

# CSE113: Parallel Programming

May 13, 2021

- **Topic:** Finish DOALL & Memory Consistency
  - DOALL schedules in OpenMP
  - Sequential Consistency
  - Total Store Order
  - Relaxed memory models

# Announcements

- HW 3 is out:
  - ask questions on Piazza!
  - Thanks to those who are having good discussions!
  - Due date Friday May 21
- Midterm grades are released today by midnight
  - Please ask questions within two weeks
- Guest lecture in 1 week!
  - Message passing concurrency and testing GPU compilers

# Announcements

- Thanks for those who find typos; it helps improve the slides!

# Quiz

# Quiz

- Discuss Answers

# Schedule

- Parallel schedules in OpenMP
- Memory consistency models:
  - Total store order
  - Relaxed memory consistency
  - Examples

# Schedule

- **Parallel schedules in OpenMP**
- Memory consistency models:
  - Total store order
  - Relaxed memory consistency
  - Examples

# Parallelize DOALL Loops with OpenMP

- We studied DOALL loops last week:
  - What is a DOALL loop?



# Parallelize DOALL Loops with OpenMP

- We studied DOALL loops last week:
  - What is a DOALL loop?

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + c[i];  
}
```

# Parallelize DOALL Loops with OpenMP

- We studied DOALL loops last week:
  - What is a DOALL loop?

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + c[i];  
}
```

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + c[i+1];  
}
```

# Parallelize DOALL Loops with OpenMP

- We studied DOALL loops last week:
  - What is a DOALL loop?

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + c[i];  
}
```

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + c[i+1];  
}
```

```
for (int i = 0; i < SIZE; i++) {  
    a[i] = b[i] + a[i+1];  
}
```

# Parallelize DOALL Loops with OpenMP

- We studied DOALL loops last week:
  - What is a DOALL loop?
- We talked about very complicated ways to implement parallelism over these loops
- But what if I was to tell you that there was an easier way?

# OpenMP<sup>®</sup>

- Built on top of C++ and Fortran
- First released in 1997 (way before C++11 threads!)
  - Still used widely today, esp. in HPC and ML
- consists of:
  - pragma based compiler directives
  - runtime

# OpenMP<sup>®</sup>

- Many features
  - atomic RMWs
  - thread spawn and join
  - shared memory
- Perhaps best known for supporting parallel DOALL loops

# Why is it so popular?

```
for (int i = 0; i < SIZE; i++) {  
    c[i] = a[i] + b[i];  
}
```

```
#pragma omp parallel for  
for (int i = 0; i < SIZE; i++) {  
    c[i] = a[i] + b[i];  
}
```

parallelize a loop with one line!

code works with or without compiler support!

Have to also add compile line: `-fopenmp`

Lets try it out



# Customization in OpenMP pragmas

```
#pragma omp parallel for num_threads(N)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

Number of threads is great for running scaling experiments or reducing the load on the machine

By default OpenMP will try to saturate your machine

# Customization in OpenMP pragmas

```
#pragma omp parallel for schedule(S,C)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

Specify the parallel schedule. There are several options:

static - evenly chunks iterations across cores

dynamic - workstealing

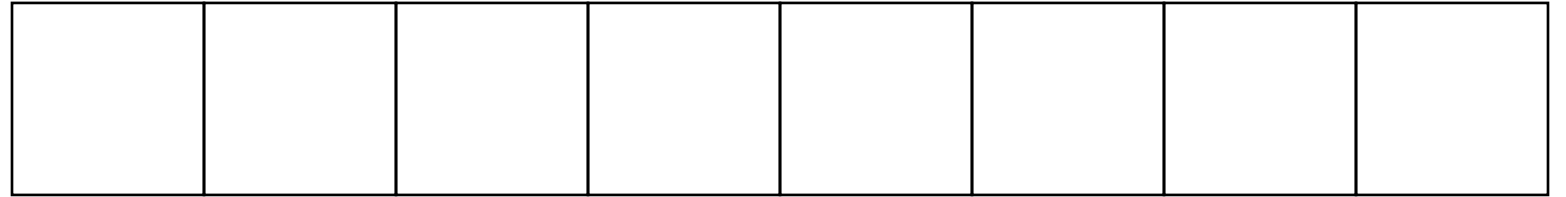
others - we won't get into them in the class

Can specify the chunk size with C

By default OpenMP will select a good chunk size based on your architecture!

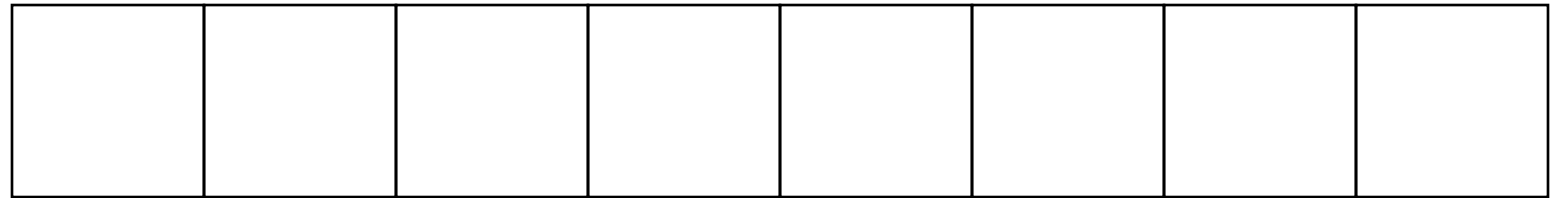
```
#pragma omp parallel for num_threads(N) schedule(S,C)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

array a



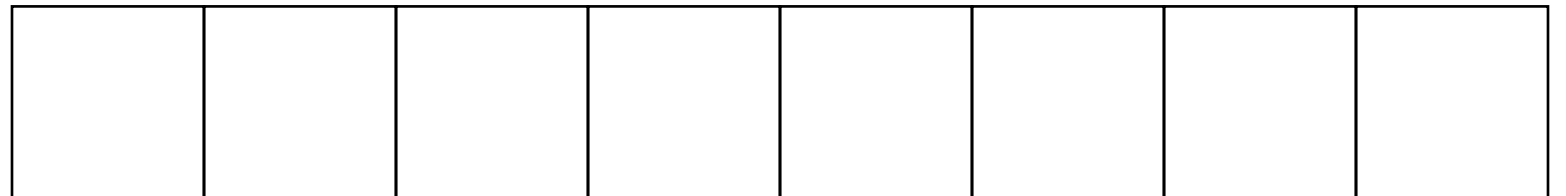
+ + + + + + + +

array b



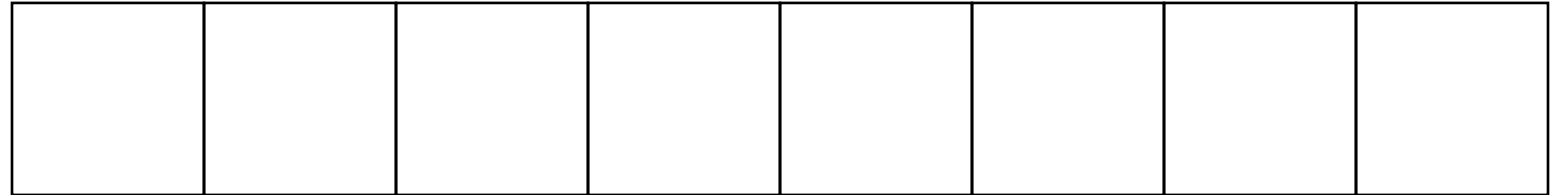
= = = = = = = =

array c



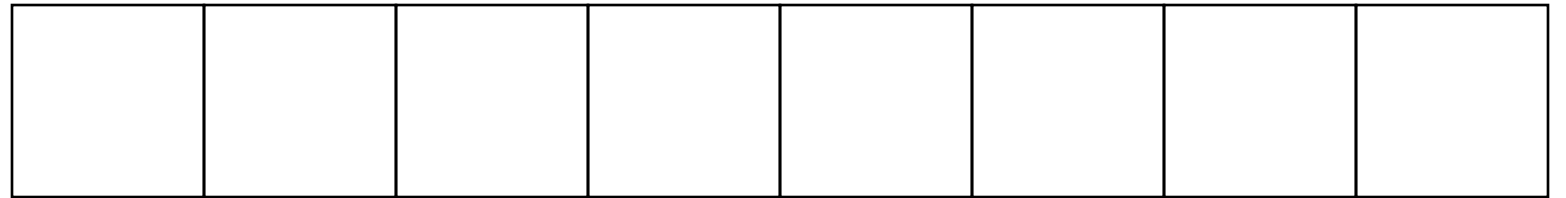
```
#pragma omp parallel for num_threads(4) schedule(S,C)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

array a



+ + + + + + + +

array b



= = = = = = = =

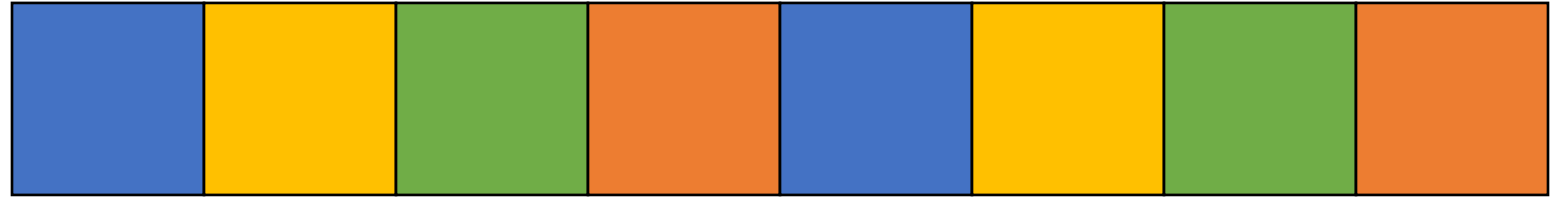
array c



- Thread 0 - Blue
- Thread 1 - Yellow
- Thread 2 - Green
- Thread 3 - Orange

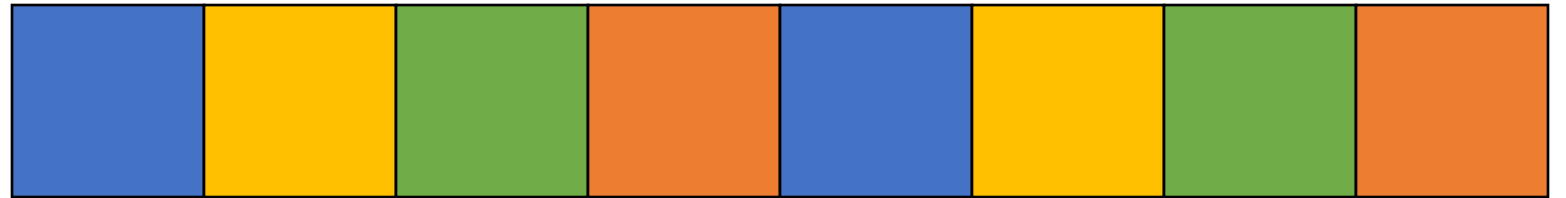
```
#pragma omp parallel for num_threads(4) schedule(static,1)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

array a



+ + + + + + + +

array b



= = = = = = = =

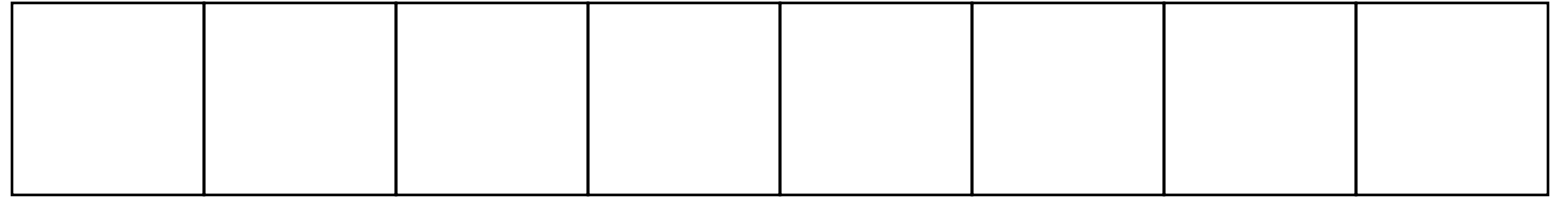
array c



Thread 0 - Blue  
Thread 1 - Yellow  
Thread 2 - Green  
Thread 3 - Orange

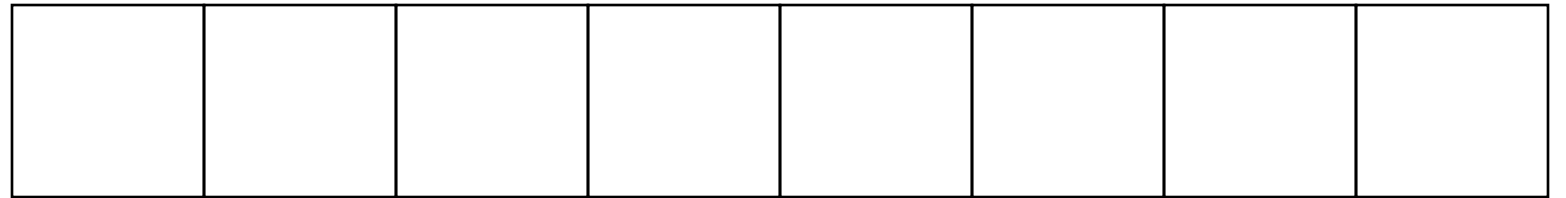
```
#pragma omp parallel for num_threads(4) schedule(static,2)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

array a



+ + + + + + + +

array b



= = = = = = = =

array c



Thread 0 - Blue  
Thread 1 - Yellow  
Thread 2 - Green  
Thread 3 - Orange

```
#pragma omp parallel for num_threads(4) schedule(static,2)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

array a



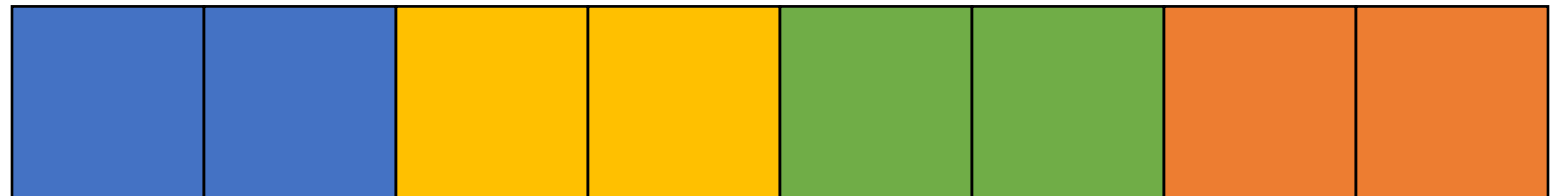
+ + + + + + + +

array b



= = = = = = = =

array c



Thread 0 - Blue  
Thread 1 - Yellow  
Thread 2 - Green  
Thread 3 - Orange

# What about workstealing?

```
#pragma omp parallel for num_threads(4) schedule(dynamic)
for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
}
```

*what happens when we run this?*



# What about workstealing?

*What about a loop that has load imbalance? Recall this loop from the previous lecture*

```
#pragma omp parallel for num_threads(2) schedule(dynamic)
for (x = 0; x < SIZE; x++) {
    for (y = x; y < SIZE; y++) {
        a[x,y] = b[x,y] + c[x,y];
    }
}
```

Inner loop does a variable amount of work depending on the outer loop iteration

# OpenMP takeaways

- Great for DOALL loops!
  - Rapid experimentation for different schedules and parameters
- Dynamic schedules are expensive: use with caution
- Specification includes:
  - RMWs
  - Mutexes
- Widely used in HPC community

# Schedule

- Parallel schedules in OpenMP
- **Memory consistency models:**
  - Total store order
  - Relaxed memory consistency
  - Examples

# Memory Consistency

# Memory Consistency

- We have been very strict about using atomic types in this class
  - and the methods (.load and .store)
  - why?
- Architectures do very strange things with memory loads and stores
- Compilers do too (but we won't talk too much about them today)
- C++ gives us sequential consistency if we use atomic types and operations
- What do we remember sequential consistency from?

# Sequential consistency for atomic memory

- Let's play our favorite game:

Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```



Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```

*Is it possible for*

`t0 == 0 and t1 == 1`



Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

```
x.store(1);
```

```
y.store(1);
```



Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```

*Is it possible for*

`t0 == 0 and t1 == 1`

```
int t0 = y.load();
```

```
int t1 = x.load();
```

Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

```
int t0 = y.load();
```

```
x.store(1);
```

```
y.store(1);
```

```
int t1 = x.load();
```

Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```

*Is it possible for*  
t0 == 0 and t1 == 1

yes!

Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

```
x.store(1);
```

```
y.store(1);
```



Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```

*Is it possible for*

`t0 == 1 and t1 == 0`

```
int t0 = y.load();
```

```
int t1 = x.load();
```

Global variable:

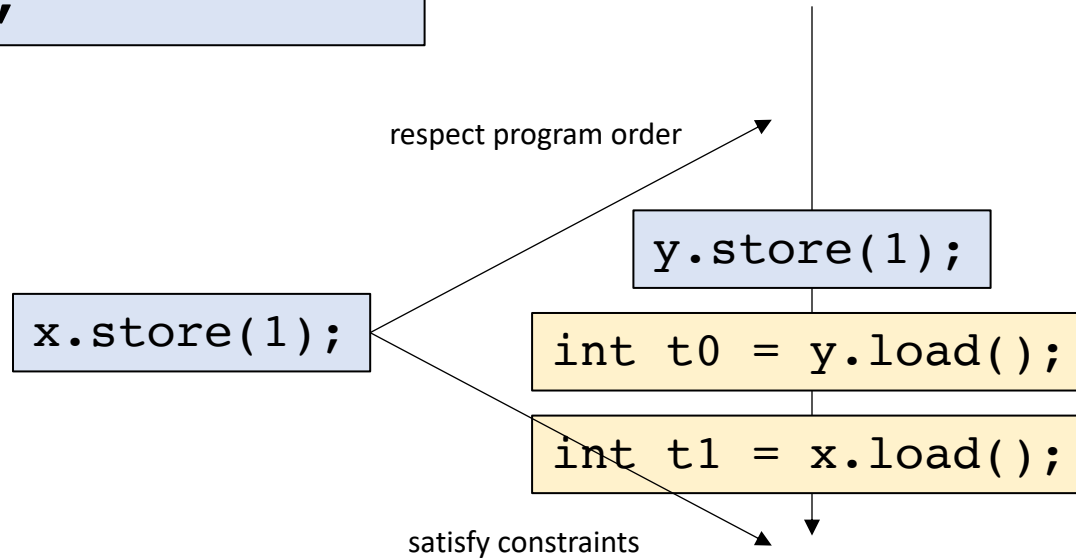
```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
y.store(1);
```

Thread 1:

```
int t0 = y.load();  
int t1 = x.load();
```



*Is it possible for*

`t0 == 1 and t1 == 0`

Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

```
x.store(1);  
int t0 = y.load();
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
y.store(1);  
int t1 = x.load();
```



Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
x.store(1);  
int t0 = y.load();
```

```
x.store(1);
```

```
int t0 = y.load();
```

Thread 1:

```
y.store(1);  
int t1 = x.load();
```

```
y.store(1);
```

```
int t1 = x.load();
```

Global variable:

```
atomic_int x(0);  
atomic_int y(0);
```

Thread 0:

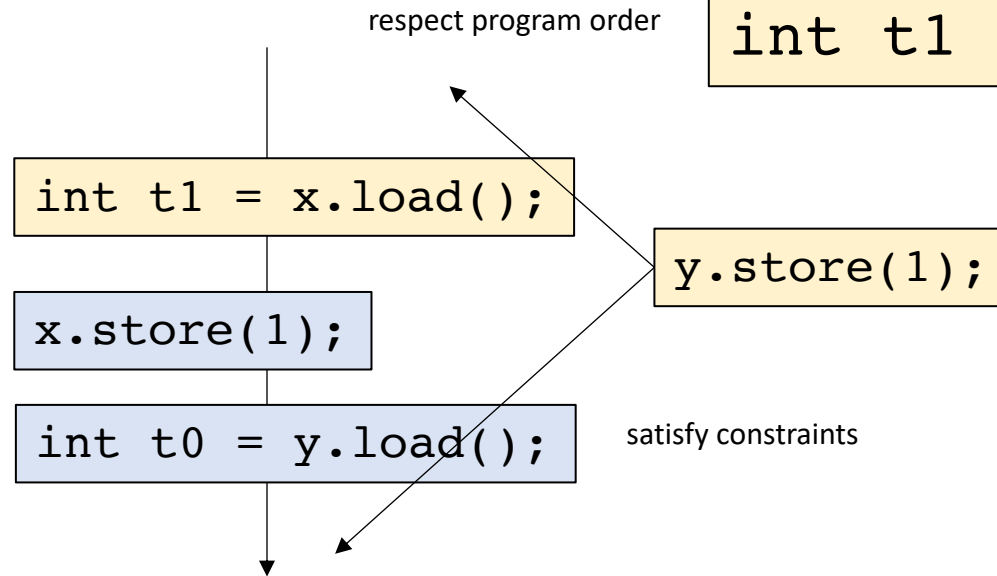
```
x.store(1);  
int t0 = y.load();
```

Another test

Can  $t0 == t1 == 0$ ?

Thread 1:

