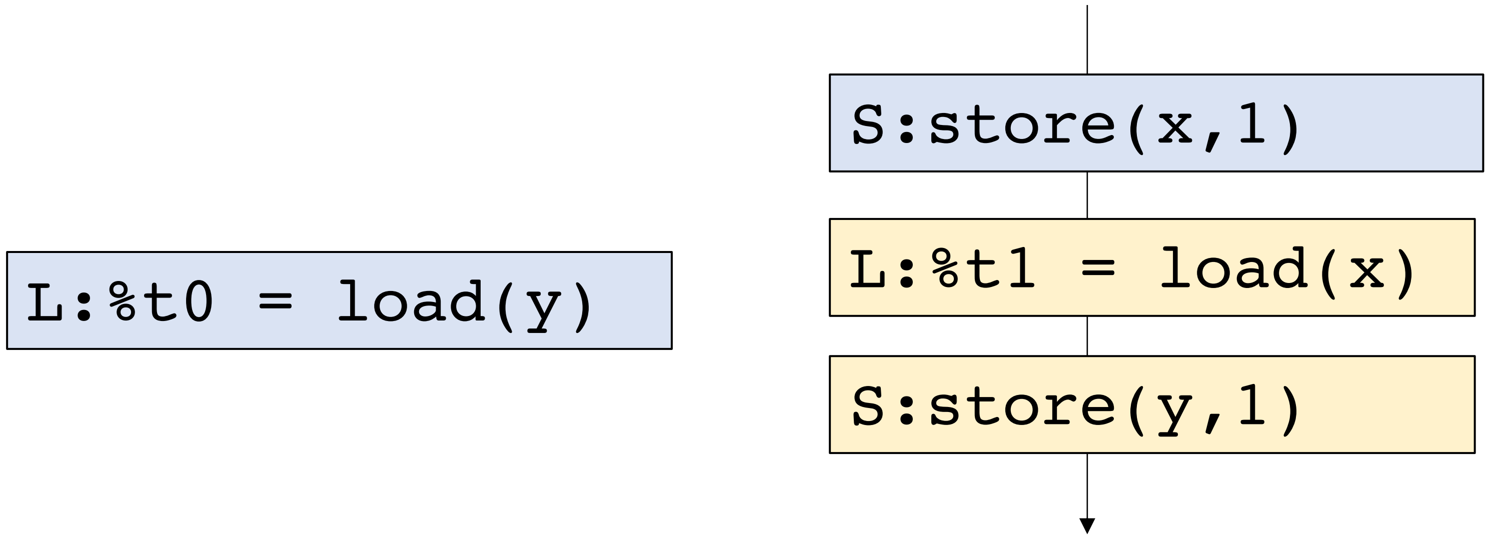
CSE211: Compiler Design

Nov. 17, 2022

- **Topic:** Compiling relaxed memory models



Announcements

- Homework 3 is due on Monday
  - Office hours tomorrow 3-5 pm
  - Feel free to share results (not code!) on piazza or discord

- Guest lecture for Nov. 29
  - Felix Klock will be talking about Rust

Announcements

- Working on grading HW 2 still
    - Will have more of an update on Tuesday

Paper and project proposals

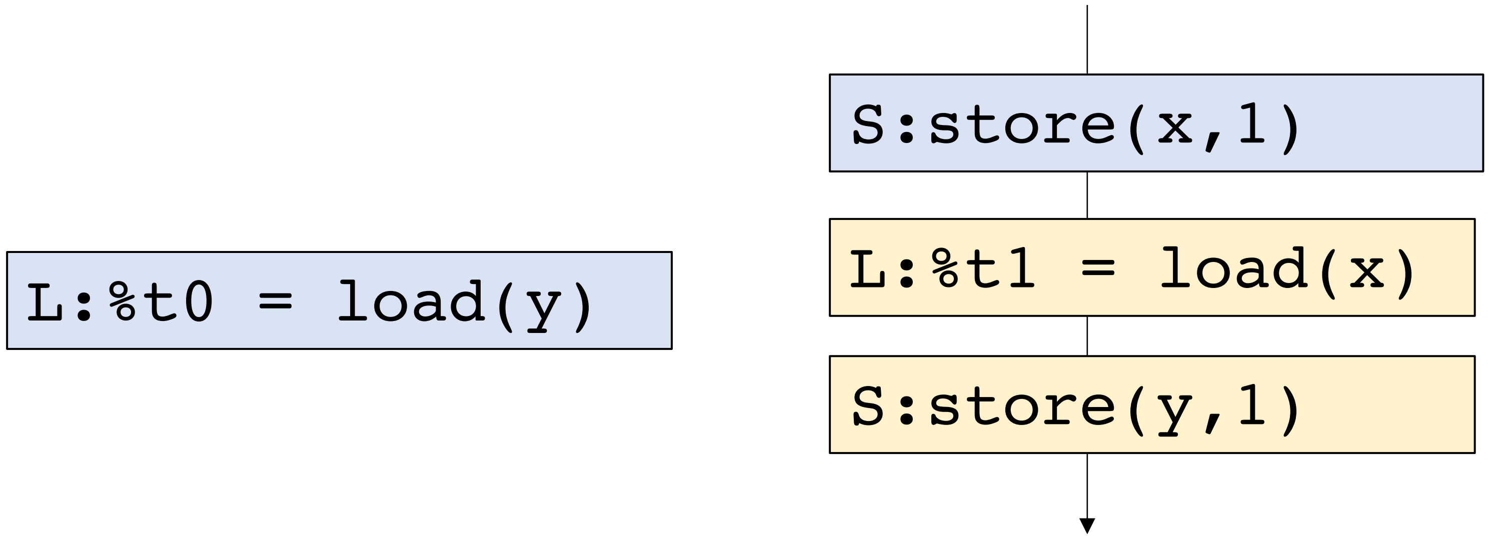
- Paper and project proposals
  - I have left everyone a comment on both. Everything looks great thanks for submitting them!

- Remember
  - Reports are due the day of the final: Dec 8.
  - I highly suggest not saving these until the last minute. They have a late deadline to give you flexibility, not to enable procrastination.
  - one more homework (will be assigned on Monday)

CSE211: Compiler Design

Nov. 17, 2022

- **Topic:** Compiling relaxed memory models



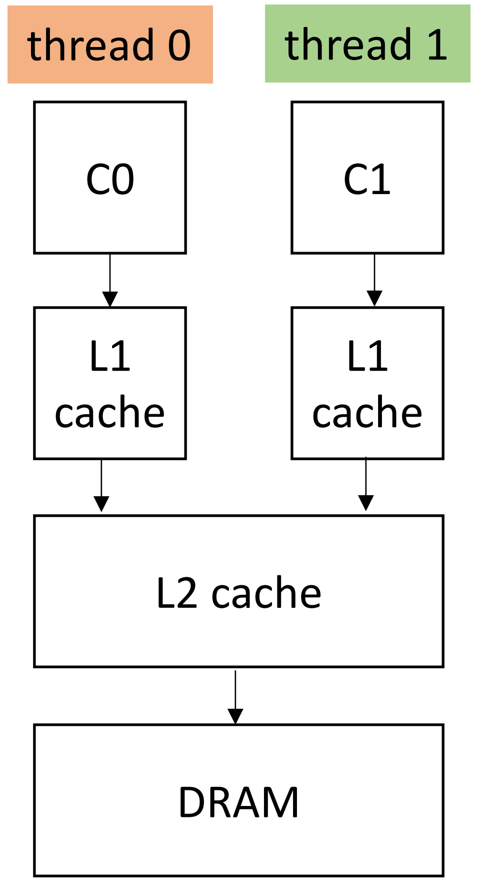
```
L:%t0 = load(y)
```

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

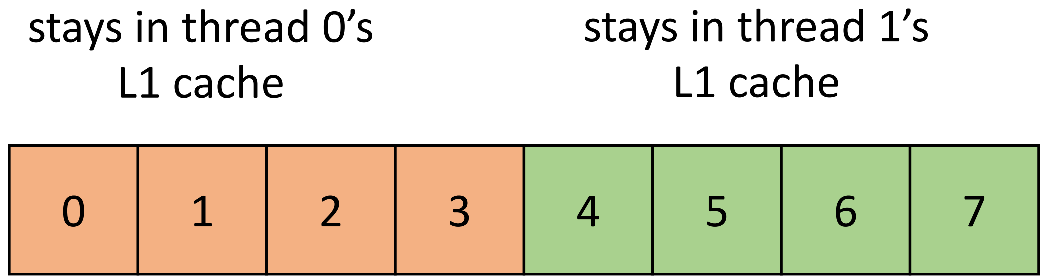
Review

- How to implement parallelism in DOALL loops
  - Regular parallelism?

DOALL regular parallelism on SMP system



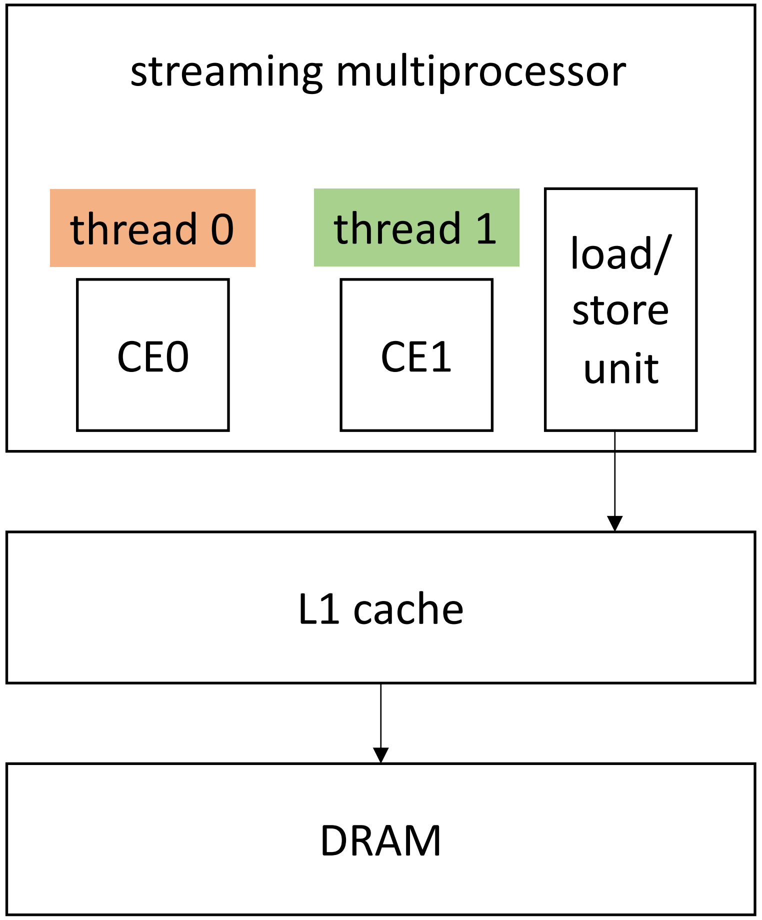
SMP parallelism



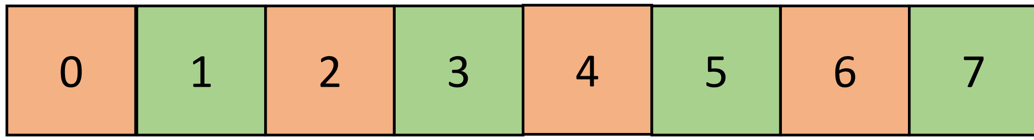
DOALL regular parallelism on GPUs

one streaming multiprocessor contains many small Compute Elements (CE)

CEs Can load adjacent memory locations simultaneously

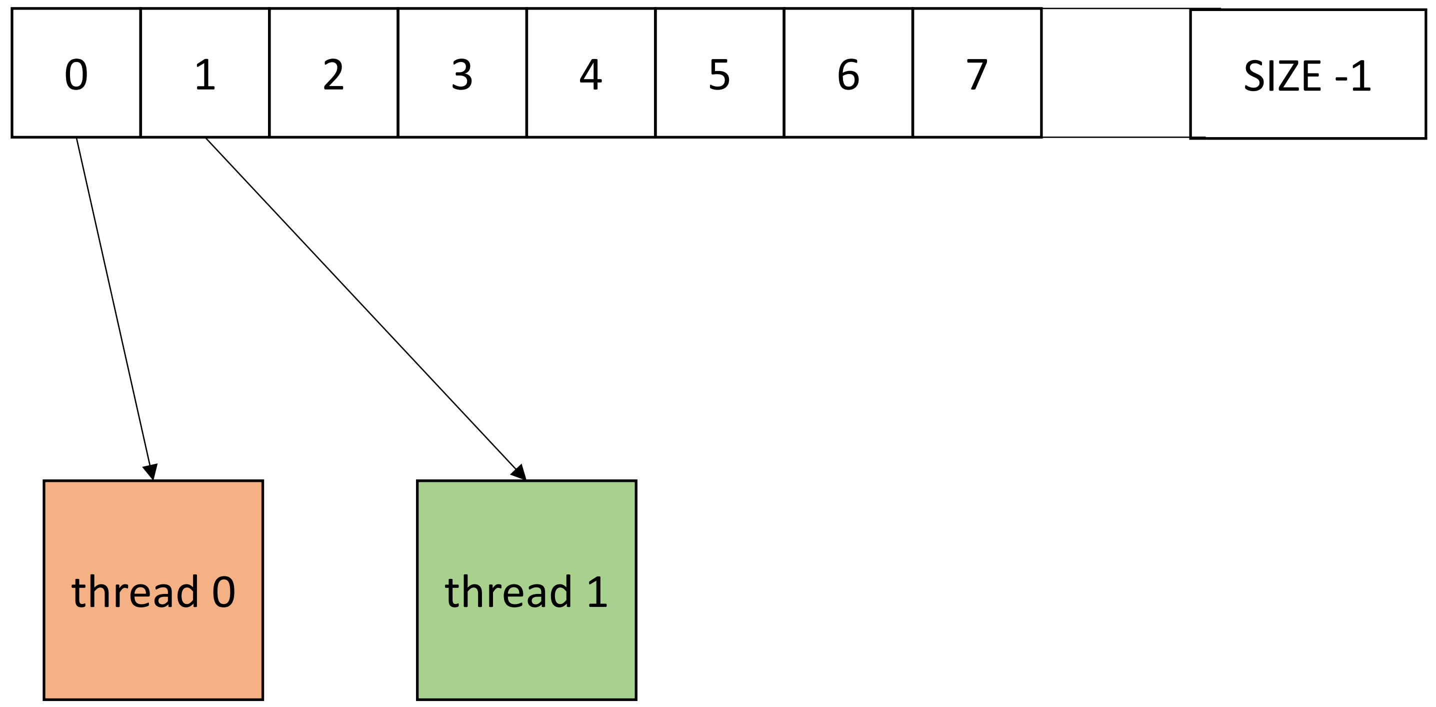


What about a striped pattern?



Work stealing - global implicit worklist

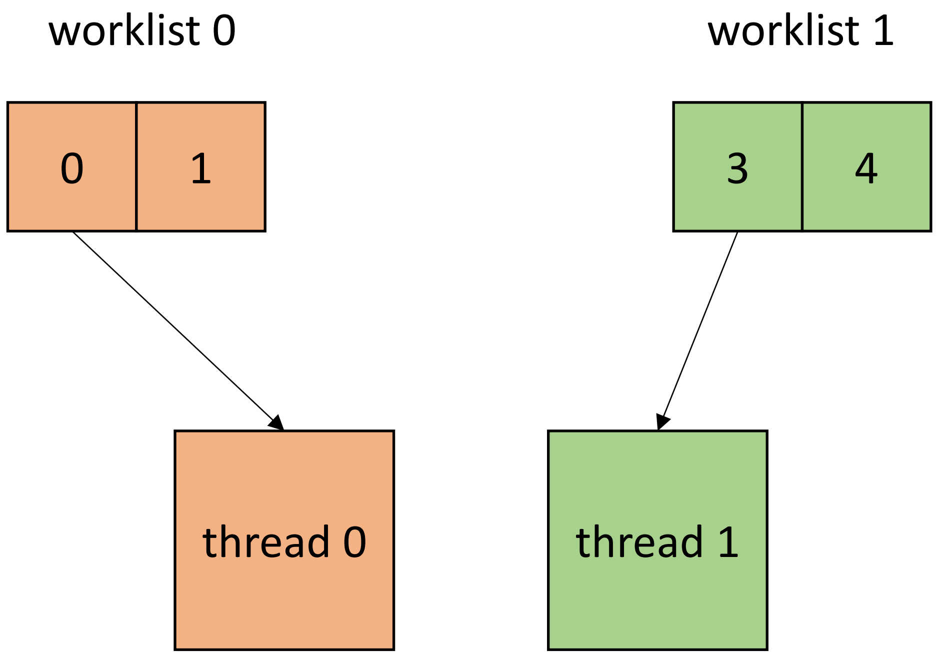
- Global worklist: threads take tasks (iterations) dynamically



Pros? Cons?

Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



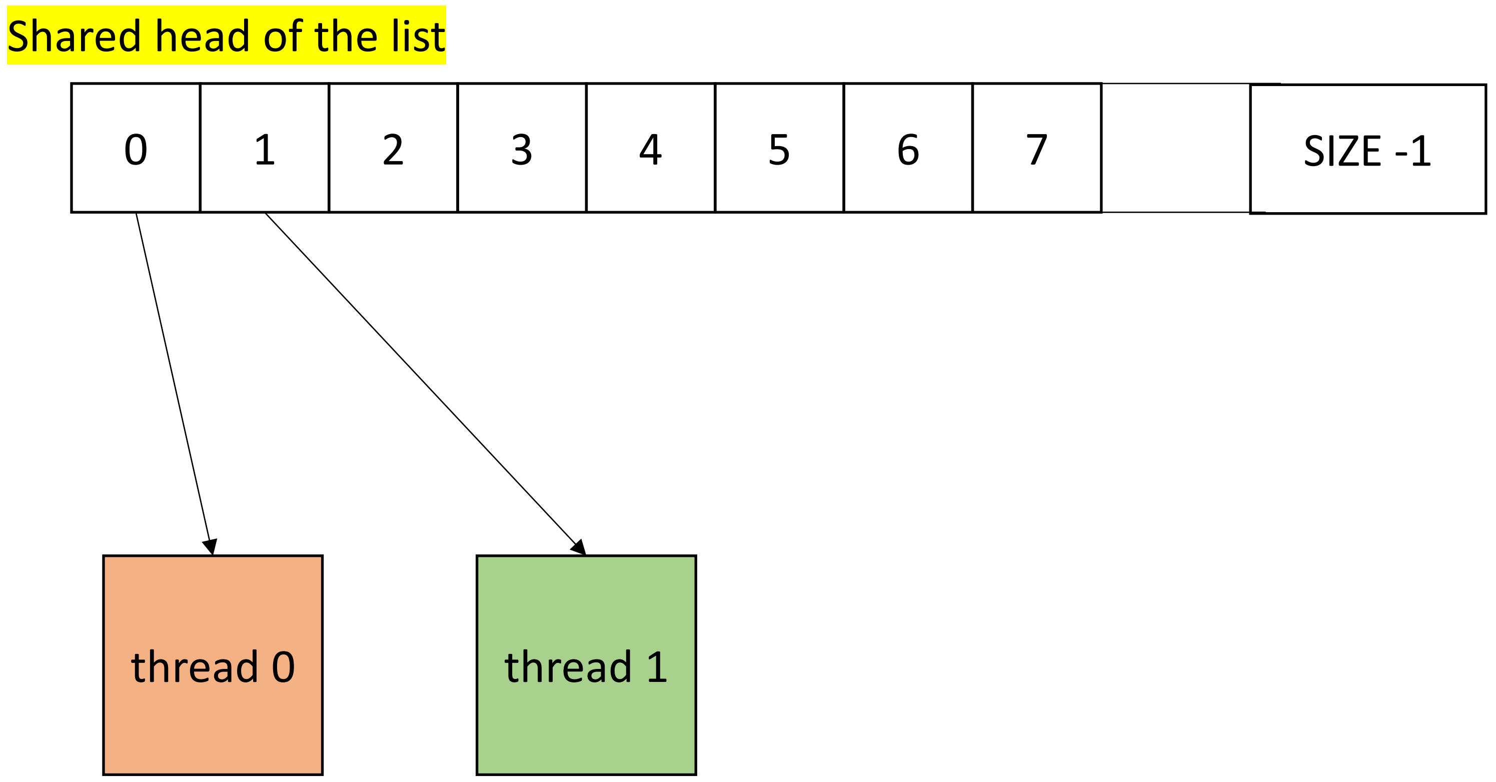
Pros?
Cons?

Today’s lecture

- We have been assuming DOALL loops:
  - Threads access completely disjoint memory
  - This might not be the case
  - Examples?

Work stealing - global implicit worklist

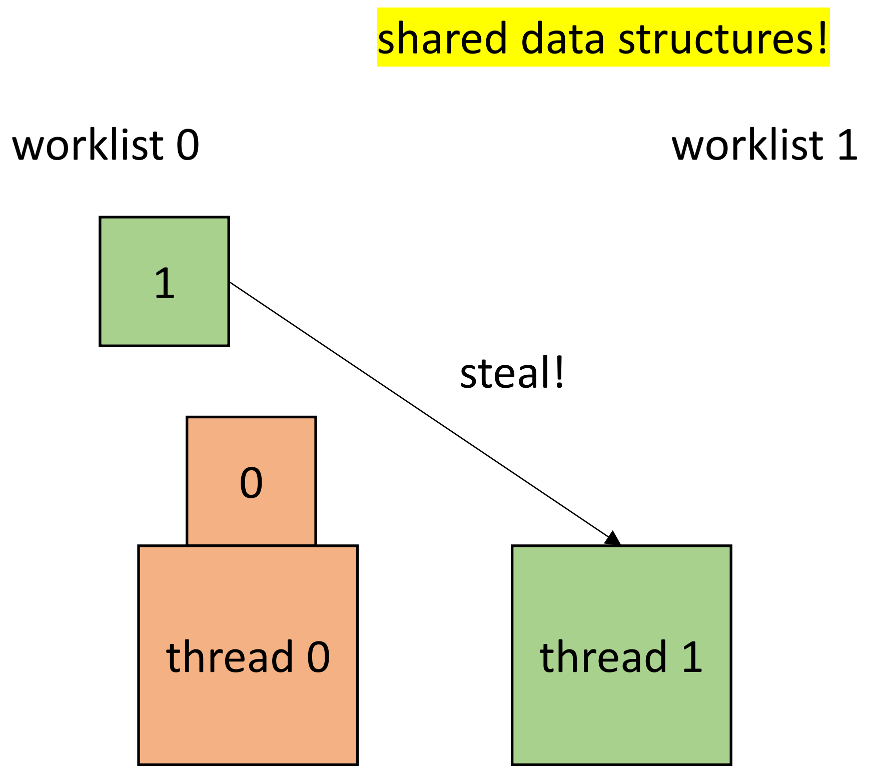
- Global worklist: threads take tasks (iterations) dynamically



Pros? Cons?

Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



What happens when threads share data?

Global variable:

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

Thread 0:

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

Thread 1:

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

Global variable:

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

Thread 0:

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

Thread 1:

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

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

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

```
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);
```

Thread 1:

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

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

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

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

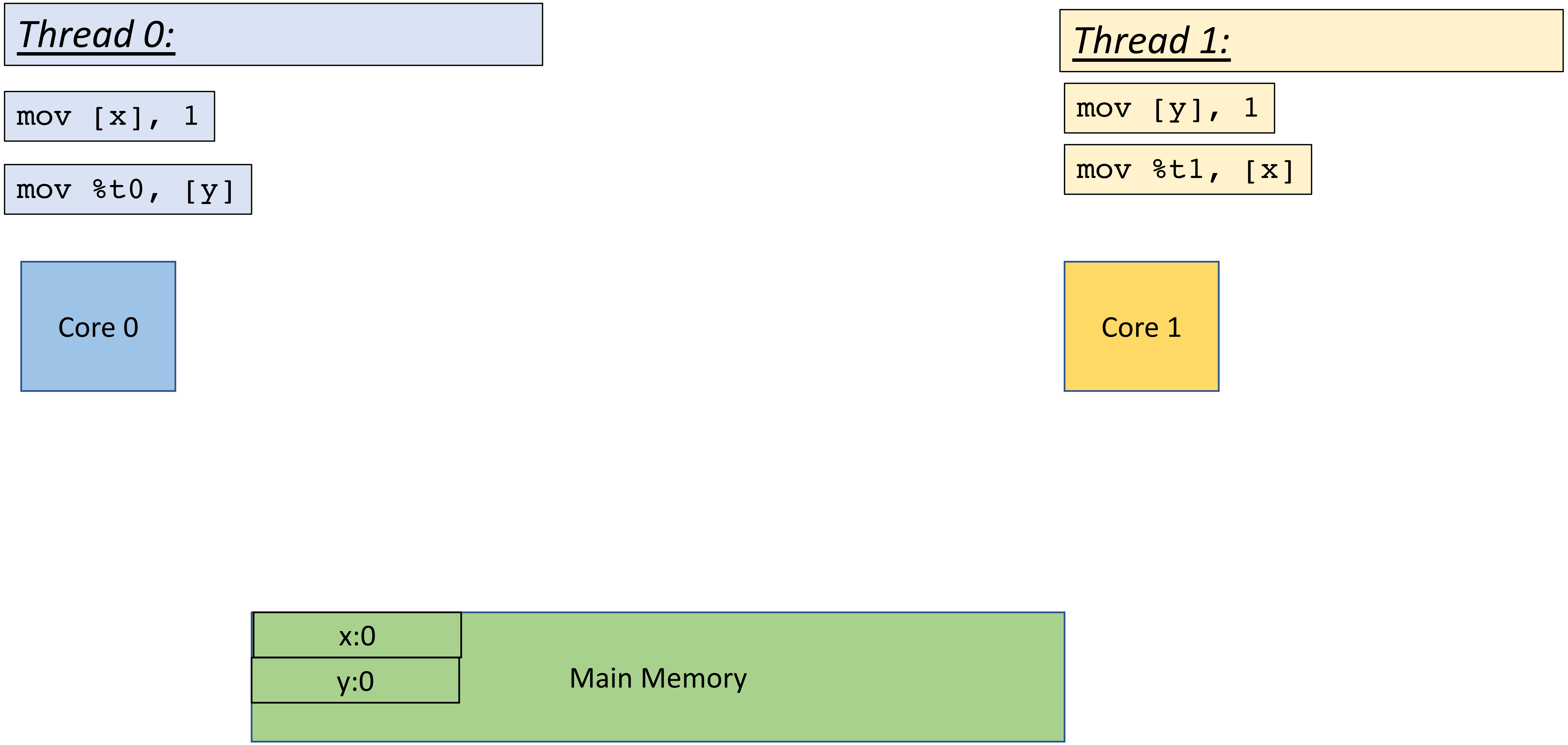
```
L:%t1 = load(x);
```

pick from the top of the pile of either thread

Can t0 == t1 == 0 at the end of the execution?

Demo

- What is going on?



Thread 0:

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

Thread 1:

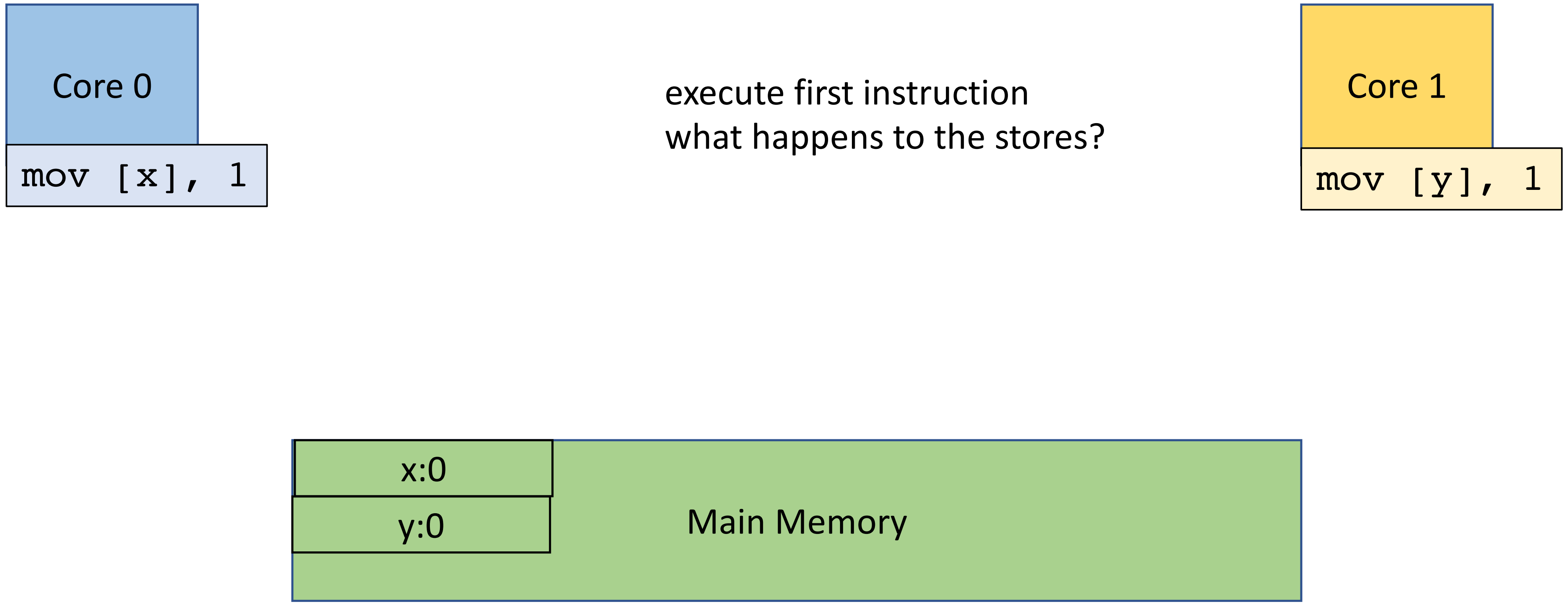
```
mov [y], 1
mov %t1, [x]
```

Thread 0:

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

Thread 1:

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



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

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

execute first instruction
what happens to the stores?

