

CSE211: Compiler Design

Dec. 1, 2023

- **Topic:** Compiling weak memory models

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

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

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

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

Announcements

- Homework 3 is due on TODAY
 - Had an extension
 - No extension on HW 4
 - It was released on Wednesday
 - It is due on Dec. 15. No extensions will be possible.
 - Have partners by Monday
- Second paper needs to be selected by Monday

Announcements

- Homework 3 is due on TODAY
 - Had an extension
 - No extension on HW 4
 - It was released on Wednesday
 - It is due on Dec. 15. No extensions will be possible.
 - Have partners by Monday
- Second paper needs to be selected by Monday

Announcements

- Final project presentations are required by Dec. 6 (Next Wednesday)
 - 10 minutes
 - Things don't have to be completed
 - But you should be able to present at least one result and your approach.
- By noon today you cannot back out of the project without consequence (e.g., 20%)
- I will randomly select a subset of people to do final presentations in class over Dec. 6 and Dec. 8.
- If you are not selected, then a zoom recording of your presentation is due on Dec. 8.

Announcements

- Final Exam:
 - Tuesday Dec 12: 8 AM – 11 AM
 - 3 pages of notes allowed
 - Inclusive material
 - Same style as midterm, but probably ~2x as long.

Announcements

- SETs are out!
 - Please fill them out!
 - Helps us improve
 - Helps us justify the class

Announcements

- Busy time for everyone!
- Please manage your time and be on time:
 - Very difficult (impossible?) to give extensions at the end of the quarter
- You got this!

Review

Hope everyone enjoyed the guest lectures!

Memory model review

Global variable:

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

Thread 0:

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

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

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

Thread 1:

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

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

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



pick from the top of the pile of either thread

Sequential Consistency

- Sequential interleaving of atomic instructions
- What are "atomic instructions"?

Global variable:

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

Thread 0:

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

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

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

Thread 1:

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

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

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



pick from the top of the pile of either thread
Can `t0 == t1 == 0` at the end of the execution?

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
```

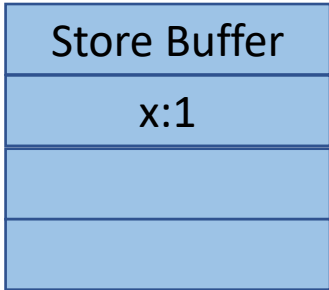
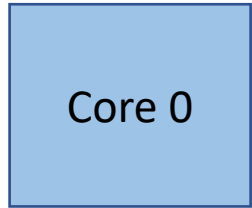
x:0

y:0

Main Memory

Thread 0:

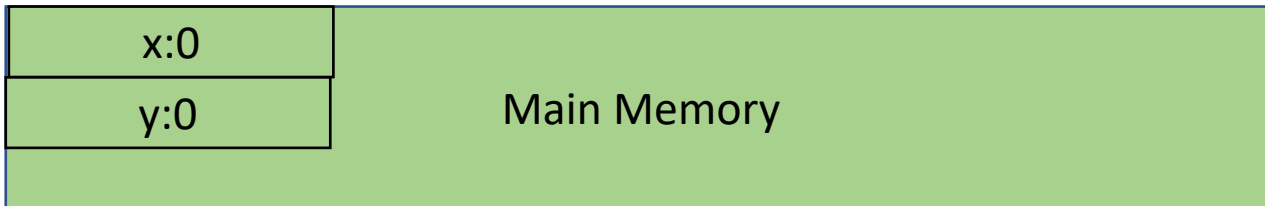
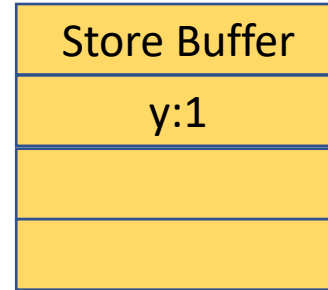
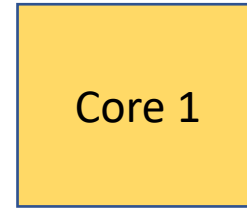
```
mov %t0, [y]
```



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

Thread 1:

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



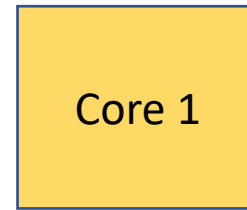
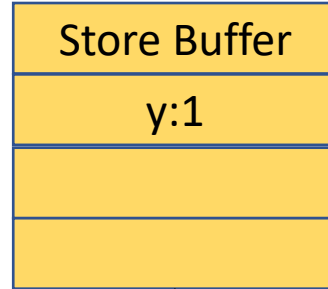
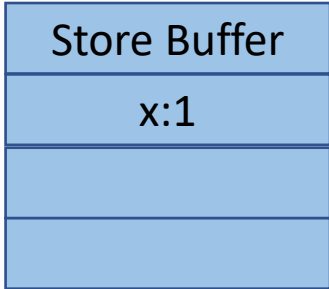
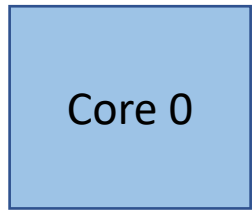
Thread 0:

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

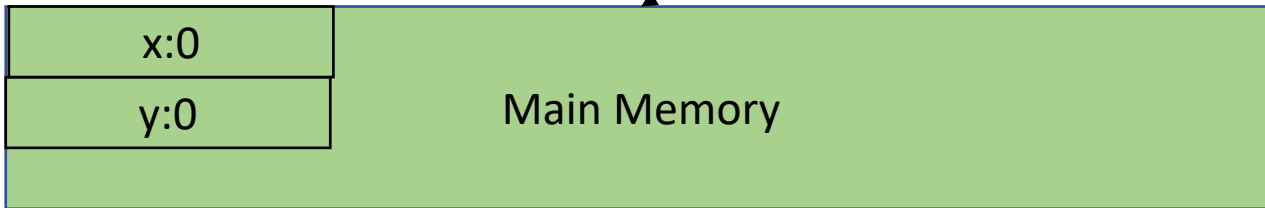
Thread 1:

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

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



eventually they flush to main memory



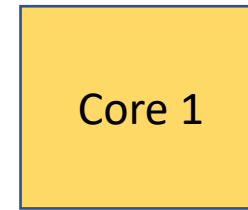
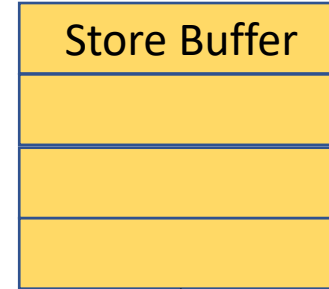
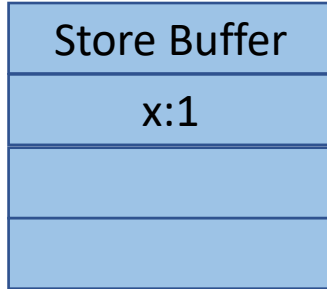
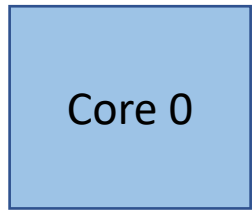
Thread 0:

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

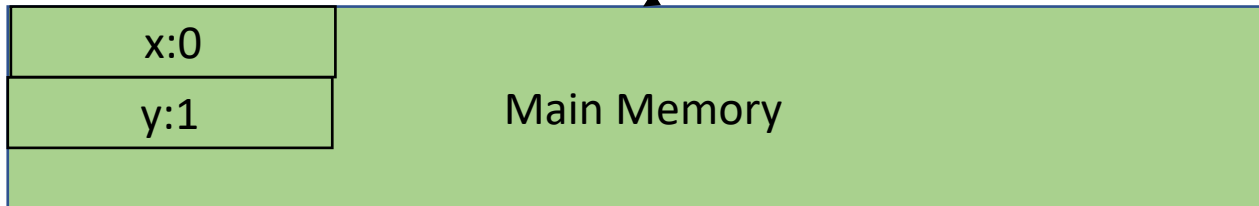
Thread 1:

```
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 [x], 1
```

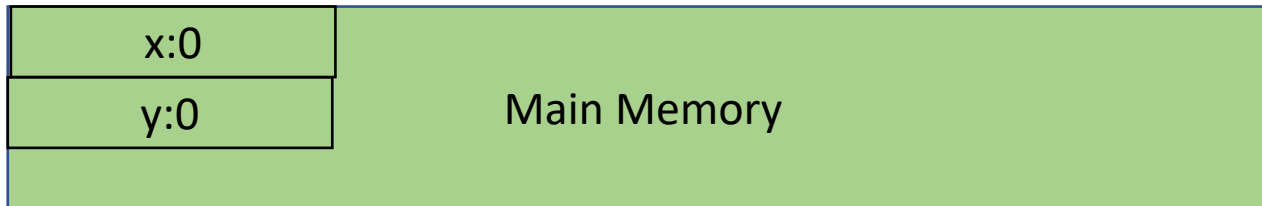
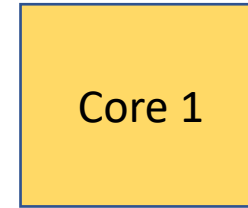
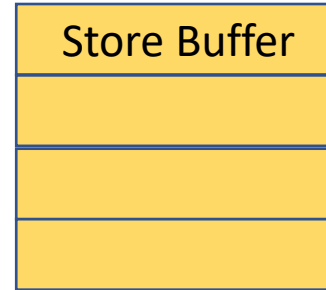
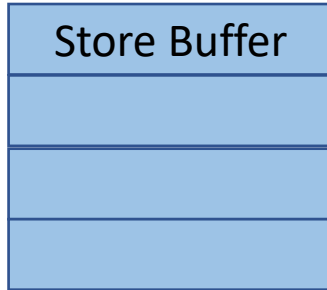
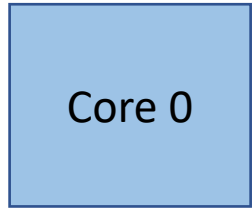
```
mov %t0, [y]
```

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

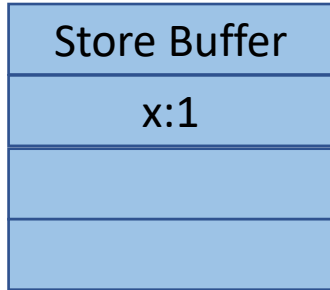
y:0

Main Memory

Thread 0:

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

Core 0

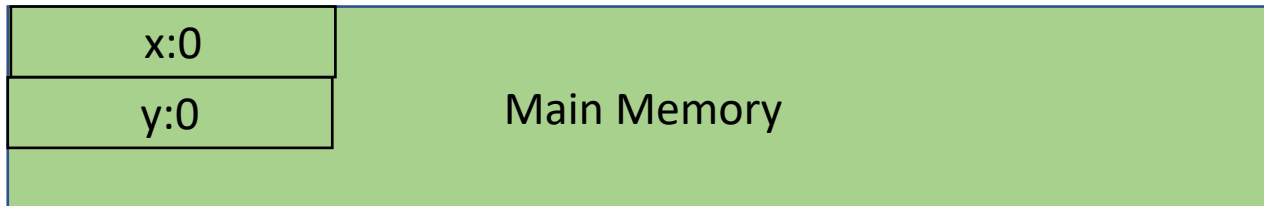
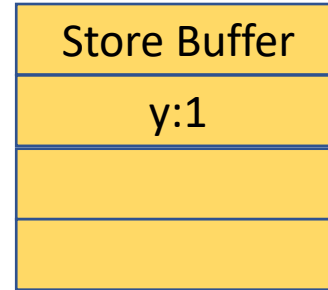


values get stored in SB

Thread 1:

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

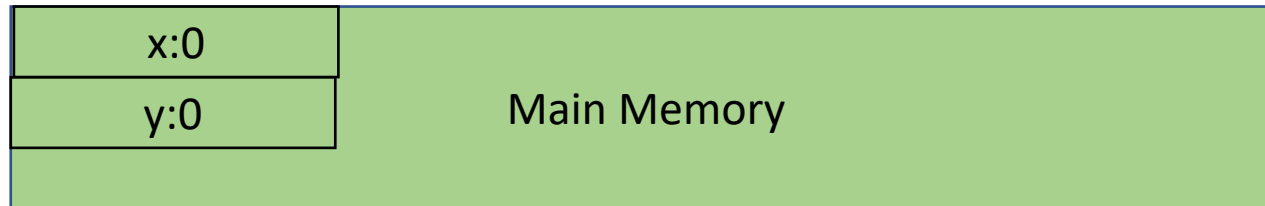
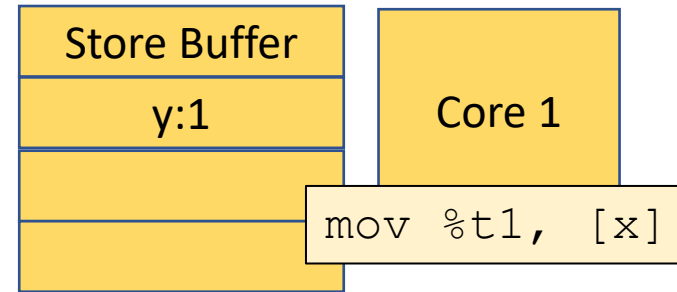
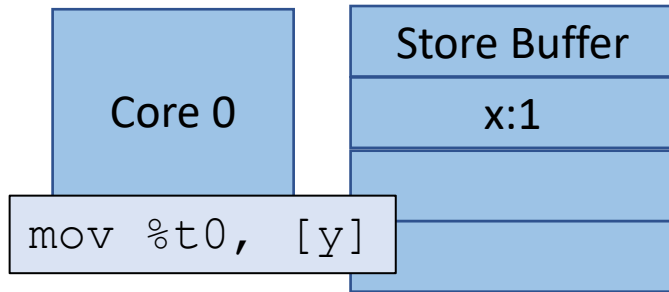
Core 1



Thread 0:

Thread 1:

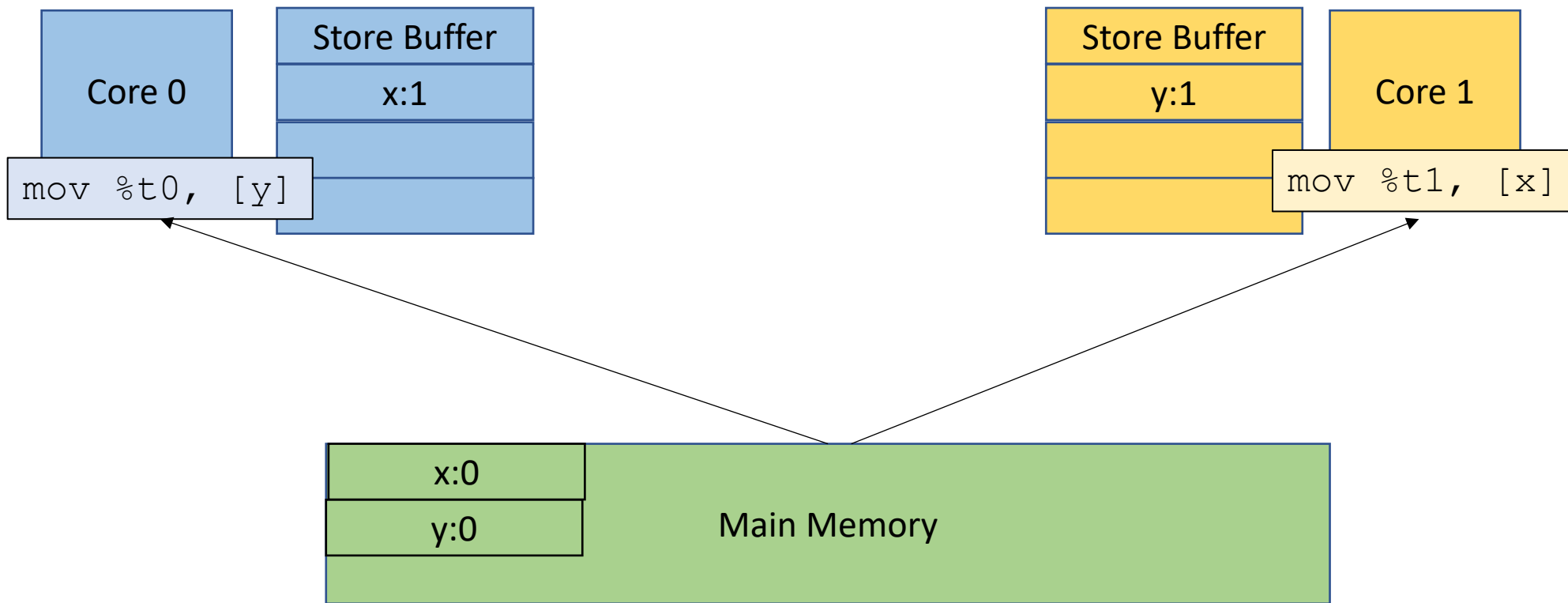
Execute next instruction



Thread 0:

Thread 1:

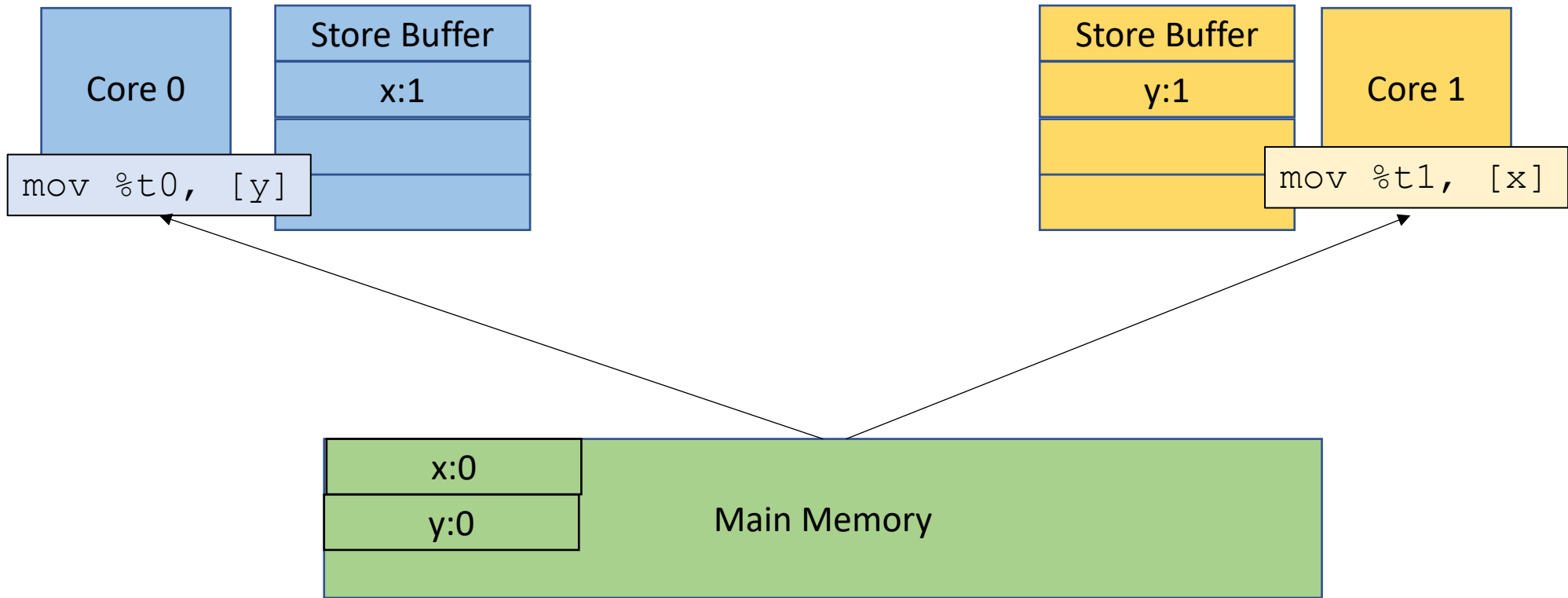
Values get loaded from memory



Thread 0:

Thread 1:

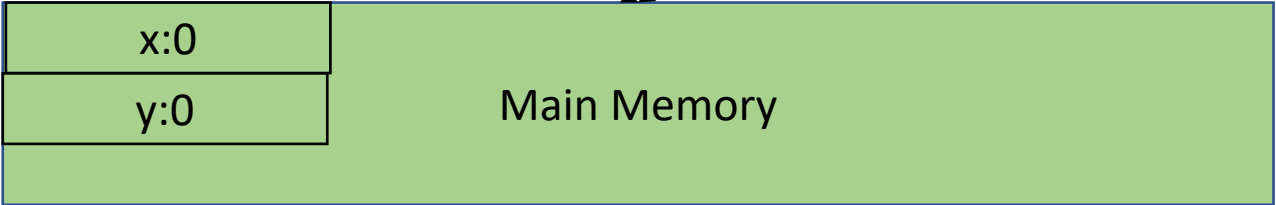
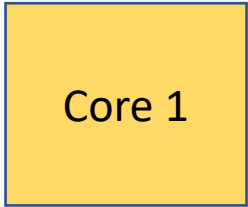
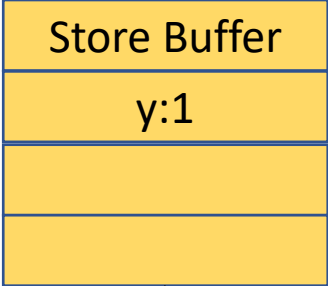
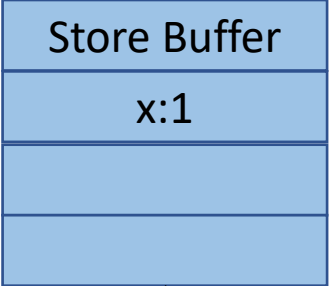
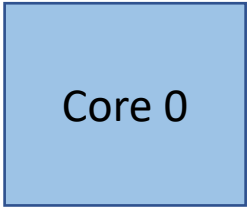
we see `t0 == t1 == 0!`



Thread 0:

Thread 1:

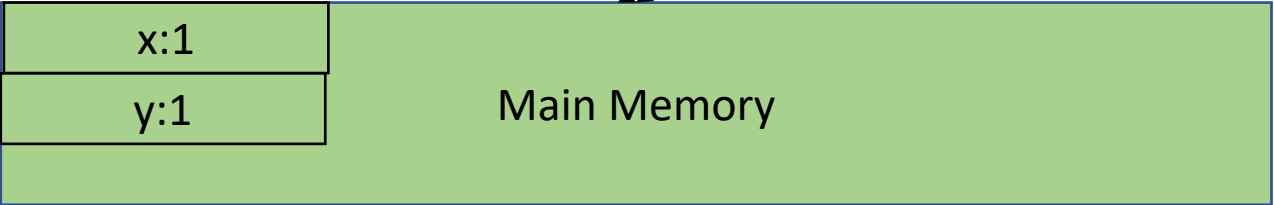
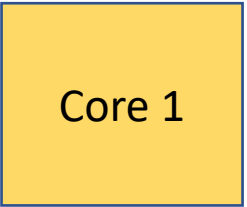
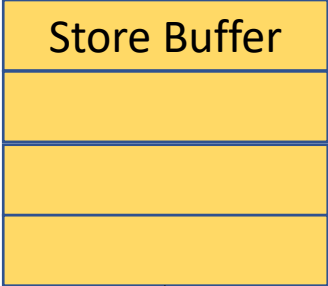
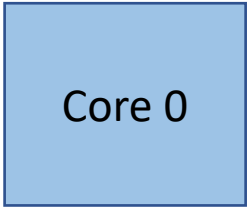
Store buffers are drained eventually



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

Litmus tests

- Small concurrent programs that check for relaxed memory behaviors
- Vendors have a long history of under documenting 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]
```

