CSE211: Compiler Design

Nov. 27, 2023

- **Topic**: Loop structure and DSLs

- **Discussion questions**:
  - *Lots of discussions throughout about loops and DSLs*

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Announcements

- Homework 3 is due on FRIDAY
  - Two day extension
  - No extension on HW 4
    - It will be 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
  - 10 minutes
  - Things don’t have to be completed
  - But you should be able to present at least one result and your approach.

- 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.

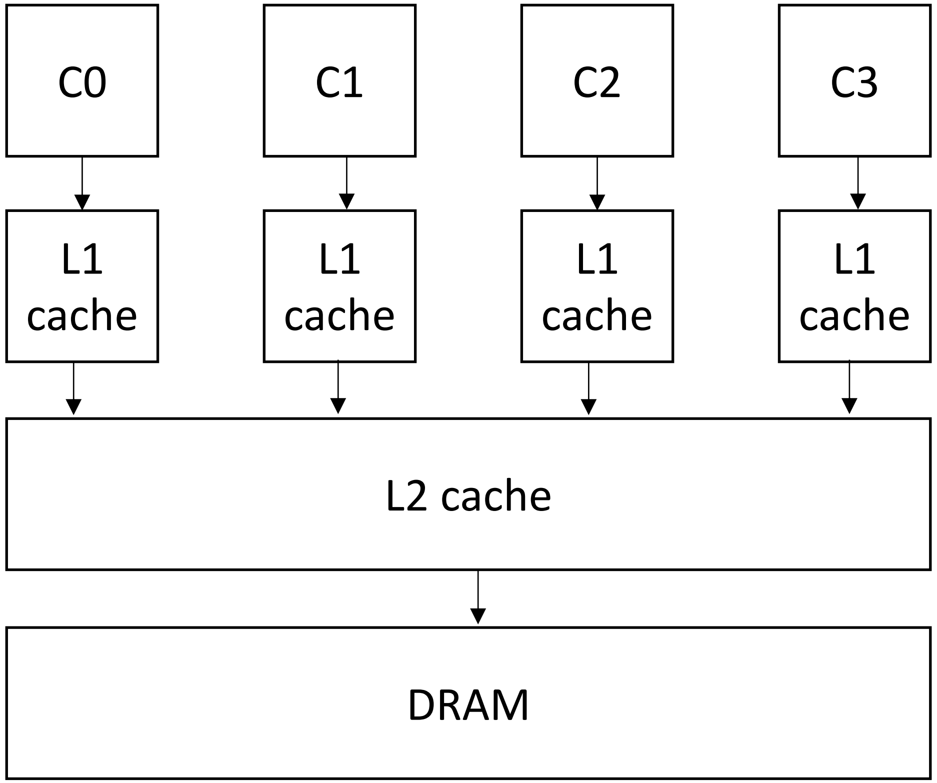
Guest lecture next time!

- Two presenters from Google about using ML in compilers:
  - Ondrej Sykora – GRANITE: using ML to estimate the throughput of basic blocks
  - Mircea Trofin – MLGO: using ML to pick when to apply compiler optimizations
- Both papers linked in canvas announcement: please try to overview the papers before the lecture
- Mircea will be around for the day. Let me know if you’d like to meet with him and I can organize.

Review

Shifting our focus back to a single core

- We need to consider single threaded performance
- Good single threaded performance can enable better parallel performance
  - **Memory locality** is key to good parallel performance.



Discussion

Discussion questions:

What is a DSL?

What are the benefits and drawbacks of a DSL?

What DSLs have you used?

Halide:

pretty straight
forward computation
for brightening

(1 pass over all pixels)

This computation is known as the "Local Laplacian Filter". Requires visiting all pixels 99 times

We want to be able to do this
fast and efficiently!

Main results in from Halide show a 1.7x speedup with 1/5 the LoC over hand optimized versions at Adobe

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Decoupling computation from optimization

- We love Halide not only because it can make pretty pictures very fast
- We love it because it changed the level of abstraction for thinking about computation and optimization
- (Halide has been applied in many other domains now, turns out everything is just linear algebra)

Halides approach

- Decouple
  - what to compute (the program)
  - with how to compute (the optimizations, also called the schedule)

```
for (int y = 0; y < y_size; y++) {
  for (int x = 0; x < x_size; x++) {
        a[x,y] = b[x,y] + c[x,y];
   }
}
```

program

```
add(x,y) = b(x,y) + c(x,y)
```

schedule

```
add.order(x,y)
```

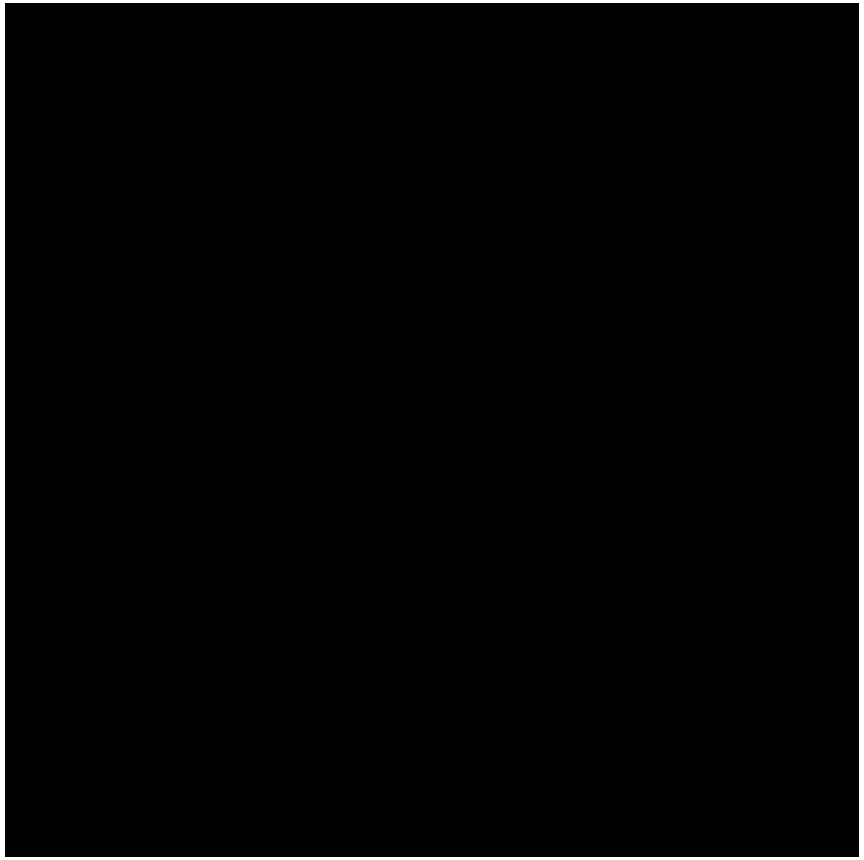
Halide (high-level)

C++:

```
Halide::Func gradient;
Halide::Var x, y;
gradient(x, y) = x + y;
Halide::Buffer<int32_t> output =
                gradient.realize({16, 16});
```

Schedule

```
Var x_outer, x_inner, y_outer, y_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.split(y, y_outer, y_inner, 4);
gradient.reorder(x_inner, y_inner, x_outer, y_outer);
```



```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

```
gradient.tile(x, y,
              x_outer, y_outer,
              x_inner, y_inner, 4, 4);
```

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Finally: a fast schedule that they found:

```
Halide::Func gradient_fast;
Halide::Var x, y;
gradient_fast(x, y) = x + y;
Halide::Buffer<int32_t> output =
              gradient.realize({2, 2});



Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient_fast
              .tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64)
              .fuse(x_outer, y_outer, tile_index)
              .parallel(tile_index);

Var x_inner_outer, y_inner_outer, x_vectors, y_pairs;
gradient_fast
       .tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 2)
       .vectorize(x_vectors)
       .unroll(y_pairs);
```

New material

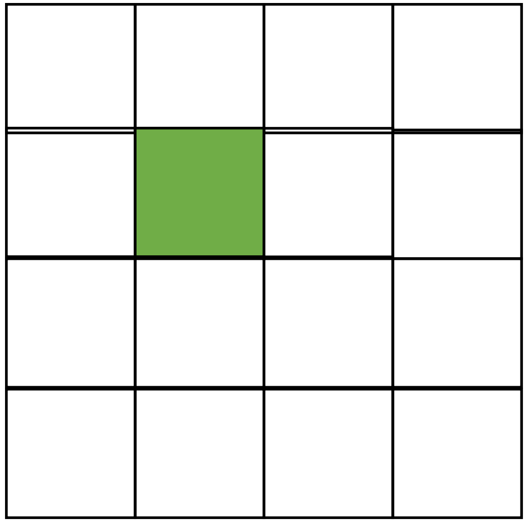
function fusing...

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

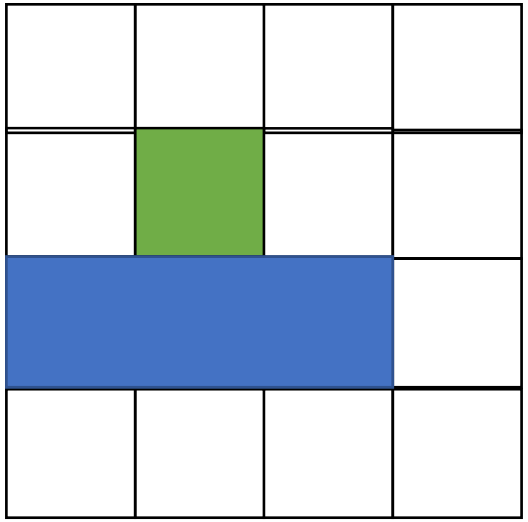
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



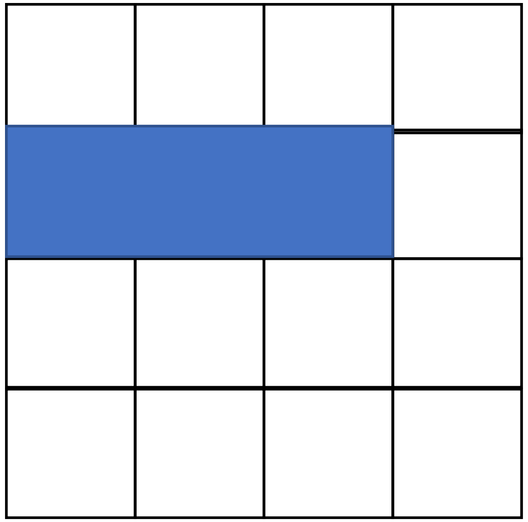
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



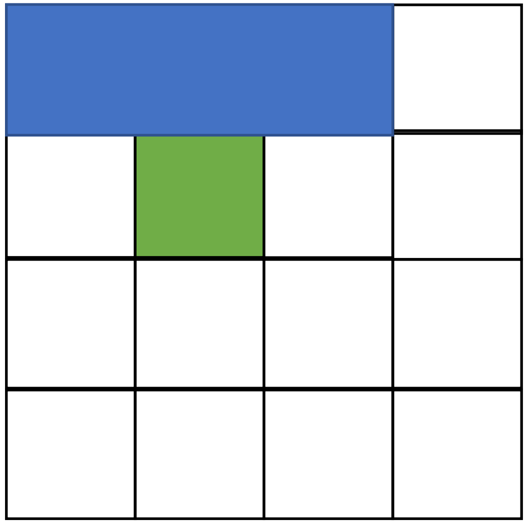
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



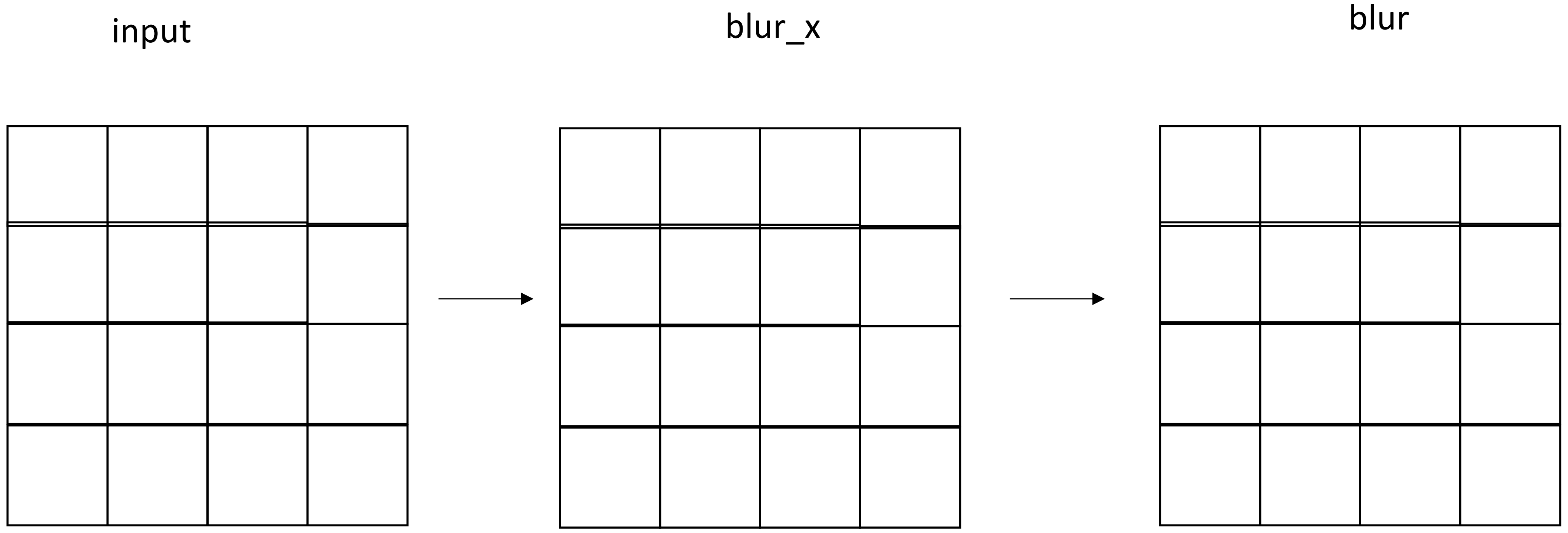
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

how to compute?

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



Example: unnormalized blur

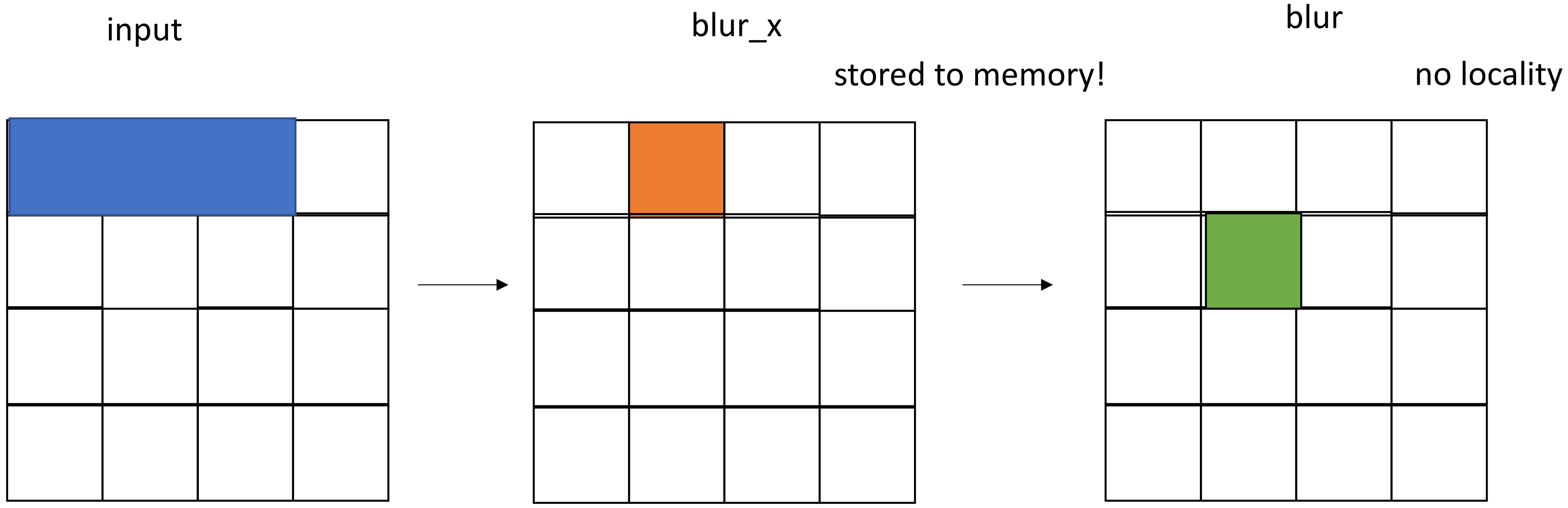
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

```
alloc blurx[2048][3072]
foreach y in 0..2048:
    foreach x in 0..3072:
      blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]


alloc out[2046][3072]
foreach y in 1..2047:
    foreach x in 0..3072:
      out[y][x] = blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

pros?
cons?

