Extending the C/C++ Memory Model with Inline Assembly

joint work with Ori Lahav, Viktor Vafeiadis, and Azalea Raad

presented by **Paulo Emílio de Vilhena**

on the 23rd of October, 2024

Overview

Problem.
Specify the semantics of *inline x86 assembly* in a *concurrent* setting with *shared memory*.

Motivation.
Inline assembly allows one to insert
snippets of *x86 assembly* in C/C++ code.

Applications of inline assembly include

- Writing *efficient code* directly in *x86 assembly*
- Access to *instructions* that *do not exist* in C/C++
- Customizing *call conventions*

Challenge.
Architecture (x86) and *source language (C/C++)* follow *incompatible* concurrency models.

Overview

Problem.
Specify the semantics of *inline x86 assembly* in a *concurrent* setting with *shared memory*.

Motivation.

Inline assembly allows one to insert snippets of *x86 assembly* in C/C++ code.

Applications of inline assembly include

- Writing *efficient code* directly in *x86 assembly*
- Access to *instructions* that *do not exist* in C/C++
- Customizing *call conventions*

```
int src = 42, dst = 0;

// non-temporal store
asm volatile (
    "movnti %[src], %[dst]"
    : [dst] "=m" (dst)
    : [src] "r"  (src)
    : "memory");

assert(dst == 42);
```

Challenge.

Architecture (x86) and *source language (C/C++)* follow *incompatible* concurrency models.

Overview

Problem.
Specify the semantics of *inline x86 assembly* in a *concurrent* setting with *shared memory*.

Motivation.

Inline assembly allows one to insert snippets of *x86 assembly* in C/C++ code.

Applications of inline assembly include

- Writing *efficient code* directly in *x86 assembly*
- Access to *instructions* that *do not exist* in C/C++
- Customizing *call conventions*

```
int src = 42, dst = 0;

// non-temporal store
asm volatile (
    "movnti %[src], %[dst]"
    : [dst] "=m" (dst)
    : [src] "r"  (src)
    : "memory");

assert(dst == 42);
```

Challenge.

Architecture (x86) and *source language (C/C++)* follow *incompatible* concurrency models.

Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.

T0	T1
x :=sc 1	a =sc y
y :=sc 1	b =sc x

na → **rlx** → **rel**, **acq** → **acqrel** → **sc**

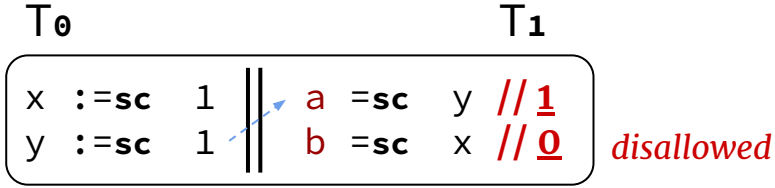
atomics

Ex86 [Raad et al.] is a model for *x86 assembly*.

T0	T1
x := 1	a = y
y := 1	b = x

Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.



disallowed

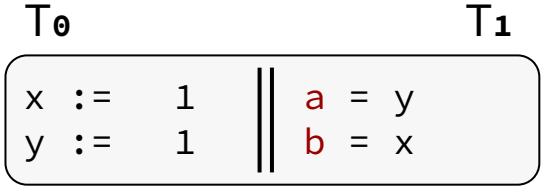
`na` → `rlx` → `rel, acq` → `acqrel` → **`sc`**

atomics

`sc` is the *strongest* mode:

$$a = 1 \implies b = 1$$

Ex86 [Raad et al.] is a model for *x86 assembly*.



Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.

T0	T1	
`x :=rlx 1`	`a =rlx y`	`// 1`
`y :=rlx 1`	`b =rlx x`	`// 0`

allowed

`na` → **`rlx`** → `rel, acq` → `acqrel` → `sc`

atomics

`rlx` is the *weakest* atomic mode:

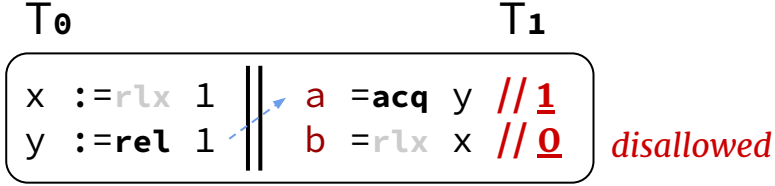
`a = 1` $\not\Rightarrow$ `b = 1`

Ex86 [Raad et al.] is a model for *x86 assembly*.