Can `t0 == t1 == 0`?

New material

Restoring sequential consistency

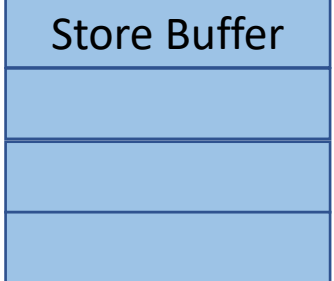
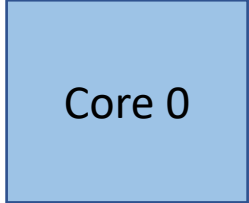
- It is typical that relaxed memory models provide special instructions which can be used to disallow weak behaviors.
- These instructions are called Fences
- The X86 fence is called `mfence`. It flushes the store buffer.

Thread 0:

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

```
mfence
```

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

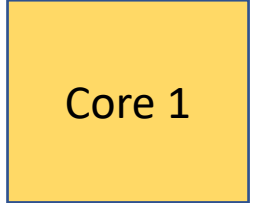
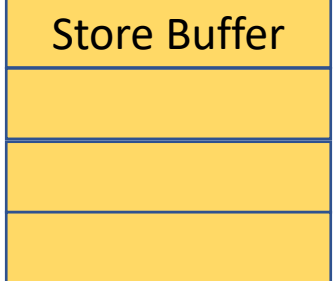


Thread 1:

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

```
mfence
```

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



Thread 0:

mfence

mov %t0, [y]

Core 0

mov [x], 1

Store Buffer

Thread 1:

mfence

mov %t1, [x]

Core 1

mov [y], 1

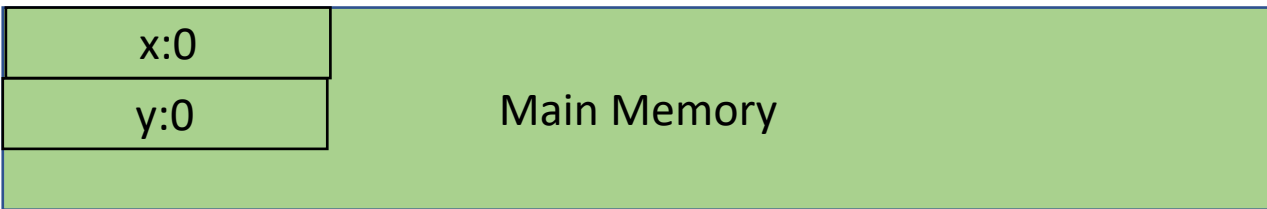
Store Buffer

Execute first instruction

x:0

y:0

Main Memory



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

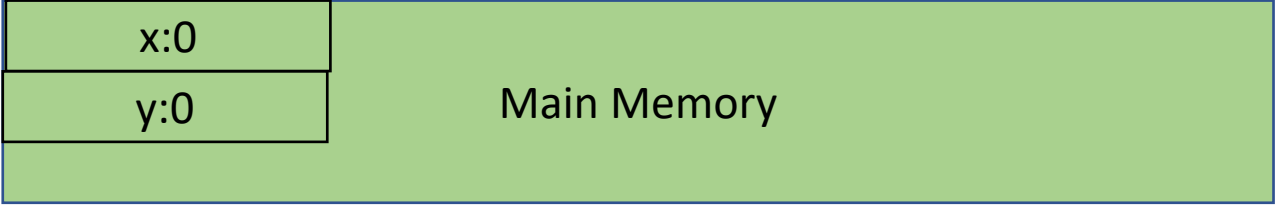
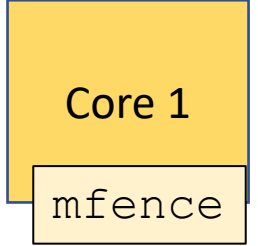
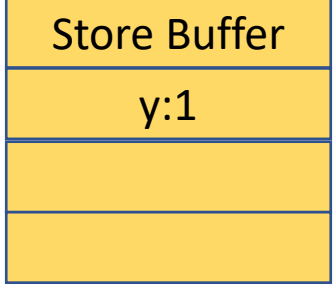
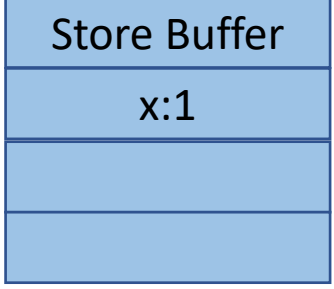
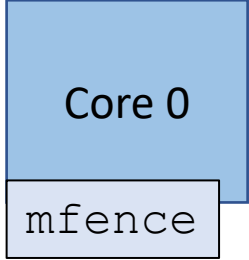
Thread 0:

Thread 1:

Execute next instruction

`mov %t0, [y]`

`mov %t1, [x]`



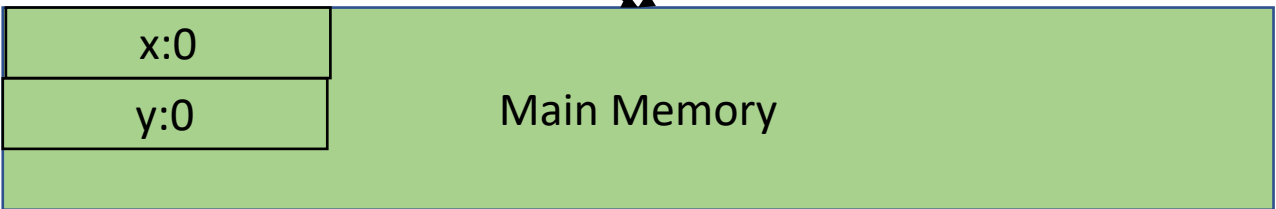
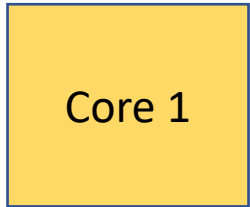
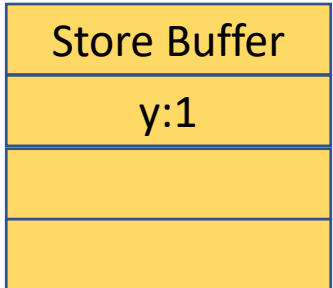
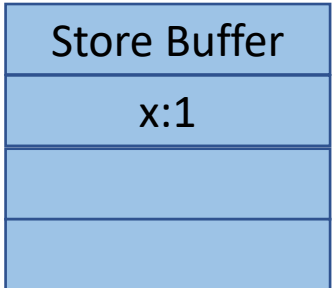
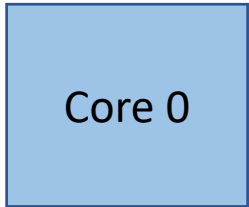
Thread 0:

Thread 1:

store buffers are flushed

`mov %t0, [y]`

`mov %t1, [x]`



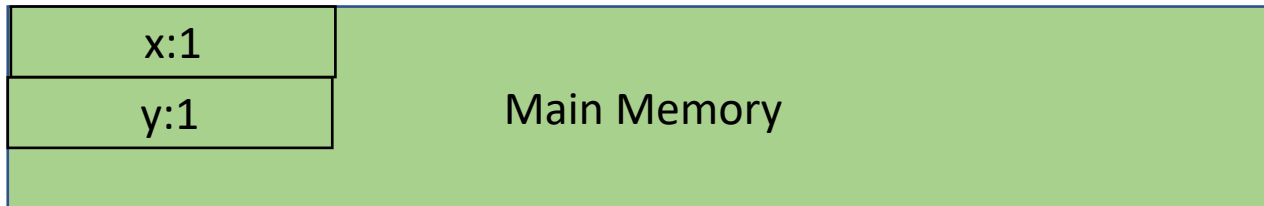
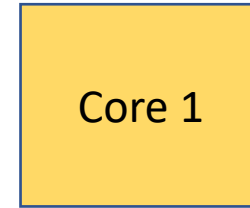
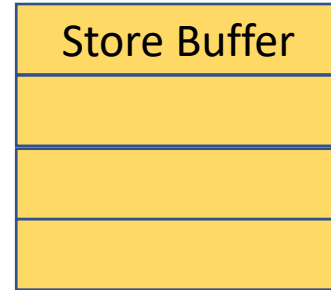
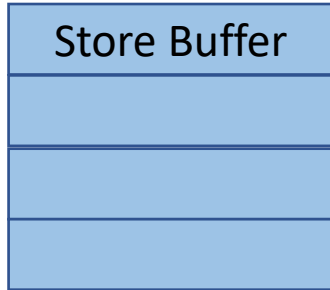
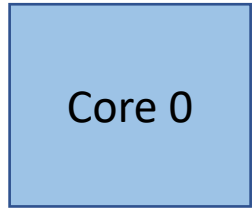
Thread 0:

Thread 1:

store buffers are flushed

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

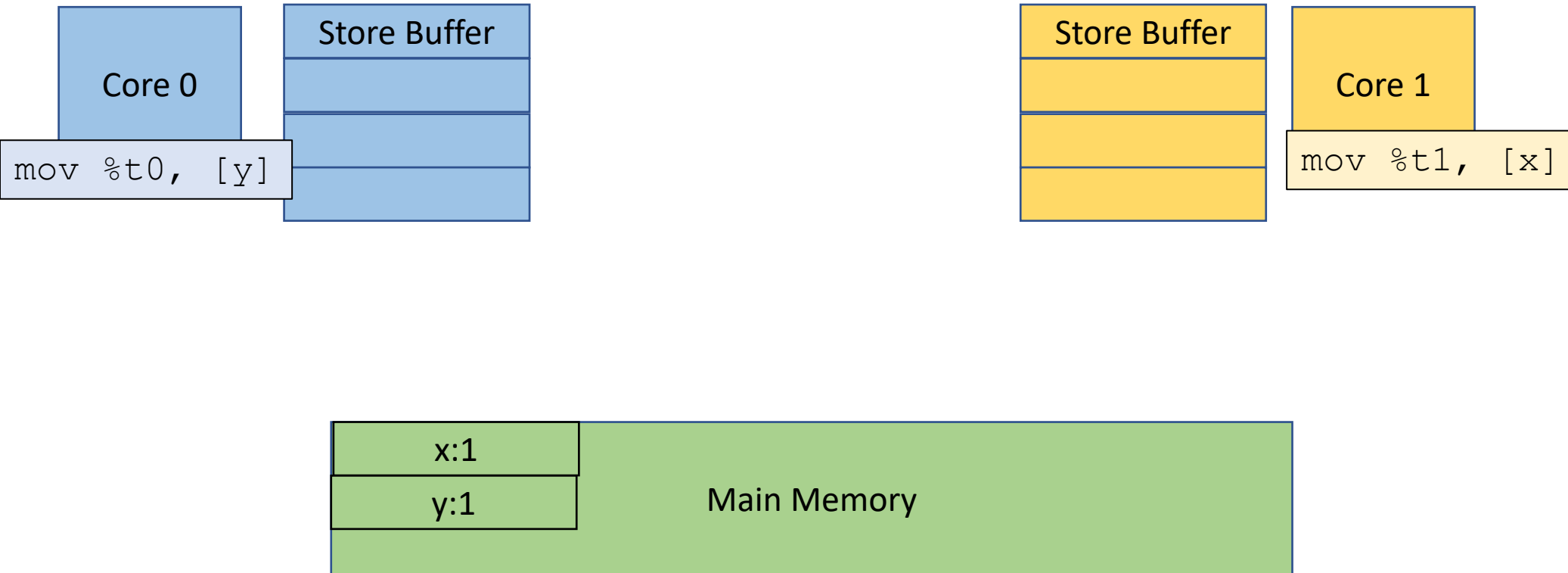
```
mov %t1, [x]
```



Thread 0:

Thread 1:

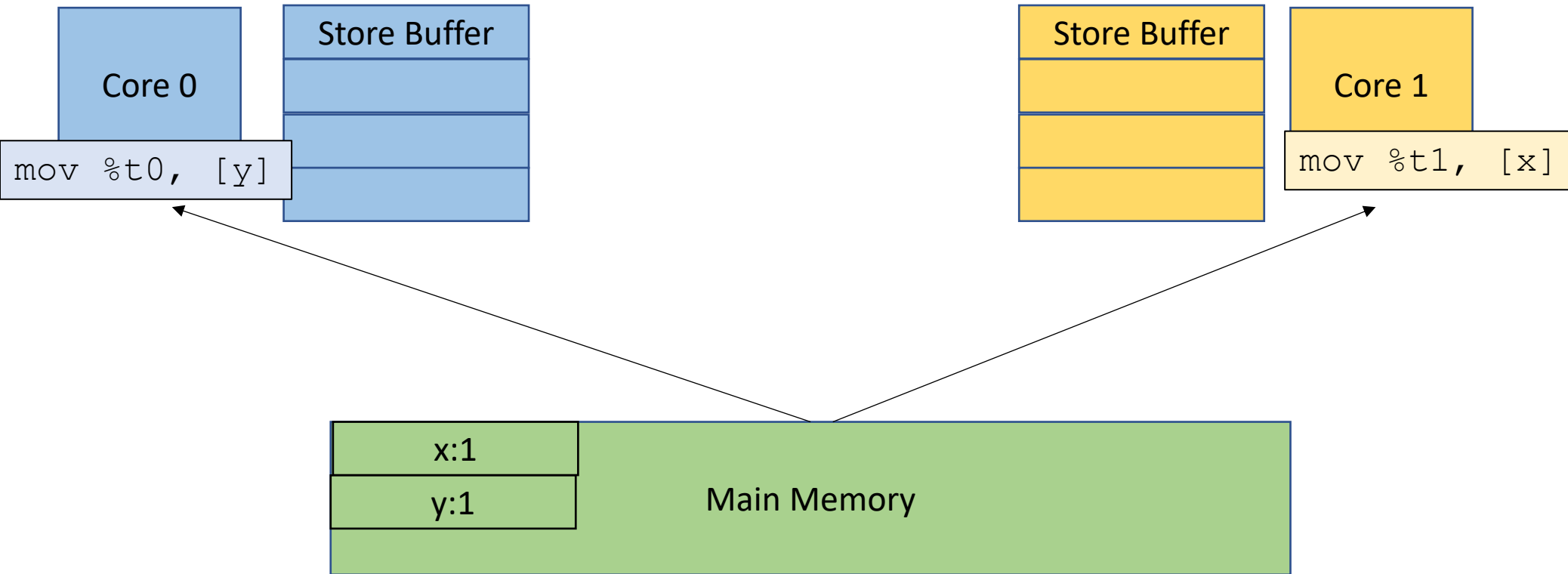
execute next instruction



Thread 0:

Thread 1:

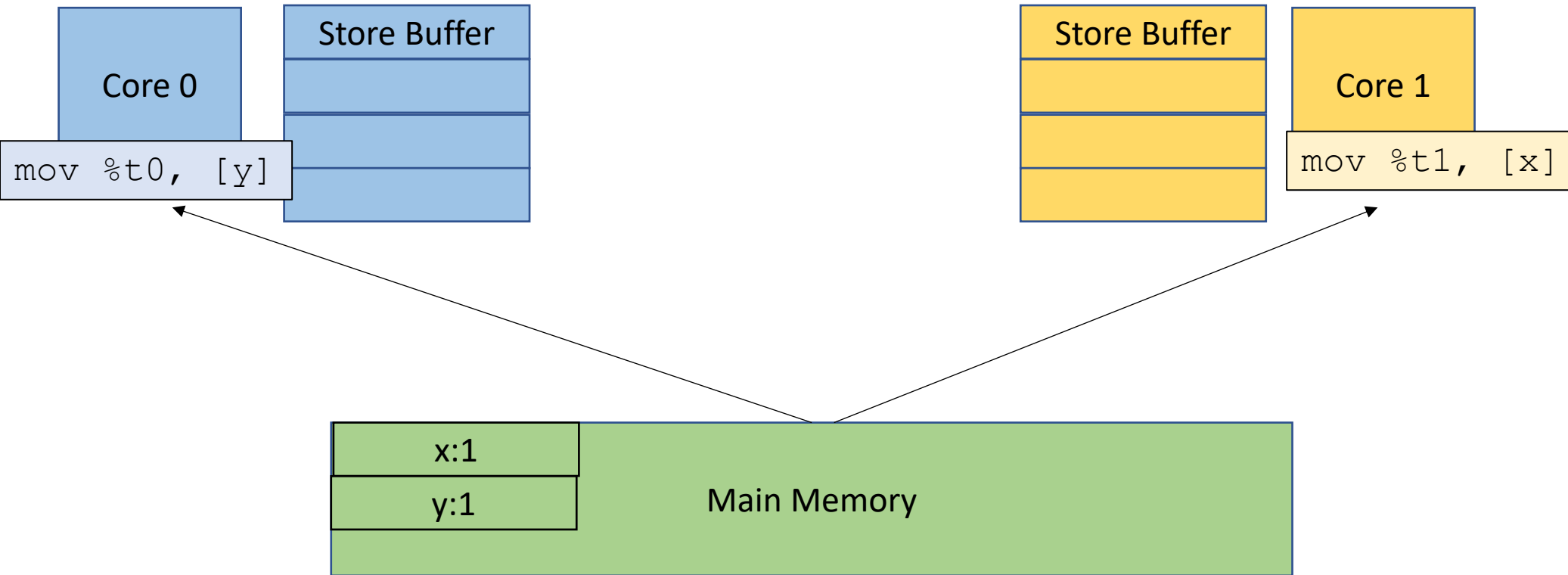
values are loaded from memory



Thread 0:

Thread 1:

We don't get the problematic behavior: `t0 != 0` and `t1 != 0`



Next example

Thread 0:

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

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

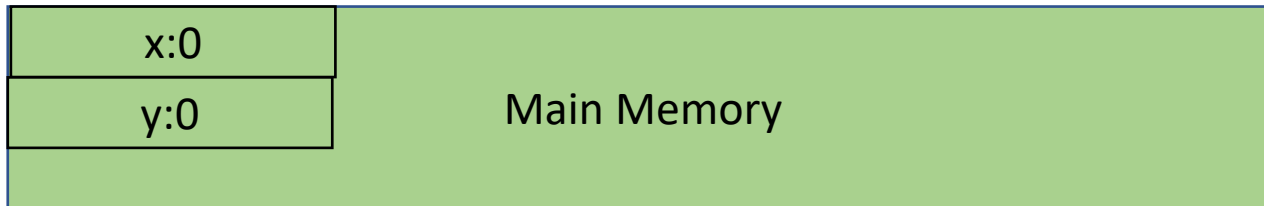
single thread
same address

possible outcomes:

t0 = 1

t0 = 0

Which one do you expect?