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

Other options?

Example: unnormalized blur

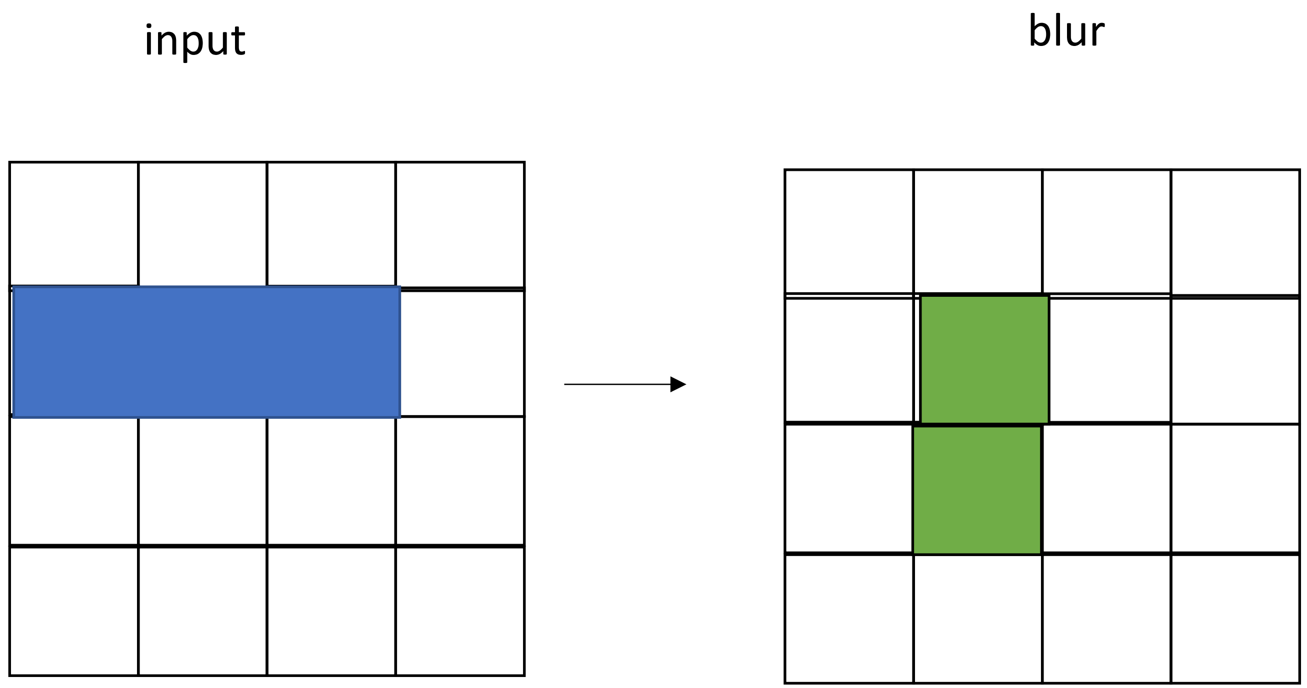
```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

completely inline

```
alloc out[2046][3072]
foreach y in 1..2047:
    foreach x in 0..3072:
      out[y][x] = in[y-1][x] + in[y][x] + in[y+1][x] +
                  in[y-1][x-1] + in[y][x-1] + in[y+1][x-1]
                  in[y-1][x+1] + in[y][x+1] + in[y+1][x+1]
```

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```



These two squares will both sum up the same values in blue

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

other ideas?

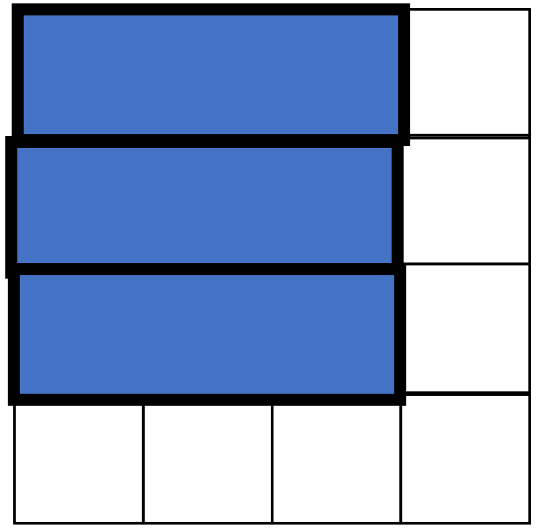
Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

first iteration, only compute blur_x

blur

sliding window

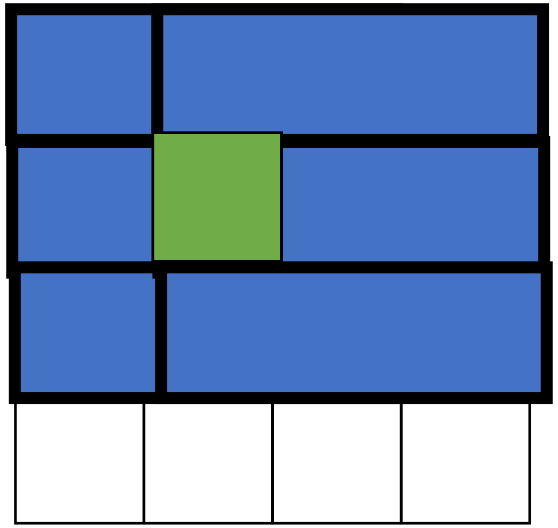


Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

sliding window

blur



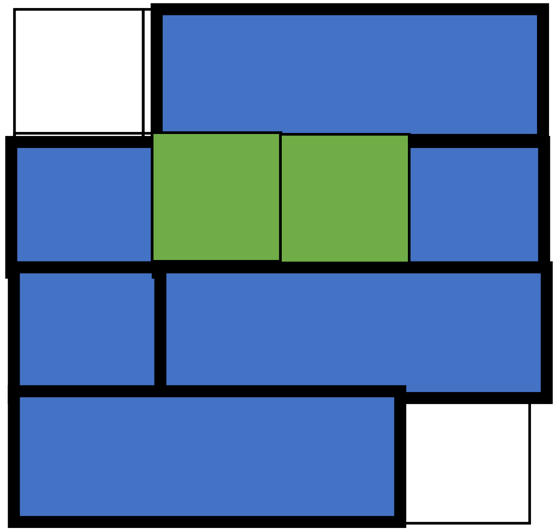
first iteration, only compute blur_x
second iteration, compute blur_x again:
Compute first blur

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

sliding window

blur



first iteration, only compute blur_x
second iteration, compute blur_x again:
Compute first blur

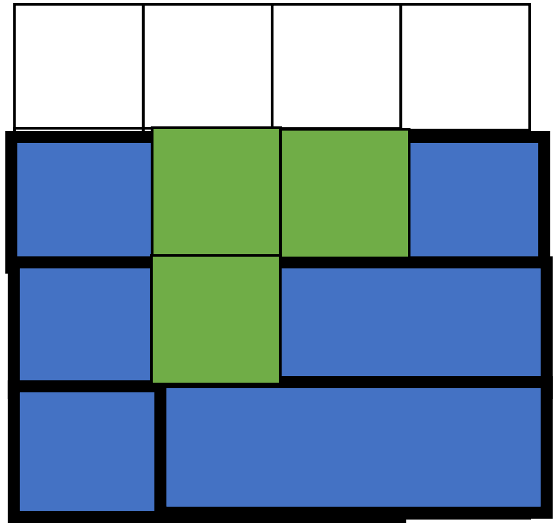
Third iteration
drop first bar
Compute second blur
compute one next row

Example: unnormalized blur

```
Halide::Func blur_x(x,y) = in(x-1,y) + in(x,y) + in(x+1,y);

Halide::Func blur(x,y) = blur_x(x,y+1) + blur_x(x,y) + blur_x(x,y-1);
```

sliding window

blur



first iteration, only compute blur_x
second iteration, compute blur_x again:
Compute first blur

Third iteration
drop first bar
Compute second blur
compute one next row

Fourth iteration
Drop second bar
Compute third blur
Compute one next row

Fusing functions

- Can compose with all other optimizations
  - Tiling, loop order, unrolling, etc.

  - Creates a very powerful optimization framework, and automatically produces code that you do not want to write by hand!

End Halide

Next topic: Compiling concurrency

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);
```

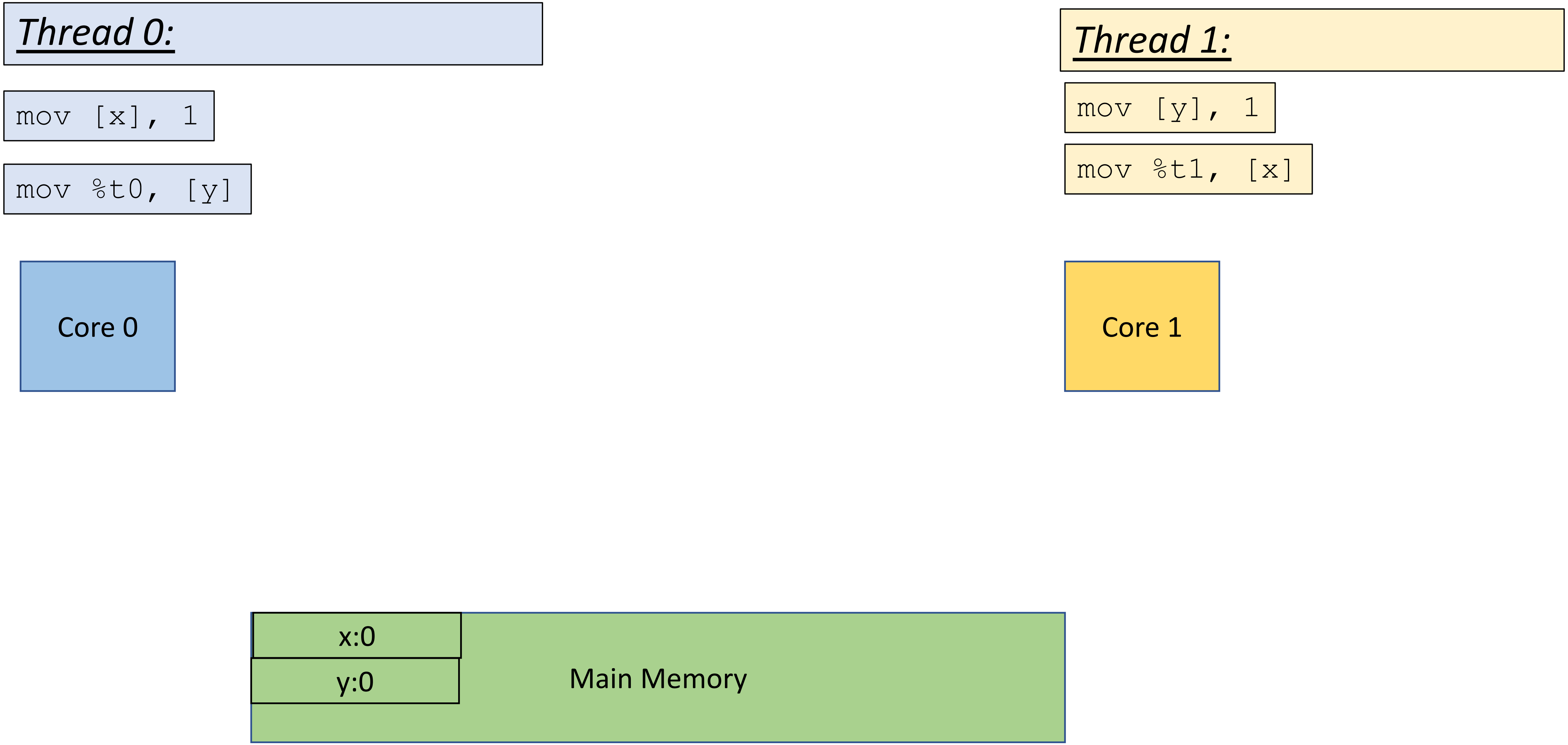
```
S:store(x, 1);
S:store(y, 1);
L:%t0 = load(y);
L:%t1 = load(x);
```

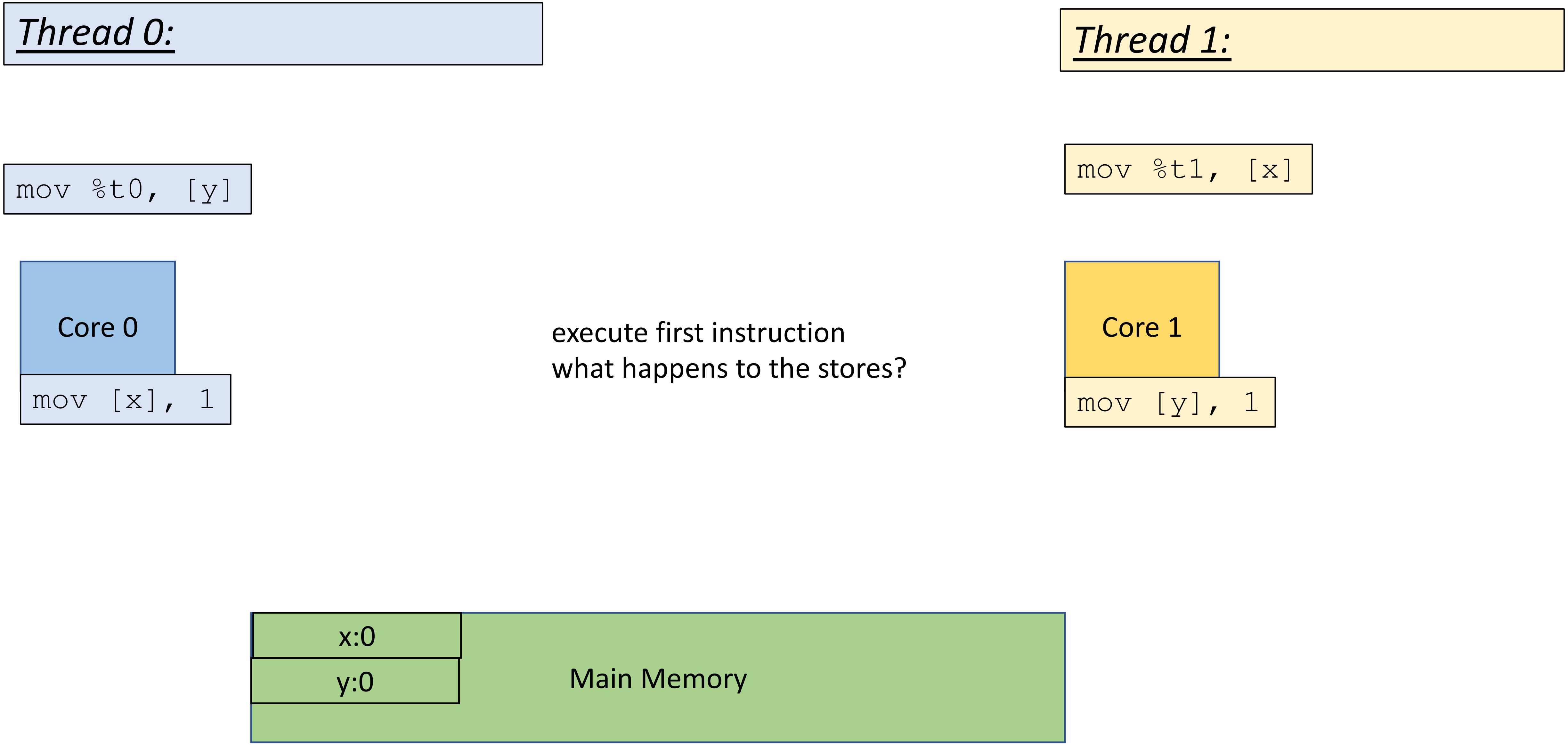
pick from the top of the pile of either thread

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]
Core 0
Core 1
x:0
y:0
Main Memory


Thread 0:
mov %t0, [y]
Core 0
mov [x], 1
Thread 1:
mov %t1, [x]
Core 1
mov [y], 1
execute first instruction
what happens to the stores?
x:0
y:0
Main Memory