```
y.store(1);  
int t1 = x.load();
```



# C++

- Plain atomic accesses are documented to be sequentially consistent (SC)
- Why wasn't SC very good for concurrent data structures?
  - Compossibility: two objects that are SC might not be SC when used together
  - Programs contain only 1 shared memory though; no reason to compose different main memories.



# Schedule

- Parallel schedules in OpenMP
- Memory consistency models:
  - **Total store order**
  - Relaxed memory consistency
  - Examples

# What about ISAs?

- Remember, it is important for us to understand how our code executes on the architecture to write high performing programs
- Lets think about x86
  - Instructions:
  - `MOV %t0 [x]` - loads the value at x to register t0
  - `MOV [y] 1` - stores the value 1 to memory location y

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

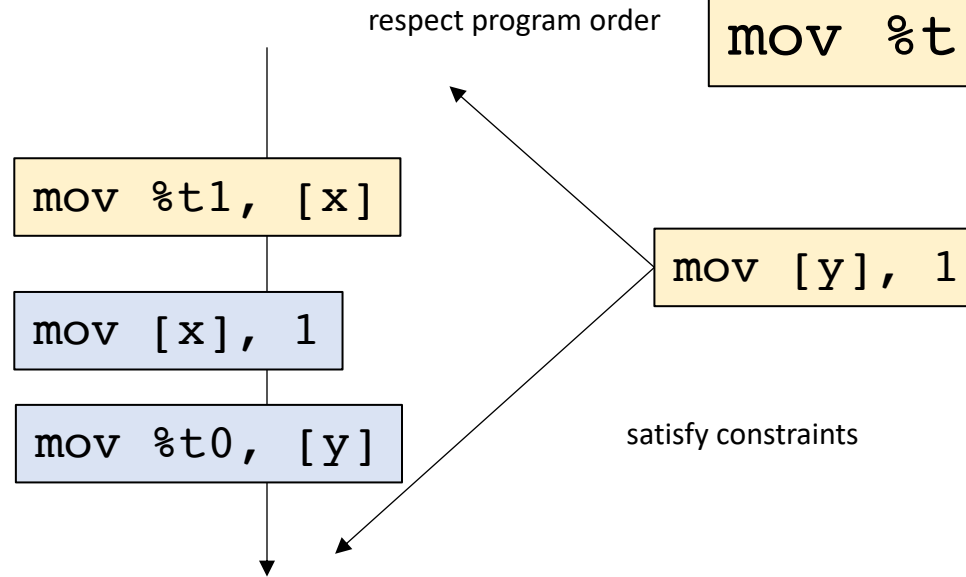
```
mov [x], 1  
mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



This is great for C++!

What about this test in x86?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

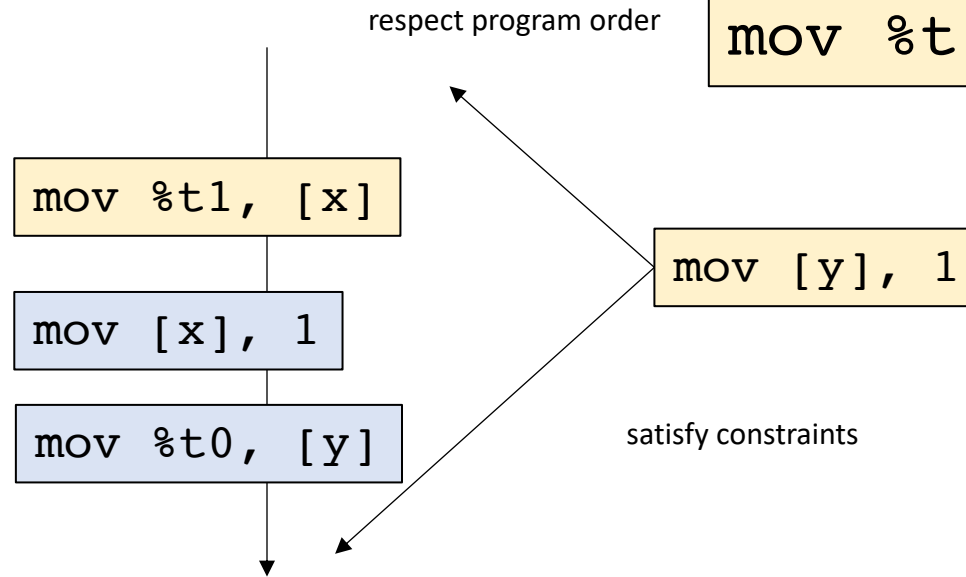
```
mov [x], 1  
mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



shouldn't be allowed under sequential consistency!

This is great for C++!  
What about this test in x86?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

*But if we run this program on hardware:*

*We would see the condition satisfied!*

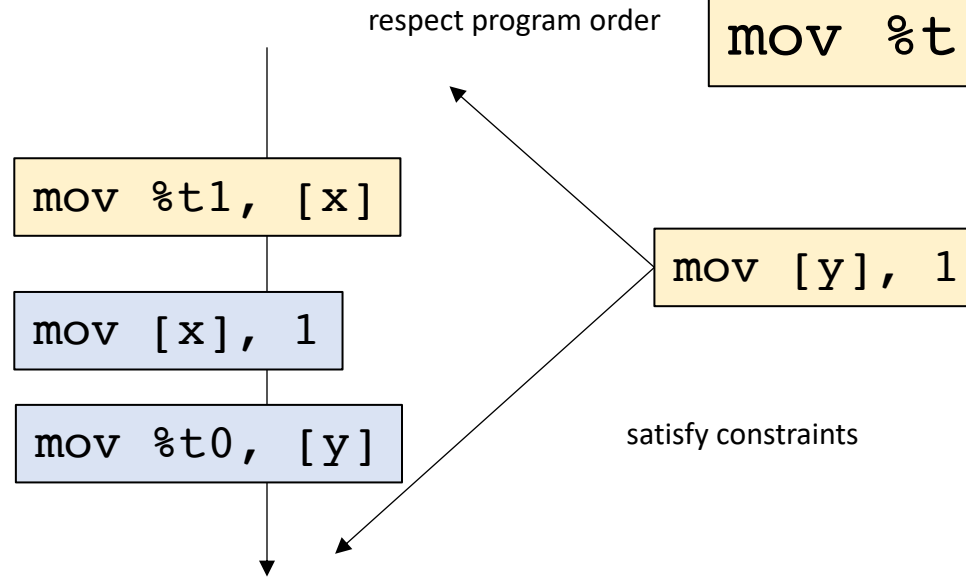
*What is going on?!*

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



shouldn't be allowed under sequential consistency!

This is great for C++!

What about this test in x86?

Thread 0:

```
mov [x], 1
```

```
mov %t0, [y]
```

Core 0

Thread 1:

```
mov [y], 1
```

```
mov %t1, [x]
```

Core 1

x:0

y:0

Main Memory

Thread 0:

```
mov %t0, [y]
```

Core 0

```
mov [x], 1
```

execute first instruction  
what happens to the stores?

Thread 1:

```
mov %t1, [x]
```

Core 1

```
mov [y], 1
```

x:0

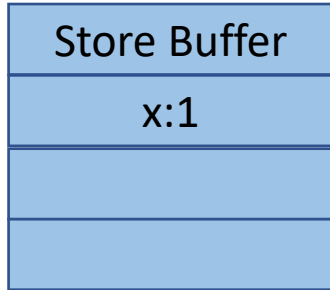
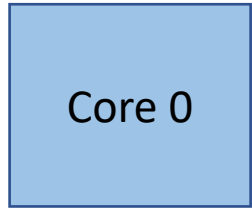
y:0

Main Memory



Thread 0:

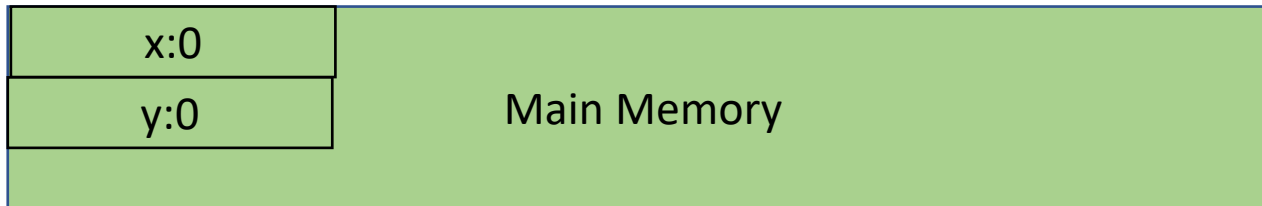
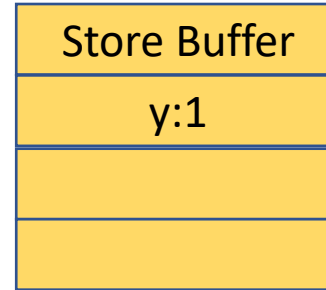
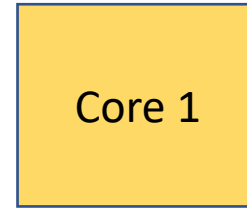
```
mov %t0, [y]
```



X86 cores contain a store buffer; holds stores before going to main memory

Thread 1:

```
mov %t1, [x]
```



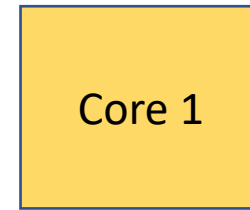
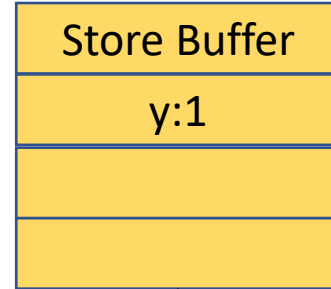
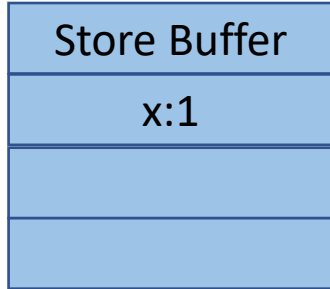
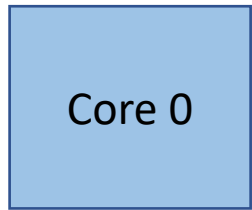
Thread 0:

```
mov %t0, [y]
```

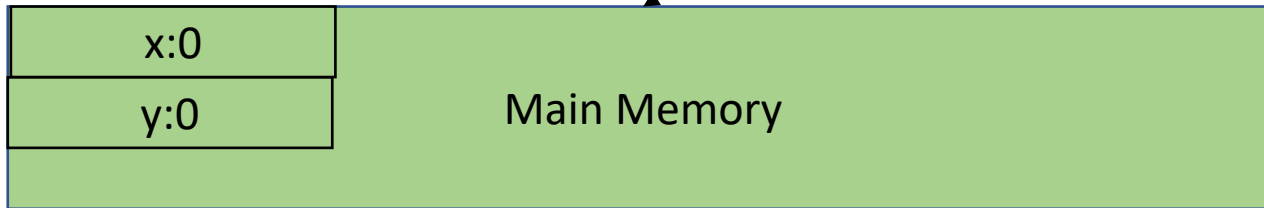
Thread 1:

```
mov %t1, [x]
```

X86 cores contain a store buffer; holds stores before going to main memory

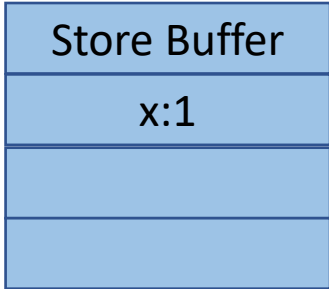
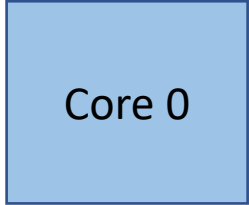


eventually they flush to main memory



Thread 0:

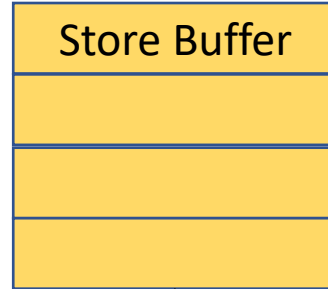
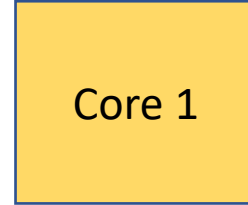
```
mov %t0, [y]
```



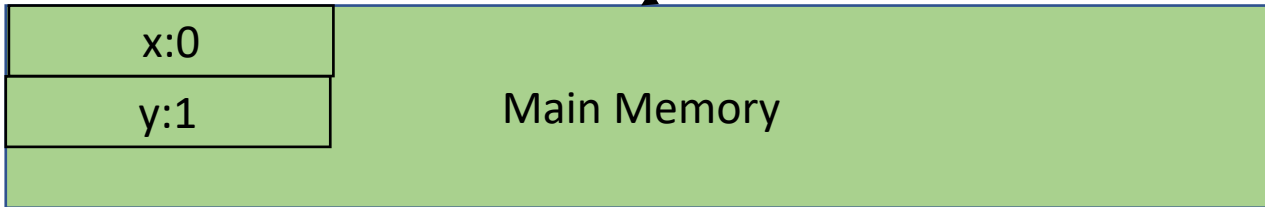
X86 cores contain a store buffer; holds stores before going to main memory

Thread 1:

```
mov %t1, [x]
```



eventually they flush to main memory



Thread 0:

```
mov [x], 1
```

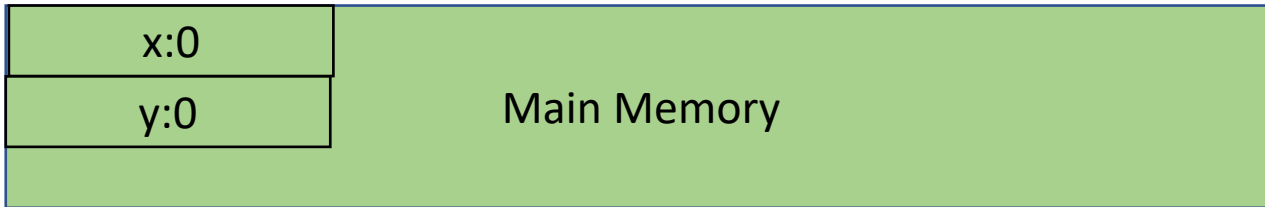
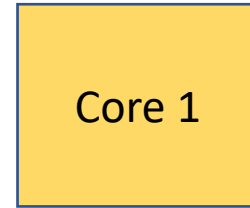
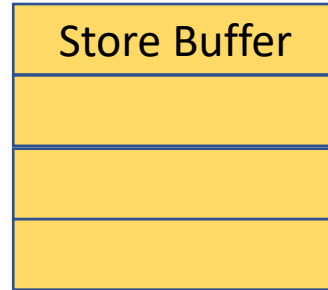
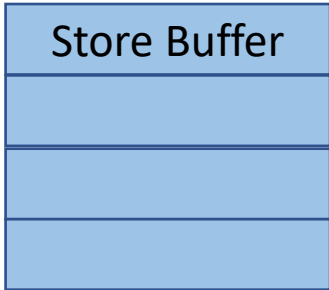
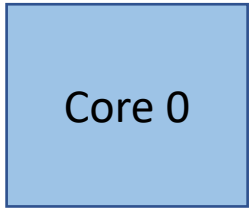
```
mov %t0, [y]
```

Thread 1:

```
mov [y], 1
```

```
mov %t1, [x]
```

rewind



Thread 0:

mov %t0, [y]

Core 0

mov [x], 1

Store Buffer

Thread 1:

mov %t1, [x]

Store Buffer

Core 1

mov [y], 1

execute first instruction

x:0

y:0

Main Memory

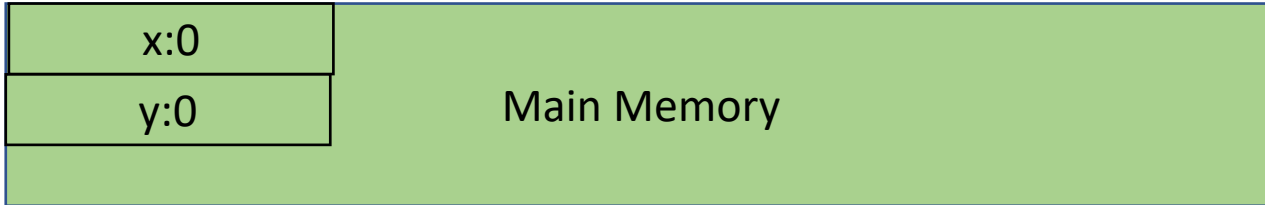
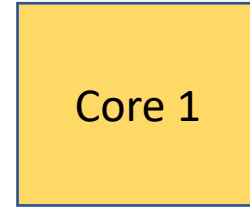
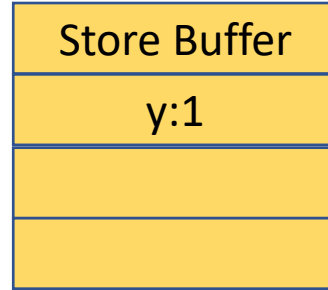
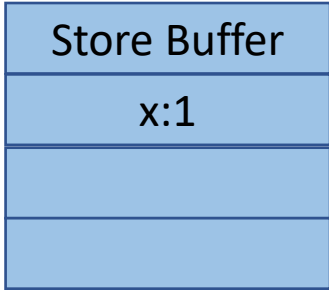
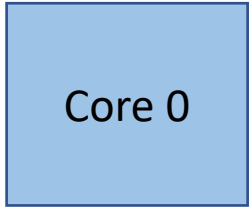
Thread 0:

Thread 1:

mov %t0, [y]

values get stored in SB

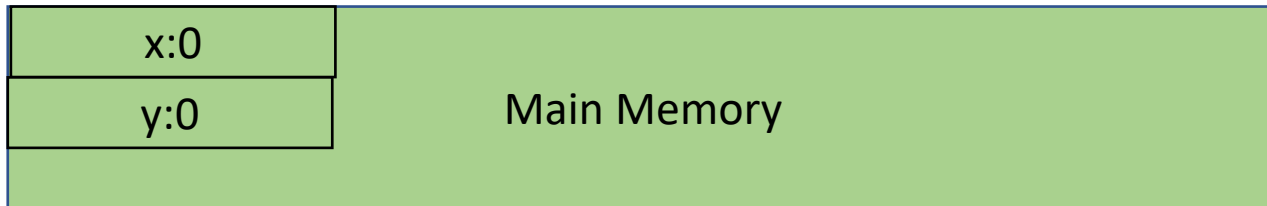
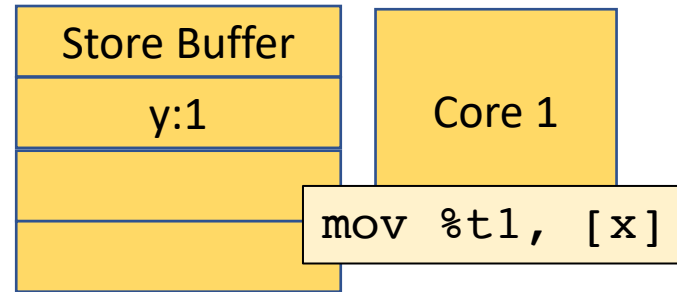
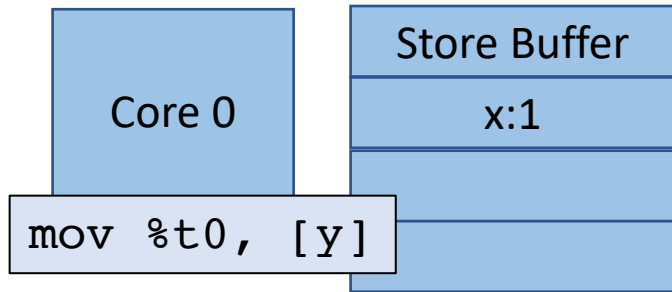
mov %t1, [x]



Thread 0:

Thread 1:

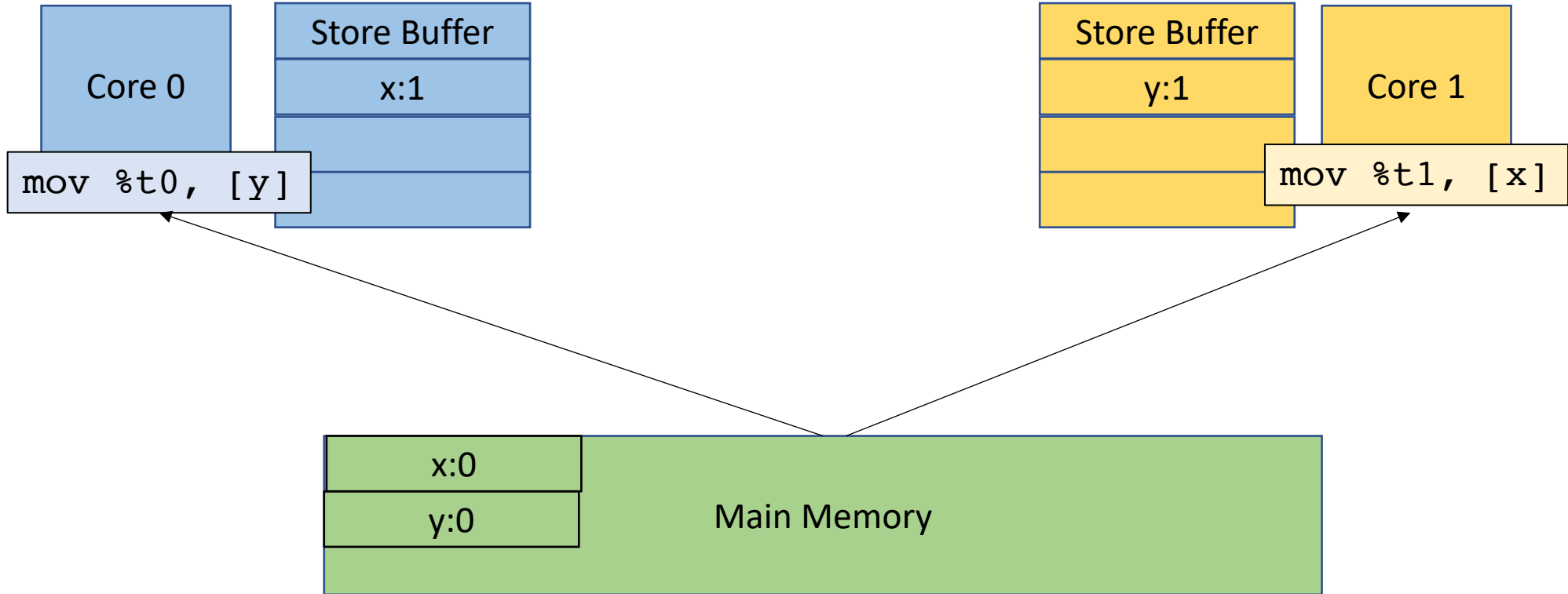
Execute next instruction



Thread 0:

Thread 1:

Values get loaded from memory

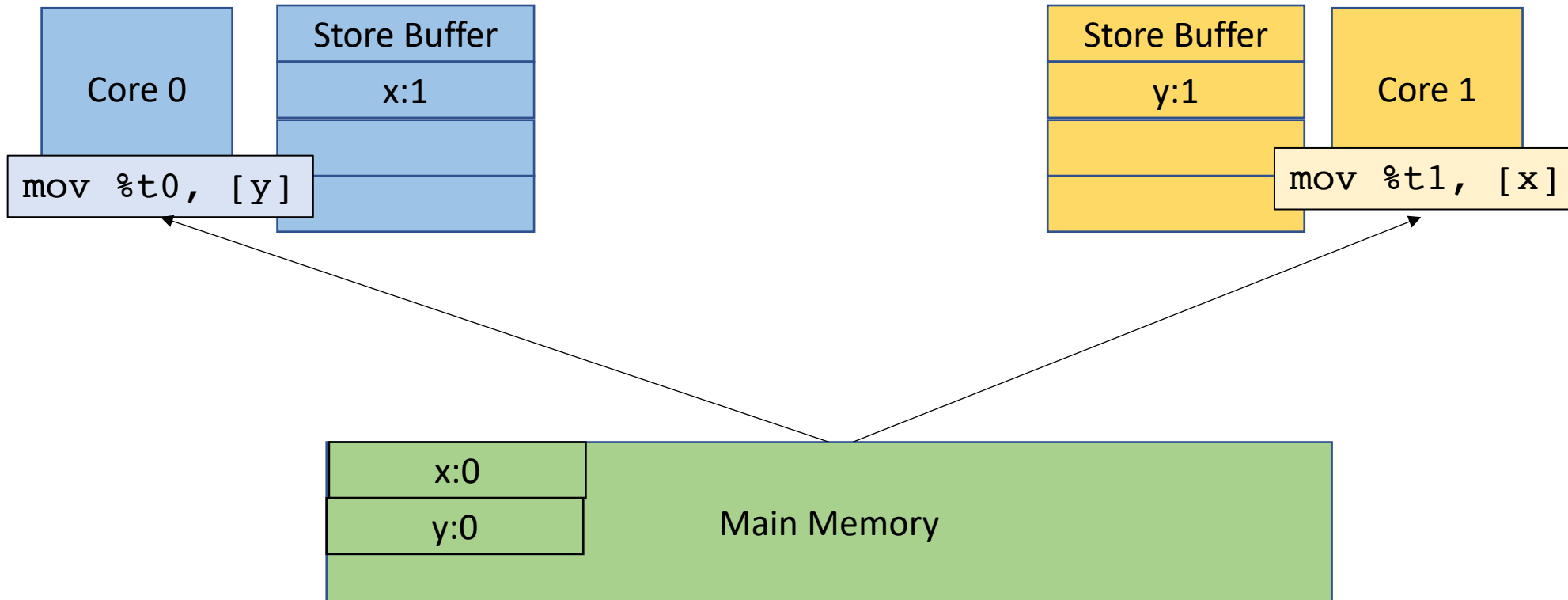




Thread 0:

Thread 1:

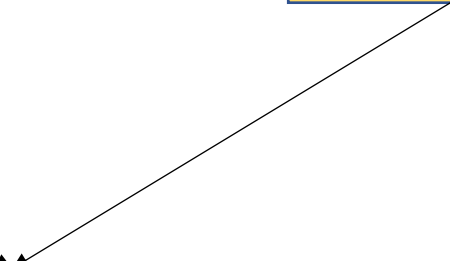
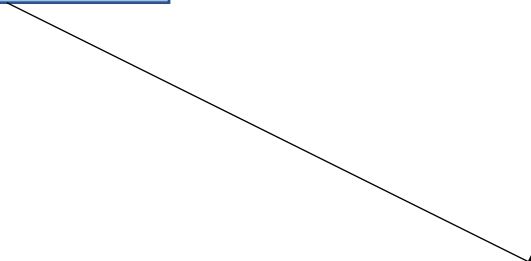
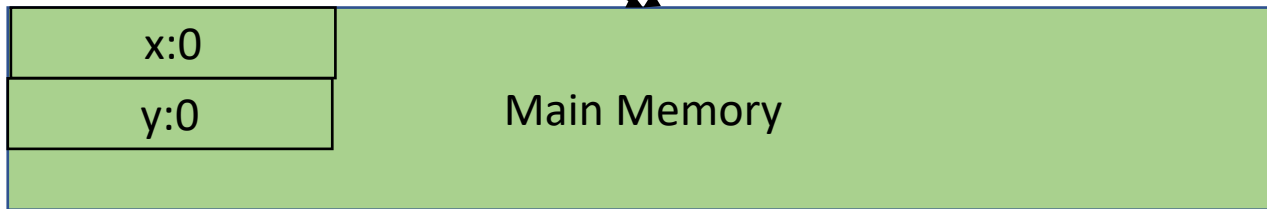
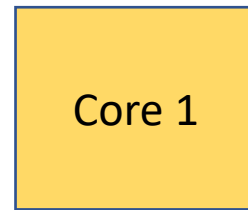
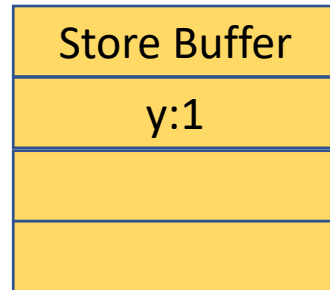
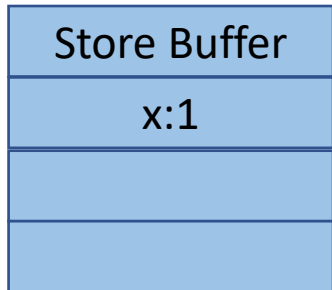
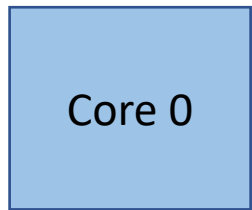
we see `t0 == t1 == 0!`



Thread 0:

Thread 1:

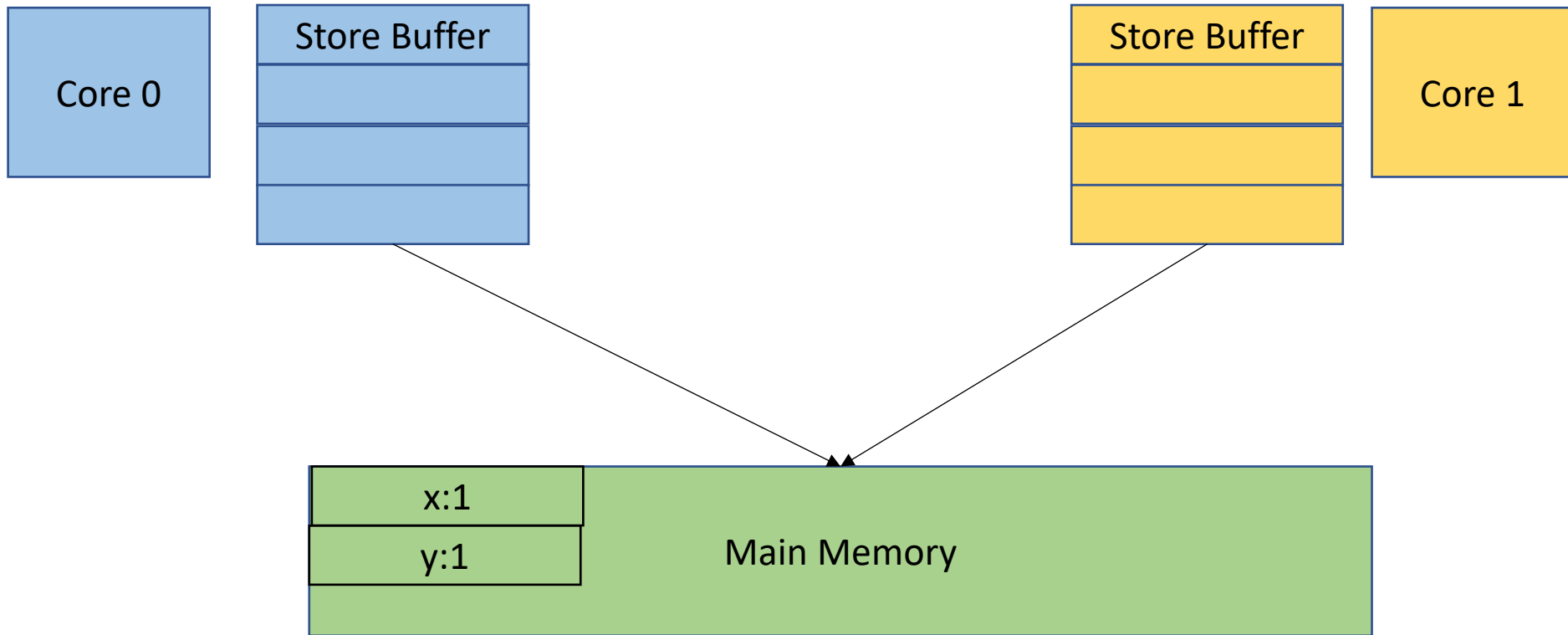
Store buffers are drained eventually



Thread 0:

Thread 1:

Store buffers are drained eventually  
but we've already done our loads



# Our first relaxed memory execution!

- also known as weak memory behaviors
- An execution that is NOT allowed by sequential consistency
- A memory model that allows relaxed memory executions is known as a relaxed memory model
  - X86 has a relaxed memory model due to store buffering
  - If you restrict yourself to use only default atomic operations, C++ has does NOT have a weak memory model

# Litmus tests

- Small concurrent programs that check for relaxed memory behaviors
- Vendors have a long history of under documented memory consistency models
- Academics have empirically explored the memory models
  - Many vendors have unofficially endorsed academic models
  - X86 behaviors were documented by researchers before Intel!

# Litmus tests

This test is called “store buffering”

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```

Another test

Can  $t0 == t1 == 0$ ?

# Restoring sequential consistency

- It is typical that relaxed memory models provide special instructions which can be used to disallow weak behaviors.
- These instructions are called Fences
- The X86 fence is called `mfence`. It flushes the store buffer.

Thread 0:

```
mov [x], 1
```

```
mfence
```

```
mov %t0, [y]
```

Core 0

Store Buffer

Thread 1:

```
mov [y], 1
```

```
mfence
```

```
mov %t1, [x]
```

Core 1

Store Buffer

x:0

y:0

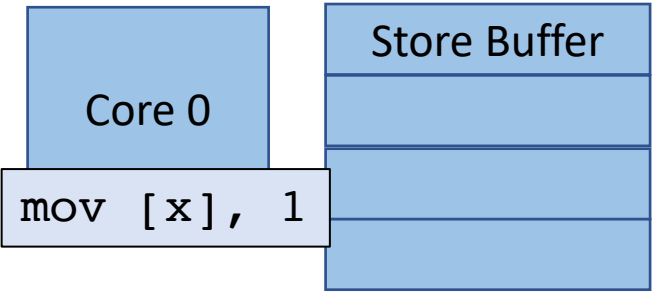
Main Memory



Thread 0:

mfence

mov %t0, [y]

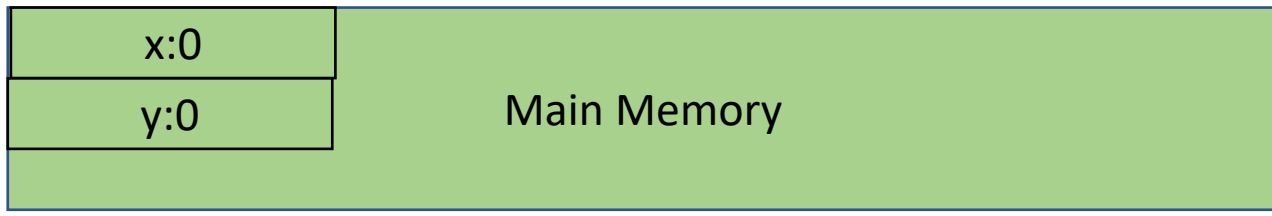
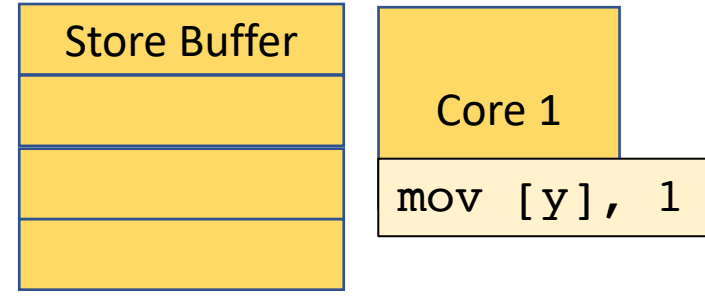


Execute first instruction

Thread 1:

mfence

mov %t1, [x]



Thread 0:

mfence

mov %t0, [y]

Core 0

Store Buffer

x:1

Values go into the store buffer

Store Buffer

y:1

Thread 1:

mfence

mov %t1, [x]

Core 1

x:0

y:0

Main Memory

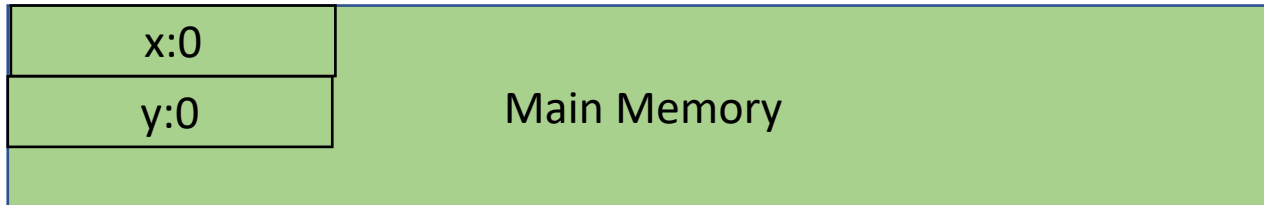
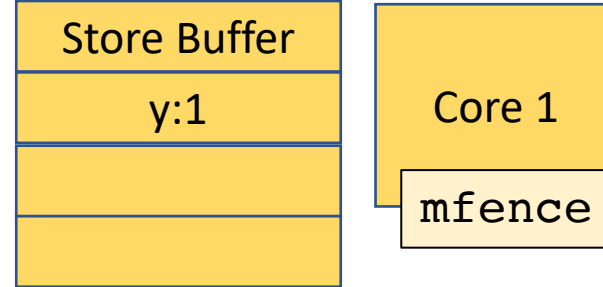
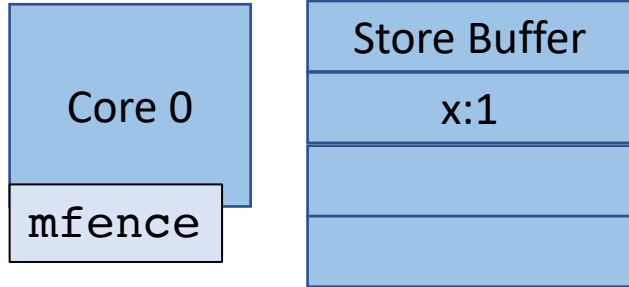
Thread 0:

Thread 1:

Execute next instruction

`mov %t0, [y]`

`mov %t1, [x]`



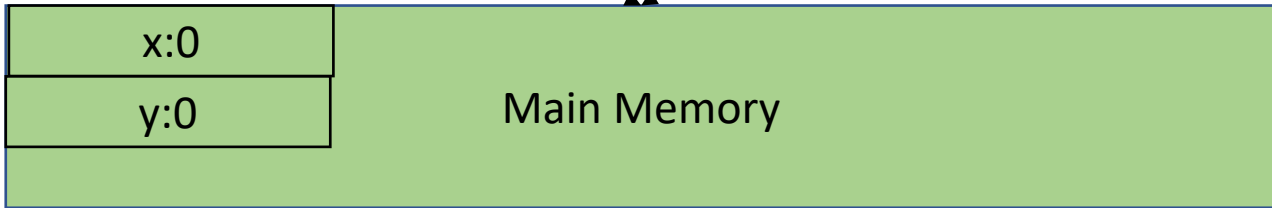
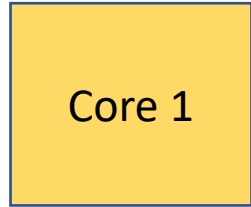
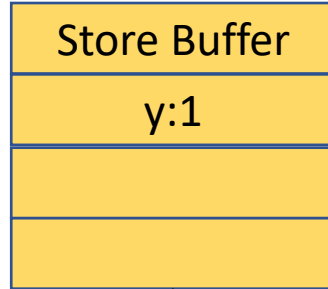
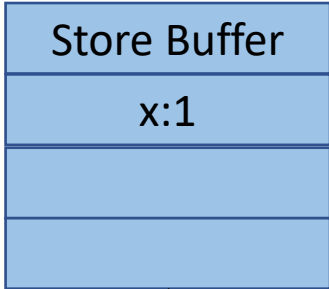
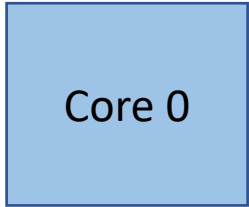
Thread 0:

Thread 1:

store buffers are flushed