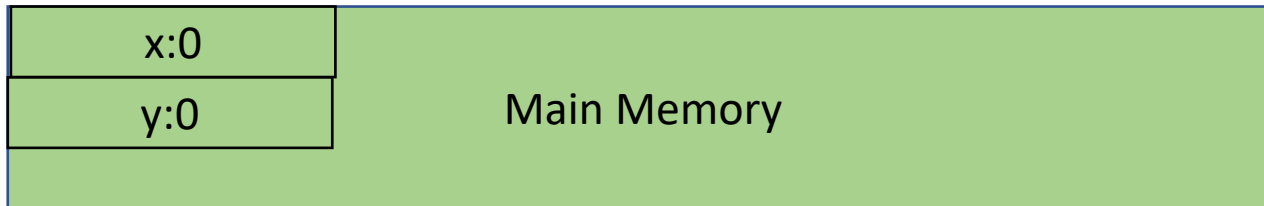
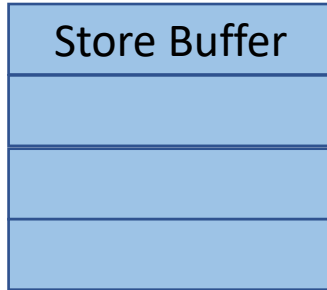
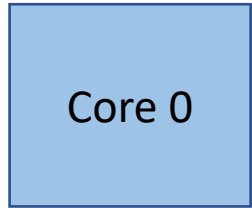


Thread 0:

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

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

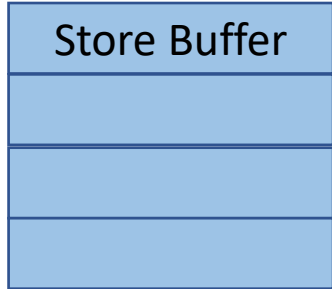
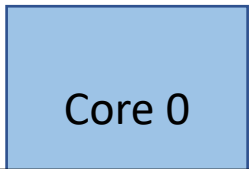
How does this execute?



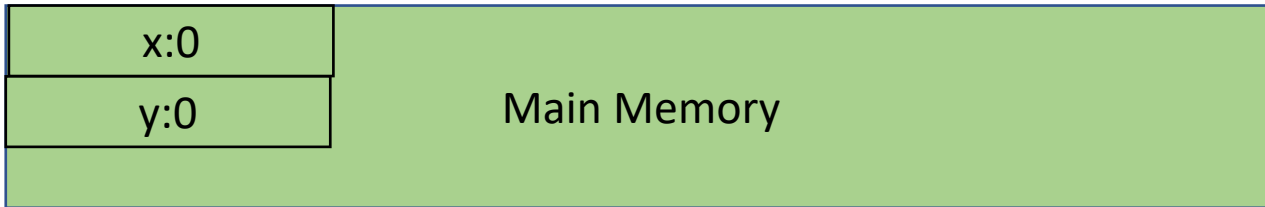
Thread 0:

execute first instruction

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



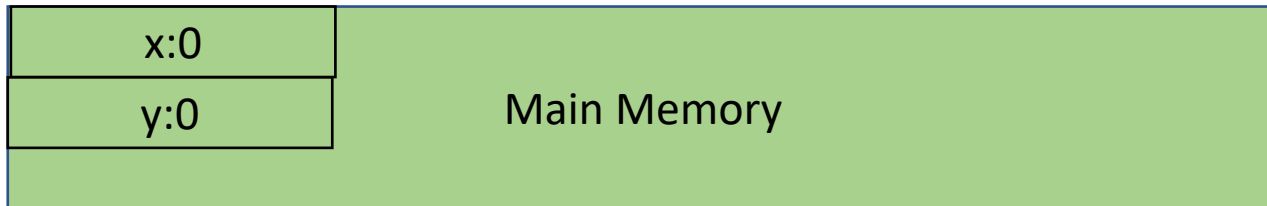
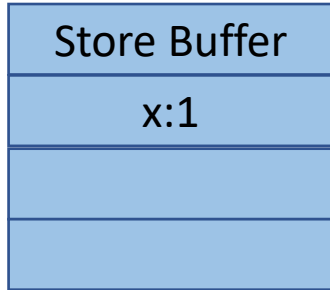
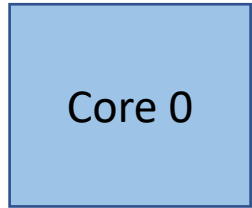
```
mov [x], 1
```



Thread 0:

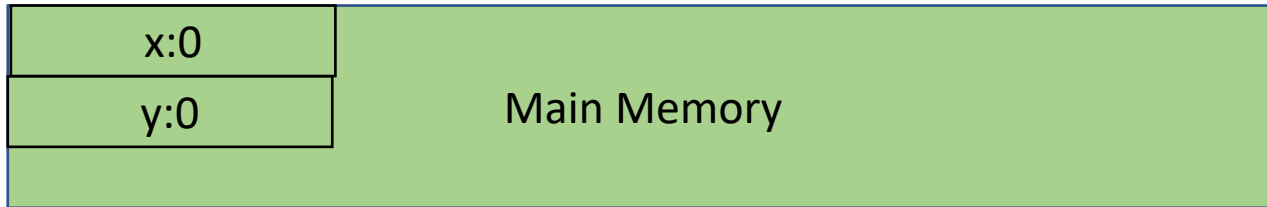
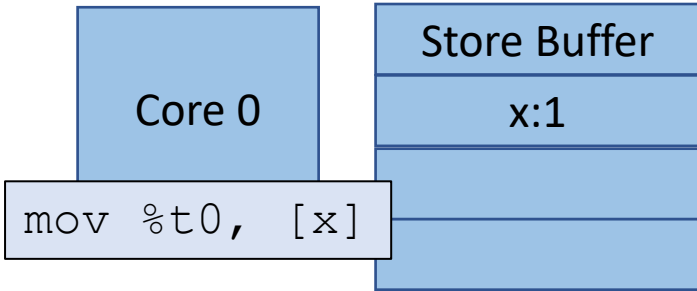
Store the value in the store buffer

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



Thread 0:

Next instruction

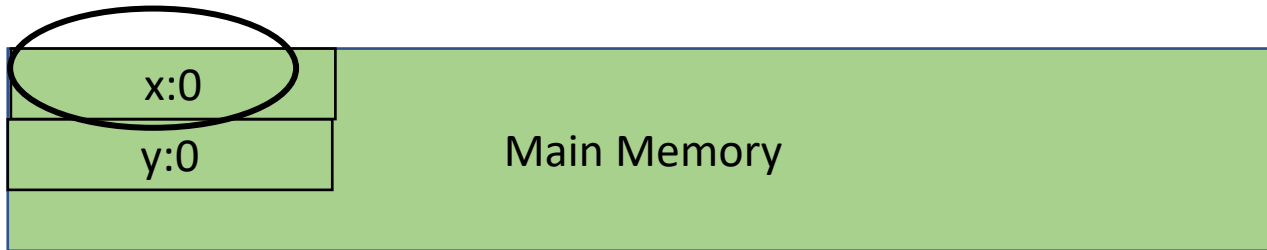
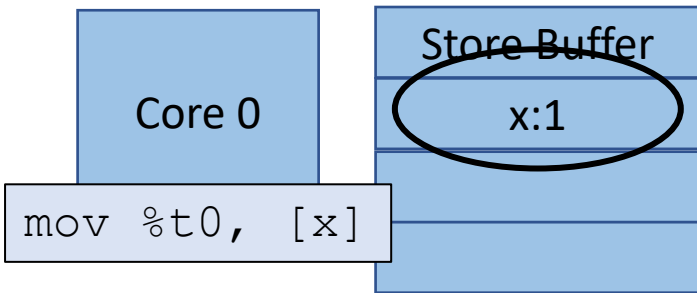


Thread 0:

Where to load??

Store buffer?

Main memory?

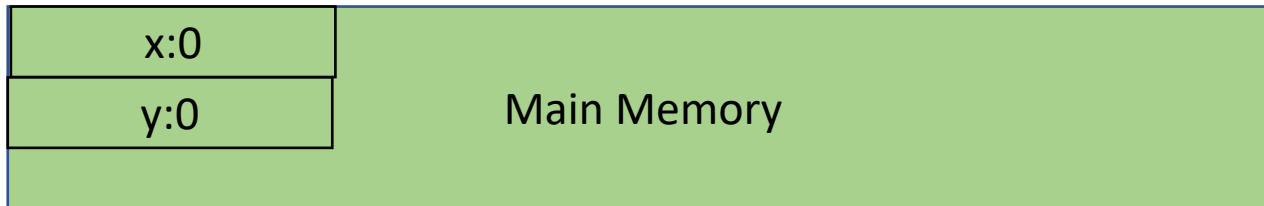
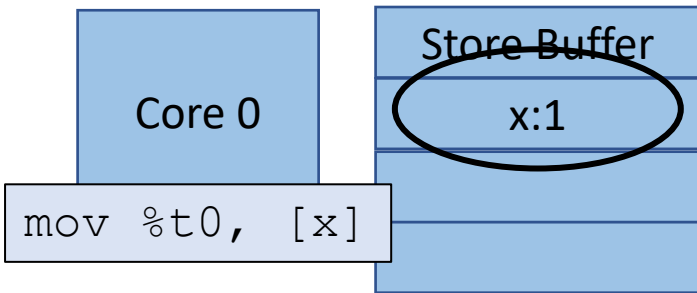


Thread 0:

Where to load??

Threads check store buffer before going to main memory

It is close and cheap to check.



Question

- Can stores be reordered with stores?

Global variable:

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

Thread 0:

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

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

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

Can `t0 == t1 == 0`?

