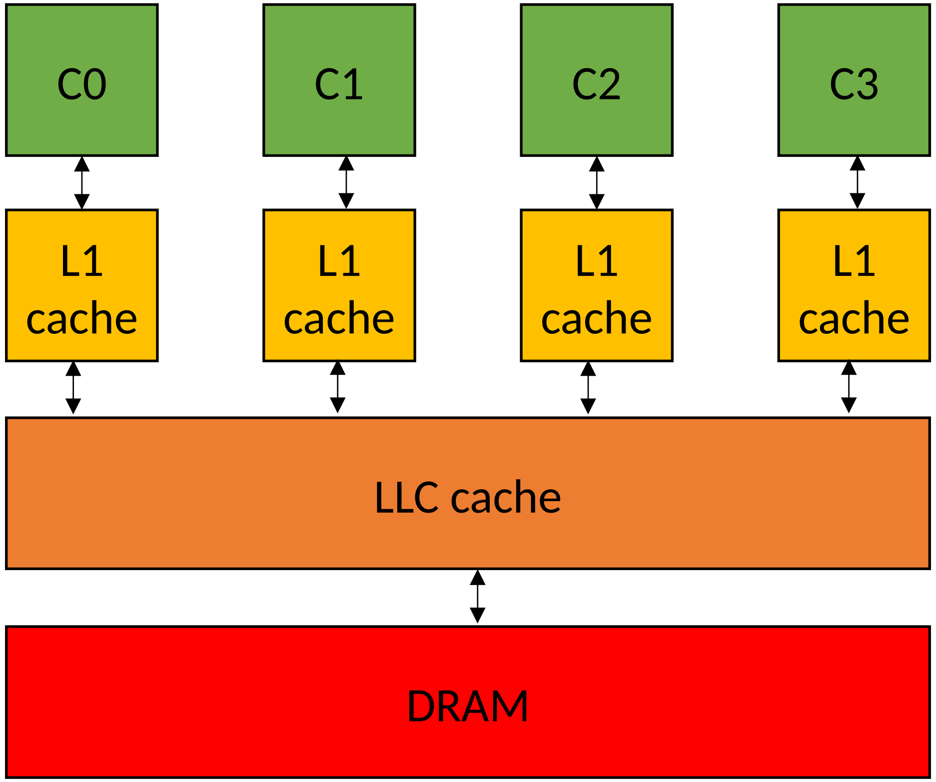
CSE113: Parallel Programming

- **Topic**: Memory Hierarchy and C++ threads
    - Caches
    - Cache lines
    - Coherence
    - C++ threads
    - False Sharing



Announcements

- Homework due on Oct 15
  - Three free late days
  - Plenty of office hours remaining to get help
  - Work on your design doc before asking for help

Announcements

- Reminder on quiz and design doc: Not heavily graded, but low effort responses are liable to lose points
- (Hopefully) Last lecture of Module 1
    - Moving into Module 2: mutual exclusion next time!
- Should be able to do part 1 and part 2 of homework
    - Hopefully part 3 by today, maybe next time though.

Quiz

The following statement in a language like C or Java would be compiled to how many instructions in low-level code?

z = x + x + x + x;

- ○ 0
- ○ 1
- ○ 2
- ○ 4

Quiz

How many levels of caches does a typical x86 system have?

- ○ 1
- ○ 2
- ○ 3
- ○ 4

Quiz

Write a few reasons why it may be difficult to reason about program performance when using a high-level language like Python

Quiz

Using your best guess, how much faster do you think a program written in C/Java is than a program written in Python? Give a few reasons explaining your guess. Feel free to run a simple experiment and see what happens!

Quiz

How many cores does the computer you're working on have:

1		0 %
2	4 respondents	7 %
4	20 respondents	33 %
8	34 respondents	56 %
At least 16	8 respondents	13 %

Quiz

Modern-day compilers and runtimes will automatically make your code parallel. Because of this, most programmers do not need to think about parallelism when writing programs.

Justify your answer above using a few sentences

Review

Instruction-level Parallelism (ILP)

- Parallelism from a single stream of instructions.
  - Output of program must match exactly a sequential execution!

- Widely applicable:
  - most mainstream programming languages are sequential
  - most deployed hardware has components to execute ILP

- Done by a combination of programmer, compiler, and hardware

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in parallel if they are independent

```
x = z + w;
a = b + c;
```

Two instructions are independent if the operand registers are disjoint from the result registers

(assume all letter variables are registers)

instructions that are not independent cannot be executed in parallel

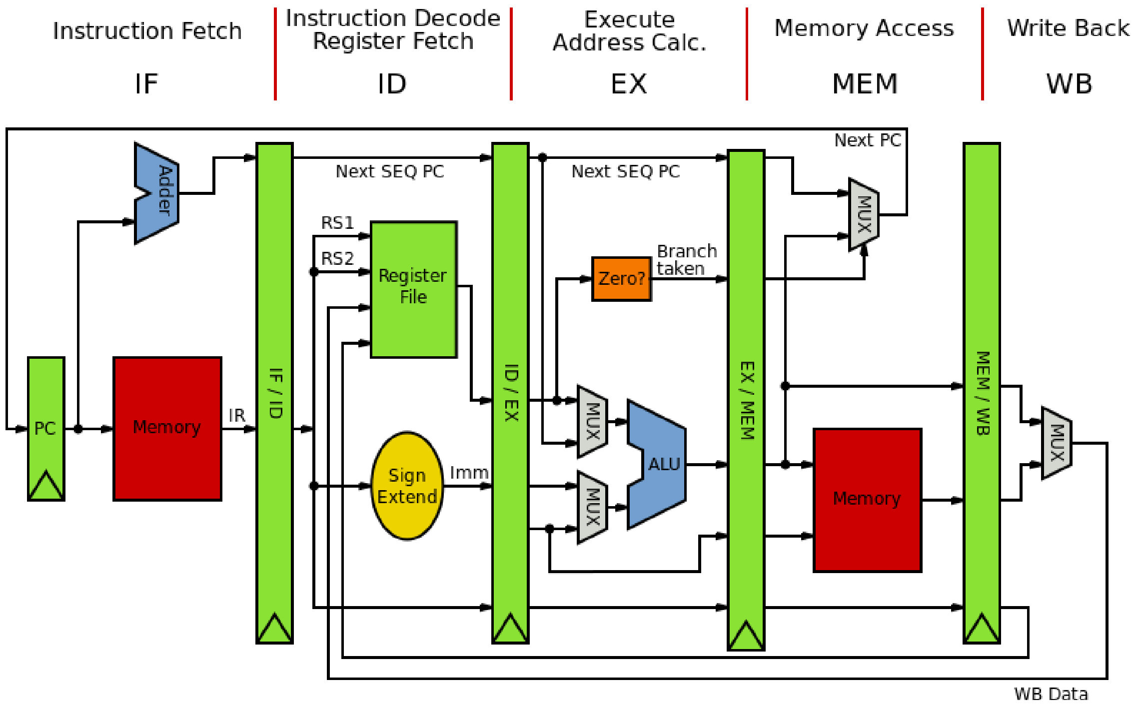
```
x = z + w;
a = b + x;
```

Many times, dependencies can be easily tracked in the compiler:

How can hardware execute ILP?

- Pipeline parallelism

- Abstract mental model:
  - N-stage pipeline
  - N instructions can be in-flight
  - Dependencies stall pipeline



MIPS pipeline image from:
https://commons.wikimedia.org/wiki/Pipeline_(computer_hardware)

How can hardware execute ILP?

- Executing multiple instructions at once:

- Superscalar architecture:
  - Several sequential operations are issued in parallel
  - hardware detects dependencies

issue-width is maximum number of instructions that can be issued in parallel

```
instr0;
instr1;
instr2;
```

if instr0 and instr1 are independent, they will be issued in parallel

What does this look like in the real world?

- Intel Haswell (2013):
  - Issue width of 4
  - 14-19 stage pipeline
  - OoO execution

- Intel Nehalem (2008)
  - 20-24 stage pipeline
  - Issue width of 2-4
  - OoO execution

- ARM
  - V7 has 3 stage pipeline; Cortex V8 has 13
  - Cortex V8 has issue width of 2
  - OoO execution

- RISC-V
  - Ariane and Rocket are In-Order
  - 3-6 stage pipelines
  - some super scaler implementations (BOOM)

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
      SEQ(i,1);
      SEQ(i,2);
      ...
      SEQ(i,N); // end iteration for i
      SEQ(i+1,1);
      SEQ(i+1,2);
      ...
      SEQ(i+1, N); // end iteration for i + 1
}
```

Let `SEQ(i,j)` be the jth instruction of `SEQ(i)`.

Let each instruction chain have N instructions

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

```
for (int i = 0; i < SIZE; i+=2) {
    SEQ(i,1);
    SEQ(i+1,1);
    SEQ(i,2);
    SEQ(i+1,2);
    ...
    SEQ(i,N);
    SEQ(i+1, N);
}
```

They can be interleaved

two instructions can be pipelined, or executed on a superscalar processor

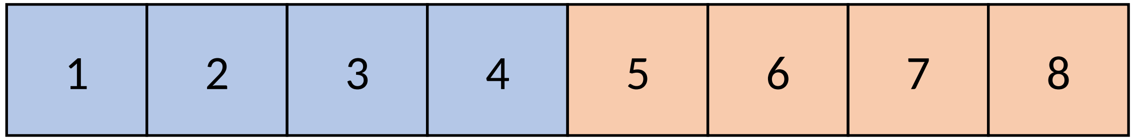
Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:

1	2	3	4	5	6	7	8

Loop Unrolling for Reduction Loops

- chunk array in equal sized partitions and do local reductions
- Consider size 2:



Loop Unrolling for Reduction Loops

- Simple implementation:

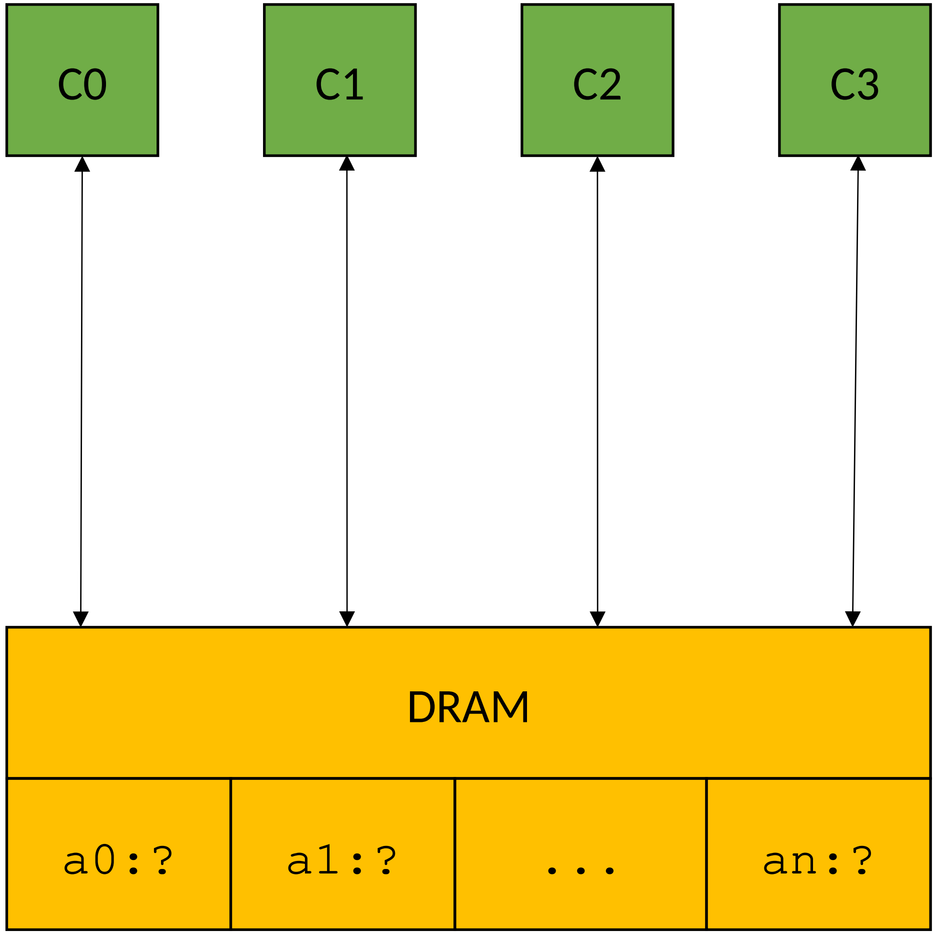
```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2)+i]);
}