```
mov %t0, [y]
```

```
mov %t1, [x]
```



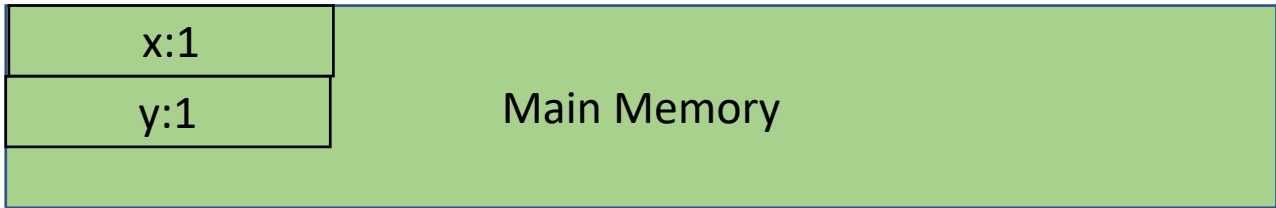
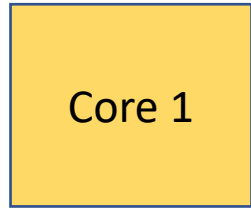
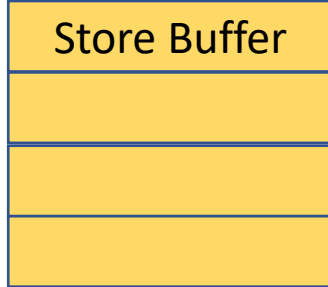
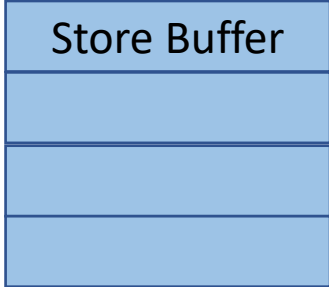
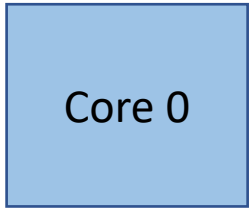
Thread 0:

Thread 1:

store buffers are flushed

```
mov %t0, [y]
```

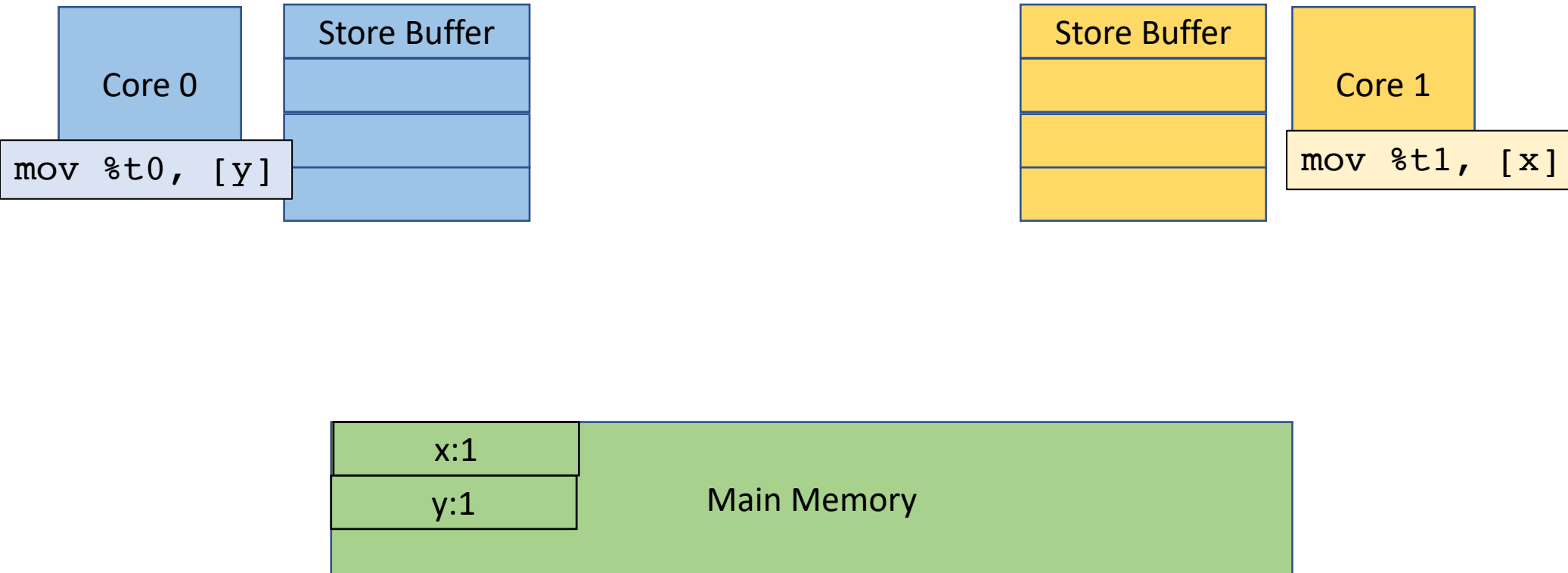
```
mov %t1, [x]
```



Thread 0:

Thread 1:

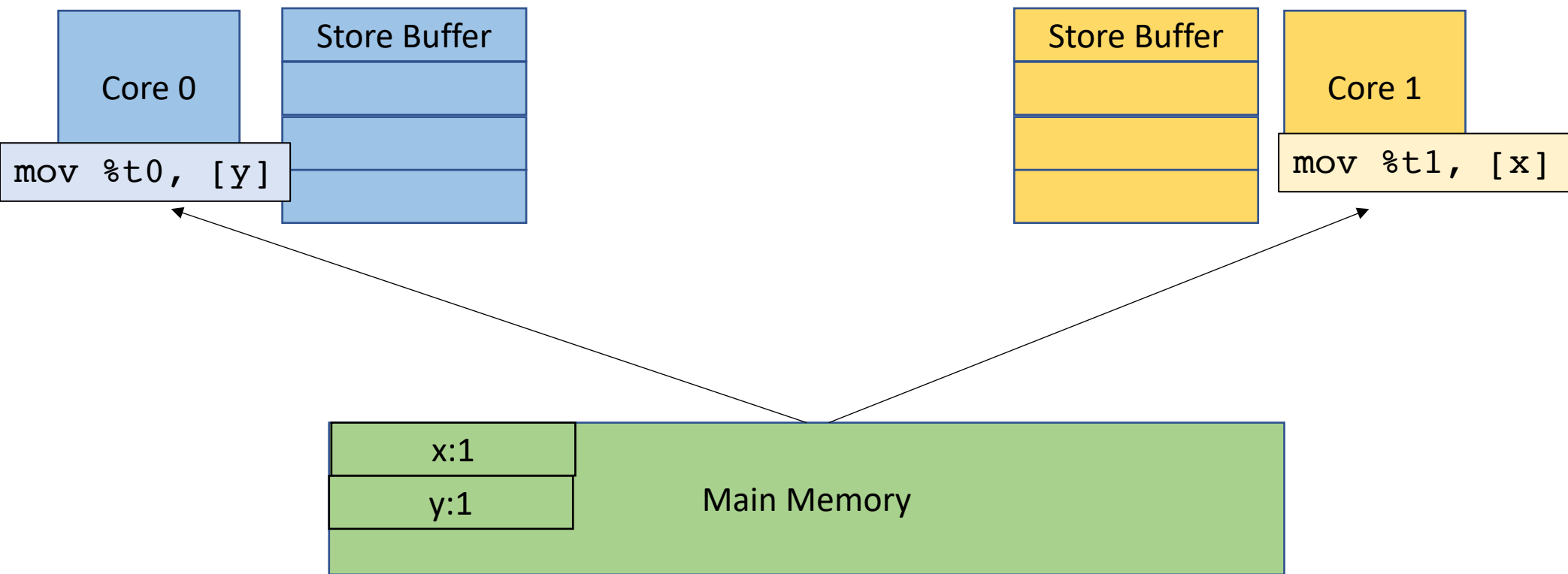
execute next instruction



Thread 0:

Thread 1:

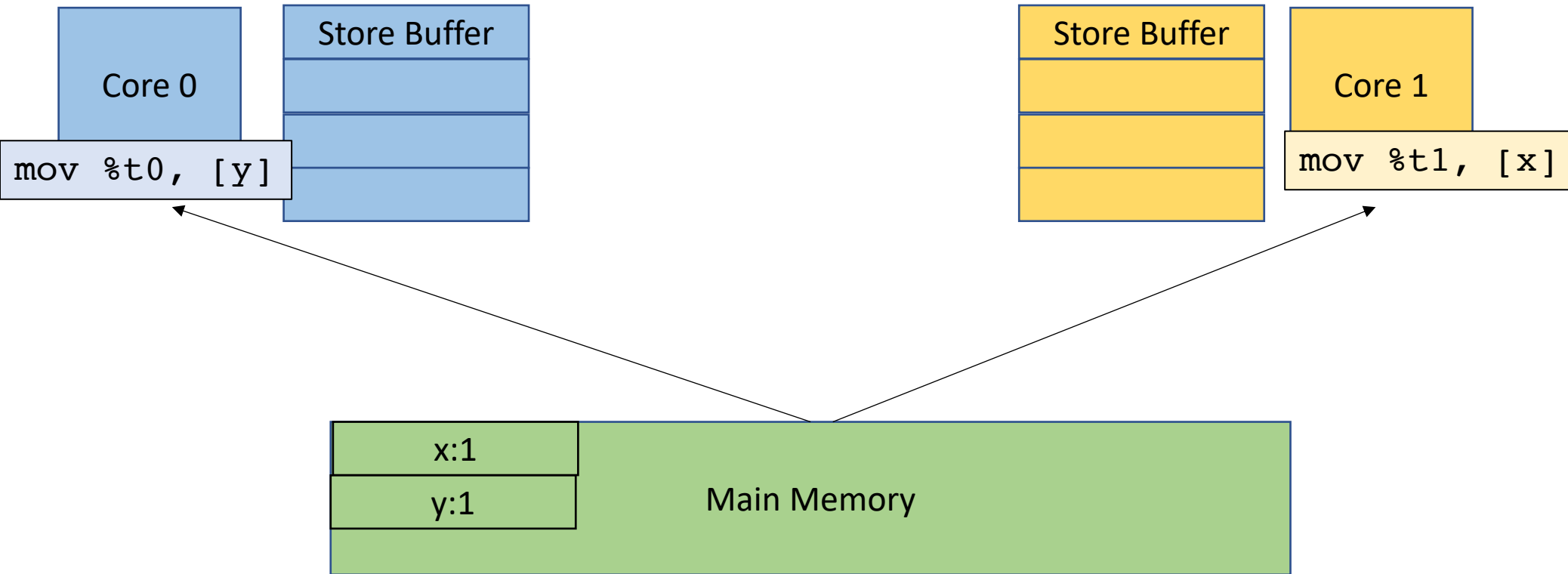
values are loaded from memory



Thread 0:

Thread 1:

We don't get the problematic behavior:  $t0 \neq 0$  and  $t1 \neq 0$





Next example

Thread 0:

```
mov [x], 1
```

```
mov %t0, [x]
```

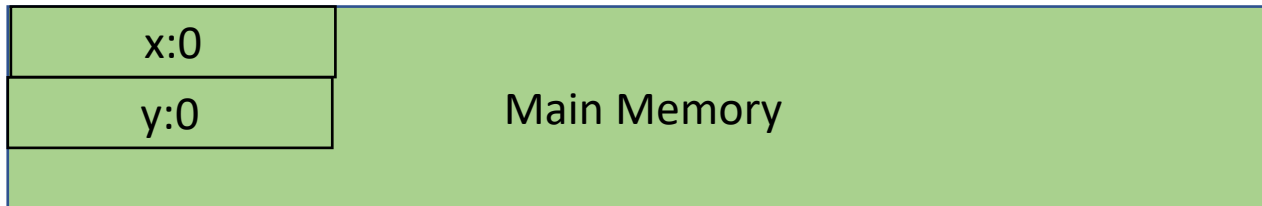
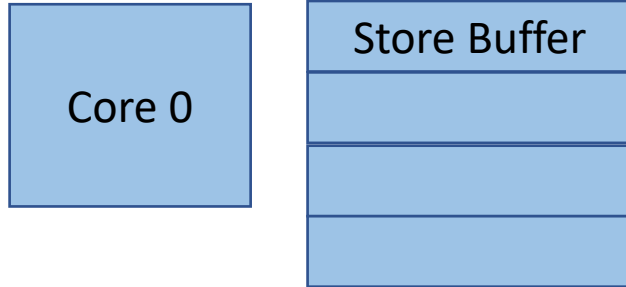
single thread  
same address

possible outcomes:

t0 = 1

t0 = 0

Which one do you expect?

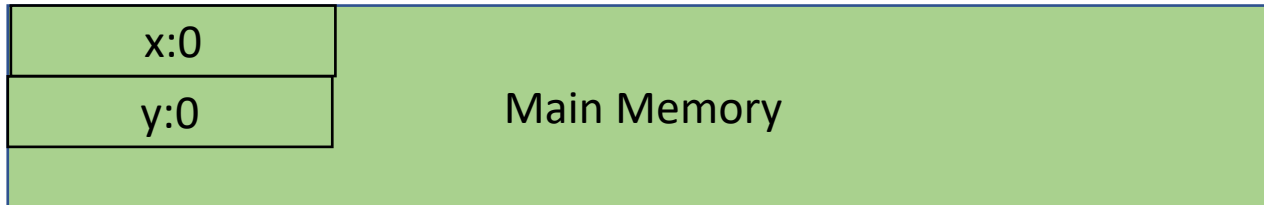
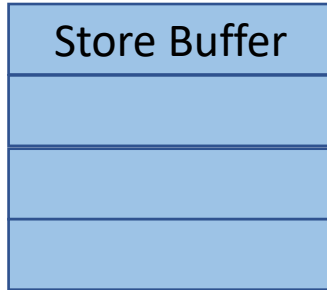
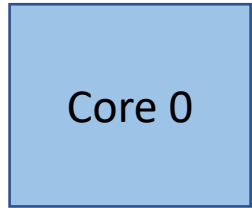


Thread 0:

```
mov [x], 1
```

```
mov %t0, [x]
```

How does this execute?

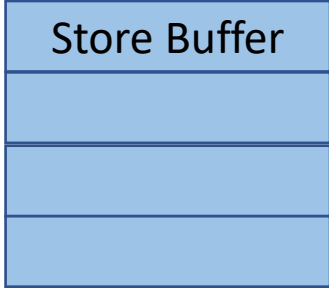


Thread 0:

execute first instruction

```
mov %t0, [x]
```

Core 0



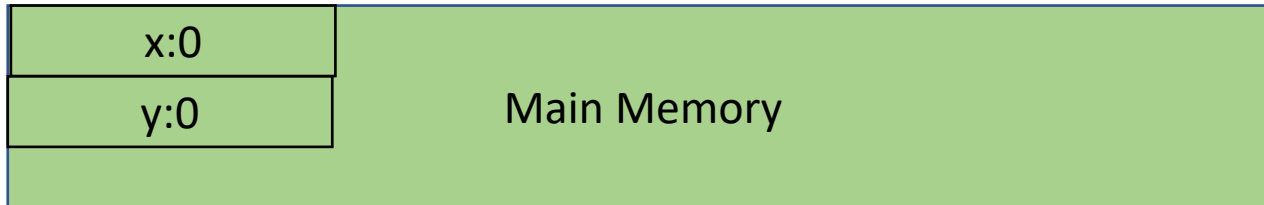
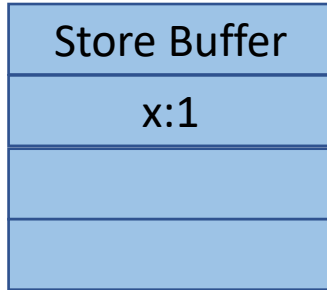
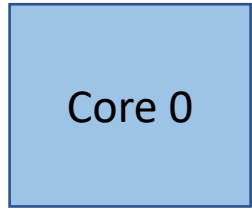
```
mov [x], 1
```



Thread 0:

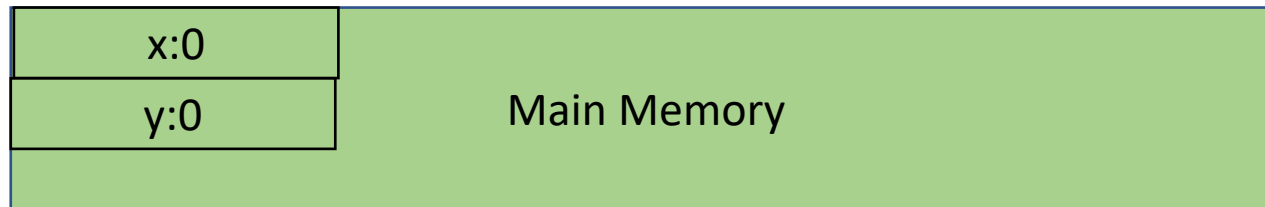
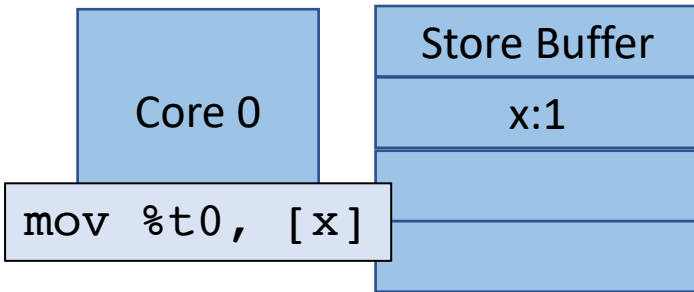
Store the value in the store buffer