T0	T1
`x := 1`	`a = y`
`y := 1`	`b = x`

Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.



disallowed

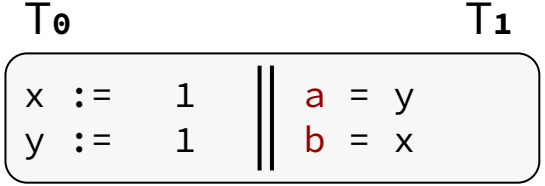
na → rlx → **rel, acq** → acqrel → sc

atomics

rel-acq pairs restore *synchronization*:

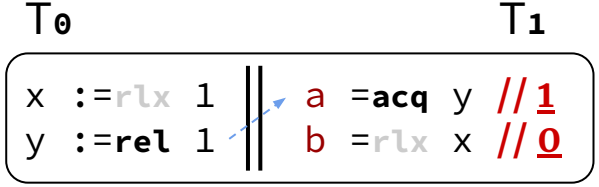
$a = 1 \implies b = 1$

Ex86 [Raad et al.] is a model for *x86 assembly*.



Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.



disallowed

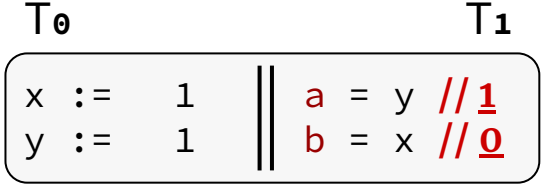
na → rlx → **rel, acq** → acqrel → sc

atomics

rel-acq pairs restore *synchronization*:

$a = 1 \implies b = 1$

Ex86 [Raad et al.] is a model for *x86 assembly*.



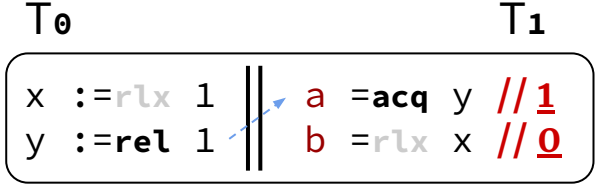
disallowed

In *x86*, *plain reads* and *writes* follow *TSO*: Only *write-read pairs* can be *reordered*.

$a = 1 \implies b = 1$

Introduction to RC11 and Ex86

RC11[Lahav et al.] is a model for *C/C++*.



disallowed

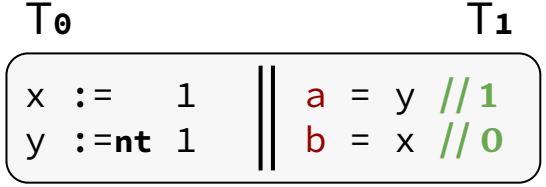
na → rlx → **rel, acq** → acqrel → sc

atomics

rel-acq pairs restore *synchronization*:

$a = 1 \implies b = 1$

Ex86 [Raad et al.] is a model for *x86 assembly*.



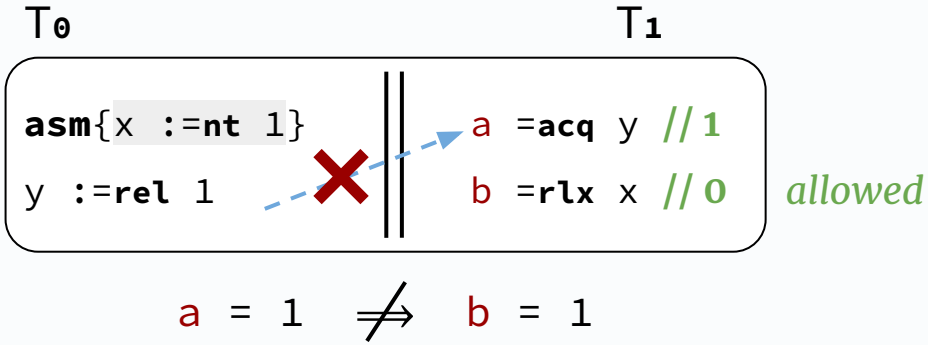
allowed

Non-temporal stores bypass *the cache*: They can *reordered* with other *writes*.