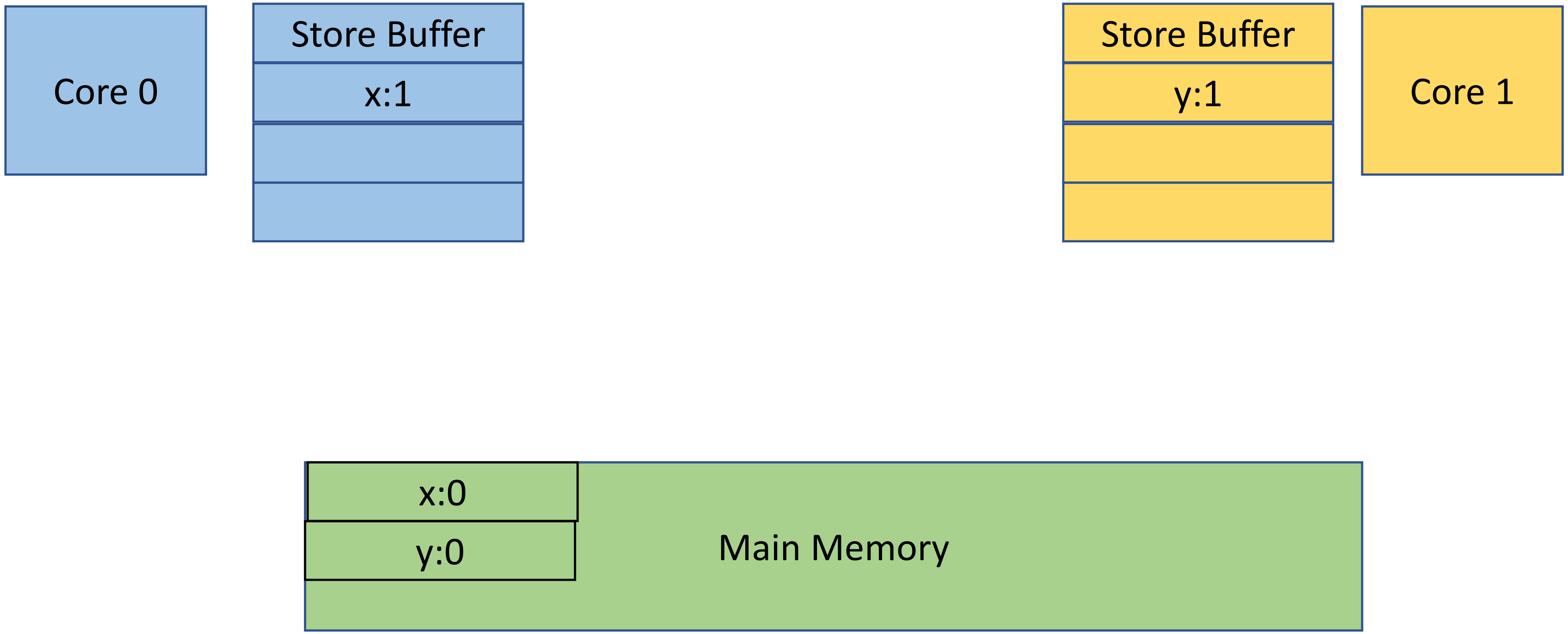
Thread 0:

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

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

Thread 1:

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



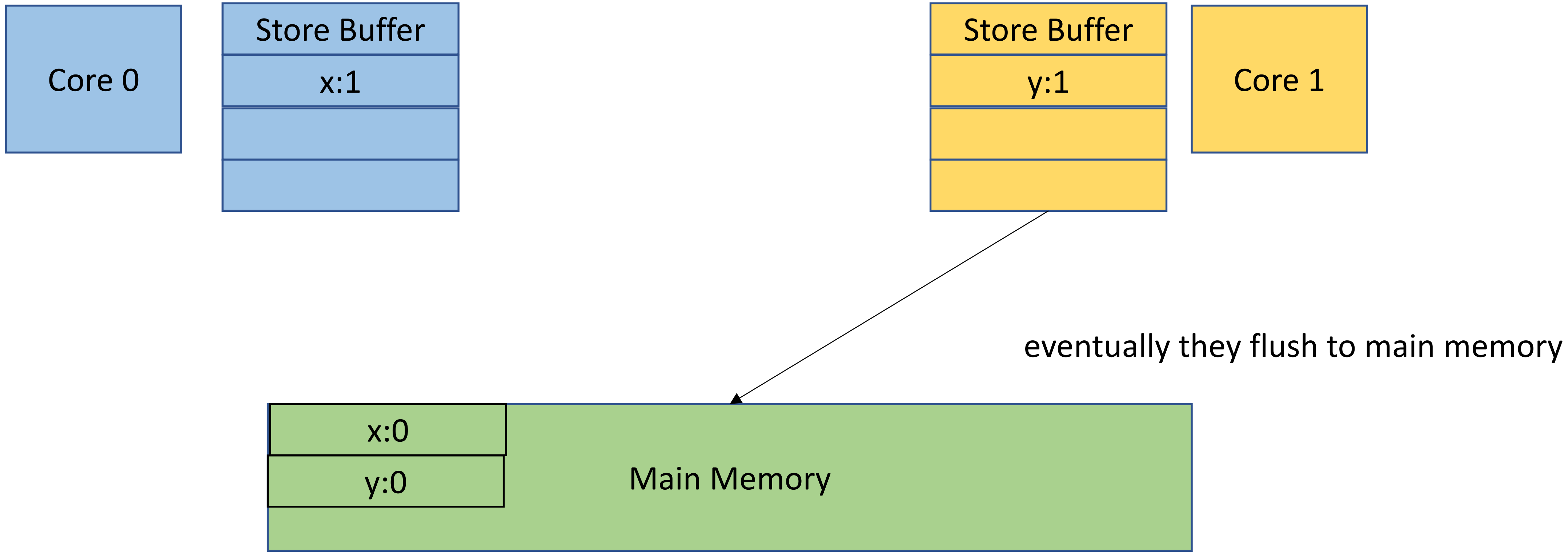
Thread 0:

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

Thread 1:

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

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



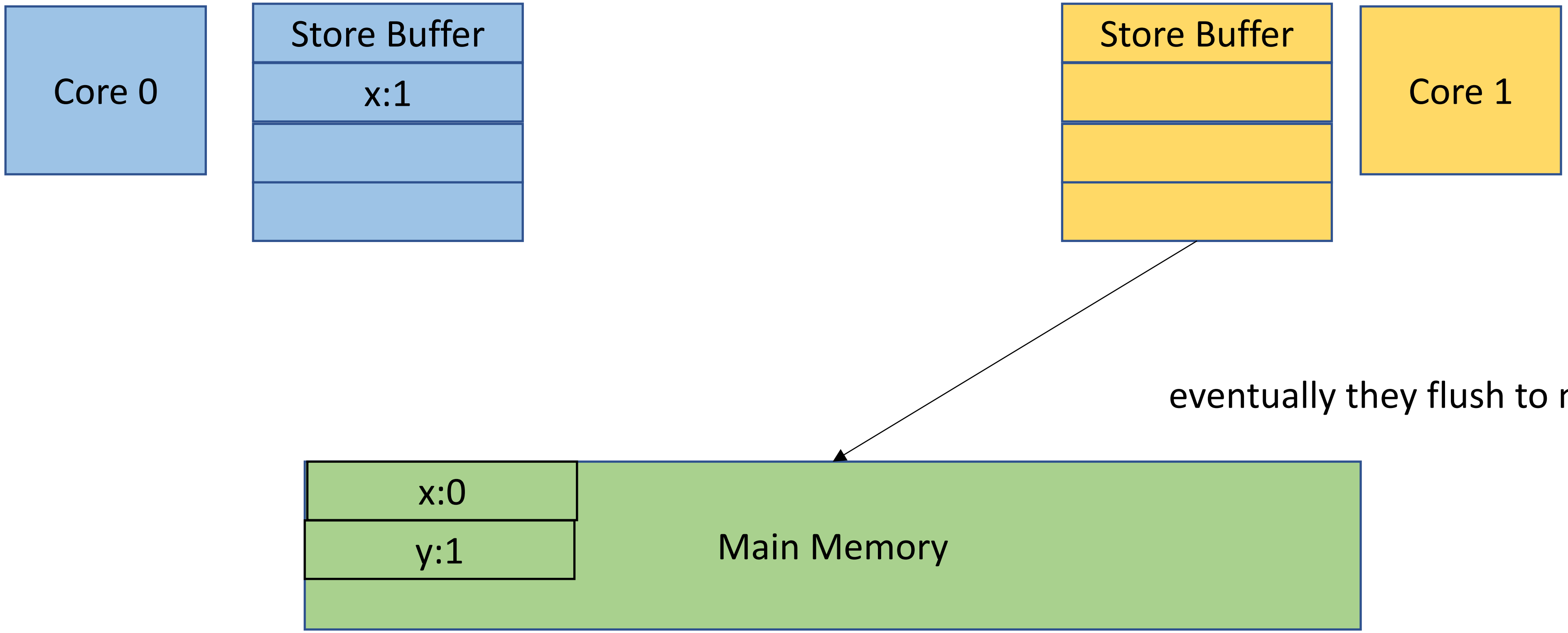
Thread 0:

```
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

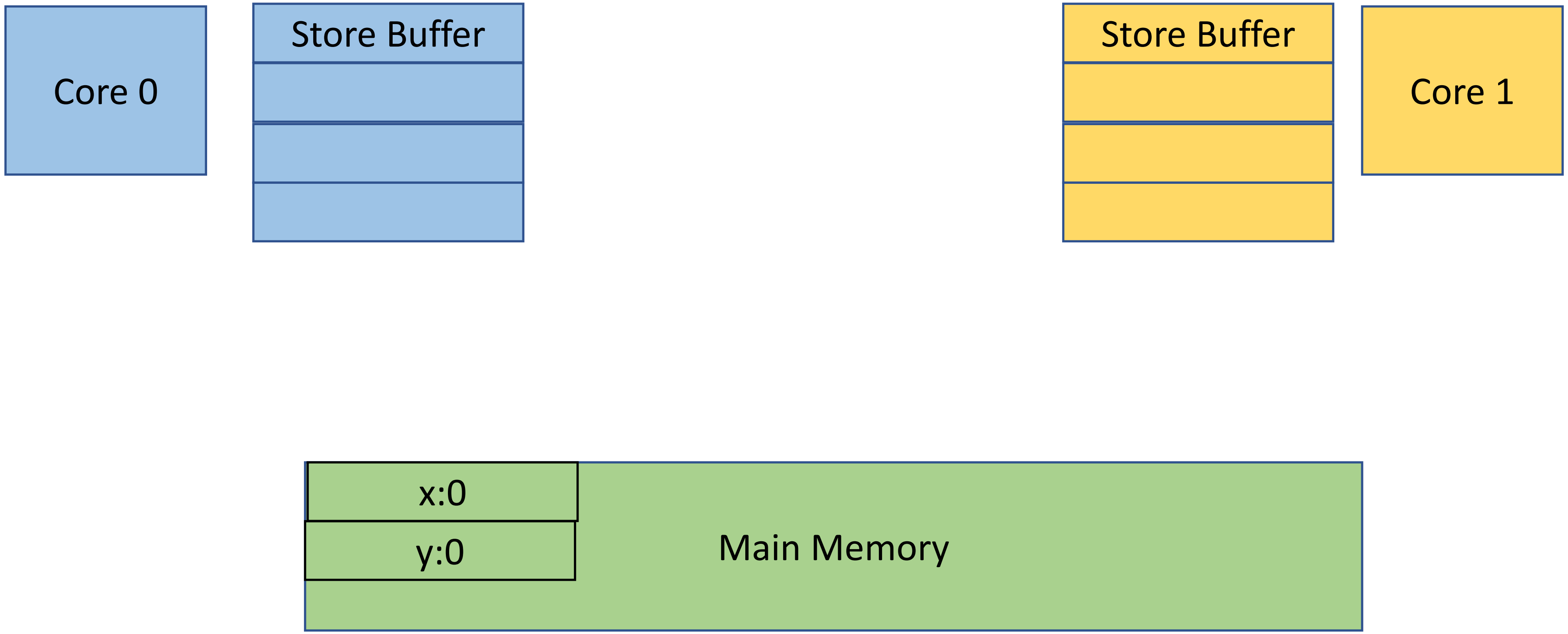
Thread 0:

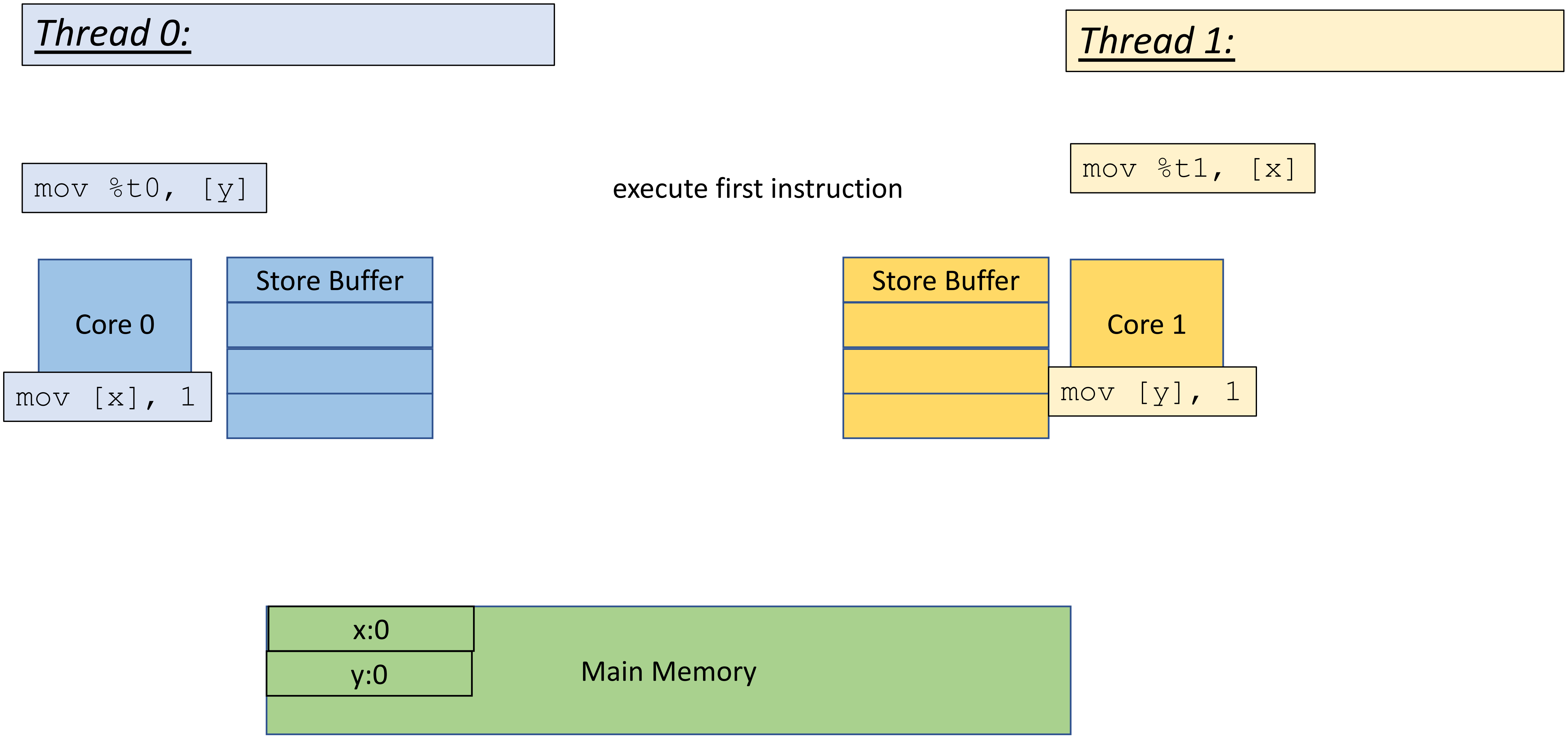
```
mov [x], 1
mov %t0, [y]
```

Thread 1:

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

rewind




Thread 0:
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

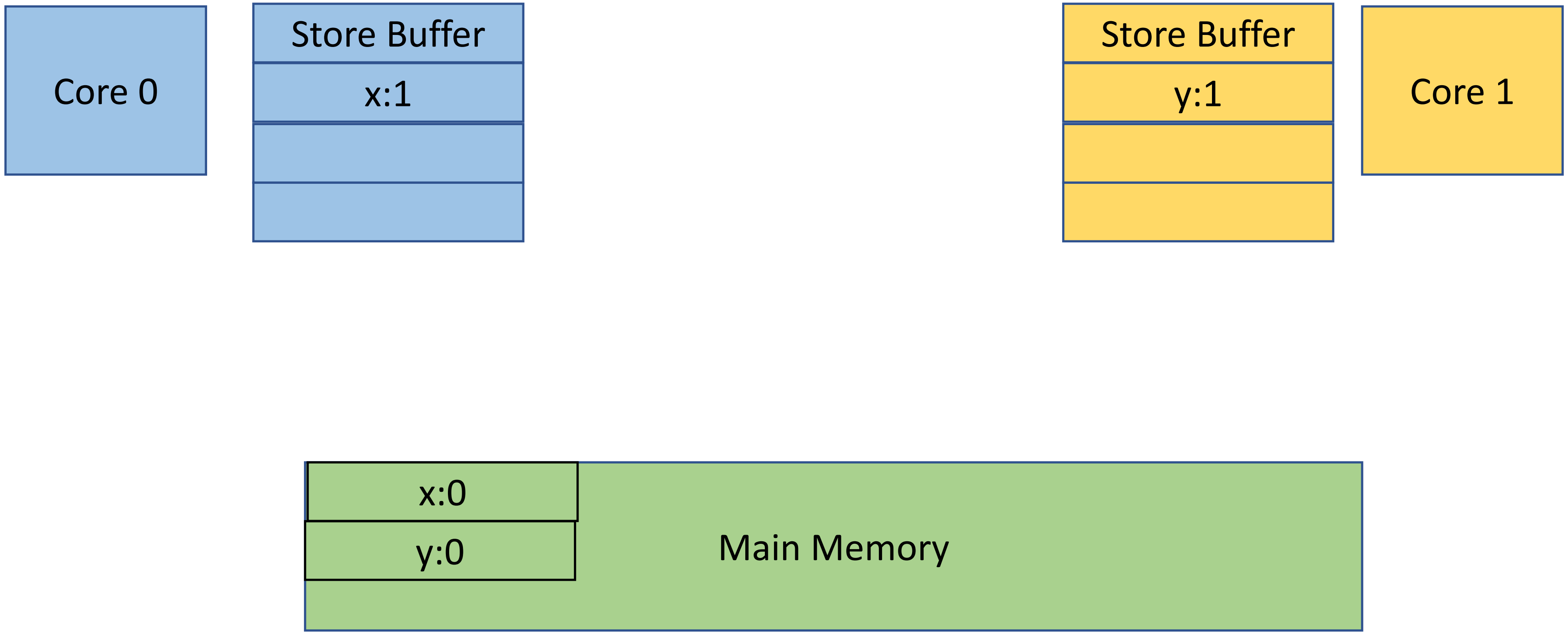
Thread 0:

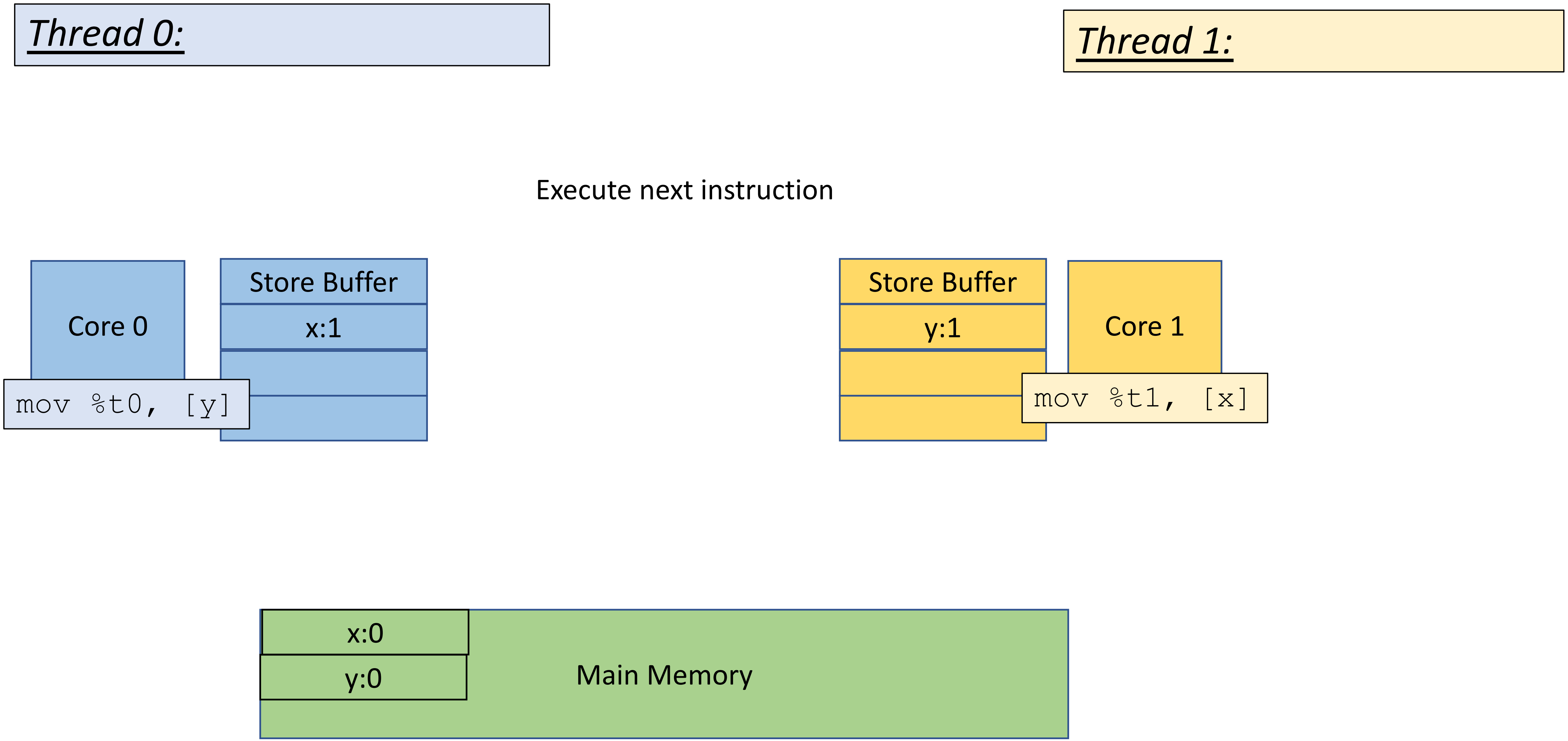
```
mov %t0, [y]
```

Thread 1:

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

values get stored in SB

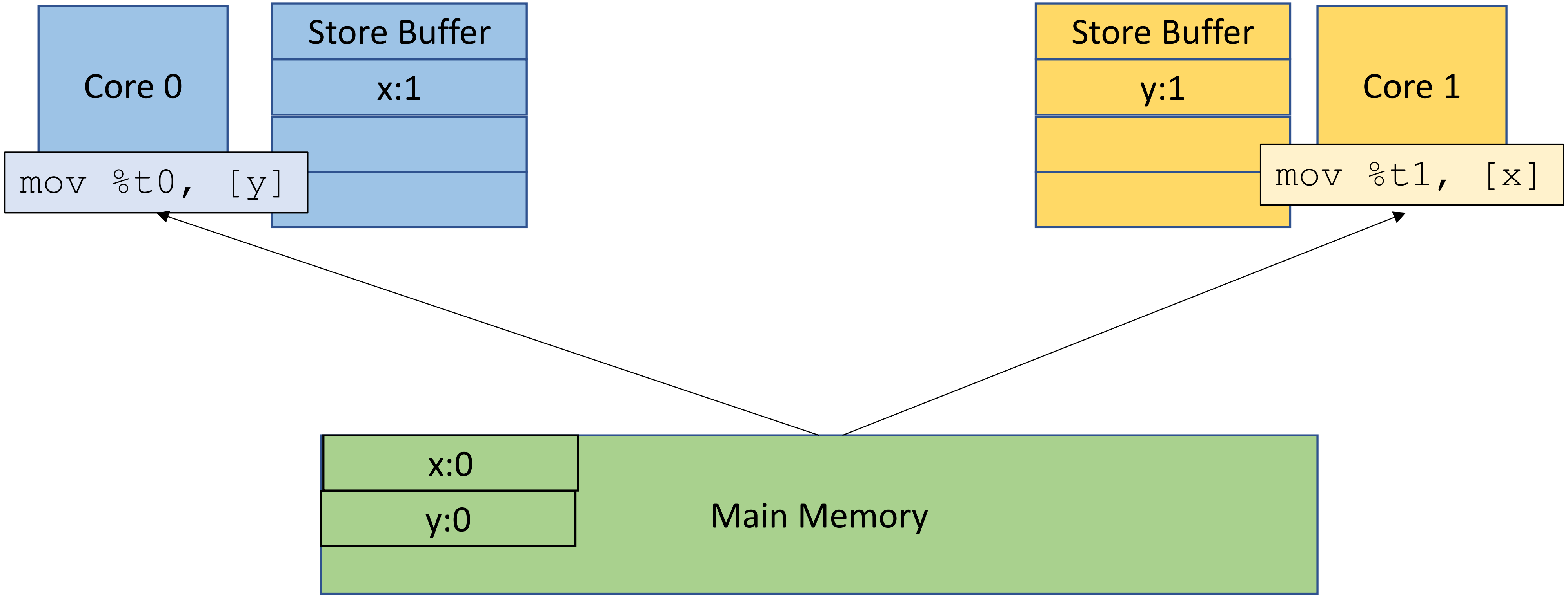



Thread 0:
Thread 1:
Execute next instruction
Core 0
Store Buffer
x:1
mov %t0, [y]
Store Buffer
y:1
Core 1
mov %t1, [x]
x:0
y:0
Main Memory

Thread 0:

Thread 1:

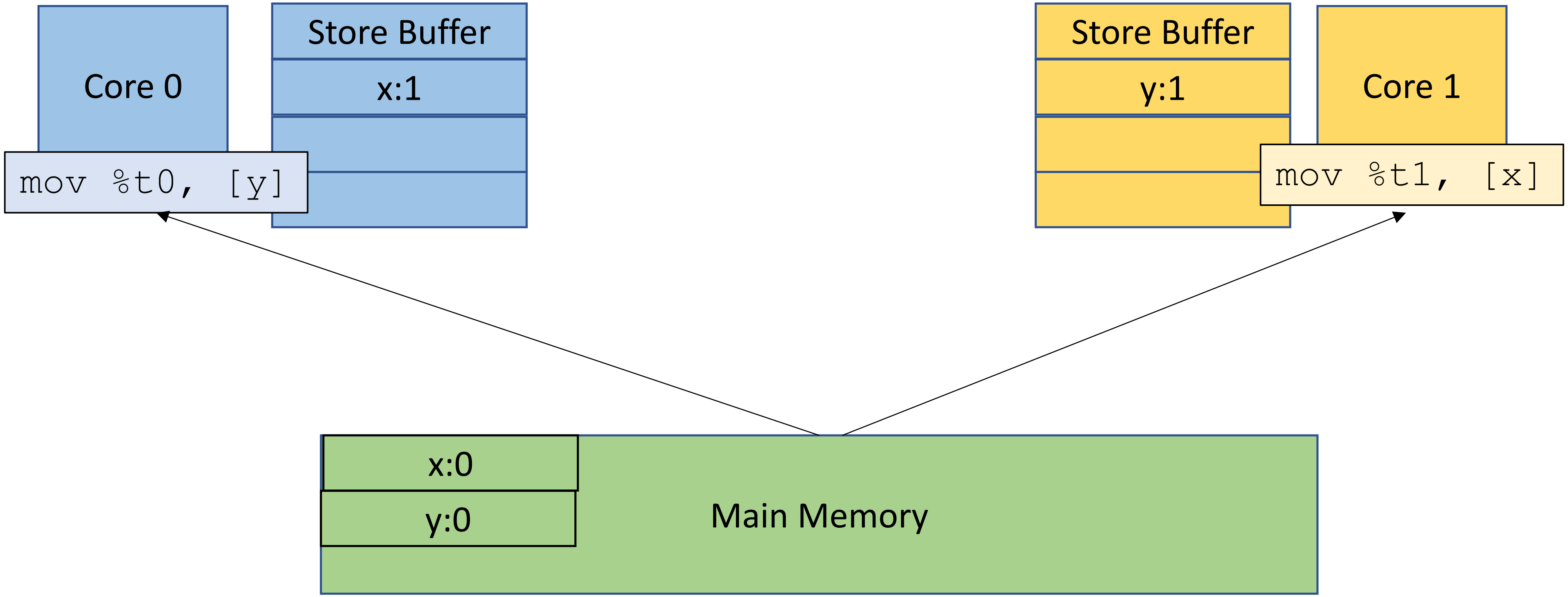
Values get loaded from memory



Thread 0:

Thread 1:

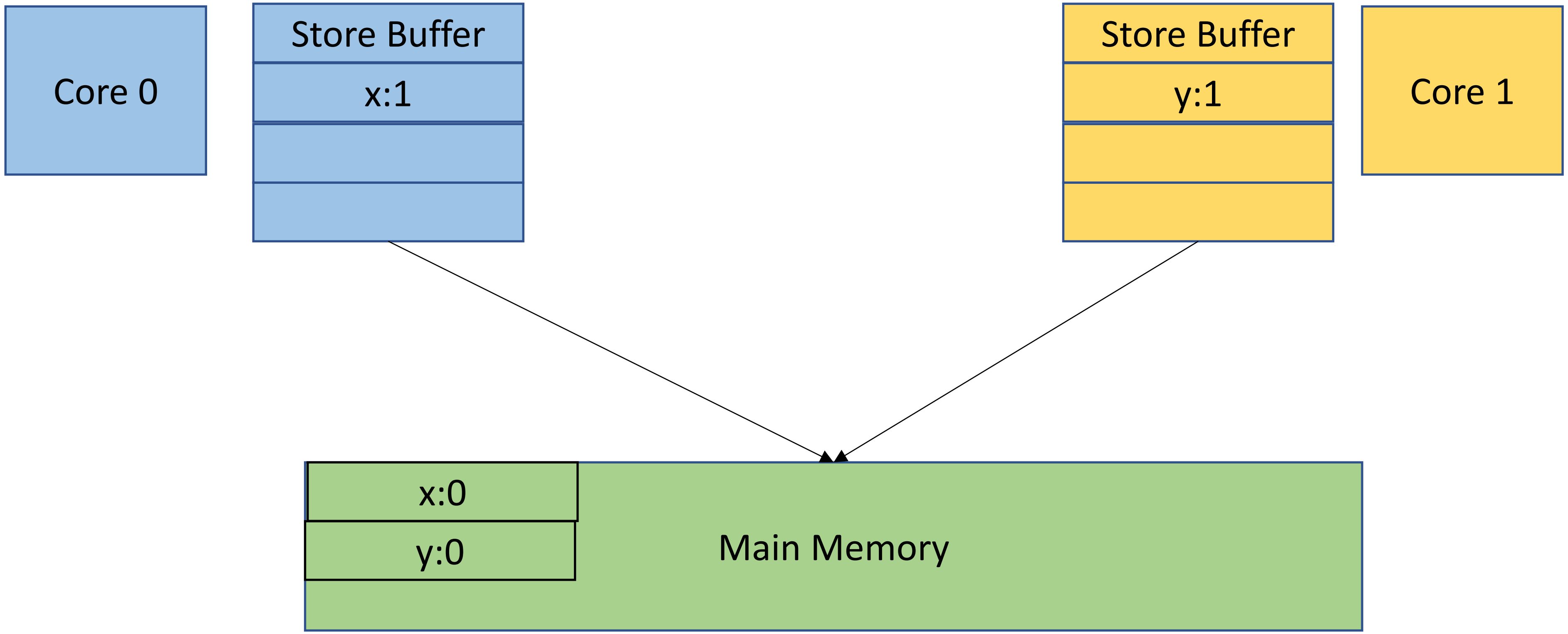
we see `t0 == t1 == 0`!



Thread 0:

Thread 1:

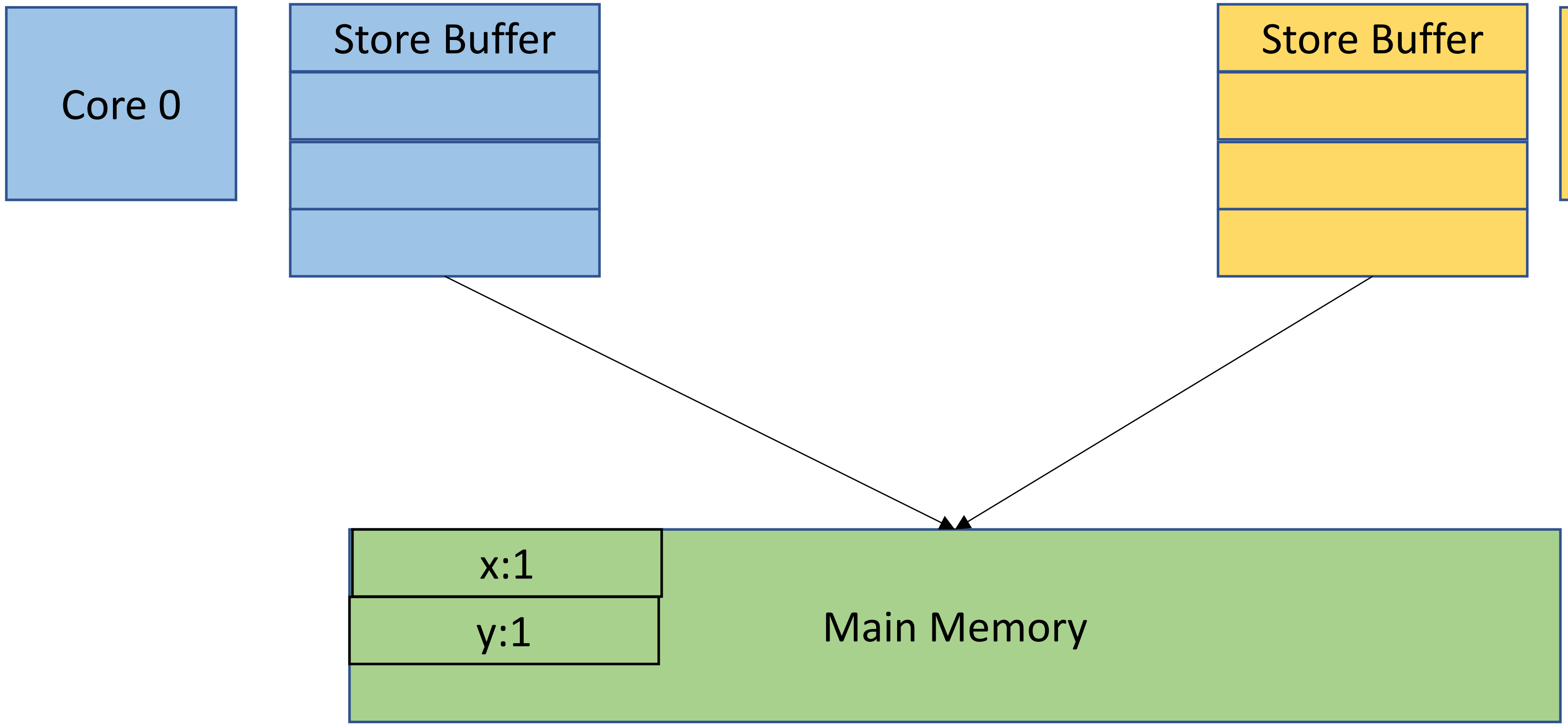
Store buffers are drained eventually



__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]
```

