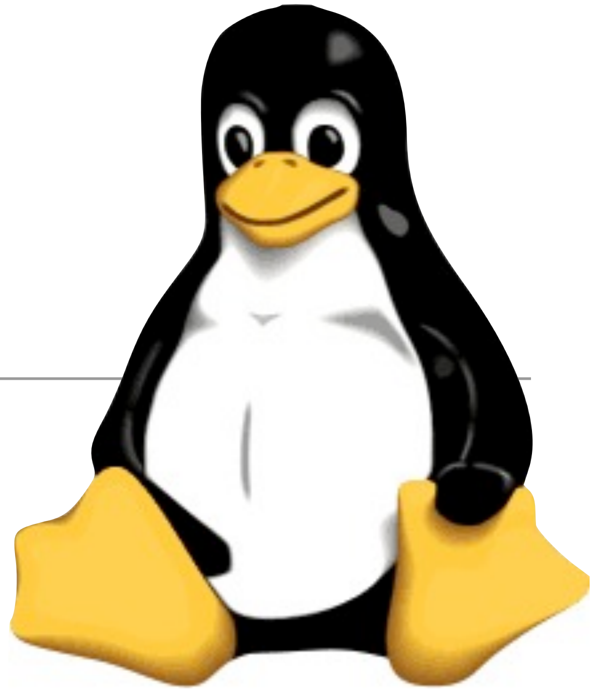
The Linux memory model



My intuition:

Annoying facts:

The Linux memory model



My intuition:

kinda of lowest common denominator between all hardware memory models of architectures Linux can be compiled to, taking into account also some common gcc optimisations, with some weirdnesses.

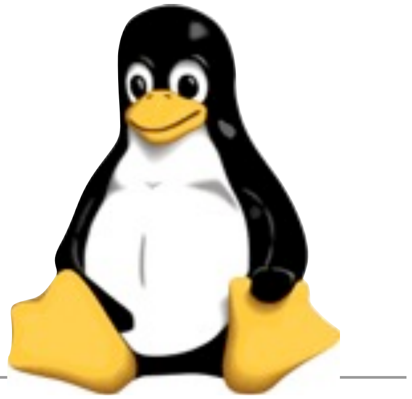
Annoying facts:

semantics of "read barriers" really weak, unclear how to formalise it

compilation of barriers on Itanium looks broken -- hardware might exhibit behaviours prohibited by the MM.

...let's read the doc...

The Linux memory model: macros



on x86:

```
#define mb()   asm volatile("mfence":::"memory")
#define rmb()  asm volatile("lfence":::"memory")
#define wmb()  asm volatile("sfence" ::: "memory")
```

as far as we know, lfence and sfence are noop in x86TSO

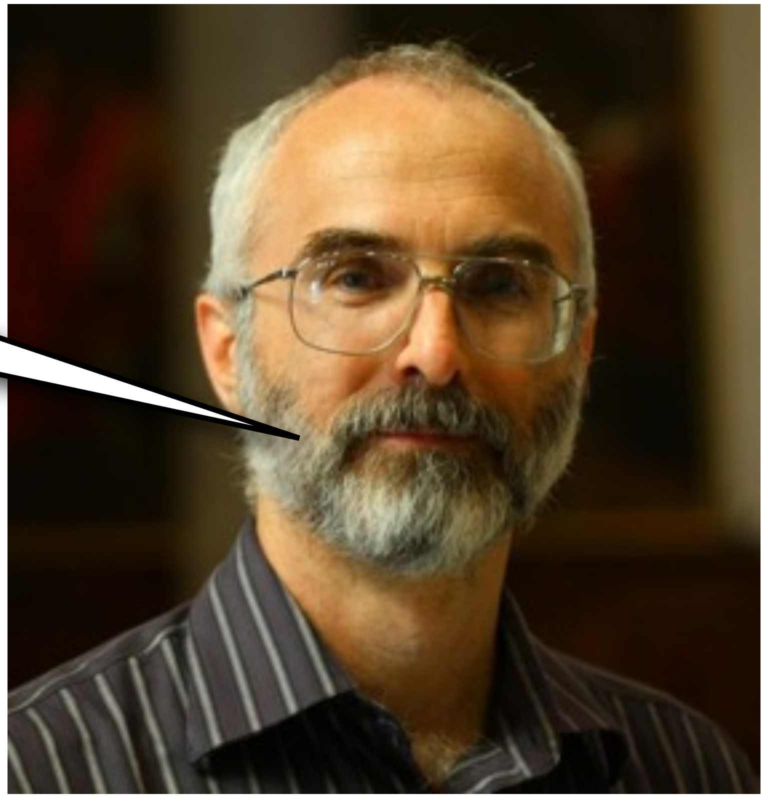
on Power:

```
#define mb()   __asm__ __volatile__ ("sync" : : : "memory")
#define rmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define wmb()  __asm__ __volatile__ ("sync" : : : "memory")
#define read_barrier_depends()  do { } while(0)
```

Internship proposal on the fly...

Sort out what the REAL Linux memory model is

Yes. Of course, if people come up with lots of situations where the more-complex programming model would help significantly, then it might be worth revisiting this.



Pros:

Challenging!

Can have a huge impact!

Collaboration with Paul Mc Kenney possible!

2. The C++11 memory model

a good example of an axiomatic memory model

The C++11 memory model

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

Welcome to the official home of

JTC1/SC22/WG21 - The C++ Standards Committee

2011-09-15: standards | projects | papers | mailings | internals | meetings | contacts

News 2011-09-11: The new C++ standard - C++11 - is published!

The syntactic divide

```
// for regular programmers:
atomic_int x = 0;
x.store(1);
y = x.load();

// for experts:
x.store(2, memory_order);
y = x.load(memory_order);
atomic_thread_fence(memory_order);
```

where *memory_order* is one of the following:

```
  mo_seq_cst  mo_release  mo_acquire
  mo_acq_rel  mo_consume  mo_relaxed
```

How may a program execute?

Two layer semantics:

1) an operational semantics processes programs, identifying memory actions, and constructs candidate executions (E_{opsem});

$$P \longrightarrow E_1, \ldots , E_n$$

2) an axiomatic memory model judges E_{opsem} paired with a memory ordering X_{witness}

$$E_i \longrightarrow X_{i1},\ldots,X_{im}$$

3) searches the consistent executions for races and uncostrained reads

is there an X_{ij} with a race?

Relations

An E_{opsem} part containing:

sb sequenced before, program order

asw additional synchronizes with, inter-thread ordering

An X_{witness} part containing:

rf relates a write to any reads that take its value

sc a total order over mo_seq_cst and mutex actions

mo modification order, per location total order of writes

From these, compute synchronise-with (*sw*) and happens-before (*hb*).

We ignore *consume* atomics, which enables us to live in a simplified model.

Full details in Batty et al., POPL 11.

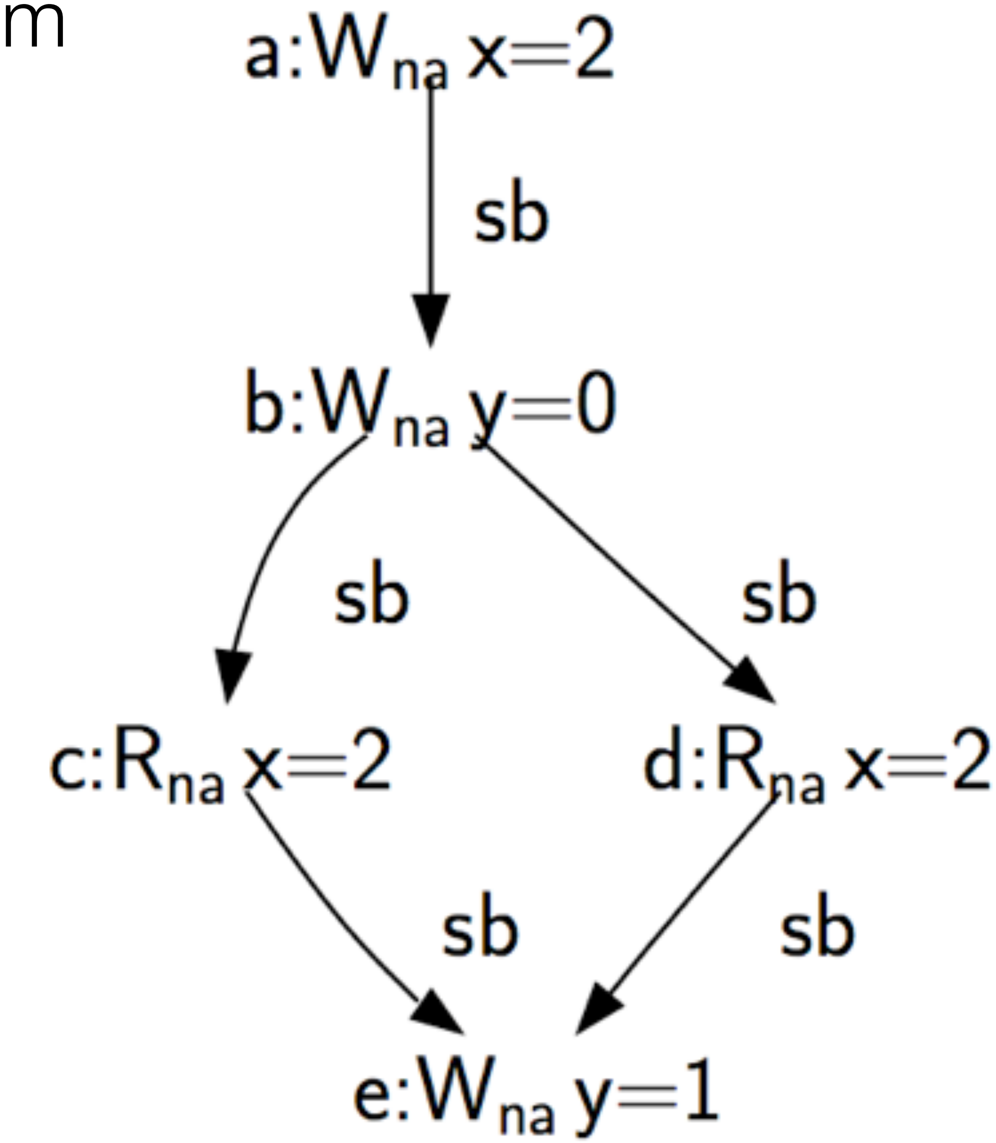
Formally

```
cpp_memory_model_opsem (p : program) =
let pre_executions =
  {($E_{opsem}$,$X_{witness}$).  opsem p $E_{opsem}$ ∧
    consistent execution ($E_{opsem}$, $X_{witness}$)}
in
if ∃X ∈ pre executions.
   (indeterminate reads X = {}) ∨
   (unsequenced races X = {}) ∨
   (data races X = {})
then NONE
else SOME pre_executions
```

A single-threaded example

1. sequenced before (sb) - given by opsem

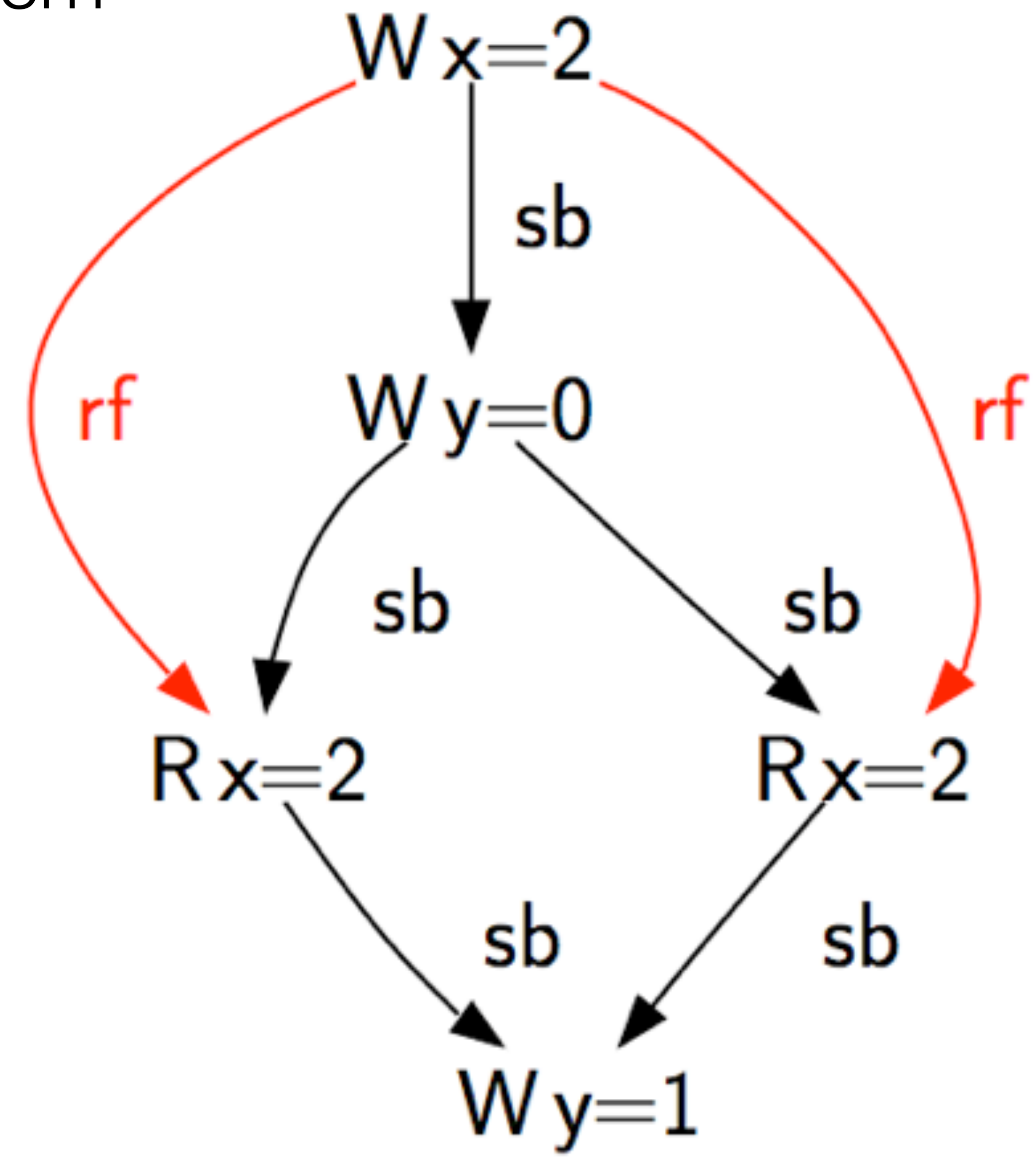
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded example

1. sequenced before (sb) - given by opsem
2. read-from (rf) - part of the witness

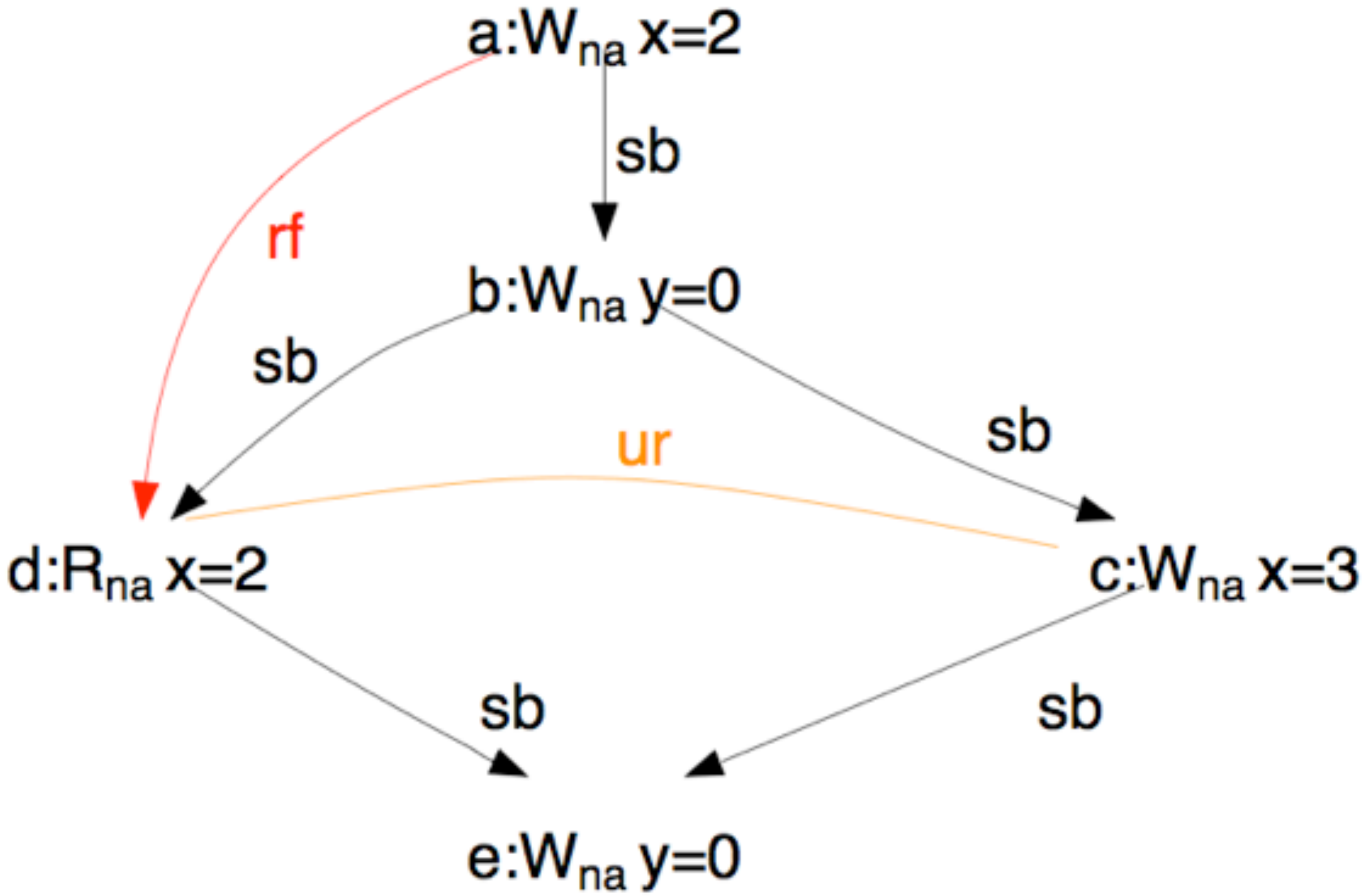
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0;
}
```



A single-threaded ex. with undefined behaviour

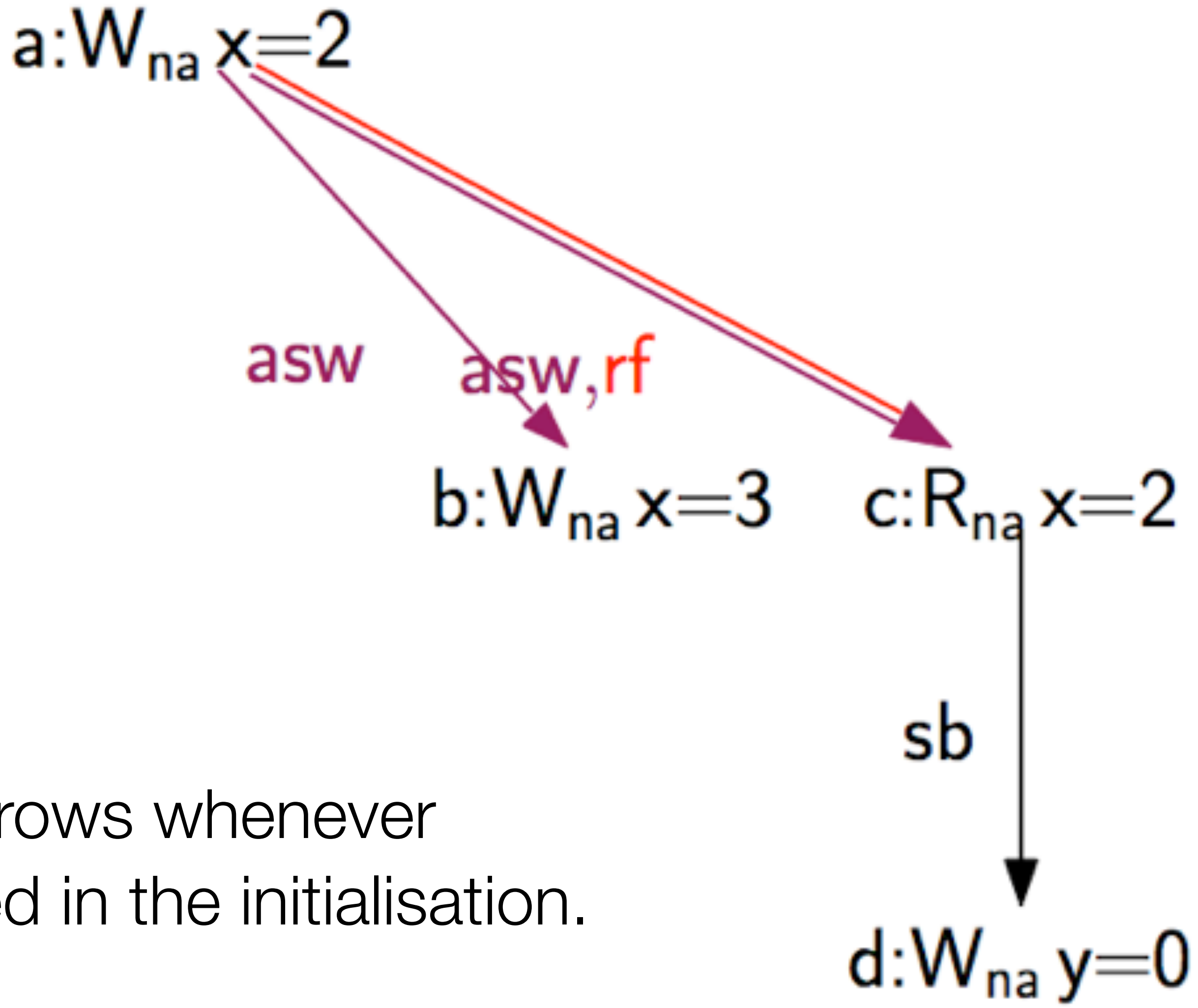
An unsequenced race.

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==(x=3));
  return 0;
}
```



A simple concurrent program

```
int y, x = 2;
x = 3;                  | y = (x==3);
```

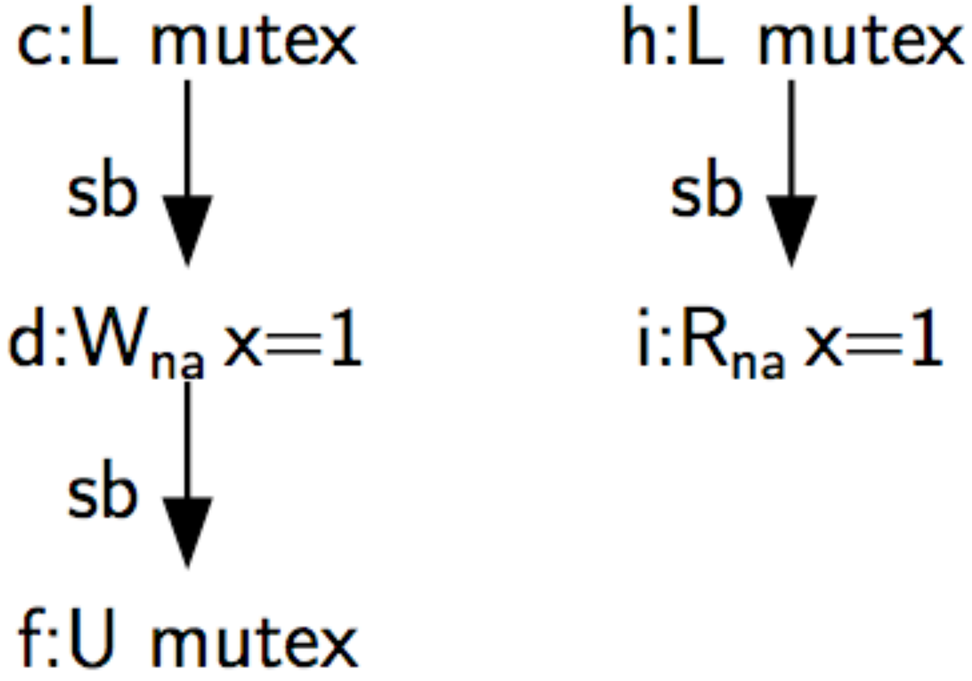


We will omit `asw` arrows whenever we are not interested in the initialisation.

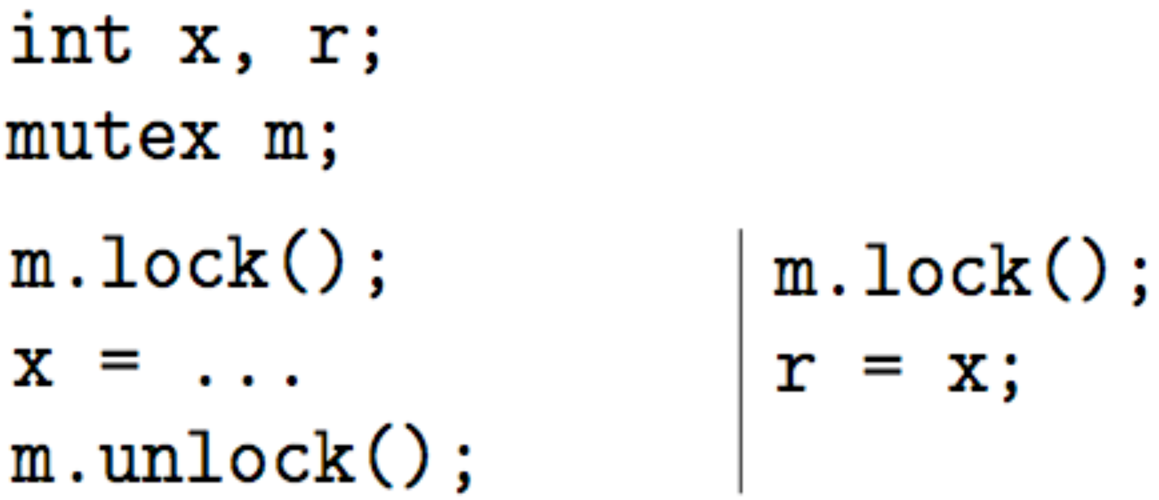
Locks and unlocks

```
int x, r;
mutex m;
m.lock();        | m.lock();
x = ...          | r = x;
m.unlock();      |
```

1. the operational semantics defines the sb arrows

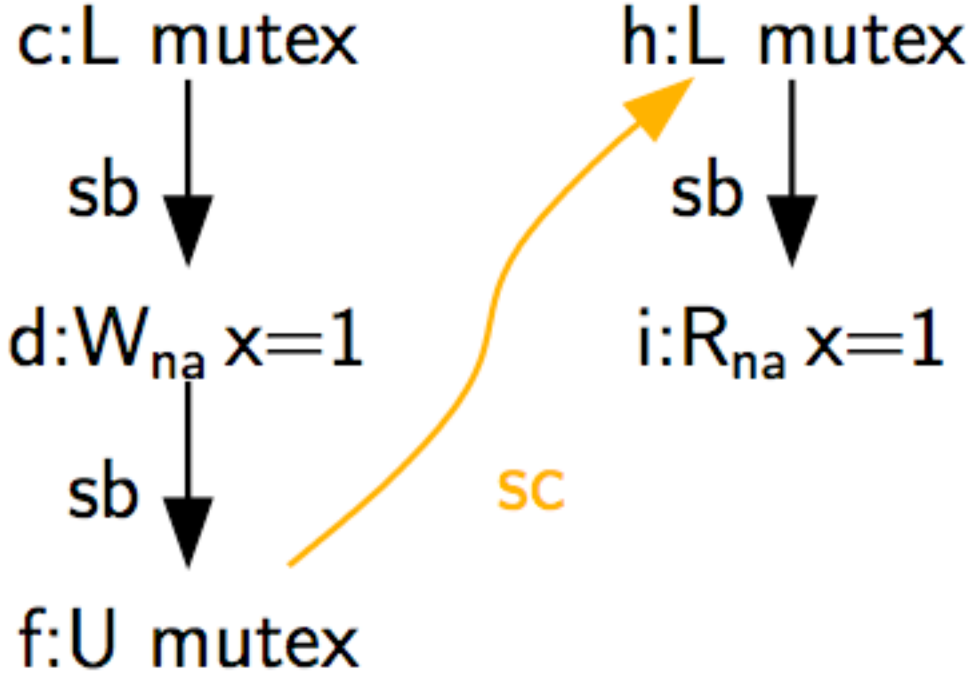


Locks and unlocks

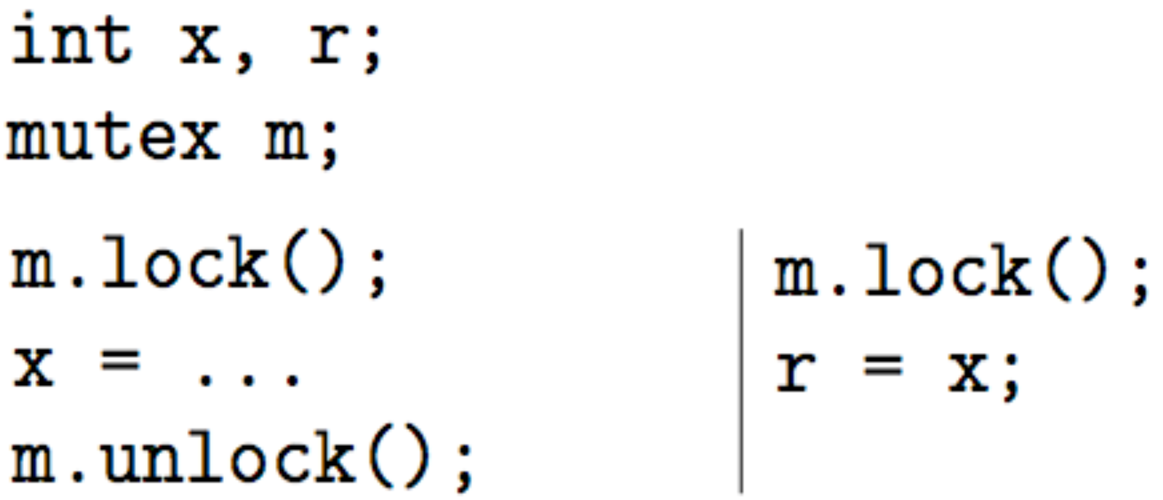


```
int x, r;
mutex m;
m.lock();        | m.lock();
x = ...          | r = x;
m.unlock();      |
```

1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)

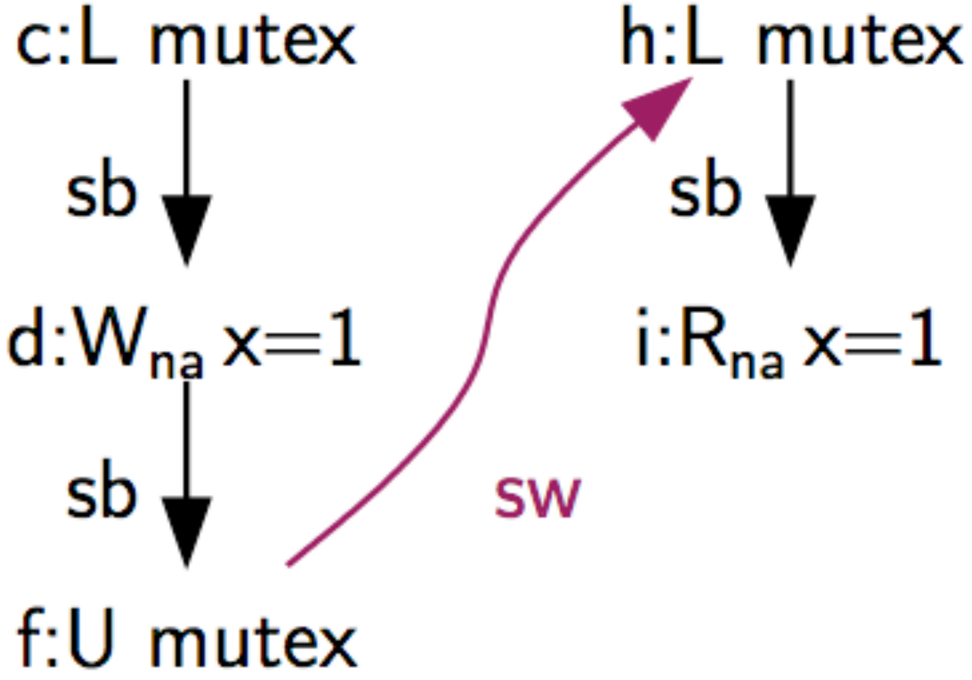


Locks and unlocks



```
int x, r;
mutex m;
m.lock();        | m.lock();
x = ...          | r = x;
m.unlock();      |
```

1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)
3. the sc order is included in the syncronised-with relation

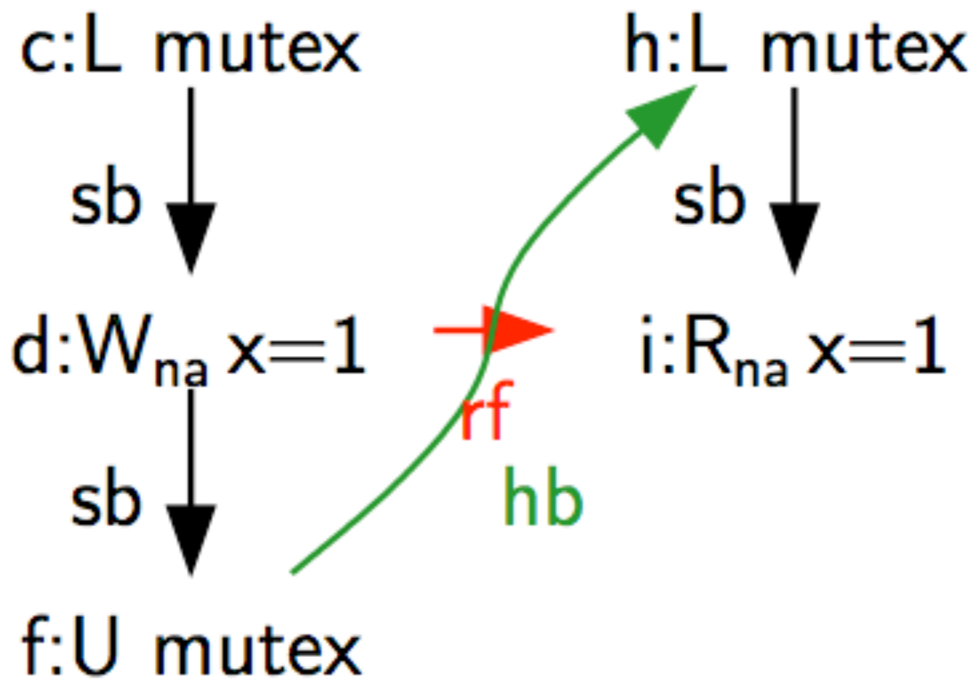


Locks and unlocks

$$\xrightarrow{\textit{simple-happens-before}} = (\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}})^+$$

```
int x, r;
mutex m;

m.lock();           | m.lock();
x = ...             | r = x;
m.unlock();         |
```

1. the operational semantics defines the sb arrows
2. guess an sc order on Unlock/Lock actions (part of the witness)
3. the sc order is included in the syncronised-with relation
4. which in turn defines the happens-before relation...