$a = 1 \not\implies b = 1$

Challenges – Non-temporal stores

Consequence to C/C++ memory model: **`nt`** stores *break* **`rel-acq`** *synchronization*!

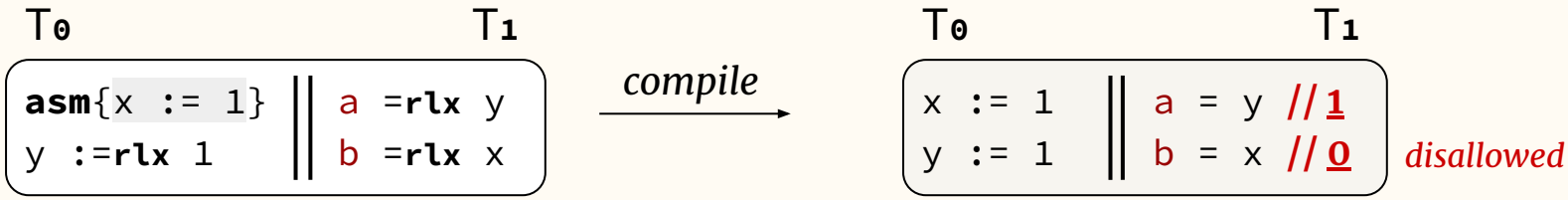


Challenges:

- *Relax RC11* in such a way that this behavior is *allowed*.
- Suggest how to *restore synchronization* (e.g. through *x86*'s *store fences*).

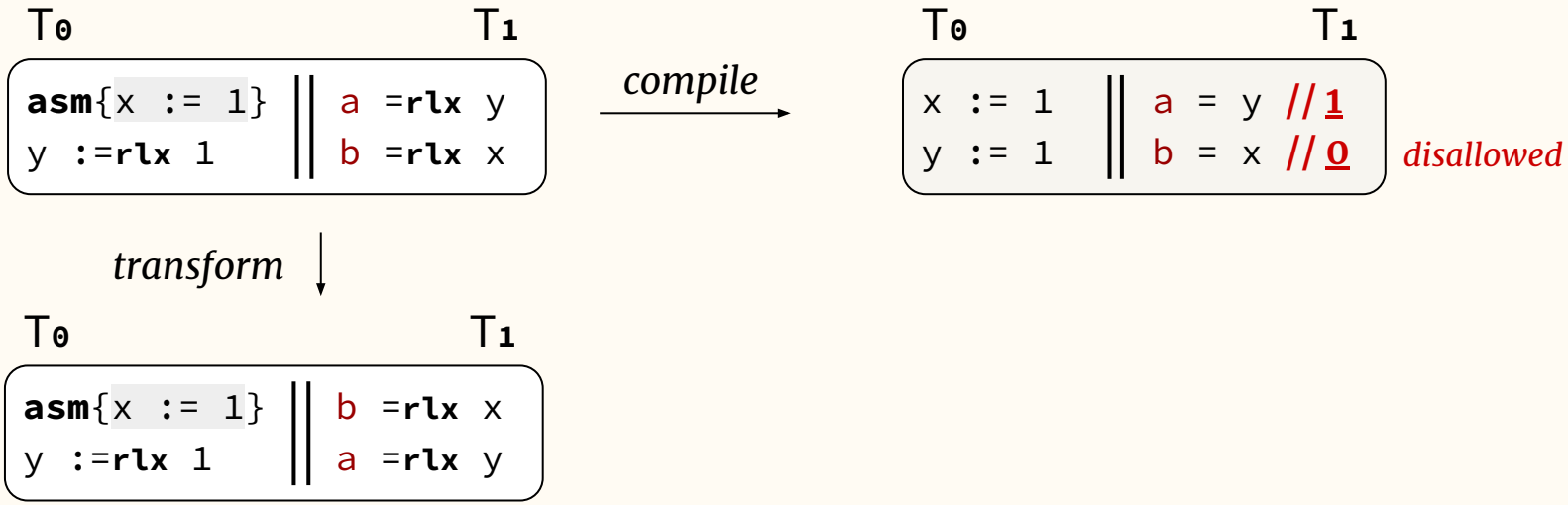
Challenges – Compiler optimizations

Compiler optimizations introduce behaviors that violate *Ex86 consistency*.



