Taming release-acquire consistency

Ori Lahav Nick Giannarakis Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

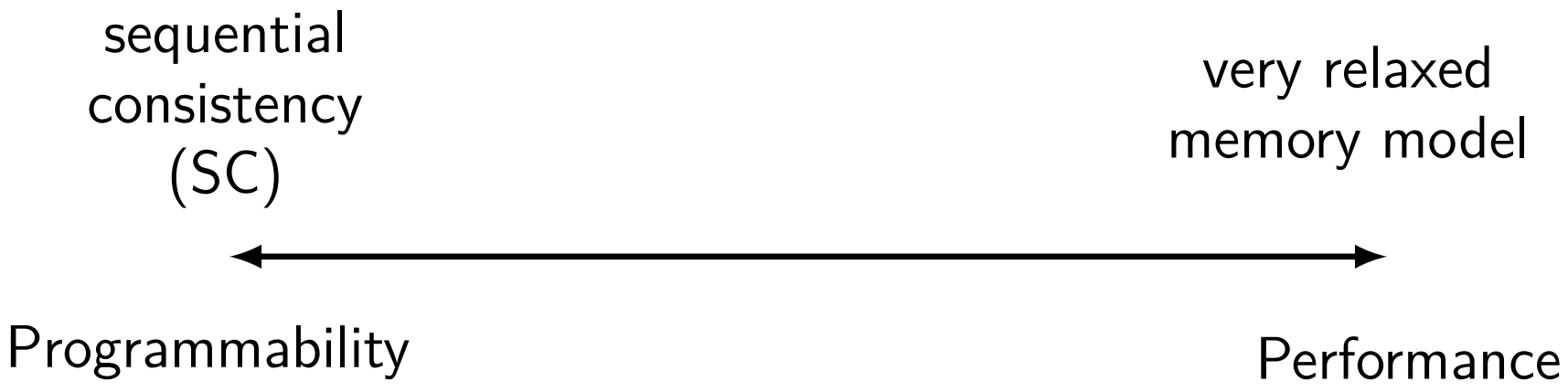
POPL 2016

Weak memory models

Weak memory models provide formal sound semantics for realistic high-performance concurrency.

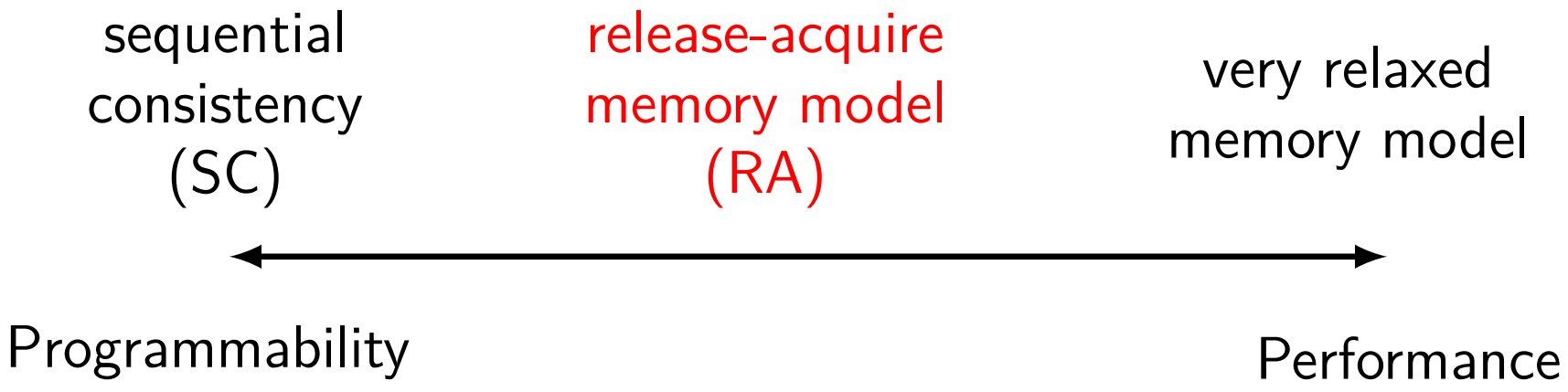
Weak memory models

Weak memory models provide formal sound semantics for realistic high-performance concurrency.



Weak memory models

Weak memory models provide formal sound semantics for realistic high-performance concurrency.



C11's release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

C11's release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Store buffering

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$\texttt{print}\ y$	$\parallel$	$\texttt{print}\ x$

both threads may print 0

Message passing

$$x = m = 0$$

$m := 42;$	$\parallel$	$\texttt{while}\ x = 0$
	$\parallel$	$\texttt{skip};$
$x := 1$	$\parallel$	$\texttt{print}\ m$

only 42 may be printed

C11's release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Store buffering

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$\texttt{print}\ y$	$\parallel$	$\texttt{print}\ x$

both threads may print 0

Message passing

$$x = m = 0$$

$m := 42;$	$\parallel$	$\texttt{while}\ x = 0$
	$\parallel$	$\quad\texttt{skip};$
$x := 1$	$\parallel$	$\texttt{print}\ m$

only 42 may be printed

C11's release-acquire memory model

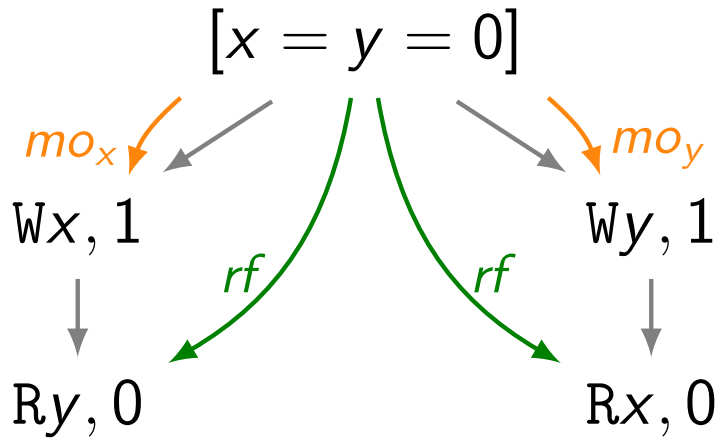
C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Store buffering

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$\texttt{print}\ y$	$\parallel$	$\texttt{print}\ x$

both threads may print 0

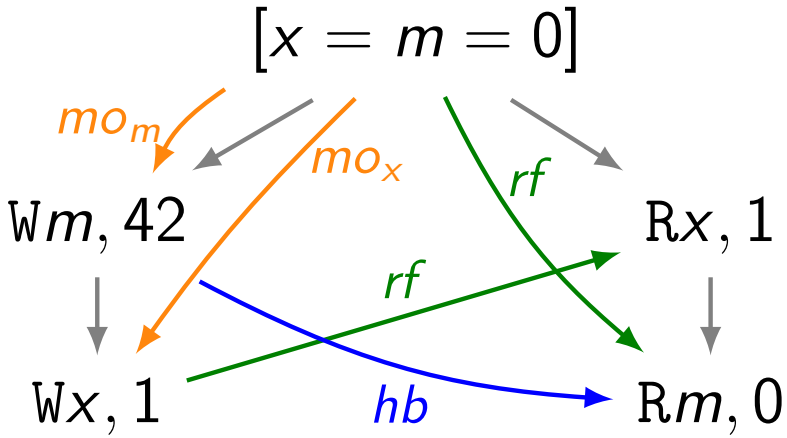


Message passing

$$x = m = 0$$

$m := 42;$	$\parallel$	$\texttt{while}\ x = 0$
$x := 1$	$\parallel$	$\texttt{skip};$
	$\parallel$	$\texttt{print}\ m$

only 42 may be printed



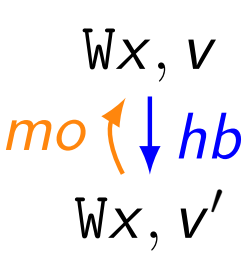
Formal definition

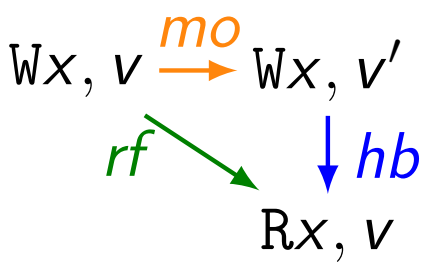
An execution is *consistent* if there are relations:

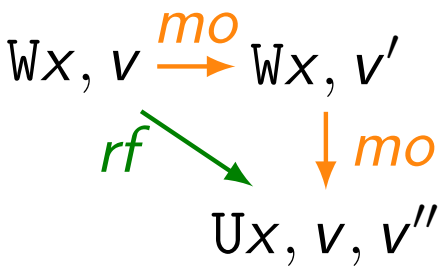
- *reads-from* (*rf*): maps every read to a corresponding write, such that:

  happens-before $= (\textit{program-order} \cup \textit{reads-from})^+$ is irreflexive.