a[0] = REDUCE(a[0], a[SIZE/2])
```

Memory hierarchy overview

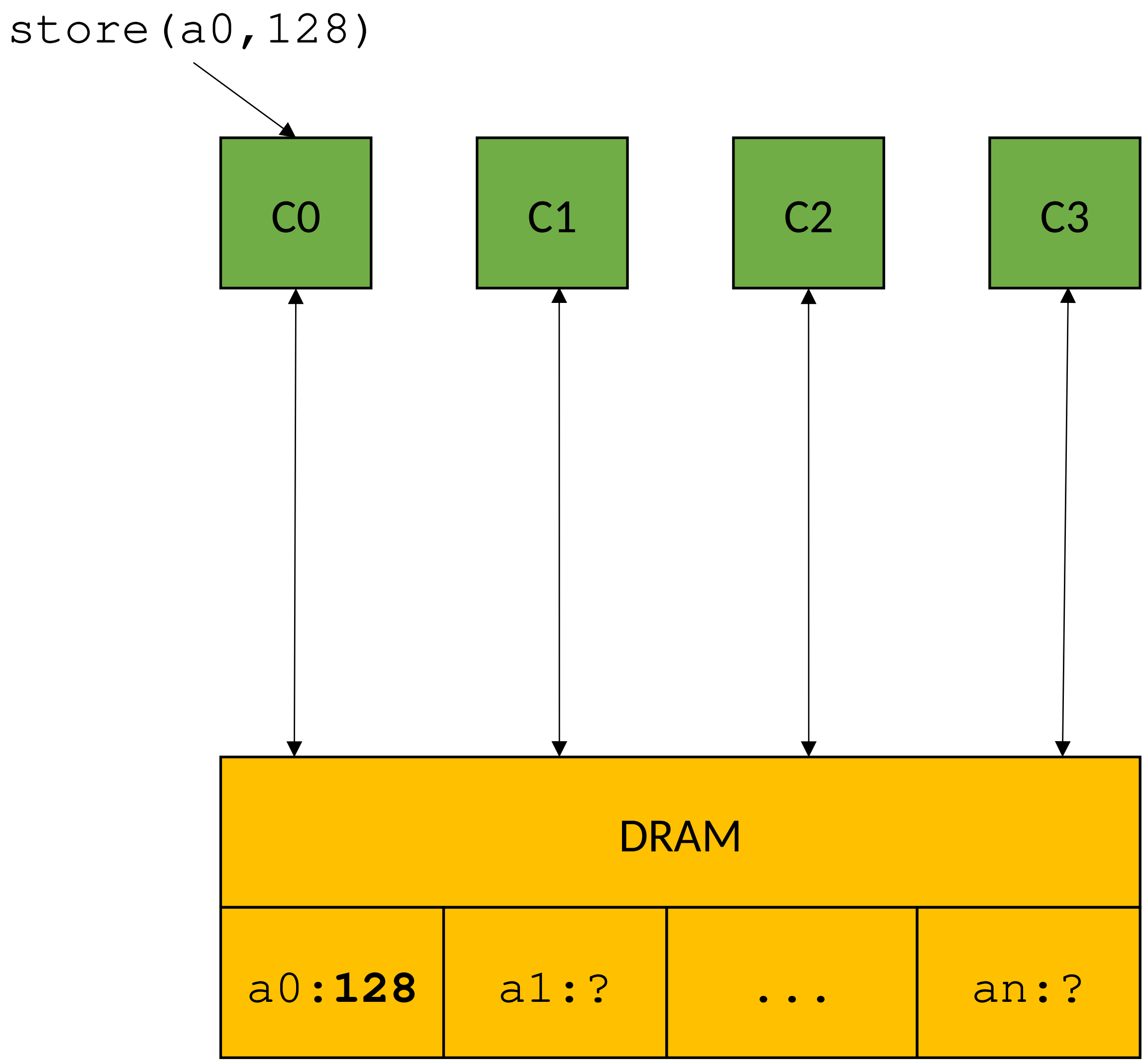
- How can threads communicate?

Main memory

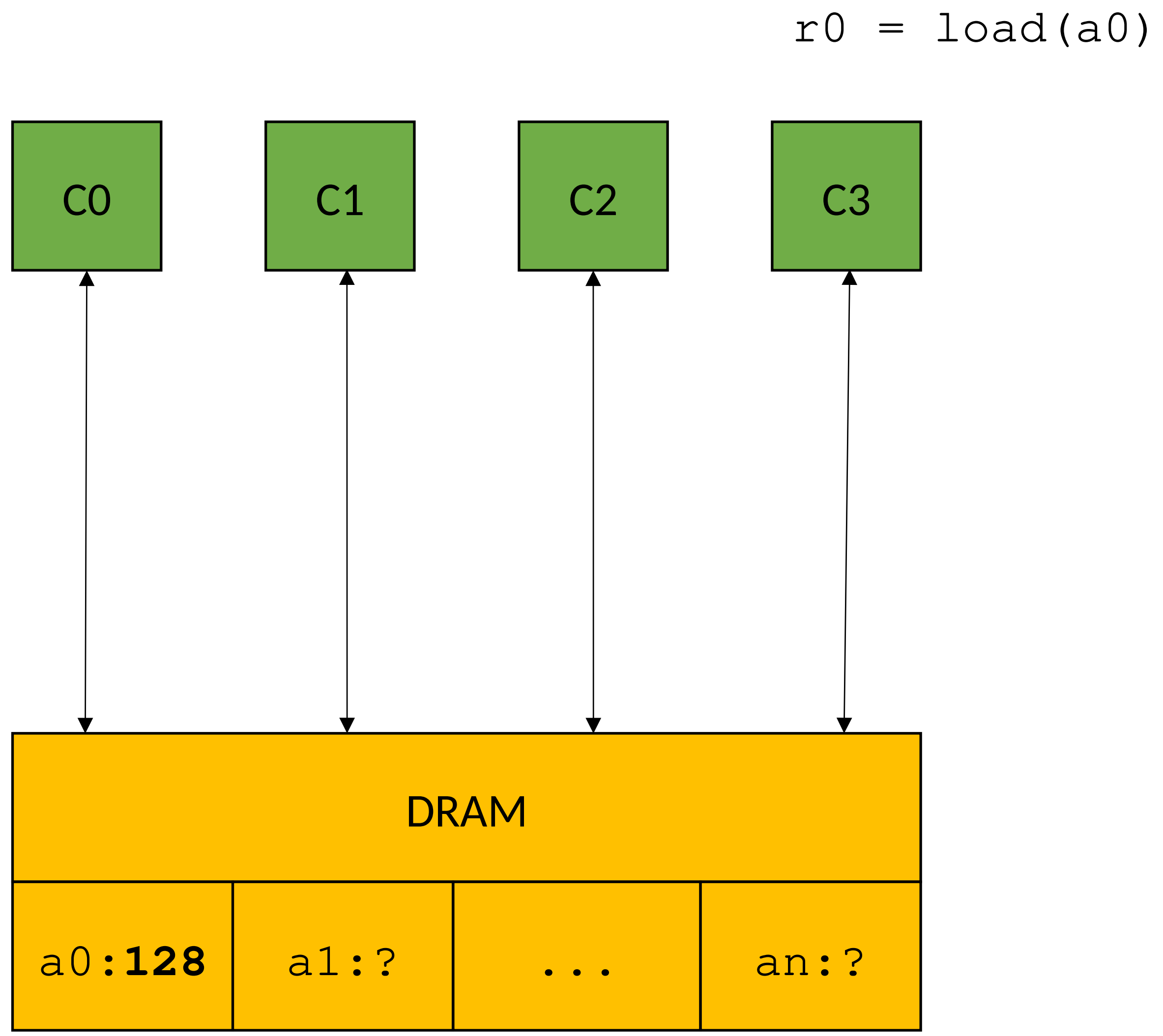
`store(a0,128)`



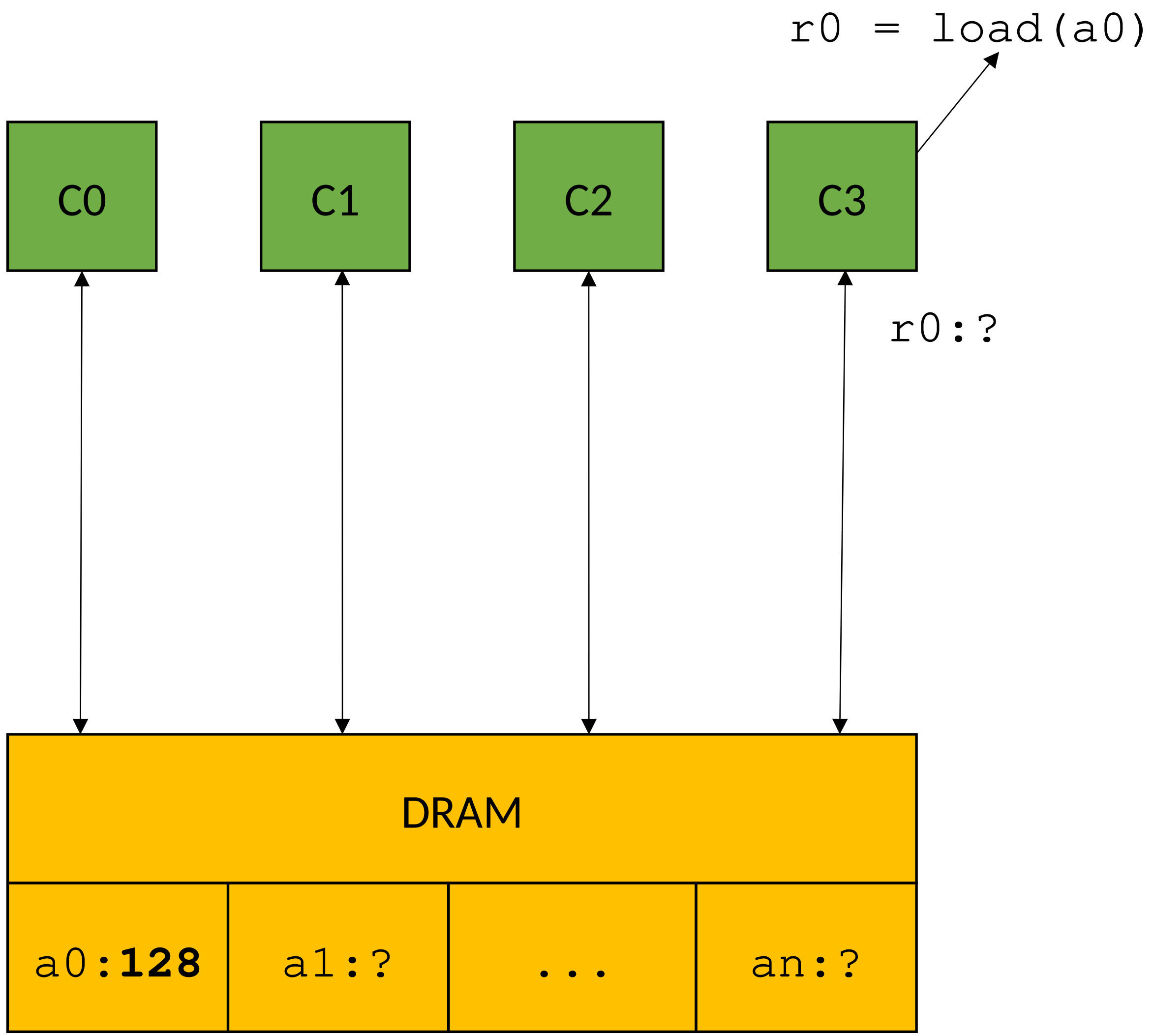
Main memory



Main memory

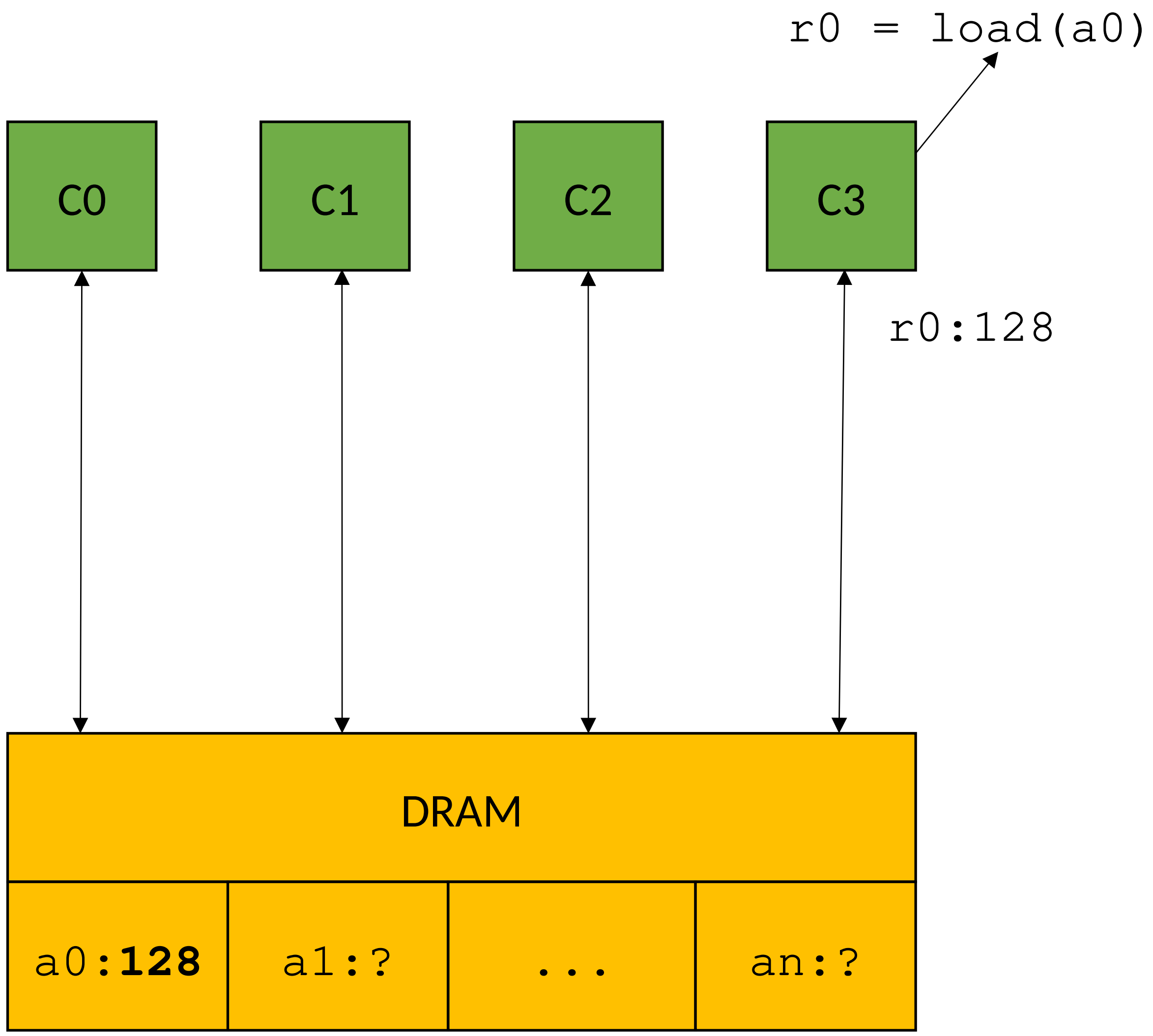


Main memory



Main memory

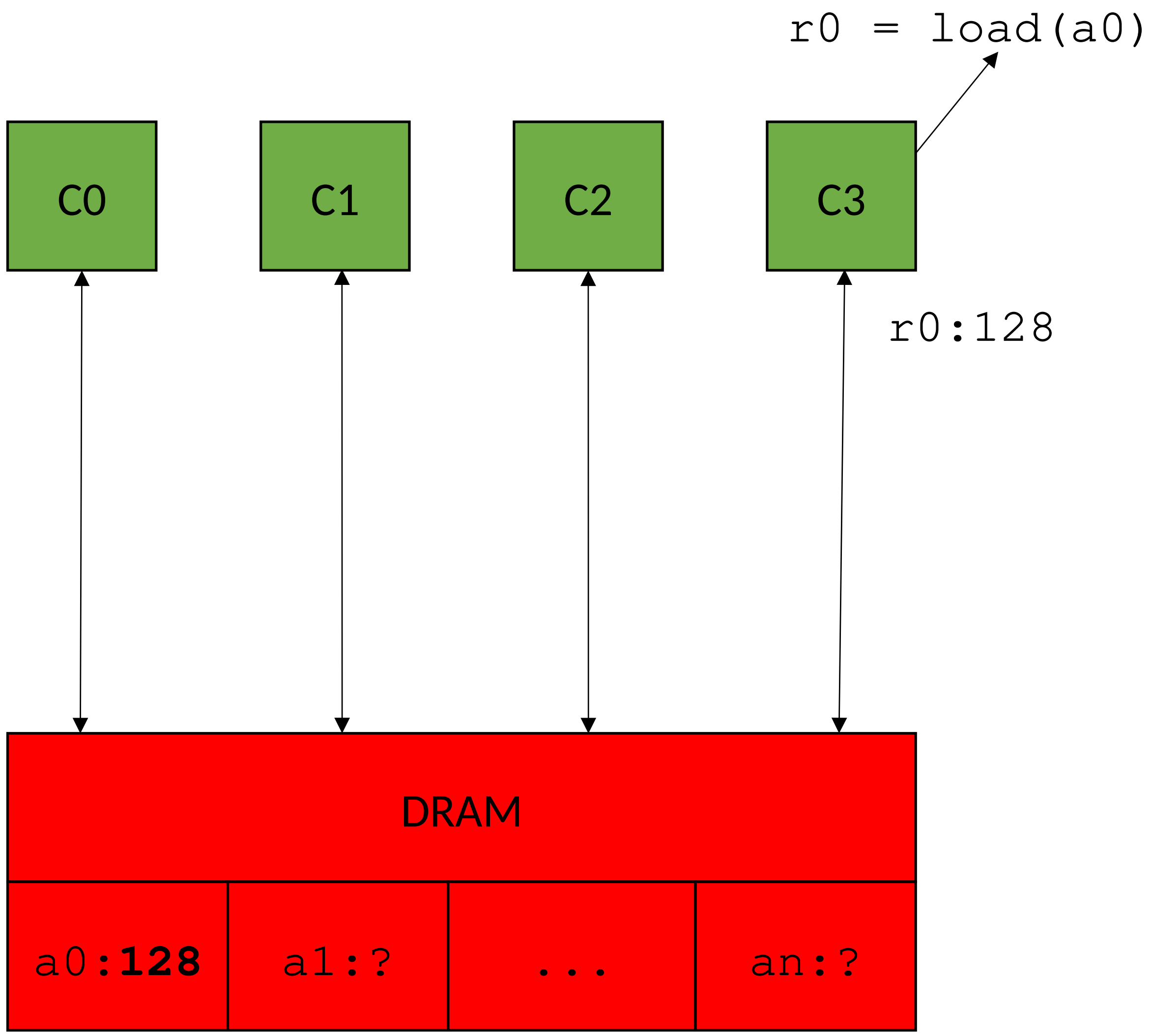