Happens before

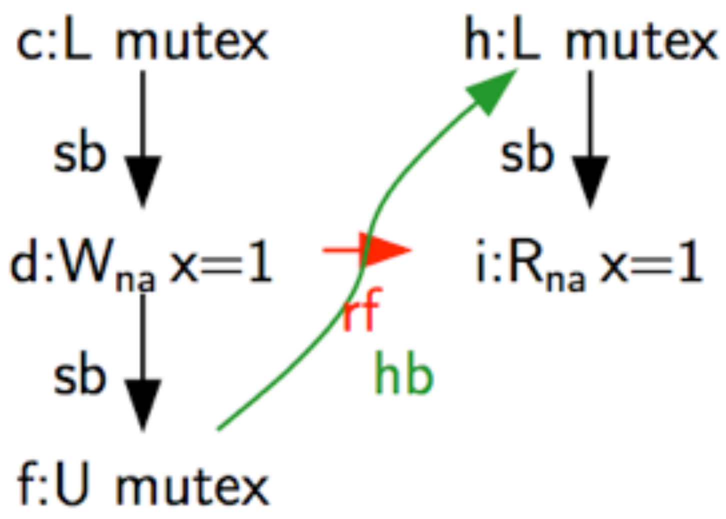
The *happens before* relation is key to the model:

1. non-atomic loads read the most recent write in happens before.
   (This is unique in DRF programs)

2. the story is more complex for atomics, as we shall see.

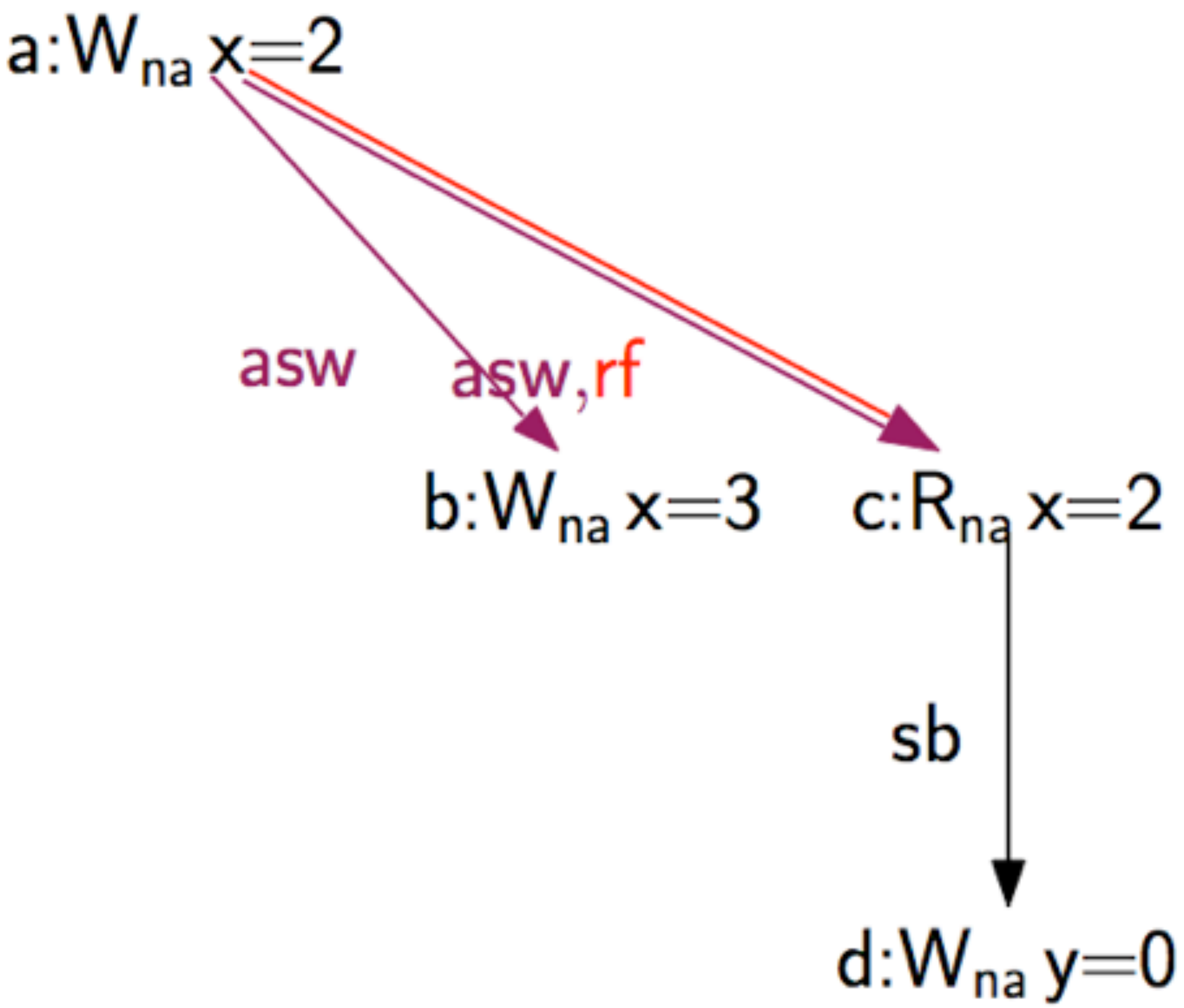
3. data races are defined as an absence of happens before
   between conflicting actions.

$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^{+}$$



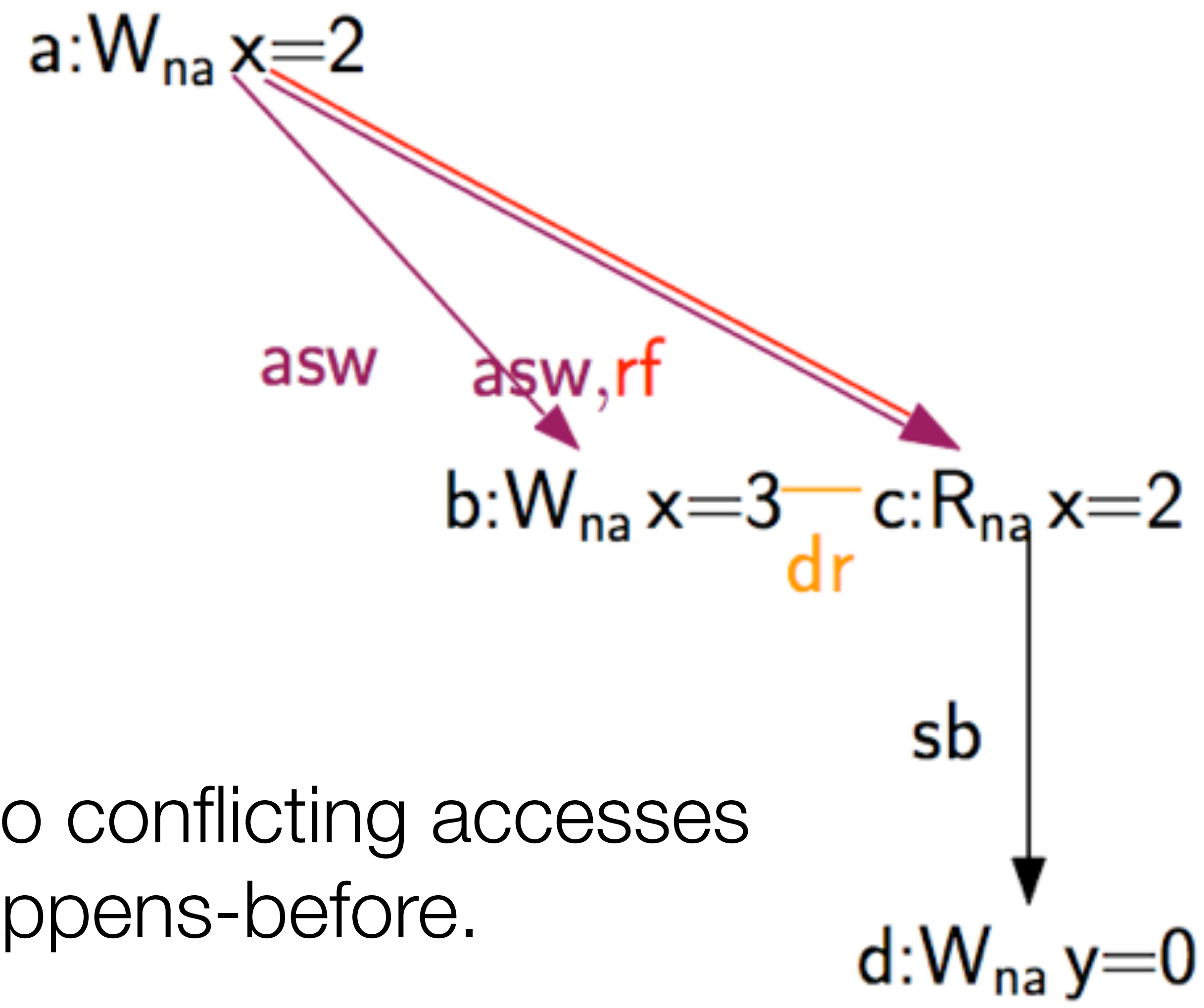
A data race

```
int y, x = 2;
x = 3;                  | y = (x==3);
```



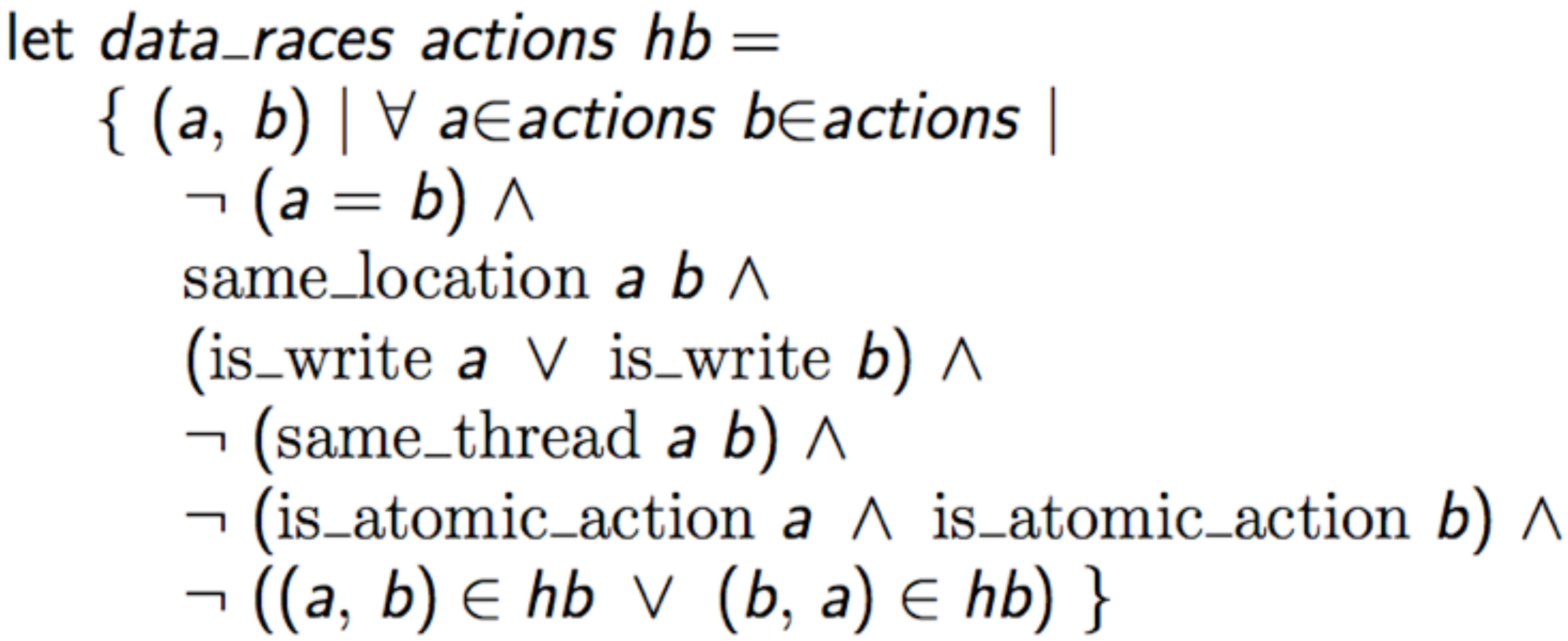
A data race

```
int y, x = 2;
x = 3;                | y = (x==3);
```



Here we have two conflicting accesses not related by happens-before.

Data race definition

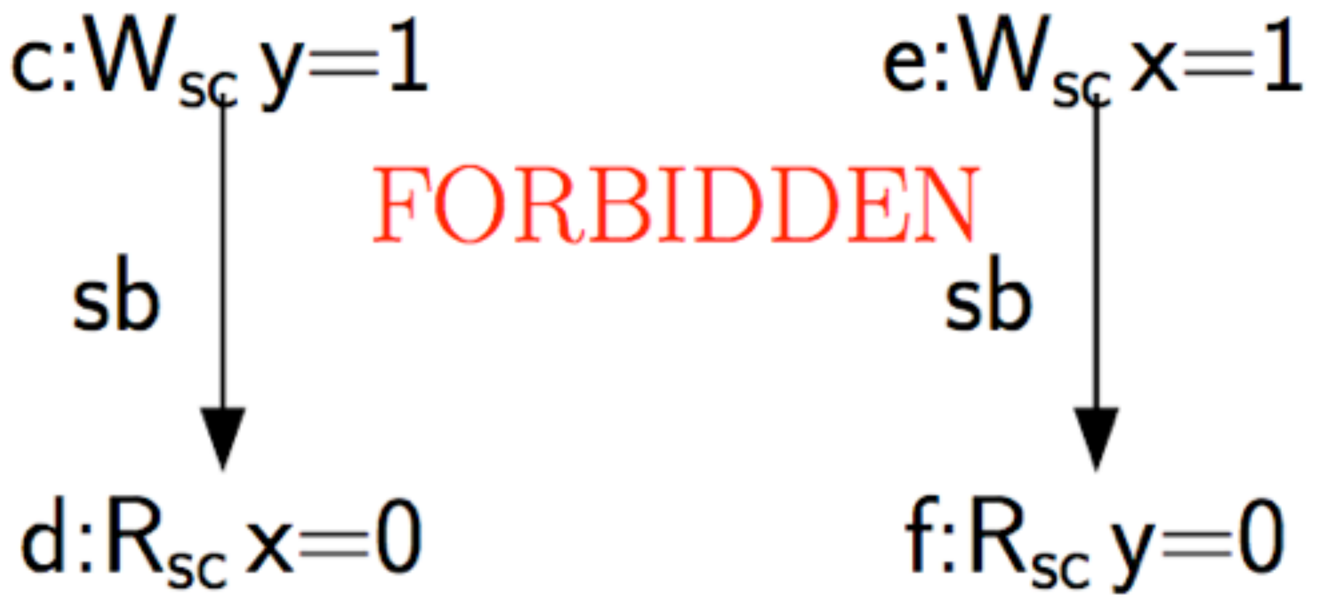


```
let $data\_races\ actions\ hb =$
    $\{\ (a,\ b) \mid \forall\ a{\in}actions\ b{\in}actions \mid$
        $\neg\ (a = b) \wedge$
        $\mathrm{same\_location}\ a\ b \wedge$
        $(\mathrm{is\_write}\ a\ \vee\ \mathrm{is\_write}\ b) \wedge$
        $\neg\ (\mathrm{same\_thread}\ a\ b) \wedge$
        $\neg\ (\mathrm{is\_atomic\_action}\ a\ \wedge\ \mathrm{is\_atomic\_action}\ b) \wedge$
        $\neg\ ((a,\ b) \in hb\ \vee\ (b,\ a) \in hb)\ \}$
```

Programs with a data race have undefined behaviour (DRF model).

Simple concurrency: Dekker's example and SC

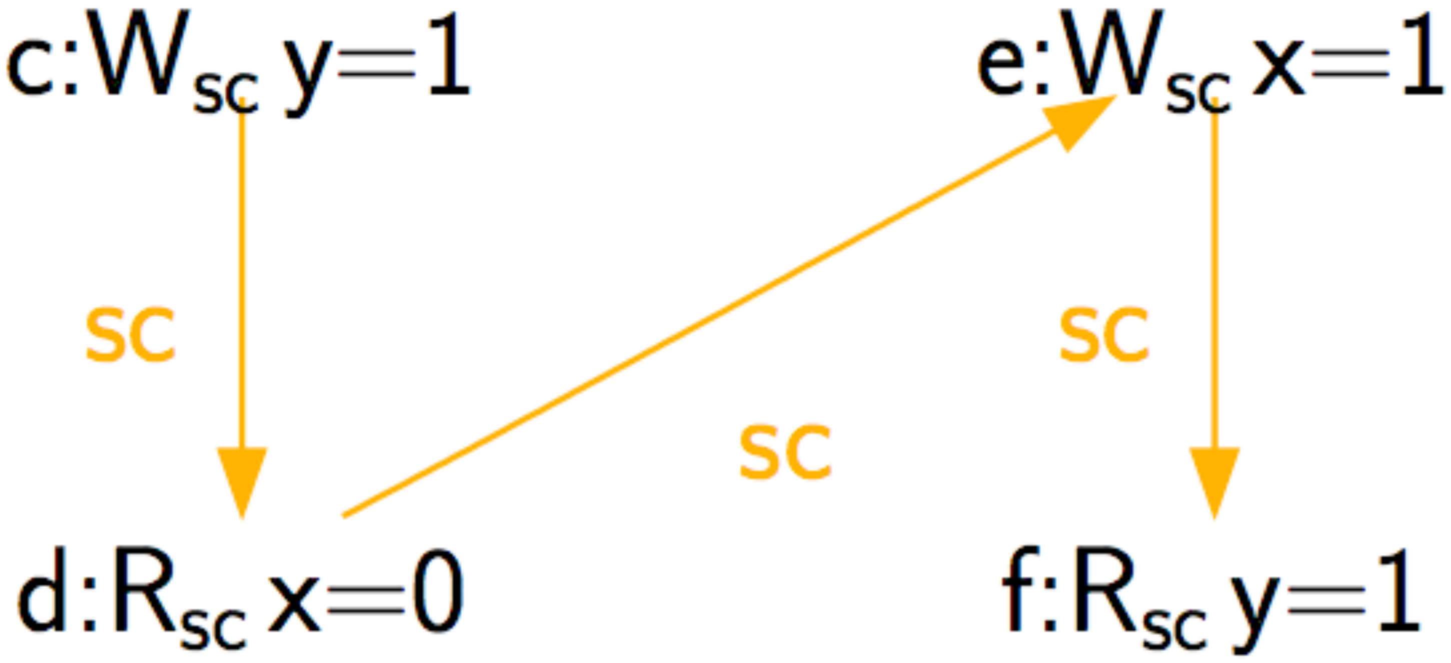
```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst); | y.store(1, seq_cst);
y.load(seq_cst);     | x.load(seq_cst);
```



Why is this behaviour forbidden?

Simple concurrency, Dekker's example and SC

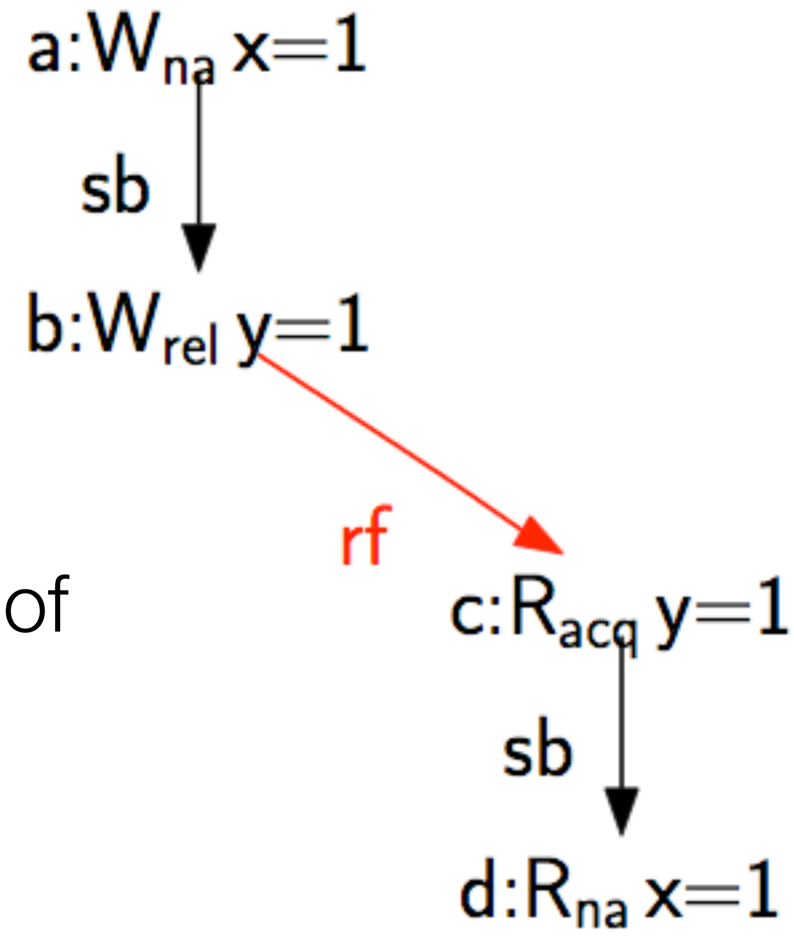
```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  | y.store(1, seq_cst);
y.load(seq_cst);      | x.load(seq_cst);
```



The `sc` relation must define a total order over unlocks/locks and `seq_cst` accesses... `sc` is included in `hb`, an `rf` must respect `hb`.

Expert concurrency: the release-acquire idiom

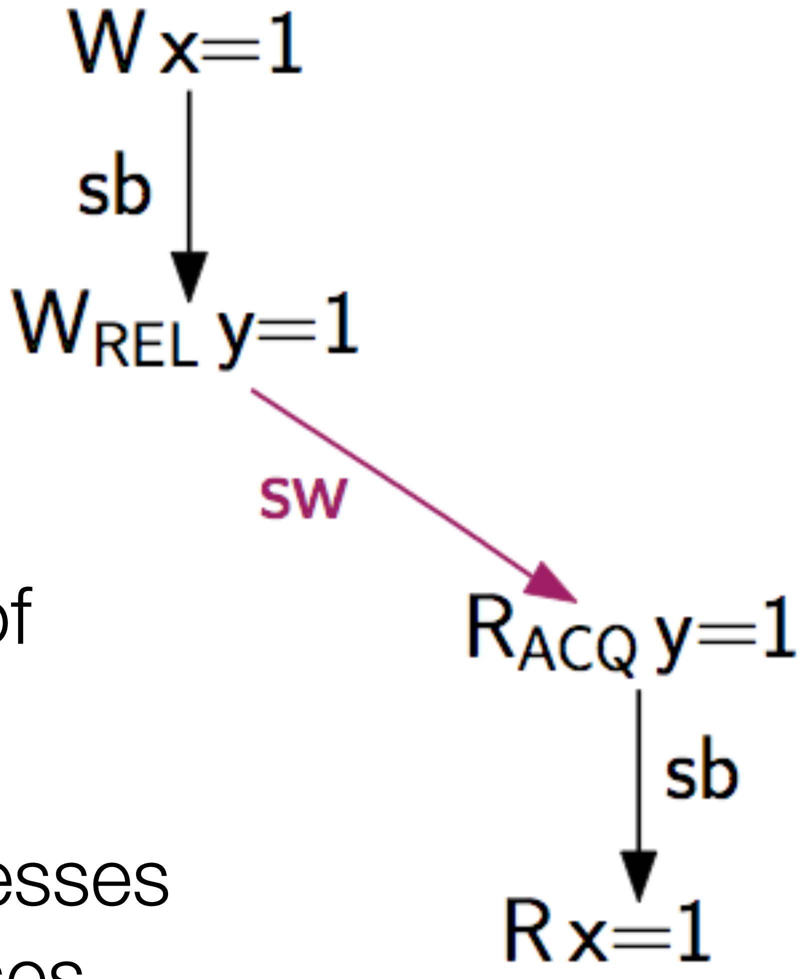
```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```



Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

Expert concurrency: the release-acquire idiom

```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```

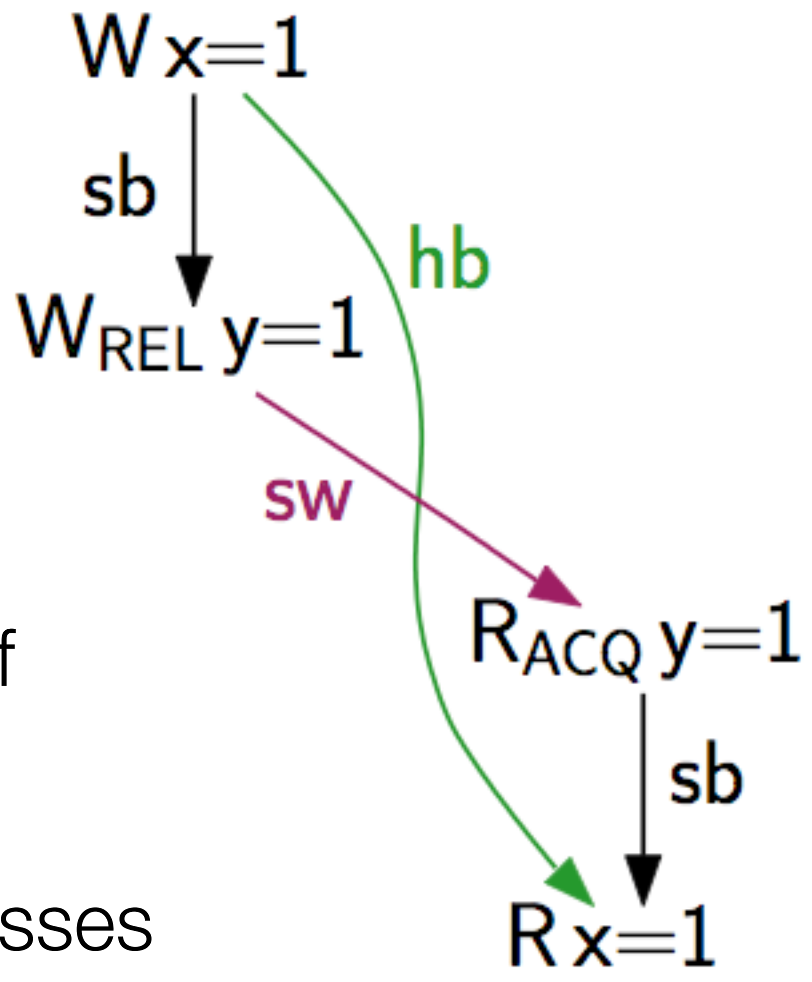


Here we have an `rf` arrow beetwen a pair of release/acquire accesses.

The `rf` arrow beetwen release/acquire accesses induces an `sw` arrow between those accesses.