```
mov %t0, [x]
```



Thread 0:

Next instruction

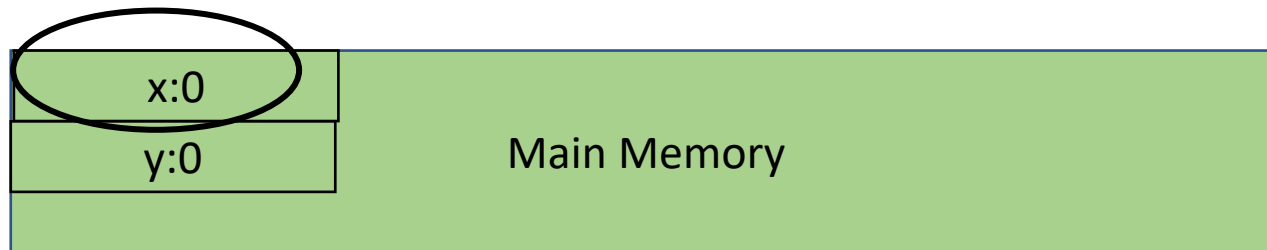
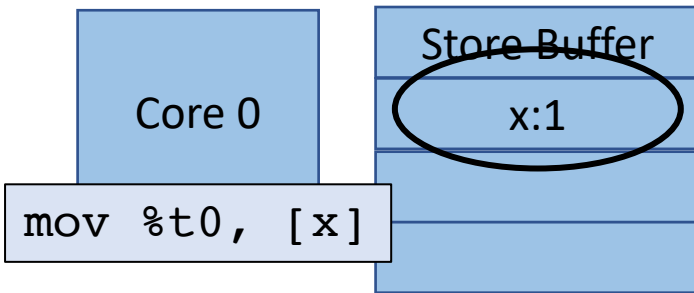


Thread 0:

Where to load??

Store buffer?

Main memory?

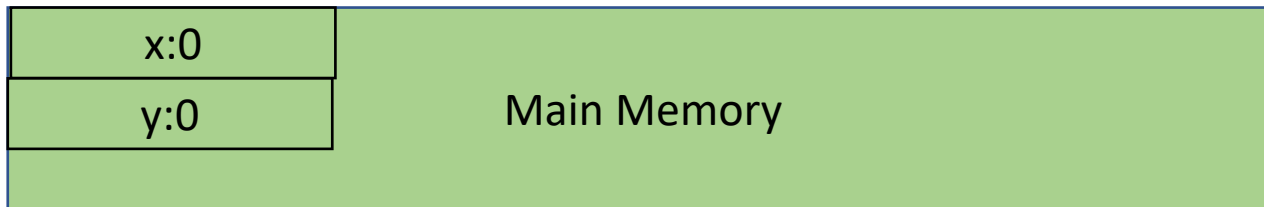
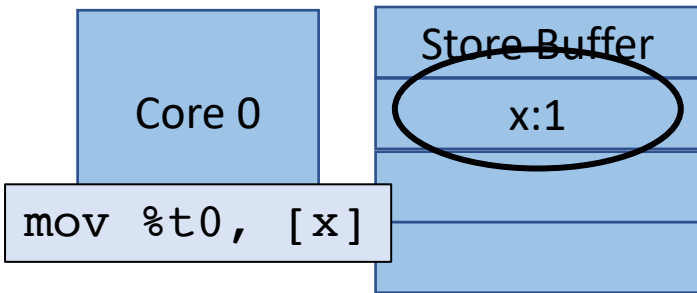


Thread 0:

Where to load??

Threads check store buffer before going to main memory

It is close and cheap to check.





# Memory Consistency

- How to specify a relaxed memory model?
- Good time for a 5 minute break!

# Memory Consistency

- How to specify a relaxed memory model?
- We can do it operationally
  - by constructing a high-level machine and reasoning about operations through the machine.
  - or we can talk about instructions that are allowed to "break" program order.

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



*We will annotate instructions with S for store, and L for loads*

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```



*We will annotate instructions with S for store, and L for loads*

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

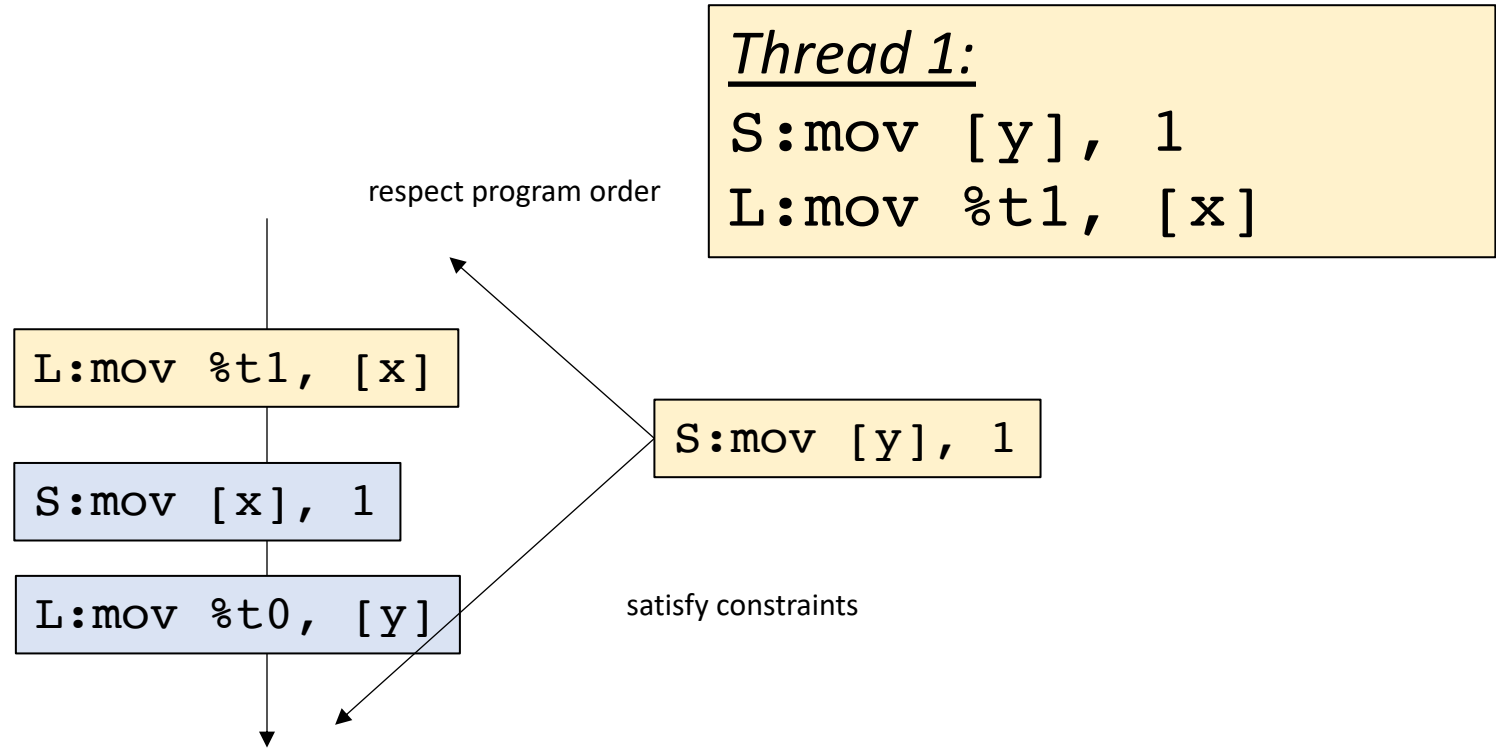
```
S:mov [x], 1  
L:mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

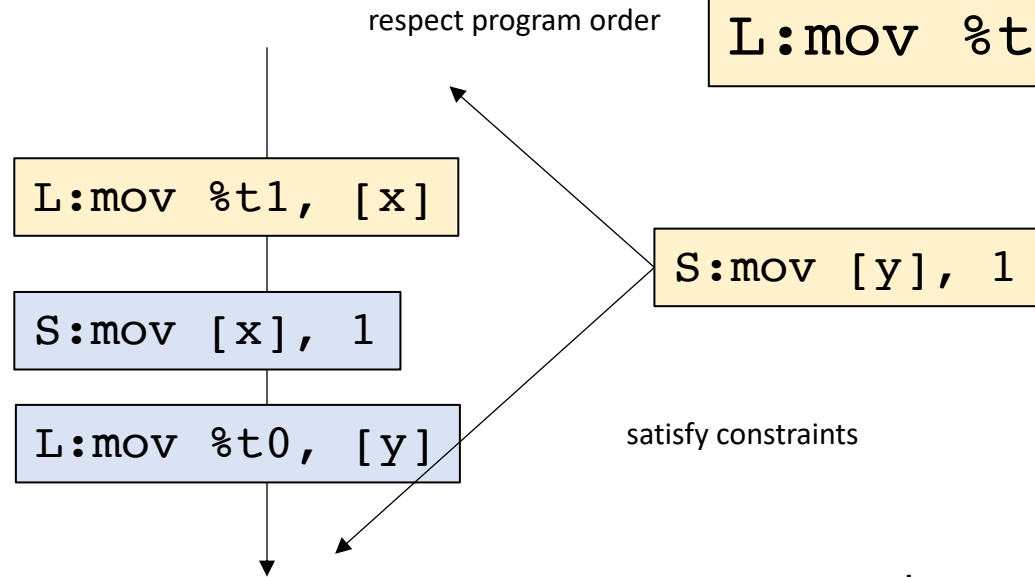
```
S:mov [x], 1  
L:mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```



Now we make a new rule:

S(tores) followed by a L(oad)  
do not have to follow program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can `t0 == t1 == 0`?

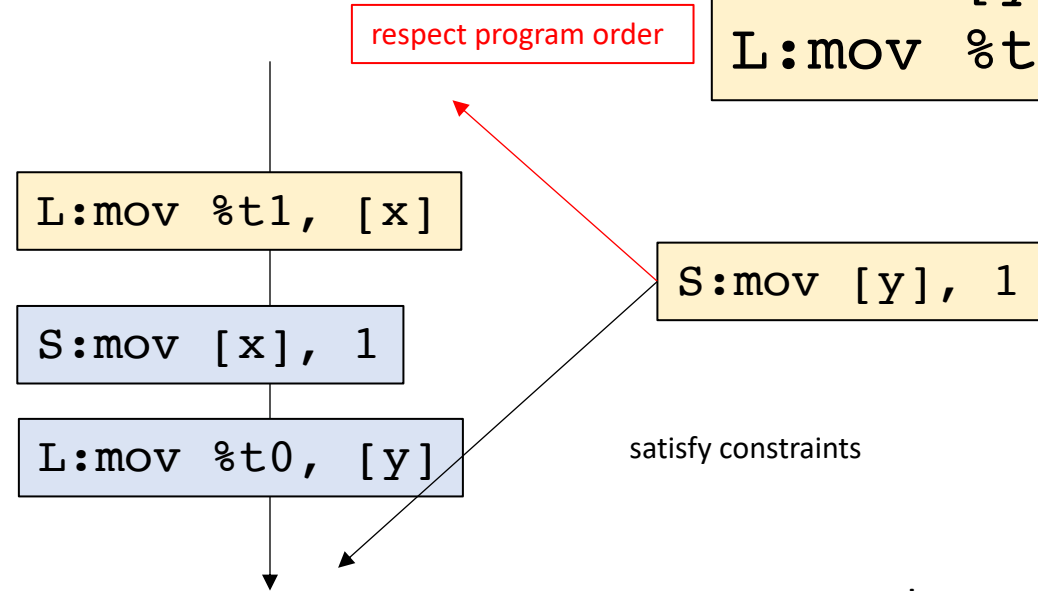
Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Now we make a new rule:

S(tores) followed by a L(oad)  
do not have to follow program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

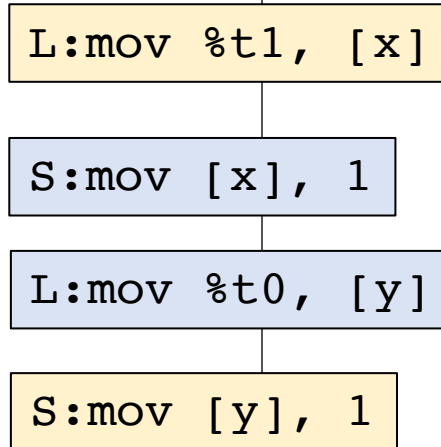
Another test

Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

we can ignore this condition!!



Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

Now we can satisfy the condition!



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

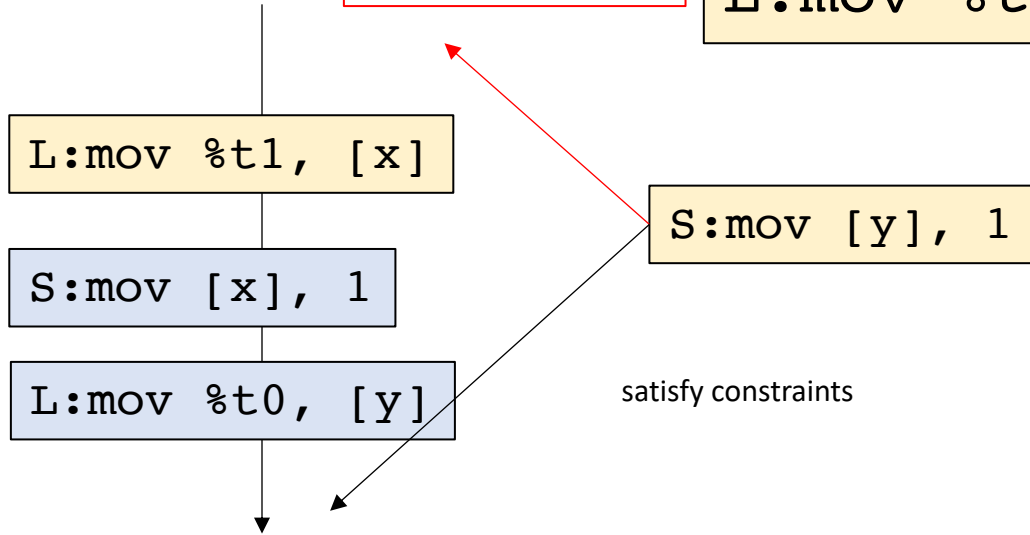
```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!

respect program order

satisfy constraints

Lets peak under the hood here



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!

respect program order

```
L:mov %t1, [x]
```

```
S:mov [x], 1
```

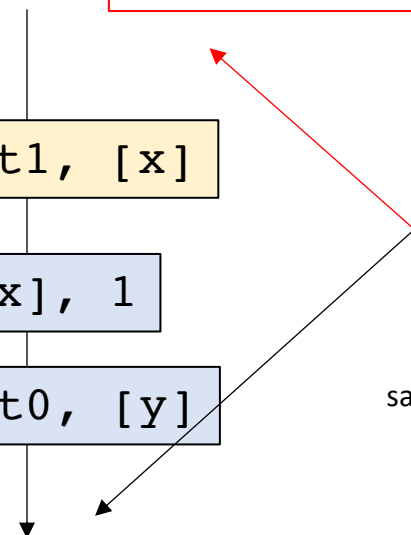
```
L:mov %t0, [y]
```

```
S:mov [y], 1
```

satisfy constraints

Lets peak under the hood here

Global timeline is when the  
Store operation becomes visible  
to other threads



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can `t0 == t1 == 0`?

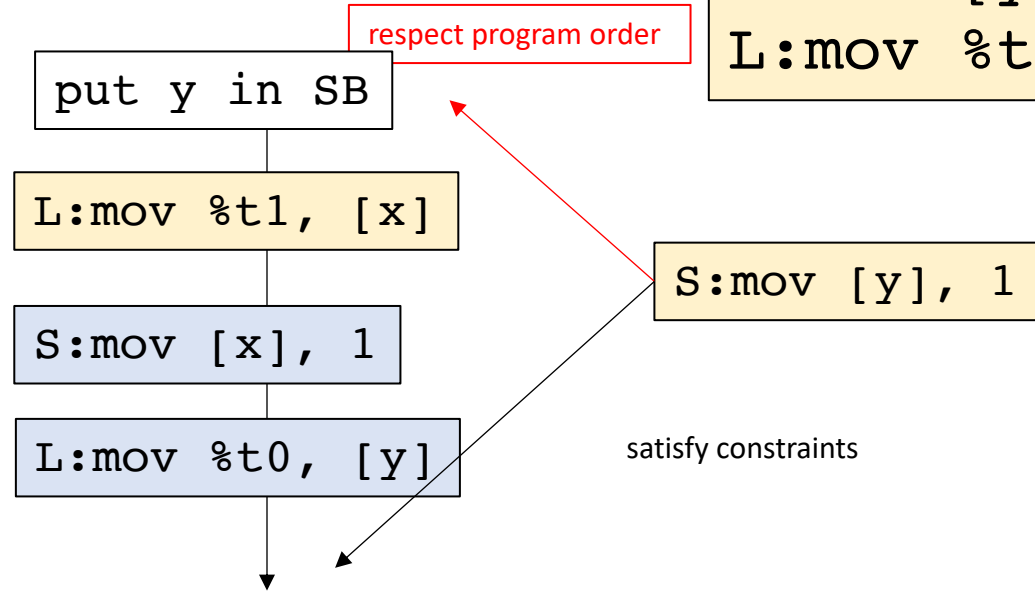
Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Lets peak under the hood here

Global timeline is when the Store operation becomes visible to other threads

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can `t0 == t1 == 0`?

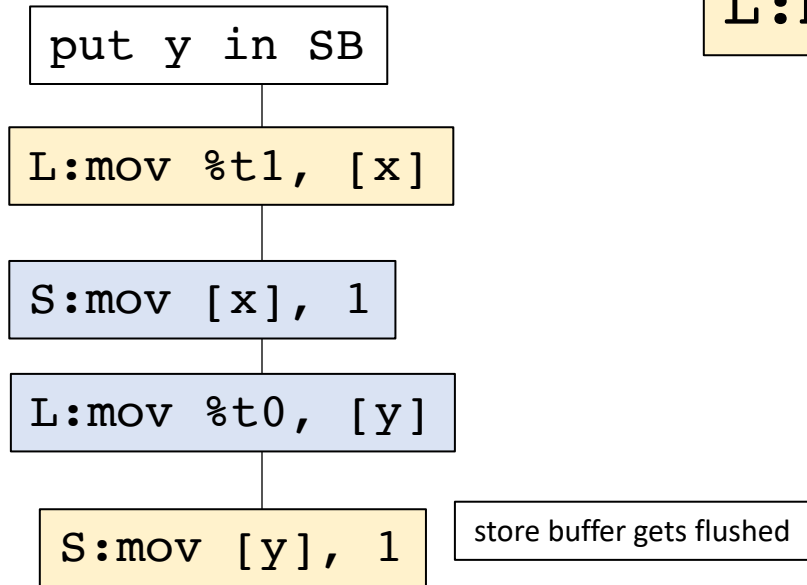
Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Lets peak under the hood here

Global timeline is when the Store operation becomes visible to other threads

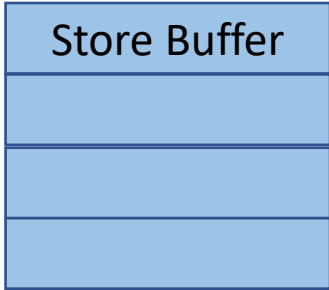
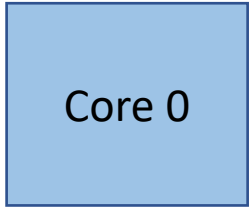
# Questions

- Can stores be reordered with stores?

Thread 0:

```
mov [x], 1
```

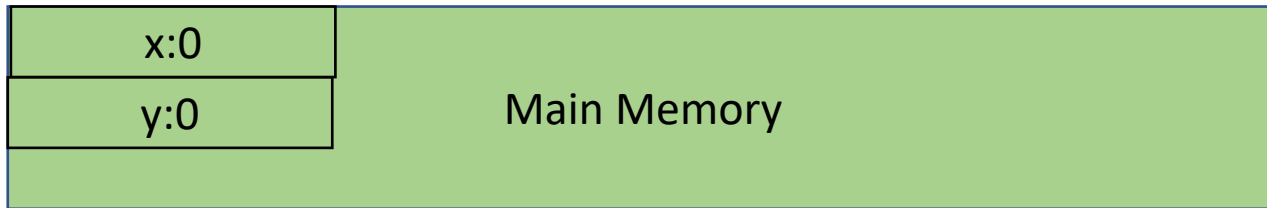
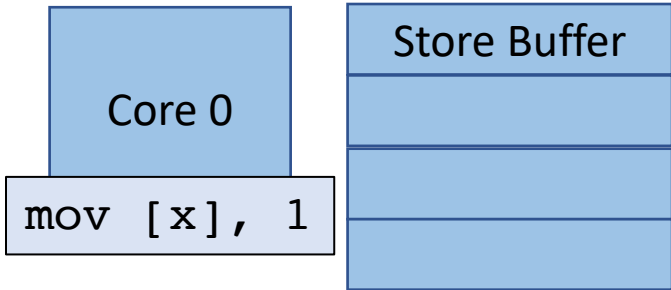
```
mov [y], 1
```



Thread 0:

mov [y], 1

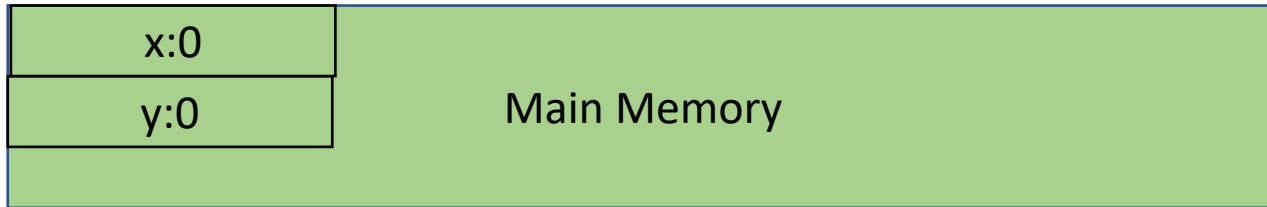
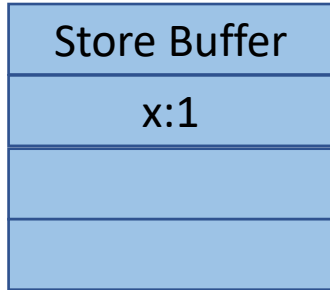
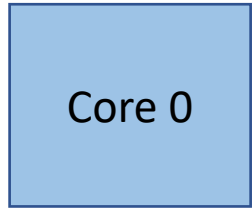
execute the first instruction