- *modification-order* (*mo*): total order on same-location writes, such that **none of the following occur:**







Good news

- Verified compilation schemes:
  - x86-TSO (trivial compilation) [Batty el al. '11]
  - Power [Batty el al. '12] [Sarkar el al. '12]
- RA supports intended optimizations:
  - In particular, write-read reordering (unlike SC):

$$\mathtt{W}x \rightarrow \mathtt{R}y \quad \rightsquigarrow \quad \mathtt{R}y \rightarrow \mathtt{W}x$$

- DRF theorem (unlike full C11):
  - No data races under SC ensures no weak behaviors
- Monotonicity (unlike full C11, and x86-TSO):
  - Adding synchronization does not introduce new behaviors
- Program logics:
  - RSL [Vafeiadis and Narayan '13]
  - GPS [Turon et al. '14]
  - OGRA [L and Vafeiadis '15]

Bad news

1. Allowed dubious behaviors

2. Overly weak SC-fences

3. No intuitive operational semantics

Bad news

1. Allowed dubious behaviors

2. Overly weak SC-fences

3. No intuitive operational semantics

Unobservable behaviors

$$x = y = a = b = 0$$

$$\begin{array}{l} x := 1; \\ y := 2; \\ a := 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{print}\ a; \\ \texttt{print}\ b; \\ \texttt{print}\ x; \\ \texttt{print}\ y \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ x := 2; \\ b := 1 \end{array}$$

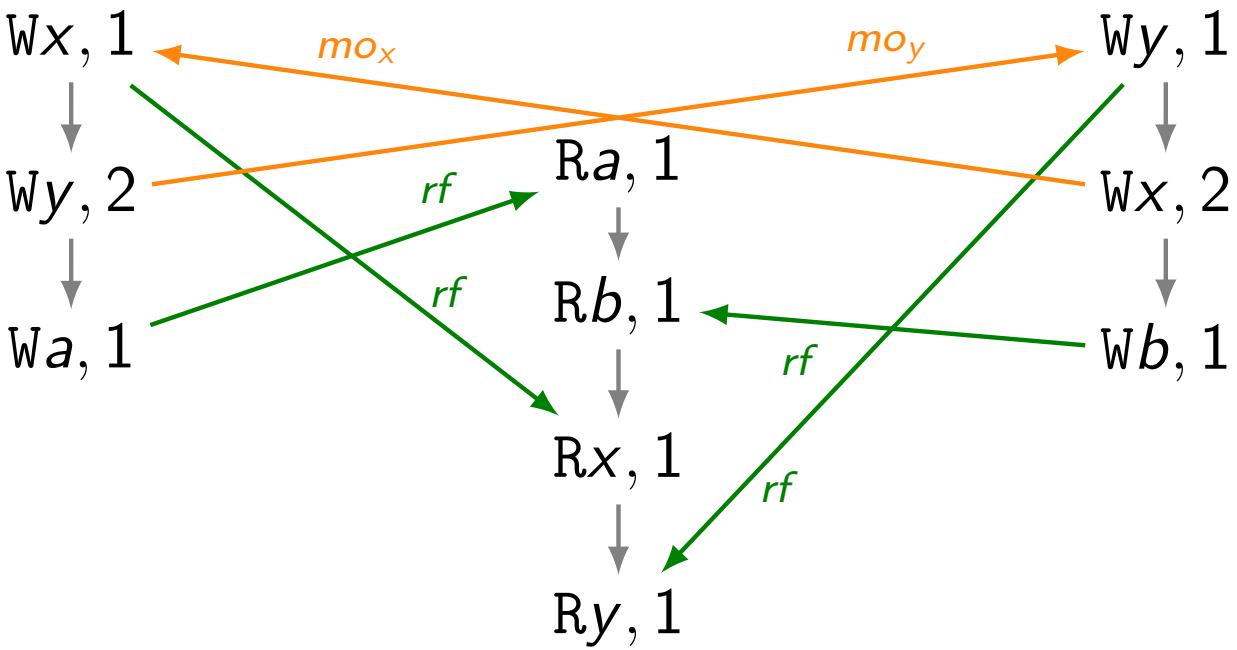
Can this program print $1, 1, 1, 1$?

Unobservable behaviors

$$x = y = a = b = 0$$

$$\begin{array}{l} x := 1; \\ y := 2; \\ a := 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{print } a; \\ \texttt{print } b; \\ \texttt{print } x; \\ \texttt{print } y \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ x := 2; \\ b := 1 \end{array}$$

Can this program print $1, 1, 1, 1$?

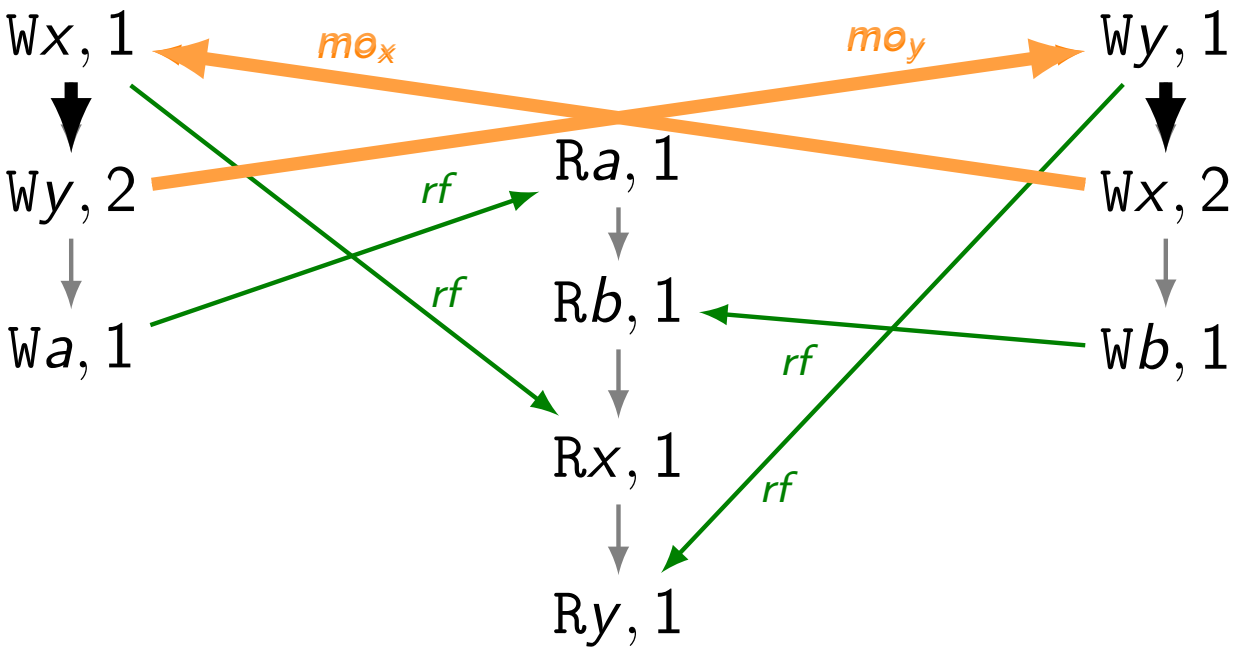


Unobservable behaviors

$$x = y = a = b = 0$$

$$\begin{array}{l} x := 1; \\ y := 2; \\ a := 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{print } a; \\ \texttt{print } b; \\ \texttt{print } x; \\ \texttt{print } y \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ x := 2; \\ b := 1 \end{array}$$

Can this program print $1, 1, 1, 1$?



Strong release/acquire consistency

Definition (SRA-consistency)

An execution is *SRA-consistent* if it is RA-consistent and $hb \cup \bigcup_x mo_x$ is acyclic.

Strong release/acquire consistency

Definition (SRA-consistency)

An execution is *SRA-consistent* if it is RA-consistent and $hb \cup \bigcup_x mo_x$ is acyclic.

If there are no write-write races then SRA and RA coincide.

Better product, same price

- Same compiler optimizations are sound.
- Compilation to x86-TSO and Power is still correct.

* *No better deal* for Power:

Power model restricted to RA accesses = SRA

(based on Power's declarative model of [Alglave et al. '14])

2. Using Fences to Recover SC

C11's SC-fences

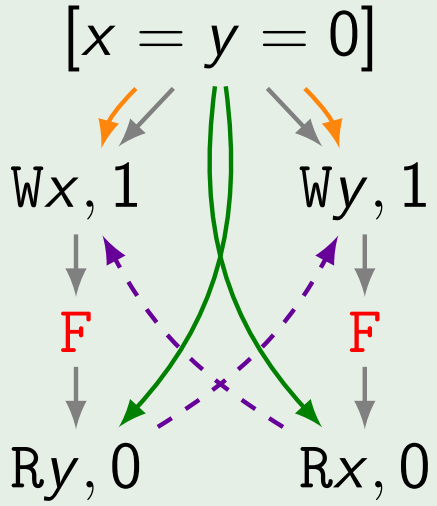
- The strongest fence instruction provided by C11 is *SC*-fence.

Example (Store Buffering)

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
`fence();`	$\parallel$	`fence();`
`print` y	$\parallel$	`print` x

printing 0 in both threads is disallowed



Inconsistent: $(\mathtt{F} \times \mathtt{F}) \cap (po^?; (hb \cup mo \cup fr); po^?)$ is cyclic.

- Using the semantics of [Batty et al. '16].

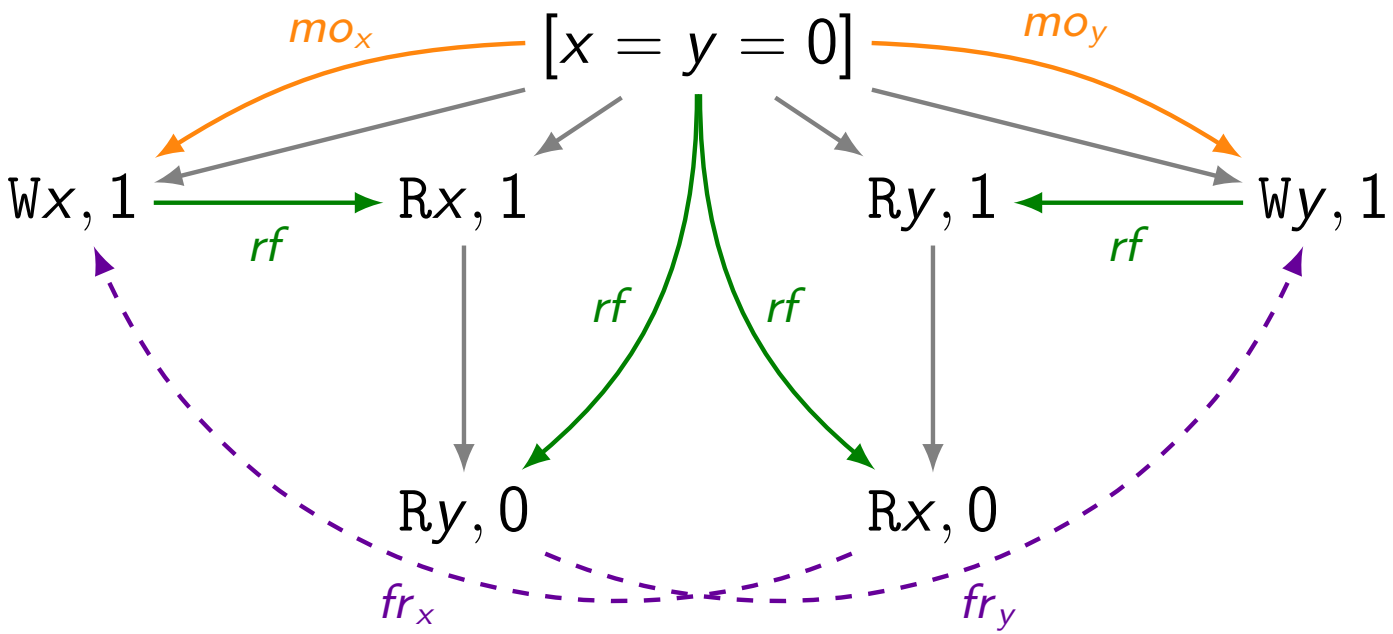
SC-fences are overly weak

Independent reads, independent writes

$$x = y = 0$$

$$x := 1 \;\Big\|\; \begin{array}{l}\texttt{print } x;\\ \\ \texttt{print } y\end{array} \;\Big\|\; \begin{array}{l}\texttt{print } y;\\ \\ \texttt{print } x\end{array} \;\Big\|\; y := 1$$