Problem solved!
Threads can communicate!



Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

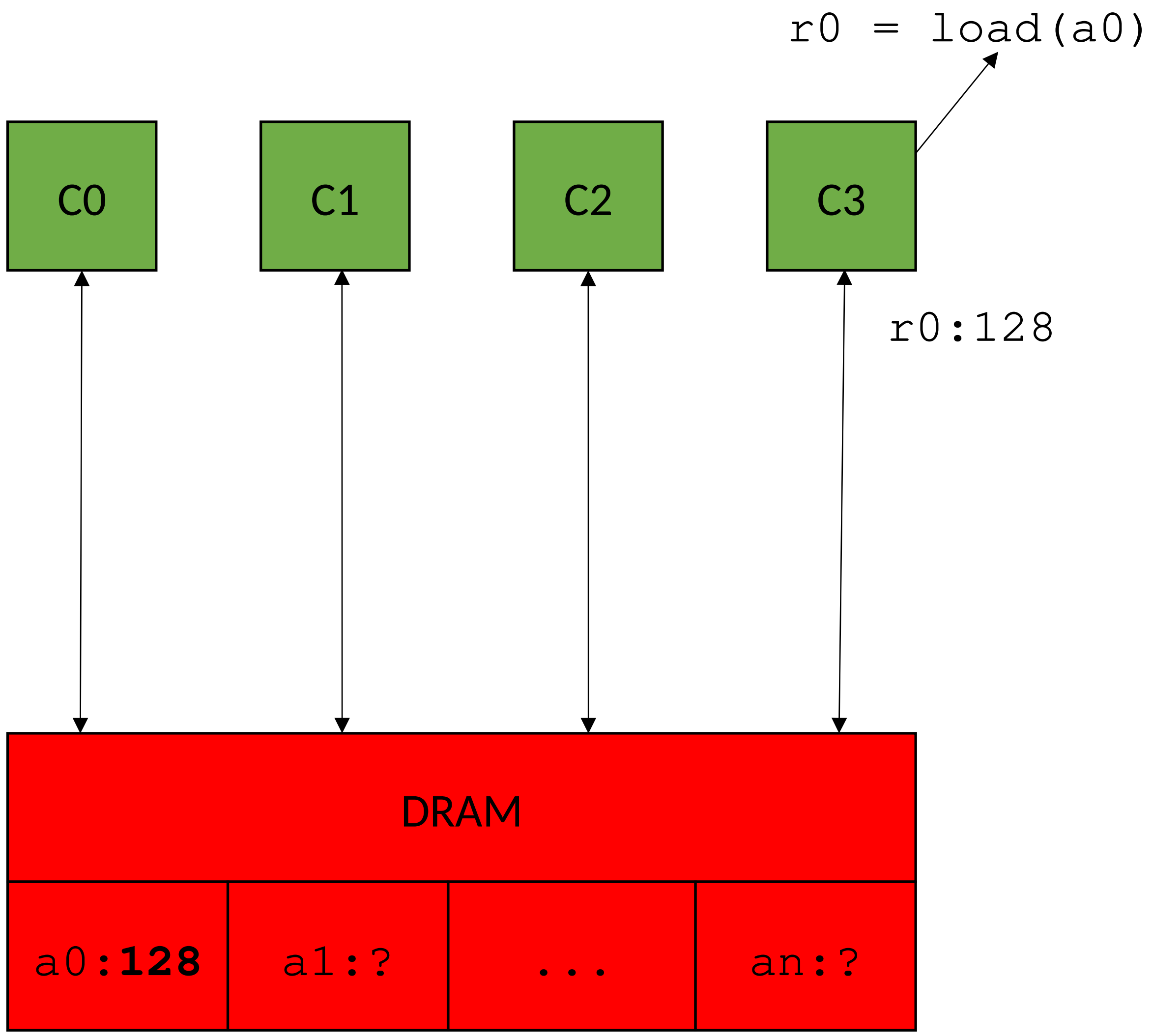


Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

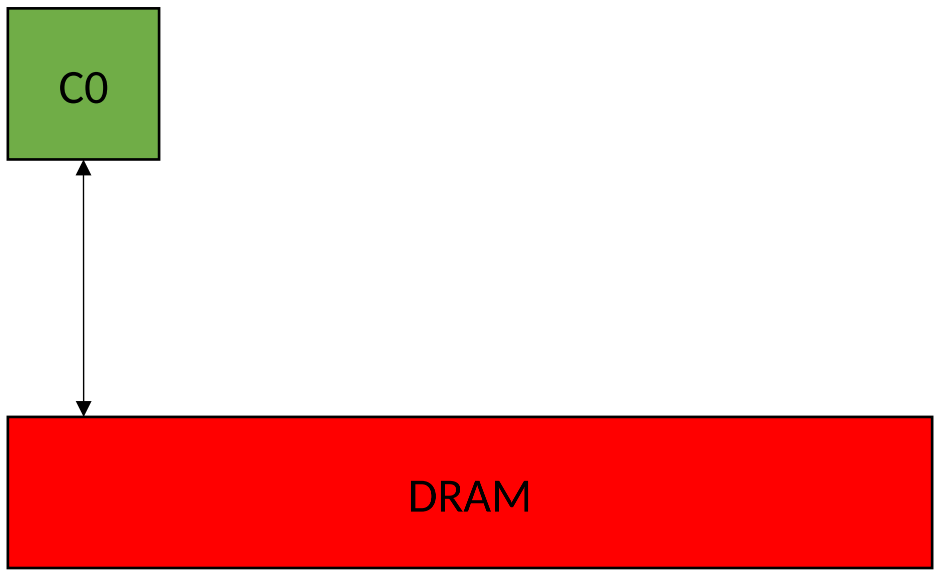
Bad for parallelism, but
also really bad for sequential
code (which we optimized for
decades!)



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