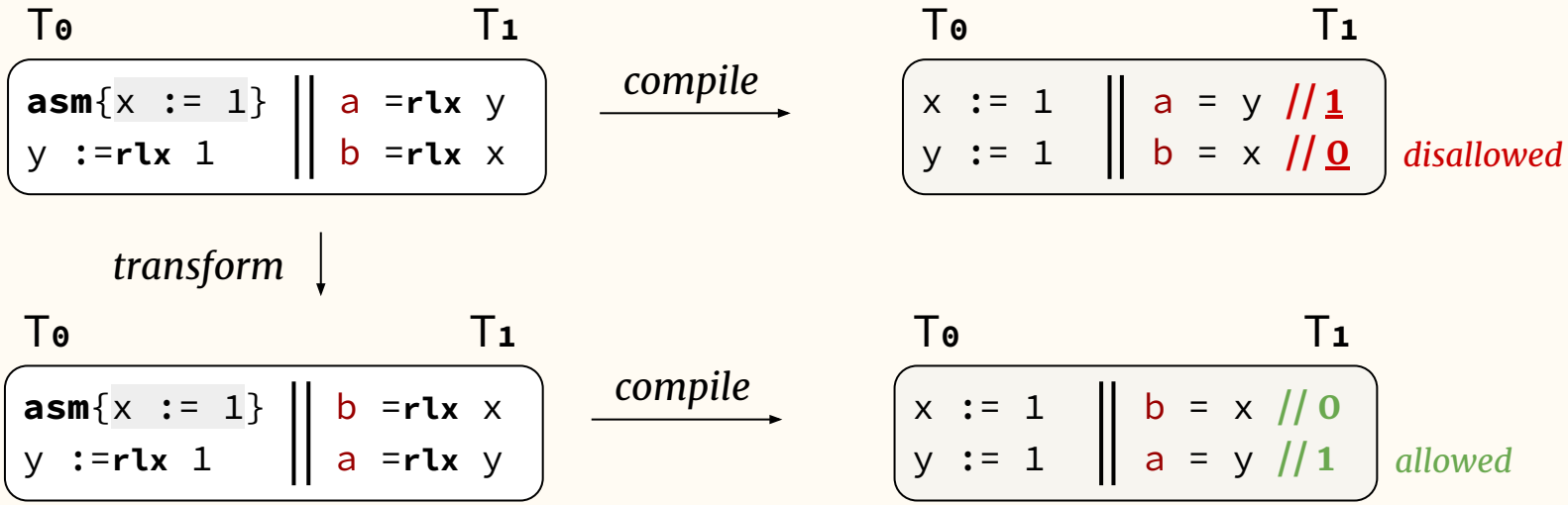
Challenges – Compiler optimizations

Compiler optimizations introduce behaviors that violate *Ex86 consistency*.