both threads may print $1, 0$



consistent execution

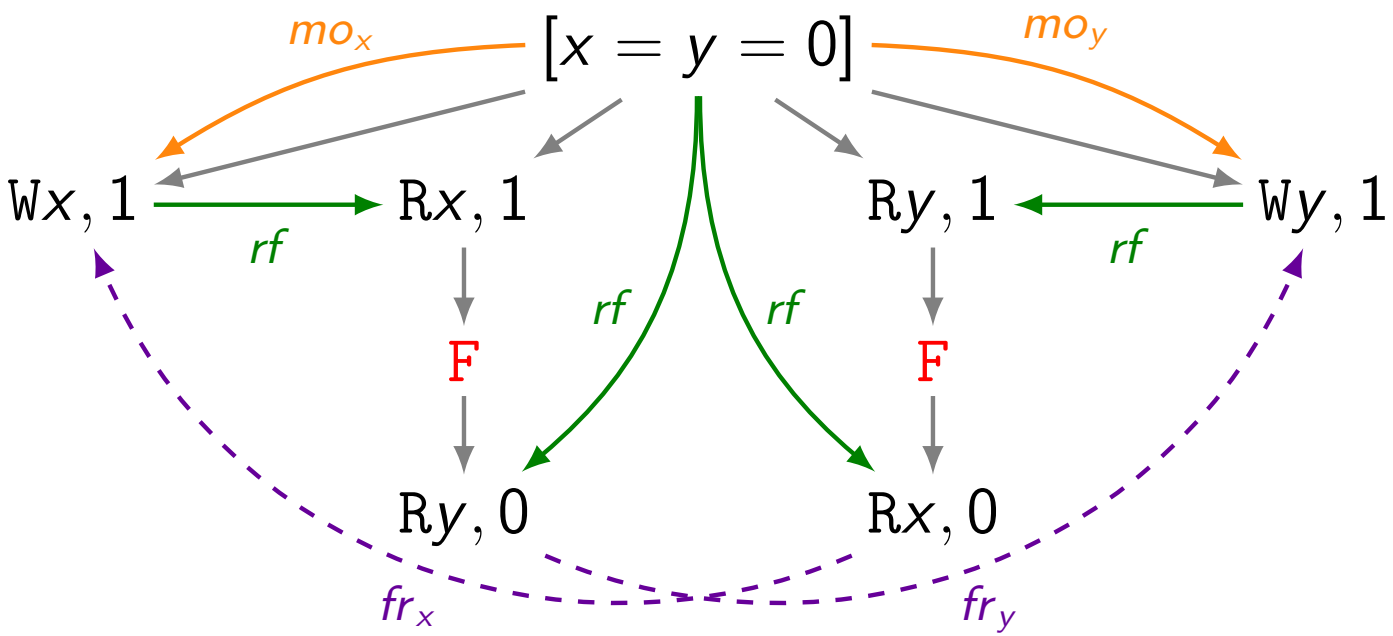
SC-fences are overly weak

Independent reads, independent writes

$$x = y = 0$$

$$x := 1 \;\Big\|\; \begin{array}{l}\texttt{print } x;\\ \texttt{fence();}\\ \texttt{print } y\end{array} \;\Big\|\; \begin{array}{l}\texttt{print } y;\\ \texttt{fence();}\\ \texttt{print } x\end{array} \;\Big\|\; y := 1$$

both threads may print $1, 0$

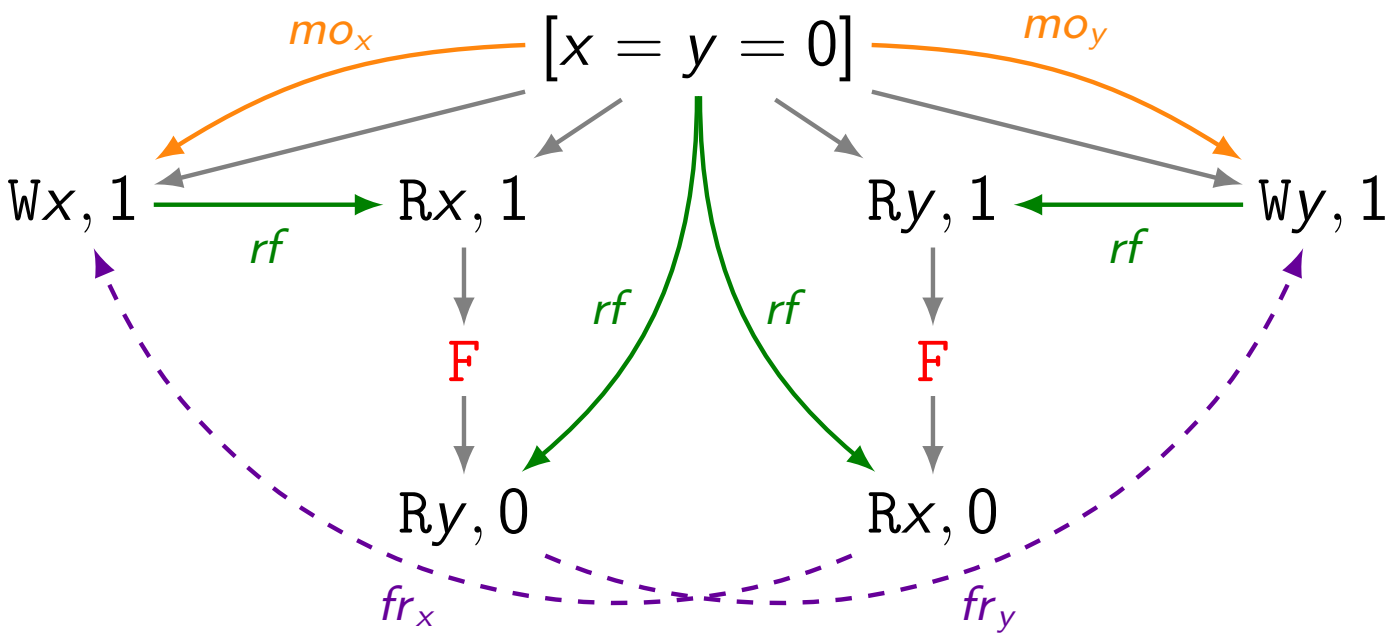


consistent execution

SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by *hb*.