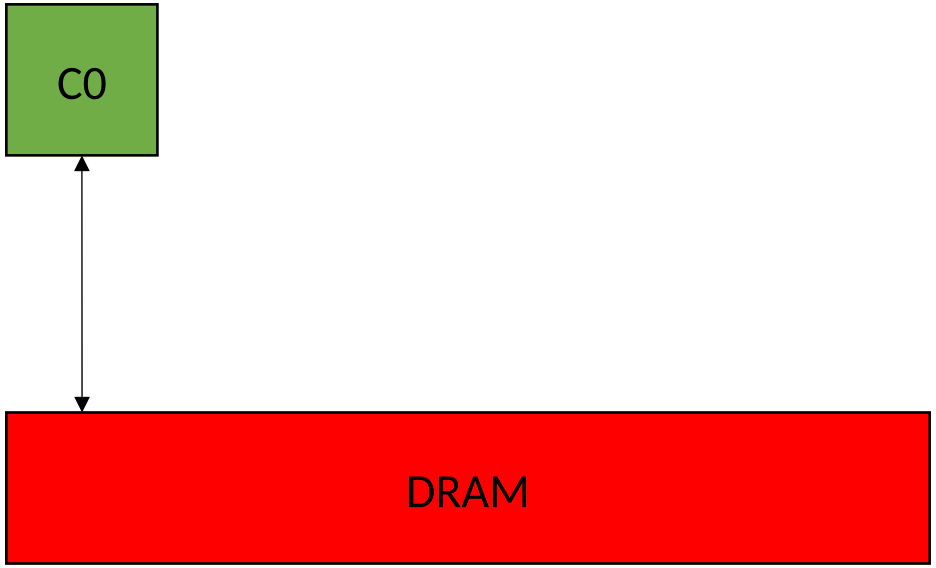


Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

200 cycles



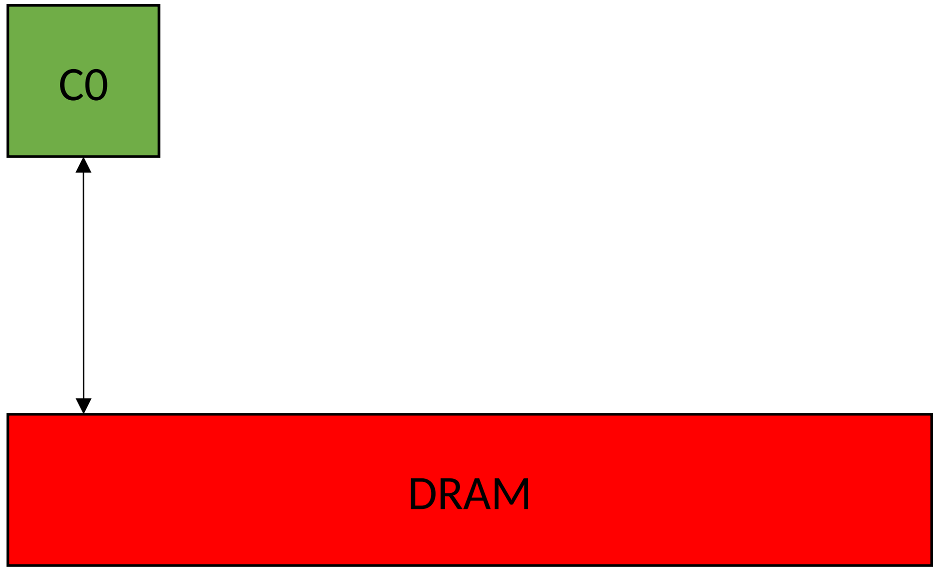
Main memory

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

200 cycles

1 cycles



Main memory

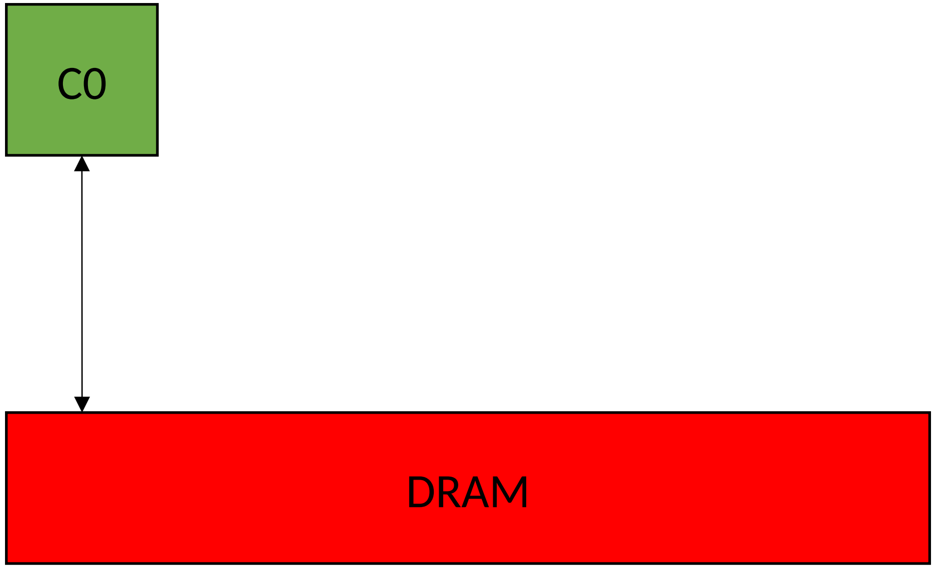
```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles

1 cycles

200 cycles



Main memory

```
int increment(int *a) {
    a[0]++;
}
```

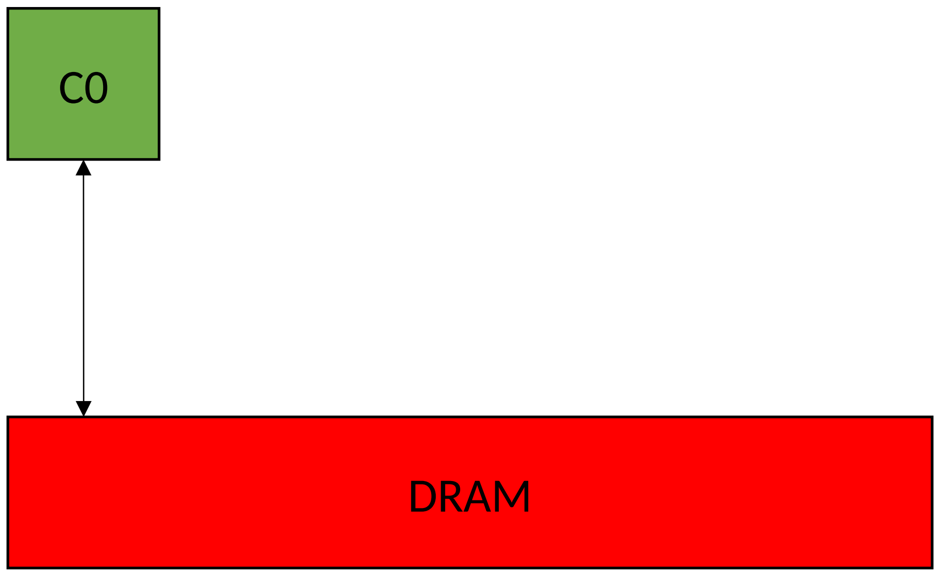
```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

200 cycles

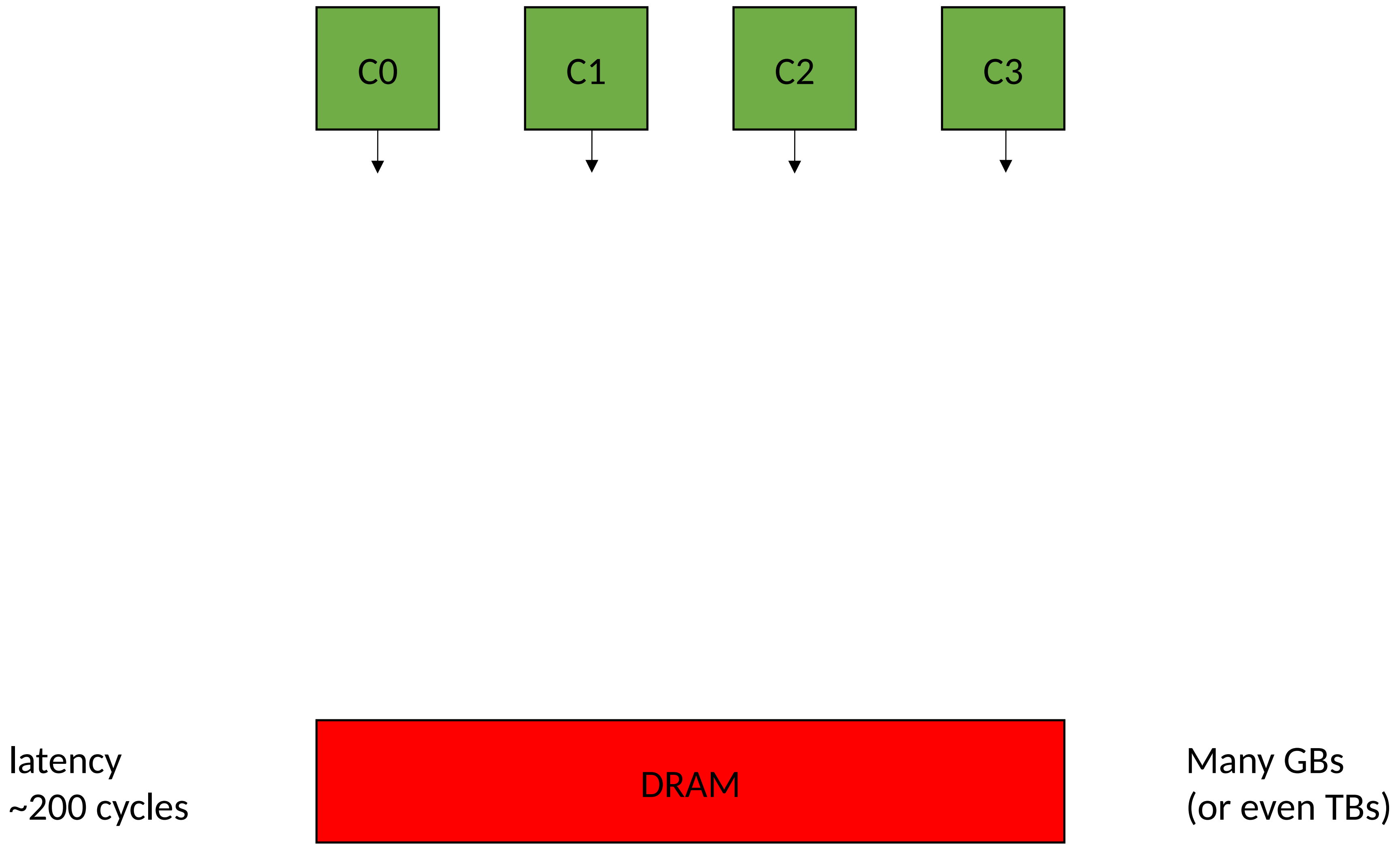
1 cycles

200 cycles

401 cycles

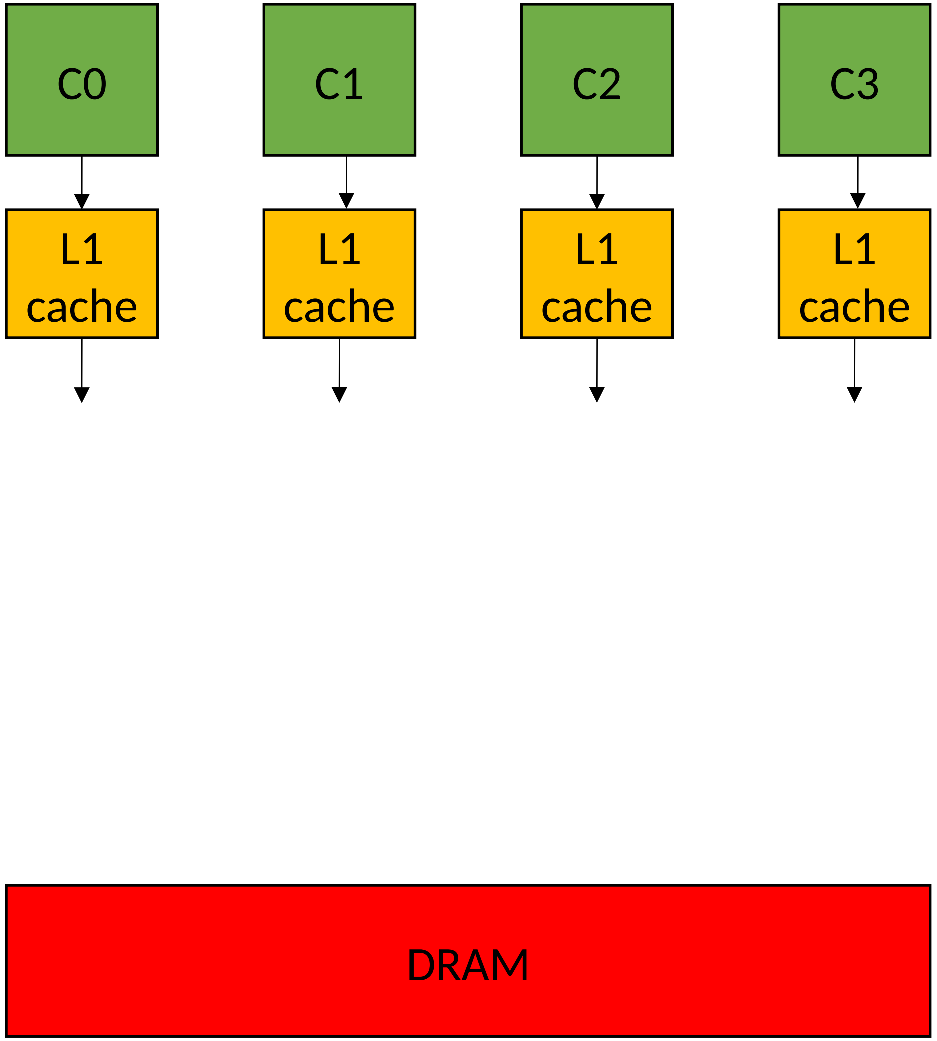


Caches



Caches

latency
~4 cycles

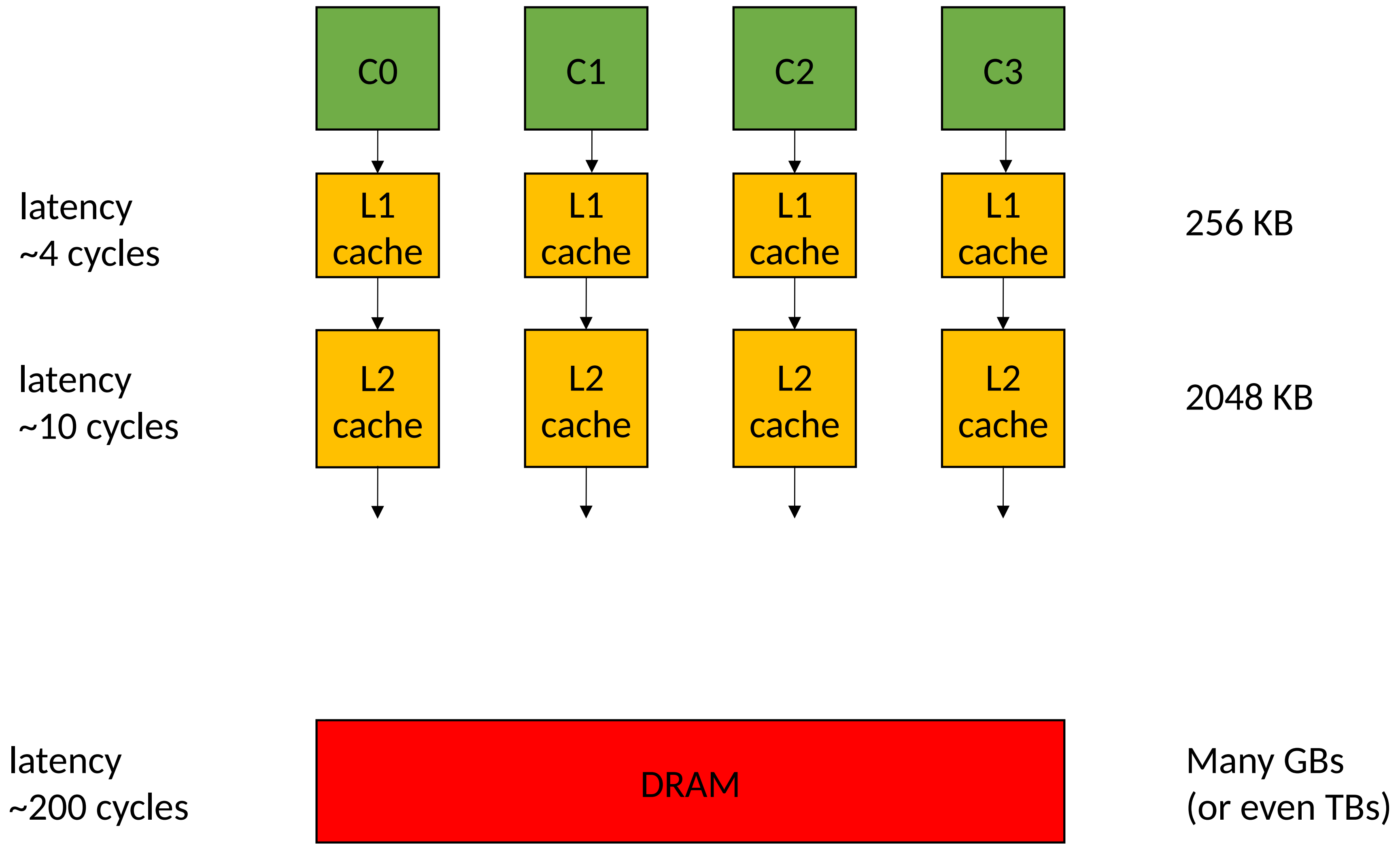


256 KB

latency
~200 cycles

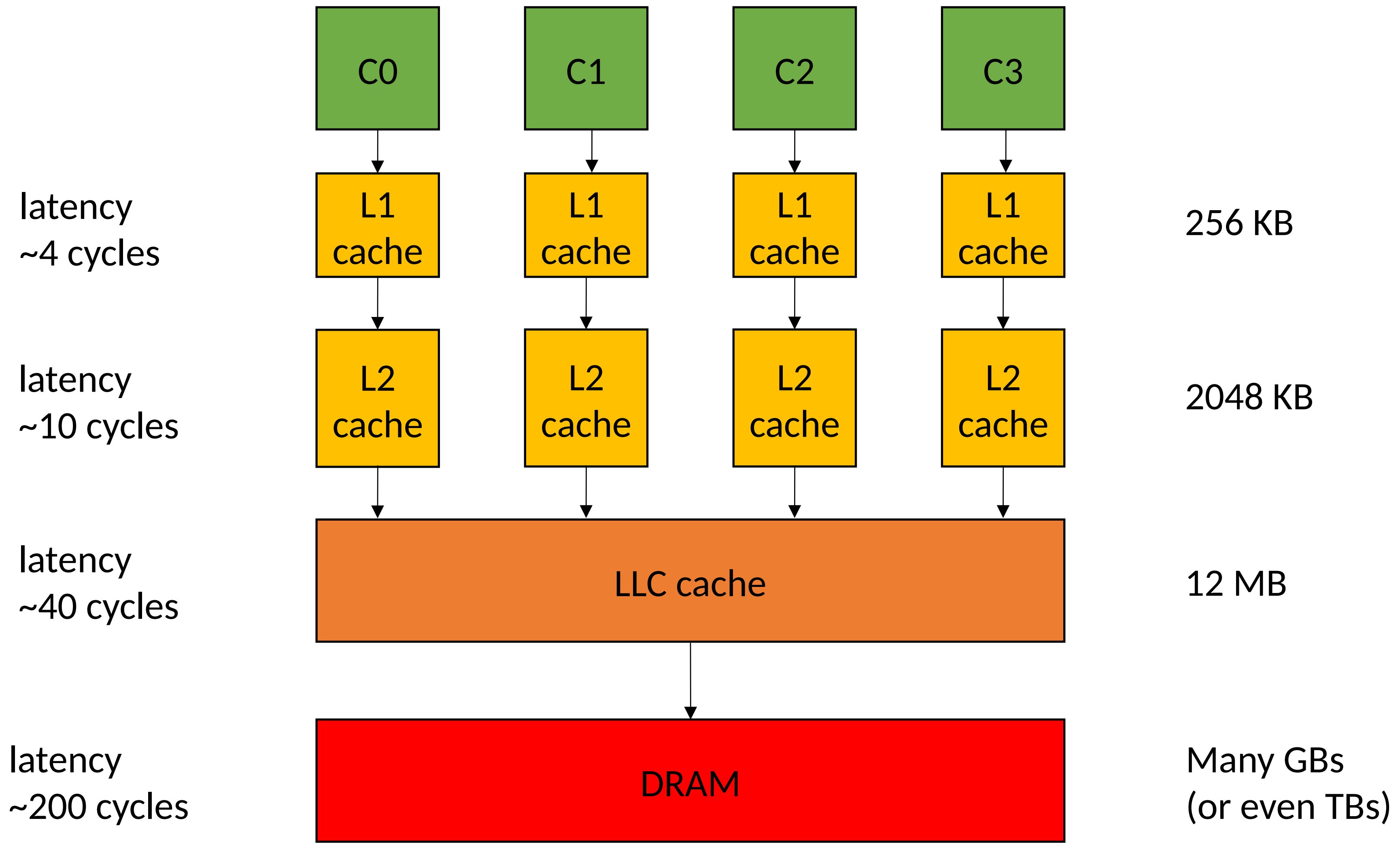
Many GBs
(or even TBs)

Caches



C0
C1
C2
C3
latency
~4 cycles
L1
cache
L1
cache
L1
cache
L1
cache
256 KB
latency
~10 cycles
L2
cache
L2
cache
L2
cache
L2
cache
2048 KB
latency
~200 cycles
DRAM