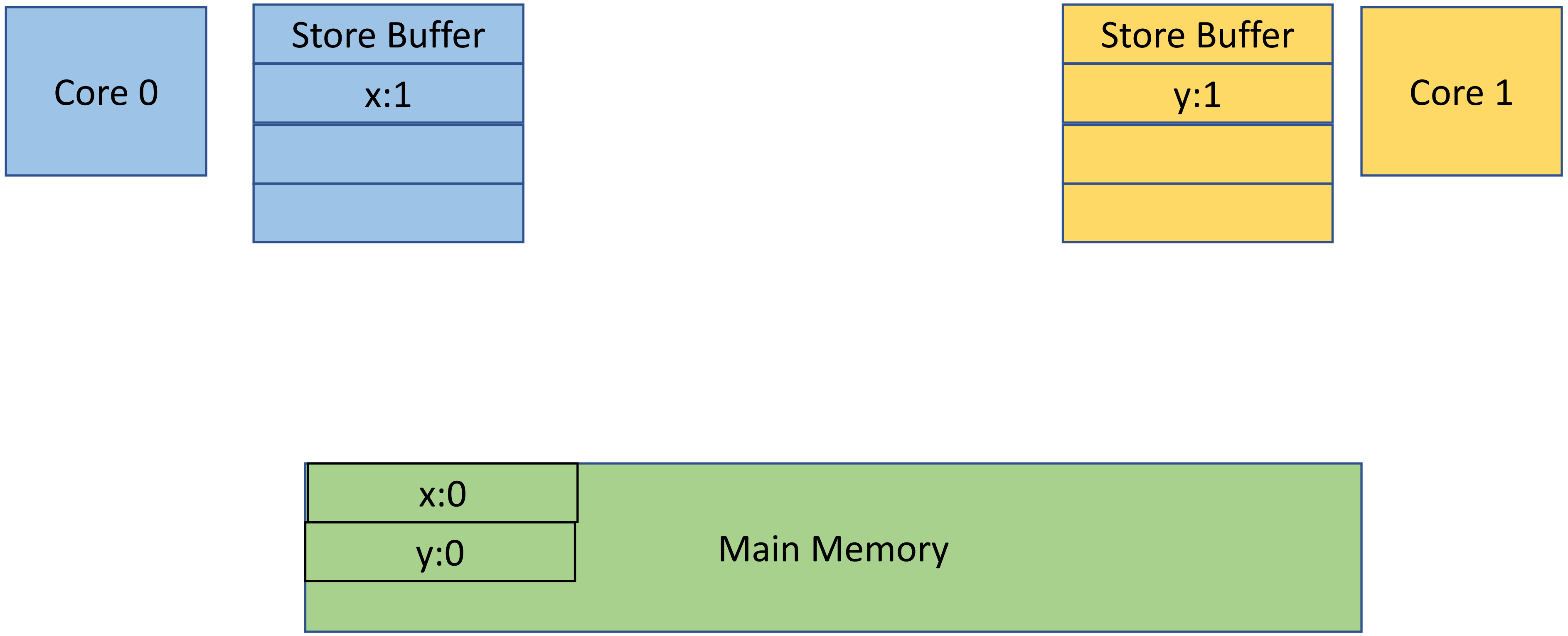
Thread 0:

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

Thread 1:

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

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



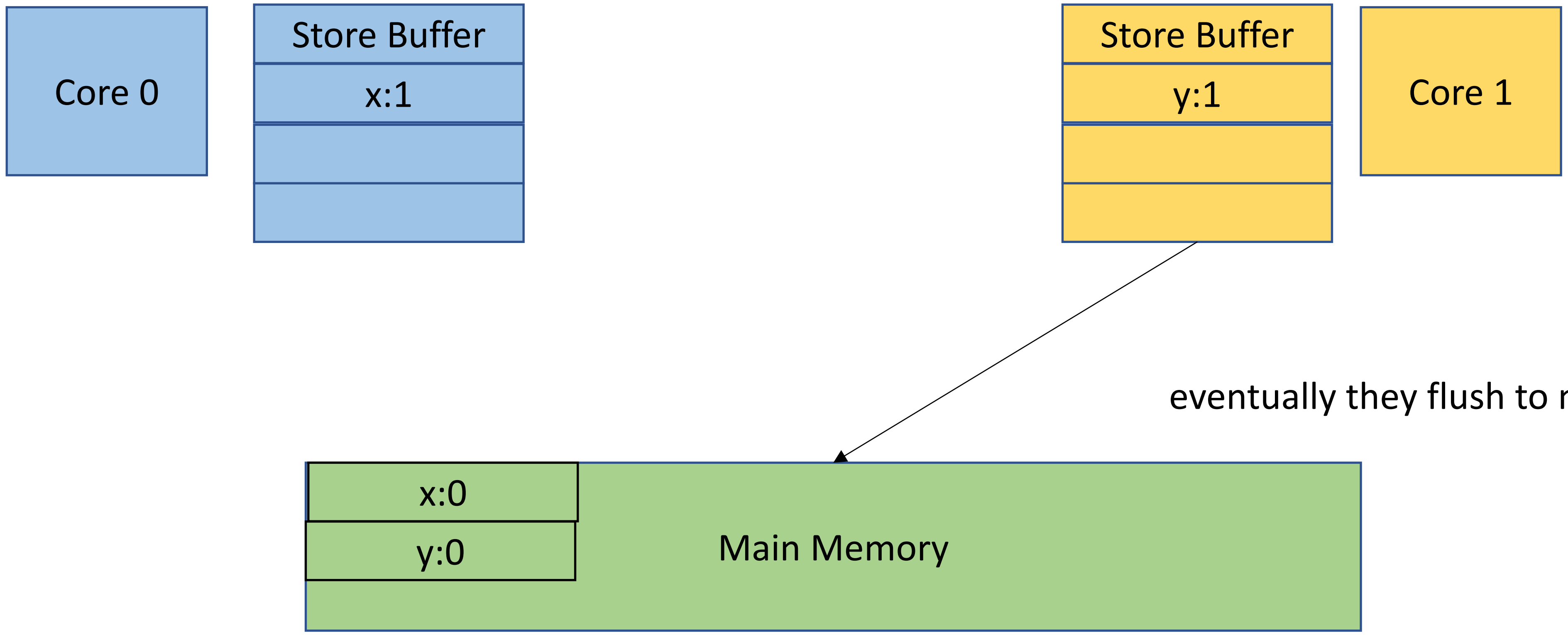
Thread 0:

```
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

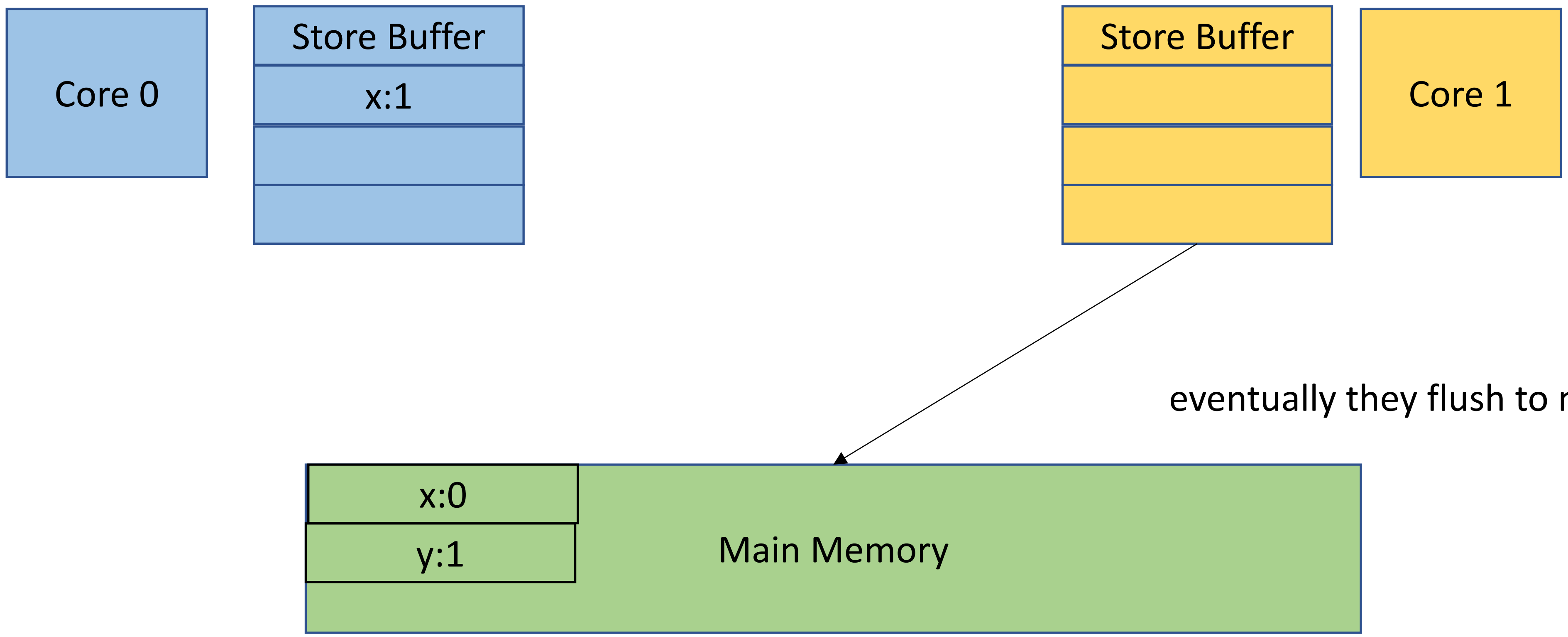
Thread 0:

Thread 1:

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

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

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



eventually they flush to main memory

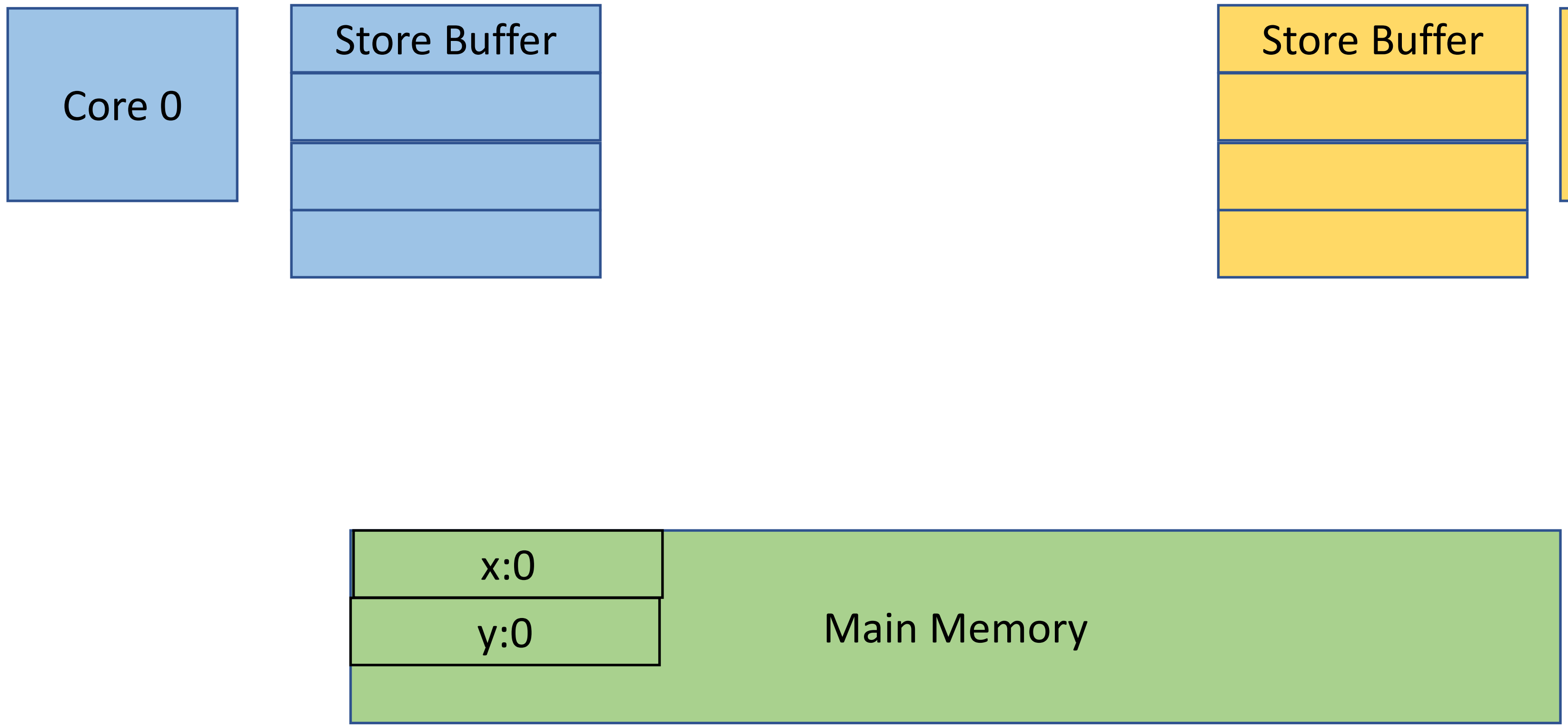
Thread 0:

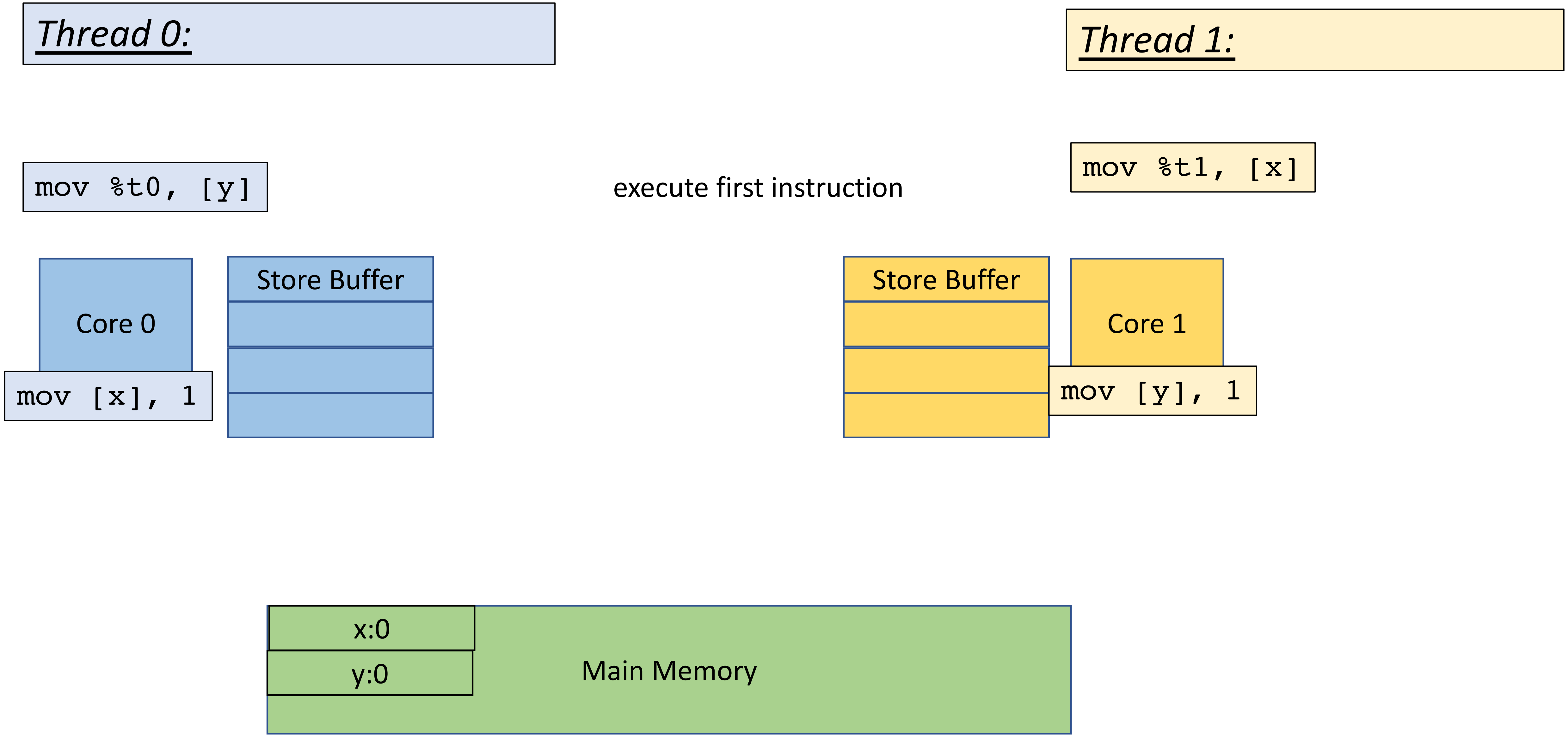
```
mov [x], 1
mov %t0, [y]
```

rewind

Thread 1:

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





Thread 0:
Thread 1:
mov %t0, [y]
execute first instruction
mov %t1, [x]
Core 0
mov [x], 1
Store Buffer
Store Buffer
Core 1
mov [y], 1
x:0
y:0
Main Memory

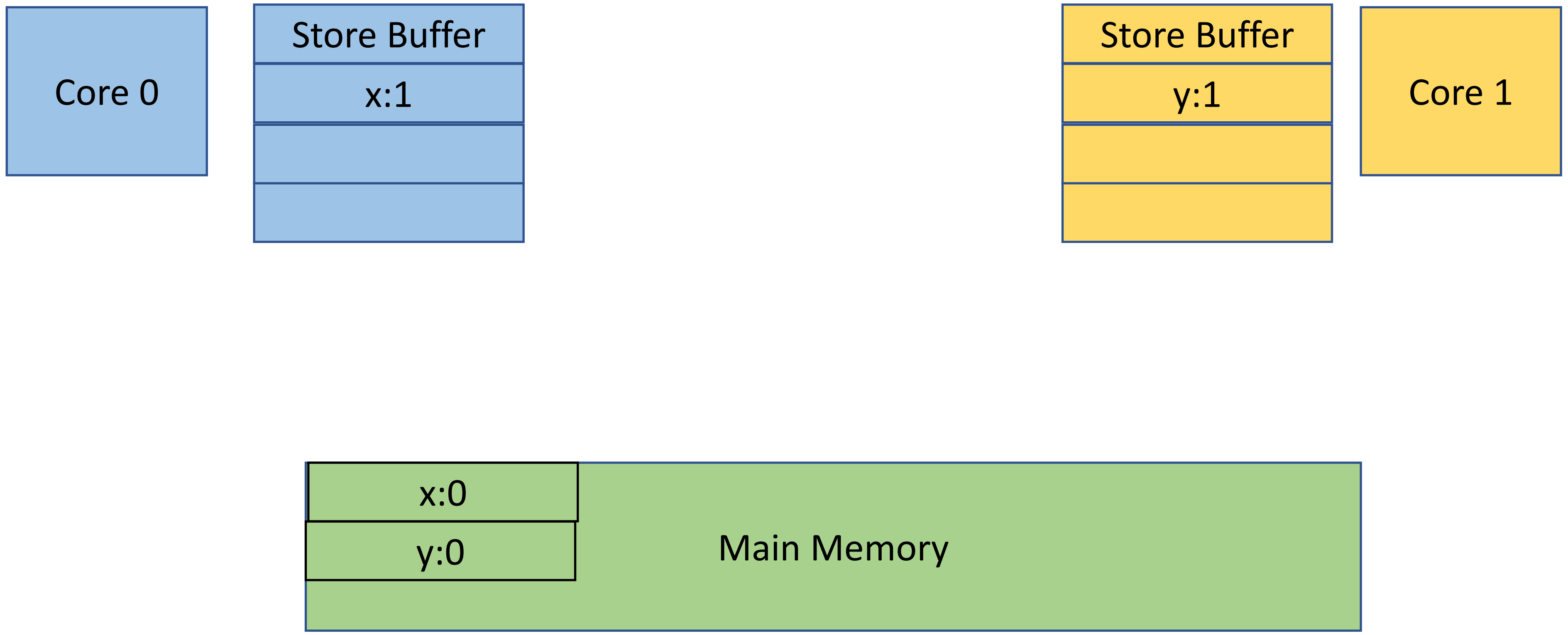
Thread 0:

Thread 1:

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

values get stored in SB

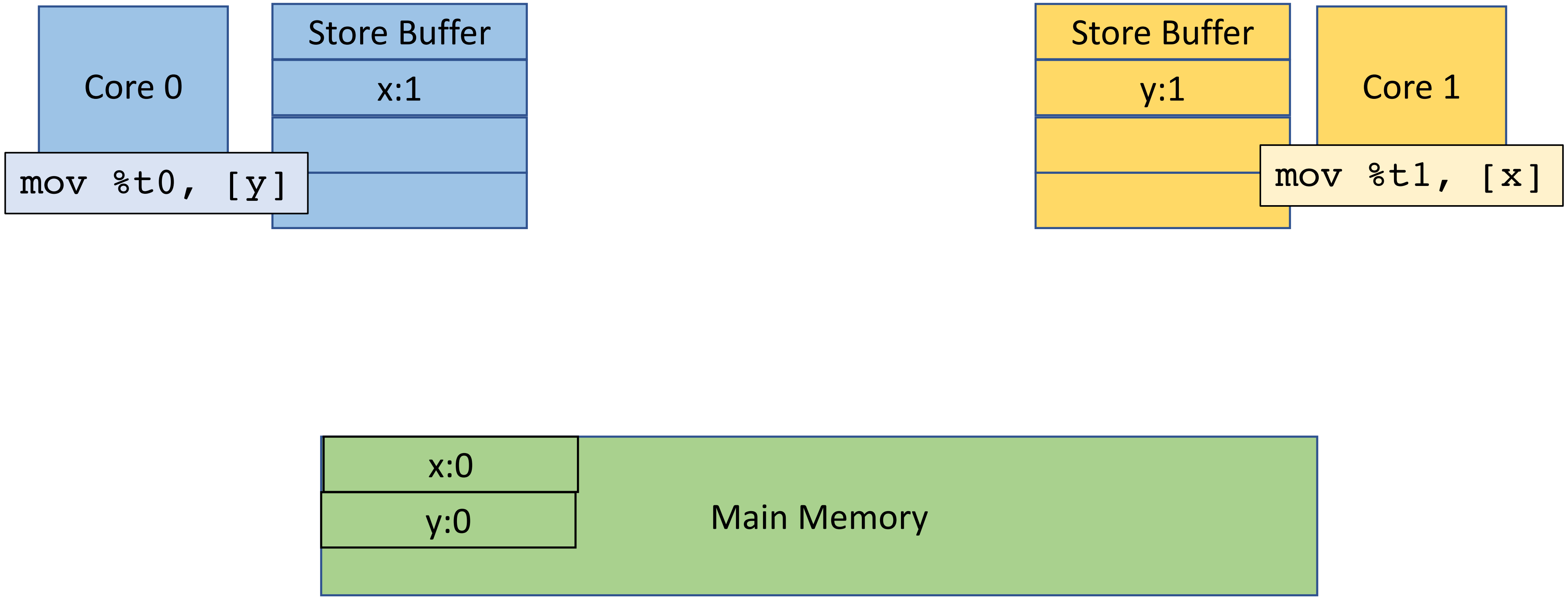
```
mov %t1, [x]
```



Thread 0:

Thread 1:

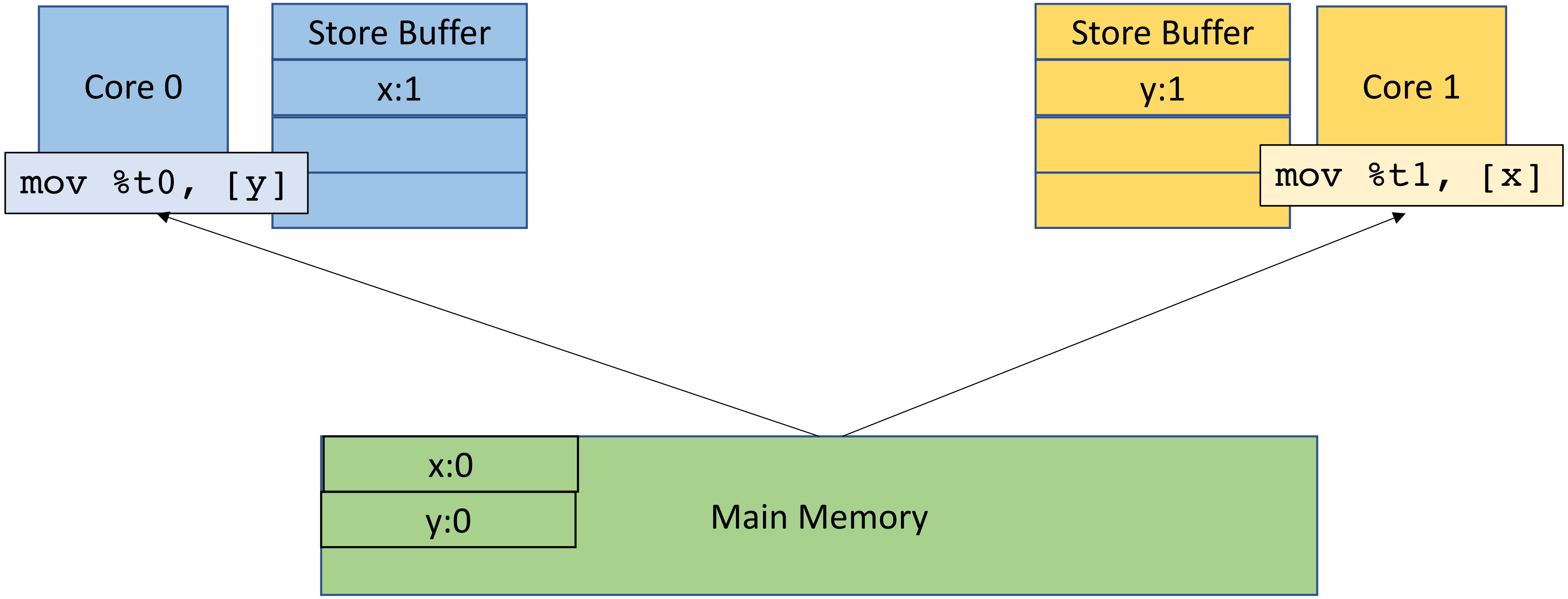
Execute next instruction



Thread 0:

Thread 1:

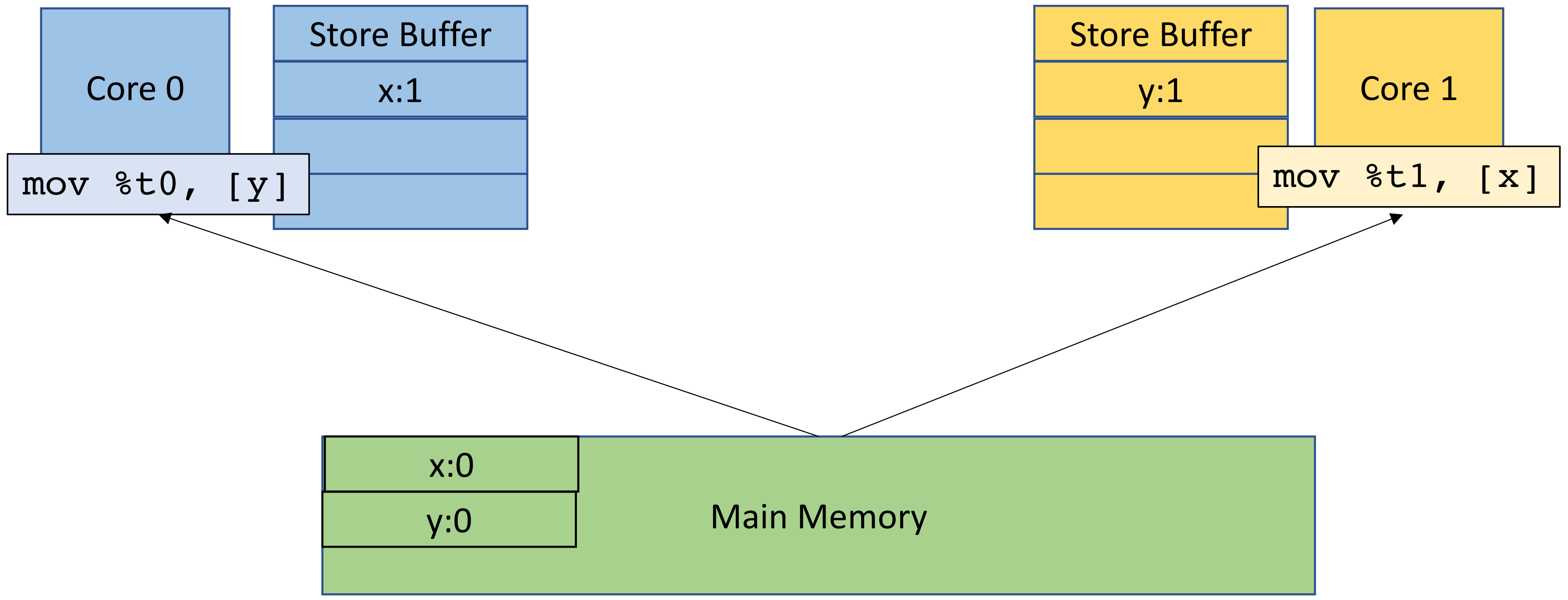
Values get loaded from memory



Thread 0:

Thread 1:

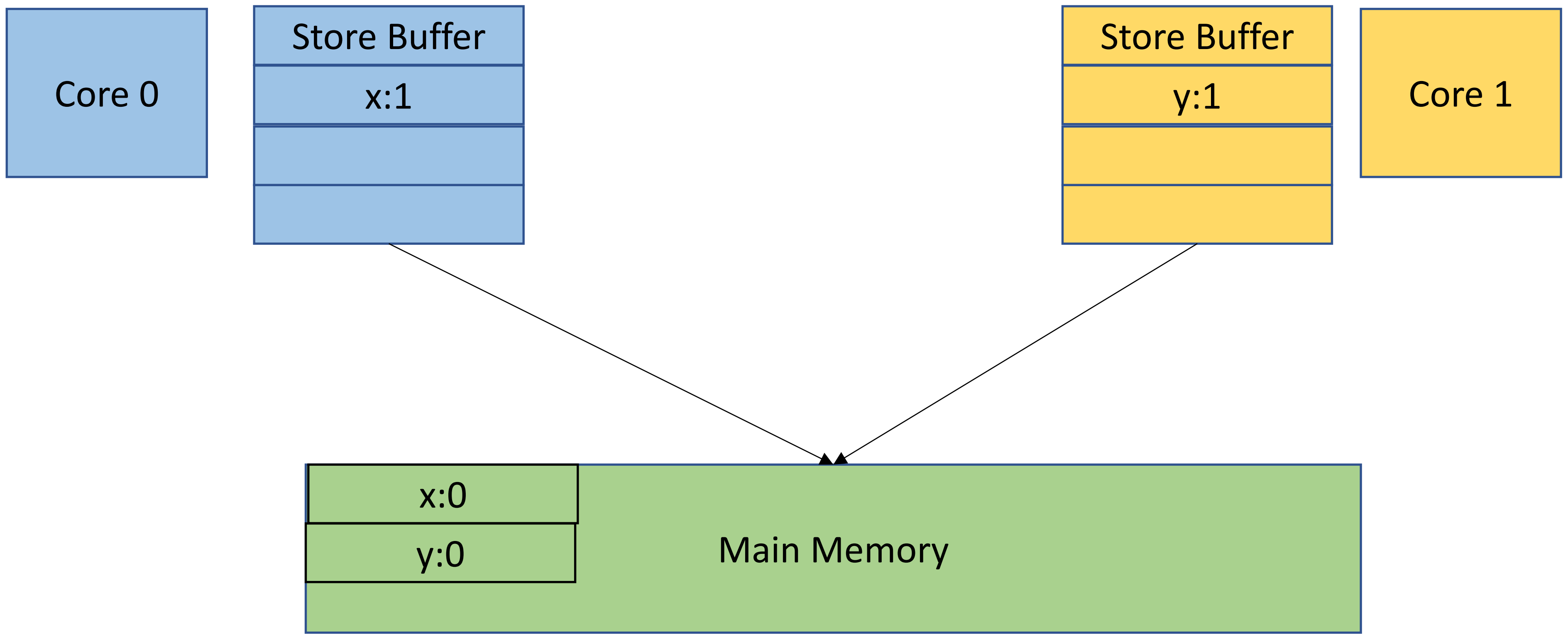
we see `t0 == t1 == 0`!



Thread 0:

Thread 1:

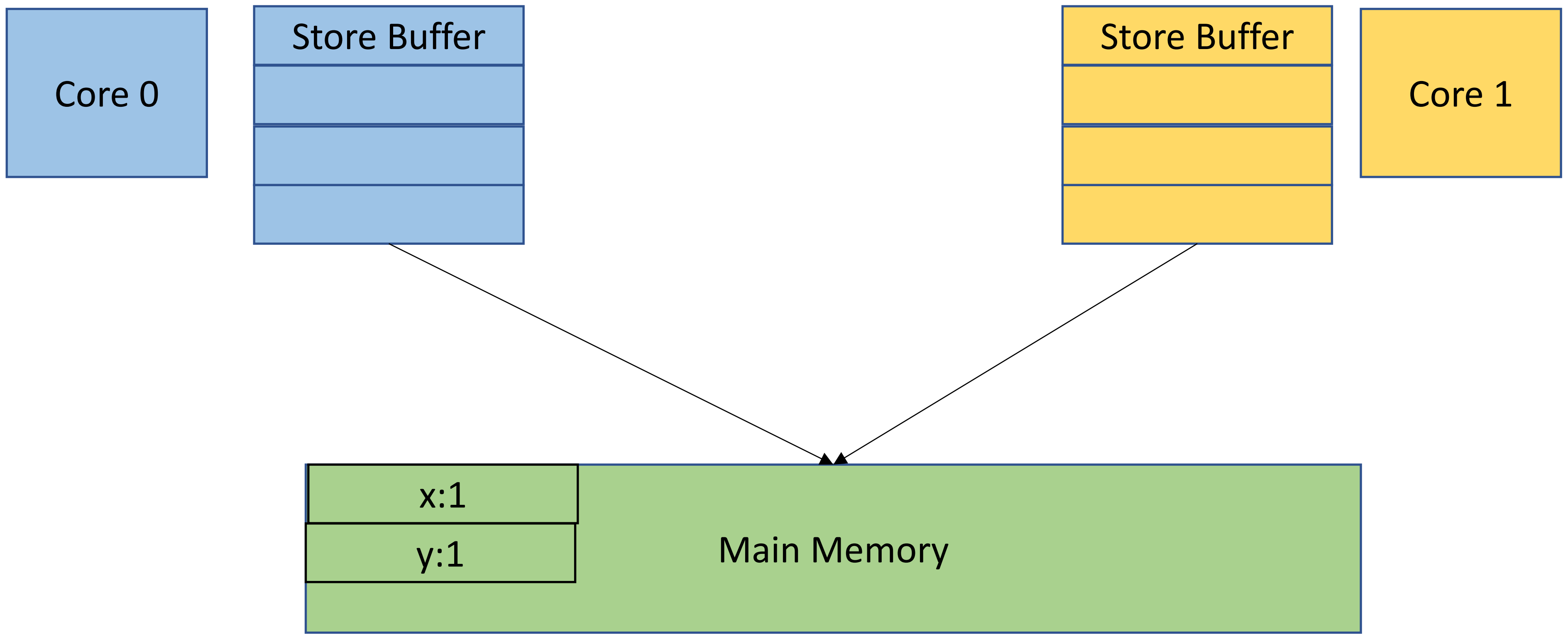
Store buffers are drained eventually



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

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]
```