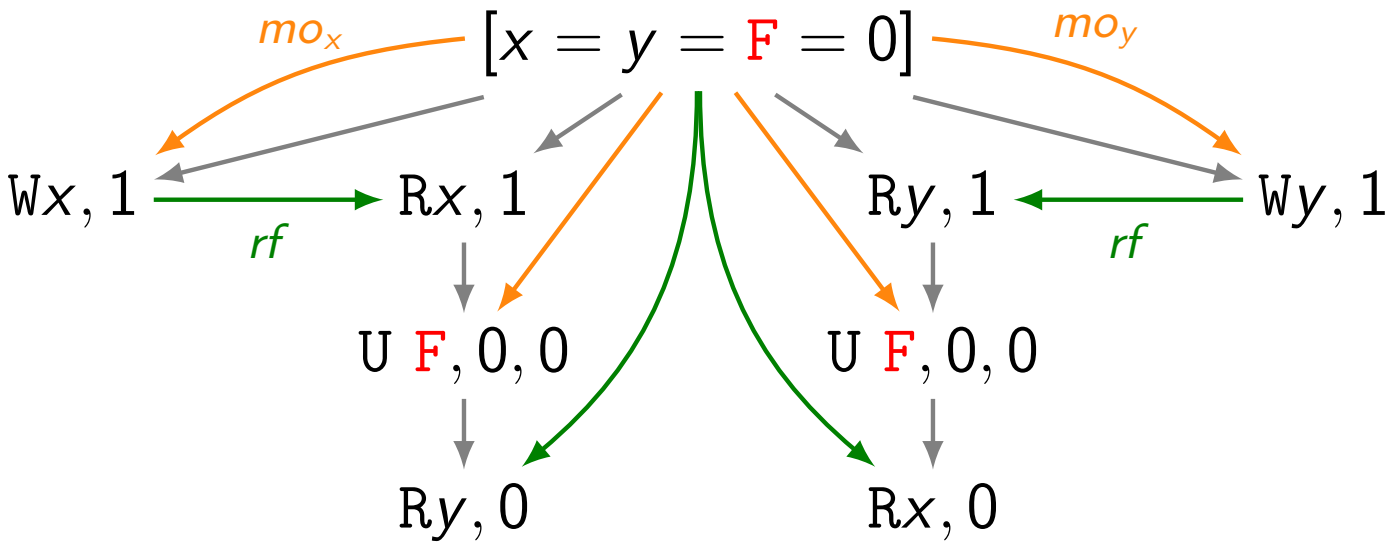


consistent execution

SC-fences are overly weak

Our suggestion

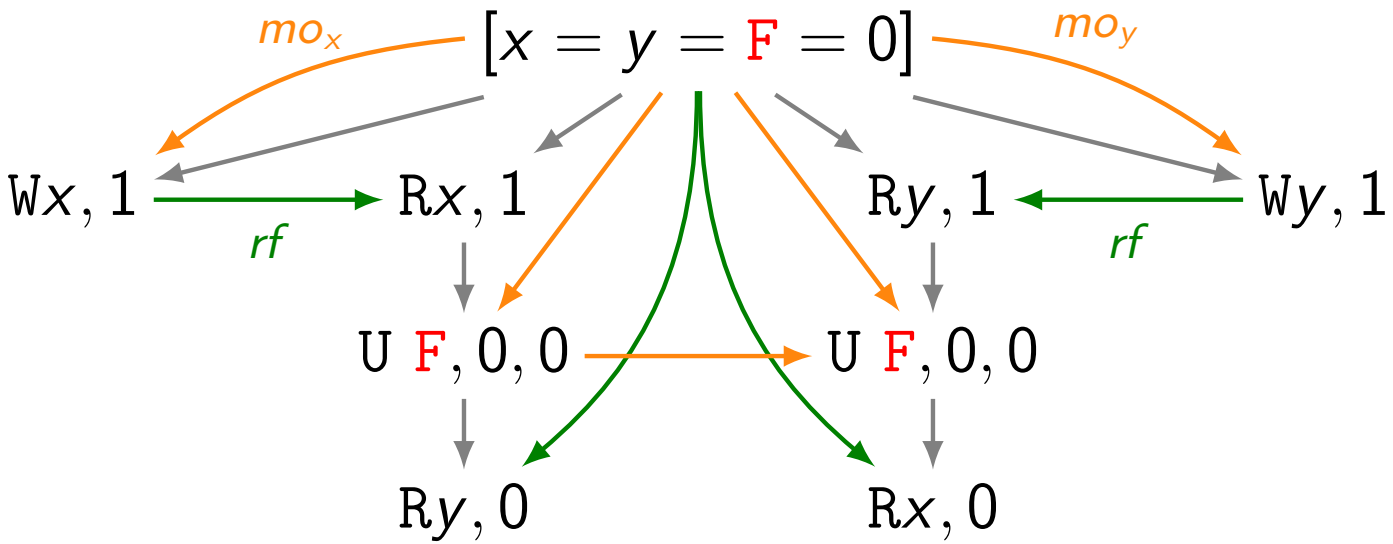
- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by *hb*.



SC-fences are overly weak

Our suggestion

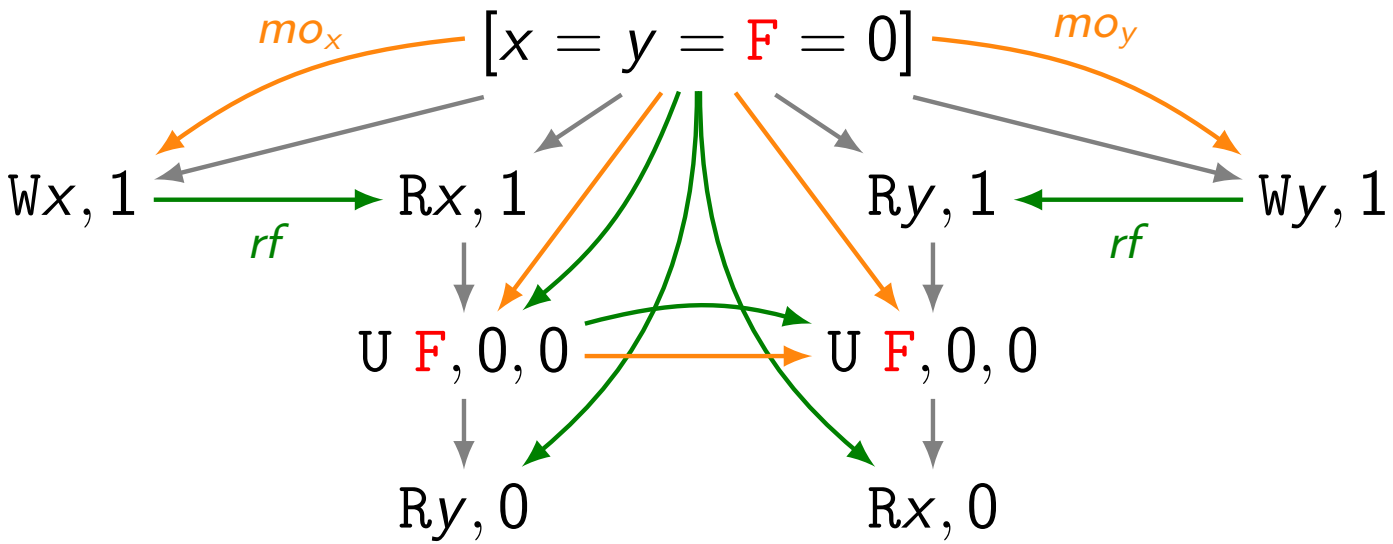
- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by *hb*.



SC-fences are overly weak

Our suggestion

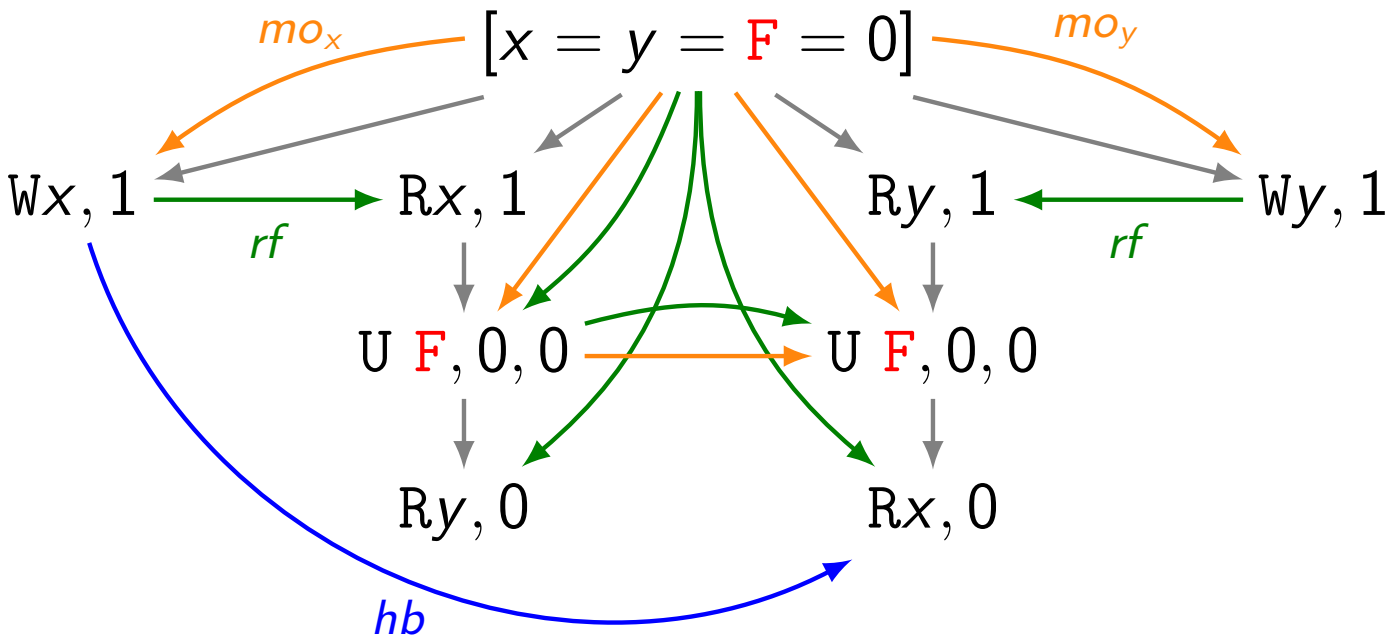
- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by *hb*.



SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by *hb*.



Inconsistent: $\mathtt{R}x, 0$ reads an overwritten value

Compilation schemes are not affected

x86-TSO

- SC-fence $\hookrightarrow$ `mfence`
- Atomic update $\hookrightarrow$ `lock xchg`
- `mfence` and `lock xchg` of an unused value have the same operational effect.

Power

- `sync` events are equivalent to a acquire/release atomic updates from an otherwise-unused location.

Reductions to SC

Theorem (basic reduction)

If a program includes a fence between every two racy accesses to different shared variables, then it has only SC behaviors.

- Under x86-TSO, it suffices to have a fence between racy writes and subsequent racy reads.

$x := e;$
`fence();`
$r := y$

Theorem (advanced reduction, simplified version)

For client-server programs, it suffices to place fences as under x86-TSO.

In the paper: application to an RCU implementation.

3. An Operational Presentation

- Easier to understand by simulating step-by-step progress
- Foundation for traditional verification techniques

Example: the SRA machine (first attempt)