Many GBs
(or even TBs)

Caches

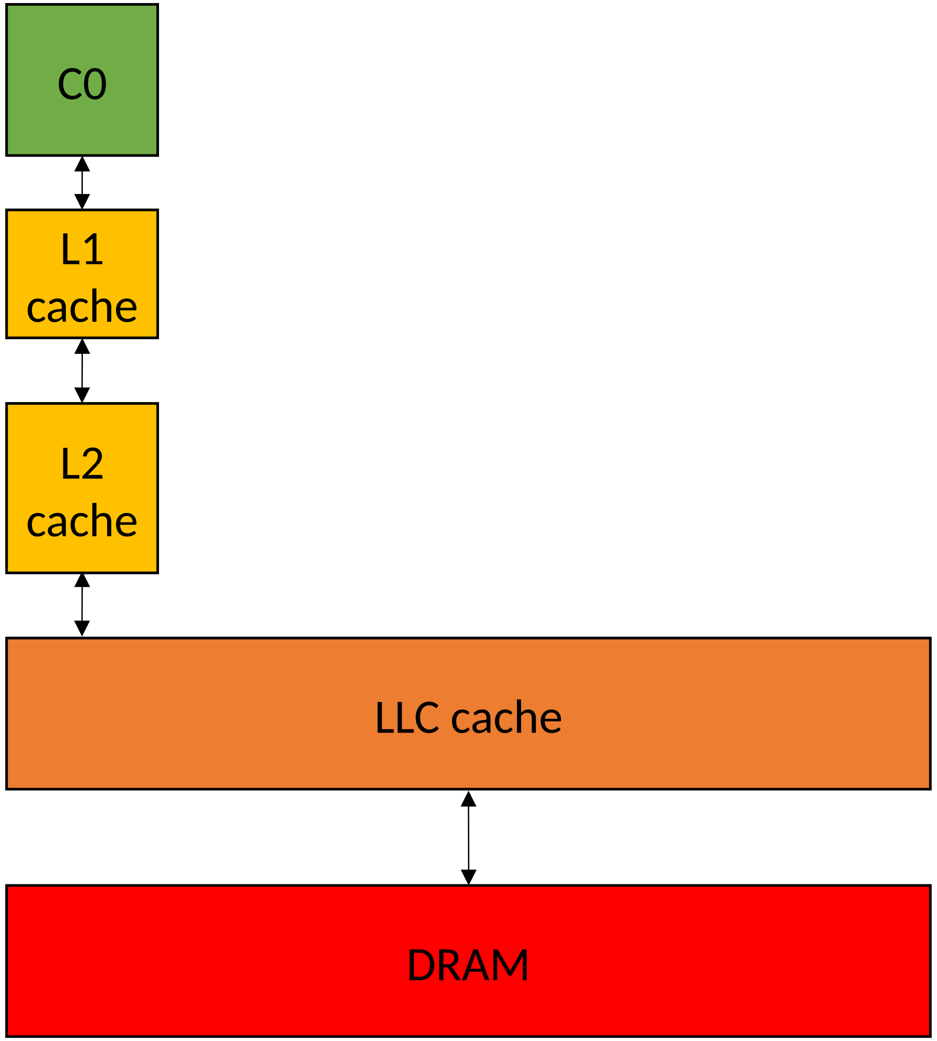


C0
C1
C2
C3
latency
~4 cycles
L1 cache
L1 cache
L1 cache
L1 cache
256 KB
latency
~10 cycles
L2 cache
L2 cache
L2 cache
L2 cache
2048 KB
latency
~40 cycles
LLC cache
12 MB
latency
~200 cycles
DRAM
Many GBs
(or even TBs)

Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```



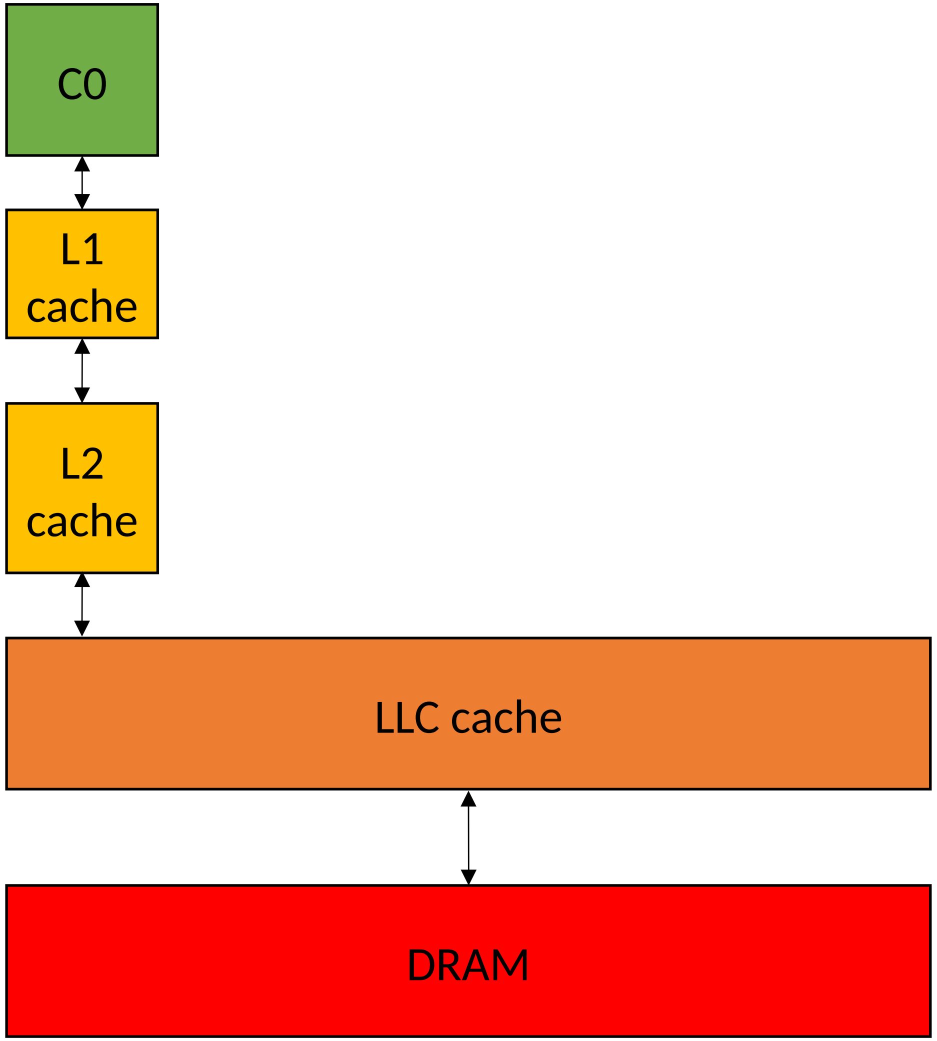
Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

4 cycles

Assuming the value is in the cache!



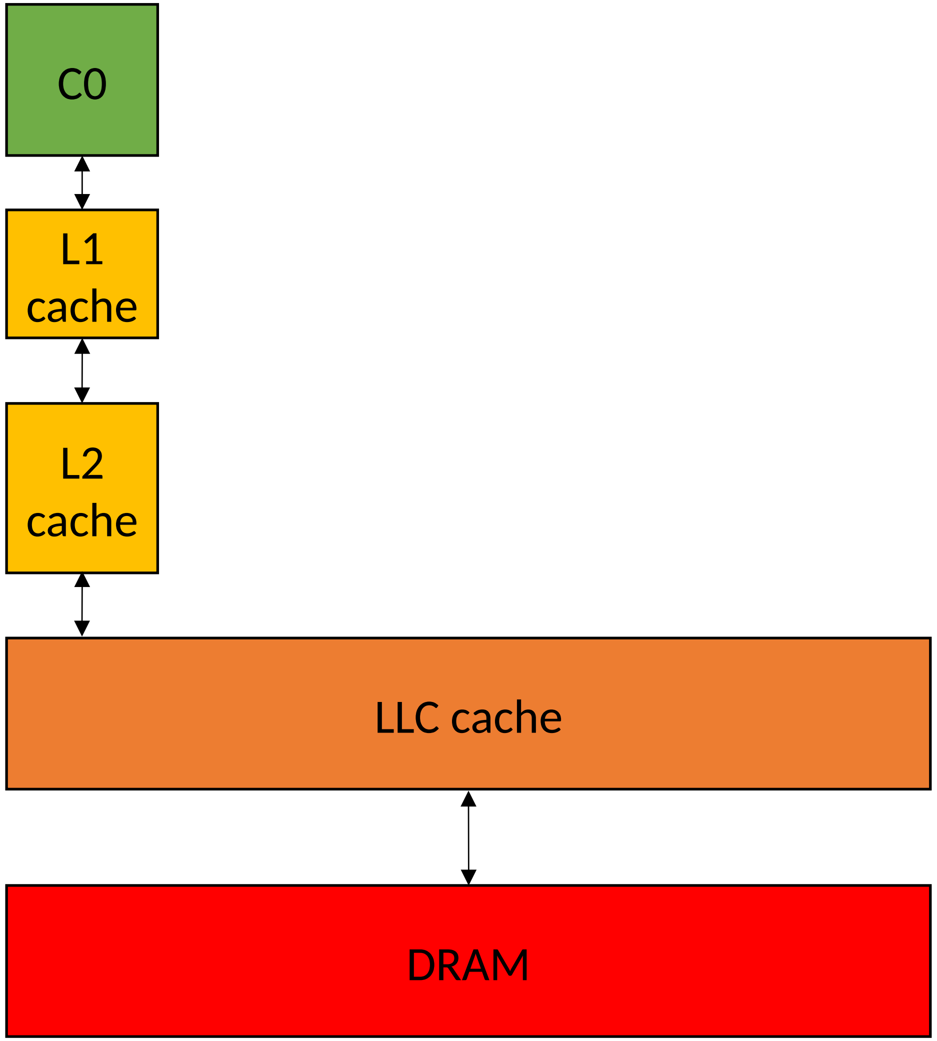
Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

4 cycles

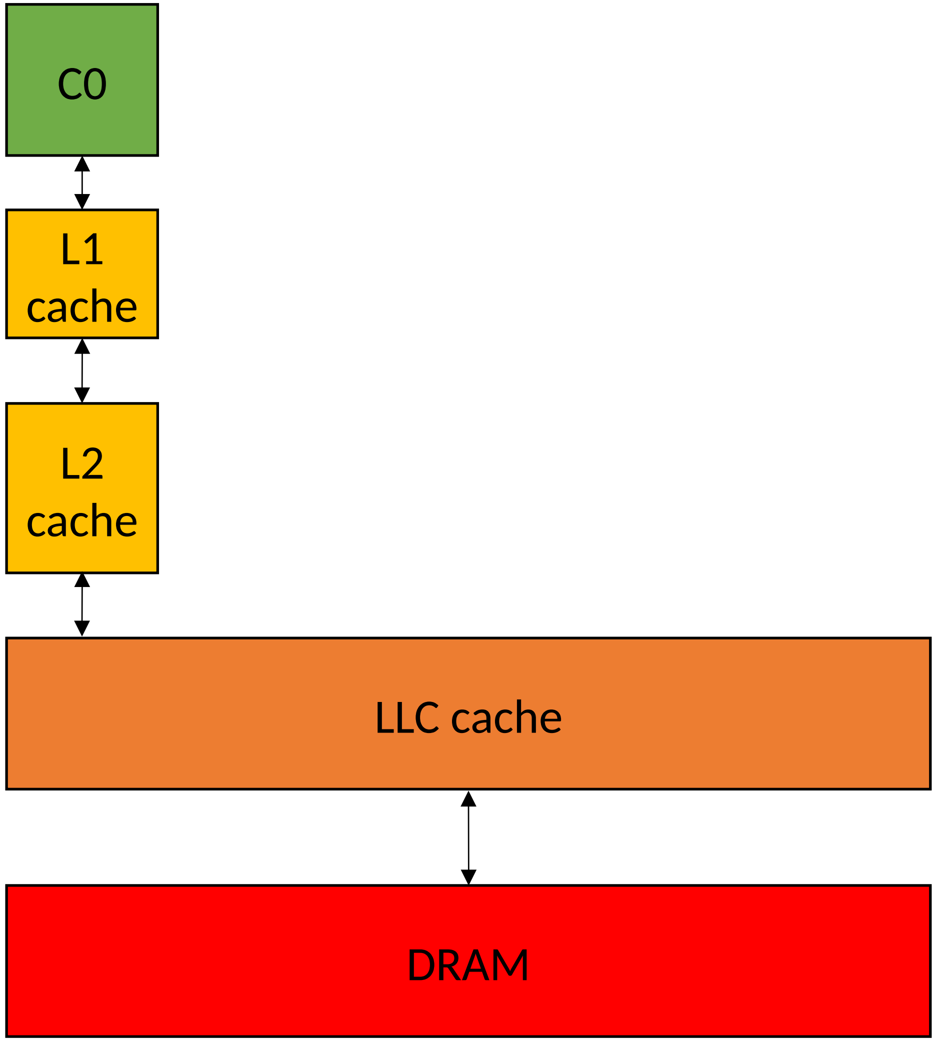
1 cycles



Caches