Expert concurrency: the release-acquire idiom

```
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```



Here we have an **rf** arrow beetwen a pair of release/acquire accesses.

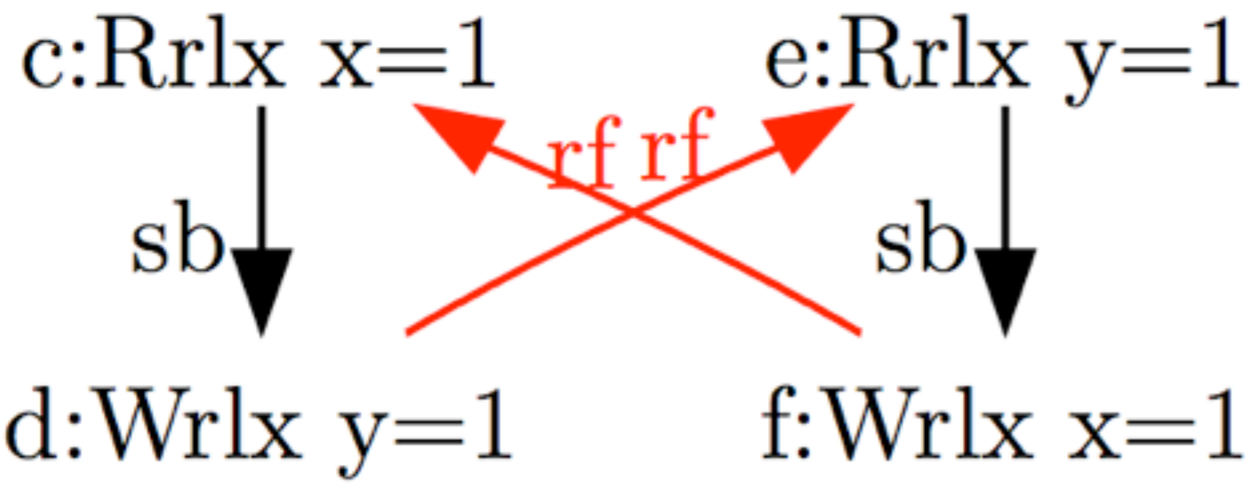
The **rf** arrow beetwen release/acquire accesses induces an **sw** arrow between those accesses.

And in turn defines an **hb** constraint.

$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^{+}$$

Relaxed writes

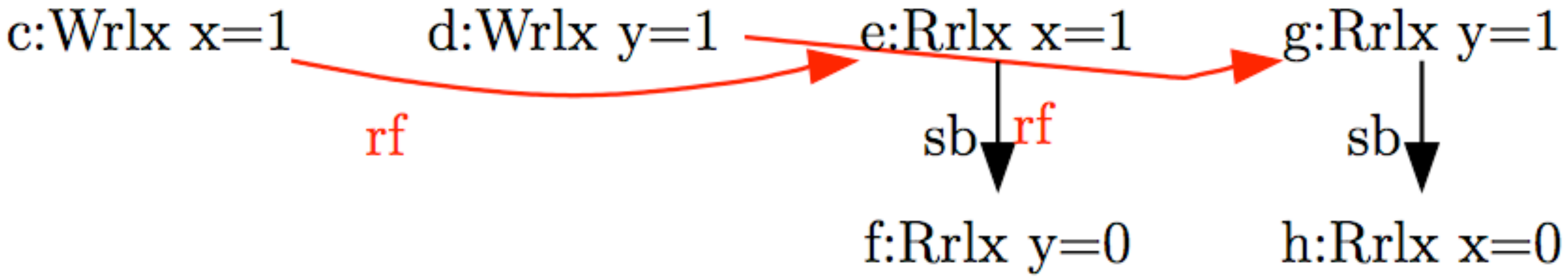
```
x.load(relaxed);        | y.load(relaxed);
y.store(1, relaxed);    | x.store(1, relaxed);
```



No data-races, no synchronisation cost, but weakly ordered.

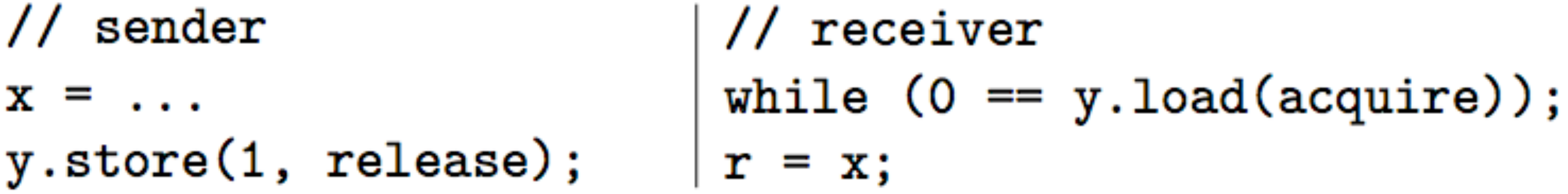
Relaxed writes, ctd.

```
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed); | y.store(2, relaxed); | x.load(relaxed); | y.load(relaxed);
                     |                      | y.load(relaxed); | x.load(relaxed);
```

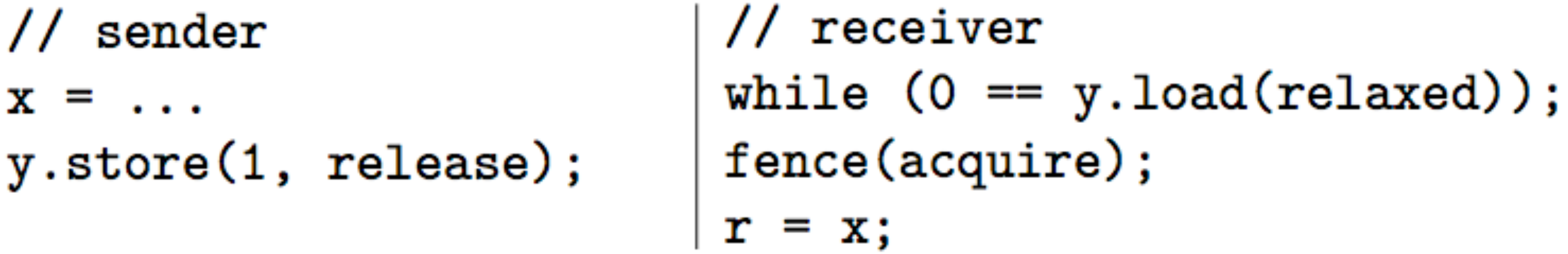


Again, no data-races, no synchronisation cost, but weakly ordered (IRIW).

Expert concurrency: fences avoid excess sync.

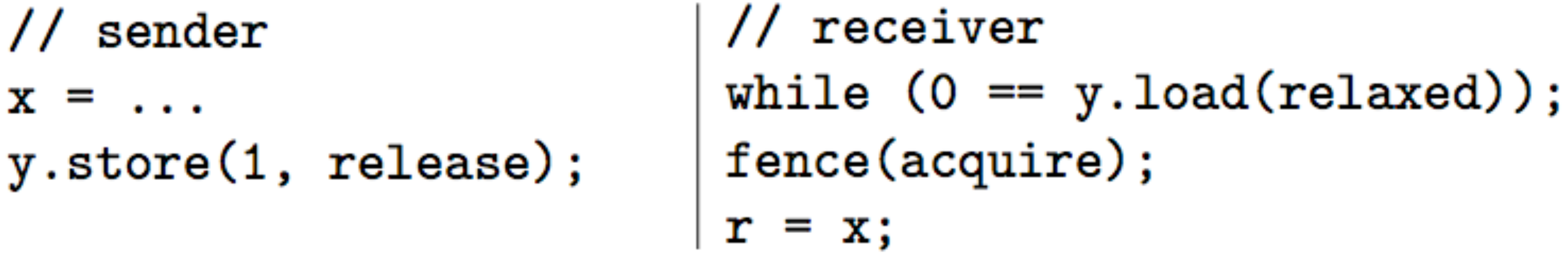


```
// sender                    | // receiver
x = ...                      | while (0 == y.load(acquire));
y.store(1, release);         | r = x;
```



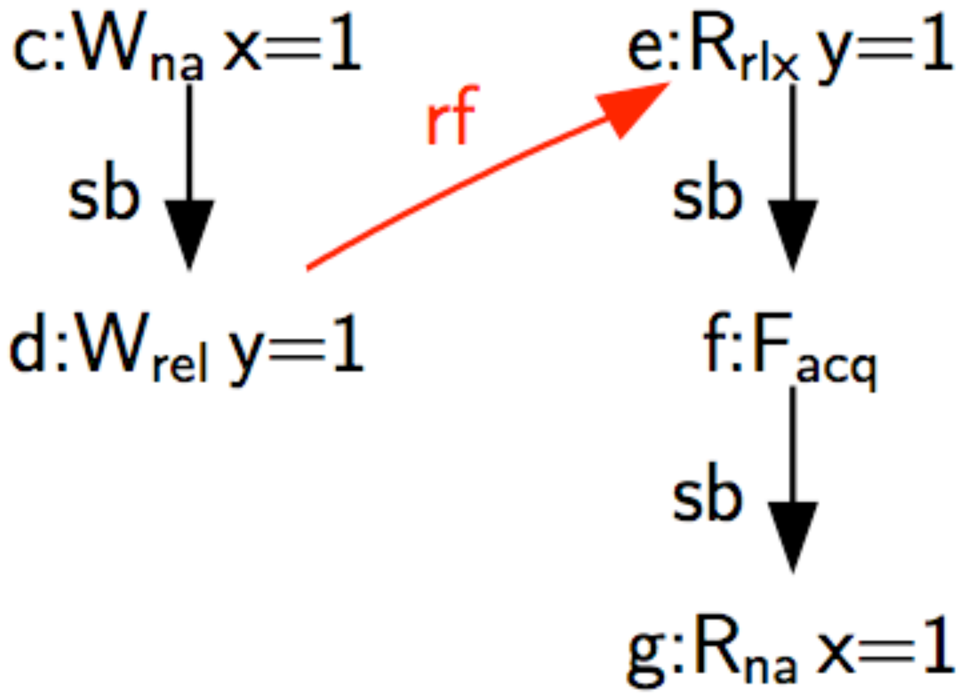
```
// sender                    | // receiver
x = ...                      | while (0 == y.load(relaxed));
y.store(1, release);         | fence(acquire);
                             | r = x;
```

Expert concurrency: fences avoid excess sync.

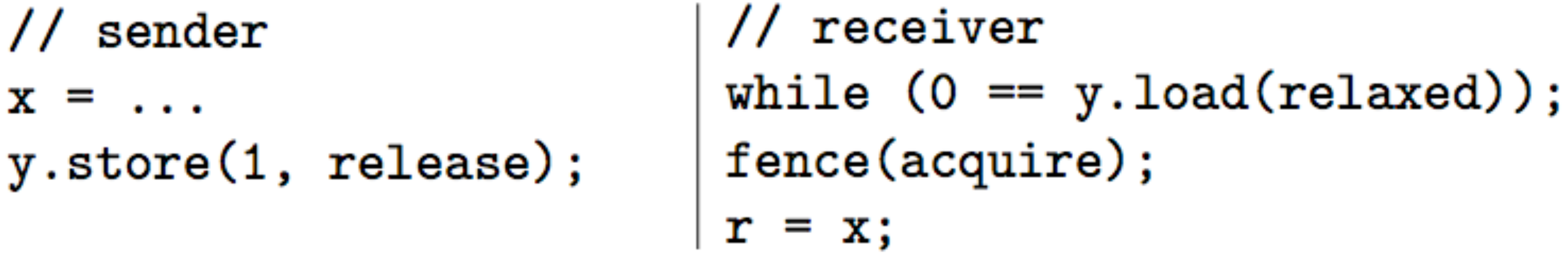


```
// sender                    // receiver
x = ...                      while (0 == y.load(relaxed));
y.store(1, release);         fence(acquire);
                             r = x;
```

Here we have an `rf` arrow beetwen a release write and a relaxed write.

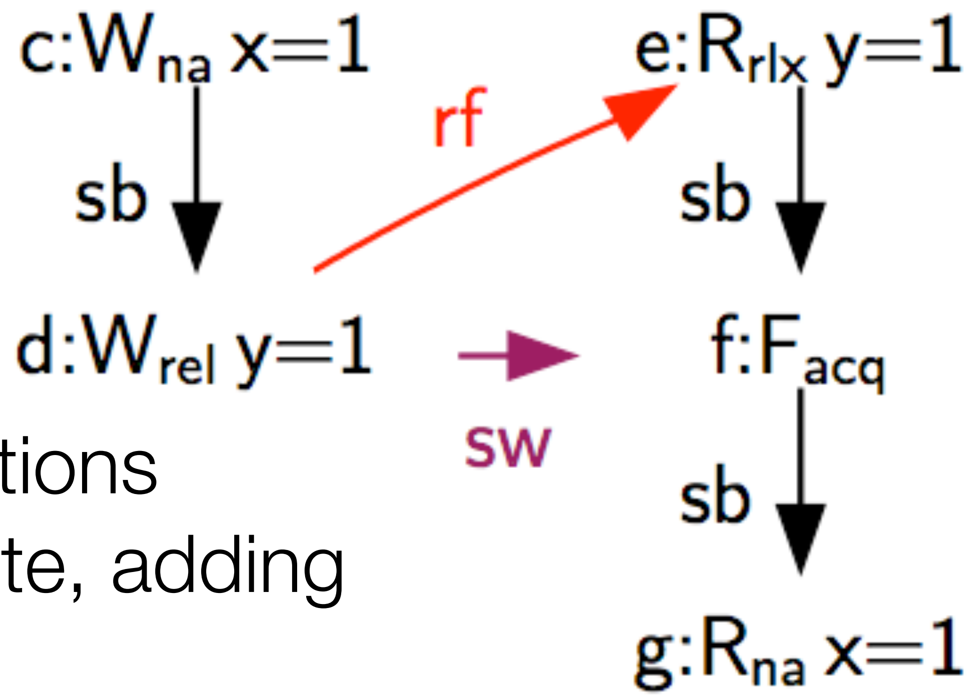


Expert concurrency: fences avoid excess sync.



```
// sender                    // receiver
x = ...                      while (0 == y.load(relaxed));
y.store(1, release);         fence(acquire);
                             r = x;
```

Here we have an `rf` arrow beetwen a release write and a relaxed write.



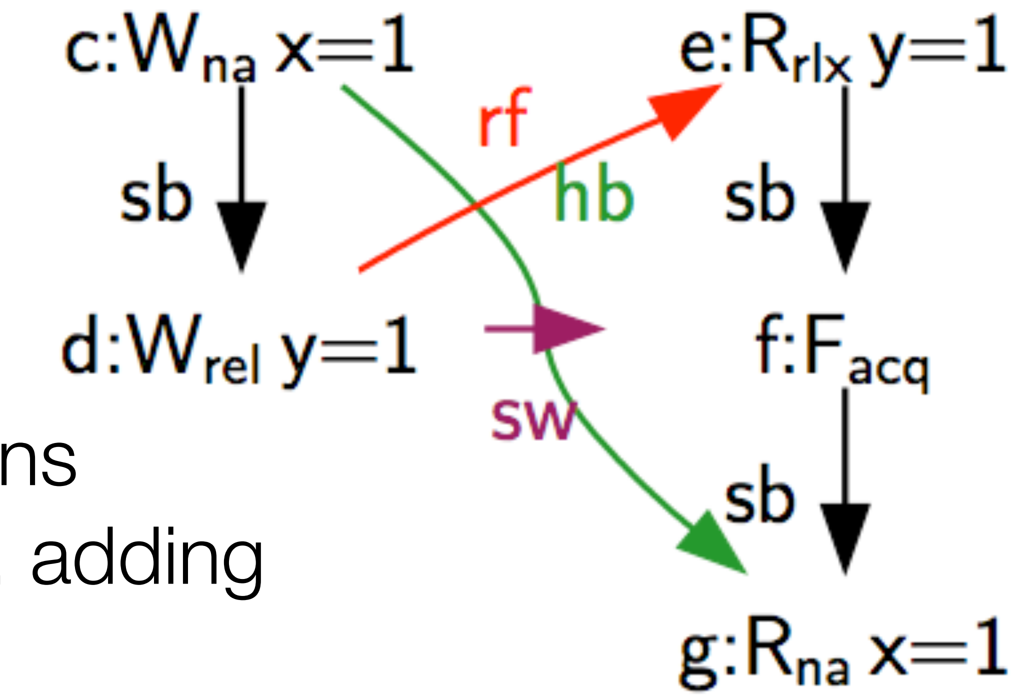
The acquire fence follows the `sb`/`rf` relations looking for the corresponding release write, adding a `sw` arrow.

Expert concurrency: fences avoid excess sync.

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Here we have an rf arrow beetwen a release write and a relaxed write.

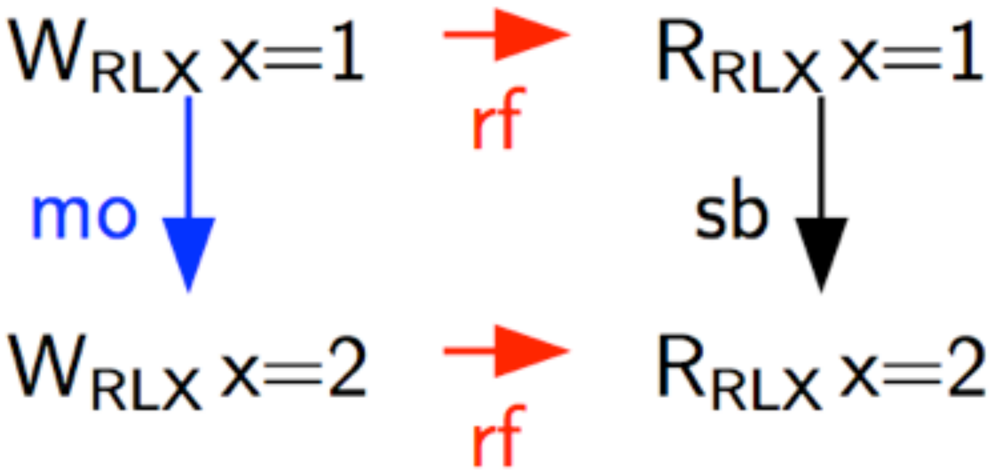


The acquire fence follows the sb/rf relations looking for the corresponding release write, adding a sw arrow.

Happens-before follows as usual...

Modification order

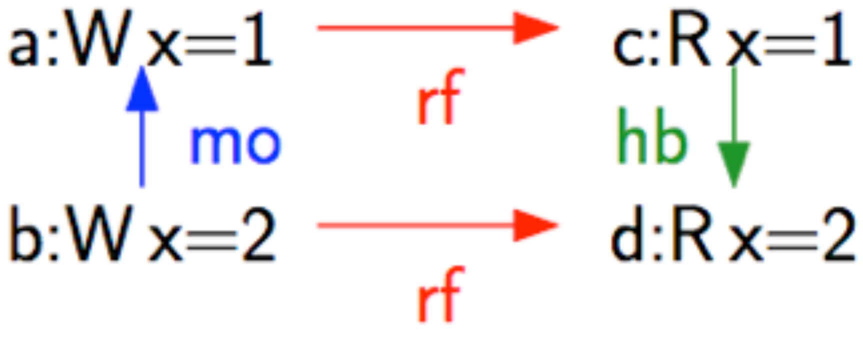
```
   atomic_int x = 0;
x.store(1, relaxed); || x.load(relaxed);
x.store(2, relaxed); || x.load(relaxed);
```



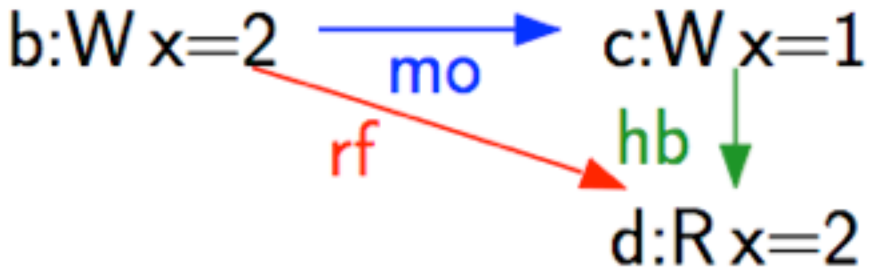
Modification order is a total order over atomic writes of any memory order.

Coherence and atomic reads

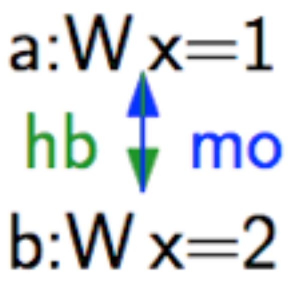
All forbidden:



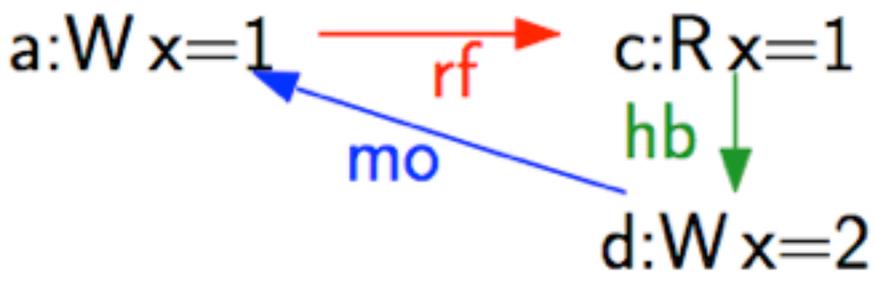
CoRR



CoWR



CoWW



CoRW

Idea: atomics cannot read from later writes in happens-before.

Coherence and atomic reads

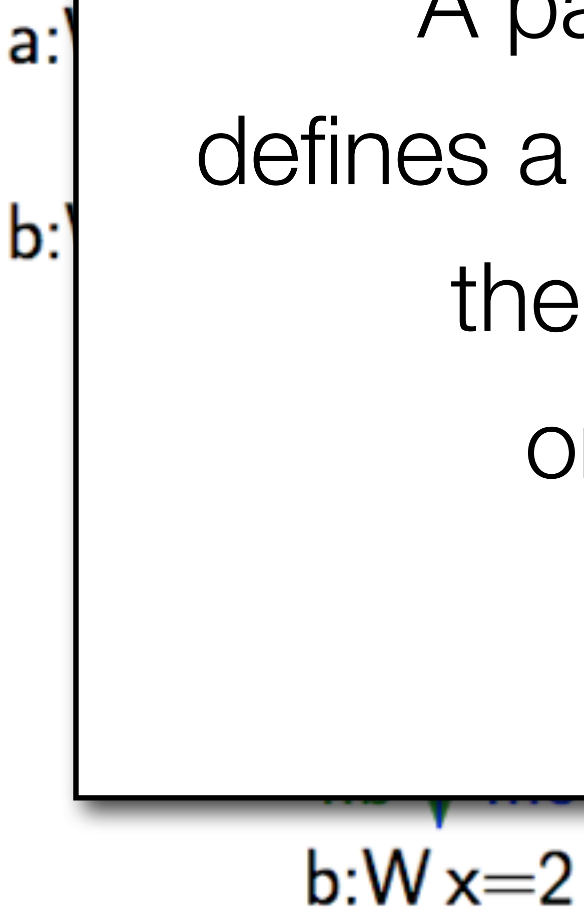
All forb

A pair $E_{\text{opsem}}, X_{\text{witness}}$ (a pre-execution)

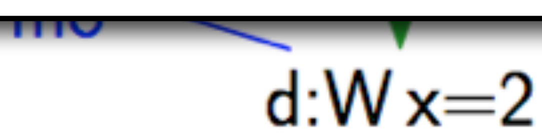
defines a *consistent execution* when it satisfies

the constraints we have sketched

on `hb`/`rf`/`mo` and is race-free.



CoWW



CoRW

Idea: atomics cannot read from later writes in happens-before.

The full model

```
$a \xrightarrow{r} b = (a, b) \in r$
```

```
$a\ r\ b = (a, b) \in r$
```

```
$a \not\xrightarrow{r} b = (a, b) \notin r$
```

```
$\xrightarrow{r} = r$
```

```
$a \xrightarrow{r} b \xrightarrow{s} c = a \xrightarrow{r} b \land b \xrightarrow{s} c$
```

```
relation_over s rel = domain rel $\subseteq$ s $\land$ range rel $\subseteq$ s
```

```
$\xrightarrow{rel}|_s = rel \cap (s \times s)$
```

```
$rel|_s = rel \cap (s \times s)$
```

```
$\xrightarrow{rel}|_s = rel \cap (s \times s)$
```

```
$rel|_s = rel \cap (s \times s)$
```

```
strict_preorder ord = irreflexive ord $\land$ trans ord
```

```
total_over s ord =
  relation_over s ord $\land$
  $(\forall x \in s.\ \forall y \in s.\ x \xrightarrow{ord} y \lor y \xrightarrow{ord} x \lor (x = y))$
```

```
strict_total_order_over s ord =
  strict_preorder ord $\land$ total_over s ord
```

```
$x \overset{ord}{\longmapsto}_{pred}\ y =$
  $pred\ x \land x \xrightarrow{ord} y \land \neg(\exists z.\ pred\ z \land x \xrightarrow{ord} z \xrightarrow{ord} y)$
```

```
$x \overset{ord}{\longmapsto} y =$
  $x \xrightarrow{ord} y \land \neg(\exists z.\ x \xrightarrow{ord} z \xrightarrow{ord} y)$
```

```
well_founded r = wf r
```

```
type_abbrev action_id : string
```

```
type_abbrev thread_id : string
```

```
type_abbrev location : string
```

```
type_abbrev val : string
```

```
memory_order_enum =
    Mo_seq_cst
  | Mo_relaxed
  | Mo_release
  | Mo_acquire
  | Mo_consume
  | Mo_acq_rel
```

```
action =
    Lock of action_id thread_id location
  | Unlock of action_id thread_id location
  | Atomic_load of action_id thread_id memory_order_enum location val
  | Atomic_store of action_id thread_id memory_order_enum location val
  | Atomic_rmw of action_id thread_id memory_order_enum location val val
  | Load of action_id thread_id location val
  | Store of action_id thread_id location val
  | Fence of action_id thread_id memory_order_enum
```

```
(action_id_of (Lock aid _ _) = aid) $\land$
(action_id_of (Unlock aid _ _) = aid) $\land$
(action_id_of (Atomic_load aid _ _ _ _) = aid) $\land$
(action_id_of (Atomic_store aid _ _ _ _) = aid) $\land$
(action_id_of (Atomic_rmw aid _ _ _ _ _) = aid) $\land$
(action_id_of (Load aid _ _ _) = aid) $\land$
(action_id_of (Store aid _ _ _) = aid) $\land$
(action_id_of (Fence aid _ _) = aid)
```

```
(thread_id_of (Lock _ tid _) = tid) $\land$
(thread_id_of (Unlock _ tid _) = tid) $\land$
(thread_id_of (Atomic_load _ tid _ _ _) = tid) $\land$
(thread_id_of (Atomic_store _ tid _ _ _) = tid) $\land$
(thread_id_of (Atomic_rmw _ tid _ _ _ _) = tid) $\land$
(thread_id_of (Load _ tid _ _) = tid) $\land$
(thread_id_of (Store _ tid _ _) = tid) $\land$
(thread_id_of (Fence _ tid _) = tid)
```

```
(memory_order (Atomic_load _ _ mem_ord _ _) =
  Some mem_ord) $\land$
(memory_order (Atomic_store _ _ mem_ord _ _) =
  Some mem_ord) $\land$
(memory_order (Atomic_rmw _ _ mem_ord _ _ _) =
  Some mem_ord) $\land$
(memory_order (Fence _ _ mem_ord) =
  Some mem_ord) $\land$
(memory_order _ =
  None)
```

```
(location (Lock _ _ l) = Some l) $\land$
(location (Unlock _ _ l) = Some l) $\land$
(location (Atomic_load _ _ _ l _) = Some l) $\land$
(location (Atomic_store _ _ _ l _) = Some l) $\land$
(location (Atomic_rmw _ _ _ l _ _) = Some l) $\land$
(location (Load _ _ l _) = Some l) $\land$
(location (Store _ _ l _) = Some l) $\land$
(location (Fence _ _ _) = None)
```

```
(value_read (Atomic_load _ _ _ _ v) = Some v) $\land$
(value_read (Atomic_rmw _ _ _ _ v _) = Some v) $\land$
(value_read (Load _ _ _ v) = Some v) $\land$
(value_read _ = None)
```

```
(value_written (Atomic_store _ _ _ _ v) = Some v) $\land$
(value_written (Atomic_rmw _ _ _ _ _ v) = Some v) $\land$
(value_written (Store _ _ _ v) = Some v) $\land$
(value_written _ = None)
```

```
is_lock a =
  case a of Lock _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_unlock a =
  case a of Unlock _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_atomic_load a =
  case a of Atomic_load _ _ _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_atomic_store a =
  case a of Atomic_store _ _ _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_atomic_rmw a =
  case a of Atomic_rmw _ _ _ _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_load a = case a of Load _ _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_store a = case a of Store _ _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_fence a = case a of Fence _ _ _ $\rightarrow$ T $\|$ _ $\rightarrow$ F
```

```
is_lock_or_unlock a = is_lock a $\lor$ is_unlock a
```

```
is_atomic_action a =
  is_atomic_load a $\lor$ is_atomic_store a $\lor$ is_atomic_rmw a
```

```
is_load_or_store a = is_load a $\lor$ is_store a
```

```
is_read a =
  is_atomic_load a $\lor$ is_atomic_rmw a $\lor$ is_load a
```

```
is_write a =
  is_atomic_store a $\lor$ is_atomic_rmw a $\lor$ is_store a
```

```
is_acquire a =
  (case memory_order a of
    Some mem_ord $\rightarrow$
      (mem_ord $\in$
        {Mo_acquire, Mo_acq_rel, Mo_seq_cst} $\land$
      (is_read a $\lor$ is_fence a)) $\lor$
      (* 29.8:5 states that consume fences are acquire fences. *)
      ((mem_ord = Mo_consume) $\land$ is_fence a)
  $\|$ None $\rightarrow$ is_lock a)
```

```
is_consume a =
  is_read a $\land$ (memory_order a = Some Mo_consume)
```

```
is_release a =
  (case memory_order a of
    Some mem_ord $\rightarrow$
      mem_ord $\in$ {Mo_release, Mo_acq_rel, Mo_seq_cst} $\land$
    (is_write a $\lor$ is_fence a)
  $\|$ None $\rightarrow$ is_unlock a)
```

```
is_seq_cst a = (memory_order a = Some Mo_seq_cst)
```

```
location_kind =
    Mutex
  | Non_atomic
  | Atomic
```

```
actions_respect_location_kinds = actions_respect_location_kinds =
  $\forall a$.
    case location a of Some l $\rightarrow$
      (case location-kind l of
        Mutex $\rightarrow$ is_lock_or_unlock a
      $\|$ Non_atomic $\rightarrow$ is_load_or_store a
      $\|$ Atomic $\rightarrow$ is_load_or_store a $\lor$ is_atomic_action a)
    $\|$ None $\rightarrow$ T
```

```
is_at_location_kind = is_at_location_kind =
  case location a of
    Some l $\rightarrow$ (location-kind l = lk0)
  $\|$ None $\rightarrow$ F
```

```
is_at_mutex_location a =
  is_at_location_kind a Mutex
```

```
is_at_non_atomic_location a =
  is_at_location_kind a Non_atomic
```

```
is_at_atomic_location a =
  is_at_location_kind a Atomic
```

```
same_thread a b = (thread_id_of a = thread_id_of b)
```

```
threadwise_relation_over s rel =
  relation_over s rel $\land$ ($\forall$(a, b) $\in$ rel. same_thread a b)
```

```
same_location a b = (location a = location b)
```

```
locations_of actions = {l. $\exists$a. (location a = Some l)}
```

```
well_formed_action a =
  case a of
    Atomic_load _ _ mem_ord _ _ $\rightarrow$ mem_ord $\in$
      {Mo_relaxed, Mo_acquire, Mo_seq_cst, Mo_consume}
  $\|$ Atomic_store _ _ mem_ord _ _ $\rightarrow$ mem_ord $\in$
      {Mo_relaxed, Mo_release, Mo_seq_cst}
  $\|$ Atomic_rmw _ _ mem_ord _ _ _ $\rightarrow$ mem_ord $\in$
      {Mo_relaxed, Mo_release, Mo_acquire, Mo_acq_rel, Mo_seq_cst, Mo_consume}
  $\|$ _ $\rightarrow$ T
```

```
well_formed_threads = well_formed_threads =
  inj_on action_id_of (actions) $\land$
  ($\forall$a. well_formed_action a) $\land$
  threadwise_relation_over actions sequenced-before $\land$
  threadwise_relation_over actions data-dependency $\land$
  threadwise_relation_over actions control-dependency $\land$
  strict_preorder sequenced-before $\land$
  strict_preorder data-dependency $\land$
  strict_preorder control-dependency $\land$
  relation_over actions additional-synchronized-with $\land$
  ($\forall$a. thread_id_of a $\in$ threads) $\land$
  actions_respect_location_kinds $\land$
  data-dependency $\subseteq$ sequenced-before
```