Can `t0 == t1 == 0`?

Restoring sequential consistency

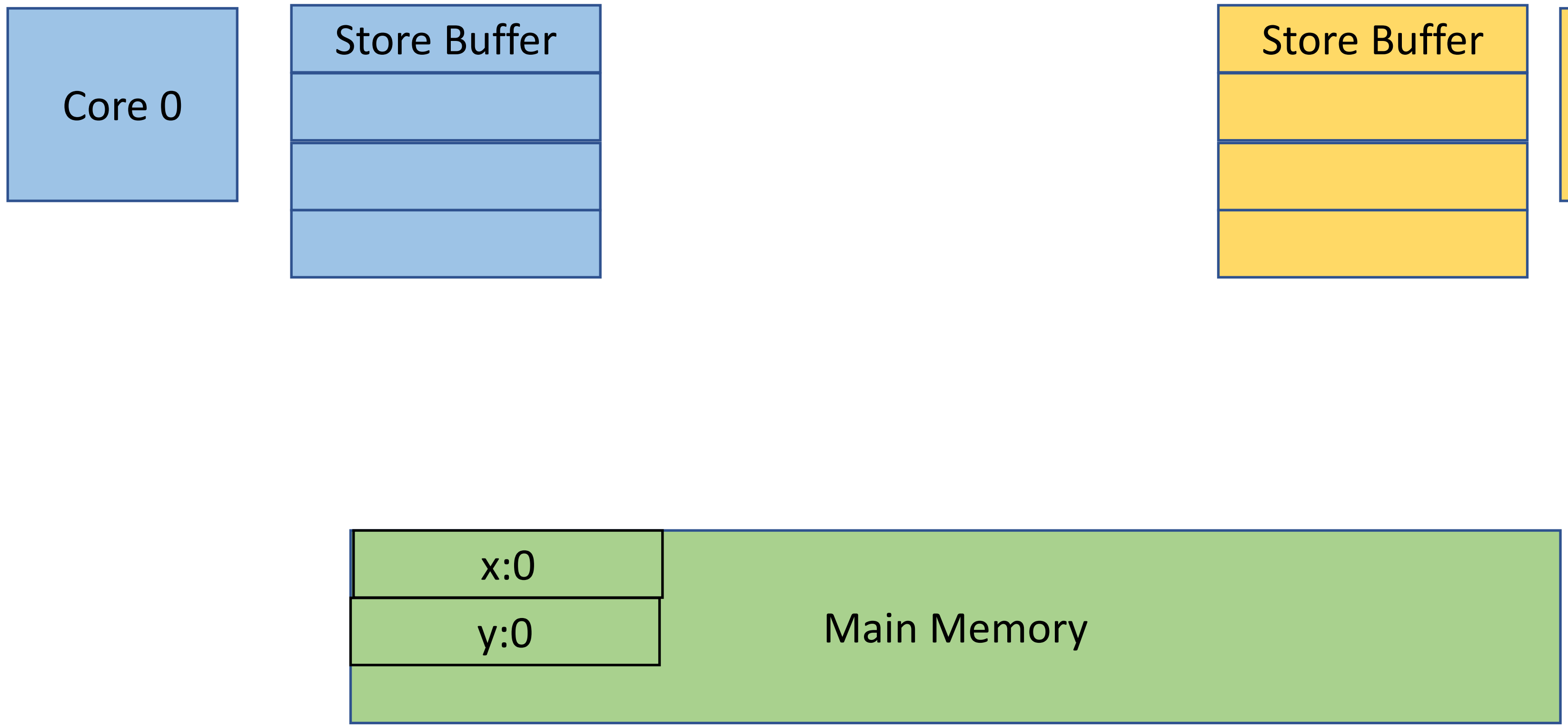
- It is typical that relaxed memory models provide special instructions which can be used to disallow weak behaviors.
- These instructions are called Fences
- The X86 fence is called `mfence`. It flushes the store buffer.

Thread 0:

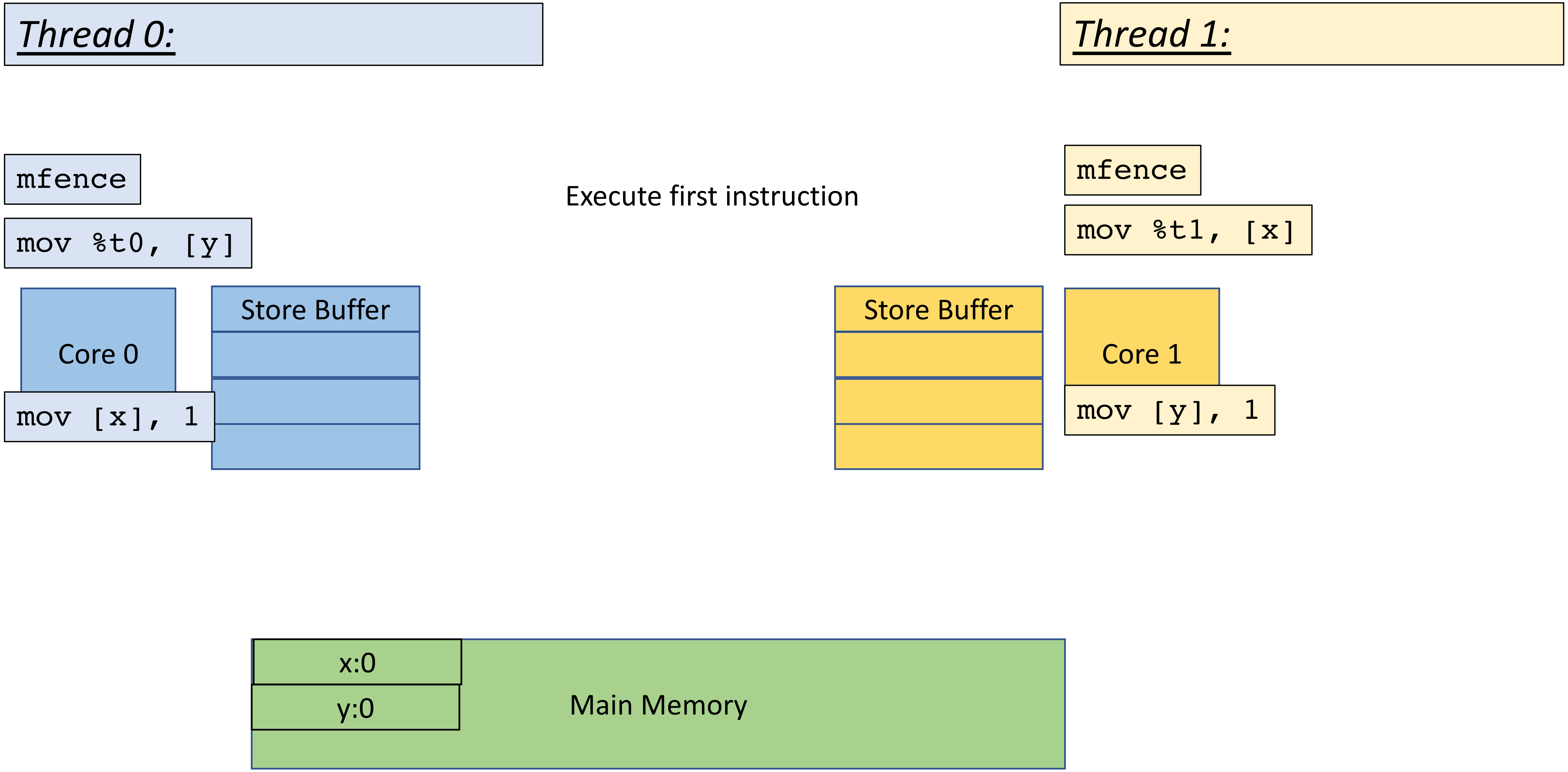
```
mov [x], 1
mfence
mov %t0, [y]
```

Thread 1:

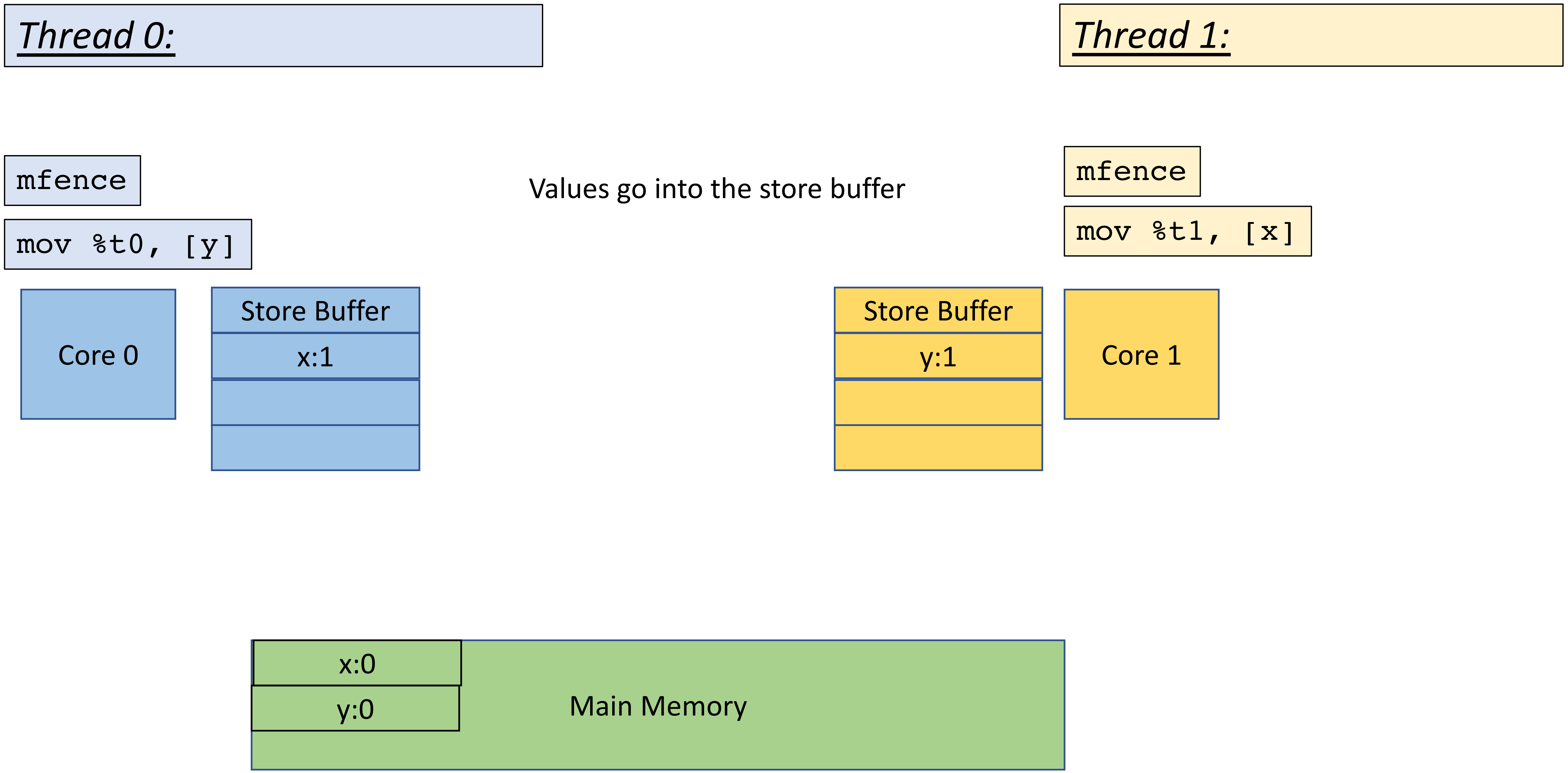
```
mov [y], 1
mfence
mov %t1, [x]
```



Execute first instruction



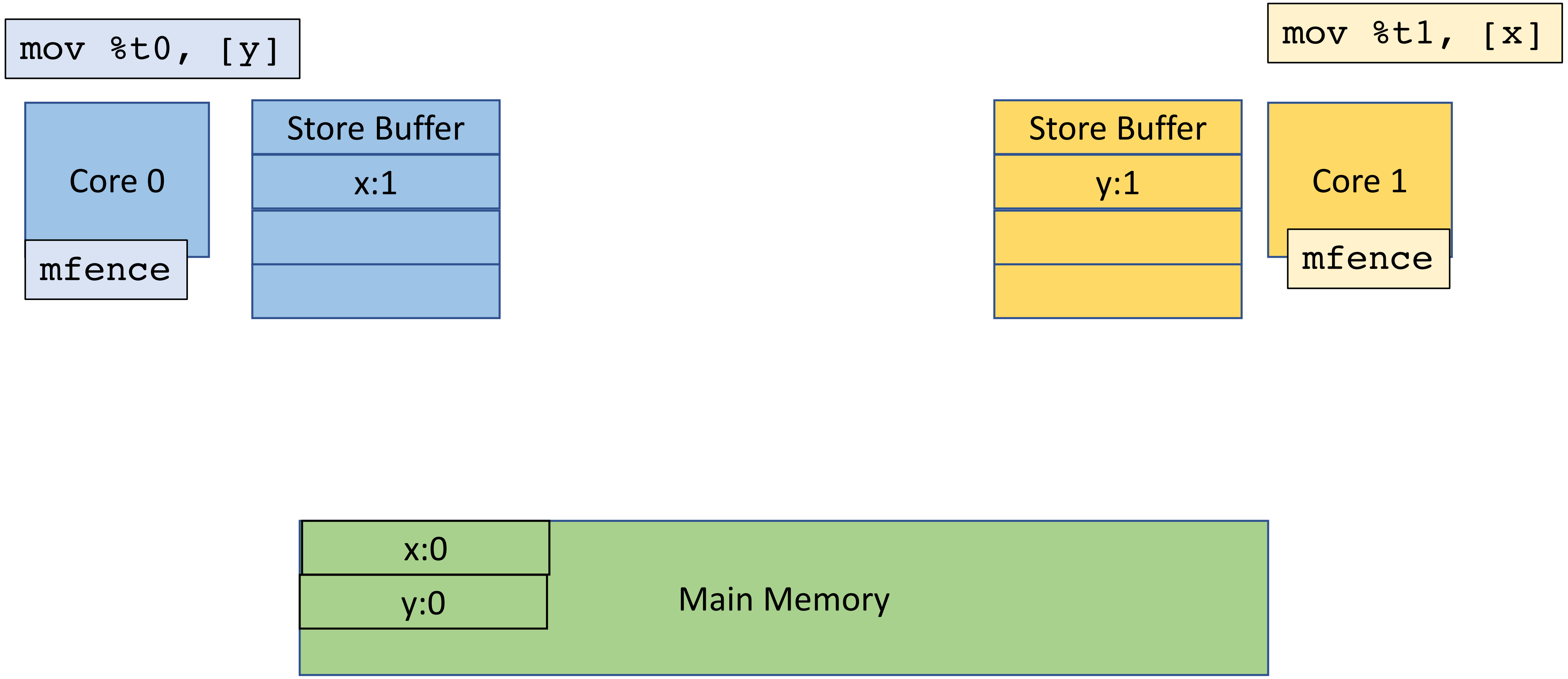
Values go into the store buffer



Thread 0:

Thread 1:

Execute next instruction



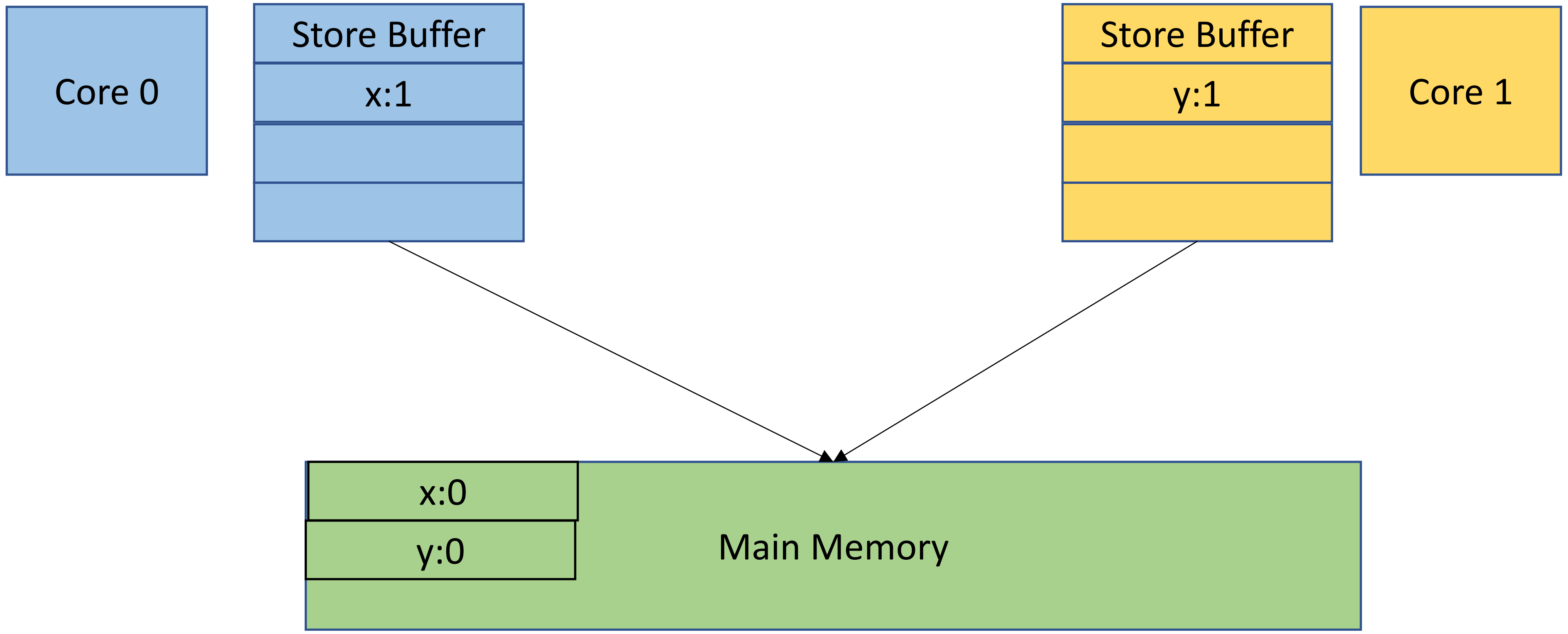
Thread 0:

Thread 1:

store buffers are flushed

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

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



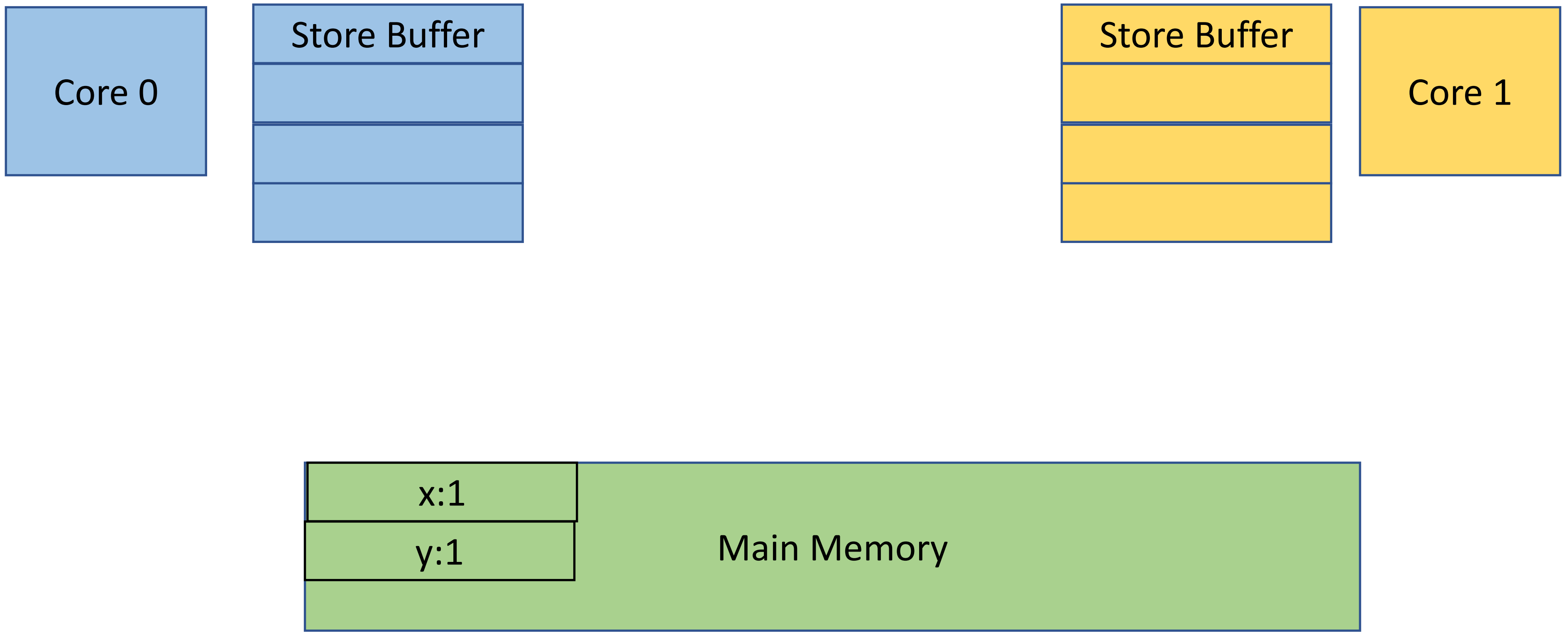
Thread 0:

Thread 1:

store buffers are flushed

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

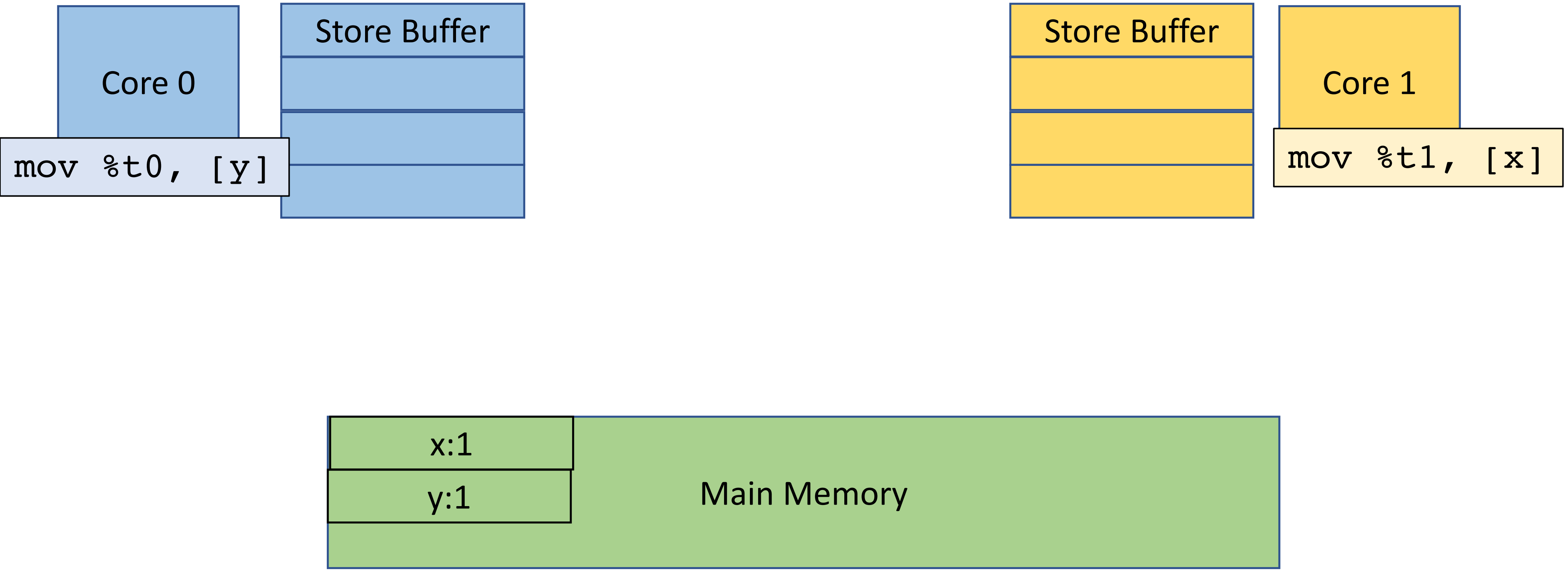
```
mov %t1, [x]
```



Thread 0:

Thread 1:

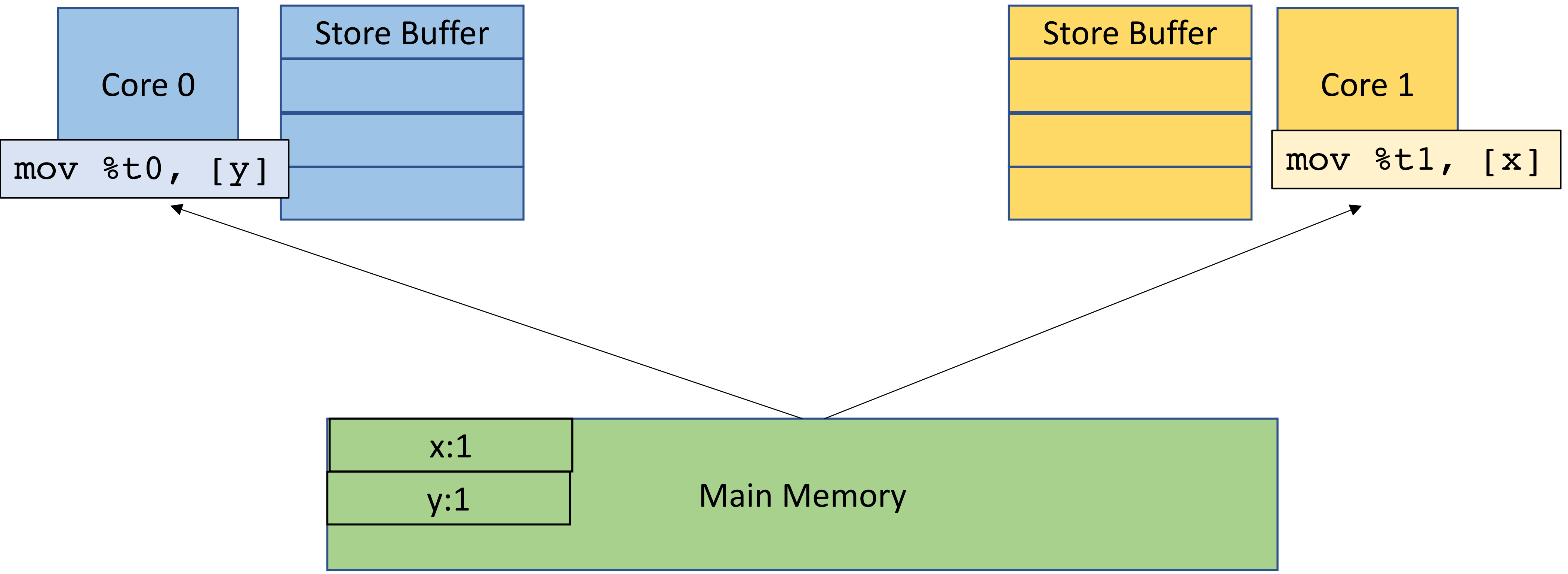
execute next instruction



Thread 0:

Thread 1:

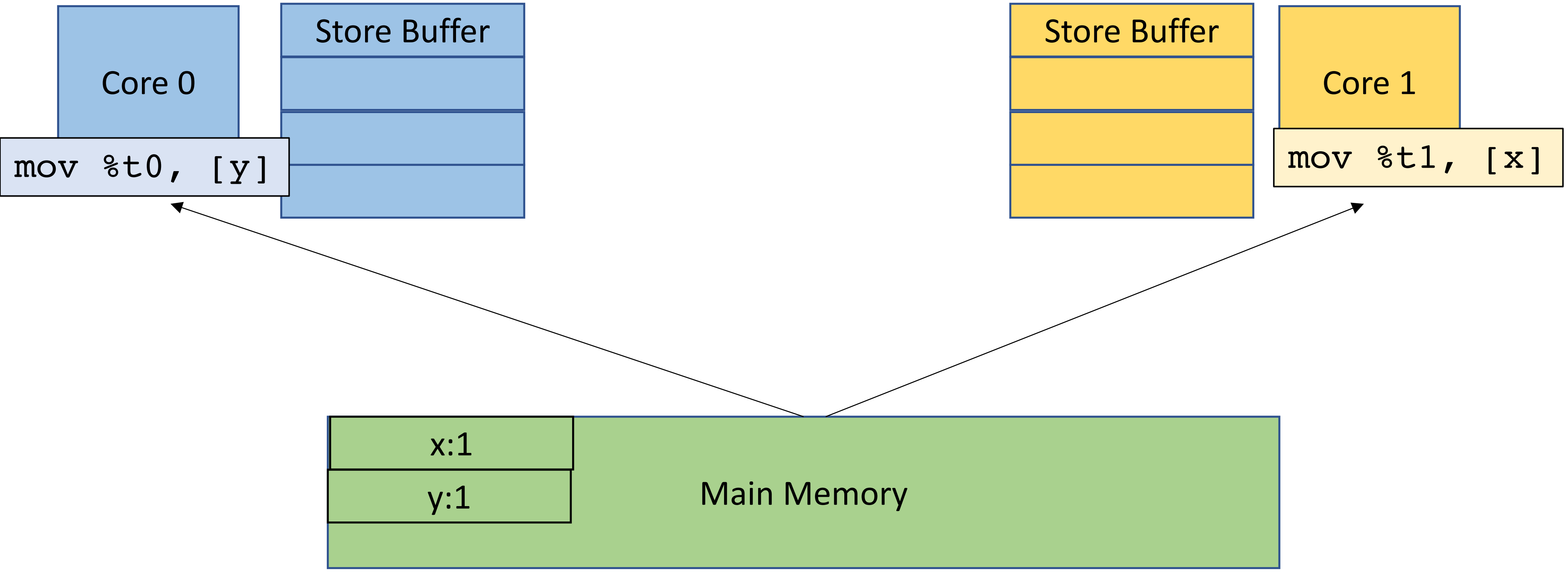
values are loaded from memory



Thread 0:

Thread 1:

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

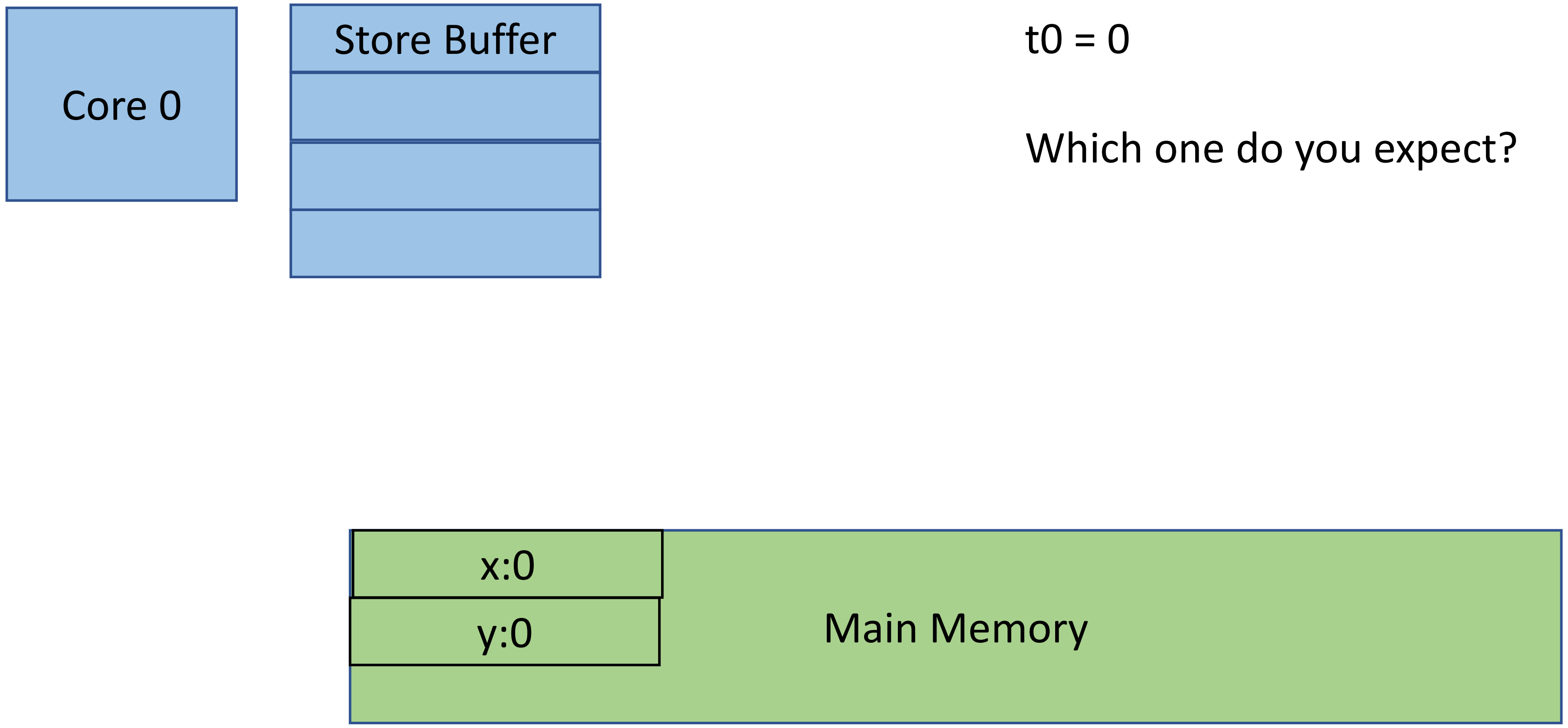


Next example

Thread 0:

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

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



single thread
same address

possible outcomes:
t0 = 1
t0 = 0

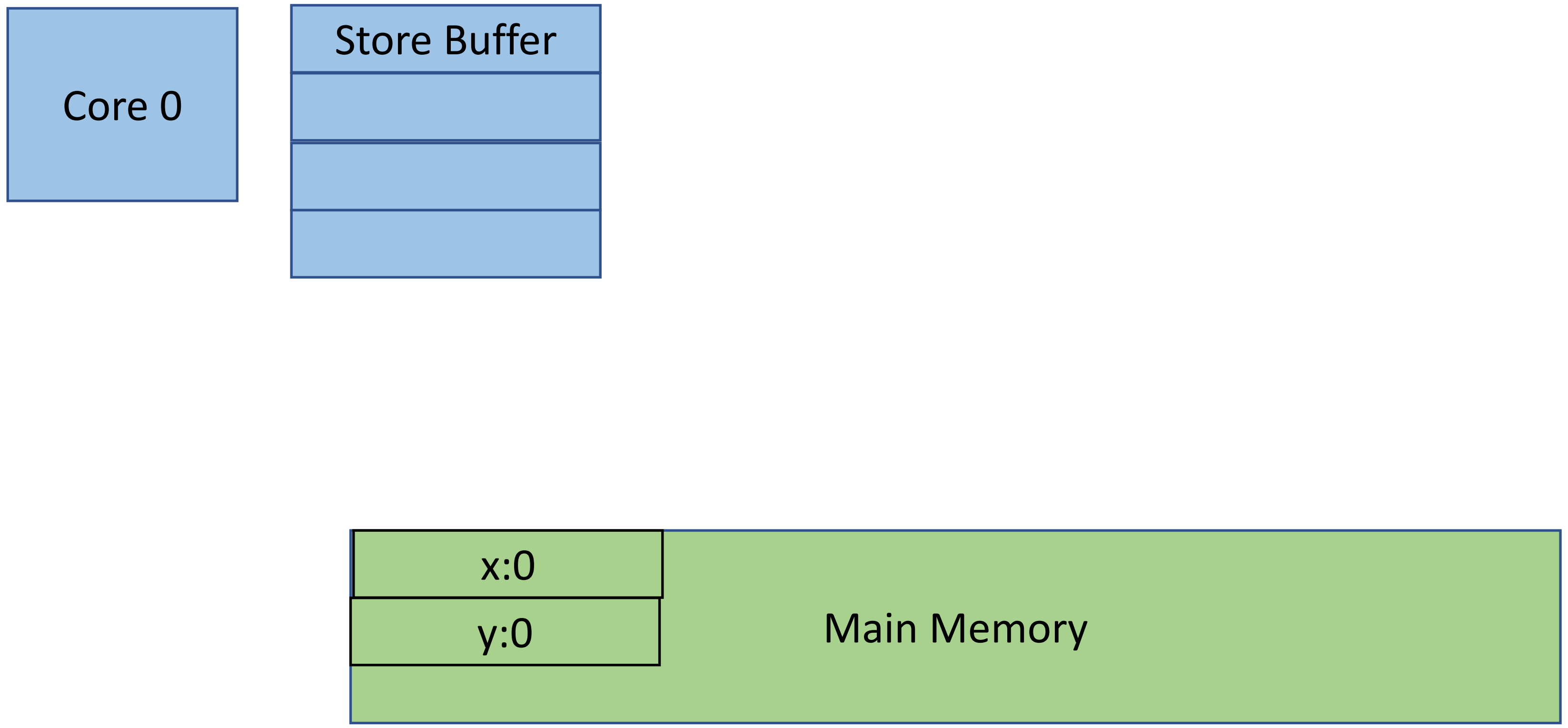
Which one do you expect?

Thread 0:

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

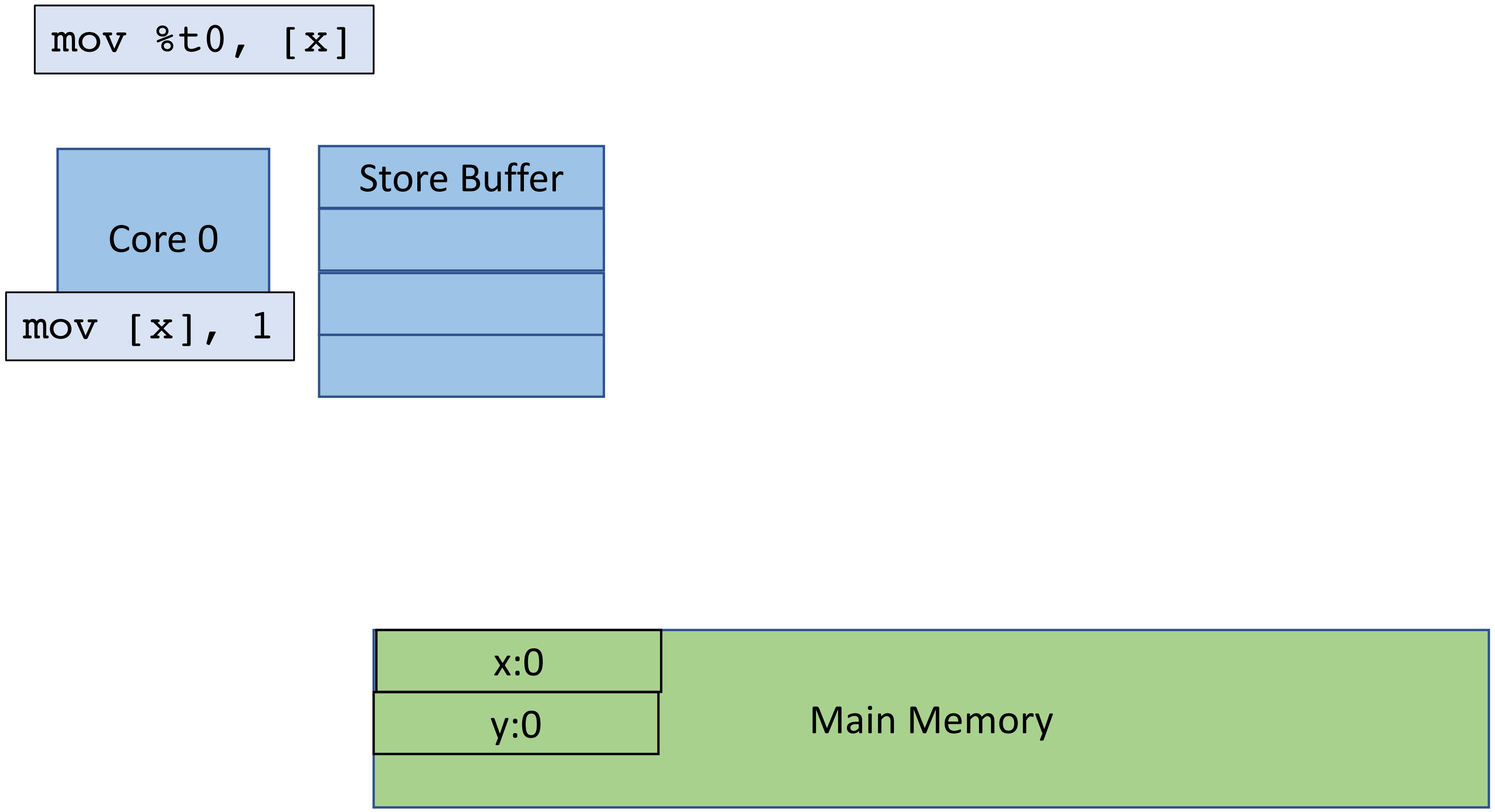
```
mov %t0, [x]
```

How does this execute?



Thread 0:

execute first instruction



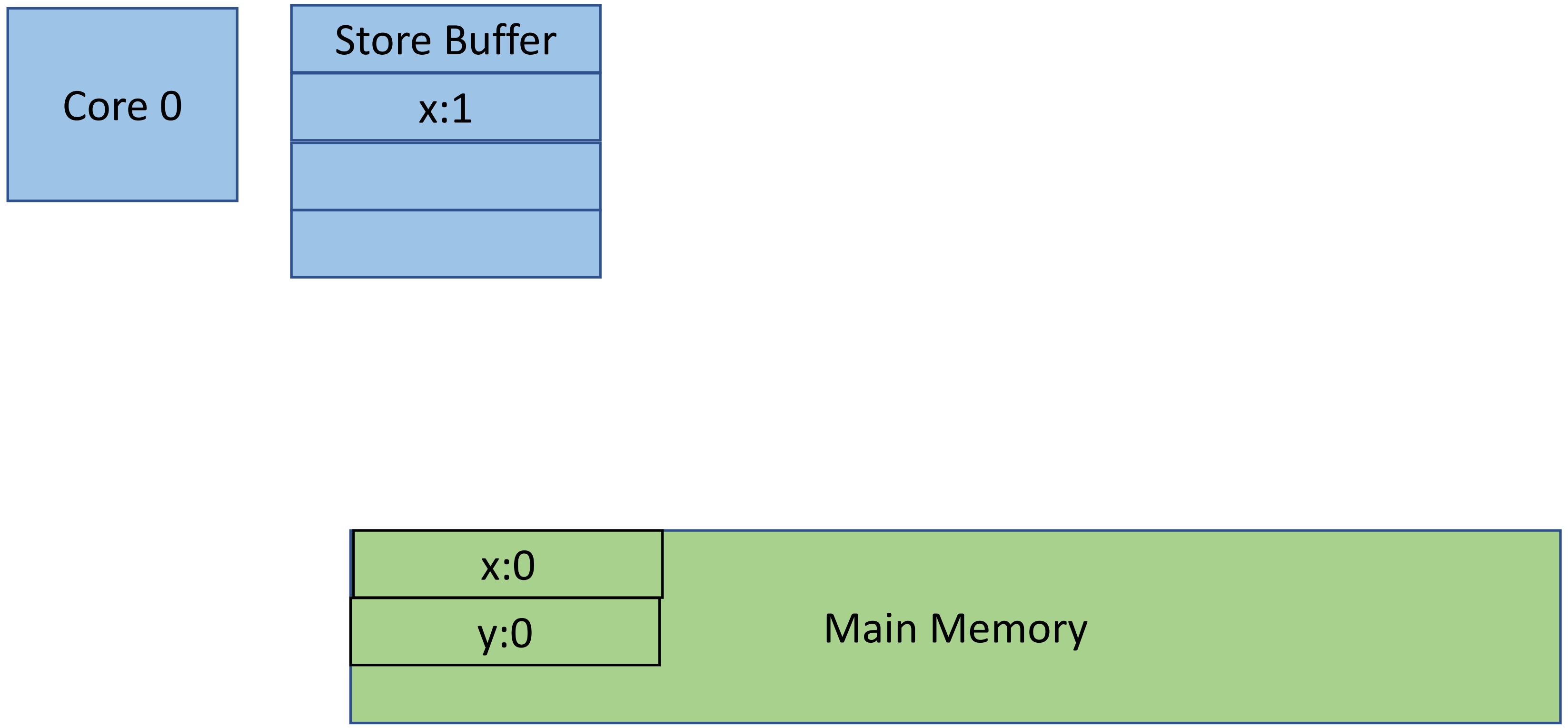
```
mov %t0, [x]
```

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

Thread 0:

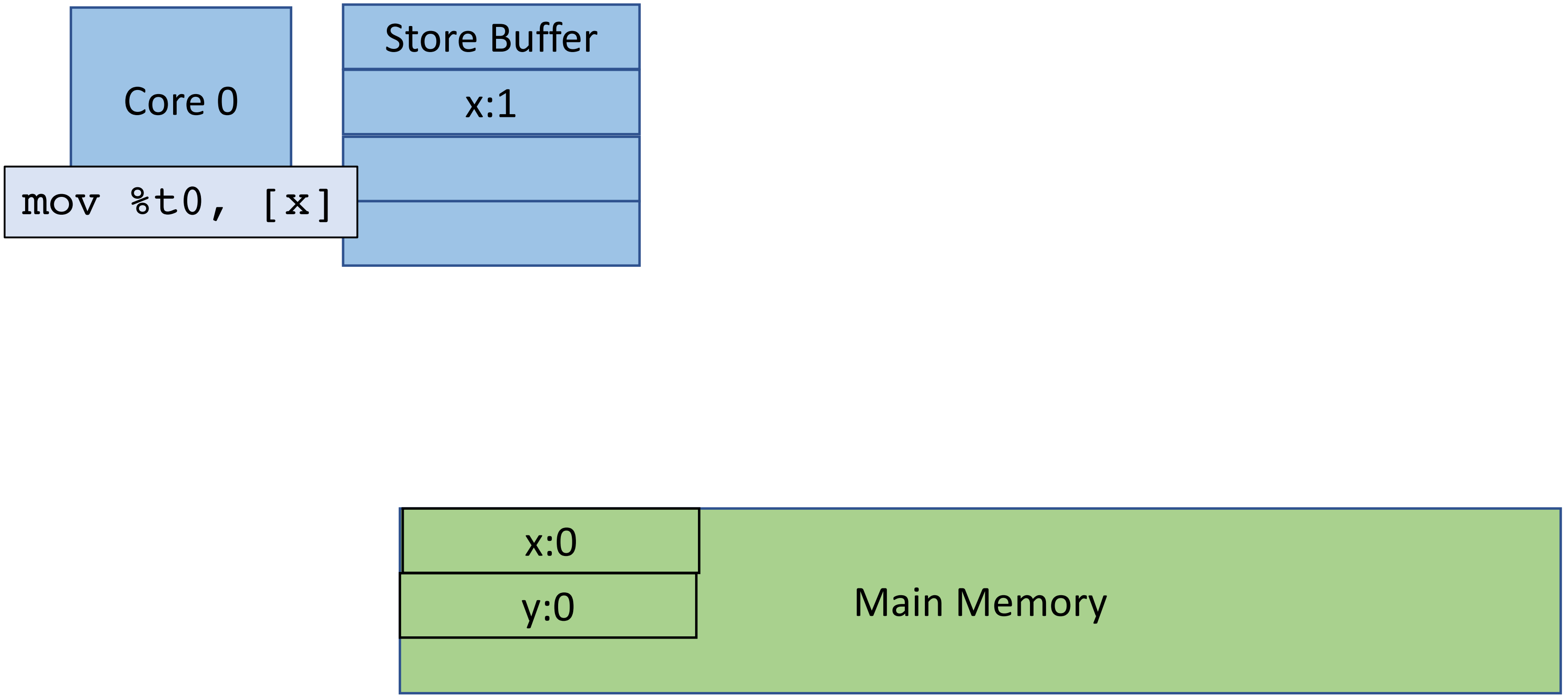
Store the value in the store buffer

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



Thread 0:

Next instruction

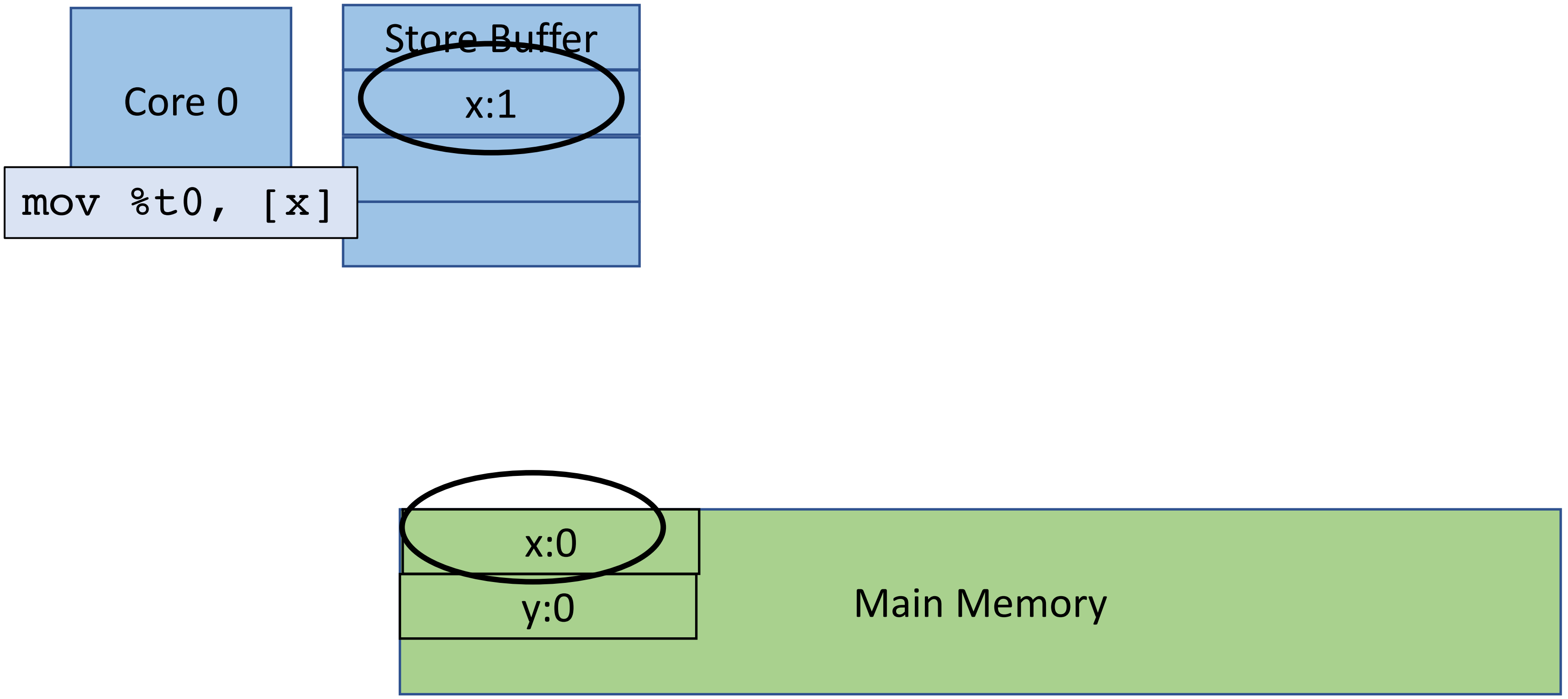


```
mov %t0, [x]
```

Thread 0:

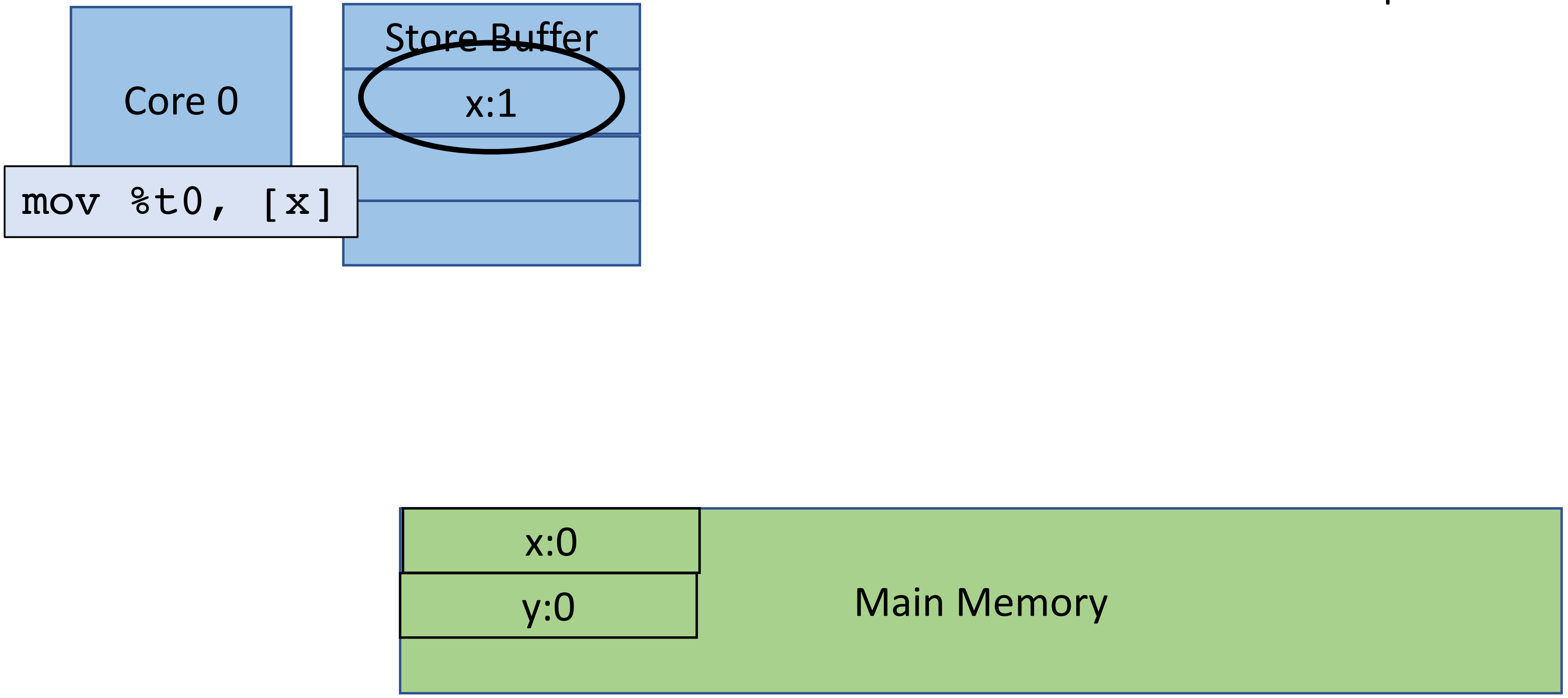
Where to load??

Store buffer?
Main memory?



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

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};
```

Can `t0 == t1 == 0?`

Thread 0:

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

Thread 1:

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

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

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

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

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

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};
```

Can `t0 == t1 == 0?`

Thread 0:

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

Thread 1:

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

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

```
mfence
```

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

```
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};
```

Can `t0 == t1 == 0?`

Thread 0:

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

Thread 1:

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

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

```
mfence
```

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

```
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

		memory access 0	
		L	S
memory access 1	L		
	S		

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

memory access 1		L	S
	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

memory access 1		L	S
	L	?	?
	S	?	?

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

PSO - partial store order

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

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

memory access 1

		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!

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};
```

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

Get out our lego bricks and try for sequential consistency

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)
```

Global variable:

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

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

Get out our lego bricks and try for TSO

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

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

Global variable:

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

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

Get out our lego bricks and try for PSO

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

memory access 1

	L	S
L	NO	Different address
S	NO	Different address

Global variable:

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

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

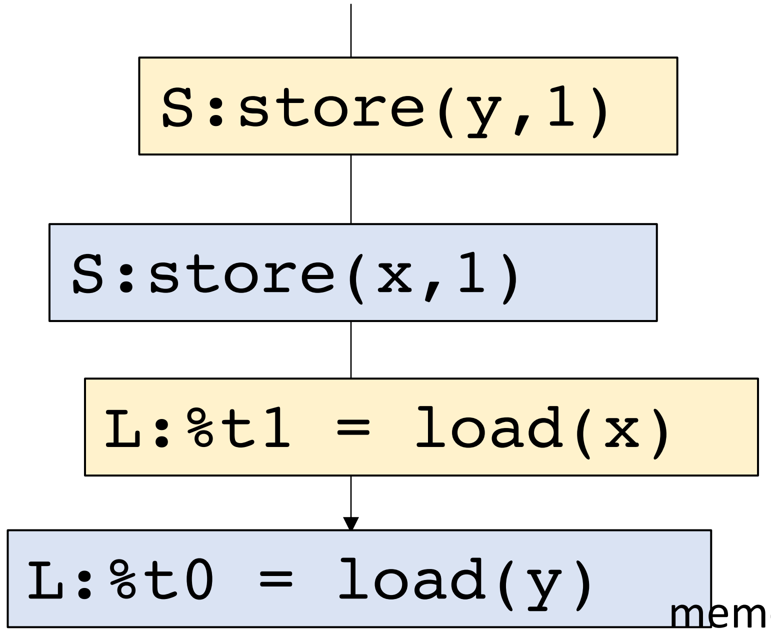
Get out our lego bricks and try for RMO

Thread 0:

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

Thread 1:

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



```
S:store(y,1)
S:store(x,1)
L:%t1 = load(x)
L:%t0 = load(y)
```

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

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

memory access 1

	L	S
L	YES	Different address
S	different address	Different address

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

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

C++

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

ISA

```
store(x,1);
fence;
```

or

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

$$\begin{tikzcd}
\text{C++} & \text{ISA} \\
\boxed{\texttt{x.store(1);}} \arrow[r] & \boxed{\begin{array}{l}\texttt{store(x,1);} \\ \texttt{fence;}\end{array}}
\end{tikzcd}$$

or

$$\begin{tikzcd}
\boxed{\texttt{z = x.load()}} \arrow[r] & \boxed{\begin{array}{l}\texttt{fence;} \\ \texttt{\%z = load(x);}\end{array}}
\end{tikzcd}$$

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

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

C++	ISA
`x.store(1);`	`store(x,1); fence;`

or

C++	ISA
`z = x.load()`	`fence; %z = load(x);`

This should help you see why you want to reduce the number of atomic load/stores in your program

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

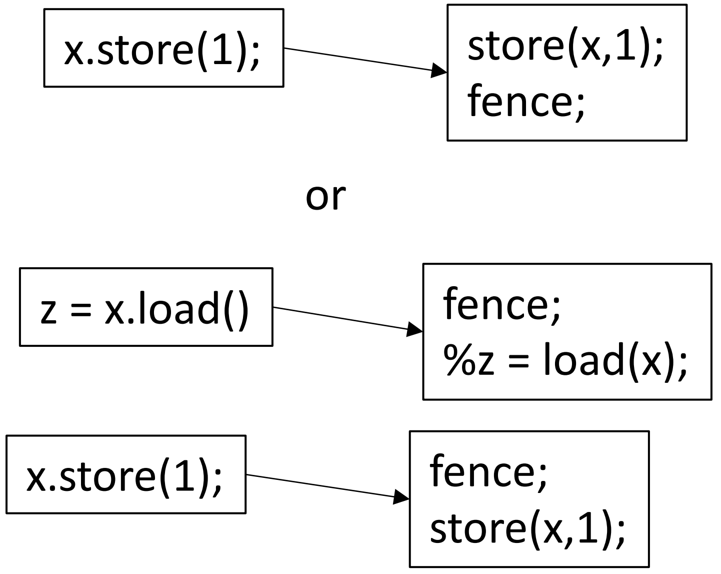
	L	S
L	NO	different address
S	NO	different address

C++11 atomic operation compilation

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

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

Memory orders

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

Where have we seen `memory_order_relaxed`?

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 ommitted)
  - `memory_order_consume`

Memory consistency in the real world

- Historic Chips:
    - X86: TSO
        - Surprising robost
        - 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

Thanks!

- Have a nice weekend!
- On Tuesday, we will talk about decoupled access execute (DAE)