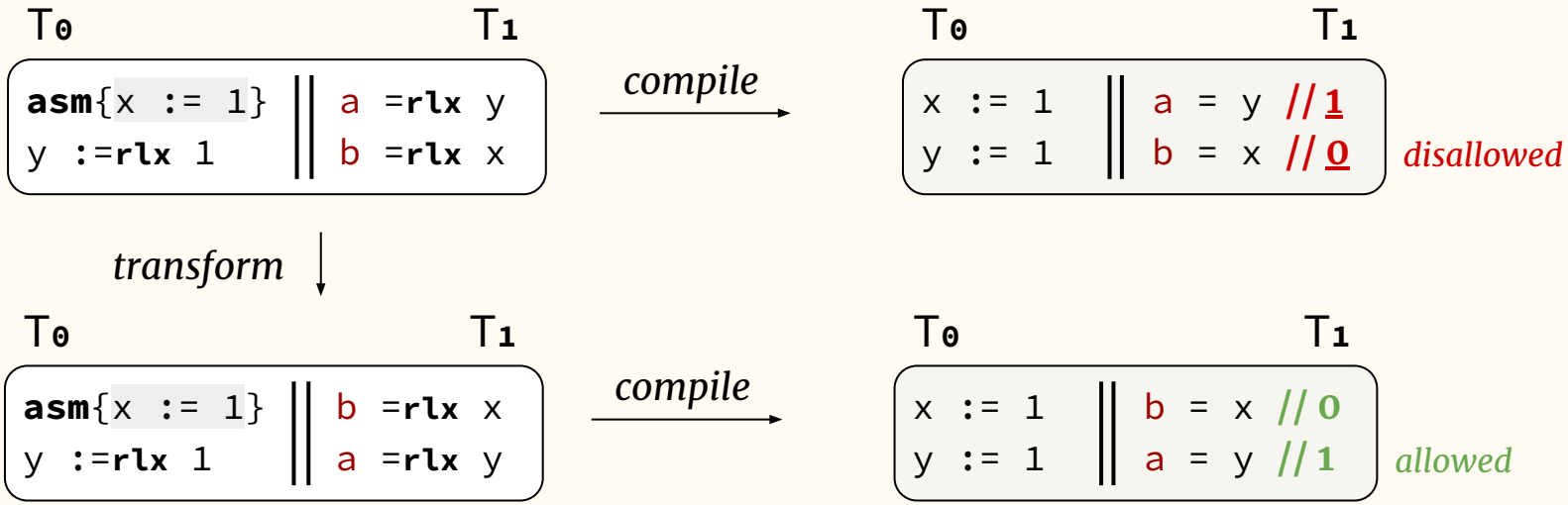
Challenges – Compiler optimizations

Compiler optimizations introduce behaviors that violate *Ex86 consistency*.



Challenges – Compiler optimizations

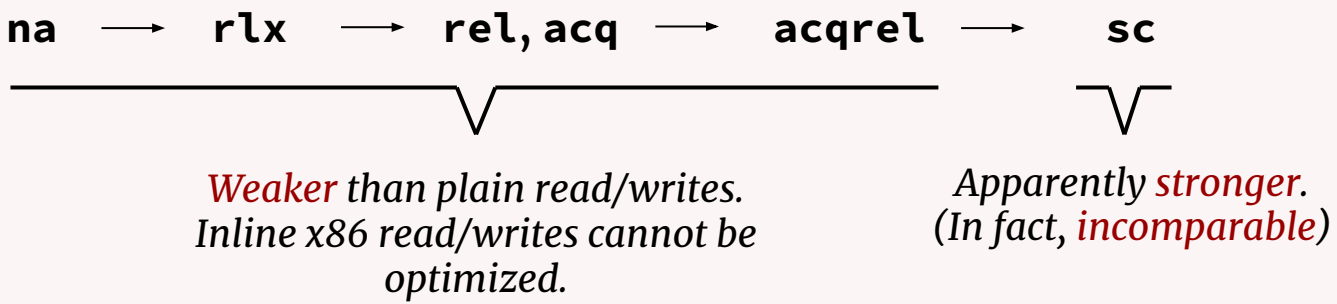
Compiler optimizations introduce behaviors that violate *Ex86 consistency*.



Challenge: Enforce *Ex86 consistency* in a way that does *not break optimizations*.

Challenges – Access modes are unfit for inline assembly

It is *not* possible to model *inline-assembly accesses* using *RC11 access modes*:



Challenges:

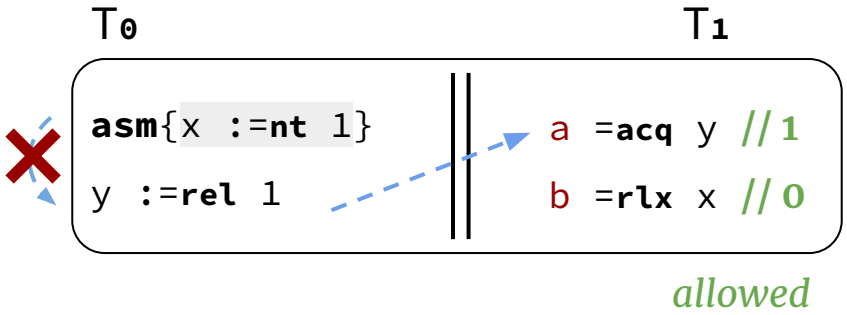
- *Invent* new *access modes* for *inline-assembly accesses*.
- *Discover* how the new access modes *relate* to the existing *RC11* ones.