```
well_formed_reads_from_mapping = well_formed_reads_from_mapping =
  relation_over actions ($\xrightarrow{rf}$) $\land$
  ($\forall a. \forall a'. \forall b.\ a \xrightarrow{rf} b \land a' \xrightarrow{rf} b \implies (a = a')$) $\land$
  ($\forall(a, b) \in \xrightarrow{rf}$.
    same_location a b $\land$
    (value_read b = value_written a) $\land$
    ($a \neq b$) $\land$
    (is_at_mutex_location a $\implies$
      is_unlock a $\land$ is_lock b) $\land$
    (is_at_non_atomic_location a $\implies$
      is_store a $\land$ is_load b) $\land$
    (is_at_atomic_location a $\implies$
      (is_atomic_store a $\lor$ is_atomic_rmw a $\lor$ is_store a)
      $\land$ (is_atomic_load b $\lor$ is_atomic_rmw b $\lor$ is_load b)))
```

```
all_lock_or_unlock_actions_at lopt as =
  {a $\in$ as. is_lock_or_unlock a $\land$ (location a = lopt)}
```

```
consistent_locks = consistent_locks =
$\forall$l $\in$ locations_of actions. (location-kind l = Mutex) $\implies$ (
  let lock_unlock_actions =
    all_lock_or_unlock_actions_at (Some l)actions in
  let lock_order = $\xrightarrow{sc}|_{lock\_unlock\_actions}$ in
  (* 30.4.1:5 - The implementation shall serialize those (lock and unlock) operations. *)
  strict_total_order_over lock_unlock_actions lock_order $\land$

  (* 30.4.1:1 A thread owns a mutex from the time it successfully calls one of the lock functions until it calls unlock.*)
  (* 30.4.1:20 Requires: The calling thread shall own the mutex. *)
  (* 30.4.1:21 Effects: Releases the calling threads ownership of the mutex.*)
  ($\forall a_u \in$ lock_unlock_actions. is_unlock $a_u \implies$
    ($\exists a_l \in$ lock_unlock_actions.
      $a_l \overset{lock\_order}{\longmapsto} a_u$ $\land$ same_thread $a_l\ a_u$ $\land$ is_lock $a_l$)) $\land$

  (* 30.4.1:7 Effects: Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.*)
  (* 30.4.1:8 Postcondition: The calling thread owns the mutex. *)
  ($\forall a_l \in$ lock_unlock_actions. is_lock $a_l \implies$
    ($\forall a_u \in$ lock_unlock_actions.
      $a_u \overset{lock\_order}{\longmapsto} a_l \implies$ is_unlock $a_u$)))
```

```
rs_element rs_head a =
  same_thread a rs_head $\lor$ is_atomic_rmw a
```

```
release_sequence = $a_{rel} \xrightarrow{release\text{-}sequence} b =$
  is_at_atomic_location b $\land$
  is_release $a_{rel}$ $\land$ (
    ($b = a_{rel}$) $\lor$
    (rs_element $a_{rel}$ b $\land$ $a_{rel} \xrightarrow{modification\text{-}order} b$ $\land$
      ($\forall c.\ a_{rel} \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b \implies$
        rs_element $a_{rel}$ c)))
```

```
release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =

release_sequence actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order a b}
```

```
hypothetical_release_sequence = $a \xrightarrow{hypothetical\text{-}release\text{-}sequence} b =$
  is_at_atomic_location b $\land$ (
  ($b = a$) $\lor$
  (rs_element a b $\land$ $a \xrightarrow{modification\text{-}order} b$ $\land$
    ($\forall c.\ a \xrightarrow{modification\text{-}order} c \xrightarrow{modification\text{-}order} b \implies$
      rs_element a c)))
```

```
hypothetical_release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order =

hypothetical_release_sequence actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order a b}
```

```
synchronizes_with = $a \xrightarrow{synchronizes\text{-}with} b =$
  (* – additional synchronization, from thread create etc. – *)
  $a \xrightarrow{additional\text{-}synchronized\text{-}with} b$ $\lor$

  (same_location a b $\land$ a $\in$ actions $\land$ b $\in$ actions $\land$ (
    (* – mutex synchronization – *)
    (is_unlock a $\land$ is_lock b $\land$ $a \xrightarrow{sc} b$) $\lor$

    (* – release/acquire synchronization – *)
    (is_release a $\land$ is_acquire b $\land$ $\neg$same_thread a b $\land$
    ($\exists c.\ a \xrightarrow{release\text{-}sequence} c \xrightarrow{rf} b$)) $\lor$

    (* – fence synchronization – *)
    (is_fence a $\land$ is_release a $\land$ is_fence b $\land$ is_acquire b $\land$
    ($\exists x. \exists y.$ same_location x y $\land$
      is_atomic_action x $\land$ is_atomic_action y $\land$ is_write x $\land$
      $a \xrightarrow{sequenced\text{-}before} x \land y \xrightarrow{sequenced\text{-}before} b$ $\land$
      ($\exists z.\ x \xrightarrow{hypothetical\text{-}release\text{-}sequence} z \xrightarrow{rf} y$))) $\lor$

    (is_fence a $\land$ is_release a $\land$
     is_atomic_action b $\land$ is_acquire b $\land$
    ($\exists x.$ same_location x b $\land$
      is_atomic_action x $\land$ is_write x $\land$
      $a \xrightarrow{sequenced\text{-}before} x$ $\land$
      ($\exists z.\ x \xrightarrow{hypothetical\text{-}release\text{-}sequence} z \xrightarrow{rf} b$))) $\lor$

    (is_atomic_action a $\land$ is_release a $\land$
     is_fence b $\land$ is_acquire b $\land$
    ($\exists x.$ same_location a x $\land$ is_atomic_action x $\land$
      $x \xrightarrow{sequenced\text{-}before} b$ $\land$
      ($\exists z.\ a \xrightarrow{release\text{-}sequence} z \xrightarrow{rf} x$)))))
```

```
synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence =

synchronizes_with actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence a b}
```

```
carries_a_dependency_to = $a \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to} b =$
  $a\ ((\xrightarrow{rf} \cap \xrightarrow{sequenced\text{-}before}) \cup \xrightarrow{data\text{-}dependency})^{+}\ b$
```

```
carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf =

carries_a_dependency_to actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf a b}
```

```
dependency_ordered_before = $a \xrightarrow{dependency\text{-}ordered\text{-}before} d =$
  a $\in$ actions $\land$ d $\in$ actions $\land$
  ($\exists b.$ is_release a $\land$ is_consume b $\land$
    ($\exists e.\ a \xrightarrow{release\text{-}sequence} e \xrightarrow{rf} b$) $\land$
    ($b \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to} d \lor (b = d)$))
```

```
dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to =

dependency_ordered_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to a b}
```

```
simple_happens_before = $\xrightarrow{simple\_happens\_before} =$
  $(\xrightarrow{sequenced\text{-}before} \cup \xrightarrow{synchronizes\text{-}with})^{+}$
```

```
consistent_simple_happens_before shb =
  irreflexive ($\xrightarrow{shb}$)
```

```
inter_thread_happens_before = $\xrightarrow{inter\text{-}thread\text{-}happens\text{-}before} =$
  let $r = \xrightarrow{synchronizes\text{-}with} \cup$
          $\xrightarrow{dependency\text{-}ordered\text{-}before} \cup$
          $(\xrightarrow{synchronizes\text{-}with} \circ \xrightarrow{sequenced\text{-}before})$ in
  $(\xrightarrow{r} \cup (\xrightarrow{sequenced\text{-}before} \circ \xrightarrow{r}))^{+}$
```

```
consistent_inter_thread_happens_before = consistent_inter_thread_happens_before =
  irreflexive ($\xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$)
```

```
happens_before = $\xrightarrow{happens\text{-}before} =$
  $\xrightarrow{sequenced\text{-}before} \cup \xrightarrow{inter\text{-}thread\text{-}happens\text{-}before}$
```

```
all_sc_actions = all_sc_actions =
  {a. (is_seq_cst a $\lor$ is_lock a $\lor$ is_unlock a)}
```

```
consistent_sc_order = consistent_sc_order =
  let sc_happens_before = $\xrightarrow{happens\text{-}before}|_{all\_sc\_actions}$ in
  let sc_mod_order = $\xrightarrow{modification\text{-}order}|_{all\_sc\_actions}$ in
  strict_total_order_over all_sc_actions ($\xrightarrow{sc}$) $\land$
  $\xrightarrow{sc\_happens\_before} \subseteq \xrightarrow{sc}$ $\land$
  $\xrightarrow{sc\_mod\_order} \subseteq \xrightarrow{sc}$
```

```
consistent_modification_order = consistent_modification_order =
  ($\forall a. \forall b.\ a \xrightarrow{modification\text{-}order} b \implies$ same_location a b) $\land$
  ($\forall$l $\in$ locations_of actions. case location-kind l of
    Atomic $\rightarrow$ (
      let actions_at_l = {a. (location a = Some l)} in
      let writes_at_l = {a_at_l. (is_store a $\lor$
        is_atomic_store a $\lor$ is_atomic_rmw a)} in
        strict_total_order_over writes_at_l
        ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) $\land$
        (* happens-before at the writes of l is a subset of mo for l *)
        $\xrightarrow{happens\text{-}before}|_{writes\_at\_l} \subseteq \xrightarrow{modification\text{-}order}$ $\land$
        (* Mo_seq_cst fences impose modification order *)
        $(\xrightarrow{sequenced\text{-}before} \circ (\xrightarrow{sc}|_{is\_fence}) \circ \xrightarrow{sequenced\text{-}before}|_{writes\_at\_l})$
          $\subseteq \xrightarrow{modification\text{-}order}$)
    $\|$ _ $\rightarrow$ (
      let actions_at_l = {a. (location a = Some l)} in
      ($\xrightarrow{modification\text{-}order}|_{actions\_at\_l}$) = {}))
```

```
visible_side_effect = $a \xrightarrow{visible\text{-}side\text{-}effect} b =$
  $a \xrightarrow{happens\text{-}before} b$ $\land$
  is_write a $\land$ is_read b $\land$ same_location a b $\land$
  $\neg(\exists c.\ (c \neq a) \land (c \neq b) \land$
    is_write c $\land$ same_location c b $\land$
    $a \xrightarrow{happens\text{-}before} c \xrightarrow{happens\text{-}before} b)$
```

```
visible_side_effect_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before =
{ab $\in$ happens-before. let (a, b) = ab in
 visible_side_effect actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before a b}
```

```
visible_sequence_of_side_effects_tail = visible_sequence_of_side_effects_tail vsse_head b =
  {c. $vsse\_head \xrightarrow{modification\text{-}order} c$ $\land$
    $\neg(b \xrightarrow{happens\text{-}before} c)$ $\land$
    ($\forall a.\ vsse\_head \xrightarrow{modification\text{-}order} a \xrightarrow{modification\text{-}order} c$
      $\implies \neg(b \xrightarrow{happens\text{-}before} a)$)}
```

```
myimage f s = {y. $\exists$x $\in$ s. (y = f x)}
```

```
visible_sequences_of_side_effects = visible_sequences_of_side_effects =
  $\lambda$(vsse_head, b).
    (b, if is_at_atomic_location b then
          {vsse_head} $\cup$
          visible_sequence_of_side_effects_tail vsse_head b
        else
          {})
```

```
visible_sequences_of_side_effects_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect =
    myimage (visible_sequences_of_side_effects actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect)visible-side-effect
```

```
consistent_reads_from_mapping = consistent_reads_from_mapping =
  ($\forall$b. (is_read b $\land$ is_at_non_atomic_location b) $\implies$
      (if ($\exists a_{vse}.\ a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b$)
       then ($\exists a_{vse}.\ a_{vse} \xrightarrow{visible\text{-}side\text{-}effect} b \land a_{vse} \xrightarrow{rf} b$)
       else $\neg(\exists a.\ a \xrightarrow{rf} b)$)) $\land$

  ($\forall$b. (is_read b $\land$ is_at_atomic_location b) $\implies$
      (if ($\exists$(b', vsse) $\in$ visible-sequences-of-side-effects. (b' = b))
       then ($\exists$(b', vsse) $\in$ visible-sequences-of-side-effects.
              (b' = b) $\land$ ($\exists c \in vsse.\ c \xrightarrow{rf} b$))
       else $\neg(\exists a.\ a \xrightarrow{rf} b)$)) $\land$

  ($\forall(x, a) \in \xrightarrow{rf}$.
    $\forall(y, b) \in \xrightarrow{rf}$.
      $a \xrightarrow{happens\text{-}before} b$ $\land$
        same_location a b $\land$ is_at_atomic_location b
        $\implies (x = y) \lor x \xrightarrow{modification\text{-}order} y$) $\land$
(* new CoWR *)
  ($\forall(a, b) \in \xrightarrow{happens\text{-}before}$.
    $\forall c$.
      $c \xrightarrow{rf} b$ $\land$
        is_write a $\land$ same_location a b $\land$ is_at_atomic_location b
        $\implies (c = a) \lor a \xrightarrow{modification\text{-}order} c$) $\land$
(* new CoRW *)
  ($\forall(a, b) \in \xrightarrow{happens\text{-}before}$.
    $\forall c$.
      $c \xrightarrow{rf} a$ $\land$
        is_write b $\land$ same_location a b $\land$ is_at_atomic_location a
        $\implies c \xrightarrow{modification\text{-}order} b$) $\land$

  ($\forall(a, b) \in \xrightarrow{rf}$. is_atomic_rmw b
    $\implies a \overset{modification\text{-}order}{\longmapsto} b$) $\land$

  ($\forall(a, b) \in \xrightarrow{rf}$. is_seq_cst b
    $\implies$ ($\neg$is_seq_cst a $\land$ ($\forall x.\ x \overset{sc}{\longmapsto}_{\lambda c.\ is\_write\ c \land same\_location\ b\ c}\ b \implies x \xrightarrow{modification\text{-}order} a$)) $\lor$
    $a \overset{sc}{\longmapsto}_{\lambda c.\ is\_write\ c \land same\_location\ b\ c}\ b$) $\land$

  (* -Fence restrictions- *)

  (* 29.3:3 *)
  ($\forall a. \forall(x, b) \in \xrightarrow{sequenced\text{-}before}. \forall y$.
    (is_fence x $\land$ is_seq_cst x $\land$ is_atomic_action b $\land$
      is_write a $\land$ same_location a b $\land$
     $a \overset{sc}{\longmapsto} x \land y \xrightarrow{rf} b$)
    $\implies (y = a) \lor a \xrightarrow{modification\text{-}order} y$) $\land$

  (* 29.3:4 *)
  ($\forall(a, x) \in \xrightarrow{sequenced\text{-}before}. \forall(y, b) \in \xrightarrow{rf}$.
    (is_atomic_action a $\land$ is_fence x $\land$ is_seq_cst x $\land$
      is_write a $\land$ same_location a b $\land$
     $x \xrightarrow{sc} b$ $\land$ is_atomic_action b)
    $\implies (y = a) \lor a \xrightarrow{modification\text{-}order} y$) $\land$

  (* 29.3:5 *)
  ($\forall(a, x) \in \xrightarrow{sequenced\text{-}before}. \forall(y, b) \in \xrightarrow{sequenced\text{-}before}. \forall z$.
    (is_atomic_action a $\land$ is_fence x $\land$ is_seq_cst x $\land$
      is_write a $\land$ is_fence y $\land$ is_seq_cst y $\land$
      is_atomic_action b $\land$ same_location a b $\land$
     $x \xrightarrow{sc} y \land z \xrightarrow{rf} b$)
    $\implies (z = a) \lor a \xrightarrow{modification\text{-}order} z$)
```

```
all_data_dependency = $\xrightarrow{all\_data\_dependency} =$
  $(\xrightarrow{rf} \cup \xrightarrow{carries\text{-}a\text{-}dependency\text{-}to})^{+}$
```

```
consistent_control_dependency = consistent_control_dependency =
  irreflexive $((\xrightarrow{control\text{-}dependency} \cup \xrightarrow{all\_data\_dependency})^{+})$
```

```
consistent_execution actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc =
well_formed_threads actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency $\land$
consistent_locks actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency sc $\land$ (
let release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let hypothetical-release-sequence = hypothetical_release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let synchronizes-with = synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence in
let carries-a-dependency-to = carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf in
let dependency-ordered-before = dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to in
let inter-thread-happens-before = inter_thread_happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency synchronizes-with dependency-ordered-before in
let happens-before = happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency inter-thread-happens-before in
let visible-side-effect = visible_side_effect_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before in
let visible-sequences-of-side-effects = visible_sequences_of_side_effects_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order happens-before visible-side-effect in
consistent_inter_thread_happens_before inter-thread-happens-before $\land$
consistent_sc_order actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order sc happens-before $\land$
consistent_modification_order actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency sc modification-order happens-before $\land$
well_formed_reads_from_mapping actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf $\land$
consistent_reads_from_mapping actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf sc modification-order happens-before visible-side-effect visible-sequences-of-side-effects)
```

```
indeterminate_reads actions threads = indeterminate_reads =
  {b. is_read b $\land \neg(\exists a.\ a \xrightarrow{rf} b)$}
```

```
unsequenced_races = unsequenced_races = {(a, b).
  ($a \neq b$) $\land$ same_location a b $\land$ (is_write a $\lor$ is_write b) $\land$
  same_thread a b $\land$
  $\neg(a \xrightarrow{sequenced\text{-}before} b \lor b \xrightarrow{sequenced\text{-}before} a)$}
```

```
data_races = data_races = {(a, b).
  ($a \neq b$) $\land$ same_location a b $\land$ (is_write a $\lor$ is_write b) $\land$
  $\neg$same_thread a b $\land$
  $\neg$(is_atomic_action a $\land$ is_atomic_action b) $\land$
  $\neg(a \xrightarrow{happens\text{-}before} b \lor b \xrightarrow{happens\text{-}before} a)$}
```

```
data_races' actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc =
let release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let hypothetical-release-sequence = release_sequence_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency modification-order in
let synchronizes-with = synchronizes_with_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc release-sequence hypothetical-release-sequence in
let carries-a-dependency-to = carries_a_dependency_to_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf in
let dependency-ordered-before = dependency_ordered_before_set actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order release-sequence carries-a-dependency-to in
let inter-thread-happens-before = inter_thread_happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency synchronizes-with dependency-ordered-before in
let happens-before = happens_before actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency inter-thread-happens-before in
data_races actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency happens-before
```

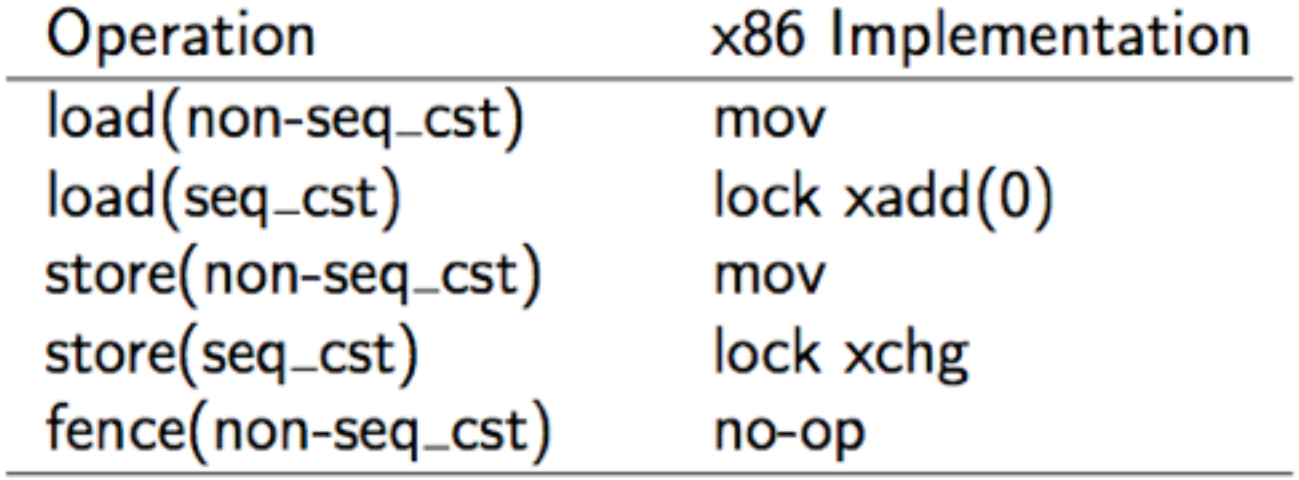
```
cpp_memory_model opsem (p $\in$ 'program) =
  let executions = {(actions, threads, location-kind, sequenced-before, additional-synchronized-with, data-dependency, control-dependency, rf, modification-order, sc).
      opsem p actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency $\land$ consistent_execution actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc} in
    if $\exists$(actions, threads, location-kind, sequenced-before, additional-synchronized-with, data-dependency, control-dependency, rf, modification-order, sc) $\in$ executions .
        (indeterminate_reads actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf $\neq$ {}) $\lor$
        (unsequenced_races actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency $\neq$ {}) $\lor$
        (data_races' actions threads location-kind sequenced-before additional-synchronized-with data-dependency control-dependency rf modification-order sc $\neq$ {})
     then {}
     else executions
```

Is C++11 hopelessly complicated?

Programmers cannot be given this model.

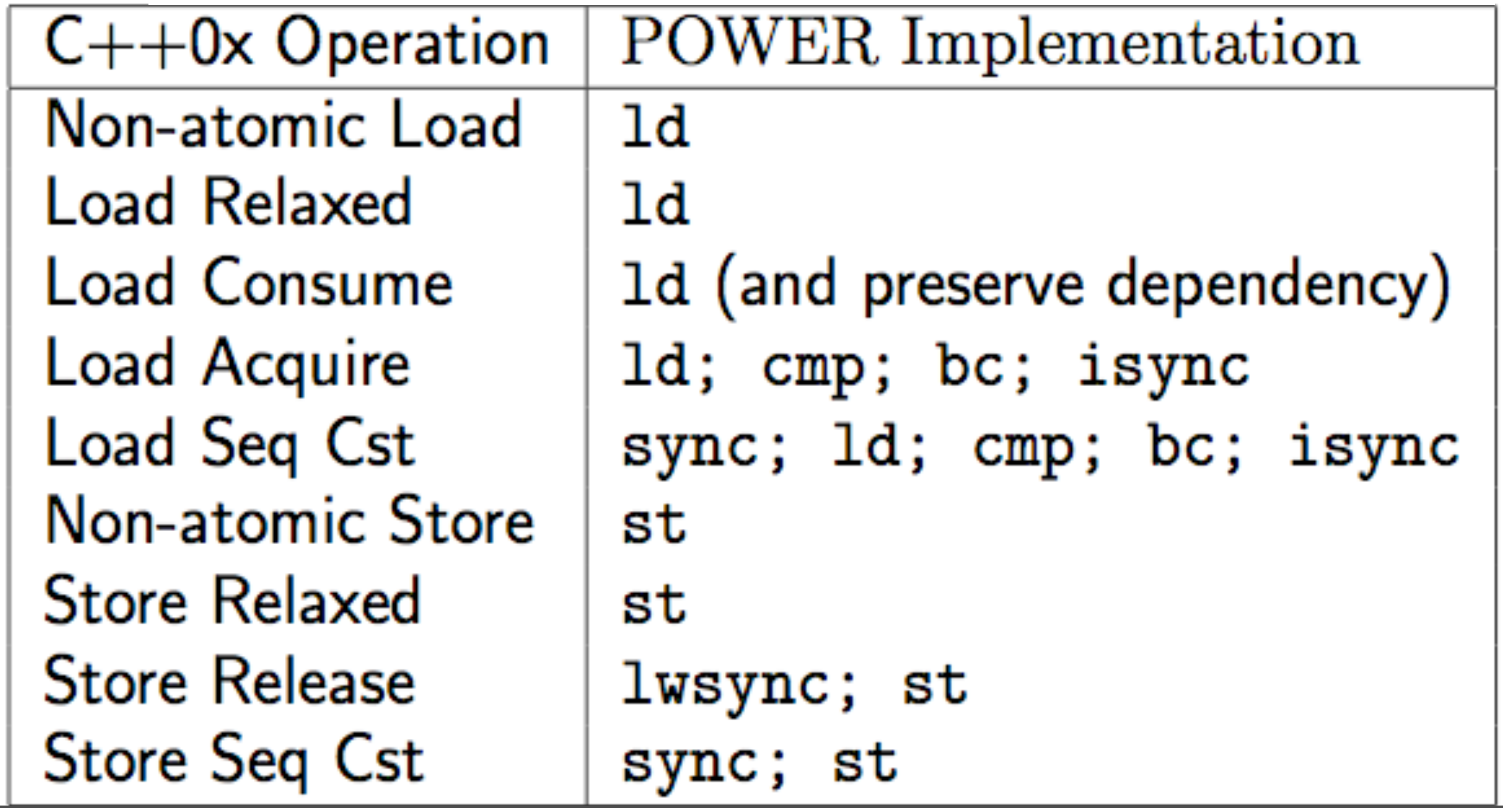
However, with a formal definition, we can do proofs!

- Can we compile to x86?



Operation	x86 Implementation
load(non-seq_cst)	mov
load(seq_cst)	lock xadd(0)
store(non-seq_cst)	mov
store(seq_cst)	lock xchg
fence(non-seq_cst)	no-op

- Can we compile to Power?



C++0x Operation	POWER Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`sync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`sync; st`

Is C++11 hopelessly complicated?

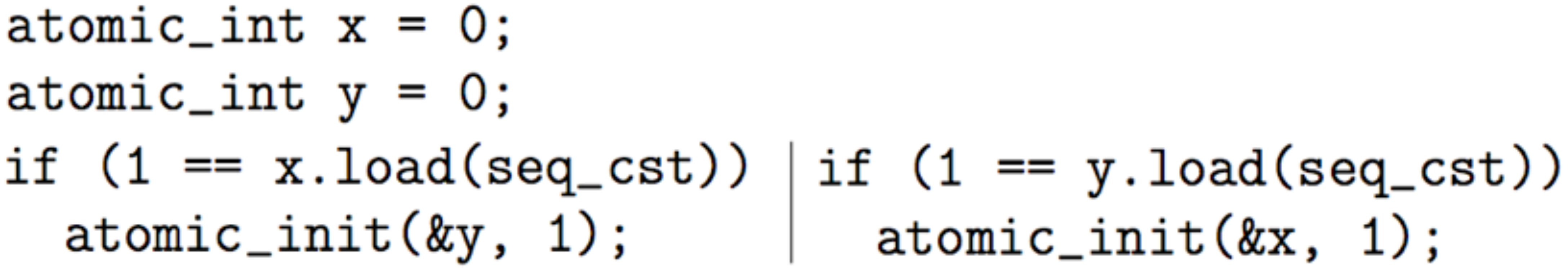
Simplifications:

Full model: *visible sequences of side effects* are unneded (HOL4)

Derivative models:

- without consume, happens-before is transitive

- DRF programs using only **`seq_cst`** atomics are SC (false)



```
atomic_int x = 0;
atomic_int y = 0;
if (1 == x.load(seq_cst))  | if (1 == y.load(seq_cst))
  atomic_init(&y, 1);      |   atomic_init(&x, 1);
```

`atomic_init` is a non-atomic write, and in C++11 they race.

The current state of the standard

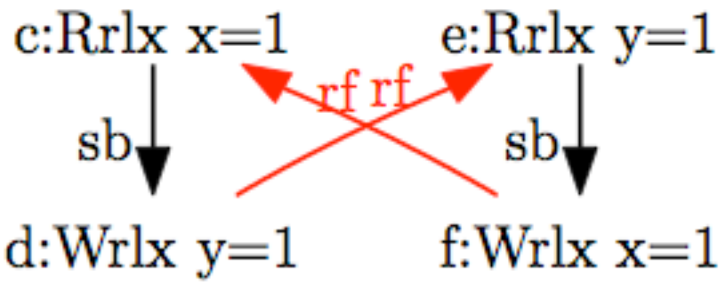
Fixed:

- in some cases, happens-before was cyclic
- coherence
- `seq_cst` atomics were more broken

Not fixed:

- self satisfying conditional

```
r1 = x.load(mo_relaxed);      | r2 = y.load(mo_relaxed);
if (r1 == 42)                 | if (r2 == 42)
  y.store(r1, mo_relaxed);    |   x.store(42, mo_relaxed);
```

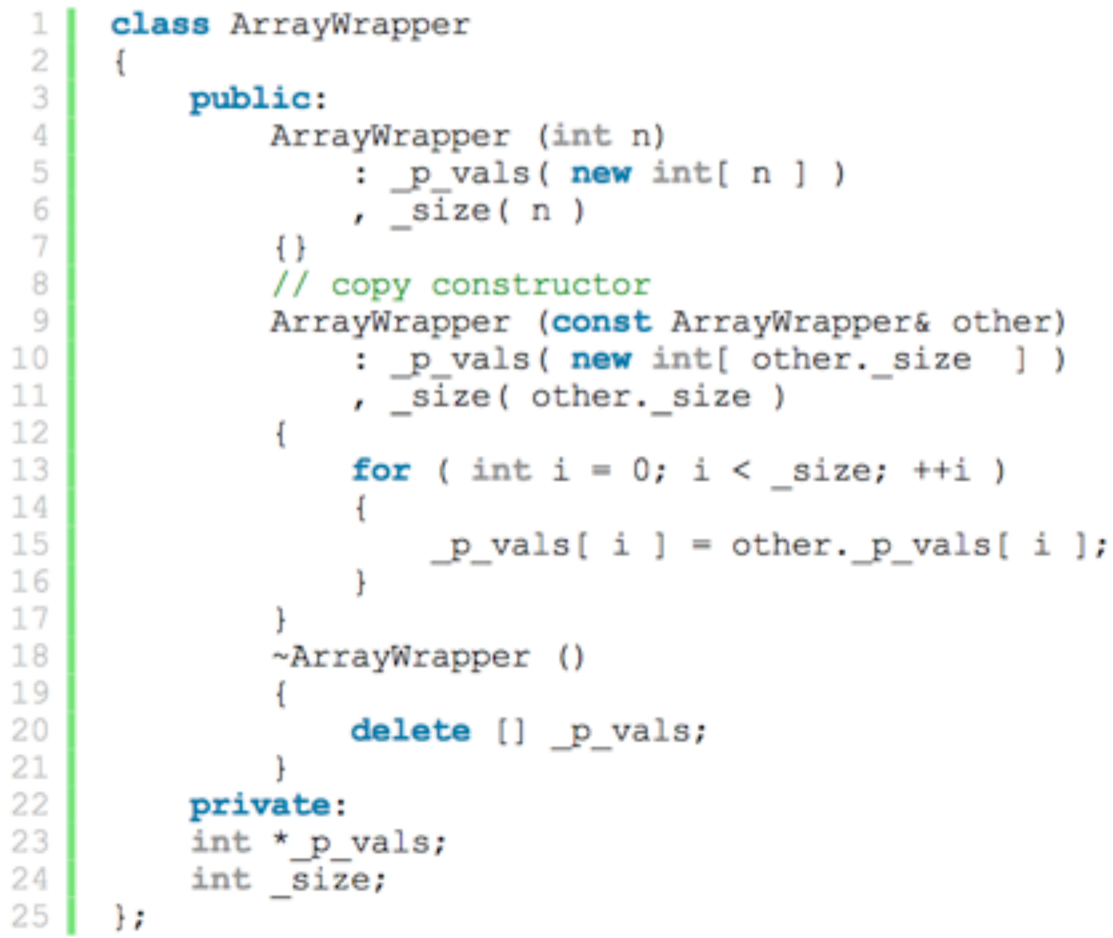


- `seq_cst` atomics are still not SC

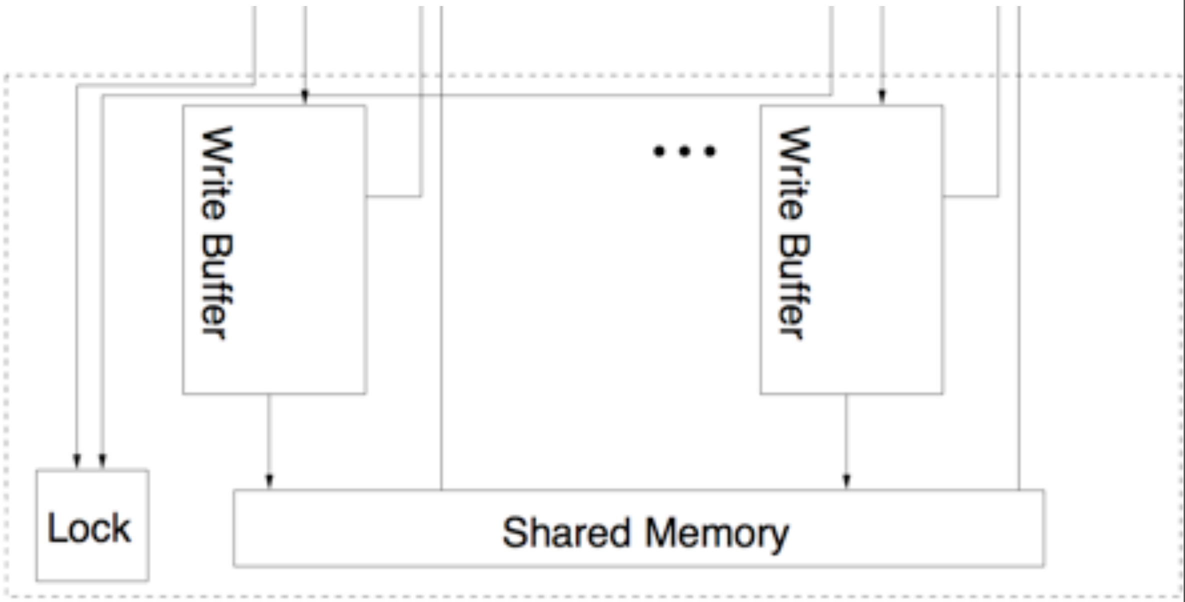
3. Sketch of an operational formalisation of x86-TSO

...starting with a formalisation of SC

Separate language and memory semantics