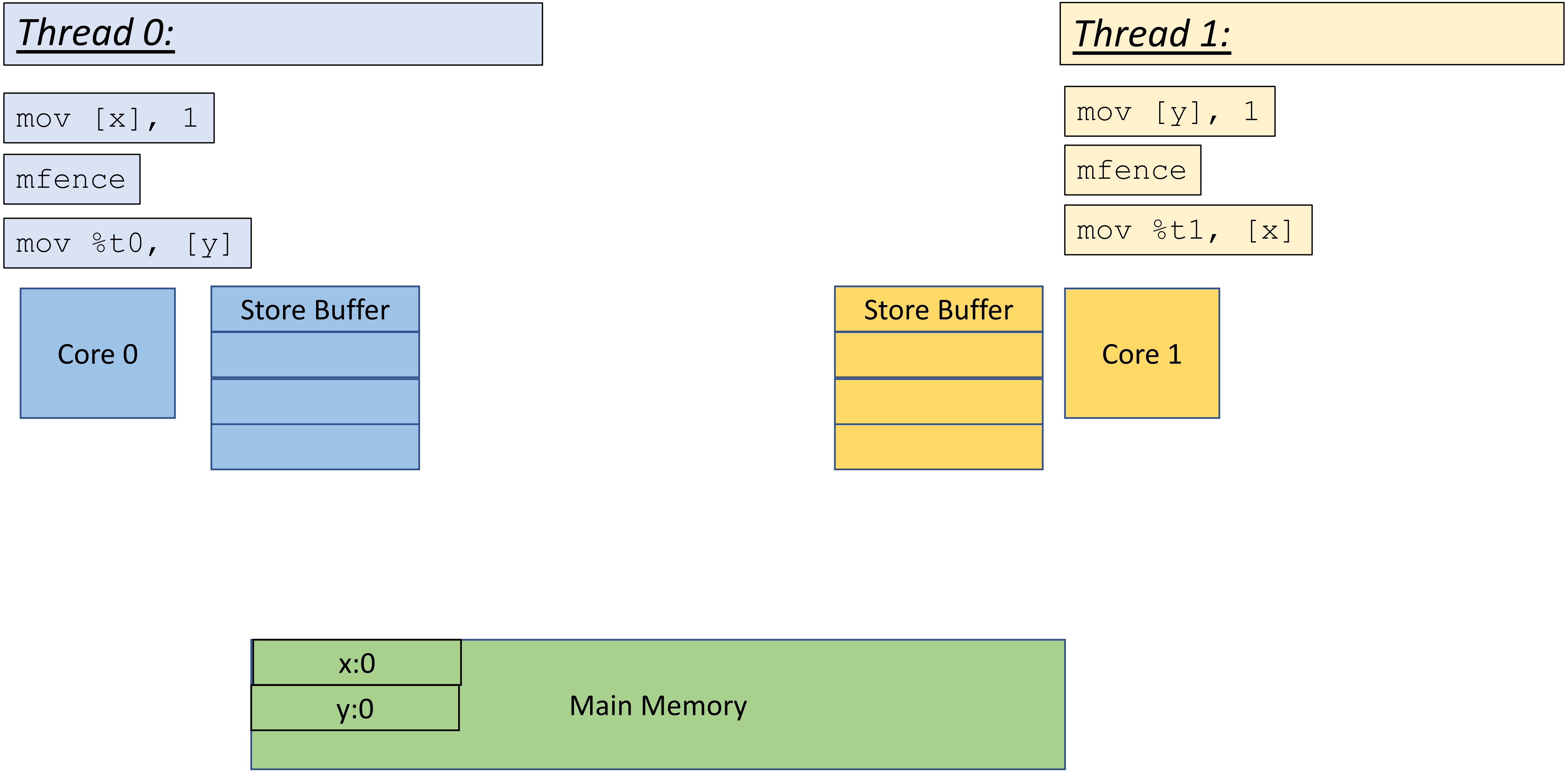
Thread 1:

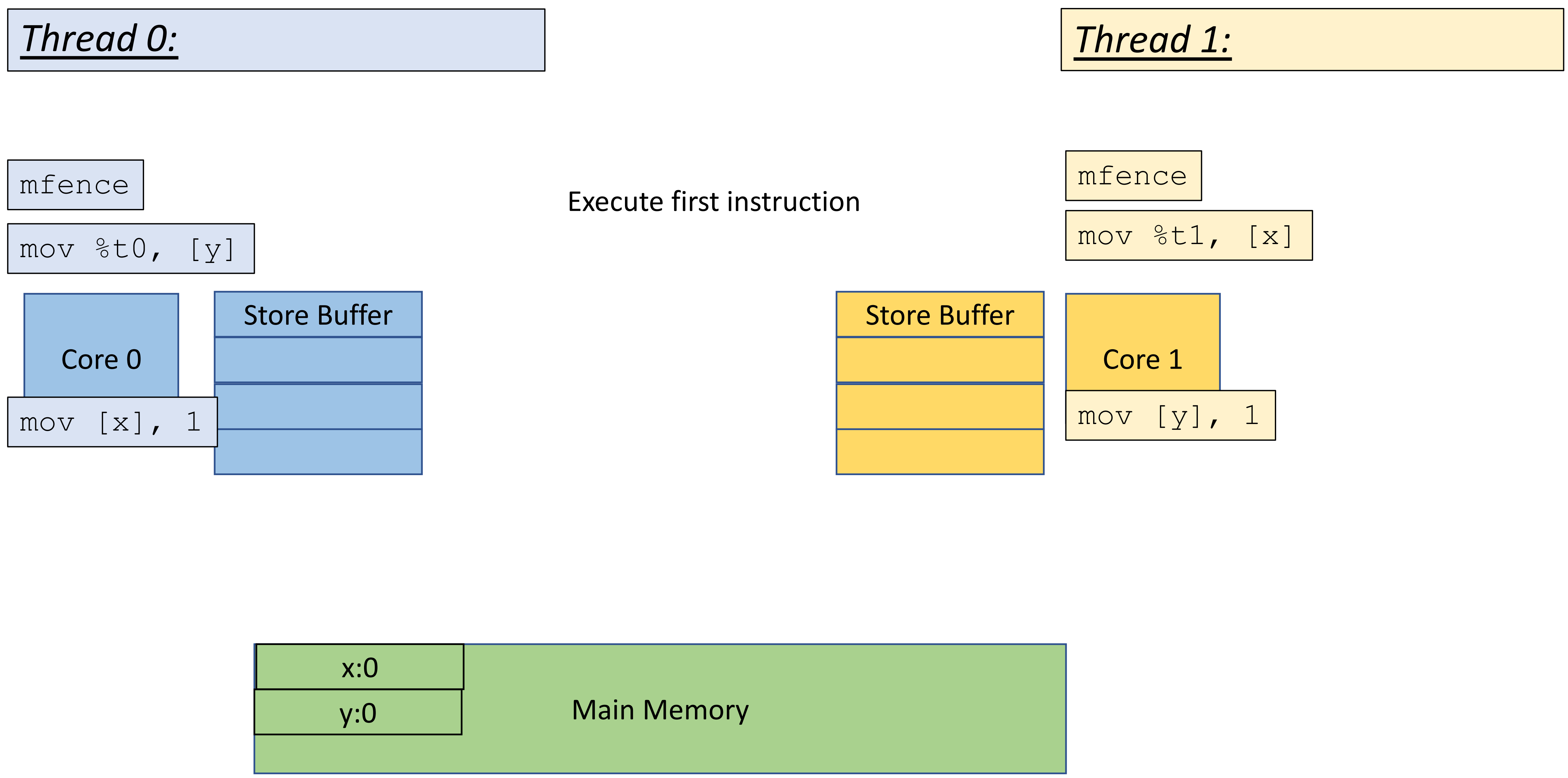
```
mov [y], 1
mov %t1, [x]
```

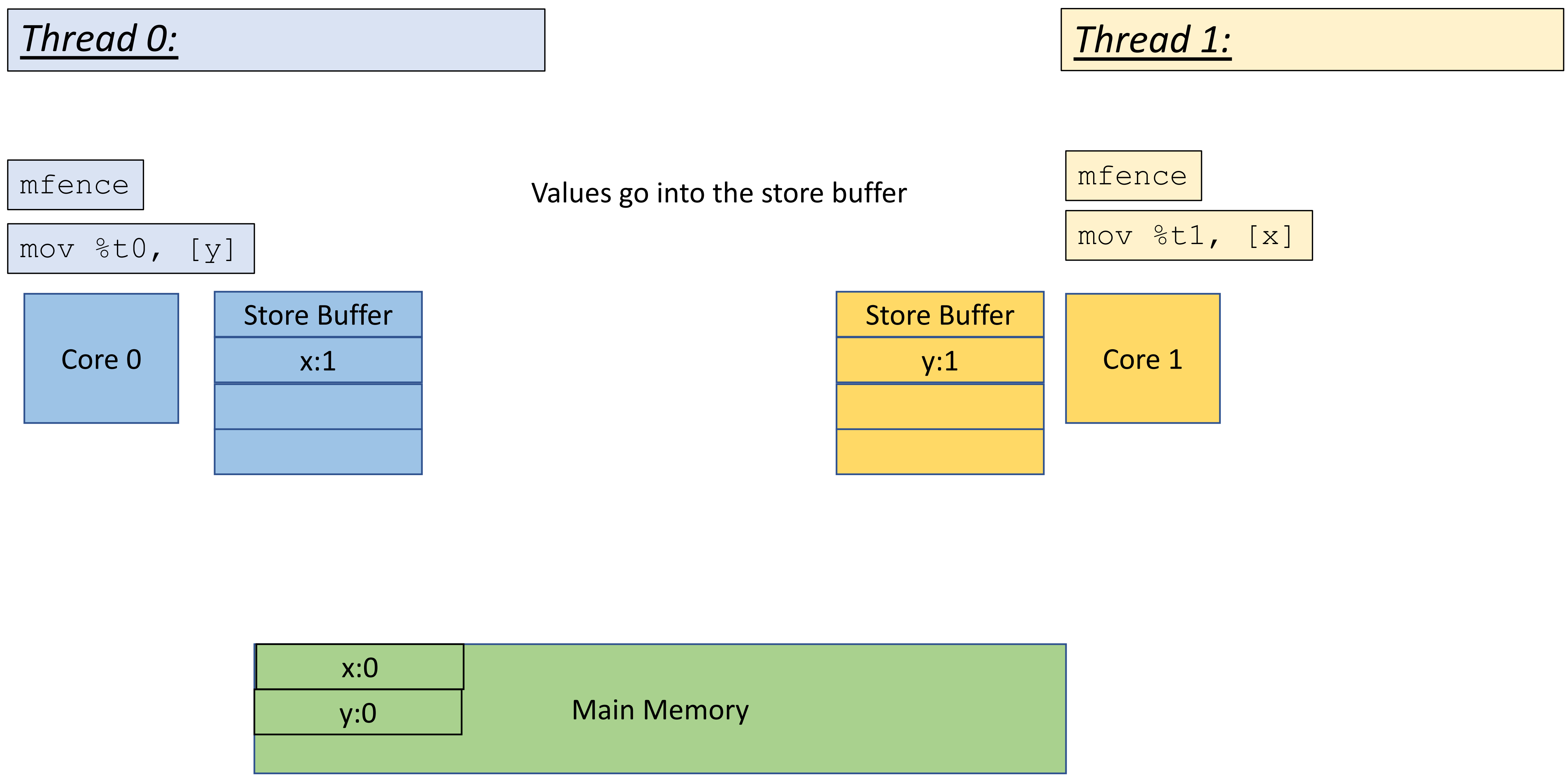
Can `t0 == t1 == 0`?

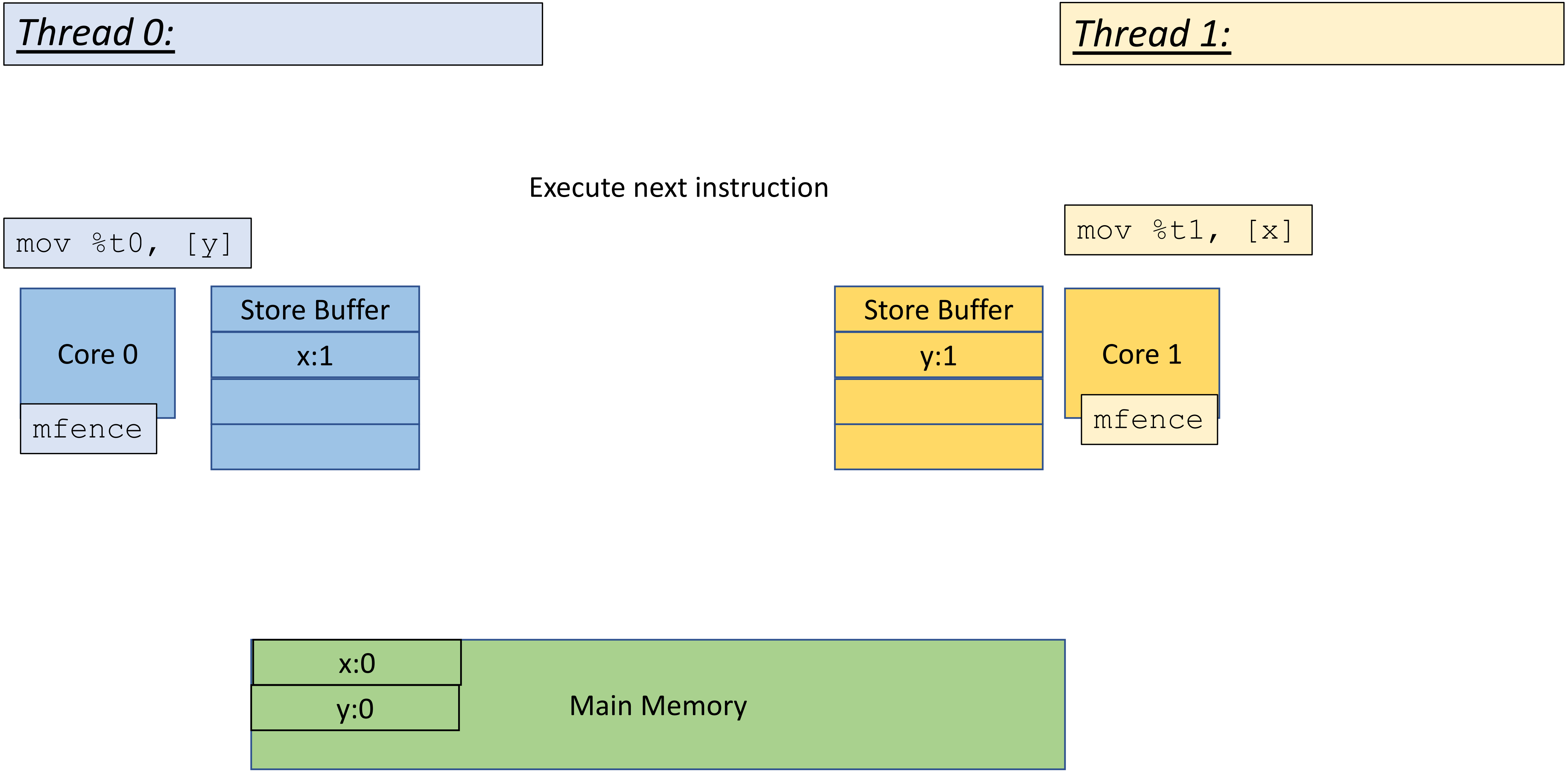
Restoring sequential consistency

- 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]
Core 0
Store Buffer
Thread 1:
mov [y], 1
mfence
mov %t1, [x]
Store Buffer
Core 1
x:0
y:0
Main Memory