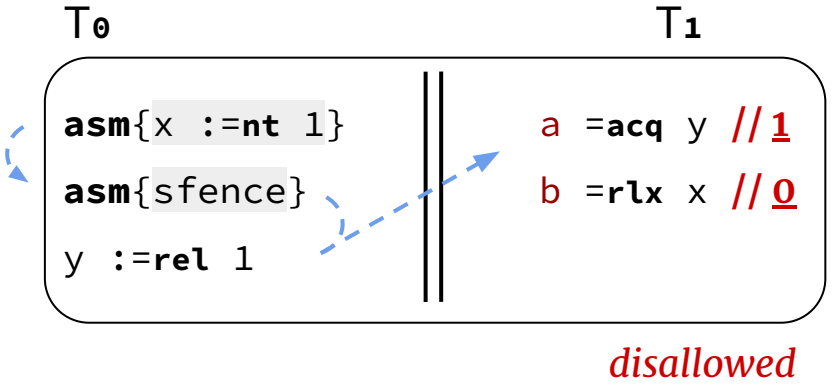
$RC11^{Ex86}$ – The extended model

We introduce ***$RC11^{Ex86}$***, an *extended model* for C/C++ with inline x86 assembly.

$RC11^{Ex86}$ handles the three aforementioned challenges:

1. *Non-temporal stores* do *not* enforce *synchronization* unless followed by a (sufficiently strong) *barrier*.





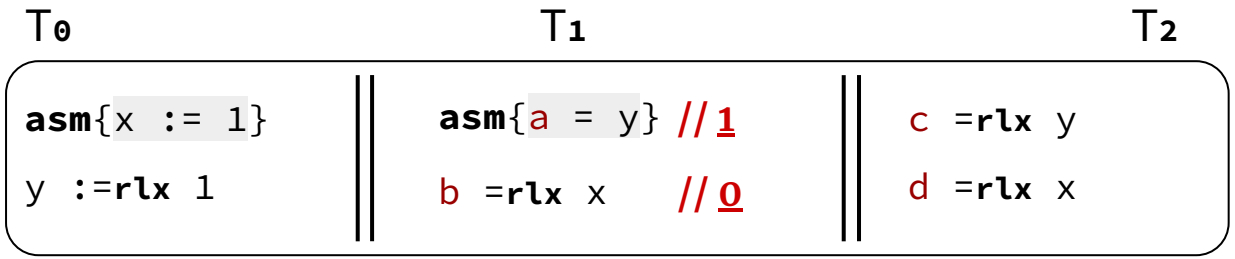
$RC11^{Ex86}$ – The extended model

We introduce ***$RC11^{Ex86}$***, an *extended model* for C/C++ with inline x86 assembly.

$RC11^{Ex86}$ handles the three aforementioned challenges:

1. *Non-temporal stores* do *not* enforce *synchronization* unless followed by a (sufficiently strong) *barrier*.

2. Threads must use *inline assembly* to abide by *Ex86 consistency*. *Compiler optimizations* can be applied to C/C++ portions of code.



disallowed

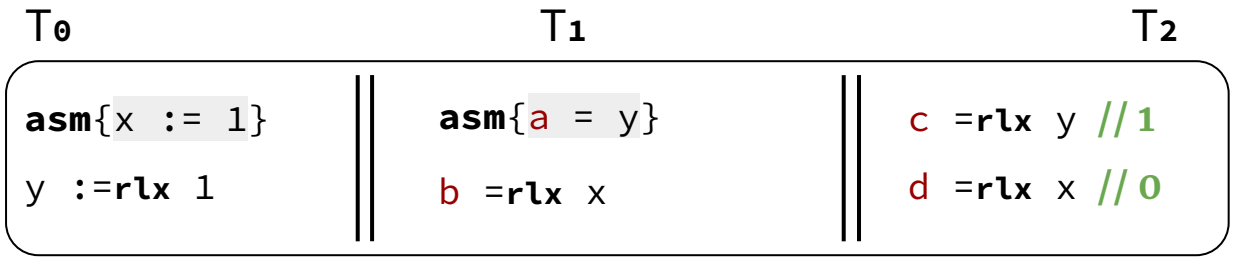
$RC11^{Ex86}$ – The extended model

We introduce ***$RC11^{Ex86}$***, an *extended model* for C/C++ with inline x86 assembly.