```
class ArrayWrapper
{
    public:
        ArrayWrapper (int n)
            : _p_vals( new int[ n ] )
            , _size( n )
        {}
        // copy constructor
        ArrayWrapper (const ArrayWrapper& other)
            : _p_vals( new int[ other._size  ] )
            , _size( other._size )
        {
            for ( int i = 0; i < _size; ++i )
            {
                _p_vals[ i ] = other._p_vals[ i ];
            }
        }
        ~ArrayWrapper ()
        {
            delete [] _p_vals;
        }
    private:
    int *_p_vals;
    int _size;
};
```



program
semantics defined via an LTS

memory
semantics defined via an LTS

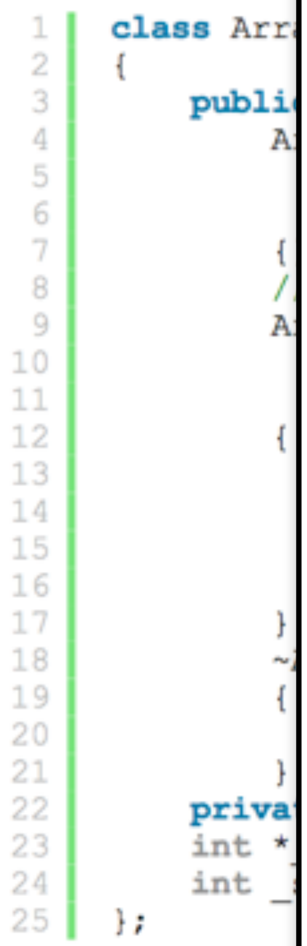
Labels for interaction:

$W_t[a]v$: a write of value **v** to address **a** by thread **t**

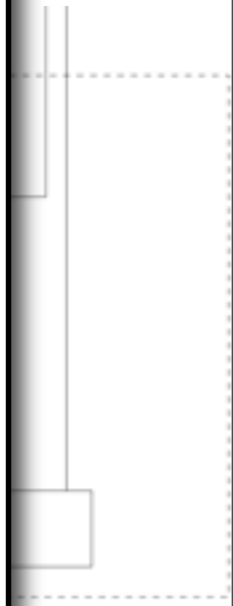
$R_t[a]v$: a read of **v** from **a** by **t** by thread **t**

+ other events for barriers and locked instructions

Separate language and memory semantics



```
class Arr
{
    publi
        A

        {
        /
        A

        {




        }
        ~
        {

        }
    priva
    int *
    int _
};
```



Separate language and state semantics

proved to be a very good choice

in many (unrelated) projects I worked on!

semantics defined via an LTS

semantics defined via an LTS

Labels for interaction:

$W_t[a]v$: a write of value **v** to address **a** by thread **t**
$R_t[a]v$: a read of **v** from **a** by **t** by thread **t**
+ other events for barriers and locked instructions

A tiny language

$location,\ x,\ m$	address (or pointer value)
$integer,\ n$	integer
$thread\_id,\ t$	thread id
$k,\ i,\ j$	

$expression,\ e$	$::=$		expression
	$\vert$	n	integer literal
	$\vert$	$*x$	read from pointer
	$\vert$	$*x = e$	write to pointer
	$\vert$	$e; e'$	sequential composition
	$\vert$	$e + e'$	plus
$process,\ p$	$::=$		process
	$\vert$	$t{:}e$	thread
	$\vert$	$p \vert p'$	parallel composition

What can a thread do in isolation?

$\boxed{e \xrightarrow{l} e'}$ e does l to become e'

$$\frac{}{*x \xrightarrow{\mathsf{R}\ x=n} n}$$ READ

$$\frac{}{*x = n \xrightarrow{\mathsf{W}\ x=n} n}$$ WRITE

$$\frac{e \xrightarrow{l} e'}{*x = e \xrightarrow{l} *x = e'}$$ WRITE_CONTEXT

$$\frac{}{n; e \xrightarrow{\tau} e}$$ SEQ

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1; e_2 \xrightarrow{l} e_1'; e_2}$$ SEQ_CONTEXT

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1 + e_2 \xrightarrow{l} e_1' + e_2}$$ PLUS_CONTEXT_1

$$\frac{e_2 \xrightarrow{l} e_2'}{n_1 + e_2 \xrightarrow{l} n_1 + e_2'}$$ PLUS_CONTEXT_2

$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\tau} n}$$ PLUS

Observe that we can read an arbitrary value from the memory.

Lifting to processes

$\boxed{p \xrightarrow{l_t} p'}$ p does l_t to become p'

$$\frac{e \xrightarrow{l} e'}{t{:}e \xrightarrow{l_t} t{:}e'} \quad \text{THREAD}$$

Actions are labelled by the thread that performed the action.

$$\frac{p_1 \xrightarrow{l_t} p_1'}{p_1|p_2 \xrightarrow{l_t} p_1'|p_2} \quad \text{PAR\_CONTEXT\_LEFT}$$

Free interleaving.

$$\frac{p_2 \xrightarrow{l_t} p_2'}{p_1|p_2 \xrightarrow{l_t} p_1|p_2'} \quad \text{PAR\_CONTEXT\_RIGHT}$$

A sequentially consistent memory

Take M to be a function from addresses to integers.

$\boxed{M \xrightarrow{l} M'}$ M does l to become M'

$$\frac{M(x) = n}{M \xrightarrow{\mathsf{R}\, x=n} M} \quad \textsc{MRead}$$

$$\frac{}{M \xrightarrow{\mathsf{W}\, x=n} M \oplus (x \mapsto n)} \quad \textsc{MWrite}$$

SC semantics: whole system transitions

$\boxed{s \xrightarrow{l_t} s'}$ s does l_t to become s'

$$\frac{\begin{array}{c} p \xrightarrow{\mathsf{R}_t\, x=n} p' \\ M \xrightarrow{\mathsf{R}\, x=n} M' \end{array}}{\langle p,\ M\rangle \xrightarrow{\mathsf{R}_t\, x=n} \langle p',\ M'\rangle}\quad \textsc{SRead}$$

$$\frac{\begin{array}{c} p \xrightarrow{\mathsf{W}_t\, x=n} p' \\ M \xrightarrow{\mathsf{W}\, x=n} M' \end{array}}{\langle p,\ M\rangle \xrightarrow{\mathsf{W}_t\, x=n} \langle p',\ M'\rangle}\quad \textsc{SWrite}$$

$$\frac{p \xrightarrow{\tau_t} p'}{\langle p,\ M\rangle \xrightarrow{\tau_t} \langle p',\ M\rangle}\quad \textsc{STau}$$

Synchronising between the processes and the memory.

SC semantics, example

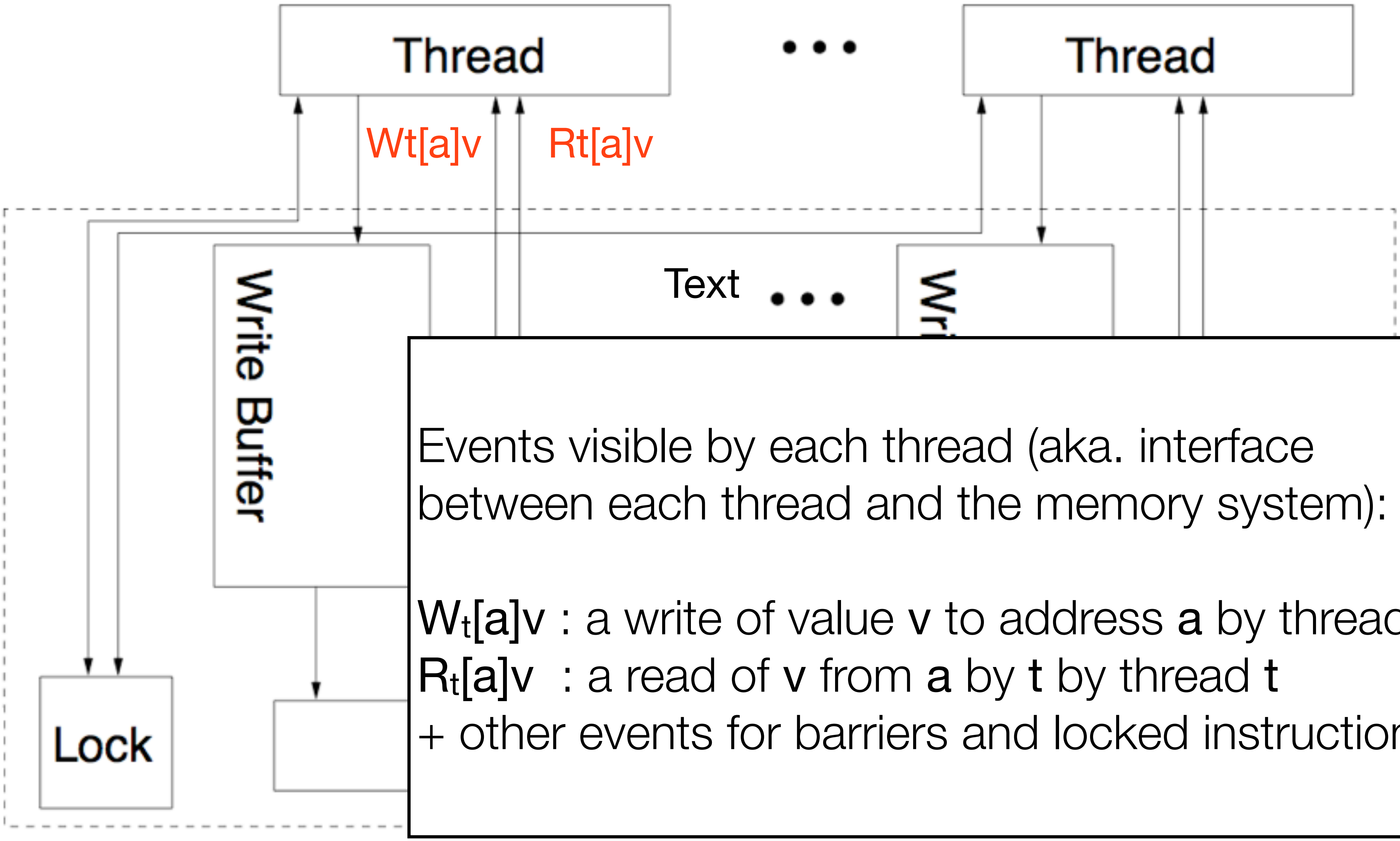
All threads read and write the shared memory. Threads execute asynchronously,the semantics allows any interleaving of the thread transitions.

$$\begin{tikzcd}
 & \langle t_1{:}{*x} = 1 | t_2{:}{*x} = 2,\ \{x \mapsto 0\}\rangle \arrow[dl, "\mathsf{W}_{t_1}\ x{=}1"'] \arrow[dr, "\mathsf{W}_{t_2}\ x{=}2"] & \\
\langle t_1{:}1 | t_2{:}{*x} = 2,\ \{x \mapsto 1\}\rangle \arrow[d, "\mathsf{W}_{t_2}\ x{=}2"'] & & \langle t_1{:}{*x} = 1 | t_2{:}2,\ \{x \mapsto 2\}\rangle \arrow[d, "\mathsf{W}_{t_1}\ x{=}1"] \\
\langle t_1{:}1 | t_2{:}2,\ \{x \mapsto 2\}\rangle & & \langle t_1{:}1 | t_2{:}2,\ \{x \mapsto 1\}\rangle
\end{tikzcd}$$

Each interleaving has a linear order of reads and writes to memory.

...now we just have to define a TSO memory...

x86-TSO abstract machine



Events visible by each thread (aka. interface between each thread and the memory system):

W_t[a]v : a write of value **v** to address **a** by thread **t**
R_t[a]v : a read of **v** from **a** by **t** by thread **t**
+ other events for barriers and locked instructions

x86-tso: a formalisation using an LTS

The machine state `s` can be represented by a tuple `(M,B,L)`:

```
M : address -> value option
B : tid -> (address * value) list
L : tid option
```

where:

`M` is the shared memory, mapping addresses to values

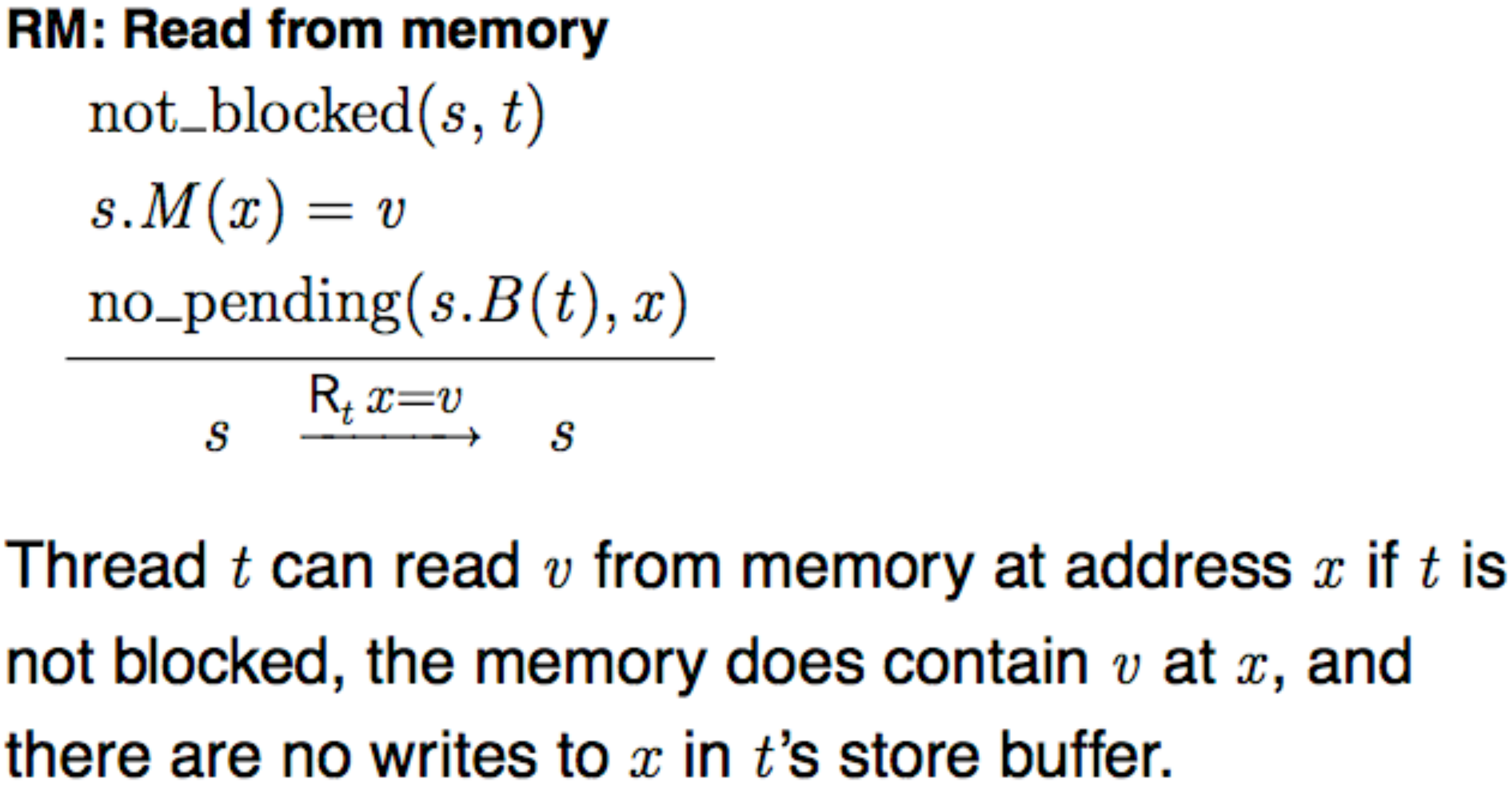
`B` gives the store buffer for each thread

`L` is the global machine lock indicating when a thread has exclusive access to memory (omitted in these slides)

x86-tso abstract machine: selected transition rules

t is *not blocked* in machine state s = (M,B,L) if [… or] the lock is not held.

In buffer B(t) there are *no pending writes* for address x if there are no (x,v) elements in B(t).

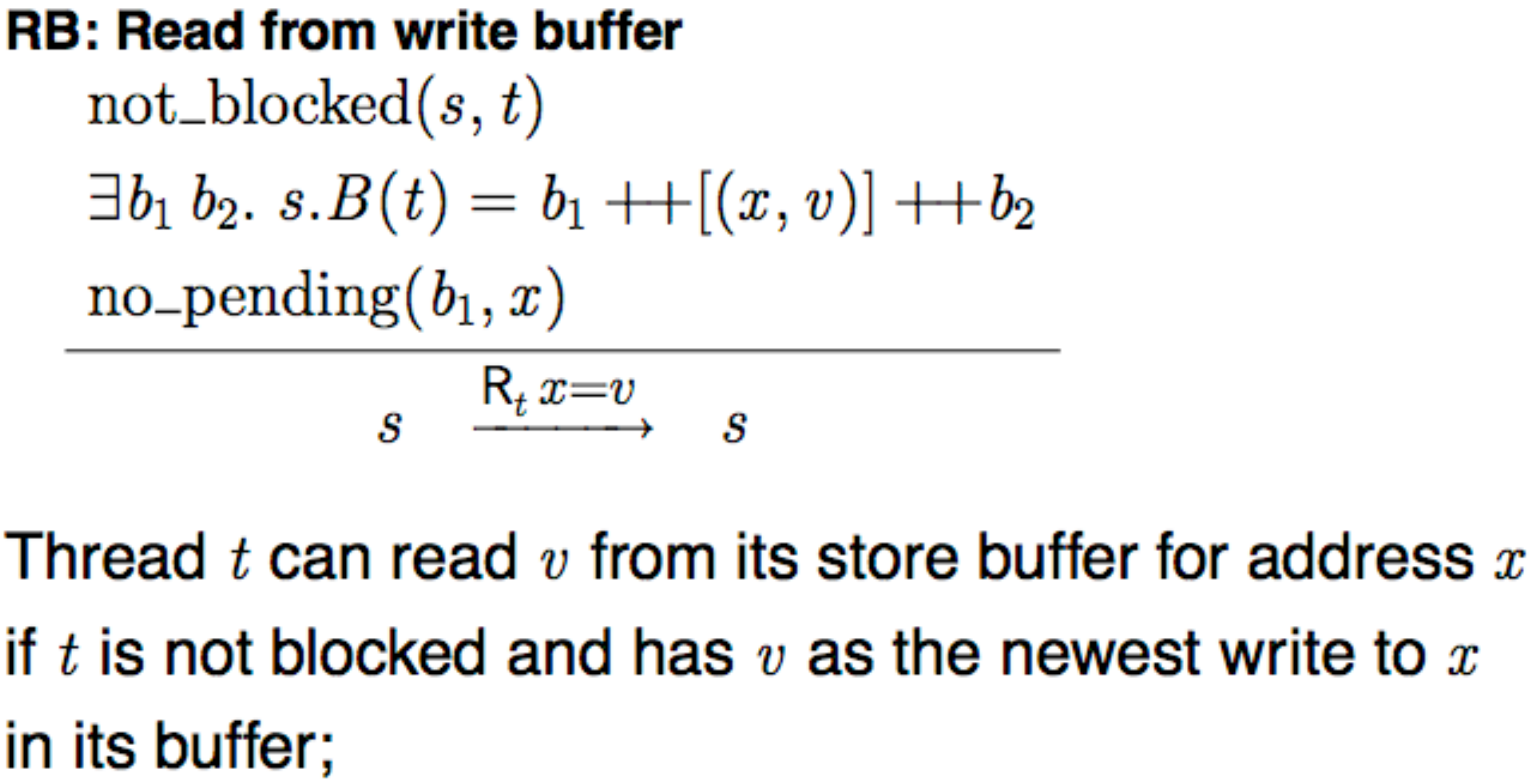


RM: Read from memory

$$\frac{\begin{array}{l}\text{not\_blocked}(s, t)\\ s.M(x) = v\\ \text{no\_pending}(s.B(t), x)\end{array}}{s \xrightarrow{\mathsf{R}_t\, x=v} s}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer.

x86-tso abstract machine: selected transition rules

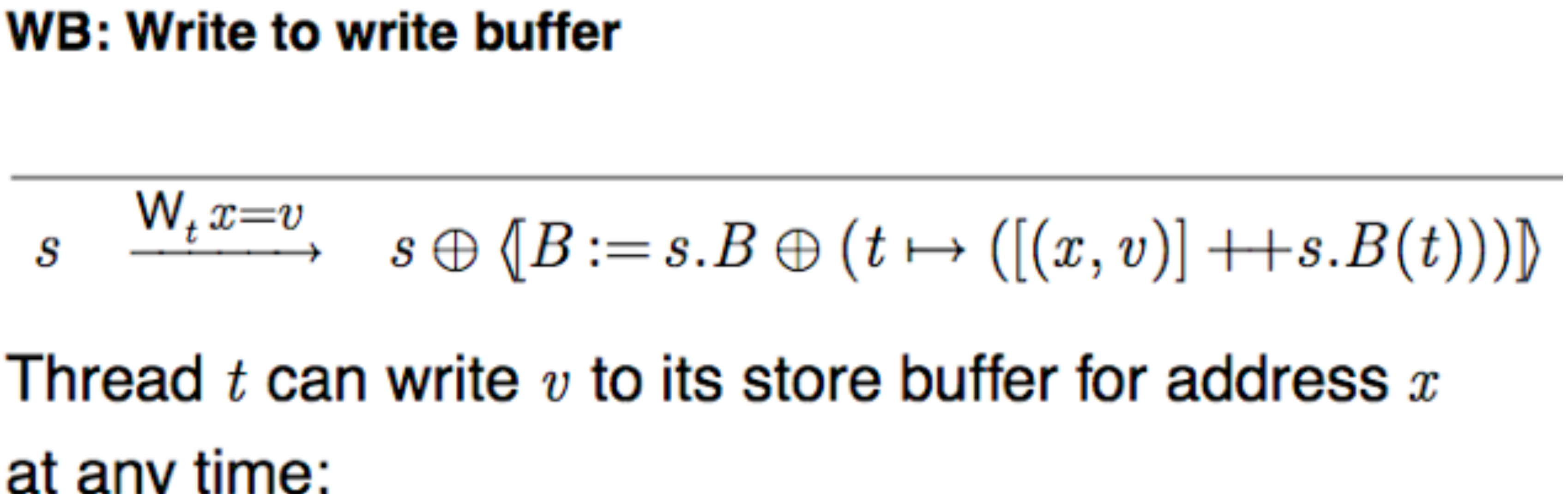


RB: Read from write buffer

$$\frac{\begin{array}{l}\text{not\_blocked}(s, t)\\ \exists b_1\, b_2.\ s.B(t) = b_1 +\!\!+ [(x, v)] +\!\!+ b_2\\ \text{no\_pending}(b_1, x)\end{array}}{s \xrightarrow{\mathsf{R}_t\, x=v} s}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the newest write to x in its buffer;

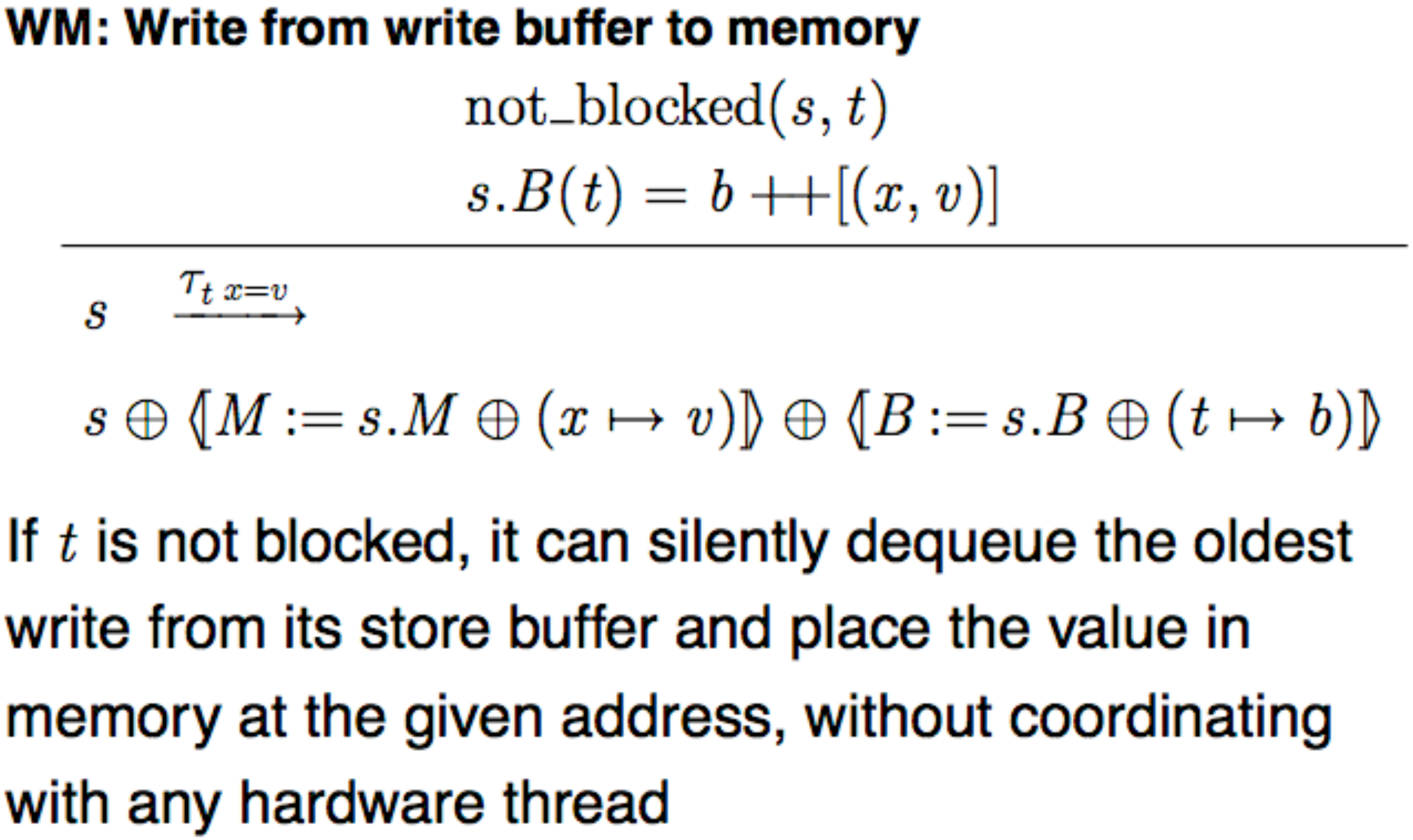
x86-tso abstract machine: selected transition rules



WB: Write to write buffer

$$\frac{}{s \xrightarrow{\mathsf{W}_t\, x=v} s \oplus \langle\!| B := s.B \oplus (t \mapsto ([(x,v)] +\!\!+ s.B(t))) |\!\rangle}$$

Thread t can write v to its store buffer for address x at any time;



WM: Write from write buffer to memory

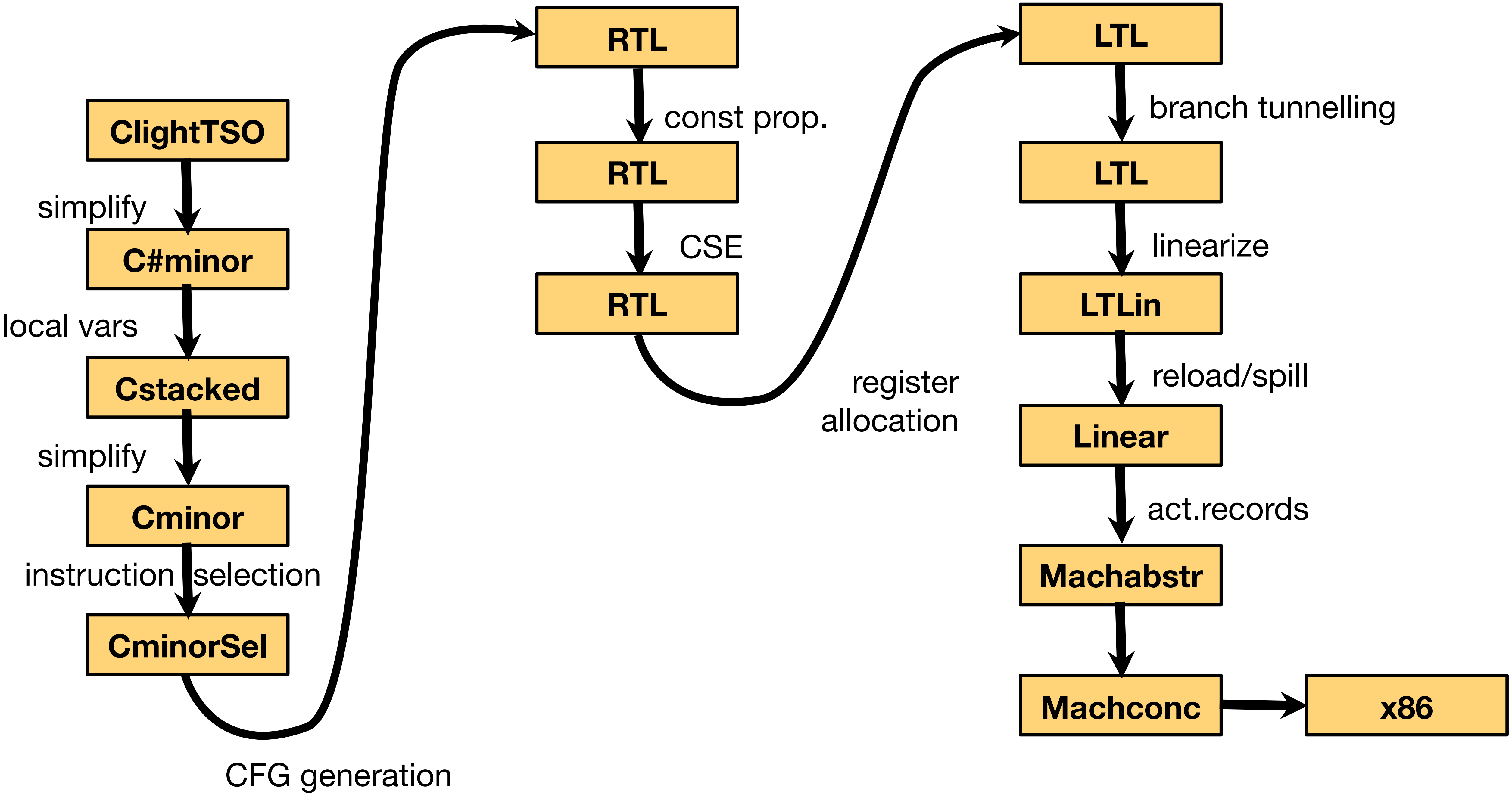
$$\frac{\begin{array}{l}\mathrm{not\_blocked}(s,t)\\ s.B(t) = b +\!\!+ [(x,v)]\end{array}}{\begin{array}{l} s \xrightarrow{\tau_t\, x=v} \\ s \oplus \langle\!| M := s.M \oplus (x \mapsto v) |\!\rangle \oplus \langle\!| B := s.B \oplus (t \mapsto b) |\!\rangle\end{array}}$$

If t is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread

4. Veryfing fence elimination optimisations

aka reasoning on the x86TSO operational memory model
and compiler correctness

CompCertTSO



ClightTSO

simplify

C#minor

local vars

Cstacked

simplify

Cminor

instruction selection

CminorSel

CFG generation

RTL

const prop.

RTL

CSE

RTL

register allocation

LTL

branch tunnelling

LTL

linearize

LTLin

reload/spill

Linear

act.records

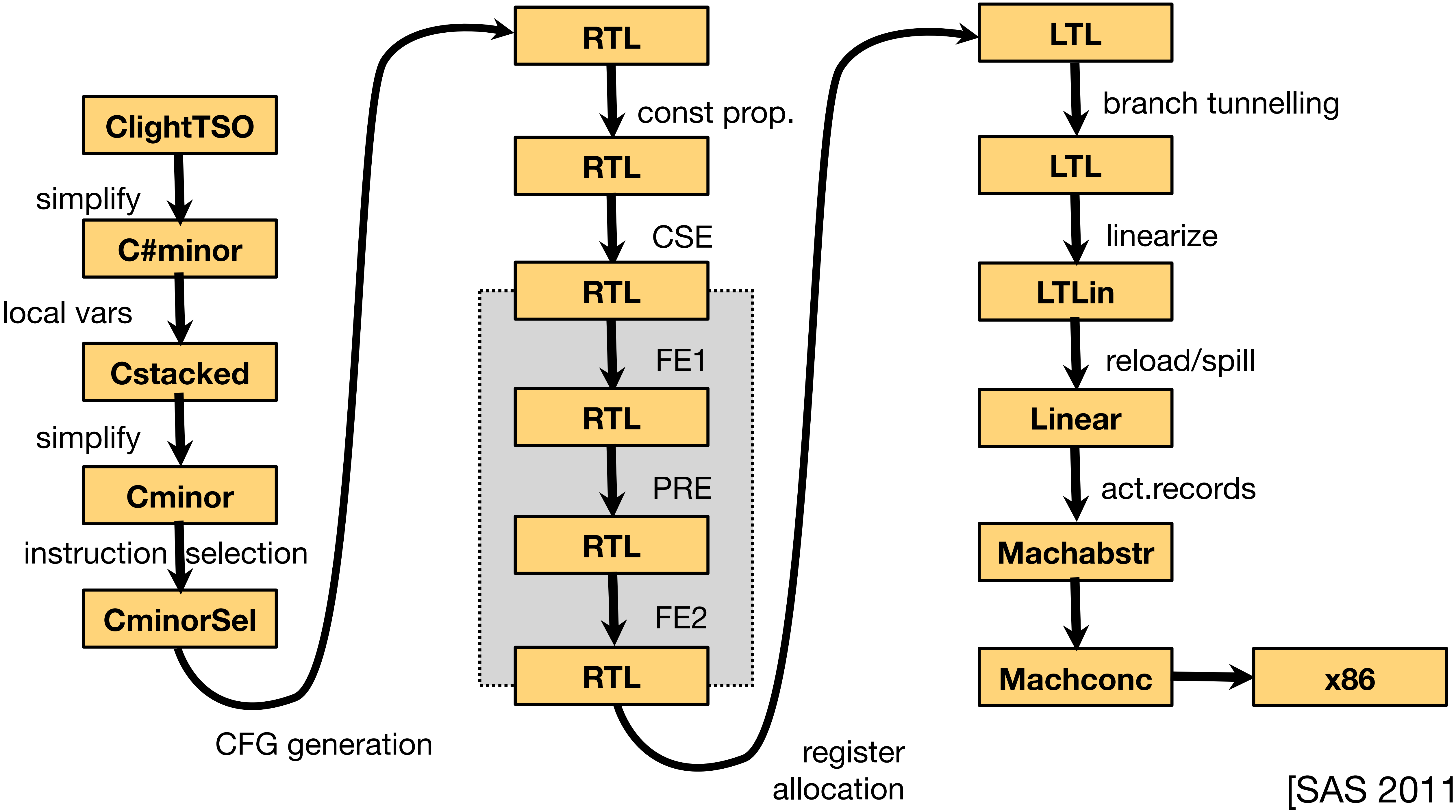
Machabstr

Machconc

x86

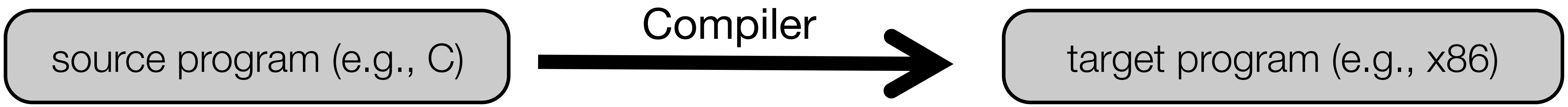
[POPL 2011]

CompCertTSO + fence optimisations



[SAS 2011]

Compilers are *ideal* for verification



source program (e.g., C)

Compiler

target program (e.g., x86)

Compilers are:

— Basic computing infrastructure

— Generally reliable, but nevertheless contain many bugs
e.g., Yang et al. [PLDI 2011] found 79 `gcc` & 202 `llvm` bugs

— “Specifiable”: compiler correctness = preservation of behaviours

— Interesting: naturally higher-order, involve clever algorithms

— Big, but modular

Language semantics

The semantics of all the CompCertTSO languages is defined by:

– a type of programs, prg

– a type of states, states

– a set of initial states for each program, $\mathsf{init} \in \mathit{prg} \to \mathbb{P}(\mathit{states})$

– a transition relation, $\to \in \mathbb{P}(\mathit{states} \times \boxed{\mathit{event}} \times \mathit{states})$

`call`, `return`, `fail`, `oom`, τ

The visible behaviour of a program is defined by the external function calls (`call`) and returns (`return`), errors (`fail`), and running out of memory (`oom`).

Traces

– *Finite sequences* of call & return events ending with:

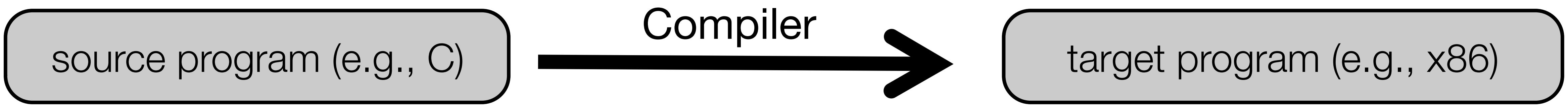
end: successful termination,
inftau: infinite execution that stops performing visible events
oom: execution runs out of memory

– *Infinite sequences* of call & return events;

$$
\begin{aligned}
\mathsf{traces}(p) \overset{\text{def}}{=}\; & \{\ell \cdot \mathsf{end} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s' \wedge s' \not\rightarrow\} \\
& \cup \{\ell \cdot tr \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \xRightarrow{\ell\cdot\mathtt{fail}} s'\} \\
& \cup \{\ell \cdot \mathsf{inftau} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s' \wedge \mathsf{inftau}(s')\} \\
& \cup \{\ell \cdot \mathsf{oom} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s'\} \\
& \cup \{tr \mid \exists s \in \mathsf{init}(p).\ s \text{ can do the infinite trace } tr\}
\end{aligned}
$$

NB: Erroneous computations become undefined after the first error.

Compiler correctness

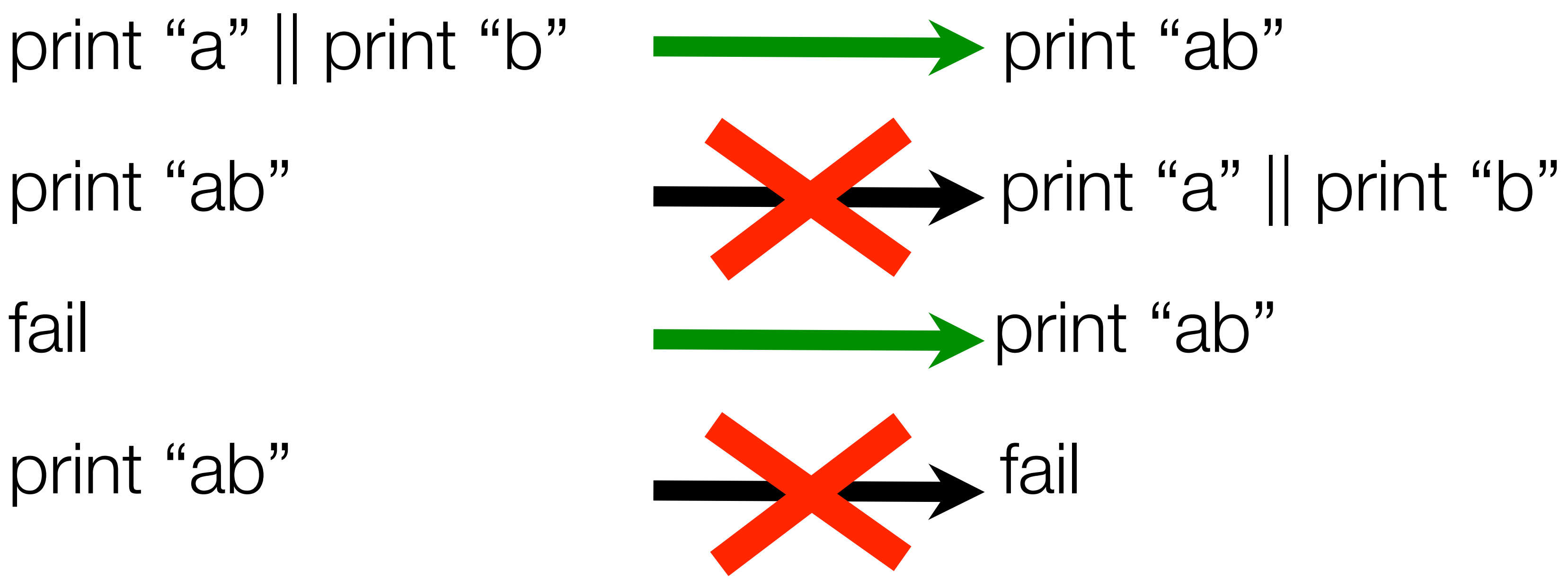


source program (e.g., C)

Compiler

target program (e.g., x86)

$$\text{traces(source\_program)} \supseteq \text{traces(target\_program)}$$



print “a” || print “b” → print “ab”

print “ab” → print “a” || print “b”

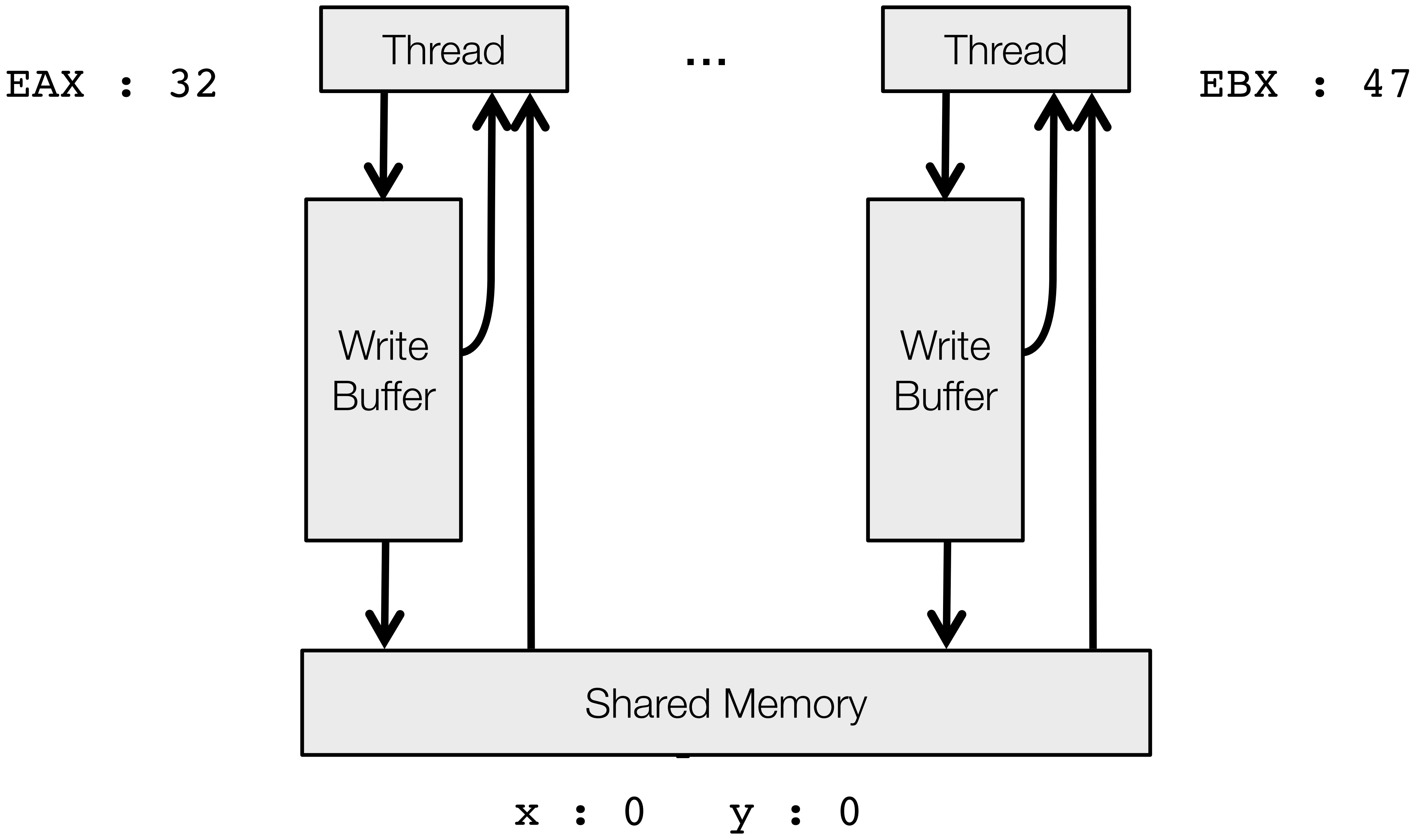
fail → print “ab”

print “ab” → fail

Store buffering

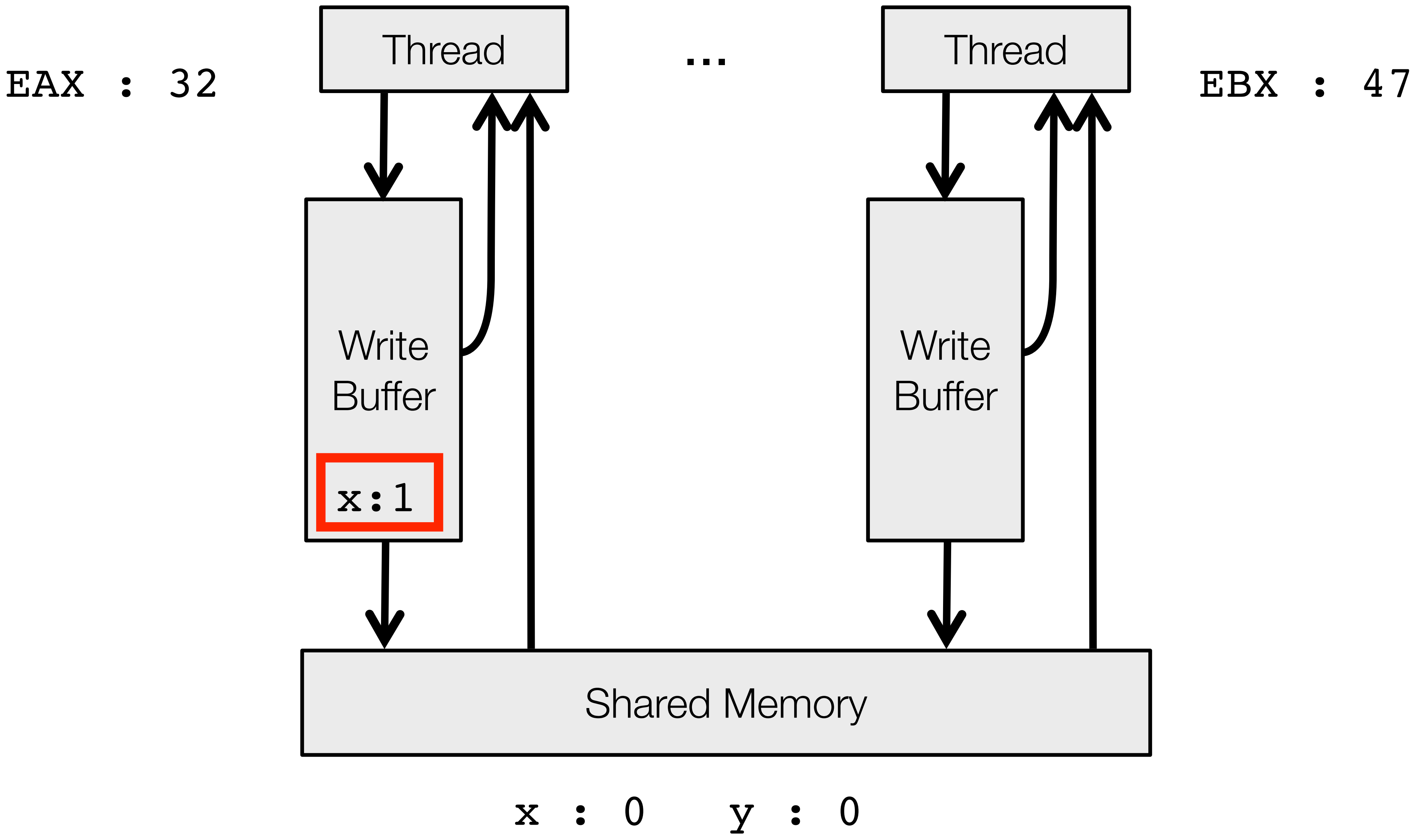
```
MOV [x] ← 1
MOV EAX ← [y]
```