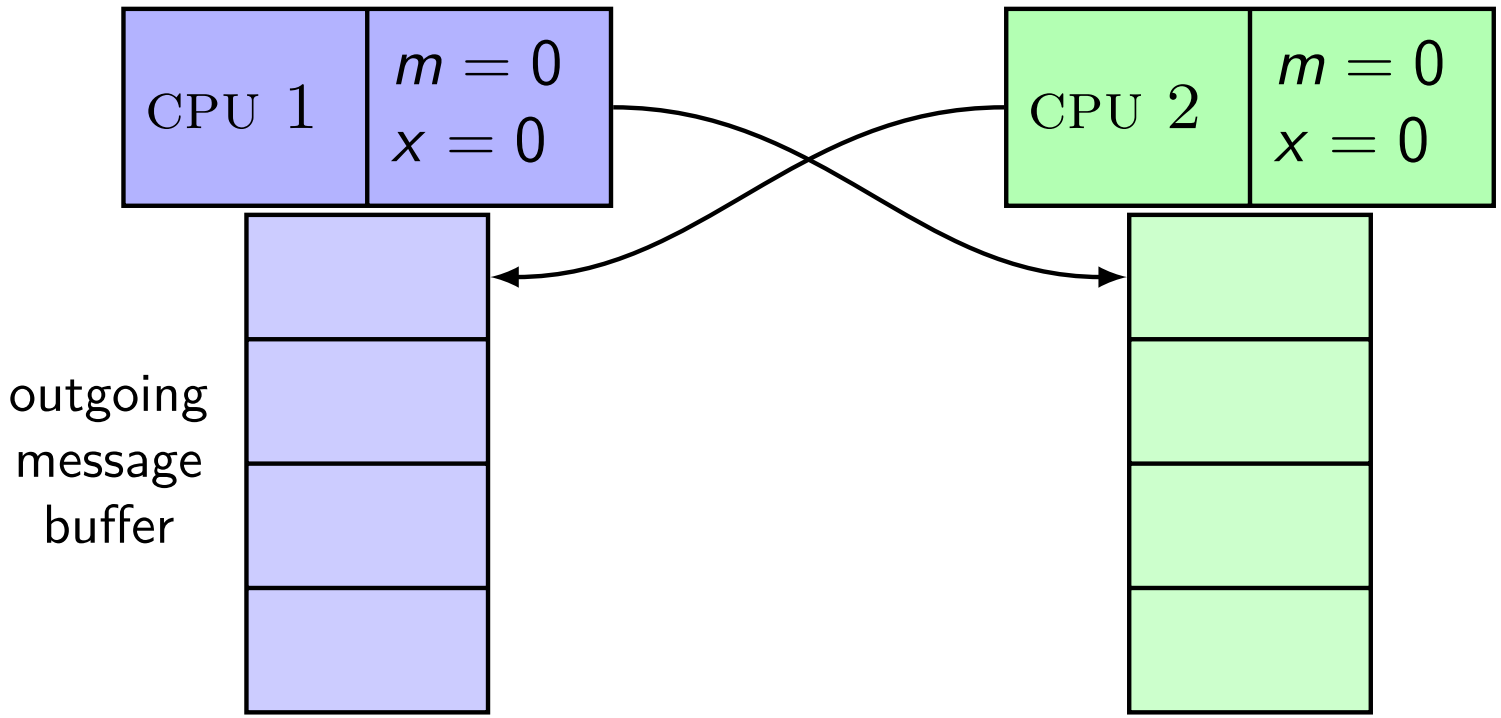
Message passing

$m = x = 0$

- $m := 42$;
  $x := 1$

$\parallel$

- `while` $x = 0$
  `skip`;
  `print` m

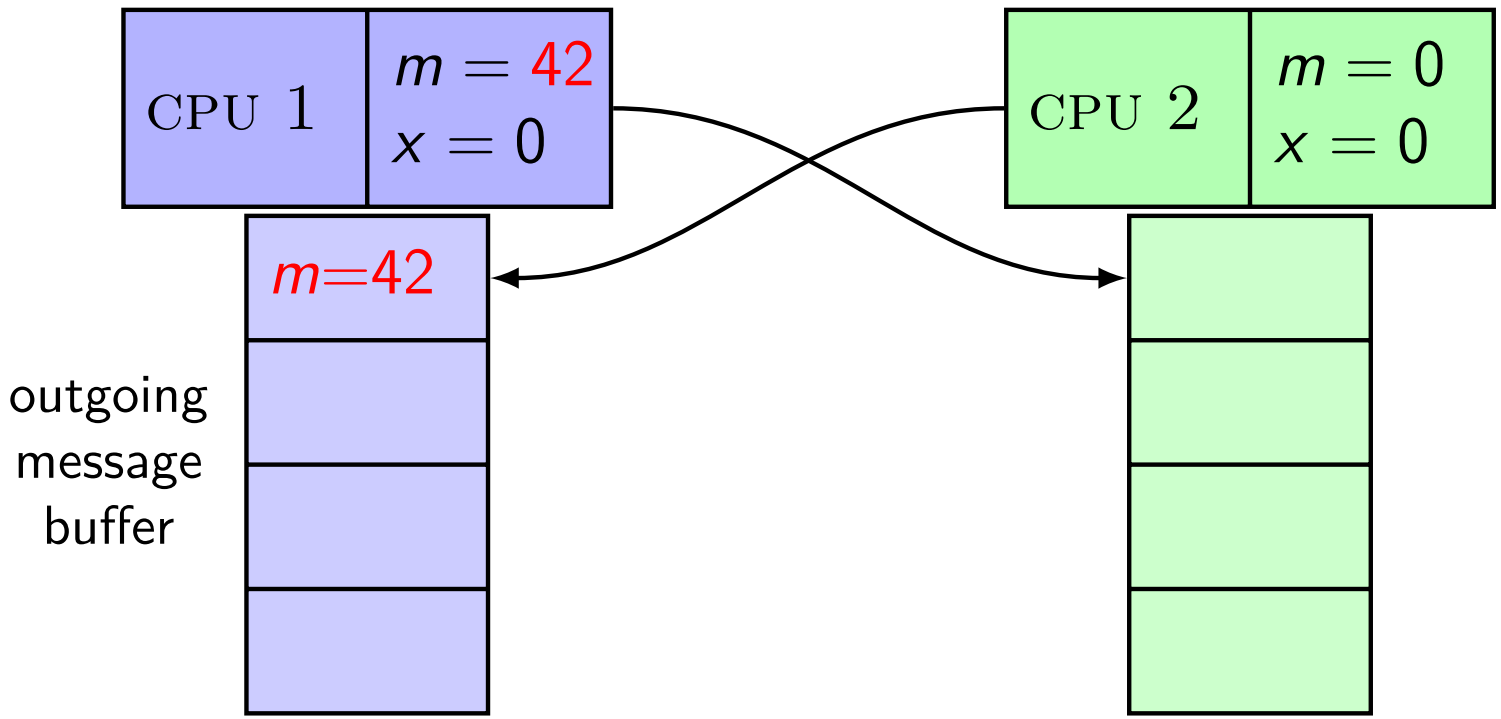


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$m := 42;$	$\parallel$	▶ `while` $x = 0$
▶ $x := 1$		`skip;`
		`print` m

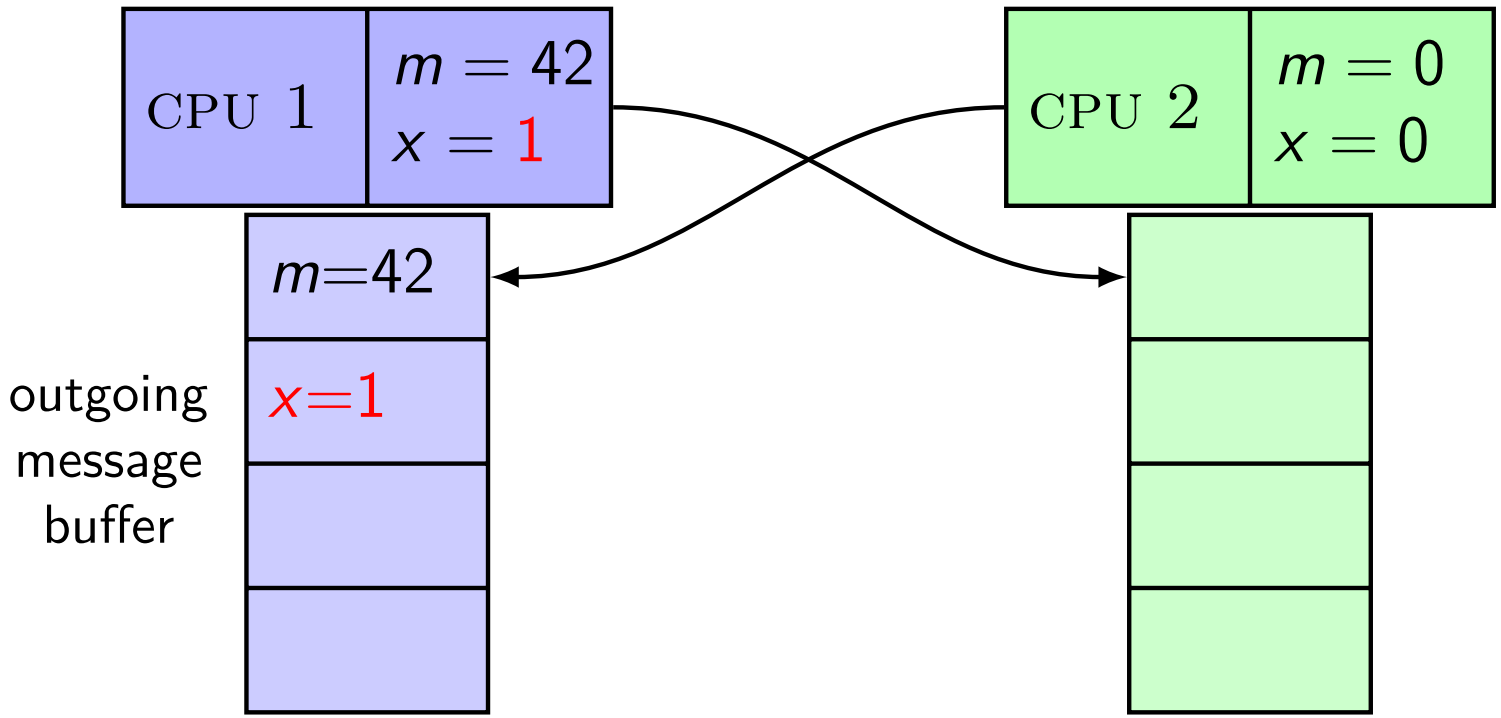


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$$m := 42; \quad x := 1 \quad \Big\| \quad \blacktriangleright \texttt{while } x = 0 \texttt{ skip; print } m$$