$RC11^{Ex86}$ handles the three aforementioned challenges:

1. *Non-temporal stores* do *not* enforce *synchronization* unless followed by a (sufficiently strong) *barrier*.

2. Threads must use *inline assembly* to abide by *Ex86 consistency*. *Compiler optimizations* can be applied to C/C++ portions of code.



allowed

$RC11^{Ex86}$ – The extended model

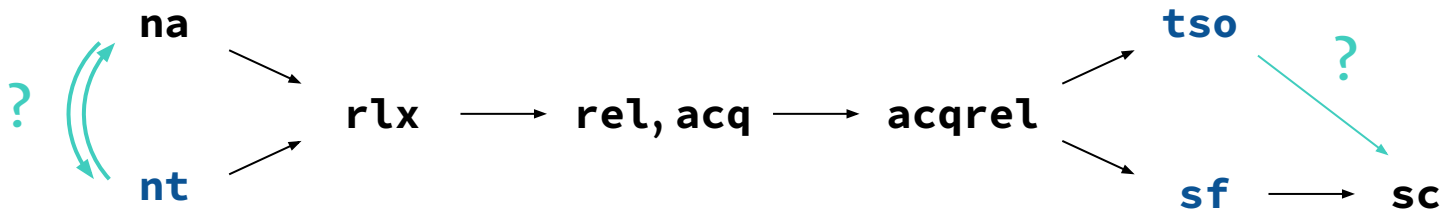
We introduce ***$RC11^{Ex86}$***, an *extended model* for C/C++ with inline x86 assembly.

$RC11^{Ex86}$ handles the three aforementioned challenges:

1. *Non-temporal stores* do *not* enforce *synchronization* unless followed by a (sufficiently strong) *barrier*.

2. Threads must use *inline assembly* to abide by *Ex86 consistency*. *Compiler optimizations* can be applied to C/C++ portions of code.

3. Introduction of *new access modes* for *non-temporal stores* (**`nt`**), *plain reads/writes* (**`tso`**), and *store fences* (**`sf`**).



$RC11^{Ex86}$ – The extended model

Non-temporal (**nt**) *stores* can be *weaker* than *non-atomic* (**na**) *accesses*.

T0	T1
asm{x :=nt 1}	a =acq y // 1
y :=rel 1	if (a == 1)
	b =rlx x // 0

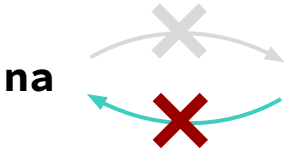
allowed

T0	T1
x :=na 1	a =acq y // 1
y :=rel 1	if (a == 1)
	b =rlx x // 0

disallowed

$RC11^{Ex86}$ – The extended model

Conversely, *non-atomic* (**na**) *accesses* can be *weaker* than *non-temporal* (**nt**) *stores*.



T_0	T_1
`x :=na 1`	`a =rlx x // 42`

allowed

T_0	T_1
`asm{x :=nt 1}`	`a =rlx x // 42`

disallowed

Why *races* on *inline-assembly accesses* are *not UB*?

There are *multiple reasons*, we cite *two*:

- Programs fully written in *inline assembly* would *not* abide by *x86 consistency*.
- *Inline assembly* is *not optimized* like **na** *accesses*.

$RC11^{Ex86}$ – The extended model

The semantics of **`sc`** *accesses* can be *weaker* than *TSO*.