```
int increment(int *a) {
    a[0]++;
}
```

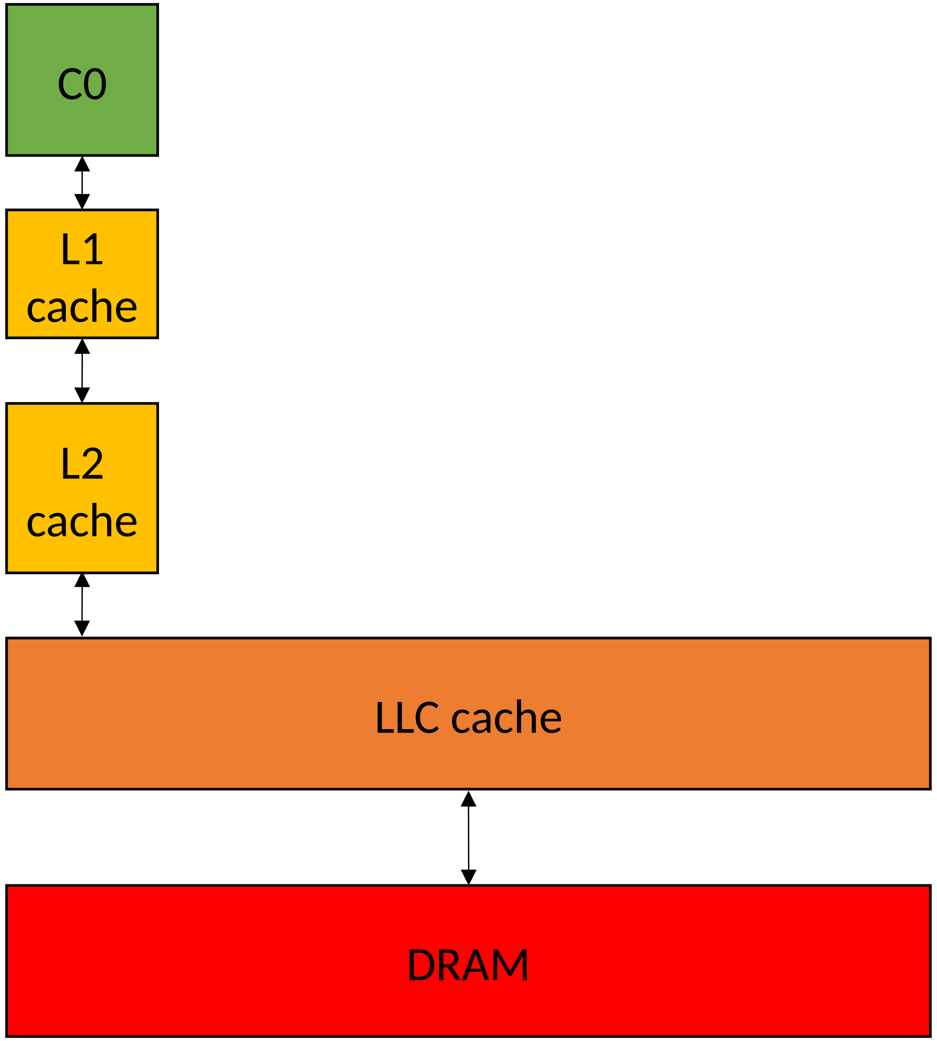
```
%5 = load i32, i32* %4          4 cycles
%6 = add nsw i32 %5, 1          1 cycles
store i32 %6, i32* %4           4 cycles
```



Caches

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4        4 cycles
                              1 cycles
%6 = add nsw i32 %5, 1        4 cycles
store i32 %6, i32* %4         9 cycles!
```



Quick overview of C/++ pointers/memory

Passing arrays in C++

```
int increment(int *a) {
    a[0]++;
}
```

```
int increment_alt1(int a[1]) {
    a[0]++;
}
```

Not checked at compile time! but hints can help with compiler optimizations. Also good self documenting code.

```
int increment_alt2(int a[]) {
    a[0]++;
}
```