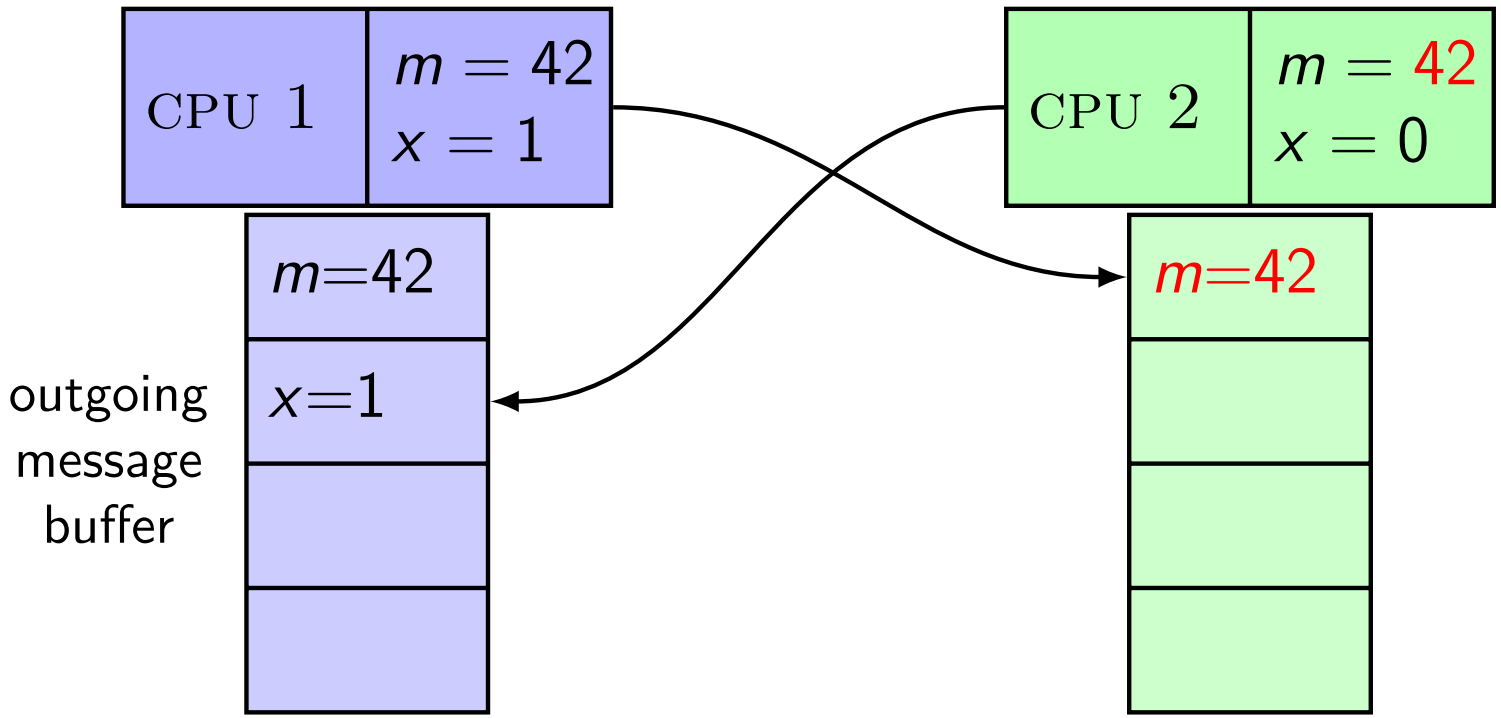


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$m := 42;$ $x := 1$	$\parallel$	▶ `while` $x = 0$ `skip;` `print` m



Example: the SRA machine (first attempt)

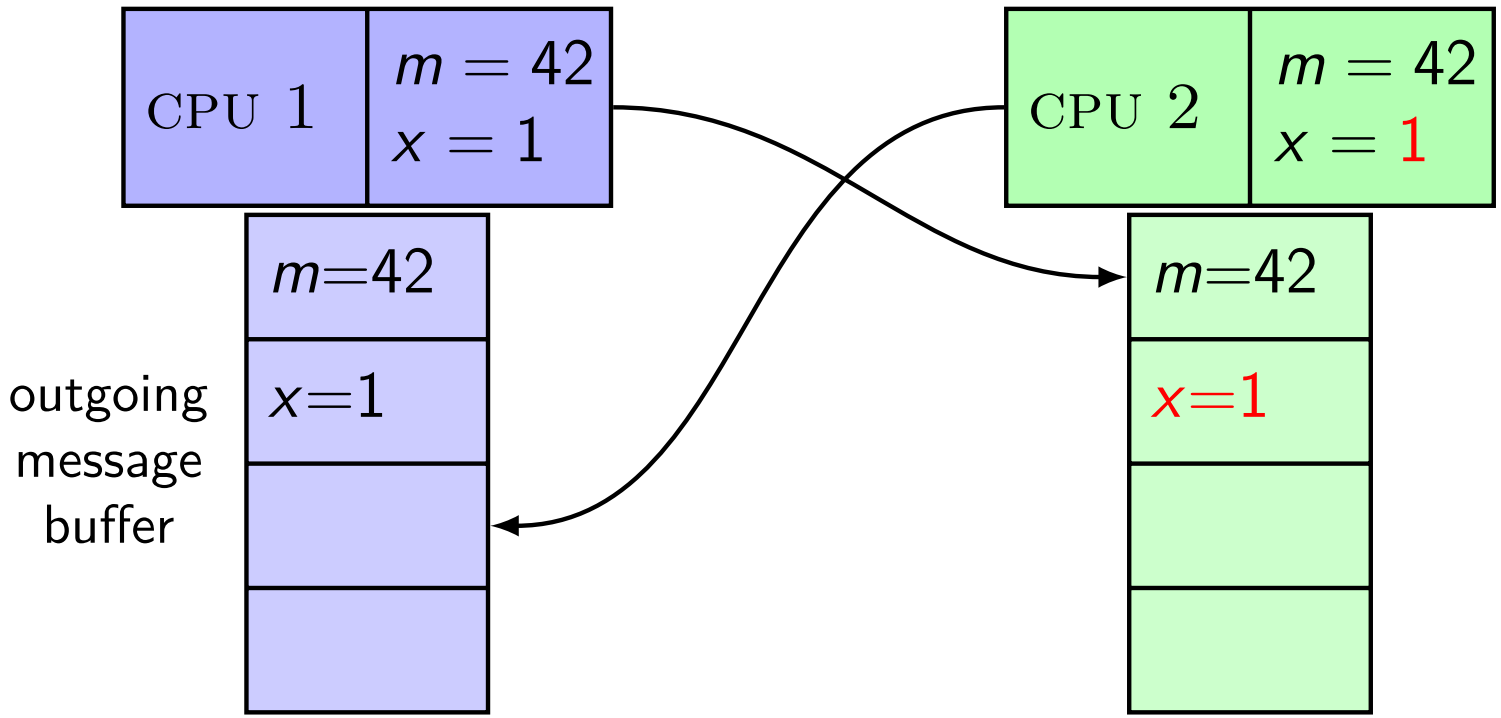
Message passing

$m = x = 0$

$m := 42;$
$x := 1$

$\parallel$

▶ `while` $x = 0$
`skip;`
`print` m

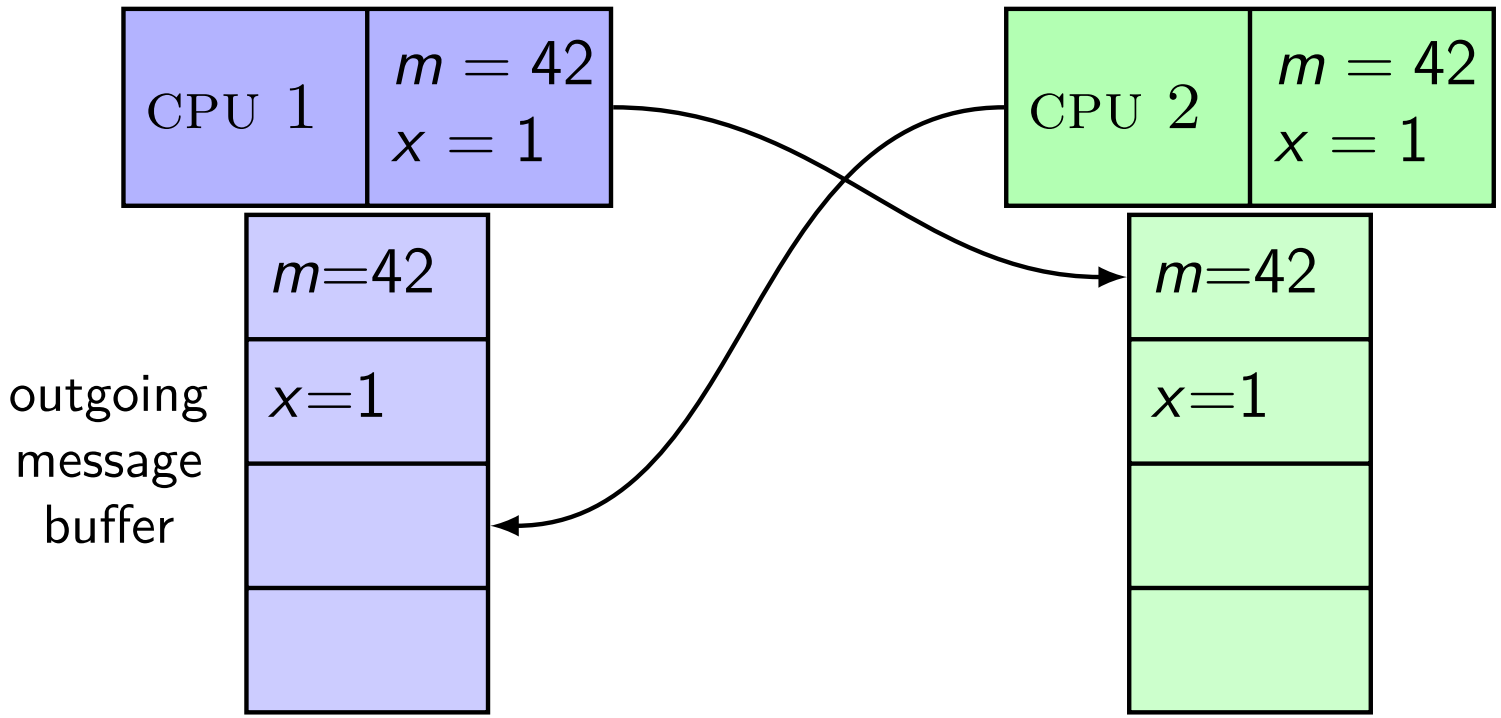


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$m := 42;$	$\parallel$	`while` $x = 0$
$x := 1$		`skip;`
		▶ `print` m