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


Thread 0:
Thread 1:
mfence
mov %t0, [y]
Values go into the store buffer
mfence
mov %t1, [x]
Core 0
Store Buffer
x:1
Store Buffer
y:1
Core 1
x:0
y:0
Main Memory

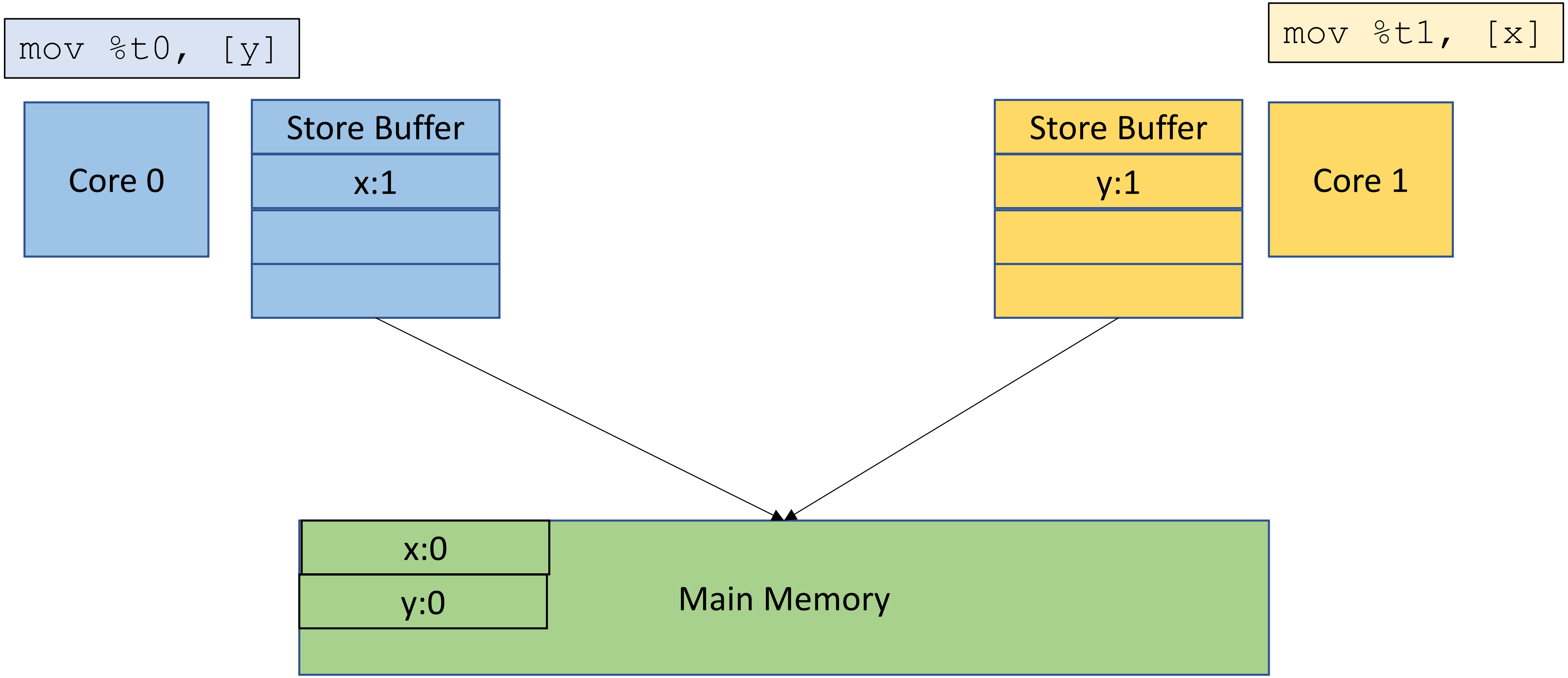


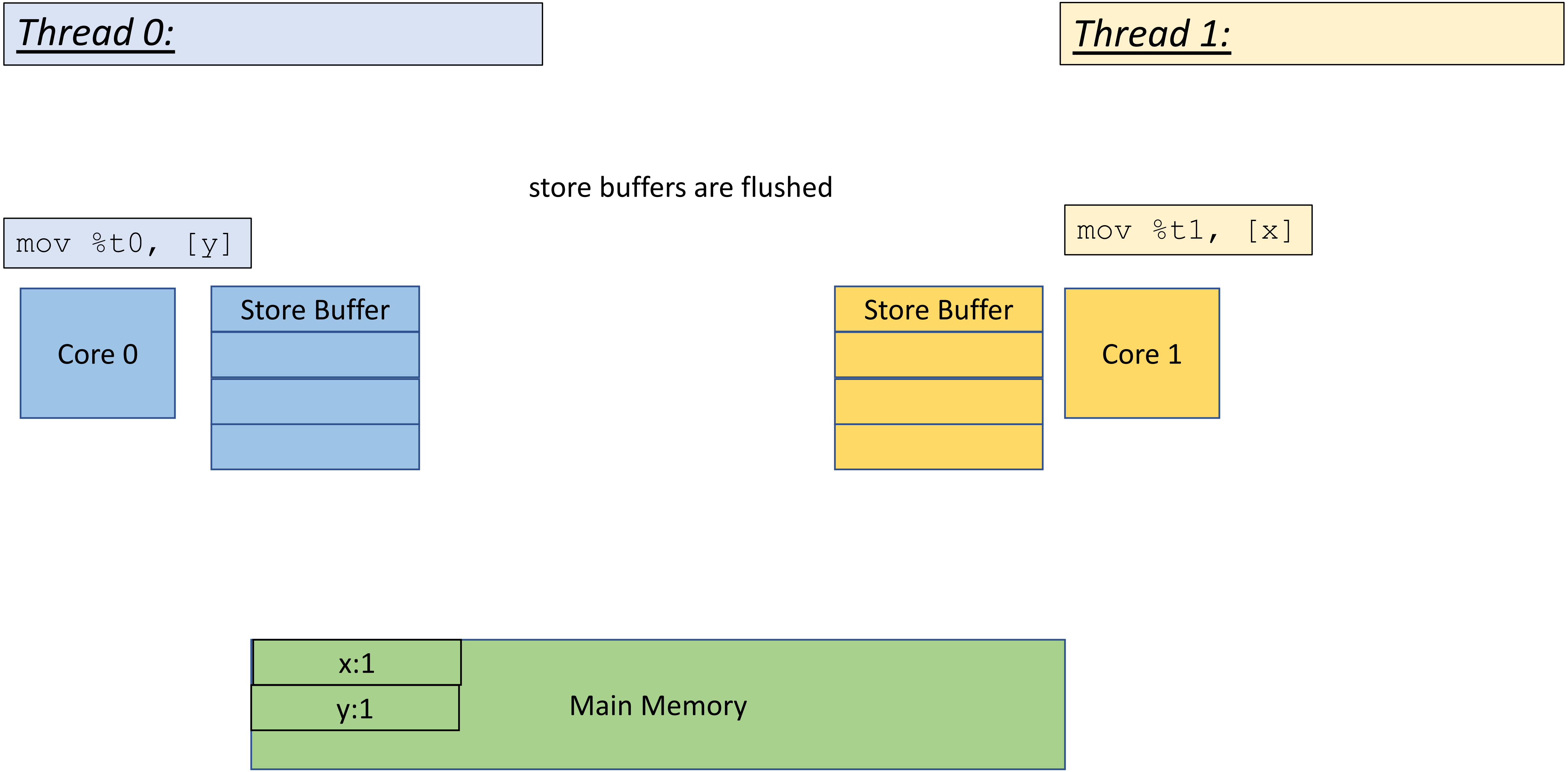
Thread 0:
Thread 1:
Execute next instruction
mov %t0, [y]
mov %t1, [x]
Core 0
mfence
Store Buffer
x:1
Store Buffer
y:1
Core 1
mfence
x:0
y:0
Main Memory

Thread 0:

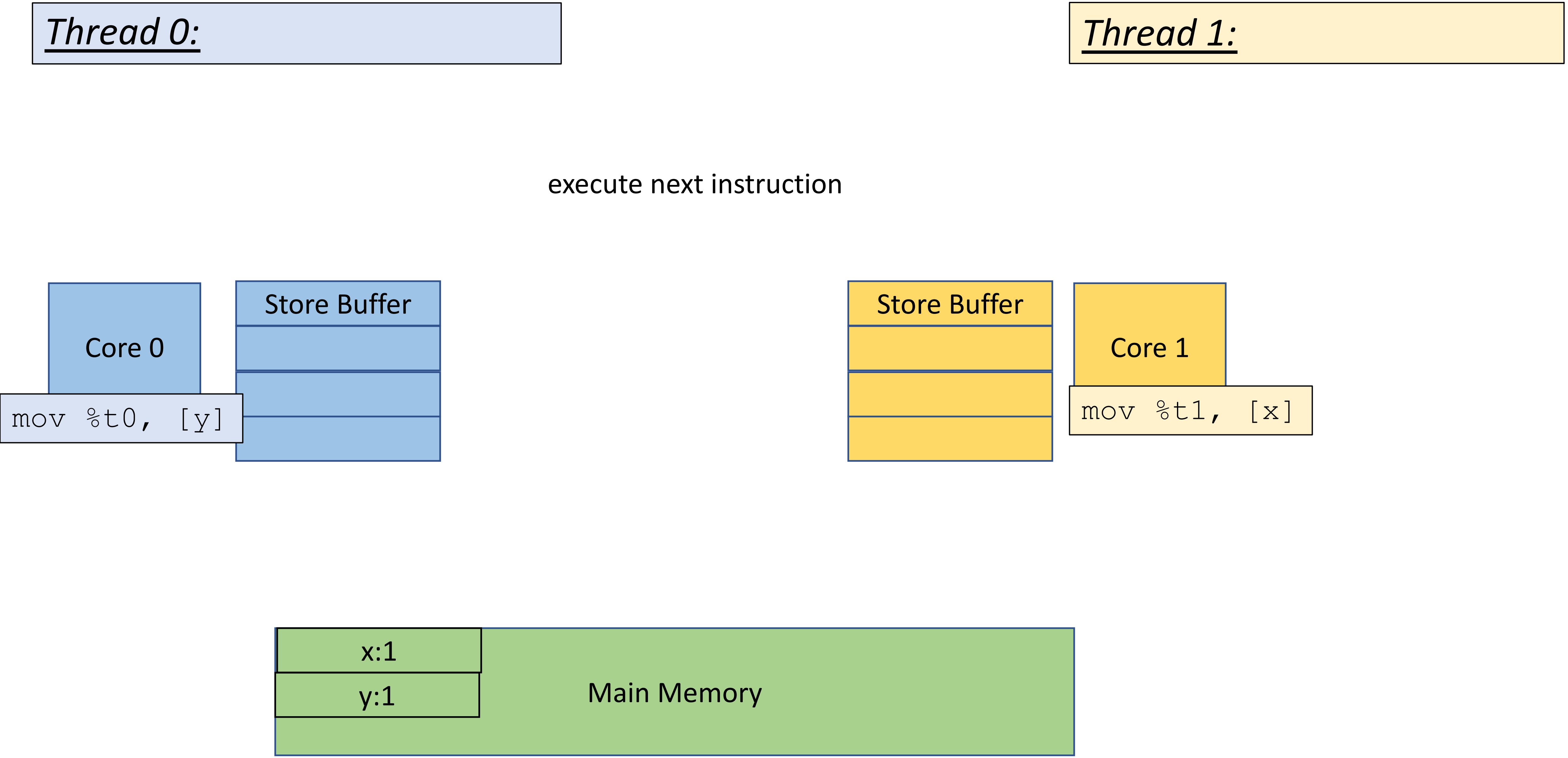
Thread 1:

store buffers are flushed





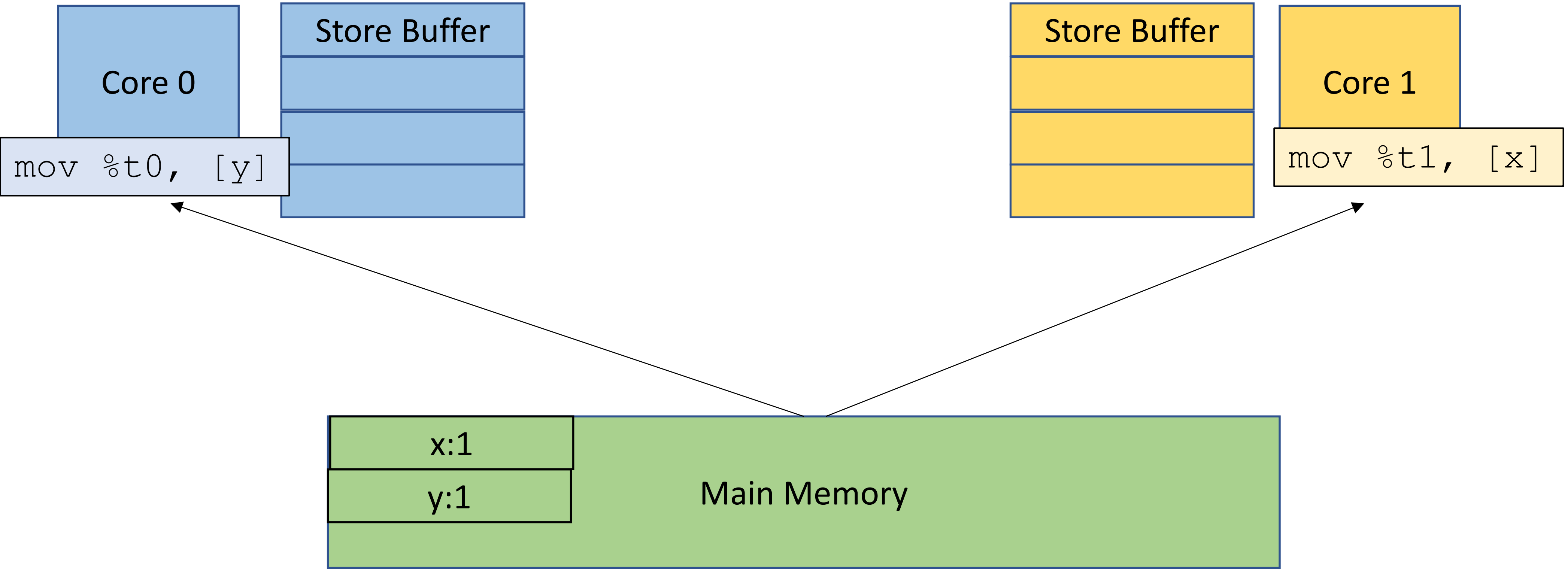
Thread 0:
Thread 1:
store buffers are flushed
mov %t0, [y]
mov %t1, [x]
Core 0
Store Buffer
Store Buffer
Core 1
x:1
y:1
Main Memory

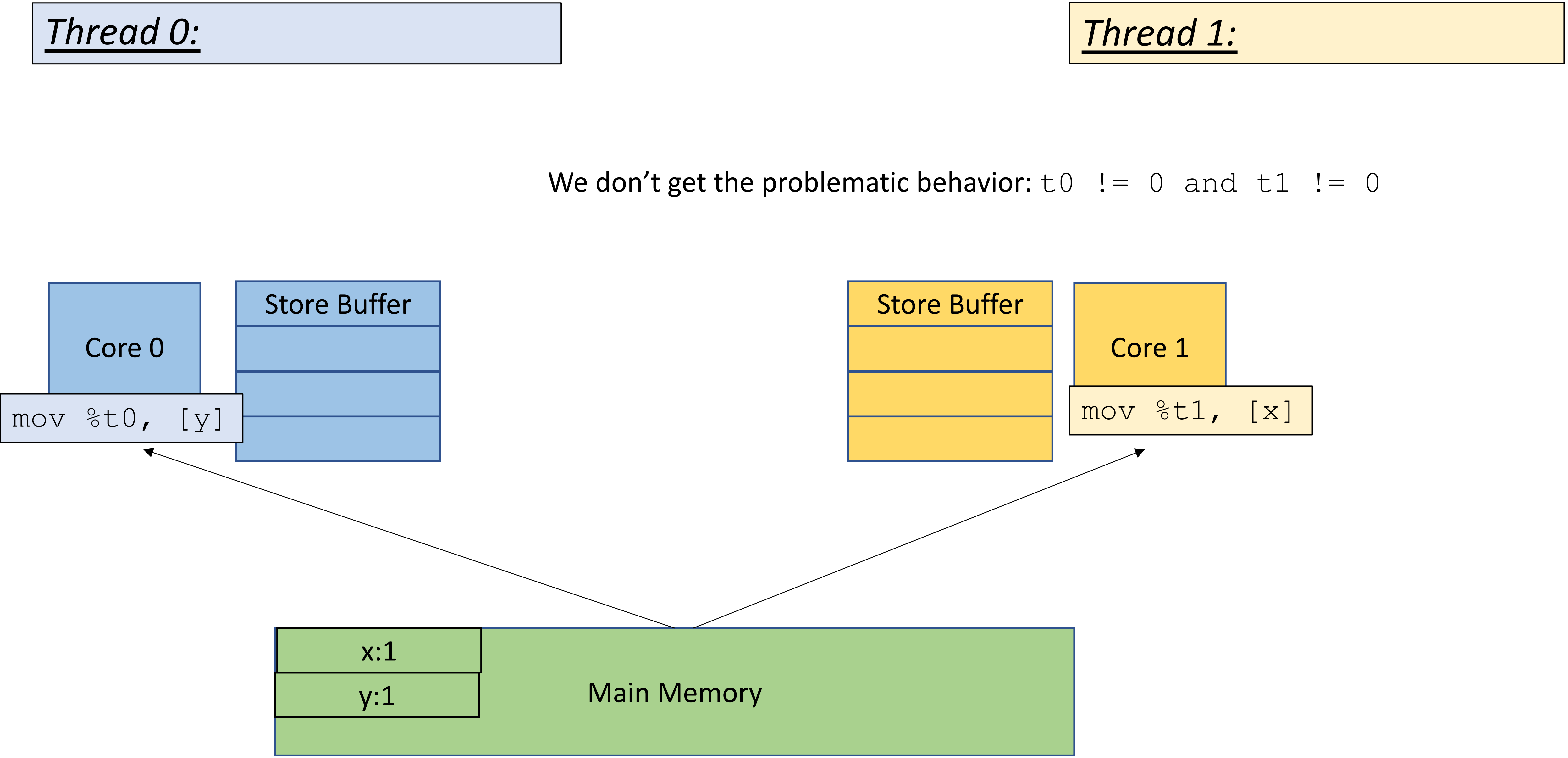

Thread 0:
Thread 1:
execute next instruction
Core 0
mov %t0, [y]
Store Buffer
Store Buffer
Core 1
mov %t1, [x]
x:1
y:1
Main Memory