Thread 0:

```
mov [y], 1
```

value goes into store buffer

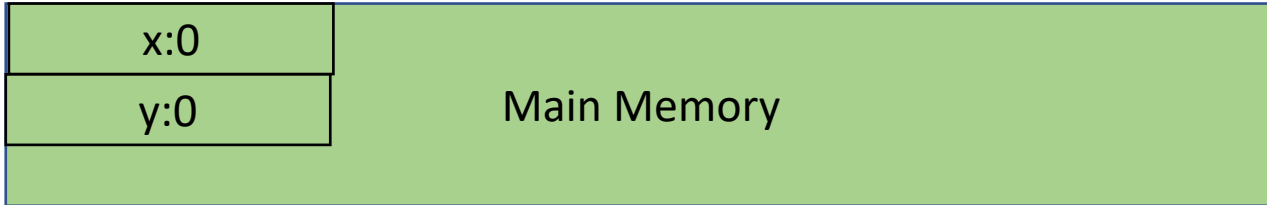
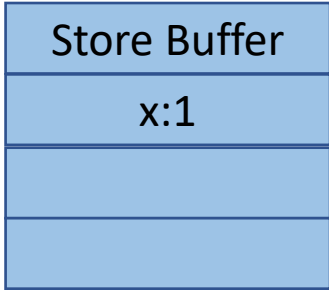
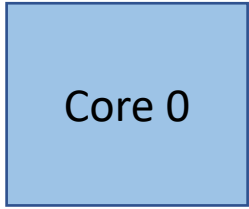




Thread 0:

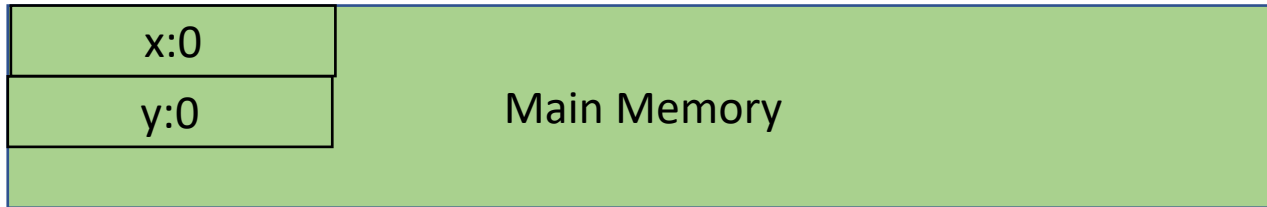
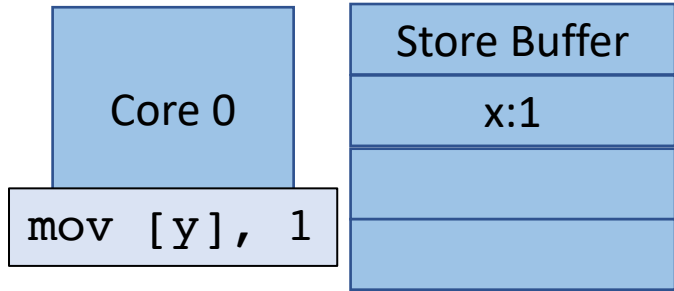
mov [y], 1

execute next instruction



Thread 0:

execute next instruction



Thread 0:

value goes into the store buffer

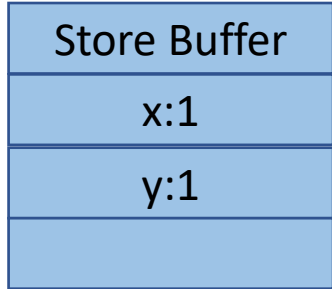
Core 0

Store Buffer
x:1
y:1

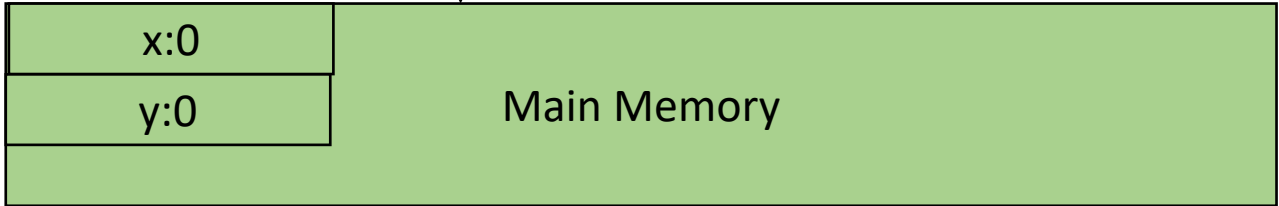
x:0	Main Memory
y:0	

Thread 0:

Core 0

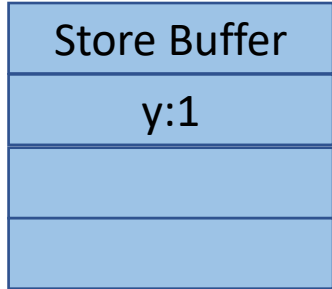


On x86, the store buffer trains in a FIFO way:  
thus stores cannot be reordered

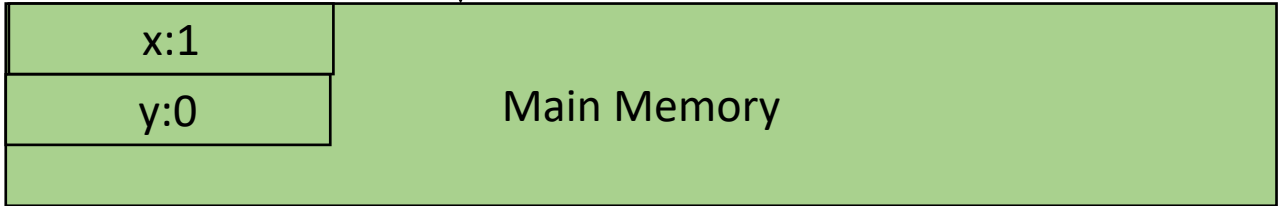


Thread 0:

Core 0

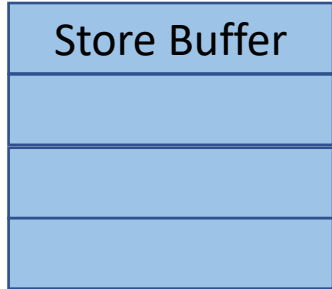


On x86, the store buffer trains in a FIFO way:  
thus stores cannot be reordered

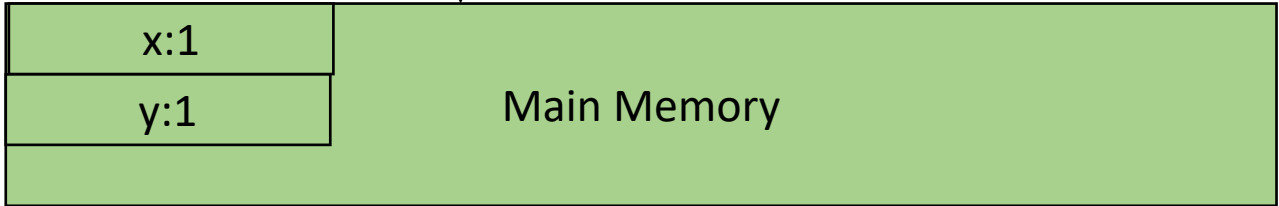


Thread 0:

Core 0



On x86, the store buffer trains in a FIFO way:  
thus stores cannot be reordered



# Questions

- Can stores be reordered with stores?
- How do we make rules about mfence?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
mfence  
L:mov %t0, [y]
```

```
S:mov [x], 1
```

```
mfence
```

```
L:mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
S:mov [y], 1  
mfence  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

```
mfence
```

```
L:mov %t1, [x]
```

Rules: S(tores) followed by a L(oad)  
do not have to follow program order.



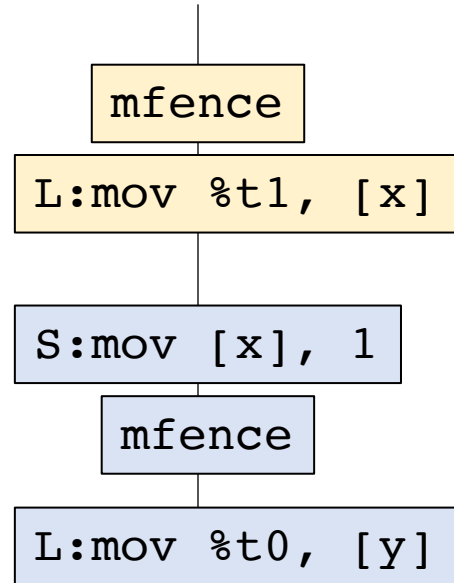
Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
mfence  
L:mov %t0, [y]
```

Another test  
Can t0 == t1 == 0?



Thread 1:

```
S:mov [y], 1  
mfence  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

Rules:

S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

Global variable:

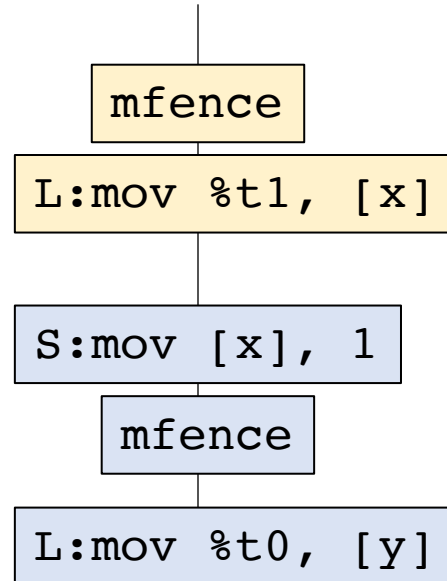
```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
mfence  
L:mov %t0, [y]
```

*So we can't  
reorder  
this instruction  
at all!*

Another test  
Can t0 == t1 == 0?



Thread 1:

```
S:mov [y], 1  
mfence  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

Rules:

S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

# Rules

- Are we done?

Rules:

S(tores) followed by a L(oad)

do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can t0 == 0?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

```
S:mov [x], 1
```

```
L:mov %t0, [x]
```



Rules:  
S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can t0 == 0?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

```
S:mov [x], 1
```

where to put this store?

```
L:mov %t0, [x]
```

Rules:

S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test  
Can t0 == 0?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

```
S:mov [x], 1
```

where to put this store?

```
L:mov %t0, [x]
```

Rules:

S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

S(tores) cannot be reordered past L(oads)  
from the same address

# TSO - Total Store Order

## **Rules:**

S(tores) followed by a L(oad)  
do not have to follow program order.

S(tores) cannot be reordered past a fence  
in program order

S(tores) cannot be reordered past L(oads)  
from the same address

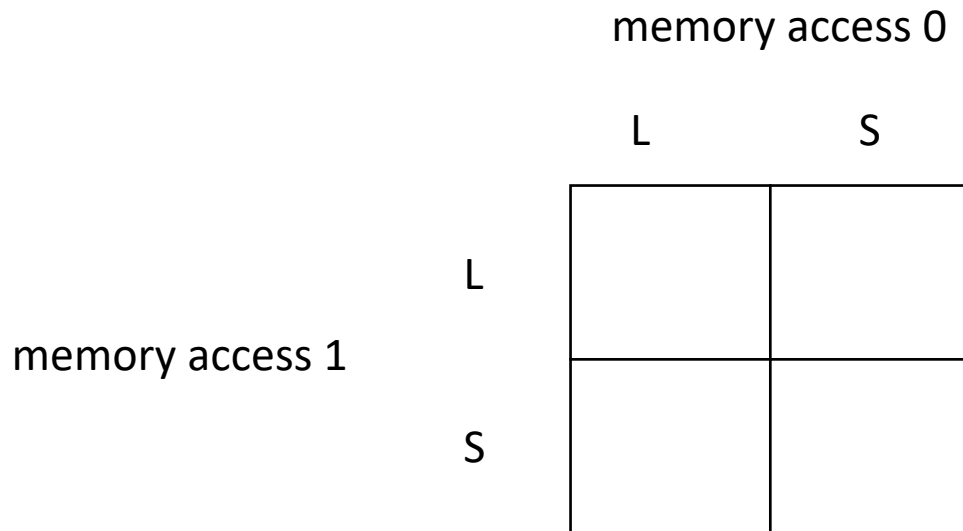
# Schedule

- Parallel schedules in OpenMP
- Memory consistency models:
  - Total store order
  - **Relaxed memory consistency**
  - Examples



# Other memory models?

- We can specify them in terms of what reorderings are allowed



If memory access 0 appears before memory access 1 in program order, can it bypass program order?

# Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	NO
	S	NO	NO

## Sequential Consistency

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

# Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	Different address
	S	NO	NO

## **TSO - total store order**

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

# Other memory models?

- We can specify them in terms of what reorderings are allowed

memory access 0

	L	S
L	?	?
S	?	?

memory access 1

The diagram illustrates a 2x2 grid representing memory access reordering. The columns are labeled 'L' and 'S' under the heading 'memory access 0'. The rows are labeled 'L' and 'S' under the heading 'memory access 1'. Each cell in the grid contains a question mark '?'.

## Weaker models?

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

# Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	Different address
	S	NO	Different address

## **PSO - partial store order**

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

*Allows stores to drain from the store buffer in any order*

# Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	YES	Different address
	S	Different address	Different address

## **RMO - Relaxed Memory Order**

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

*Very relaxed model!*

# Other memory models?

- FENCE: can always restore order using fences. Accesses cannot be reordered past fences!

		memory access 0	
		L	S
memory access 1	L	NO	NO
	S	NO	NO

## Any Memory Model

If memory access 0 appears before memory access 1 in program order, and there is a FENCE between the two accesses, can it bypass program order?

# Schedule

- Parallel schedules in OpenMP
- Memory consistency models:
  - Total store order
  - Relaxed memory consistency
  - **Examples**



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

First thing: change our syntax to pseudo code

Thread 0:

```
L:mov %t0, [y]  
S:mov [x], 1
```

Thread 1:

```
L:mov %t1, [x]  
S:mov [y], 1
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

First thing: change our syntax to pseudo code  
You should be able to find natural mappings  
to any ISA

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

```
L:%t0 = load(y)
```

```
S:store(x,1)
```

Question: can  $t0 == t1 == 1$ ?

Get out our lego bricks and try for sequential consistency

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t1 = load(x)
```

```
S:store(y,1)
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

Get out our lego bricks

Thread 0:

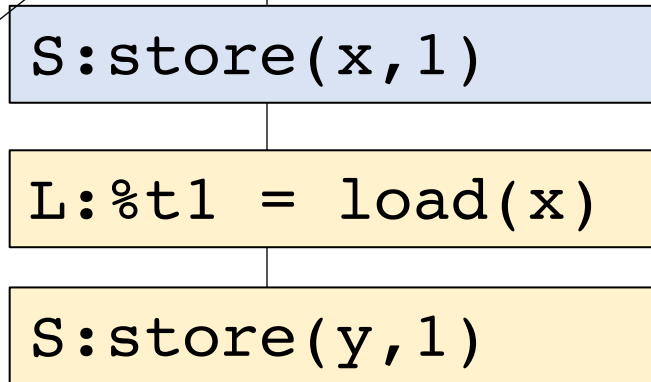
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

Not allowed under sequential consistency!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

Get out our lego bricks

Thread 0:

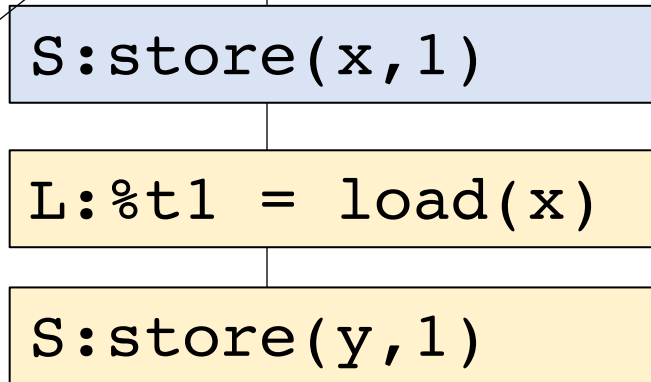
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	NO	Different address
S	NO	NO

What about TSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?  
Get out our lego bricks

Thread 0:

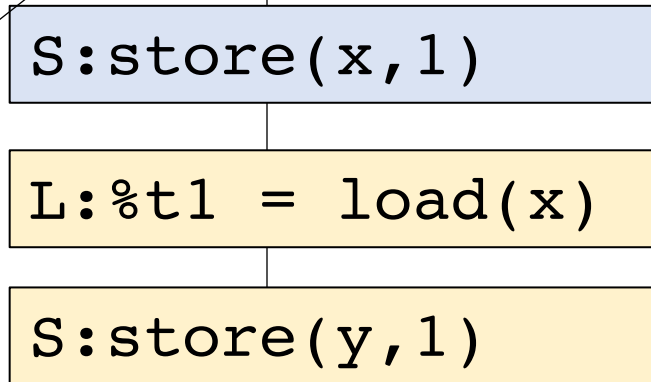
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	NO	Different address
S	NO	NO

What about TSO? NOT ALLOWED!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

Get out our lego bricks

Thread 0:

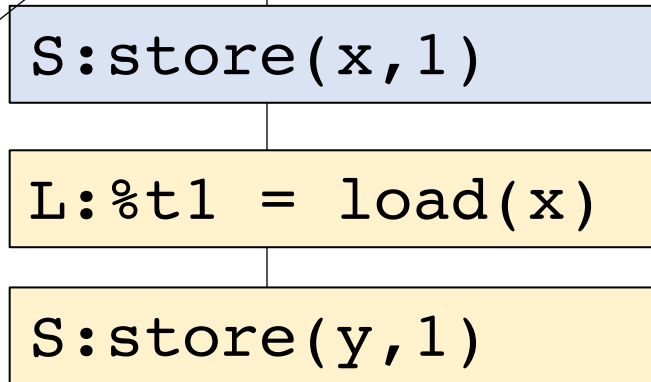
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	NO	Different address
S	NO	Different address

What about PSO?





Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?  
Get out our lego bricks

Thread 0:

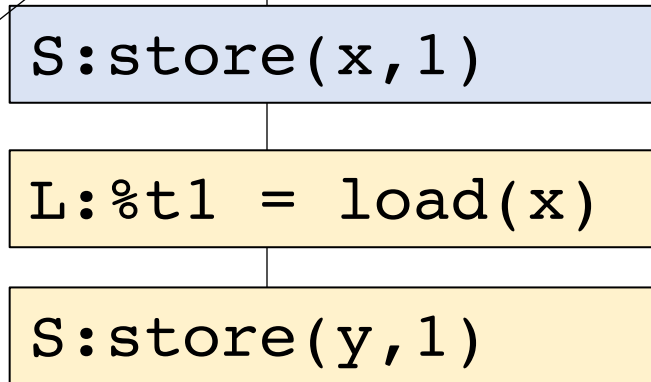
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	YES	Different address
S	different address	Different address

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Get out our lego bricks

Thread 0:

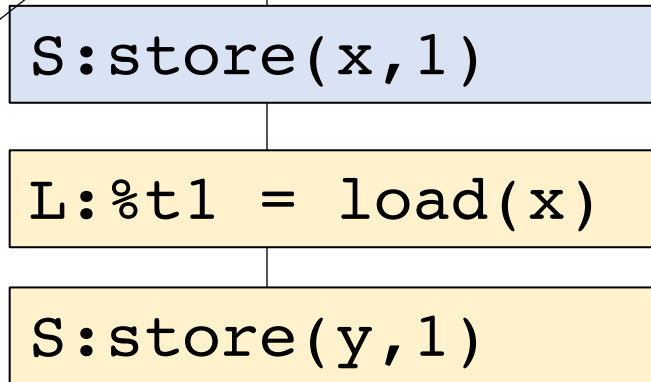
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	YES	Different address
S	different address	Different address

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order

```
S:store(x,1)
```

```
L:%t1 = load(x)
```

```
S:store(y,1)
```

satisfy constraints

memory access 1

memory access 0

	L	S
L	YES	Different address
S	different address	Different address

What about RMO? YES!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?  
Get out our lego bricks

Thread 0:

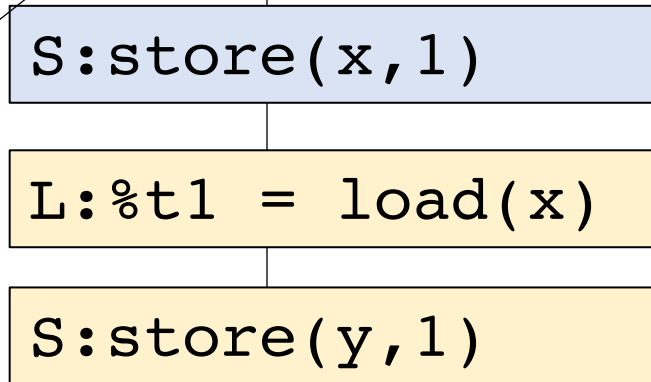
```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t0 = load(y)
```

respect program order



satisfy constraints

memory access 1

memory access 0

	L	S
L	YES	Different address
S	different address	Different address

How do we disallow the behavior in RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

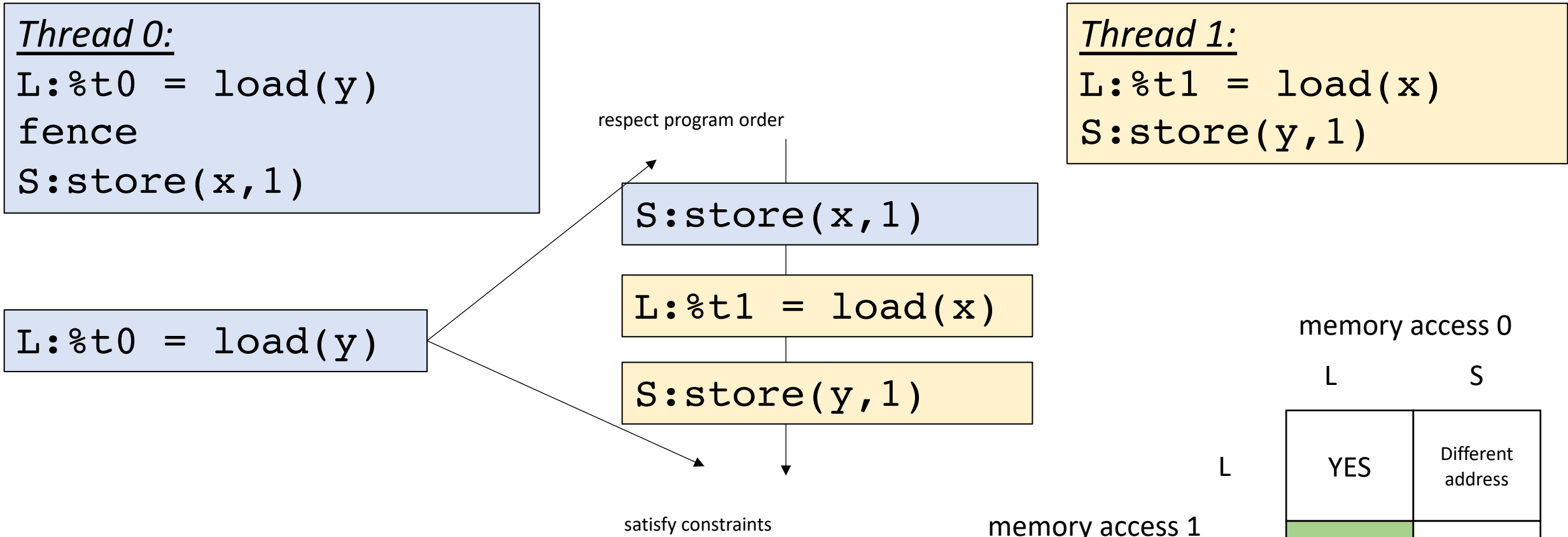
Question: can  $t0 == t1 == 1$ ?  
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```



	L	S
L	YES	Different address
S	different address	Different address

How do we disallow the behavior in RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

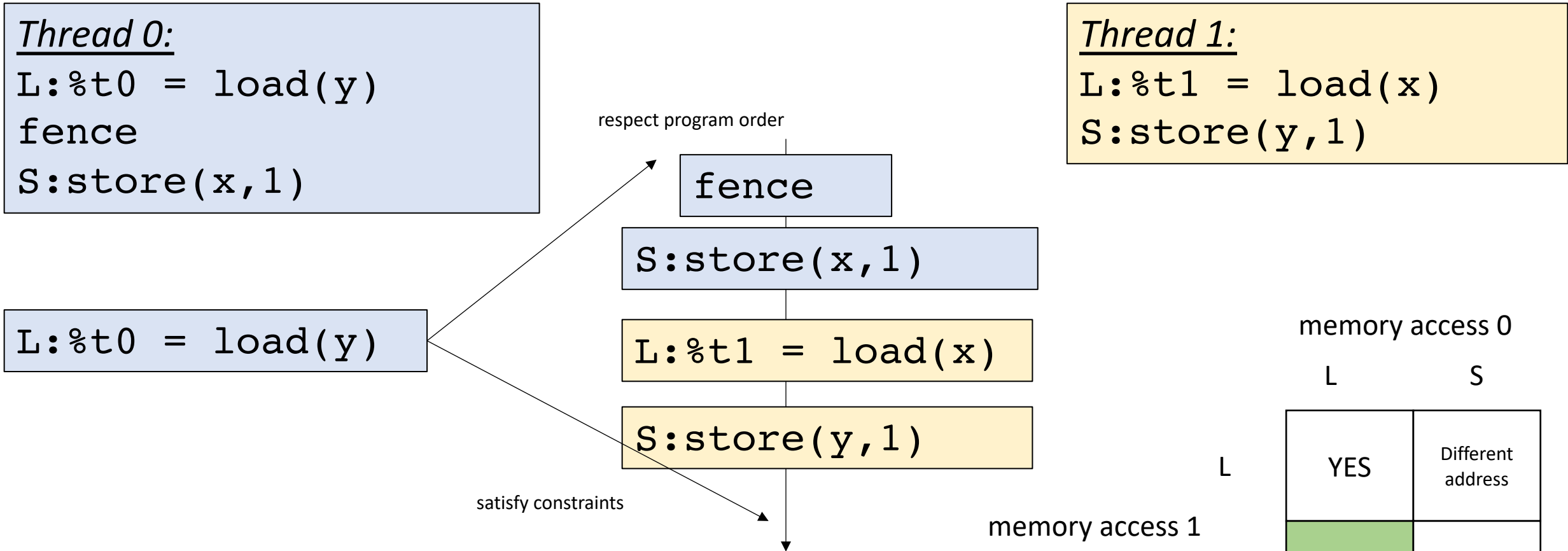
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```



memory access 0

L S

L

YES	Different address
-----	-------------------

memory access 1

S

different address	Different address
-------------------	-------------------

How do we disallow the behavior in RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

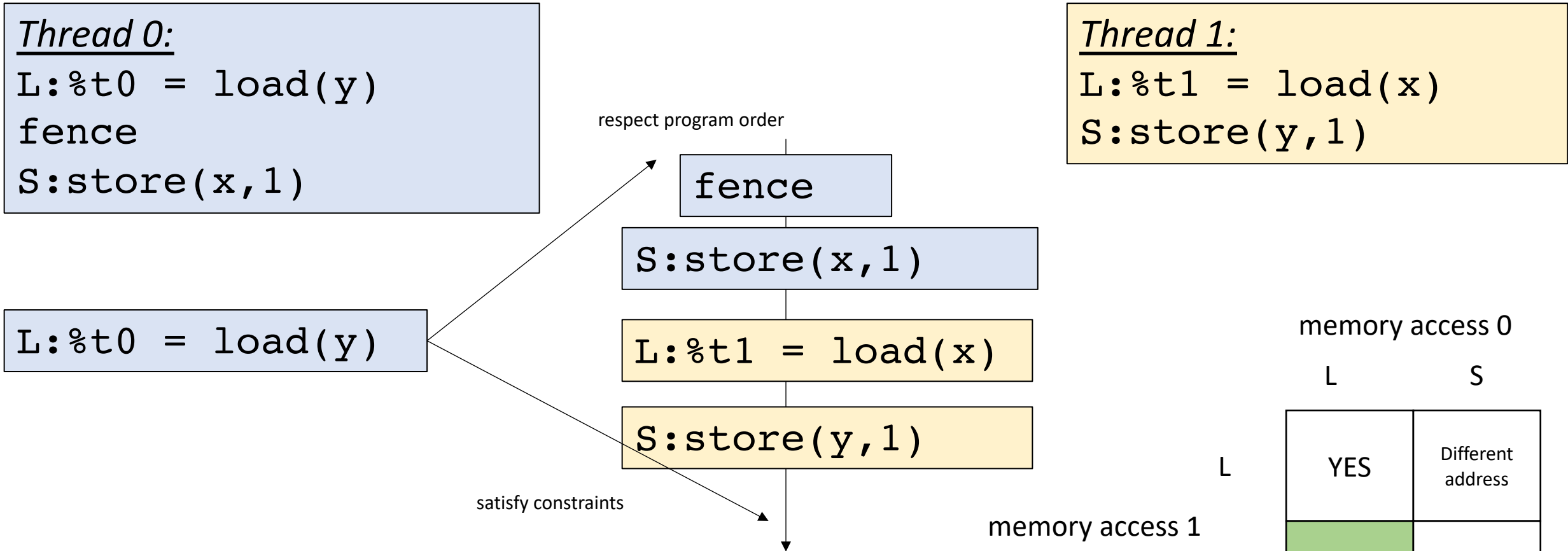
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```



memory access 0

L S

L

	L	S
L	YES	Different address
S	different address	Different address

memory access 1

S

Now we cannot break program order past the fence!  
Are we done?



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can  $t0 == t1 == 1$ ?

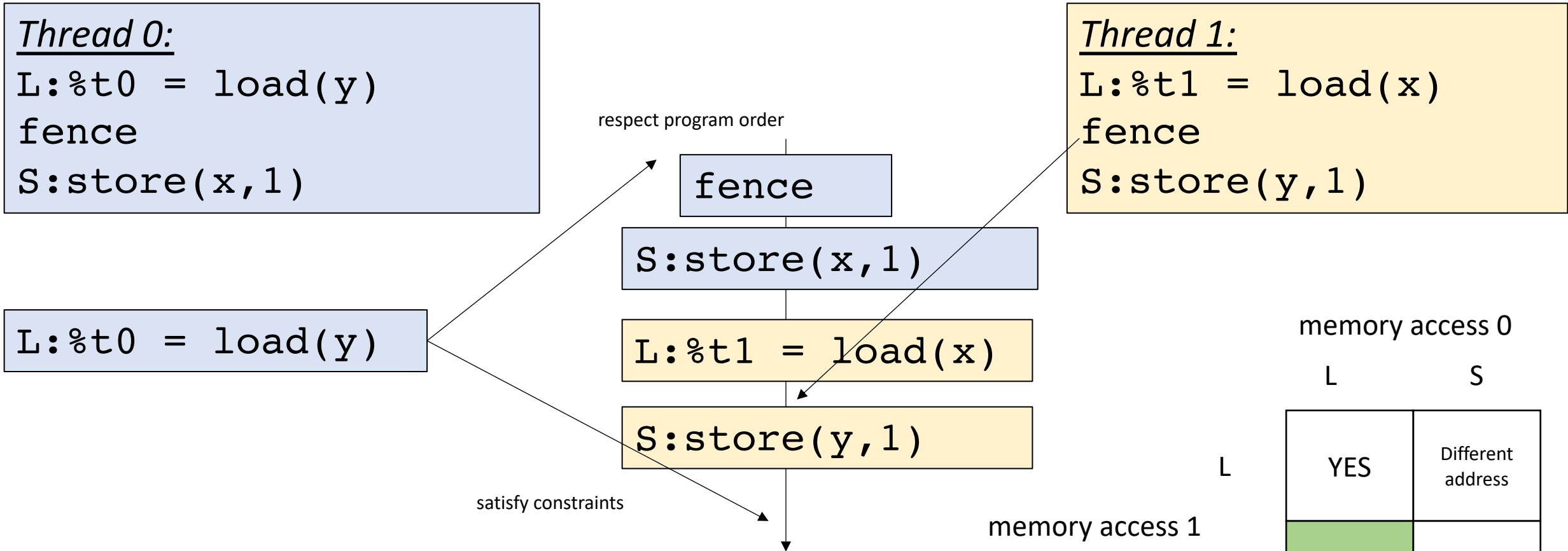
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
fence  
S:store(y,1)
```



```
L:%t0 = load(y)
```

memory access 0

L S

L

	L	S
L	YES	Different address
S	different address	Different address

memory access 1

S

Now we cannot break program order past the fence!  
Are we done?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

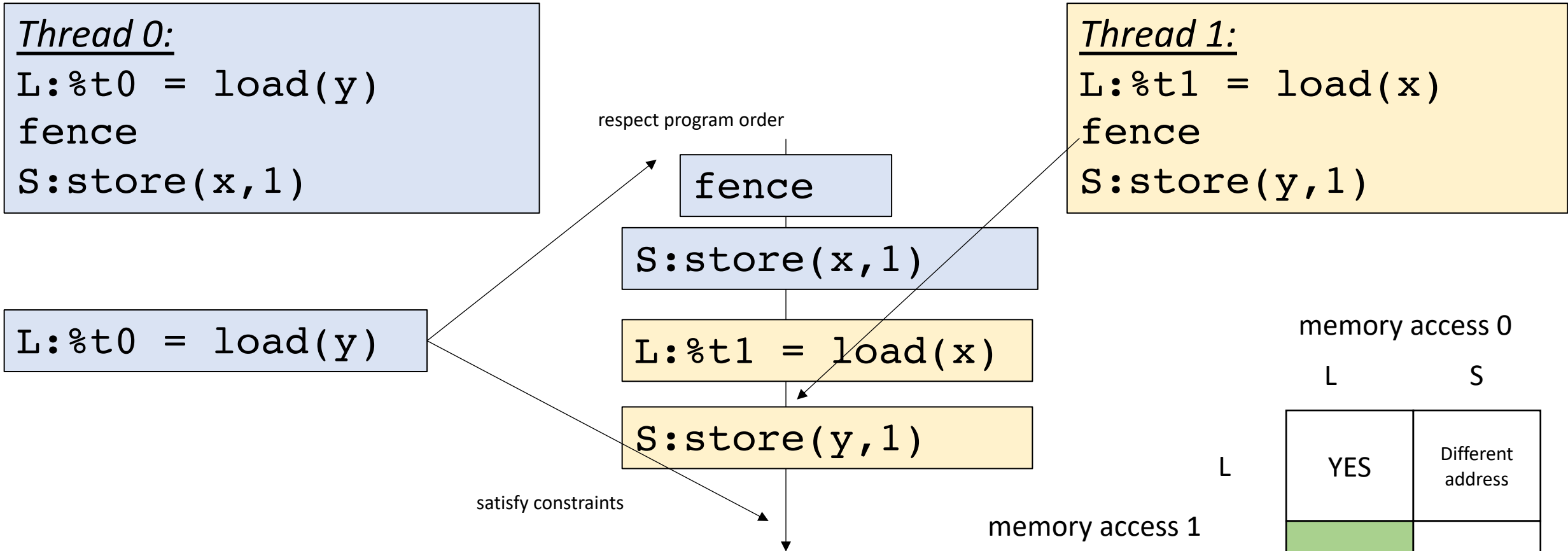
Question: can  $t0 == t1 == 1$ ?  
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
fence  
S:store(y,1)
```



	L	S
L	YES	Different address
S	different address	Different address

Now we cannot break program order past the fence!  
Are we done? The behavior is no longer allowed

One more example

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

```
S:store(x,1)
```

```
S:store(y,1)
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

```
S:store(x,1)
```

```
S:store(y,1)
```

Question: can `t0 == 1` and `t1 == 0`?

start off thinking  
about sequential  
consistency



Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

start off thinking  
about sequential  
consistency

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
S:store(x,1)
```