Timestamps

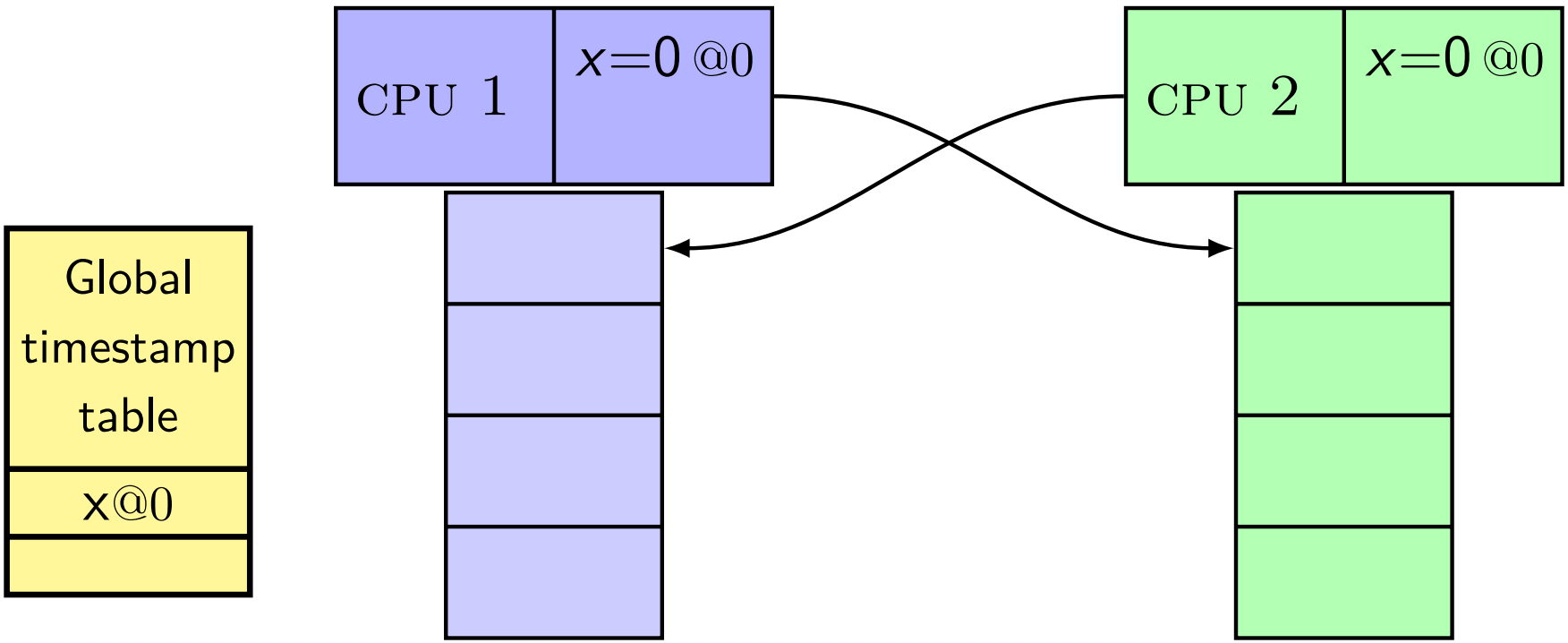
$$
\begin{array}{l}
x := 6; \\
\texttt{if } x = 7 \\
\quad \texttt{print } \text{`}go\text{'}
\end{array}
\quad \Big\| \quad
\begin{array}{l}
x := 7; \\
\texttt{if } x = 6 \\
\quad \texttt{print } \text{`}go\text{'}
\end{array}
$$

'*go*' should be printed at most once

Timestamps

▶ $x := 6$;	▶ $x := 7$;
`if` $x = 7$	`if` $x = 6$
`print` ‘*go*’	`print` ‘*go*’

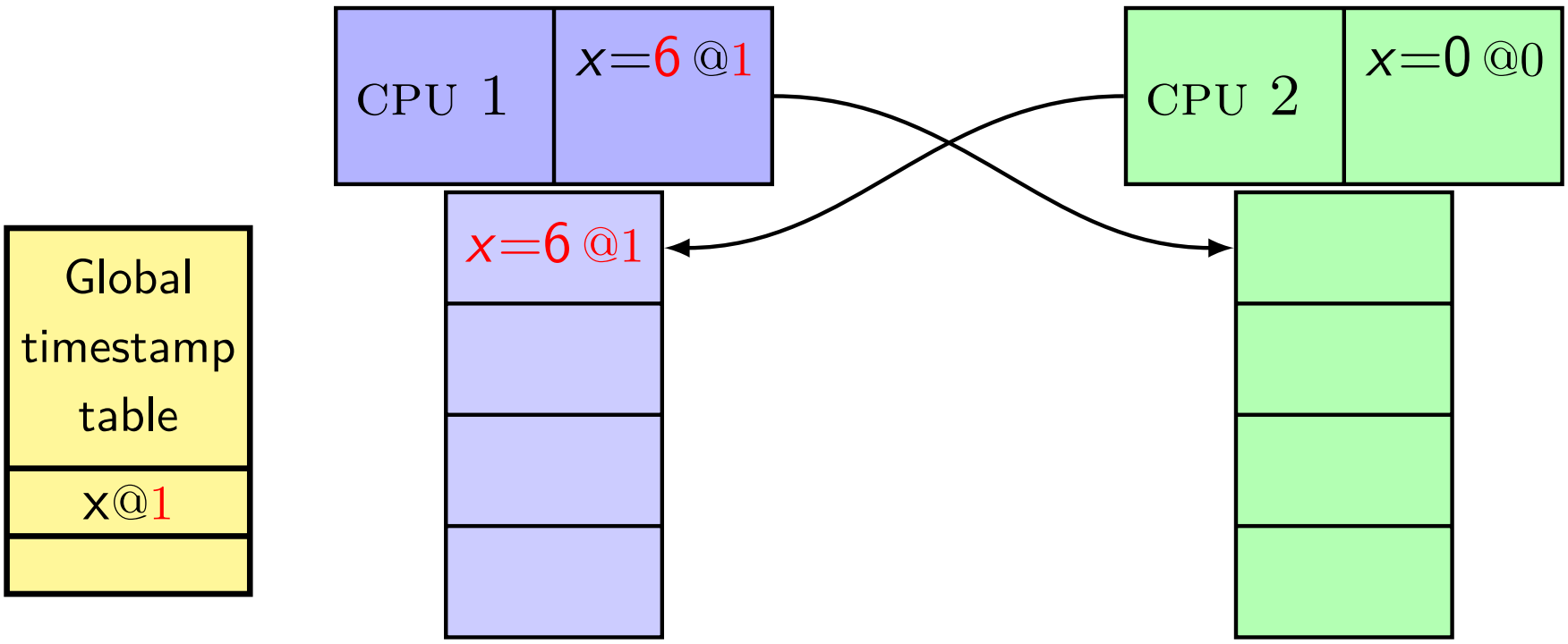
‘*go*’ should be printed at most once



Timestamps

$x := 6;$	▶ $x := 7;$
▶ `if` $x = 7$	`if` $x = 6$
`print` ‘*go*’	`print` ‘*go*’

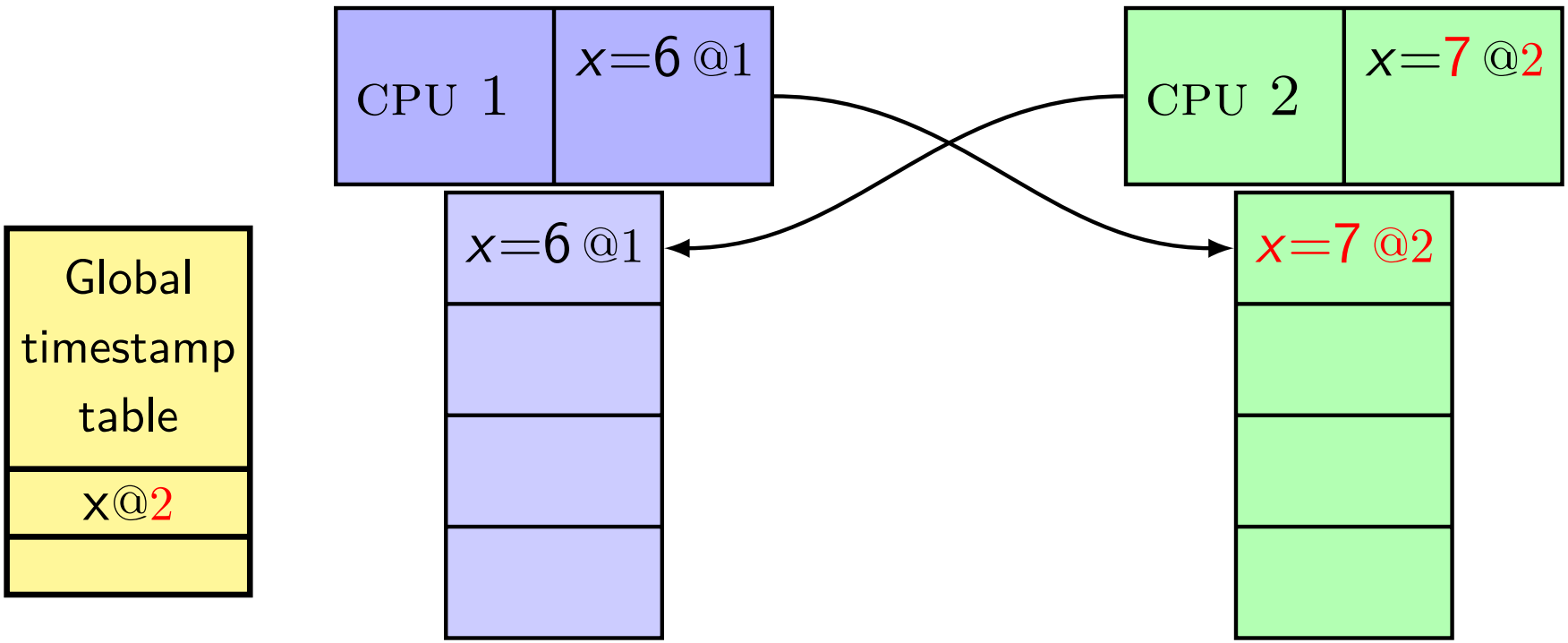
‘*go*’ should be printed at most once



Timestamps

$x := 6;$	$x := 7;$
▶ `if` $x = 7$	▶ `if` $x = 6$
`print` '*go*'	`print` '*go*'