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

Next example

Thread 0:

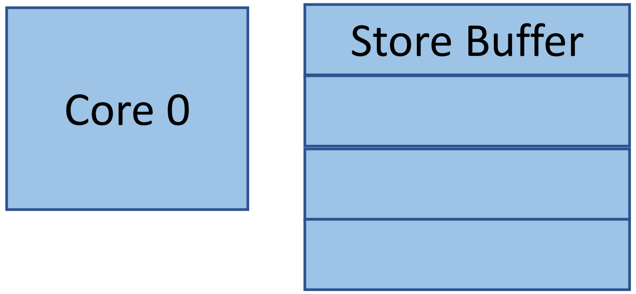
```
mov [x], 1
```

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

single thread
same address

possible outcomes:
t0 = 1
t0 = 0

Which one do you expect?

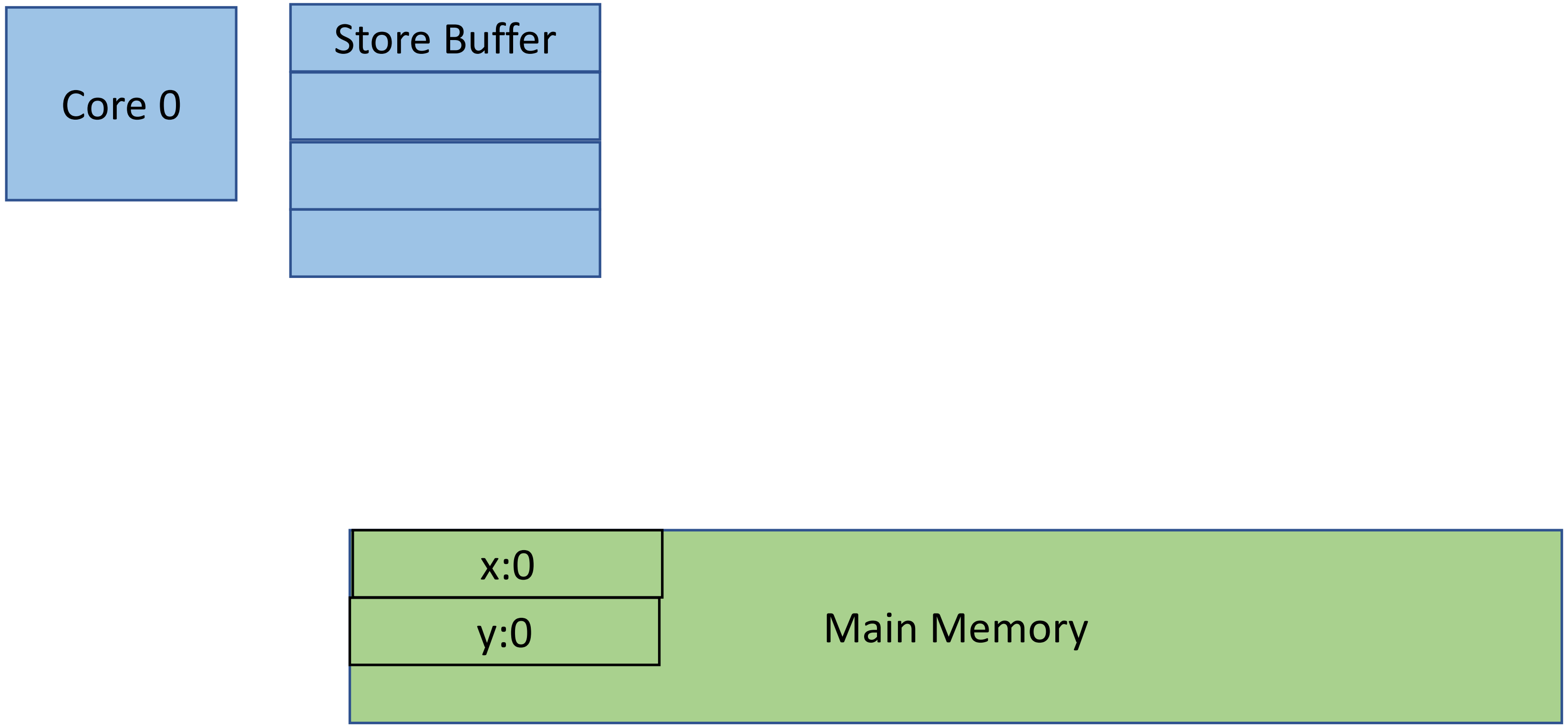


Thread 0:

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

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

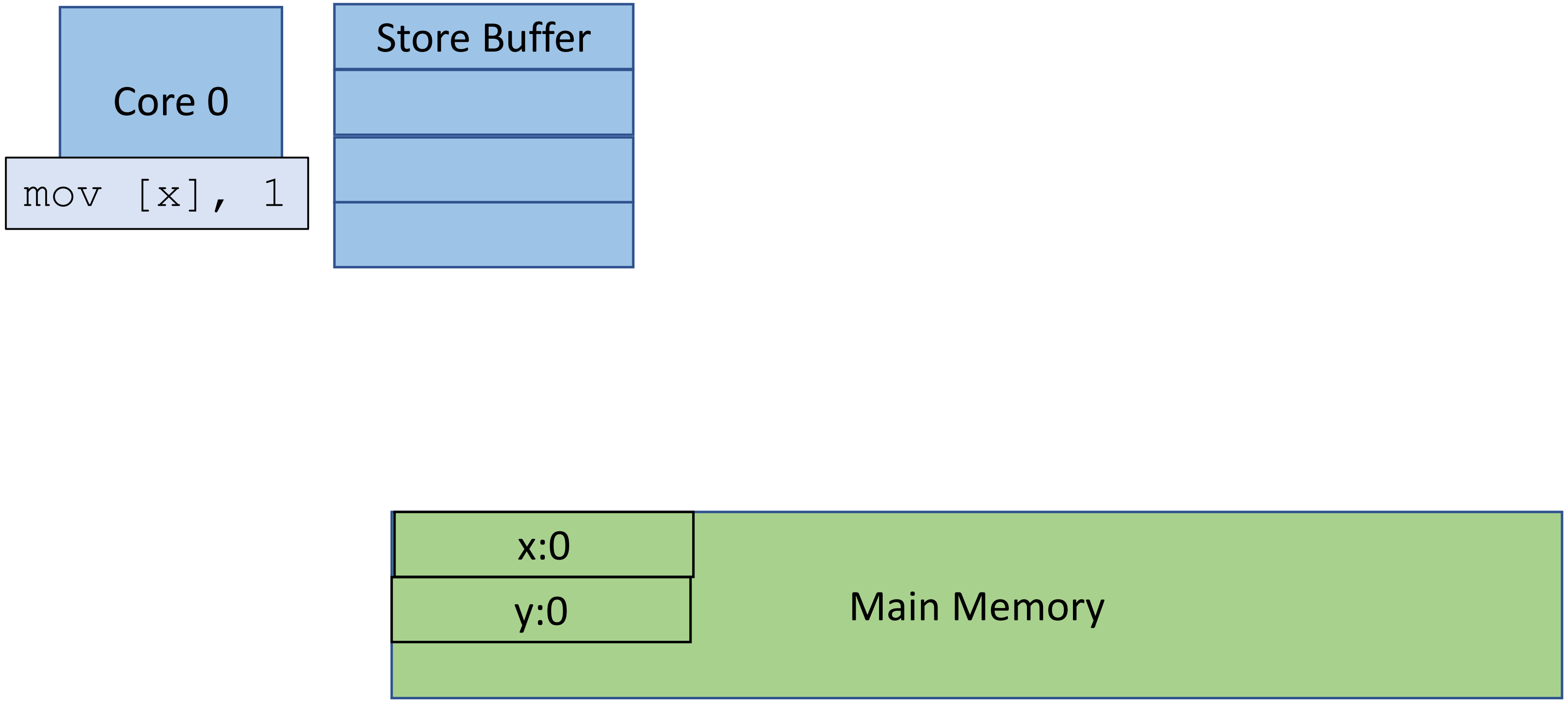
How does this execute?



Thread 0:

execute first instruction

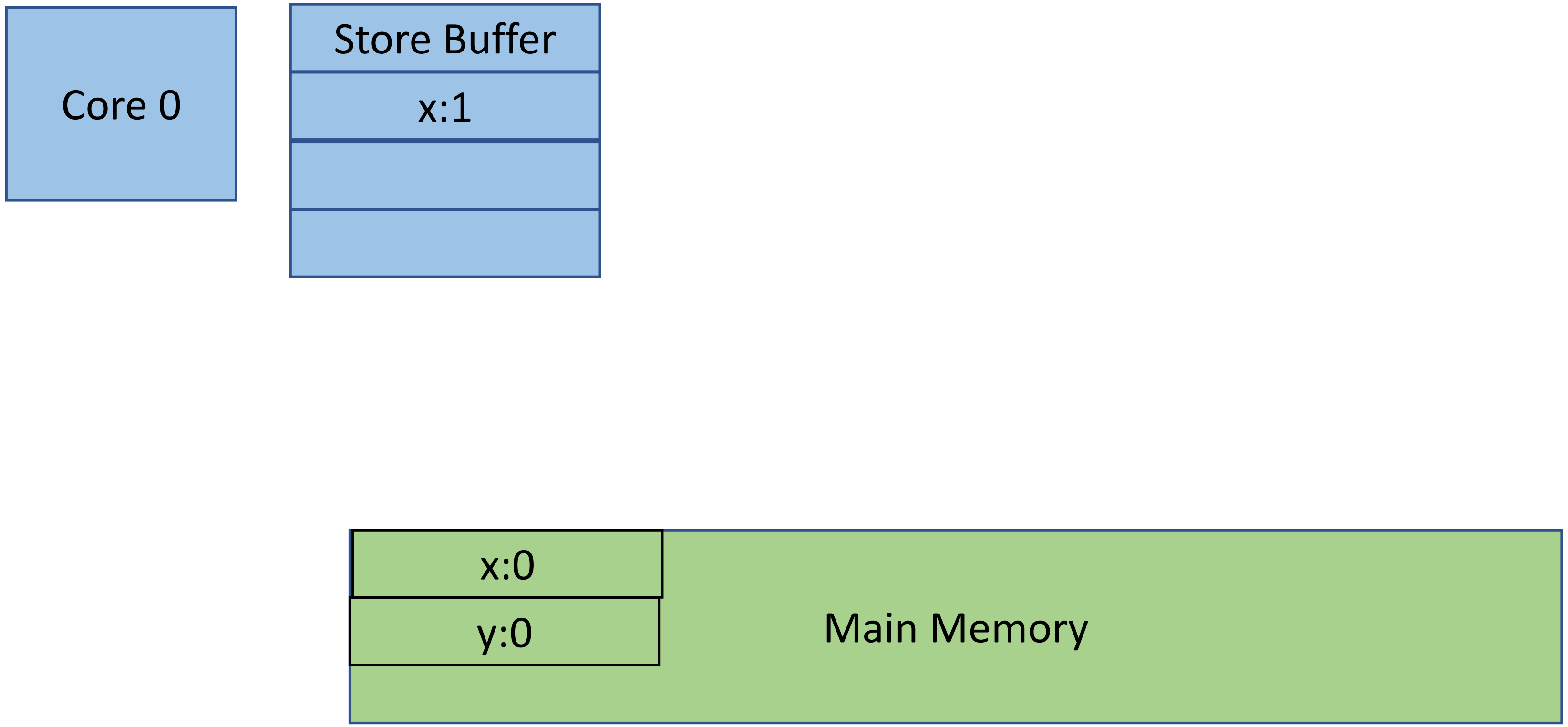
```
mov %t0, [x]
```



Thread 0:

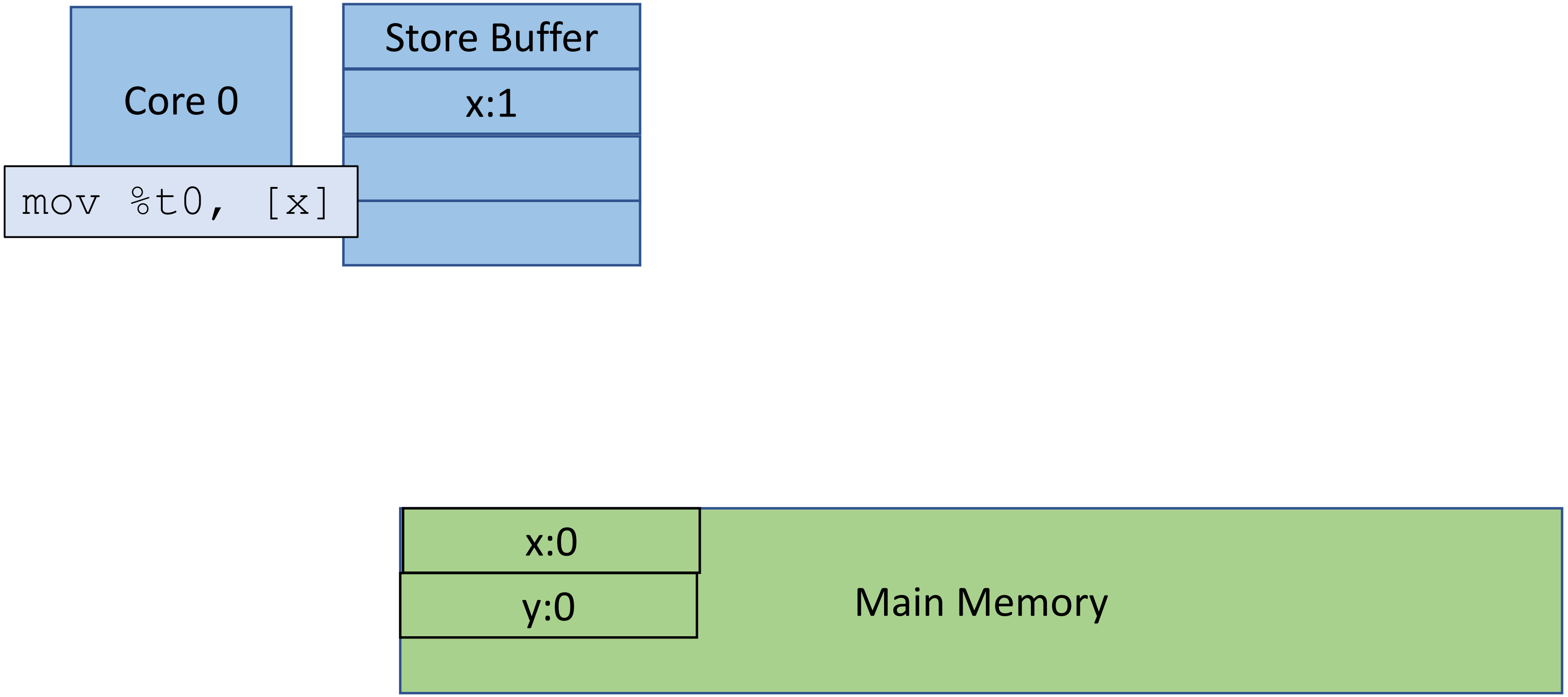
Store the value in the store buffer

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



Thread 0:

Next instruction

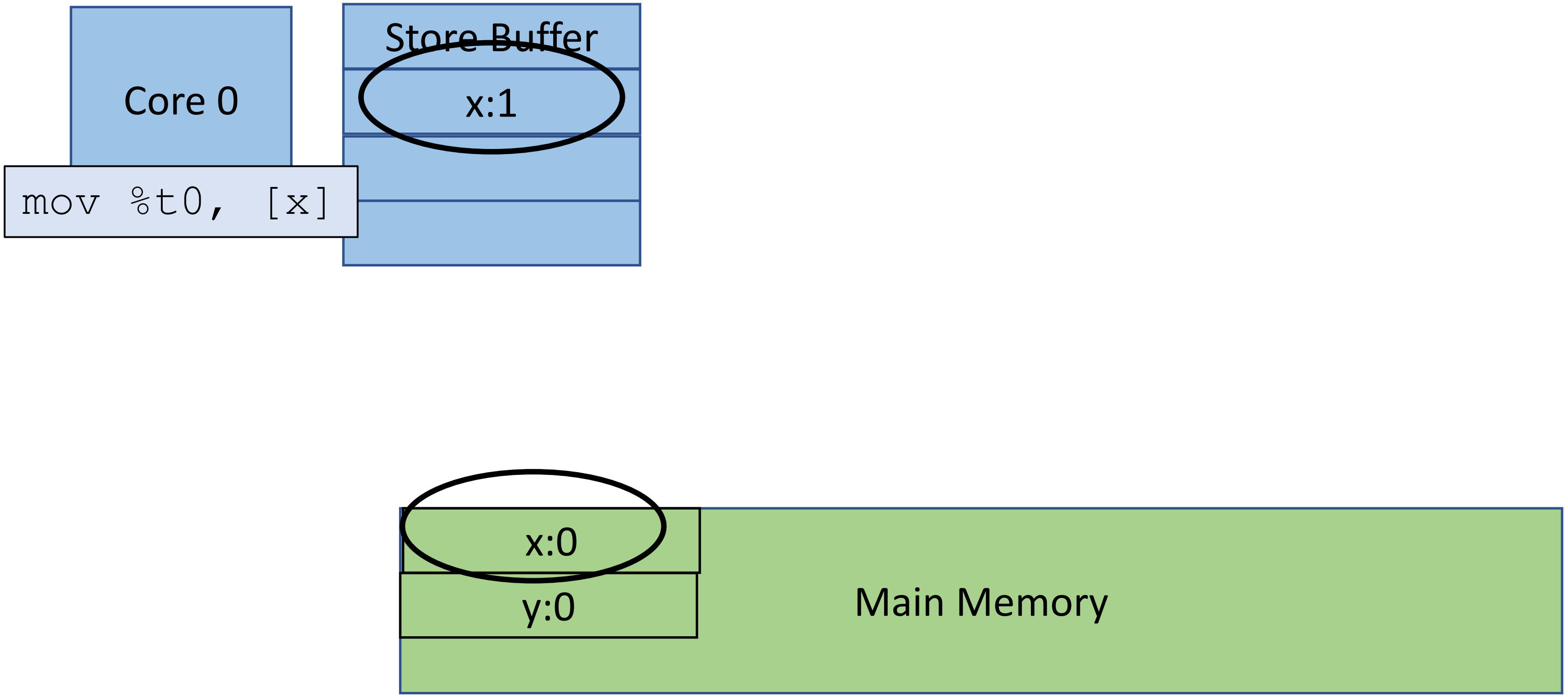


Thread 0:

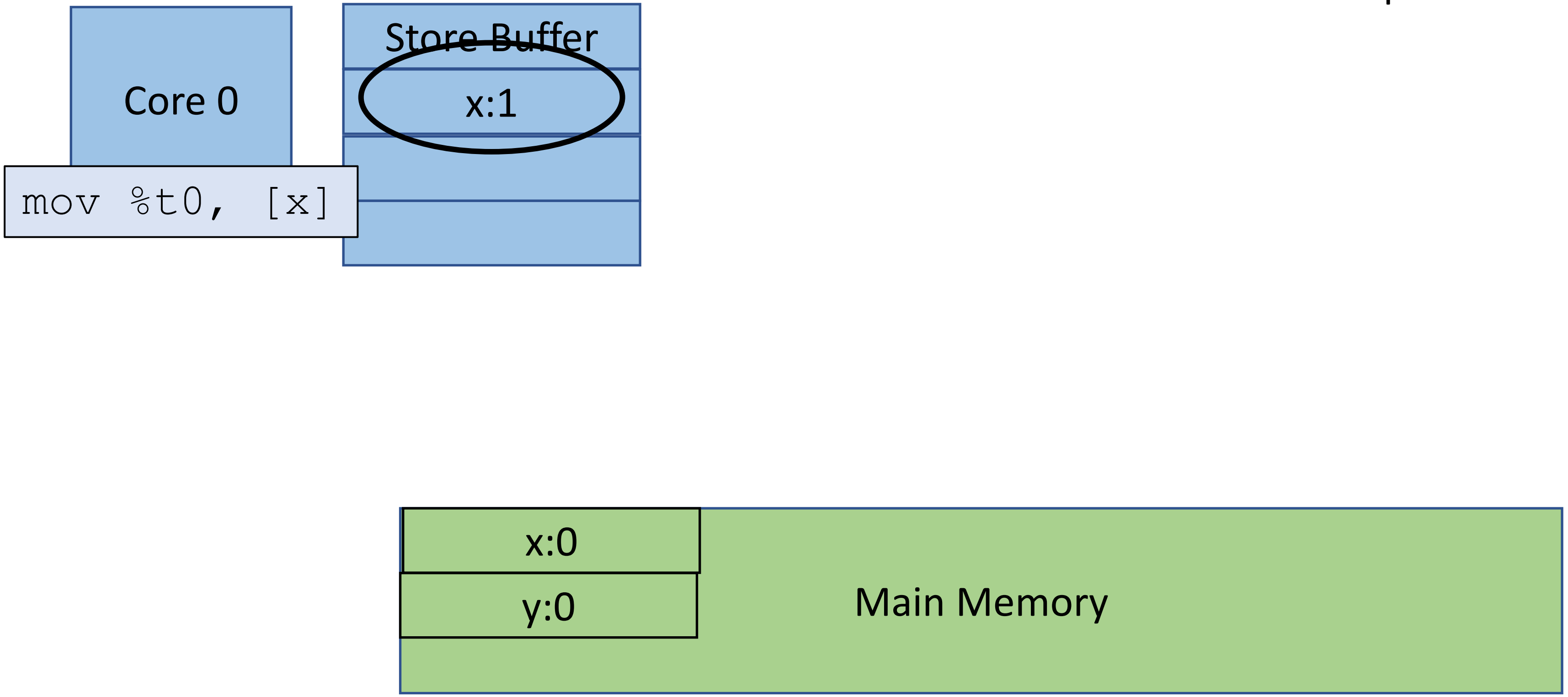
Where to load??

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};
```

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]
```

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

```
mfence
```

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

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};
```

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

memory access 1

	L	S
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 (columns), memory access 1 (rows)

	L	S
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

memory access 0 (columns) / memory access 1 (rows)

	L	S
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 (columns) / memory access 1 (rows)

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

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

memory access 0 (columns) / memory access 1 (rows)

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

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

memory access 0 (columns) / memory access 1 (rows)

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

memory access 0

memory access 1

	L	S
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++ ISA

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

→

```
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

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

or

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

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

or

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

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

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

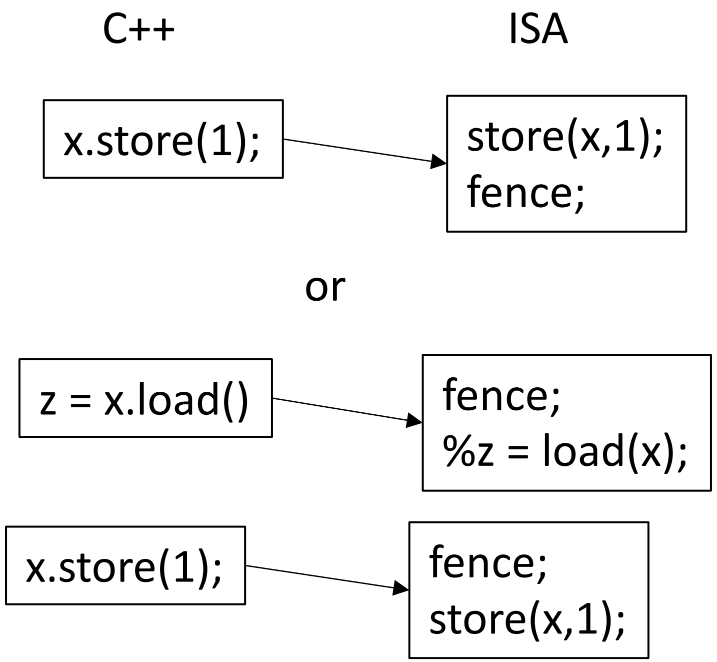
	L	S
L	NO	different address
S	NO	different address

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

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

TSO

	L	S
L	NO	Different address
S	NO	NO

PSO

	L	S
L	NO	Different address
S	NO	Different address

RMO

	L	S
L	YES	Different address
S	Different address	Different address

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

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

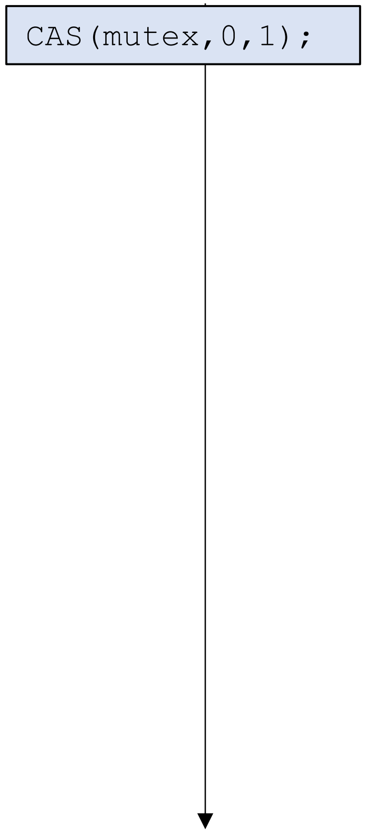
What is an execution?

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);
```



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);
```

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);
```

now yellow gets a change to go

Thread 0:

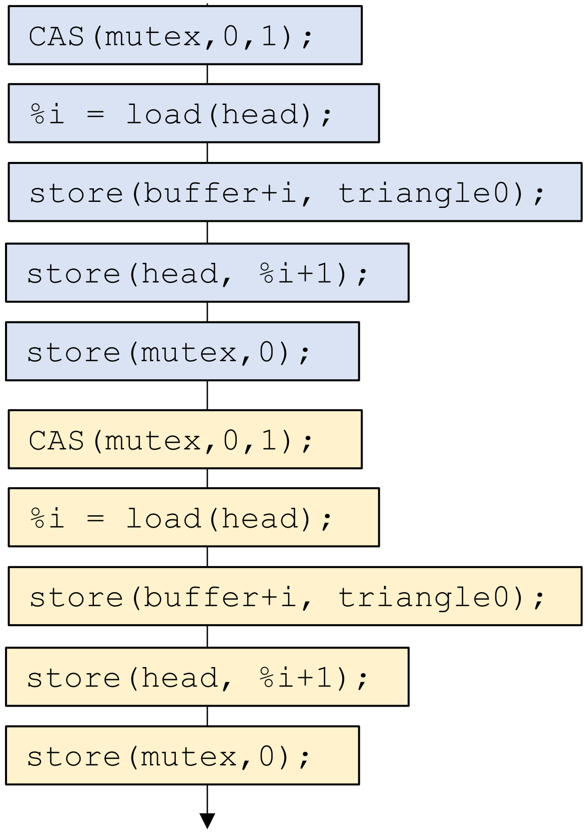
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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

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

what can happen in a PSO memory model?

	L	S
L	NO	Different address
S	NO	Different address

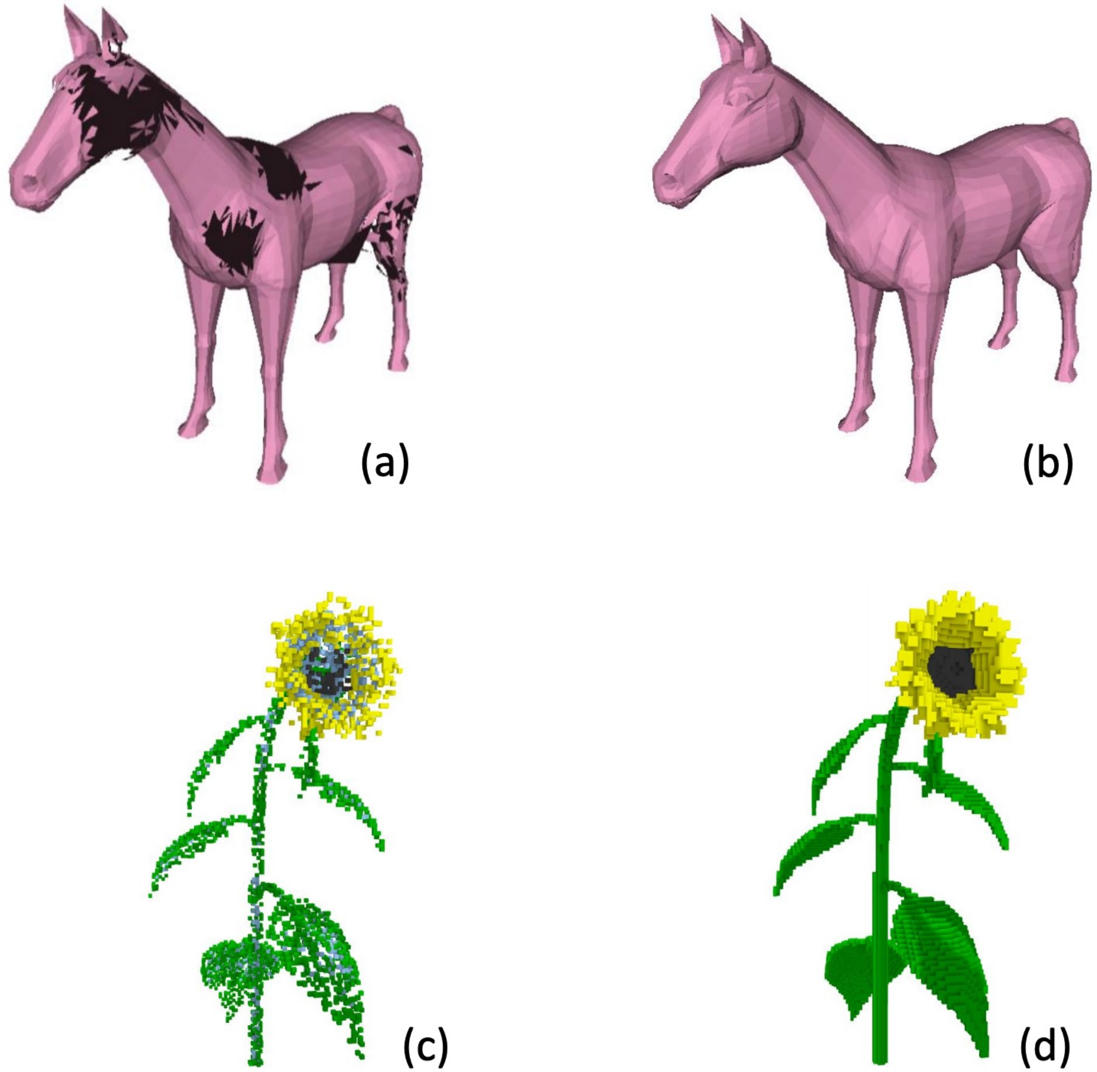
```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(head, %i+1);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

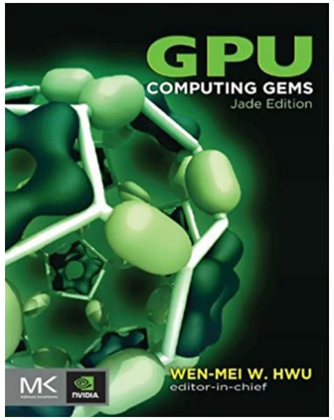
What just happened if this store moves?

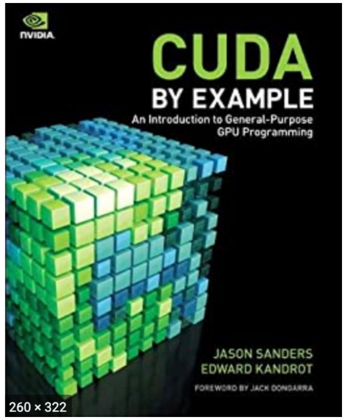
Nvidia in 2015

- 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

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

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

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

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

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

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)