```
MOV [y] ← 1
MOV EBX ← [x]
```



Store buffering

```
MOV [x] ← 1        MOV [y] ← 1
MOV EAX ← [y]      MOV EBX ← [x]
```



EAX : 32

Thread

...

Thread

EBX : 47

Write Buffer

x:1

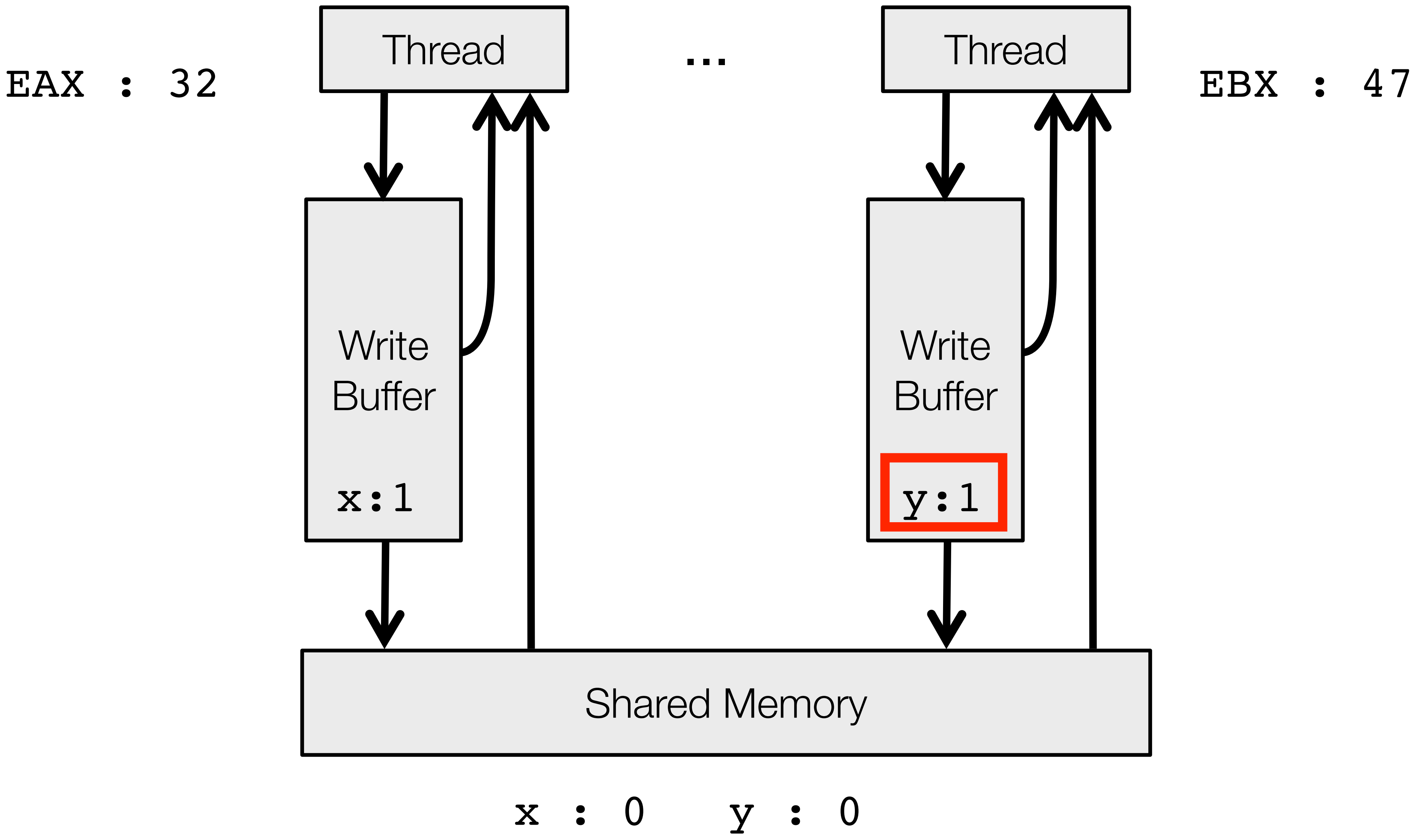
Write Buffer

Shared Memory

x : 0   y : 0

Store buffering

```
MOV [x] ← 1          MOV [y] ← 1
MOV EAX ← [y]        MOV EBX ← [x]
```



EAX : 32

Thread

...

Thread

EBX : 47

Write Buffer

x:1

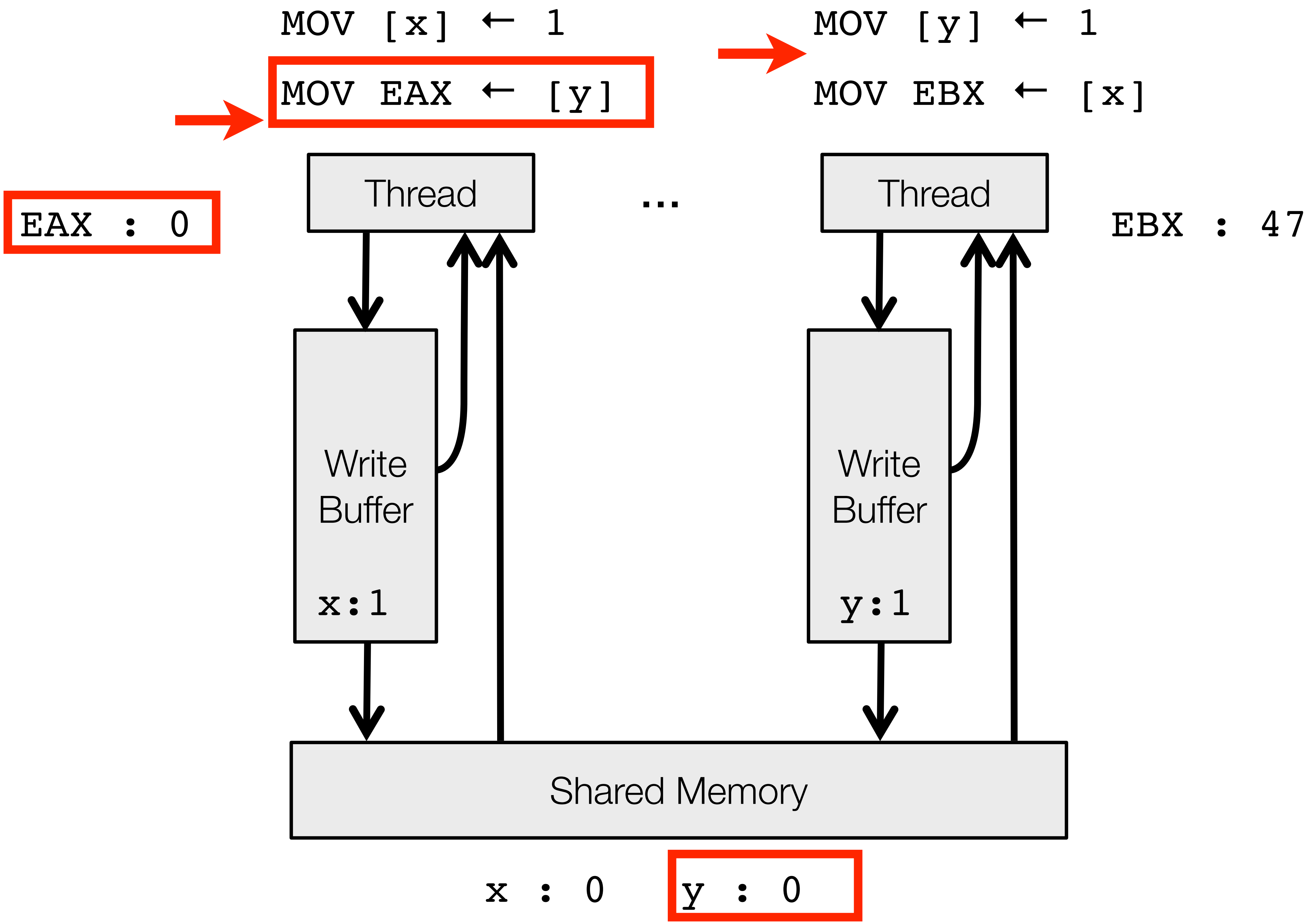
Write Buffer

y:1

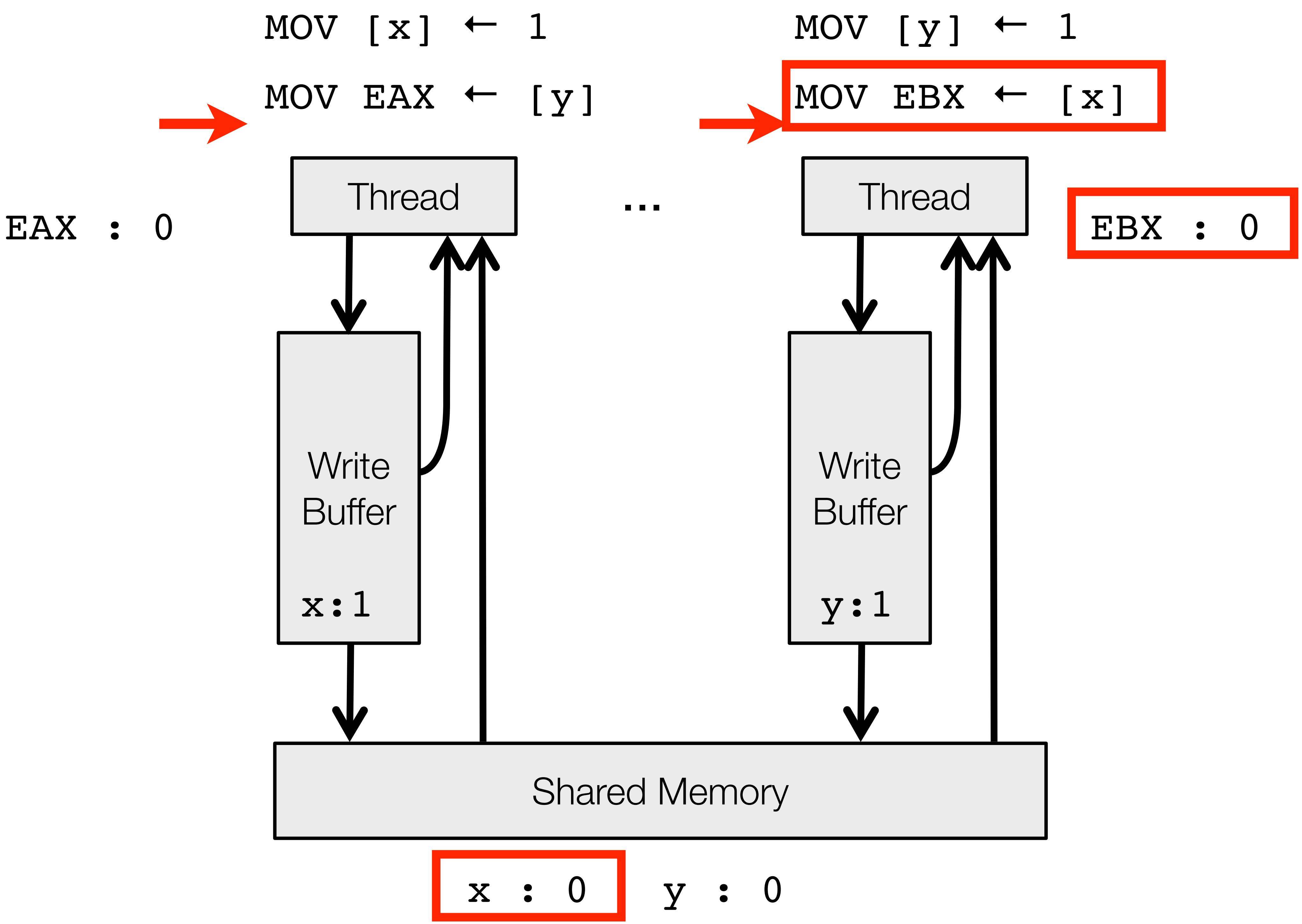
Shared Memory

x : 0   y : 0

Store buffering

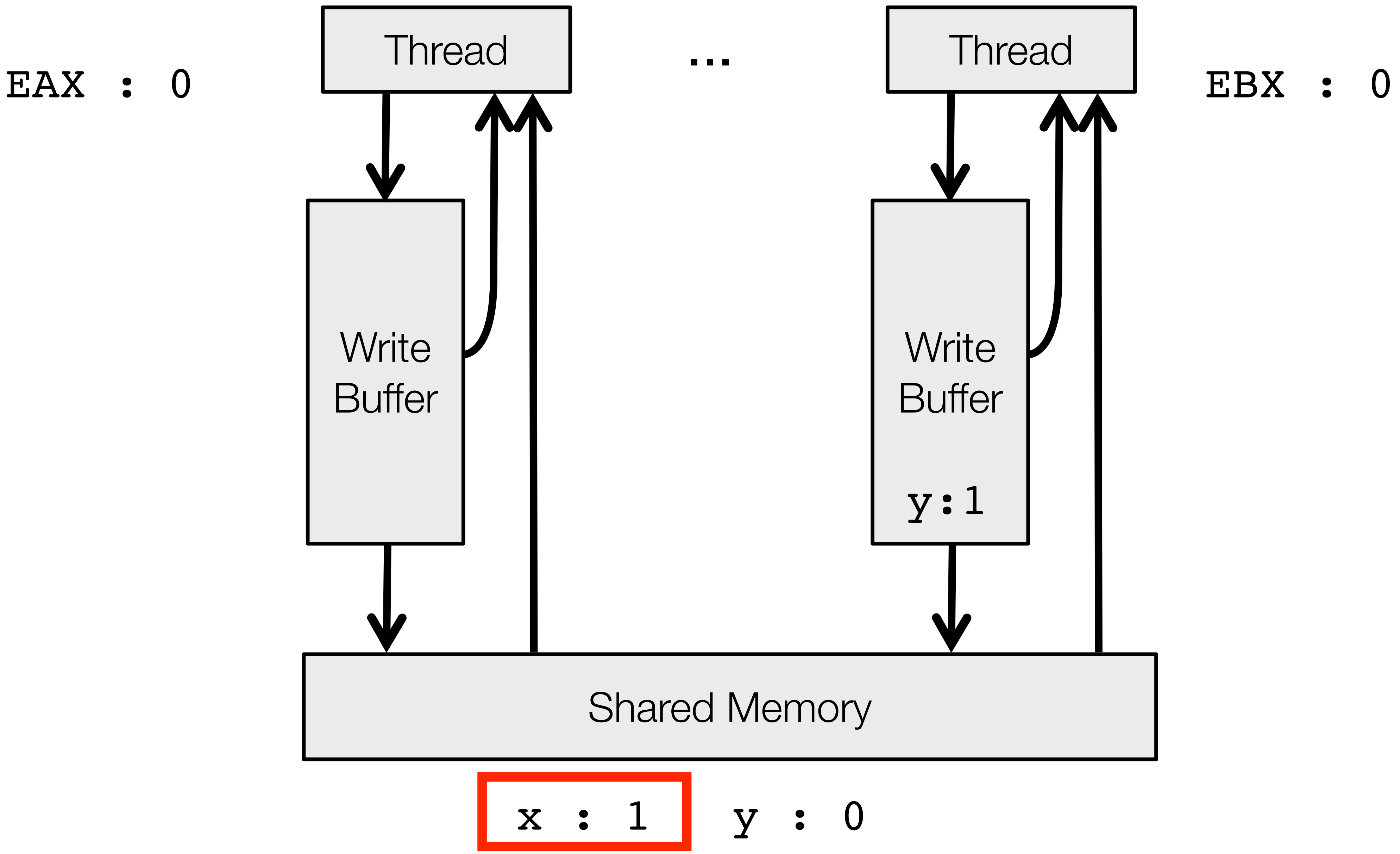


Store buffering



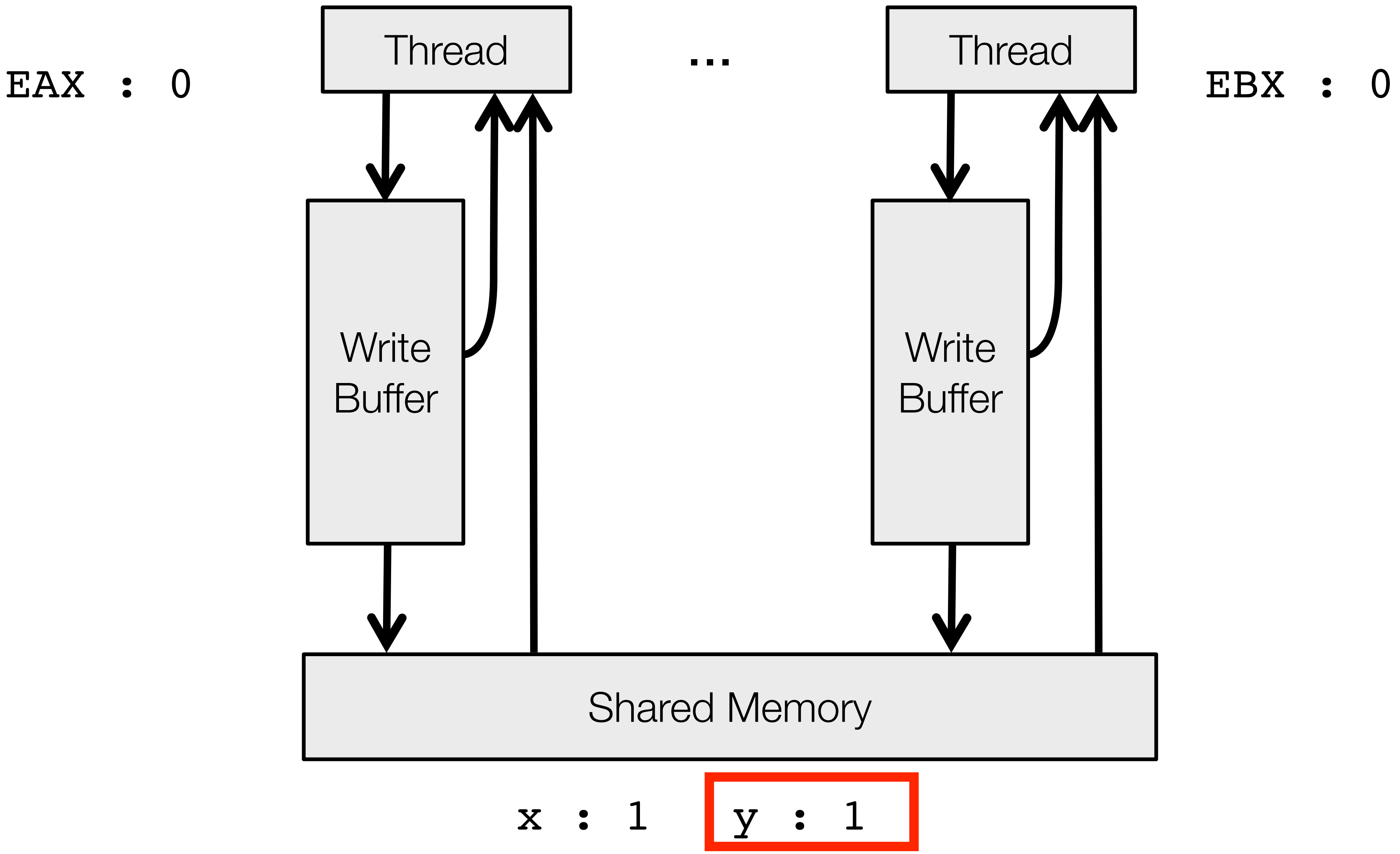
Store buffering