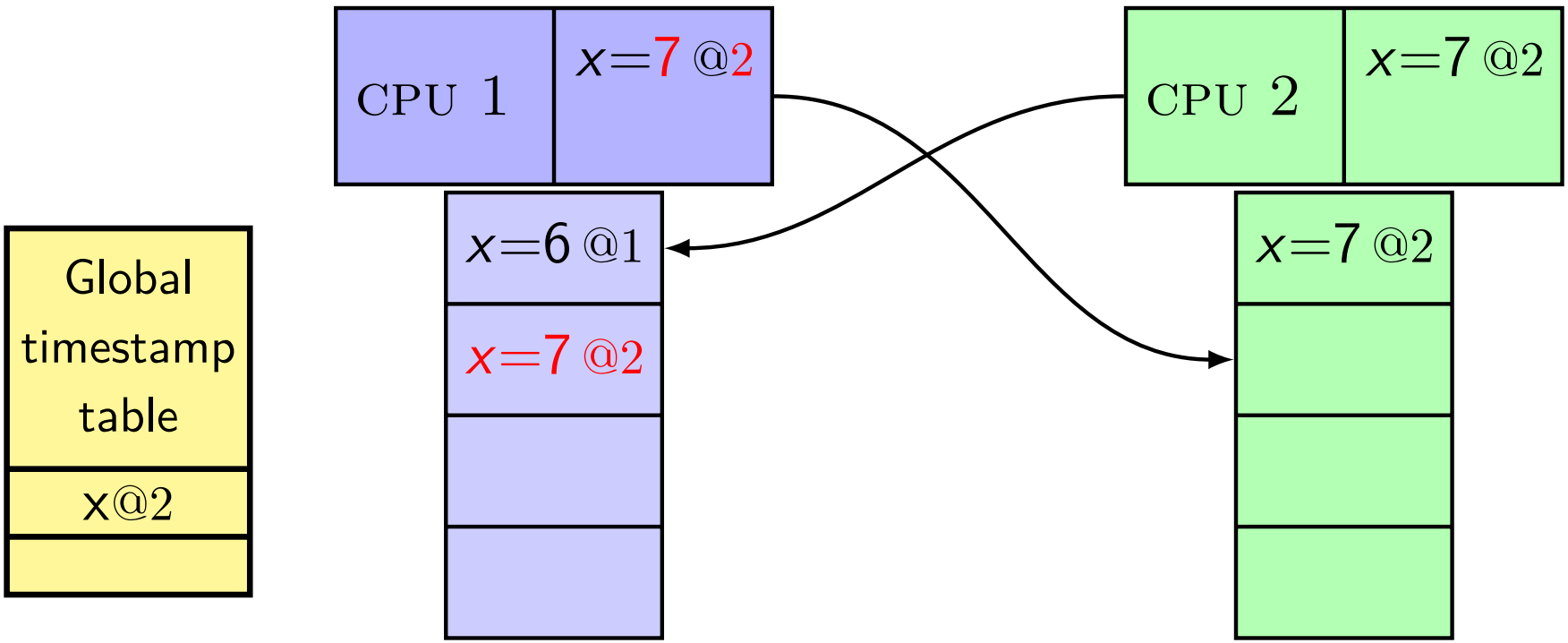
'*go*' should be printed at most once



Timestamps

$x := 6;$ ▶ `if` $x = 7$ `print` '*go*'	$x := 7;$ ▶ `if` $x = 6$ `print` '*go*'

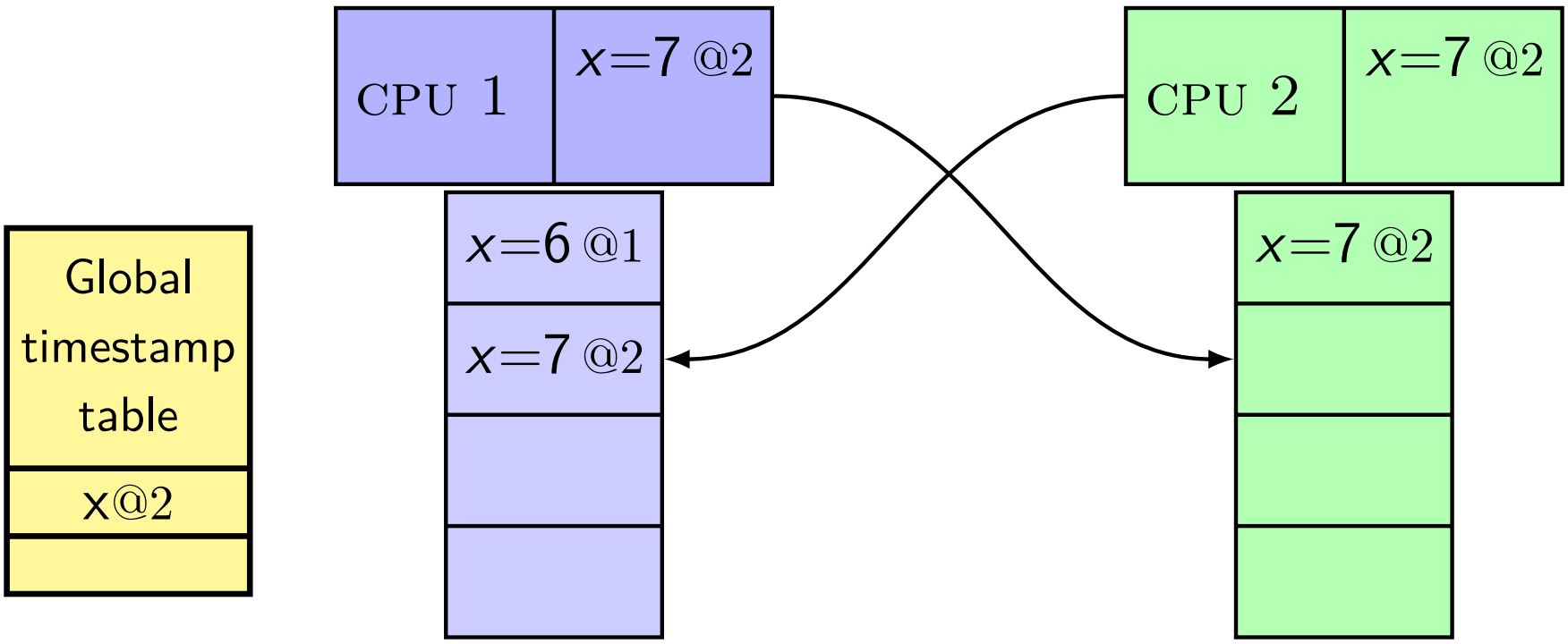
'*go*' should be printed at most once



Timestamps

$x := 6;$ ▶ `if` $x = 7$ `print` ‘*go*’	$x := 7;$ ▶ `if` $x = 6$ `print` ‘*go*’

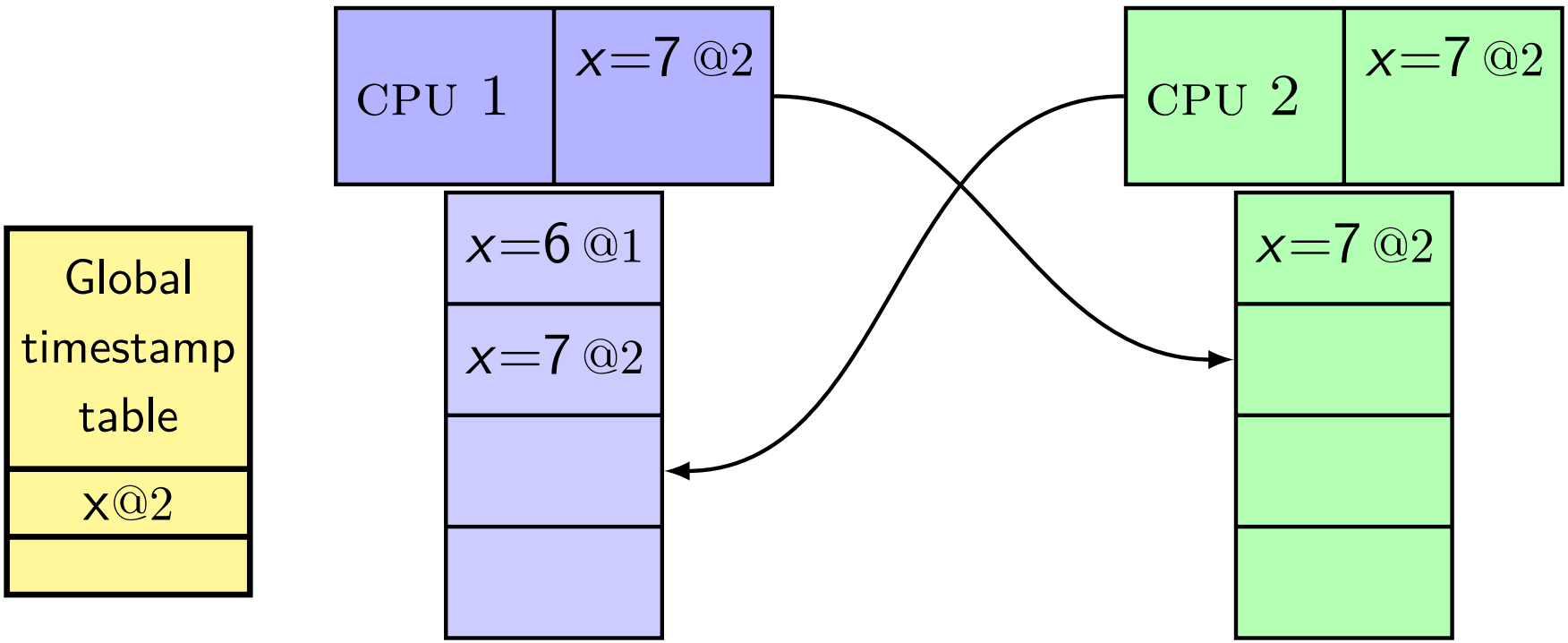
‘*go*’ should be printed at most once



Timestamps

$x := 6;$	$x := 7;$
▶ `if` $x = 7$	▶ `if` $x = 6$
`print` '*go*'	`print` '*go*'

'*go*' should be printed at most once



Summary

- The release/acquire fragment of the C/C++11 memory model, strikes a good balance between performance and programmability.
- We propose a strengthening of this memory model that:
  - forbids weak behaviors, unobservable in any implementation
  - has simple fence semantics, with SC-reduction theorems
  - admits intuitive operational semantics
- The stronger model has no additional implementation cost.

See `http://plv.mpi-sws.org/sra/` for more details and Coq proofs.

Summary

- The release/acquire fragment of the C/C++11 memory model, strikes a good balance between performance and programmability.
- We propose a strengthening of this memory model that:
  - forbids weak behaviors, unobservable in any implementation
  - has simple fence semantics, with SC-reduction theorems
  - admits intuitive operational semantics
- The stronger model has no additional implementation cost.

See `http://plv.mpi-sws.org/sra/` for more details and Coq proofs.

Thank you!