respect program order

```
S:store(y,1)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```

satisfy constraints



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

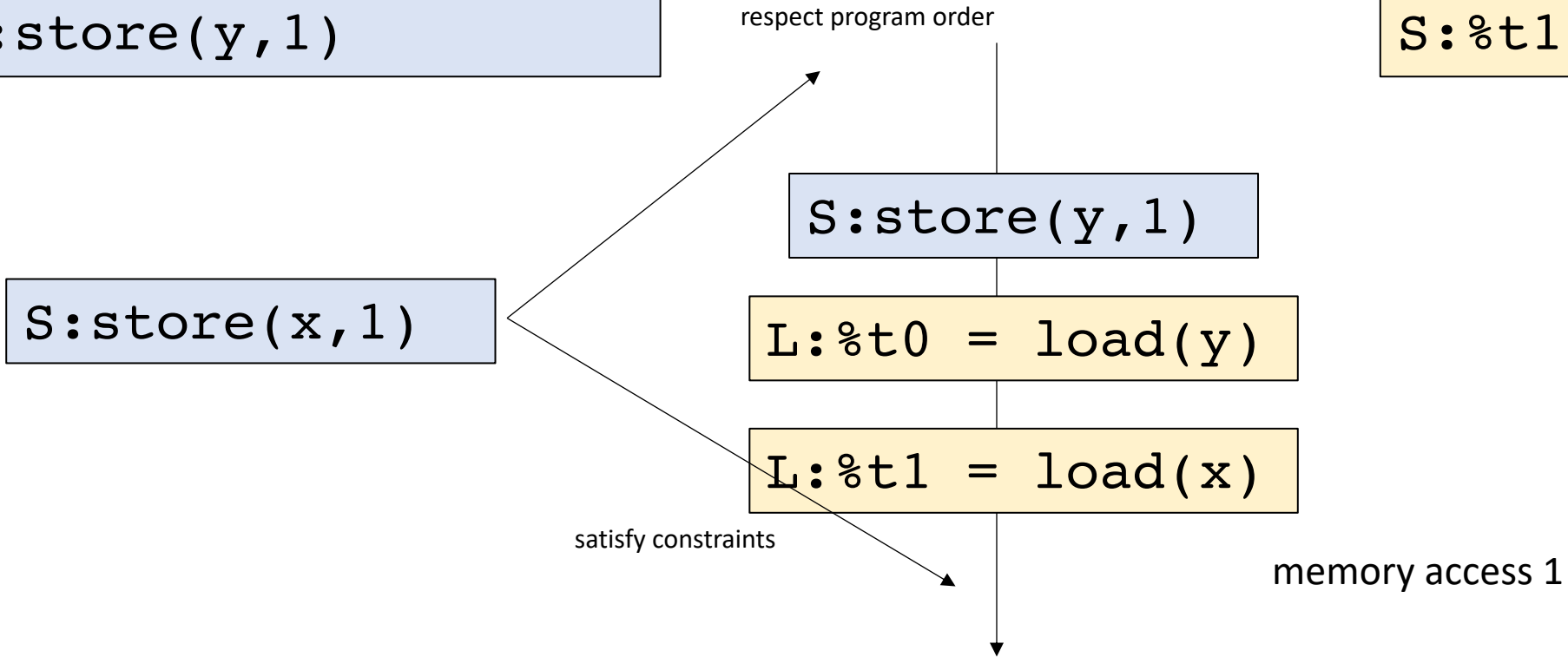
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different address

S

NO

NO

What about TSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

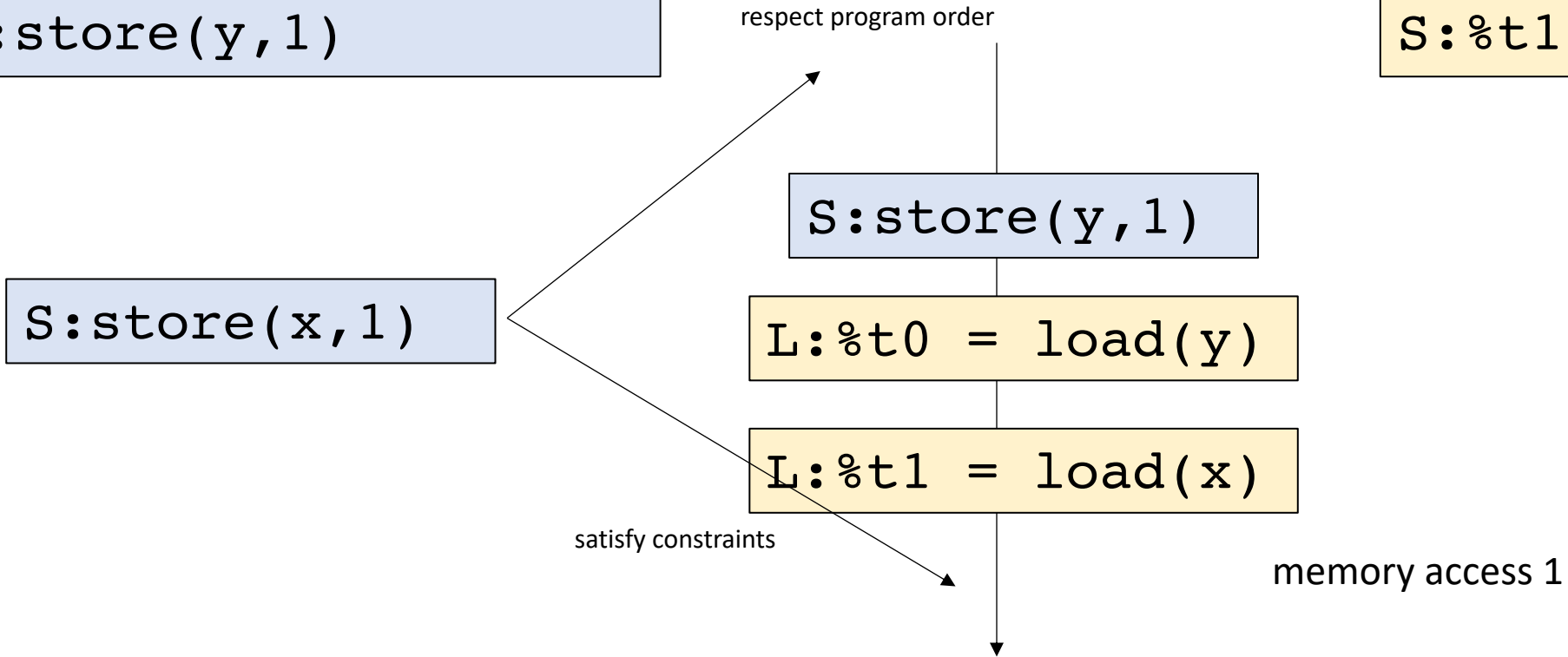
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO	Different address
NO	NO

S

What about TSO? NO



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

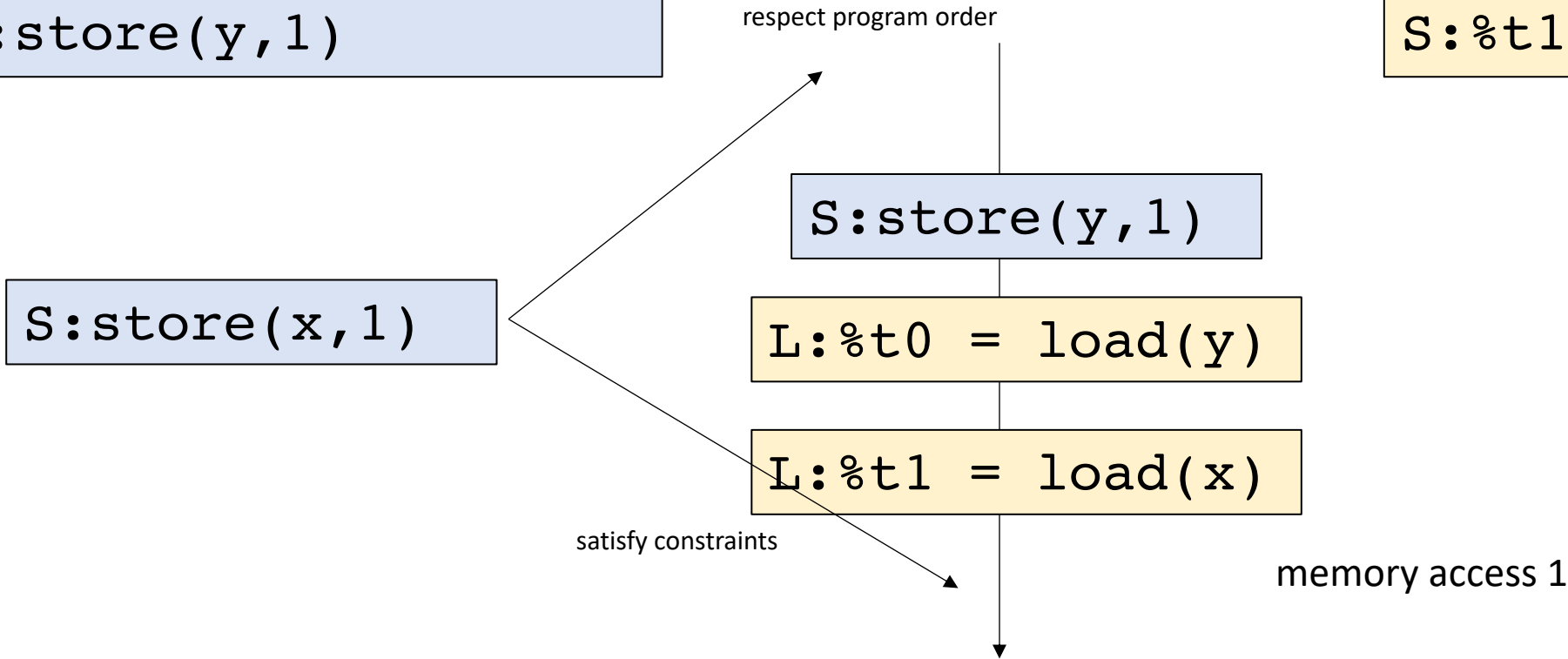
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

S

NO	Different address
NO	Different address

What about PSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

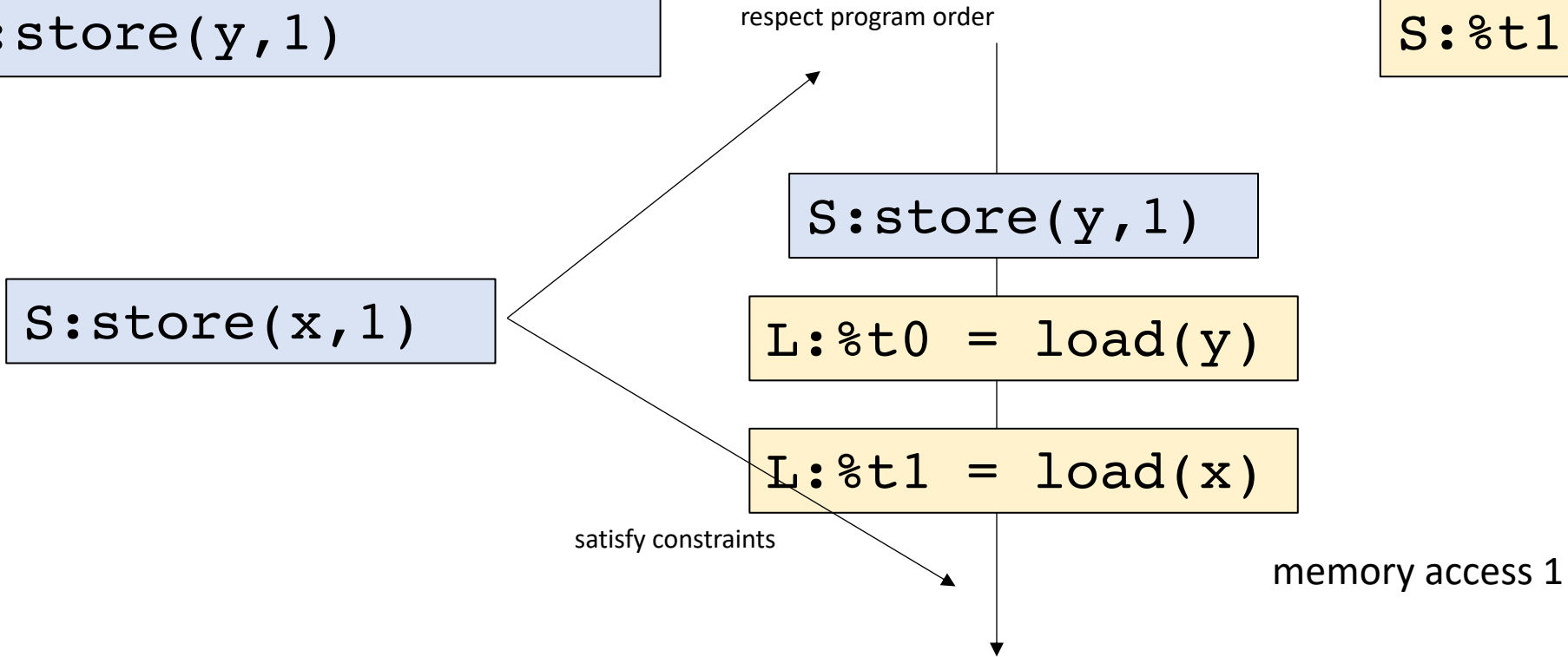
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

	L	S
L	NO	Different address
S	NO	Different address

memory access 1

S

What about PSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

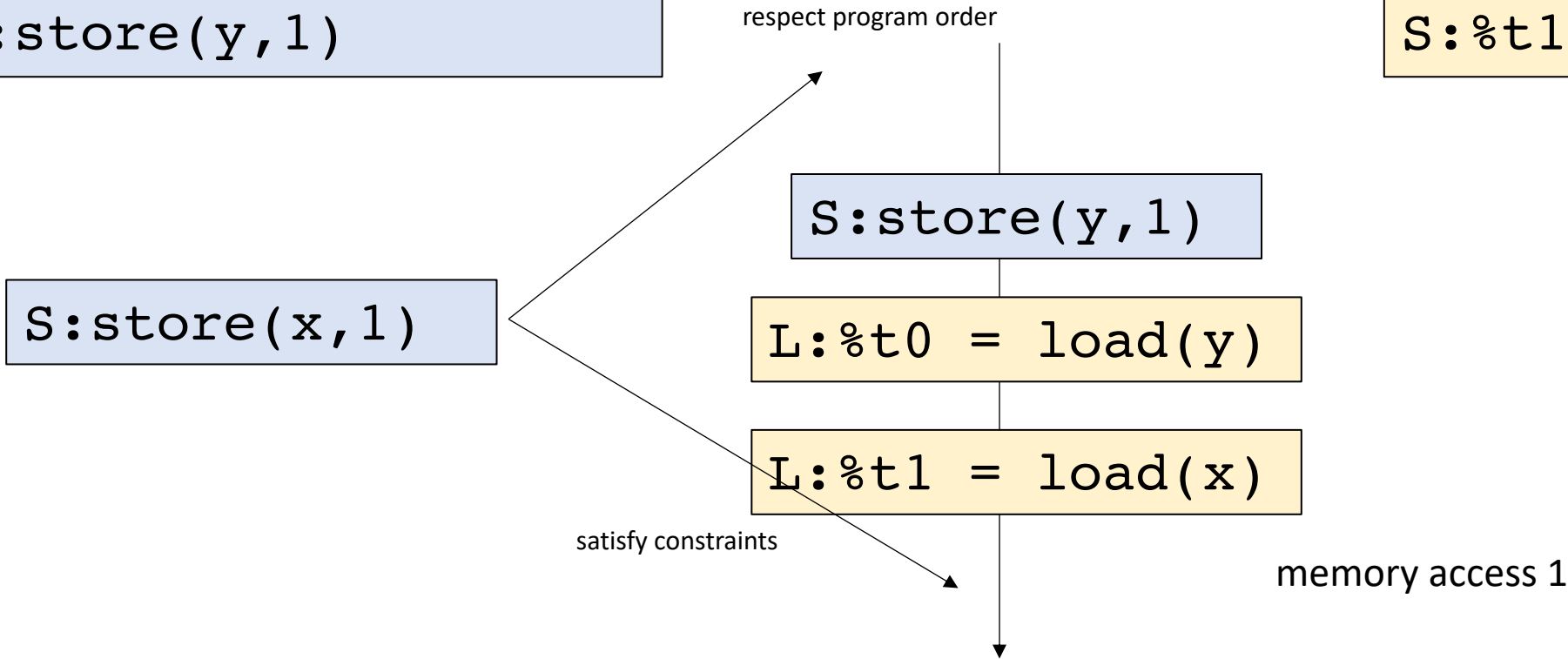
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

	L	S
L	NO	Different address
S	NO	Different address

What about PSO? YES

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
S:store(x,1)
```

respect program order

fence

```
S:store(y,1)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```

satisfy constraints

memory access 1

memory access 0

L S

L

NO

Different address

S

NO

Different address

Now it is disallowed in PSO

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

S:store(x,1)

respect program order

fence

S:store(y,1)

L:%t0 = load(y)

L:%t1 = load(x)

satisfy constraints

memory access 1

memory access 0

L S

L

S

	L	S
L	YES	Different address
S	Different address	Different address

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

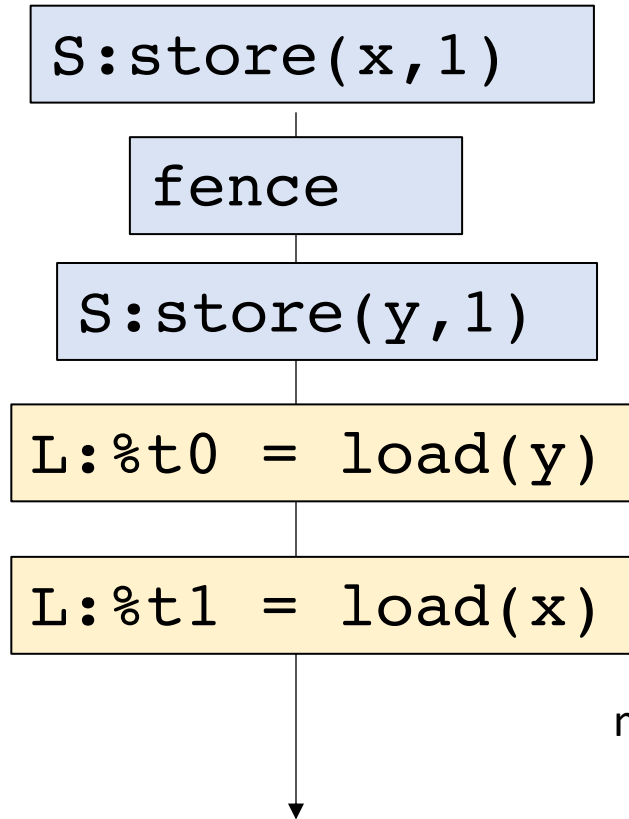
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 1

memory access 0

	L	S
L	YES	Different address
S	Different address	Different address

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

```
L:%t1 = load(x)
```

```
S:store(x,1)
```

fence

```
S:store(y,1)
```

```
L:%t0 = load(y)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

memory access 0

L S

L

	L	S
L	YES	Different address
S	Different address	Different address

memory access 1

S

What about RMO? The loads can be reordered also!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

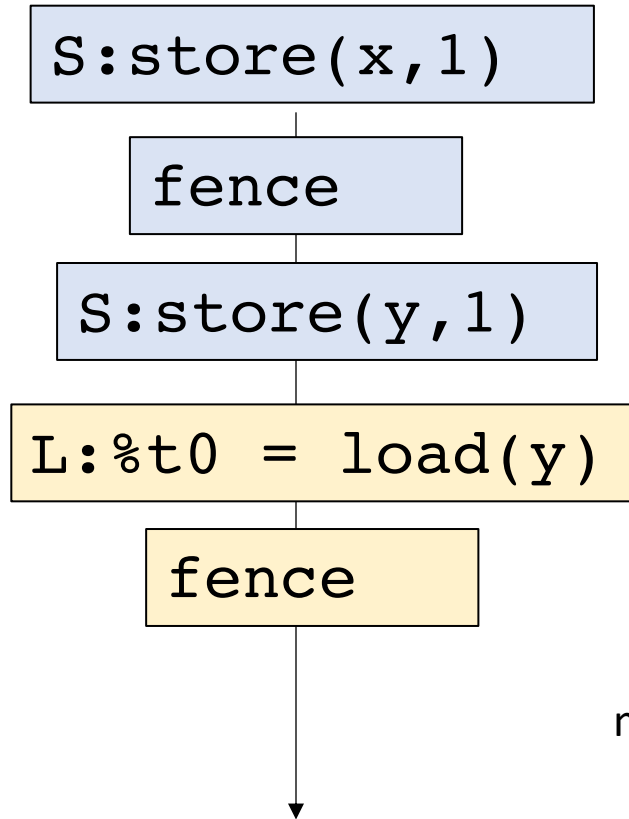
Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
fence  
S:%t1 = load(x)
```

```
L:%t1 = load(x)
```



memory access 0

	L	S
L	YES	Different address
S	Different address	Different address

memory access 1

What about RMO? add a fence



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

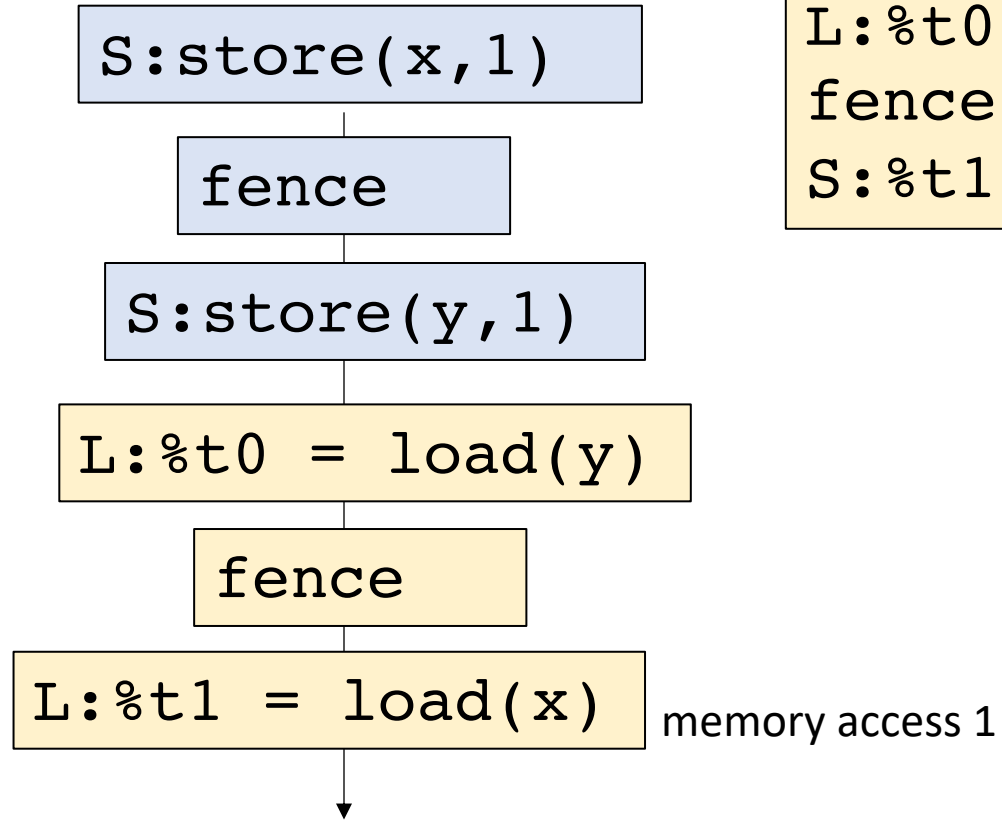
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
fence  
S:%t1 = load(x)
```



Now the relaxed behavior is disallowed

memory access 0

	L	S
L	YES	Different address
S	Different address	Different address

# Memory consistency in the real world

- Historic Chips:
  - X86: TSO
    - Surprising robust
    - mutexes and concurrent data structures generally seem to work
    - watch out for store buffering
  - IBM Power and ARM
    - Very relaxed. Similar to RMO with even more rules
    - Mutexes and data structures must be written with care
    - ARM recently strengthened theirs
- Very difficult to write correct code under! PPOP example

# Memory consistency in the real world

- Historic Chips:
  - X86: TSO
    - Surprising robust
    - mutexes and concurrent data structures generally seem to work
    - watch out for store buffering
  - IBM Power and ARM
    - Very relaxed. Similar to RMO with even more rules
    - Mutexes and data structures must be written with care
    - ARM recently strengthened theirs
- Very difficult to write correct code under! PPOP example

Companies have a history of providing insufficient documentation about their rules: academics have then gone and figured it out!

Getting better these days

# Memory consistency in the real world

- Modern Chips:
  - RISC-V : two specs: one similar to TSO, one similar to RMO
  - Apple M1: toggles between TSO and weaker
- Vulkan does not provide any fences that provide S - L ordering

# Memory consistency in the real world

- PSO and RMO were never implemented widely
  - I have not met anyone who knows of any RMO taped out chip
  - They are part of SPARC ISAs (i.e. RISC-V before it was cool)
  - These memory models might have been part of specialized chips
- Interestingly:
  - Early Nvidia GPUs appeared to informally implement RMO
- Other chips have very strange memory models:
  - Alpha DEC - basically no rules

# Next week

- Finish up memory models:
  - Compilers
- Execution barriers
- Watch for midterm grades sometime today