T0	T1
asm{x := 1}	**asm**{a = y} // **1**
y :=**rlx** 1	b =**rlx** x // **0**

disallowed

T0	T1
x :=**sc** 1	a =**sc** y // 1
y :=**rlx** 1	b =**rlx** x // 0

allowed

$RC11^{Ex86}$ – Properties

Extension of RC11. Programs *without inline assembly* have *RC11 semantics*:

$$P \in \textbf{RC11} \quad \Rightarrow \quad [\![P]\!]_{RC11^{Ex86}} = [\![P]\!]_{RC11}$$

Extension of Ex86. Programs *fully written in inline assembly* have *Ex86 semantics*:

$$P \in \textbf{Ex86} \quad \Rightarrow \quad [\![\texttt{asm}\{P\}]\!]_{RC11^{Ex86}} = [\![P]\!]_{Ex86}$$

Data-race freedom. *Data-race-free programs* have *SC semantics*:

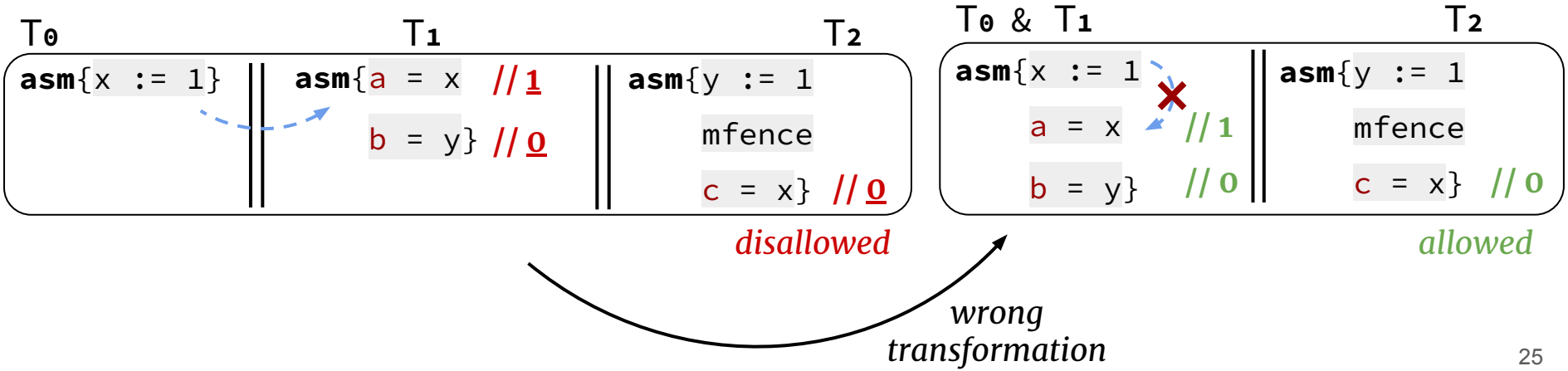
$$P \text{ has } \texttt{sc}\text{-only races} \quad \Rightarrow \quad [\![P]\!]_{RC11^{Ex86}} = [\![P]\!]_{SC}$$

$RC11^{Ex86}$ – Properties

We show *correctness of compilation* with respect to $RC11^{Ex86}$ and *Ex86*.

Moreover, *sound compiler optimizations* in *RC11* are also *sound* in $RC11^{Ex86}$;
There is however one caveat: In $RC11^{Ex86}$, *sequentialization* is *not sound* in general.

In *x86*, reading an *external* write enforces synchronization,
whereas reading an *internal* write does *not*.



Conclusion

Inline assembly is an important tool that is *not* handled by the *RC11 model*.

Many *challenges* exist.

- *Non-temporal* stores *break* **`rel-acq`** *synchronization*.
- *Ex86-consistency* is *incompatible* with many *compiler optimizations*.
- *Inline-assembly* accesses *cannot* be modeled with *RC11 access modes*.

We introduce ***RC11***^{Ex86}, an *extended model* for C/C++ with inline x86 assembly.

- One can *restore* **`rel-acq`** *synchronization* through barriers, such as *store fences*.
- The *scope* of *Ex86-consistency* is *limited* to threads with *inline assembly*.
  (*Compiler optimizations* can be applied to C/C++ portions of code.)
- *Inline-assembly accesses* are modeled using *new accesses modes*.

The ***RC11***^{Ex86} model enjoys many *properties*

- *Extension* of *RC11* and *Ex86*
- *Data-race freedom*
- *Correctness* of *compilation* and *compiler optimizations*

Questions