```
MOV [x] ← 1              MOV [y] ← 1
MOV EAX ← [y]            MOV EBX ← [x]
```



Store buffering

```
MOV [x] ← 1          MOV [y] ← 1
MOV EAX ← [y]        MOV EBX ← [x]
```



EAX : 0

Thread

...

Thread

EBX : 0

Write Buffer

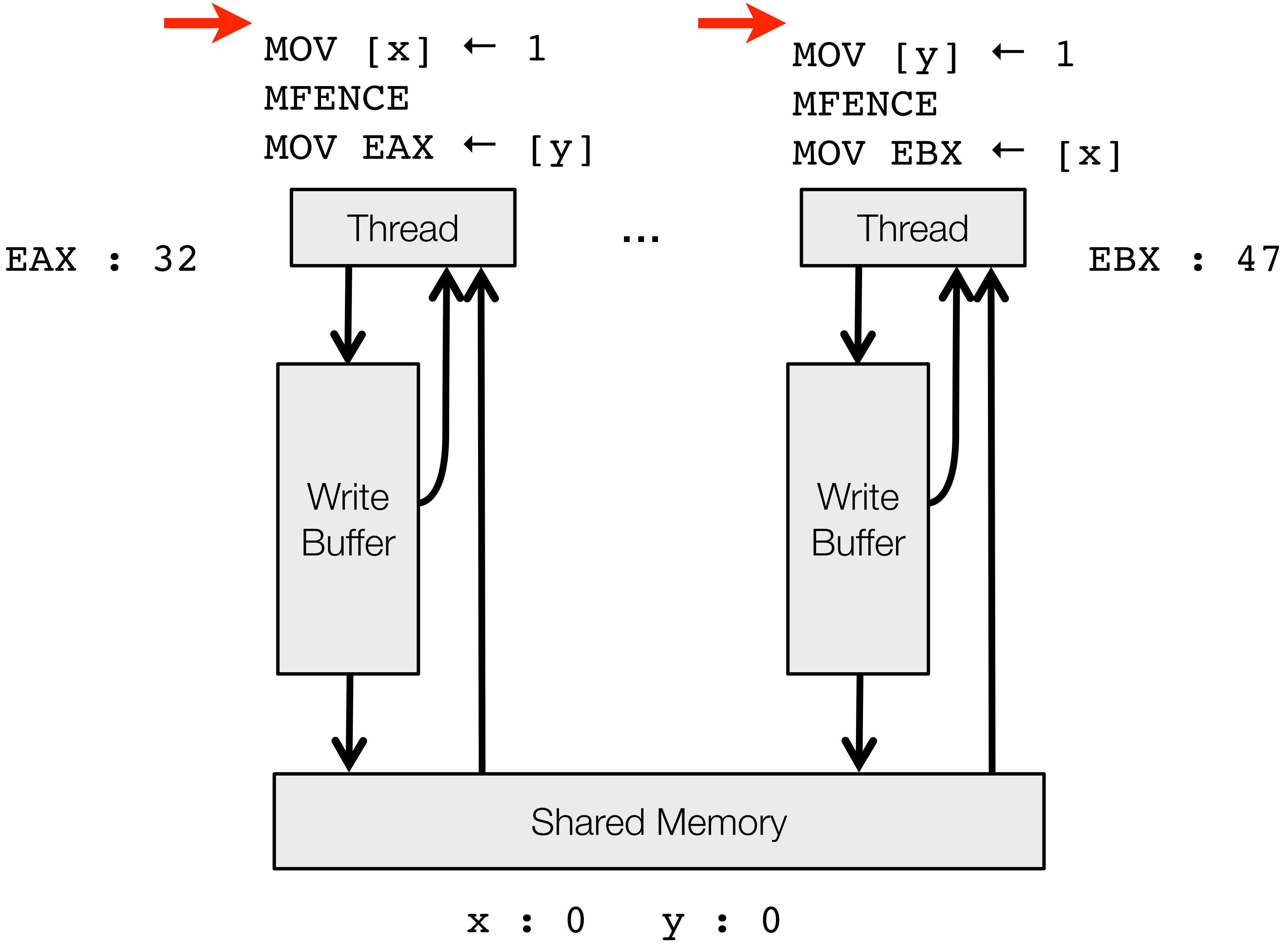
Write Buffer

Shared Memory

x : 1

y : 1

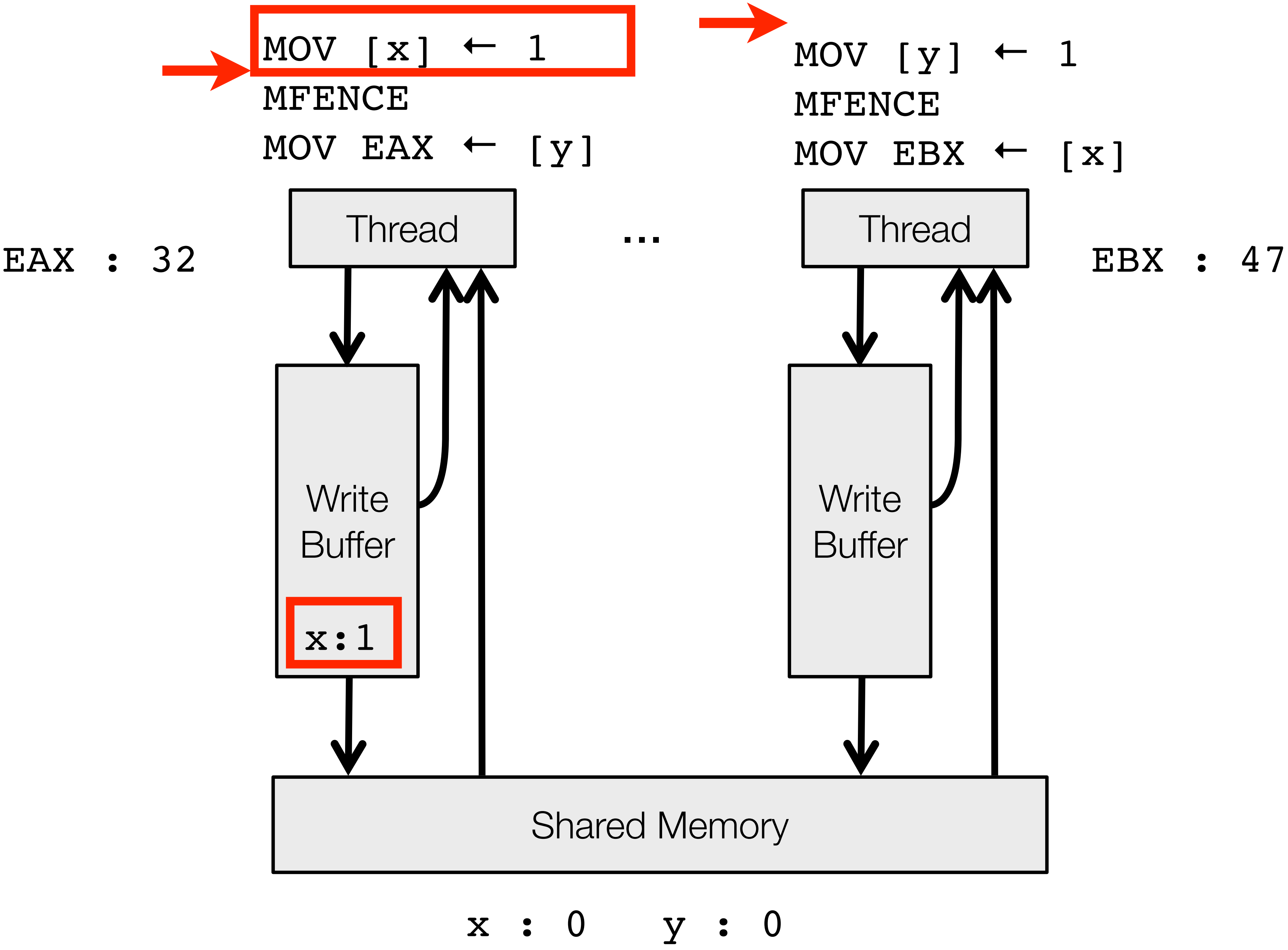
Store buffering + fences



```
MOV [x] ← 1
MFENCE
MOV EAX ← [y]
```

```
MOV [y] ← 1
MFENCE
MOV EBX ← [x]
```

Store buffering + fences



```
MOV [x] ← 1
MFENCE
MOV EAX ← [y]
```

```
MOV [y] ← 1
MFENCE
MOV EBX ← [x]
```

EAX : 32

Thread

...

Thread

EBX : 47

Write Buffer

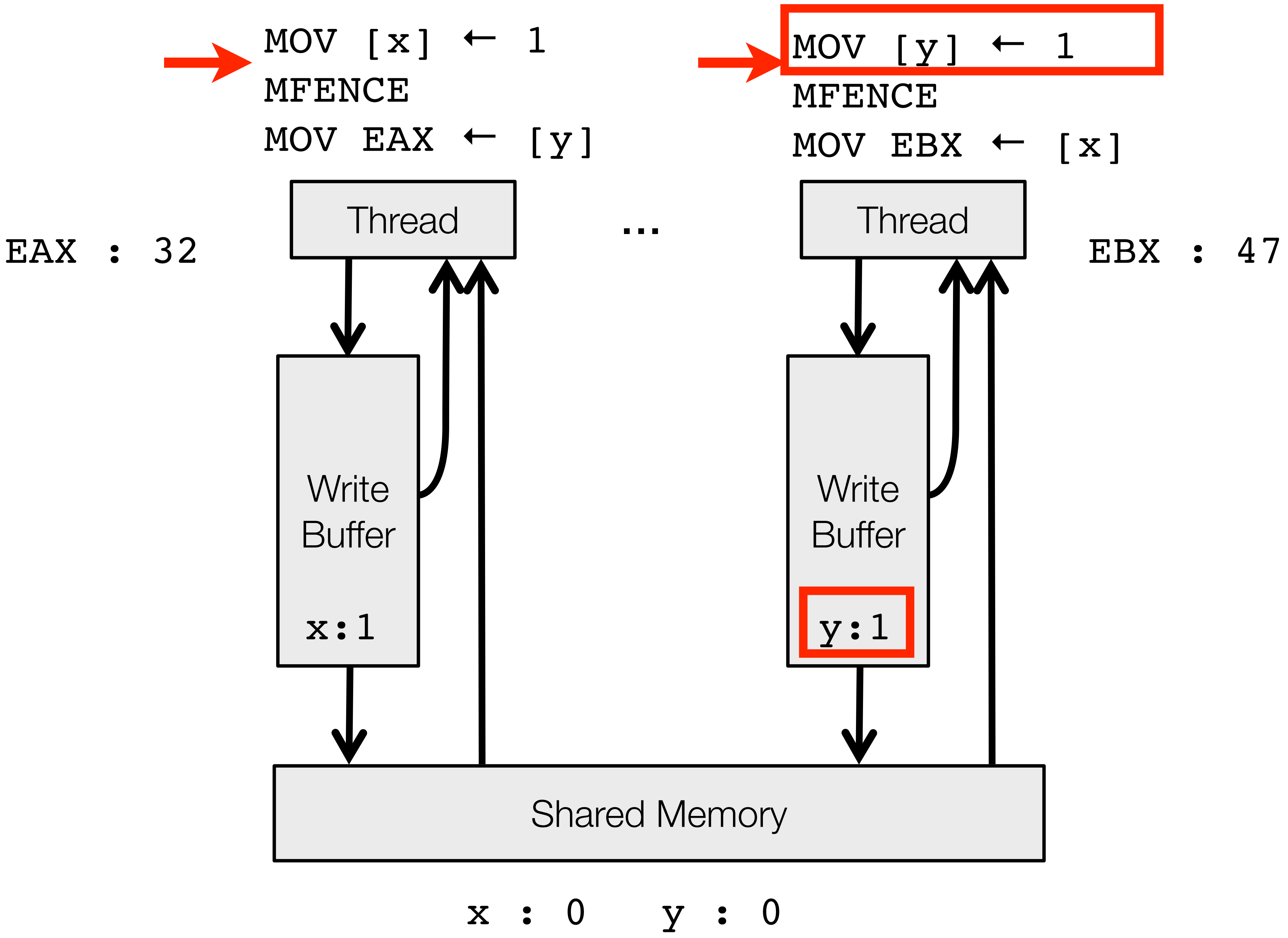
x:1

Write Buffer

Shared Memory

x : 0   y : 0

Store buffering + fences

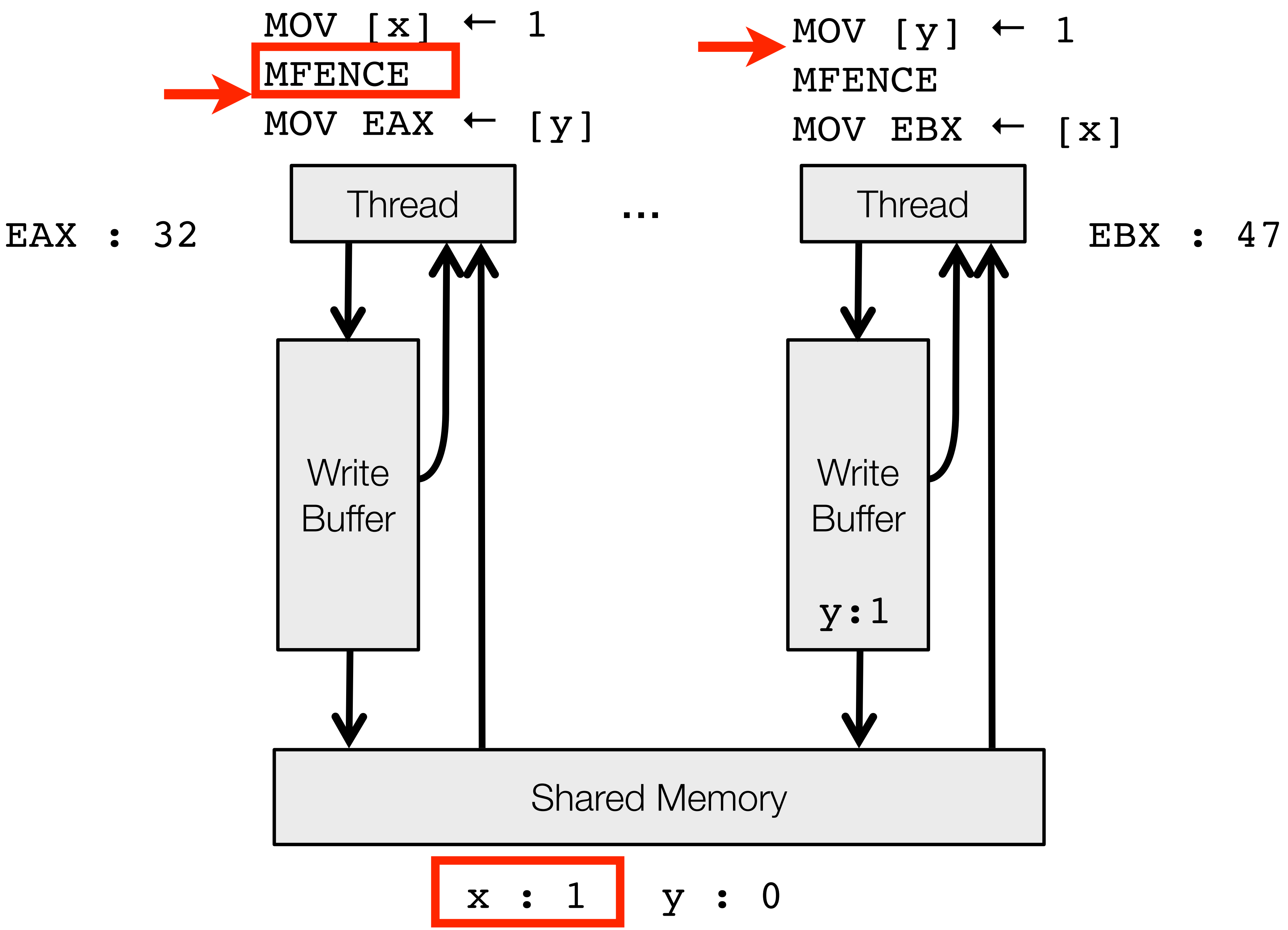


```
MOV [x] ← 1
MFENCE
MOV EAX ← [y]
```

```
MOV [y] ← 1
MFENCE
MOV EBX ← [x]
```

Store buffering + fences

`MFENCE` blocks until the thread buffer is empty



```
MOV [x] ← 1
MFENCE
MOV EAX ← [y]
```

```
MOV [y] ← 1
MFENCE
MOV EBX ← [x]
```

Thread

...

Thread

EAX : 32

EBX : 47

Write Buffer

Write Buffer

y:1

Shared Memory

x : 1

y : 0

Who inserts fences?

1. The *programmer*, explicitly. Example: Fraser's lockfree-lib:

```
/*
 * II. Memory barriers.
 *  MB():  All preceding memory accesses must commit before any later accesses.
 *
 *  If the compiler does not observe these barriers (but any sane compiler
 *  will!), then VOLATILE should be defined as 'volatile'.
 */
#define MB()  __asm__ __volatile__ ("lock; addl $0,0(%%esp)" : : : "memory")
```

2. The *compiler*, to implement a high-level memory model, e.g. `SEQ_CST` C++0x low-level atomics on x86:

Load SEQ_CST: `MFENCE; MOV`

Store SEQ_CST: `MOV; MFENCE`

Fence instructions

1. *Fences are necessary*

   to implement locks & not fully-commutative linearizable objects (e.g., stacks, queues, sets, maps).

   [Attiya et al., POPL 2011]

2. *Fences can be expensive*

Redundant fences (1)

If we have two consecutive fence instructions, we can remove the *latter*:

```
MFENCE             MFENCE
MFENCE     →       NOP
```

The *buffer is already empty* when the second fence is executed.

Generalisation:

```
MFENCE                                MFENCE
NON-WRITE INSTR                       NON-WRITE INSTR
…                          →          …
NON-WRITE INSTR                       NON-WRITE INSTR
MFENCE                                NOP
```

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A *forward* data-flow problem over the boolean domain $\{\bot, \top\}$

Associate to each program point:

$\bot$: along all execution paths there is an atomic instruction *before* the current program point, with no intervening writes;

$\top$: otherwise.

$$\begin{aligned}
T_1(\mathbf{nop}, \mathcal{E}) &= \mathcal{E} \\
T_1(\mathbf{op}(op, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_1(\mathtt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_1(\mathtt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) &= \top \\
T_1(\mathtt{call}(sig, ros, args, res), \mathcal{E}) &= \top \\
T_1(\mathtt{cond}(cond, args), \mathcal{E}) &= \mathcal{E} \\
T_1(\mathtt{return}(optarg), \mathcal{E}) &= \top \\
T_1(\mathtt{threadcreate}(optarg), \mathcal{E}) &= \top \\
T_1(\mathtt{atomic}(aop, \vec{r}, r), \mathcal{E}) &= \bot \\
T_1(\mathtt{fence}, \mathcal{E}) &= \bot
\end{aligned}$$

$$\mathcal{FE}_1(n) = \begin{cases} \top & \text{if predecessors}(n) = \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(instr(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A *forward* data-flow problem over the bo

Associ

⊥ : alo
is a
cur
no

⊤ : oth

$T_1(\mathbf{nop}, \mathcal{E}) = \mathcal{E}$

$= \mathcal{E}$

$= \mathcal{E}$

$= \top$

$) = \top$

$= \mathcal{E}$

$= \top$

$) = \top$

$= \bot$

$= \bot$

Implementation:

1. Use CompCert implementation of Kildall algorithm to solve the data-flow equations.
2. Replace `MFENCE`s for which the analysis returns ⊥ with `NOP` instructions.

$$\mathcal{FE}_1(n) = \begin{cases} \quad \emptyset \\ \bigsqcup_{p \in \mathrm{predecessors}(n)} T_1(\mathit{instr}(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

Redundant fences (2)

If we have two consecutive fence instructions, we can remove the *former*:

```
MFENCE                    NOP
MFENCE        →           MFENCE
```

Intuition: the visible effects initially published by the former fence, are now published by the latter, and nobody can tell the difference.

Generalisation:

```
MFENCE                         NOP
INSTRUCTION 1       ???        INSTRUCTION 1
…                    →         …
INSTRUCTION n                  INSTRUCTION n
MFENCE                         MFENCE
```

Redundant fences (2)

If there are reads in between the fences…

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **MFENCE** MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
forbidden

but

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **NOP** MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
allowed

Redundant fences (2)

If there are reads in between the fences…

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **MFENCE** MOV EAX ← [y] MF	MOV [y] ← 1 MFENCE

EAX = EBX = 0
forbidden

If there are reads in between, the optimisation is unsound.

but

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **NOP** MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
allowed

Redundant fences (2)

Swapping a `STORE` and a `MFENCE` is sound:

`MFENCE; STORE` → `STORE; MFENCE`

1. transformed program’s behaviours ⊆ source program’s behaviours

(source program might leave pending write in its buffer)

2. There is the new intermediate state if the buffer was initially non-empty, but this intermediate state *is not observable.*

(a local read is needed to access the local buffer)

Intuition: Iterate this swapping...

FE2

A fence is redundant if it always precedes a later fence or locked instruction in program order, and no memory read instructions are in between.

A *backward* data-flow problem over the boolean domain $\{\bot, \top\}$

Associate to each program point:

$\bot$: along all execution paths there is an atomic instruction *after* the current program point, with no intervening reads;

$\top$: otherwise.

$$
\begin{aligned}
T_2(\mathtt{nop}, \mathcal{E}) &= \mathcal{E} \\
T_2(\mathtt{op}(op, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_2(\mathtt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) &= \top \\
T_2(\mathtt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) &= \mathcal{E} \\
T_2(\mathtt{call}(sig, ros, args, res), \mathcal{E}) &= \top \\
T_2(\mathtt{cond}(cond, args), \mathcal{E}) &= \mathcal{E} \\
T_2(\mathtt{return}(optarg), \mathcal{E}) &= \top \\
T_2(\mathtt{threadcreate}(optarg), \mathcal{E}) &= \top \\
T_2(\mathtt{atomic}(aop, \vec{r}, r), \mathcal{E}) &= \bot \\
T_2(\mathtt{fence}, \mathcal{E}) &= \bot
\end{aligned}
$$

$$
\mathcal{FE}_2(n) = \begin{cases} \top & \text{if successors}(n) = \emptyset \\ \bigsqcup_{s \in \text{successors}(n)} T_2(instr(s), \mathcal{FE}_2(s)) & \text{otherwise} \end{cases}
$$

FE1 and FE2 are both useful

Removed by FE1 but not FE2:

```
MFENCE
MOV EAX <- [y]
MFENCE
MOV EBX <- [y]
```

Removed by FE2 but not FE1:

```
MOV [x] <- 1
MFENCE
MOV [x] <- 2
MFENCE
```

Informal correctness argument

Intuition: FE2 can be thought as iterating

```
MFENCE; STORE      →    STORE; MFENCE
MFENCE; non-mem    →    non-mem; MFENCE
```

and then applying

```
MFENCE; MFENCE     →    NOP; MFENCE
```

This argument works for *finite traces*, but not for *infinite traces* as the later fence might never be executed:

```
MFENCE;                 NOP;
STORE;         →        STORE;
WHILE(1);               WHILE(1);
MFENCE                  MFENCE
```

Basic simulations

A pair of relations

$\sim\in \mathbb{P}(src.states \times tgt.states)$ $\qquad$ $>\in \mathbb{P}(tgt.states \times tgt.states)$

is a *basic simulation* for $\textsf{compile} : src.prg \to tgt.prg$ if:

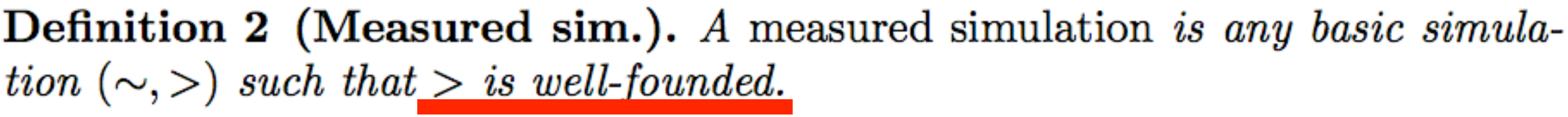
$$
\begin{aligned}
&sim\_init : \forall p\, p'.\ \textsf{compile}(p) = p' \implies \forall t \in \textsf{init}(p').\ \exists s \in \textsf{init}(p).\ s \sim t\\
&sim\_end : \forall s\, t.\ s \sim t \wedge t \not\to \_ \implies s \not\to \_\\
&sim\_step : \forall s\, t\, t'\, ev.\ s \sim t \wedge t \xrightarrow{ev} t' \wedge ev \neq \textsf{oom} \implies\\
&\qquad (s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) && \textit{— s reaches a failure}\\
&\qquad \vee (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \sim t') && \textit{— s does matching step sequence}\\
&\qquad \vee (ev = \tau \wedge t > t' \wedge s \sim t'). && \textit{— s stutters (only allowed if } t > t')
\end{aligned}
$$

Exhibiting a basic simulation implies:

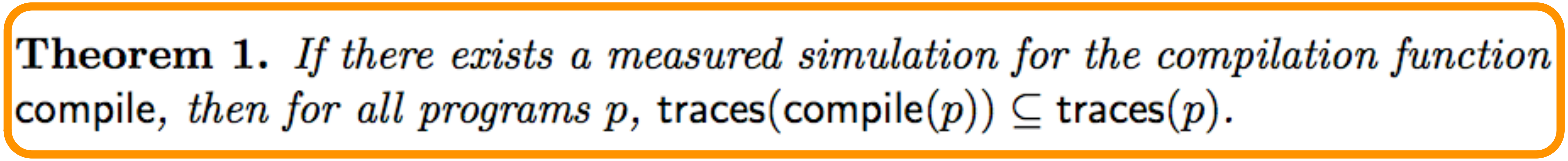
traces(**compile**(p)) \ {t · **inftau** | t trace} ⊆ **traces**(p)

“simulation can stutter forever”

Usual approach: measured simulations



Definition 2 (Measured sim.). *A* measured simulation *is any basic simulation $(\sim, >)$ such that $>$ is well-founded.*



Theorem 1. *If there exists a measured simulation for the compilation function* compile, *then for all programs p,* $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

Simulation for FE2

$s \equiv_i t$ iff thread *i* of *s* and *t* have identical pc, local states and buffers

$s \leadsto_i s'$ iff thread *i* of *s* can execute zero or more `NOP`, `OP`, `STORE` and `MFENCE` instructions and end in the state *s*'

$s \sim t$ iff

– *t*'s CFG is the optimised version of s's CFG; and

– *s* and *t* have identical memories; and

– ∀ thread *i*, either $s \equiv_i t$ or

the analysis for *i*'s pc returned ⊥ and $\exists s'$, $s \leadsto_i s'$ and $s' \equiv_i t$

"*s* is some instructions behind and can catch up"

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE`

(and *t*'s buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread *i* of *s* and *t* have identical pc, local states and buffers

$s \rightsquigarrow_i s'$ iff th

MFE

$s \sim t$ iff

– *t*’s CFG

– *s* and *t*

– ∀ threa

But if (1) all threads have non-empty buffers, and
(2) are stuck executing infinite loops, and
(3) no writes are ever propagated to memory,
then we can stutter forever.

(i.e., > is not well-founded.)

the analysis for *i*’s pc returned ⊥ and ∃*s*', $s \rightsquigarrow_i s'$ and $s' \equiv_i t$

“*s* is some instructions behind and can catch up”

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE`
(and *t*’s buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \rightsquigarrow_i s'$ iff th

MFEI

$s \sim t$ iff

– t's CFG

– s and t I

– ∀ threac

t $\equiv_i t$

Stutter conditic

$t > t'$ iff $t \rightarrow$

(anc

But if (1) all threads have non-empty buffers, and
(2) are stuck executing infinite loops, and
(3) no writes are ever propagated to memory,
then we can stutter forever.

(i.e

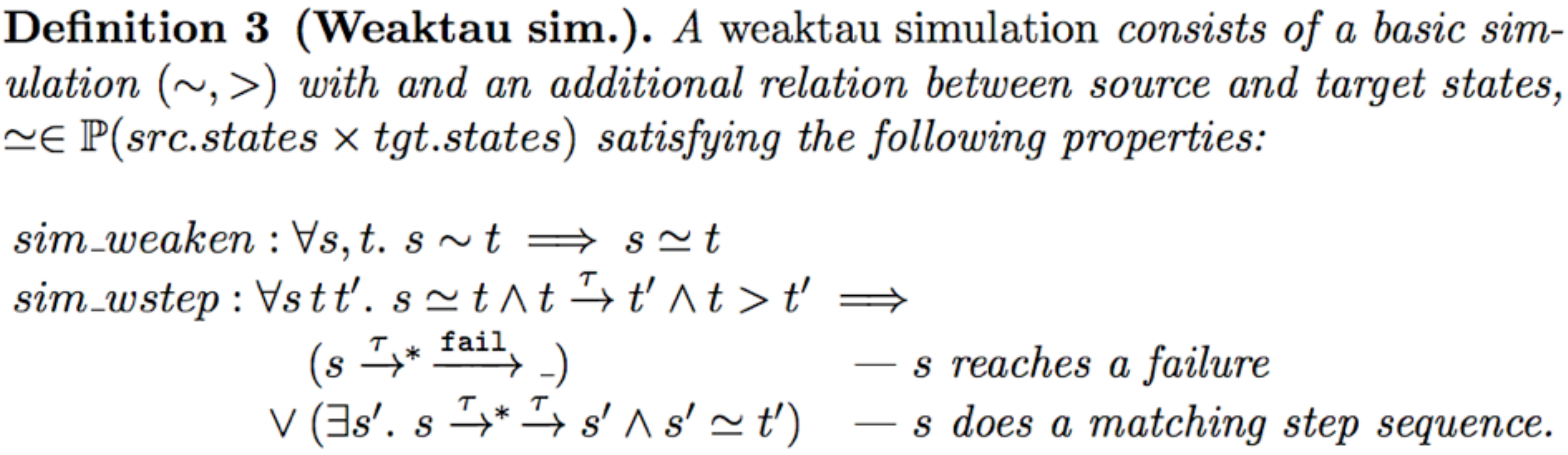
Solution 1: Assume this case never arises (*fairness*)

Solution 2: Do a case split.

— If this case does not arise, we are done.

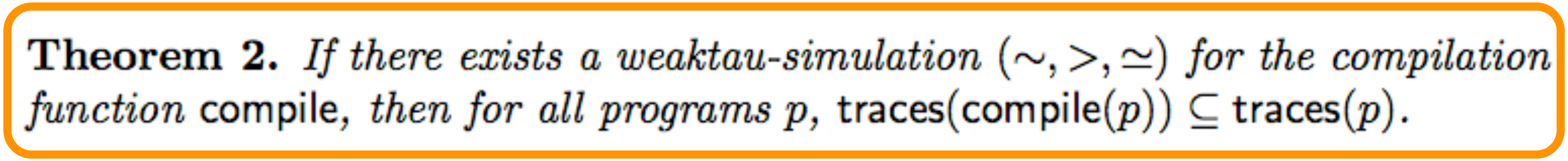
— If it does, use a different (weaker) simulation to construct an infinite trace for the source

Weaktau simulation



Definition 3 (Weaktau sim.). *A* weaktau simulation *consists of a basic simulation* $(\sim, >)$ *with and an additional relation between source and target states,* $\simeq \in \mathbb{P}(src.states \times tgt.states)$ *satisfying the following properties:*

$$\begin{array}{lll} sim\_weaken : \forall s,t.\ s \sim t \implies s \simeq t & & \\ sim\_wstep : \forall s\,t\,t'.\ s \simeq t \wedge t \xrightarrow{\tau} t' \wedge t > t' \implies & & \\ \qquad (s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) & & \text{— } \textit{s reaches a failure} \\ \qquad \vee\ (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s' \wedge s' \simeq t') & & \text{— } \textit{s does a matching step sequence.} \end{array}$$



Theorem 2. *If there exists a weaktau-simulation* $(\sim, >, \simeq)$ *for the compilation function* compile, *then for all programs* p, $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

Remarks:

— Once the simulation game moves from ~ to ≃, stuttering is forbidden;

— Can view difference between ~ and ≃ as a boolean prophecy variable.

Weaktau simulation for FE2

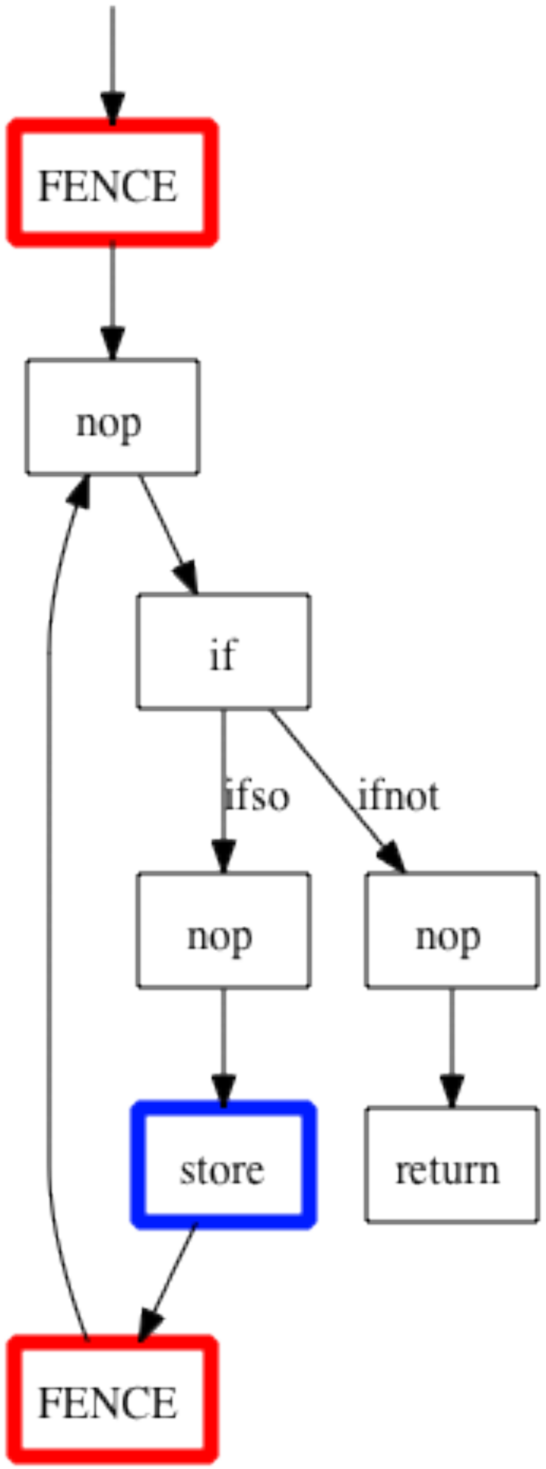
$s \sim t$, $t > t'$ as before.

$s \simeq t$ iff

- – t's CFG is the optimised version of s's CFG; and
- – $\forall i,\ \exists s'$ s.t. $s \leadsto_i s' \equiv_i t.$

(i.e., same as $s \sim t$ except that the memories memories are unrelated.)

A closer look at the RTL

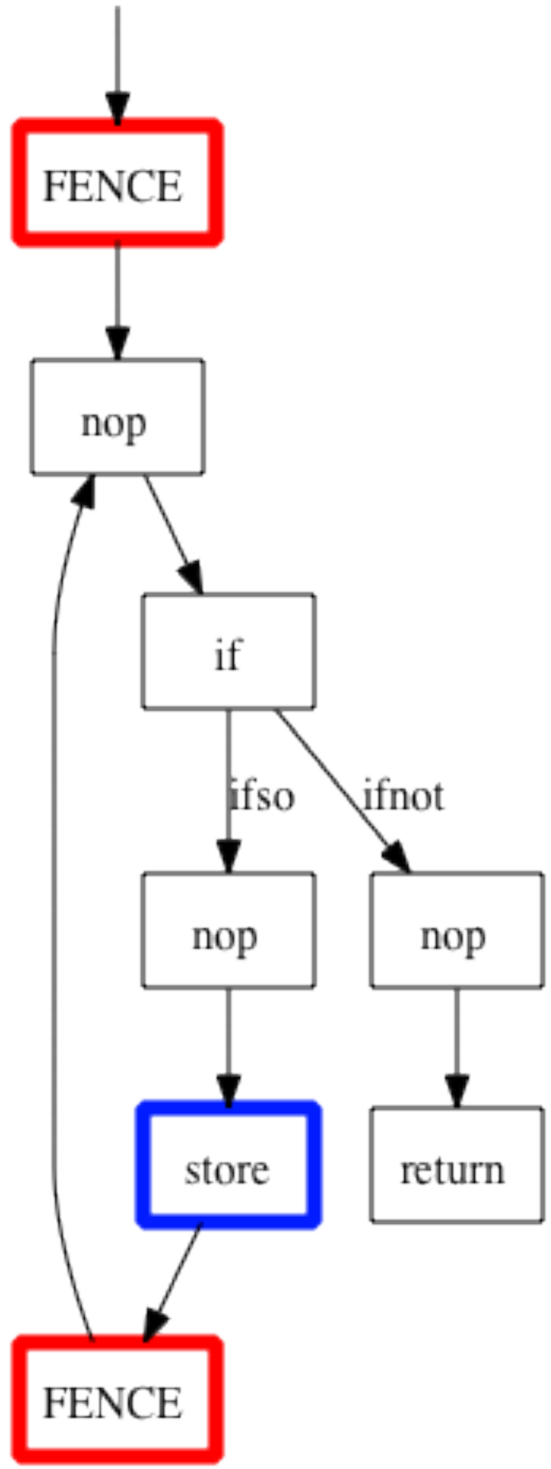


Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to hoist those fences out of the loop.

A closer look at the RTL



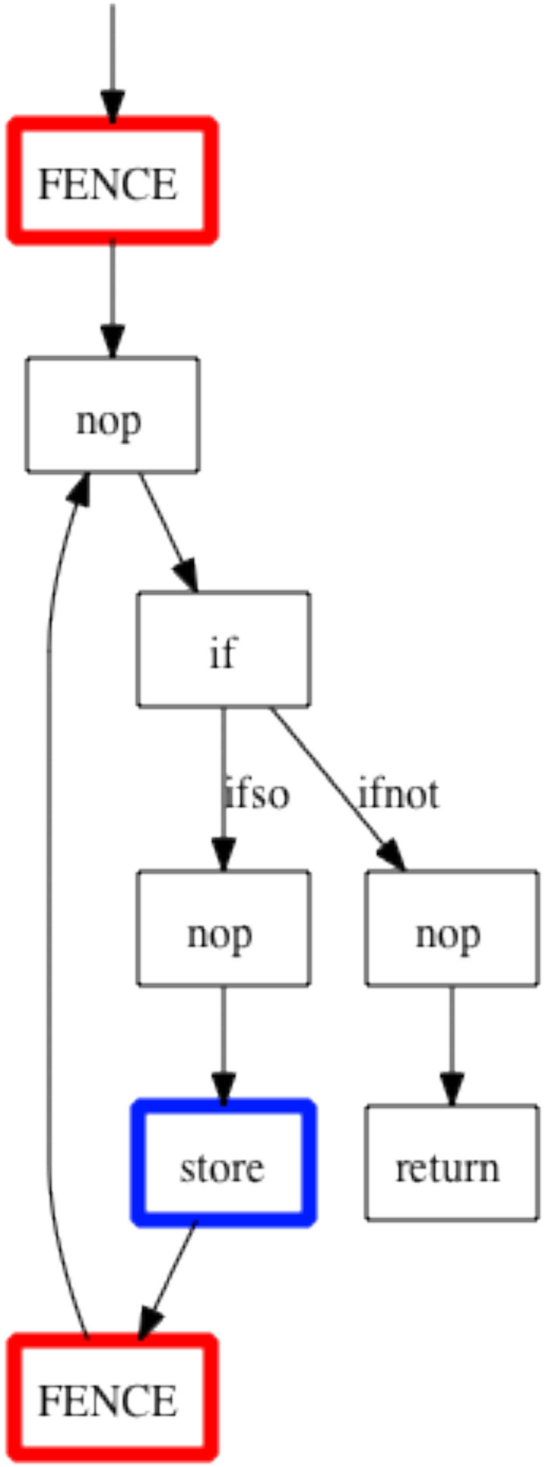
Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to [illegible] fences out of the loop.

Do you perform PRE?

A closer look at the RTL



Patterns like that on the left are common.

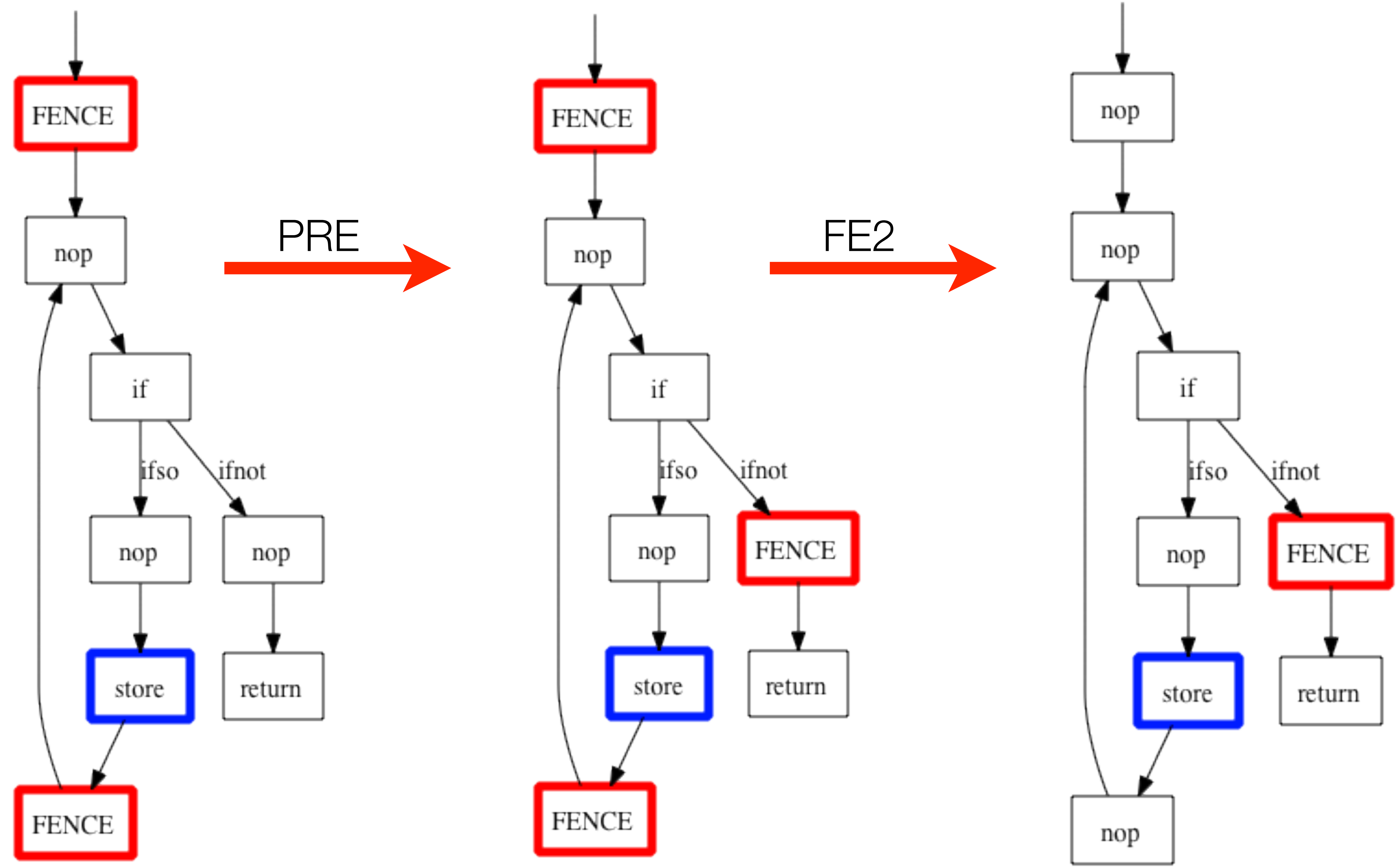
FE1 and FE2 do not optimise these patterns.

It would be nic out of the loop.

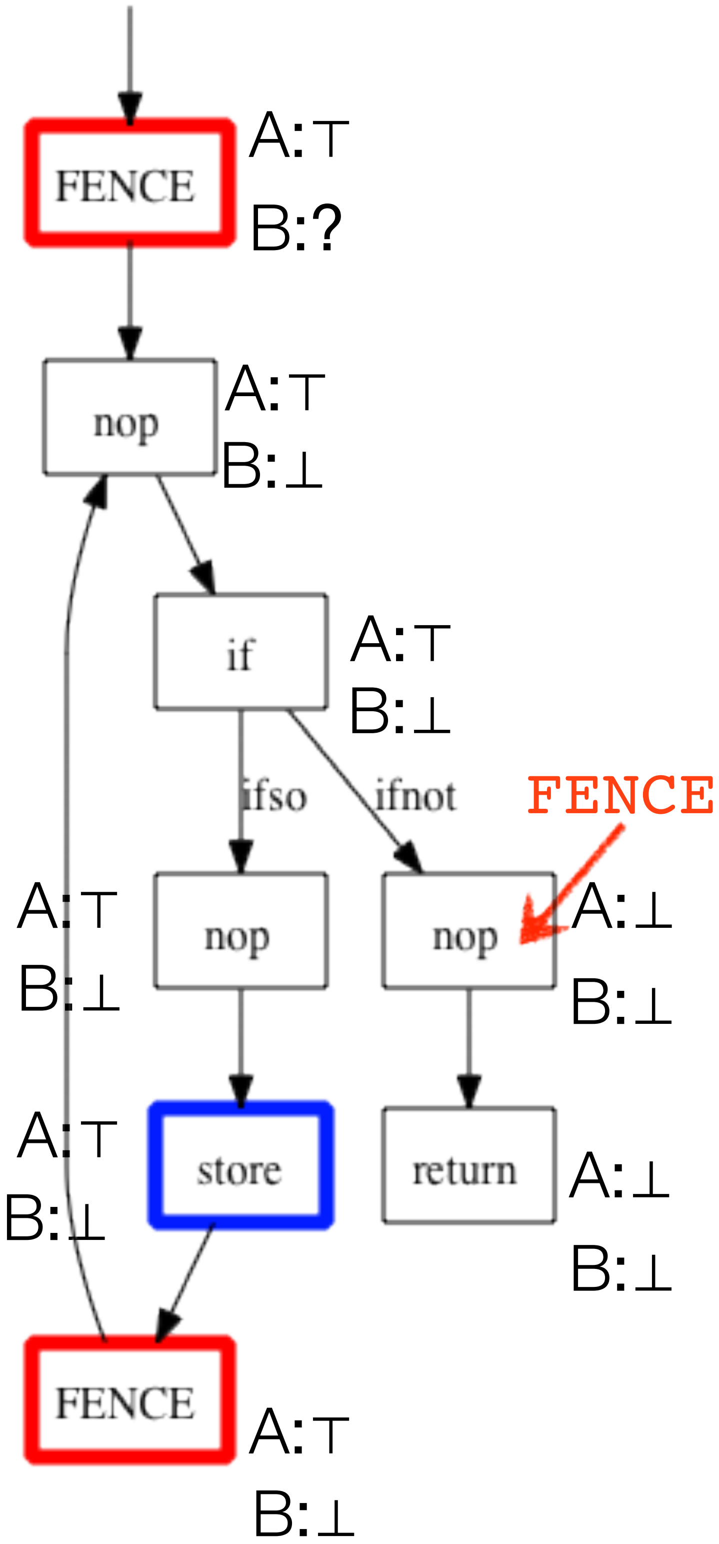
Do you perform PRE?

...adding a fence is always safe...

Partial redundancy elimination



Partial redundancy elimination



A: a backward analysis returning ⊤ if along *some path* after the current program point there is an atomic instruction with no intervening reads;

B: a forward analysis returning ⊥ if along all paths to the current program point there is a fence with no later reads or atomic instructions.

Replace ***NOP*** *with* ***FENCE*** *after conditionals if:*

- B returns ⊥
- A returns ⊥
- A returns ⊤ on the other branch

Partial redu

B returns ⊥:

a previous fence will be eliminated if we insert a fence at both branches of conditional nodes.

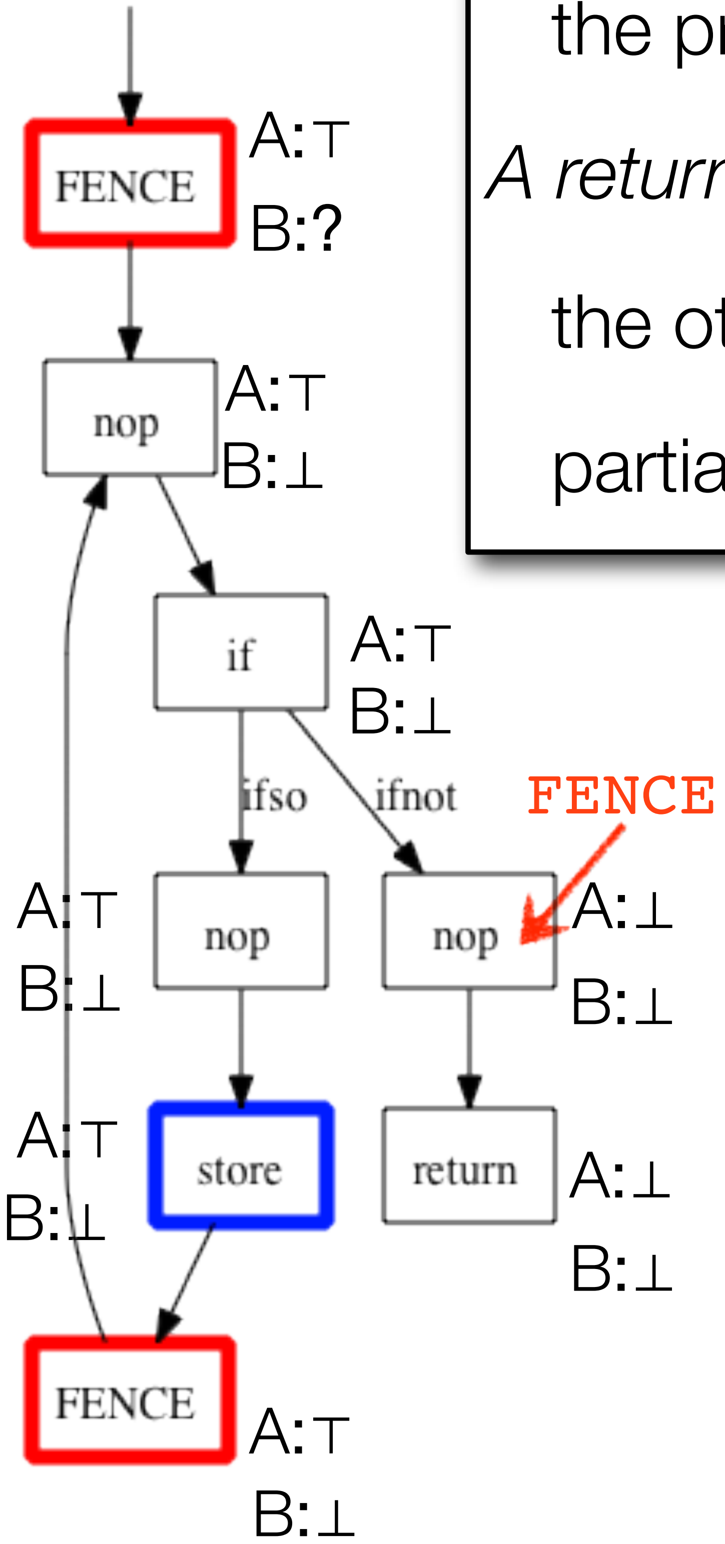
A returns ⊥:

the previous fence won't be removed by FE2.

A returns ⊤ on the other branch:

the other branch already makes the previous fence

partially redundant.



B: a forward analysis returning ⊥ if along all paths to the current program point there is a fence with no later reads or atomic instructions.

Replace ***`NOP`*** *with* ***`FENCE`*** *after conditionals if:*

- B returns ⊥
- A returns ⊥
- A returns ⊤ on the other branch

Evaluation of the optimisations

– Insert `MFENCE`s *before every read* (br), or *after every write* (aw).

– Count the `MFENCE` instructions in the generated code.

	br	br+FE1	aw	aw+FE2	aw+PRE+FE2
Dekker	3	2	5	4	4
Bakery	10	2	4	3	3
Treiber	5	2	3	1	1
Fraser	32	18	19	12	11
TL2	166	95	101	68	68
Genome	133	79	62	41	41
Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367

Evaluation of the optimisations

– Insert `MFENCE`s *before every read* (br), or *after every write* (aw).

– Cour

Important remark for your future work:

This is not a decent evaluation… we know nothing about real code, and the number of fences is not a good measure. But unclear how to do better.

Evaluation should be taken seriously by CS scientists!

`http://evaluate.inf.usi.ch/`

					2
C					
Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367

Conclusion

Syllabus

In these lectures we have covered the hardware models of two modern computer architectures (x86 and Power/ARM - at least for a large subset of their instruction set).

We have seen how compiler optimisations can also break concurrent programs and the importance of defining the memory model of high-level programming languages (and we have seen in detail the C++11 memory model).

We have also introduced some proof methods to reason about concurrency.

After these lectures, you might have the feeling that multicore programming is a mess and things can't just work.

The memory models of modern hardware are better understood.

Programming languages attempt to specify and implement reasonable memory models.

Researchers and programmers are now interested in these problems.

The memory models of modern hardware are better understood.

attempt

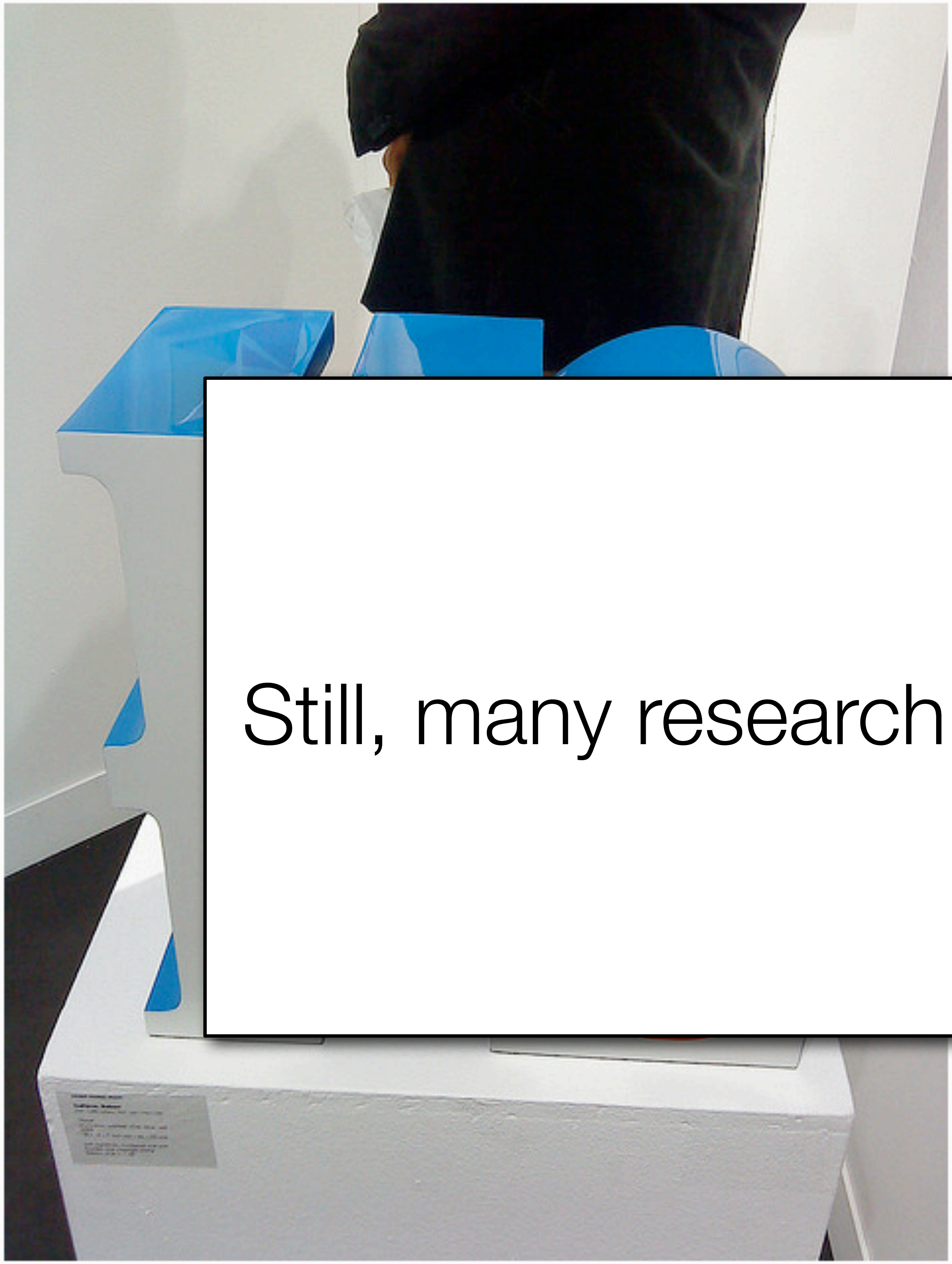
els.

mmers

se

problems.

Still, many open problems...



The memory models of modern hardware are better understood.

attempt

els.

mmers

se

problems.

Still, many research opportunities!

All these lectures are based on work done with/by my colleagues. Thank you!

And thank you all for attending these lectures!

Please, fill the course evaluation form.
It is vital feedback to make a better course next year.