Thread 1:

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

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

```
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]
```

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

```
mfence
```

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

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]
```

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

```
mfence
```

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

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.

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
```

```
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:

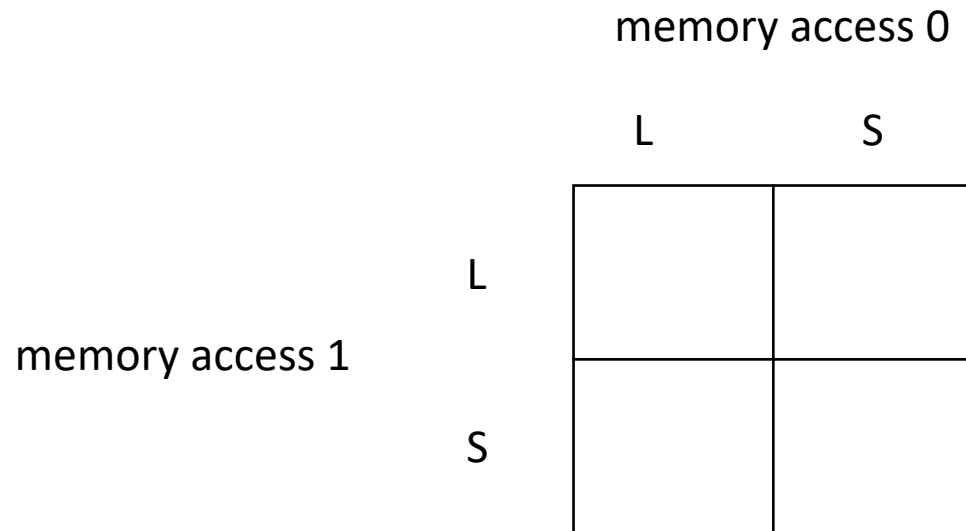
S(tores) followed by a L(oad)
do not have to follow program order.

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

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

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

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!

- GPUs?

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?

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};
```

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 TSO

Thread 1:

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

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

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



memory access 0

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | NO |

memory access 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 PSO

Thread 1:

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

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

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



memory access 0

| | L | S |
|-----------------|----|-------------------|
| L | NO | Different address |
| memory access 1 | | |
| S | NO | Different address |

Global variable:

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

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

Get out our lego bricks and try for RMO

Thread 0:

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

Thread 1:

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

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

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

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

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



memory access 0

L S

| | | |
|---|-------------------|-------------------|
| | L | S |
| L | YES | Different address |
| S | different address | Different address |

memory access 1

How do we disallow it?

Global variable:

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

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

Get out our lego bricks and try for RMO

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)
```

```
fence
```

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

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

```
fence
```

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



memory access 0

L S

| | |
|-------------------|-------------------|
| YES | Different address |
| different address | Different address |

memory access 1

S

How do we disallow it?

Compiling relaxed memory models

Compiling relaxed memory models

- C++ style:
 - Any memory conflicts (read-write or write-write) must be accessed with an atomic operation*
 - Otherwise your program is undefined
 - By default, you will get sequentially consistent behavior
- *unless they are synchronized, which is a really complicated concept in c++...
If you are interested, I can recommend papers.

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language

C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

target machine

| | L | S |
|---|---|---|
| L | ? | ? |
| S | ? | ? |

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

target machine
TSO (x86)

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | No |

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

find mismatch

target machine
TSO (x86)

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | No |

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

find mismatch

Two options:

make sure stores
are not reordered
with later loads

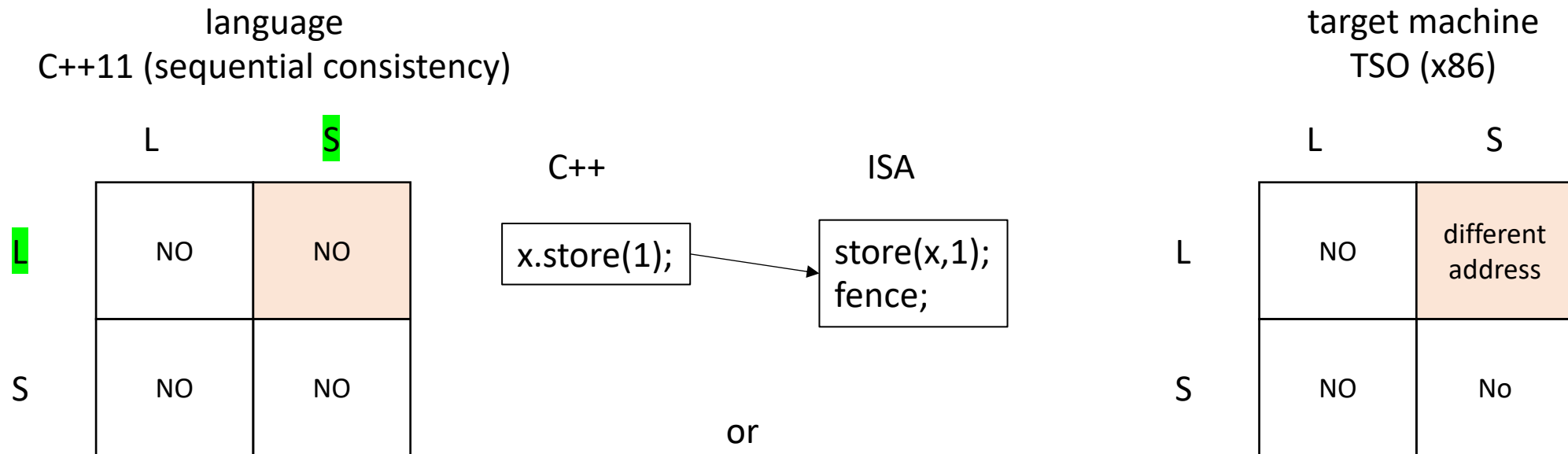
make sure loads
are not reordered
with earlier stores

target machine
TSO (x86)

| | L | S |
|---|----|----------------------|
| L | NO | different address |
| S | NO | No |

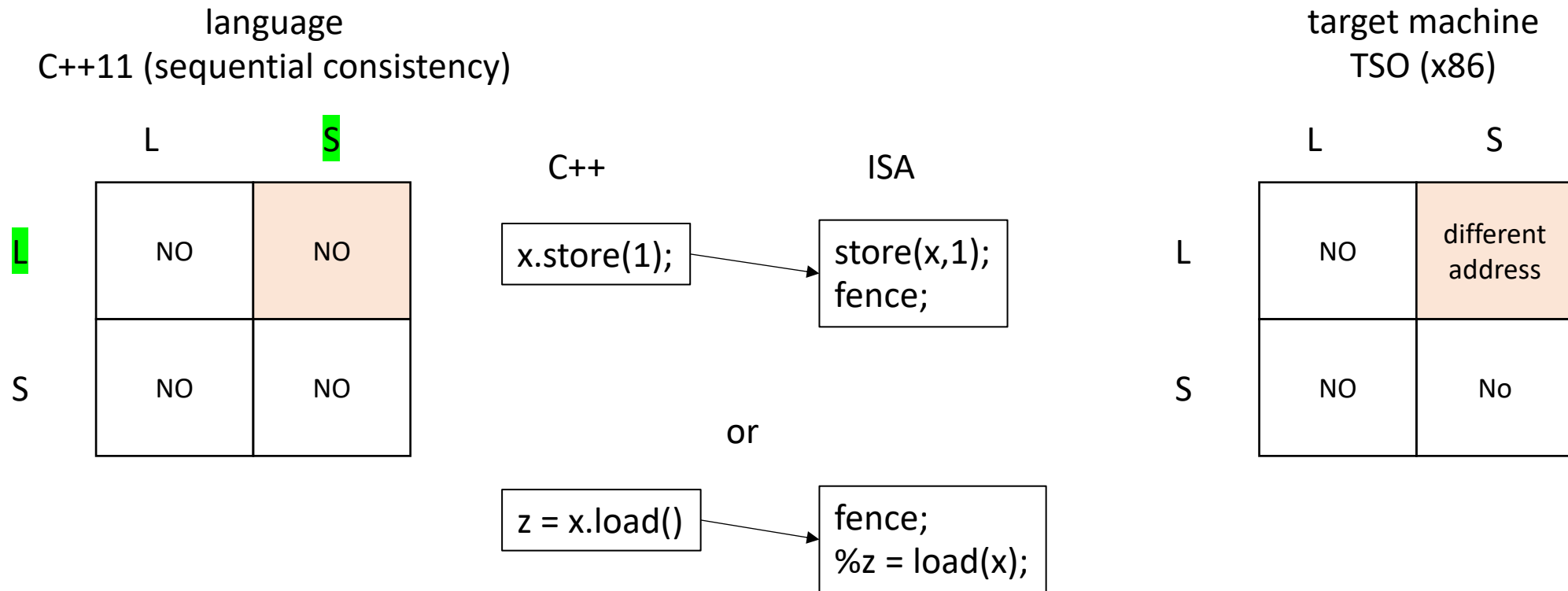
C++11 atomic operation compilation

start with both both of the grids for the two different memory models



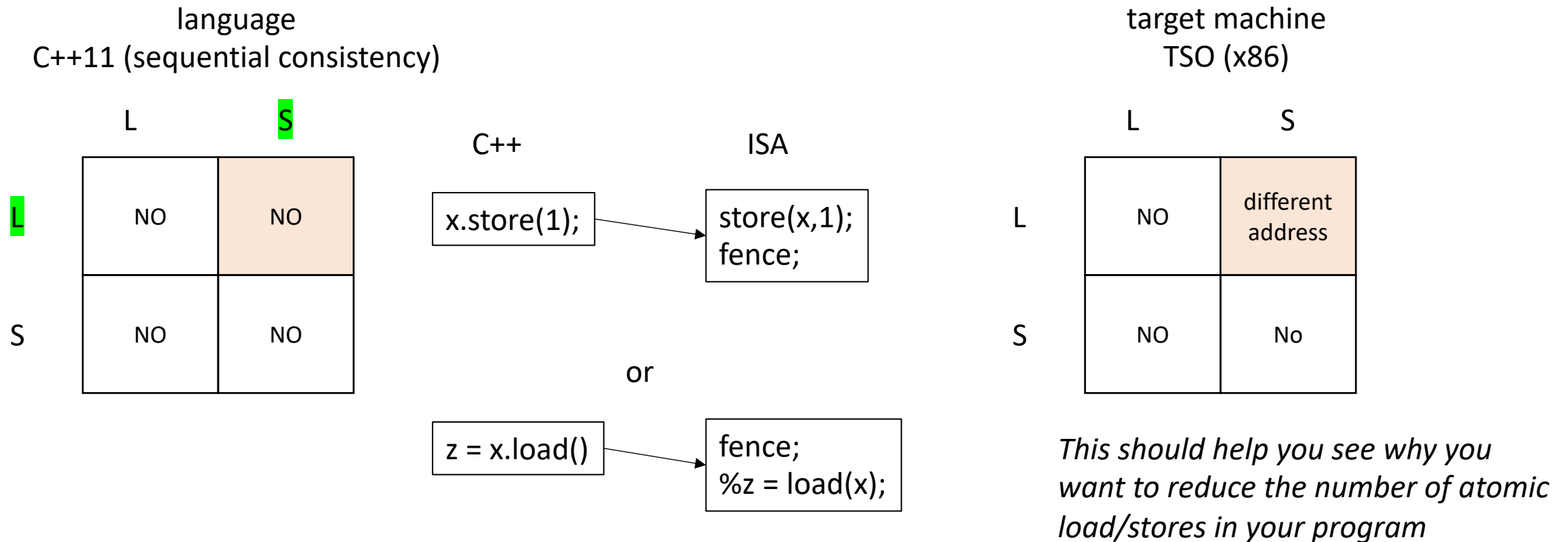
C++11 atomic operation compilation

start with both both of the grids for the two different memory models



C++11 atomic operation compilation

start with both both of the grids for the two different memory models



C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

How about this one?

target machine
PSO

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | different address |

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

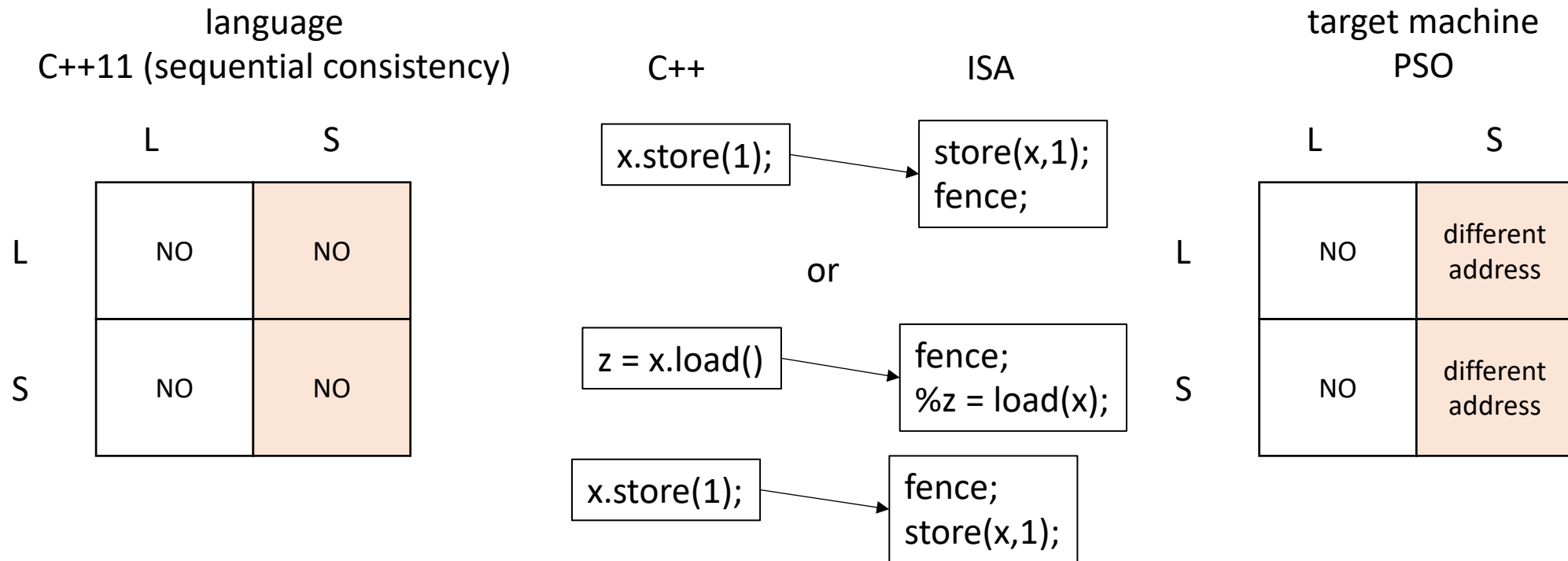
| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

target machine
PSO

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | different address |

C++11 atomic operation compilation

start with both both of the grids for the two different memory models



Memory orders

- Atomic operations take an additional “memory order” argument
 - `memory_order_seq_cst` - default
 - `memory_order_relaxed` - weakest

Relaxed memory order

language
C++11 (sequential consistency)

| | L | S |
|---|----|----|
| L | NO | NO |
| S | NO | NO |

language
C++11 (memory_order_relaxed)

| | L | S |
|---|-------------------|-------------------|
| L | different address | different address |
| S | different address | different address |

basically no orderings except for accesses to the same address

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

| | L | S |
|---|-------------------|-------------------|
| L | different address | different address |
| S | different address | different address |

target machine
TSO (x86)

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | No |

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

| | L | S |
|---|-------------------|-------------------|
| L | different address | different address |
| S | different address | different address |

lots of mismatches!

target machine
TSO (x86)

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | No |

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

| | L | S |
|---|-------------------|-------------------|
| L | different address | different address |
| S | different address | different address |

lots of mismatches!

But language is more relaxed than machine

so no fences are needed

target machine
TSO (x86)

| | L | S |
|---|----|-------------------|
| L | NO | different address |
| S | NO | No |

Compiling memory order relaxed

Do any of the ISA memory models need any fences for relaxed memory order?

language
C++11 (memory_order_relaxed)

| | L | S |
|---|-------------------|-------------------|
| L | different address | different address |
| S | different address | different address |

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | NO |

TSO

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |

PSO

| | L | S |
|---|-------------------|-------------------|
| L | YES | Different address |
| S | Different address | Different address |

RMO

Memory order relaxed

- Very few use-cases! Be very careful when using it
 - Peeking at values (later accessed using a heavier memory order)
 - Counting (e.g. number of finished threads in work stealing)

More memory orders: we will not discuss in class

- Atomic operations take an additional “memory order” argument
 - `memory_order_seq_cst` - default
 - `memory_order_relaxed` - weakest
- More memory orders (useful for mutex implementations):
 - `memory_order_acquire`
 - `memory_order_release`
- EVEN MORE memory orders (complicated: in most research it is omitted)
 - `memory_order_consume`

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

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

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

Compiler

- Previously (before C/++11):
 - Use volatile
 - Use inline assembly for fences
 - Not portable!
- Now:
 - C/++11 memory model
 - But there are still bugs: Intel OpenCL compiler, IBM C++ compiler...

Further research

- Should we provide sequential consistency by default? even without atomics?
 - How to do this?
 - Many interesting papers

A cautionary tale

Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)

Thread 0:

```
m.lock();  
display.enq(triangle0);  
m.unlock();
```

Thread 1:

```
m.lock();  
display.enq(triangle1);  
m.unlock();
```

Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)

Thread 0:

```
m.lock();  
display.enq(triangle0);  
m.unlock();
```

Thread 1:

```
m.lock();  
display.enq(triangle1);  
m.unlock();
```

We know how lock and unlock are implemented

Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)

Thread 0:

```
SPIN:CAS (mutex, 0, 1) ;  
display.enq (triangle0) ;  
store (mutex, 0) ;
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1) ;  
display.enq (triangle1) ;  
store (mutex, 0) ;
```

We know how lock and unlock are implemented
We also know how a queue is implemented

*Consider the following example: a graphics program where each thread wants to display a triangle;
the display is a queue (not thread safe)*

Thread 0:

```
SPIN:CAS(mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS(mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

We know how lock and unlock are implemented

We also know how a queue is implemented

What is an execution?

Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

CAS (mutex, 0, 1);

*if blue goes first
it gets to complete
its critical section
while thread 1 is spinning*



Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

CAS (mutex, 0, 1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex, 0);



Thread 0:

```
SPIN:CAS(mutex,0,1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex,0);
```

Thread 1:

```
SPIN:CAS(mutex,0,1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex,0);
```

CAS(mutex,0,1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex,0);

now yellow gets a change to go



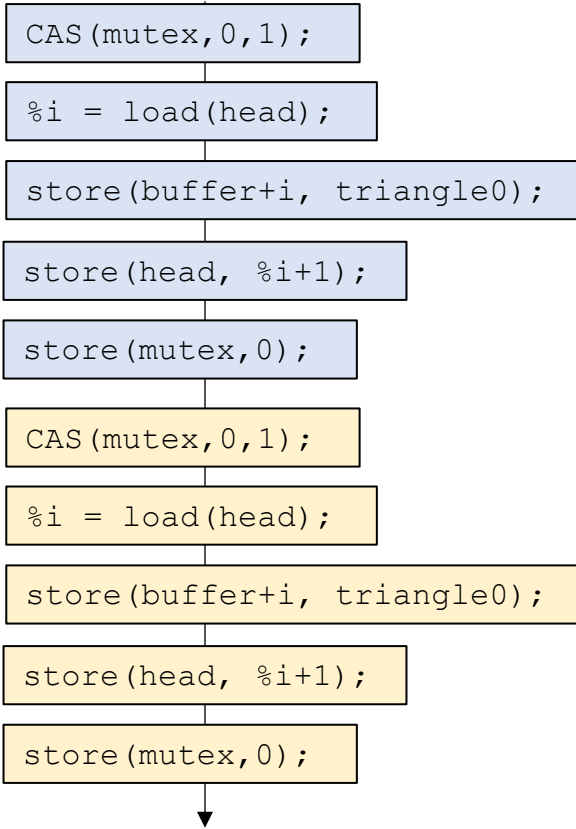
Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

now yellow gets a change to go



Thread 0:

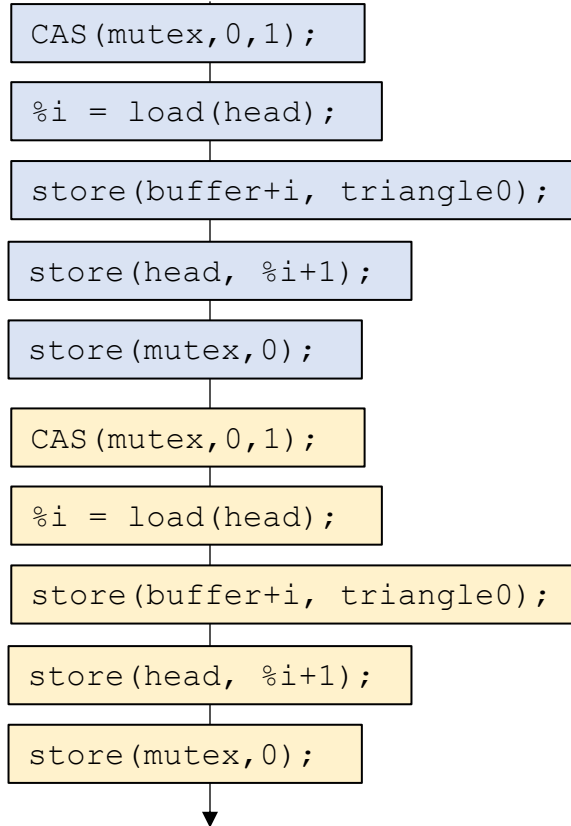
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |



Thread 0:

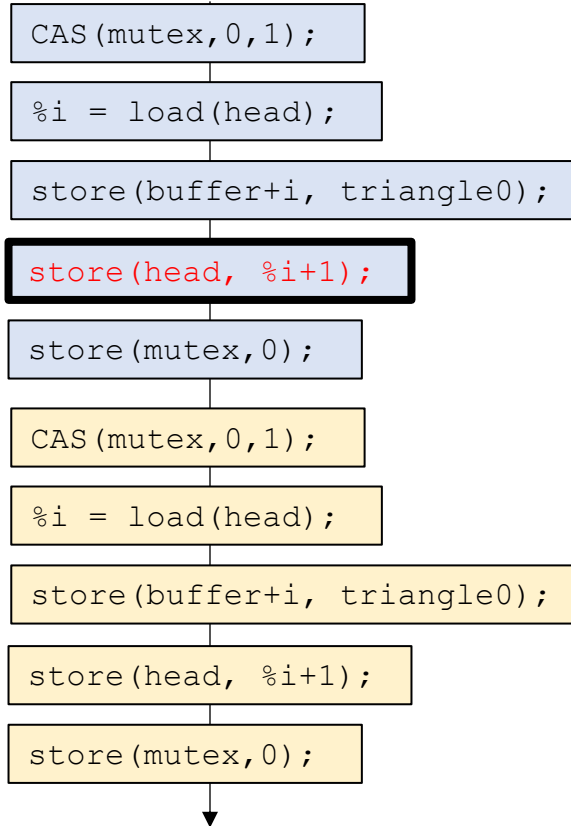
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |



Thread 0:

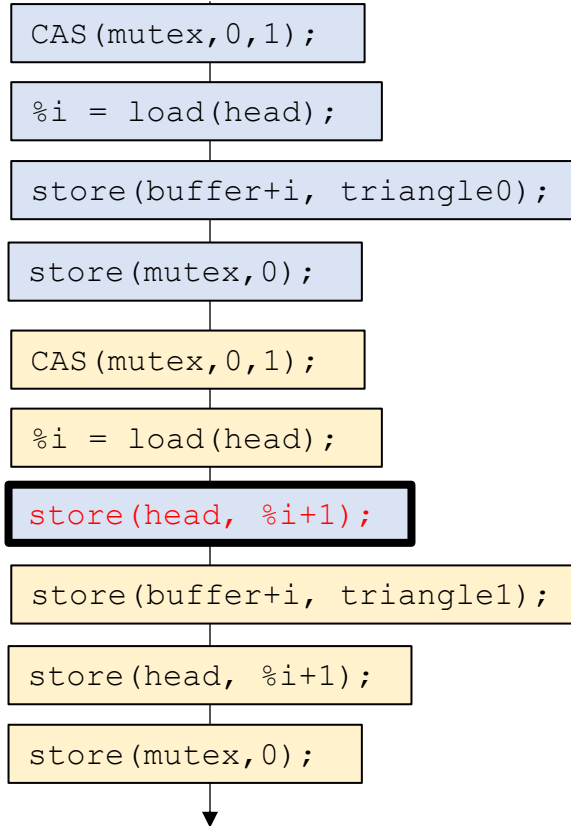
```
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex, 0);
```

what can happen in a PSO memory model?

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |

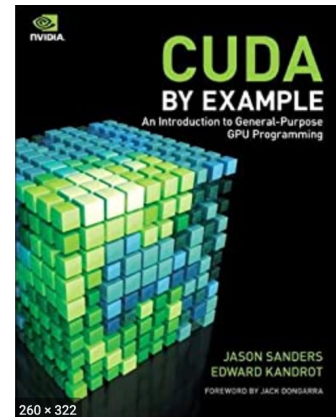
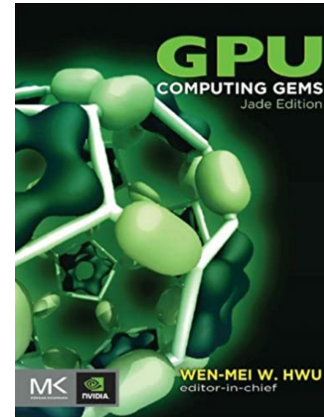
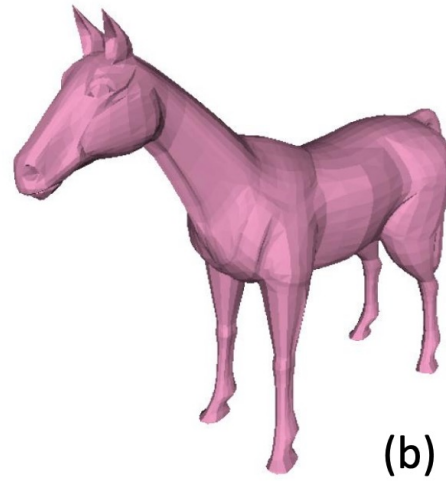
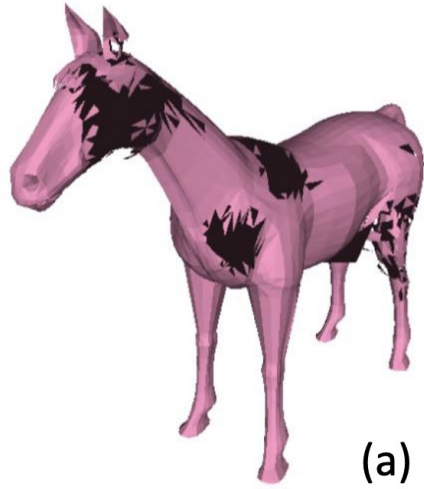


What just happened if this store moves?

Nvidia in 2015

- Nvidia architects implemented a weak memory model
- Nvidia programmers expected a strong memory model
- Mutexes implemented without fences!

Nvidia in 2015



bug found in two
Nvidia textbooks

We implemented
a side-channel attack
that made the bugs
appear more frequently

These days Nvidia has
a very well-specified
memory model!

Thread 0:

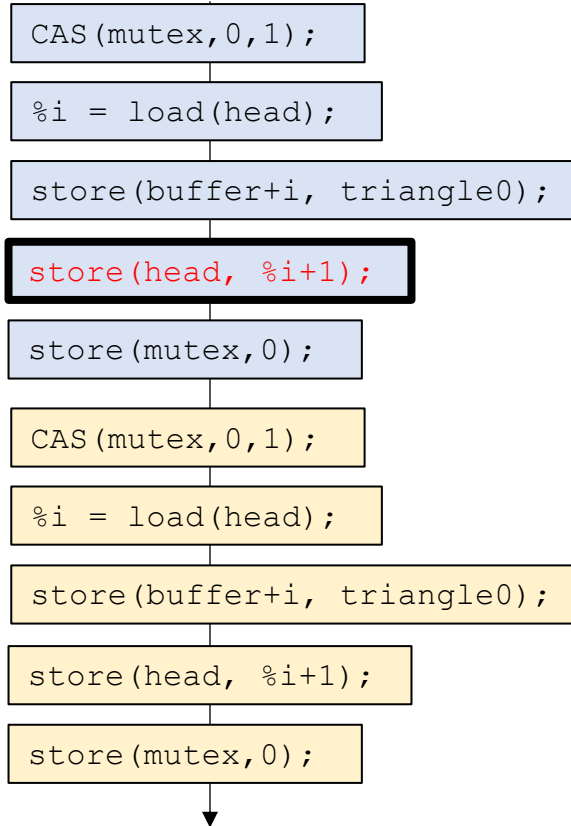
```
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex, 0);
```

what can happen in a PSO memory model?

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |



How to fix the issue?

Thread 0:

```

SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
fence;
store(mutex, 0);

```

unlock contains fence before store!

Thread 1:

```

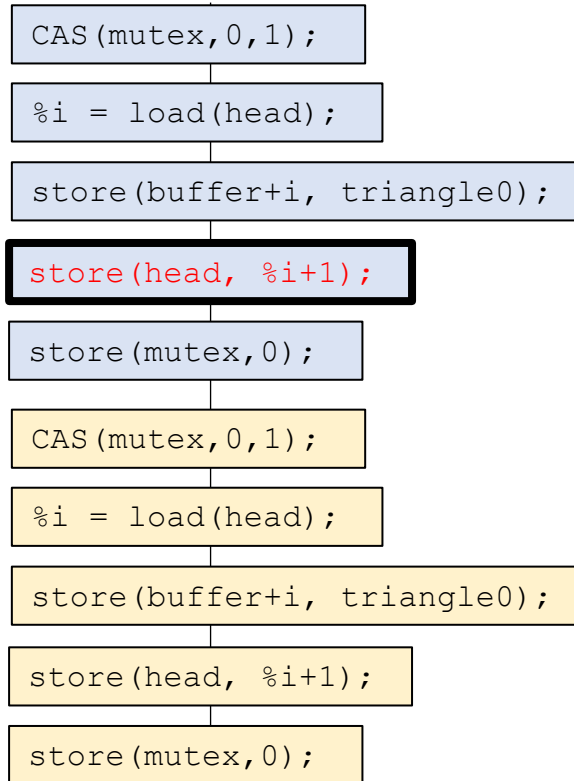
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
fence;
store(mutex, 0);

```

unlock contains fence before store!

what can happen in a PSO memory model?

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |



How to fix the issue?

your unlock function should contain a fence!

Thread 0:

```

SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
fence;
store(mutex, 0);

```

unlock contains fence before store!

Thread 1:

```

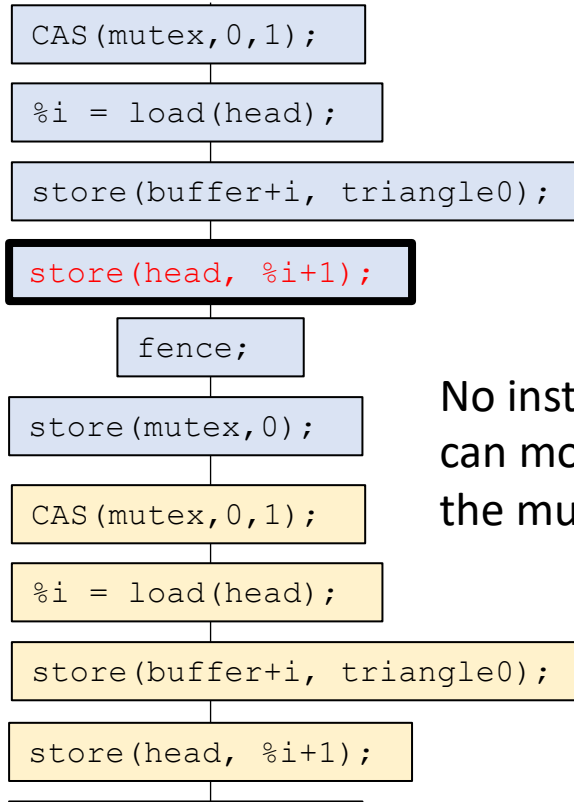
SPIN:CAS (mutex, 0, 1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
fence;
store(mutex, 0);

```

unlock contains fence before store!

what can happen in a PSO memory model?

| | L | S |
|---|----|-------------------|
| L | NO | Different address |
| S | NO | Different address |



No instructions can move after the mutex store!

How to fix the issue?

your unlock function should contain a fence!

Thanks!

- Next, we will talk about decoupled access execute (DAE)