Passing pointers

```
int foo0(int *a) {
    increment(a)
}
```

pass pointer directly through

```
int foo1(int *a) {
    increment(&(a[8]))
}
```

pass an offset of 8

```
int foo2(int *a) {
    increment(a + 8)
}
```

another way to pass an offset of 8

Memory Allocation

```
int allocate_int_array0() {
  int ar[16];
}
```

stack allocation

```
int allocate_int_array1() {
  int *ar = new int[16];
  delete[] ar;
}
```

C++ style

```
int allocate_int_array2() {
  int *ar = (int*)malloc(sizeof(int)*16);
  free(ar);
}
```

C style

Cache lines

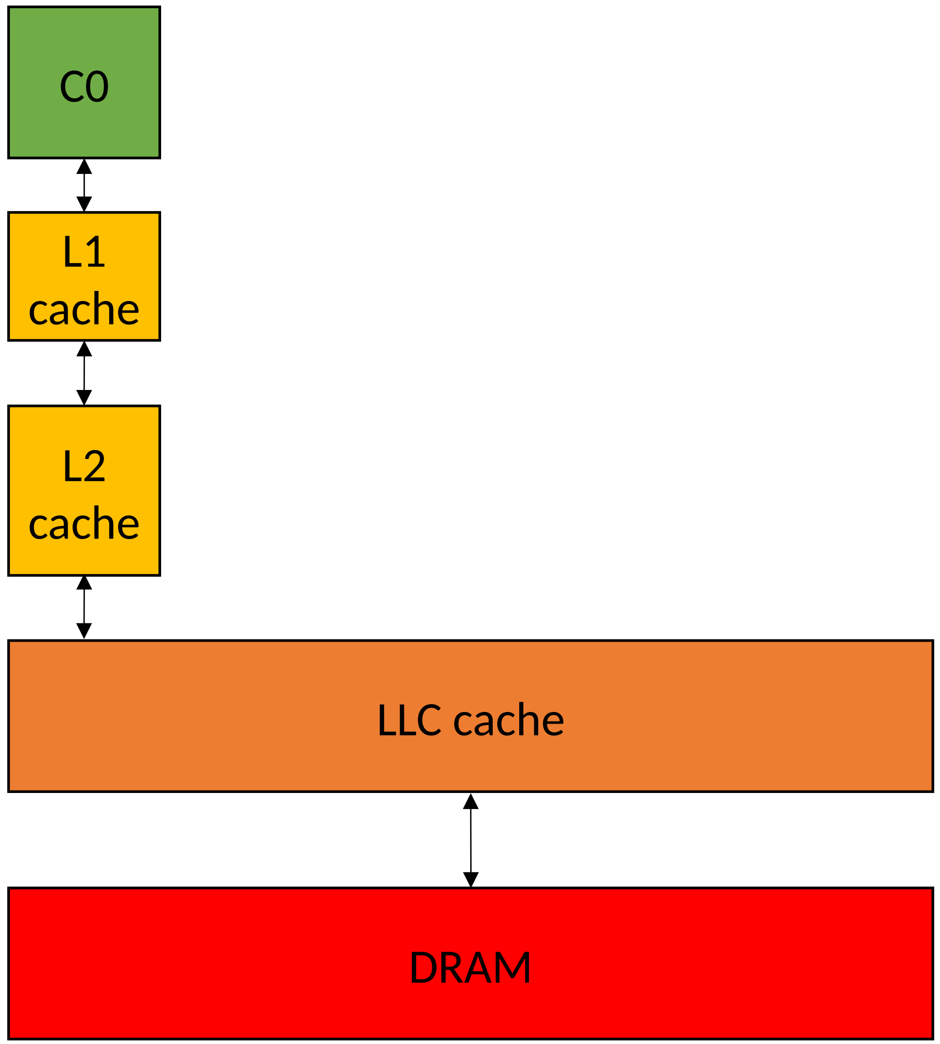
- Cache line size for x86: 64 bytes:
  - 64 chars
  - 32 shorts
  - 16 float or int
  - 8 double or long

Caches

Assume a[0] is not in the cache

```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```

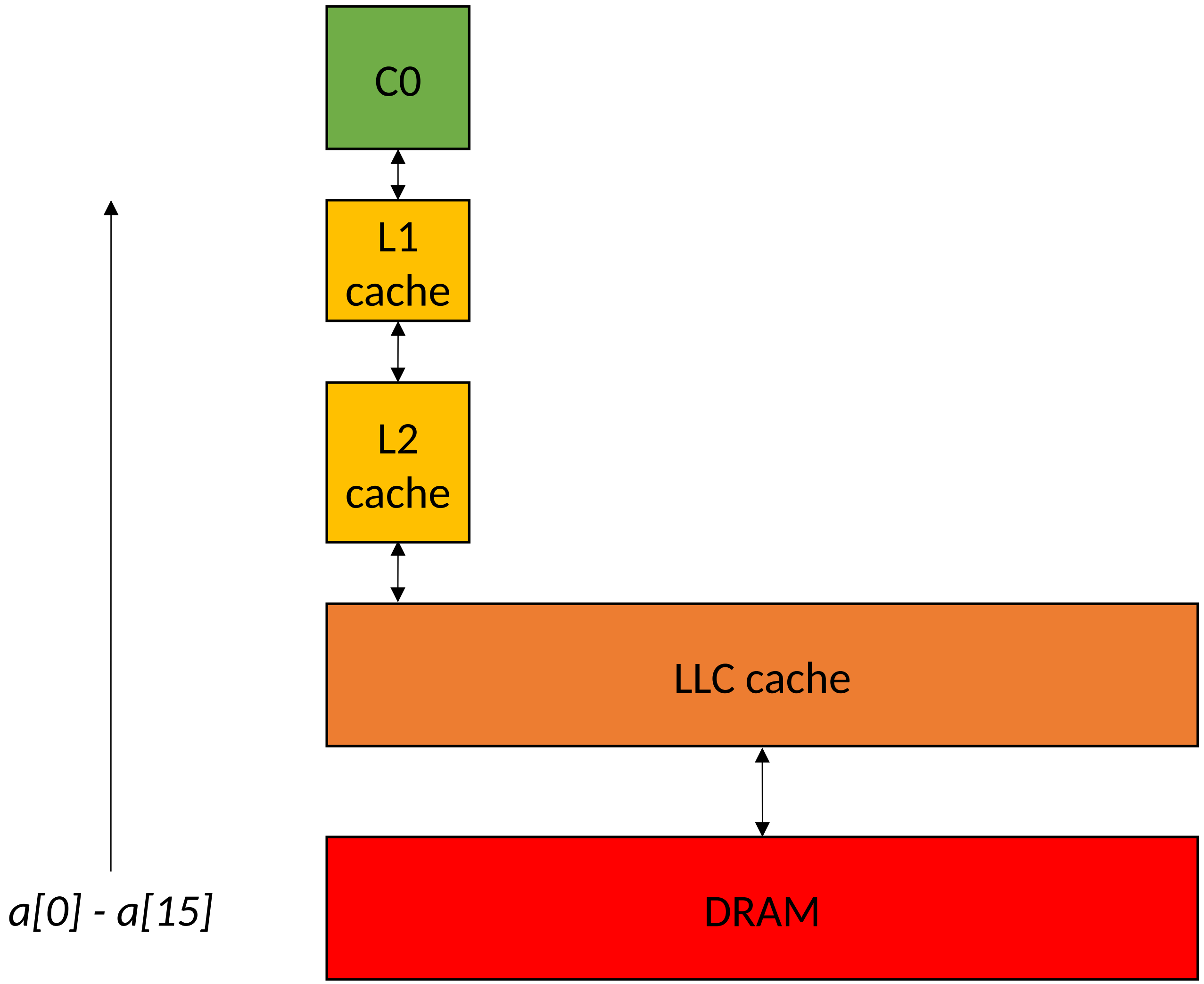


Assume a[0] is not in the cache

Caches

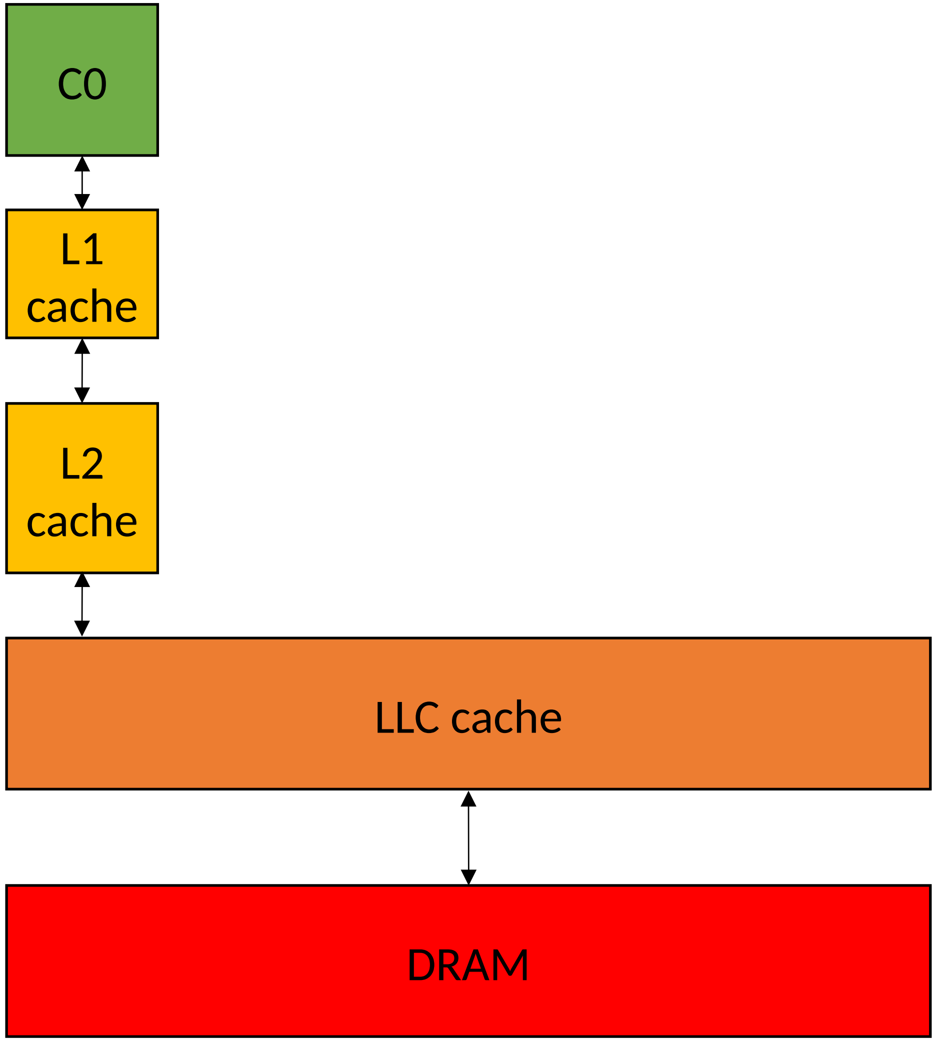
```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```



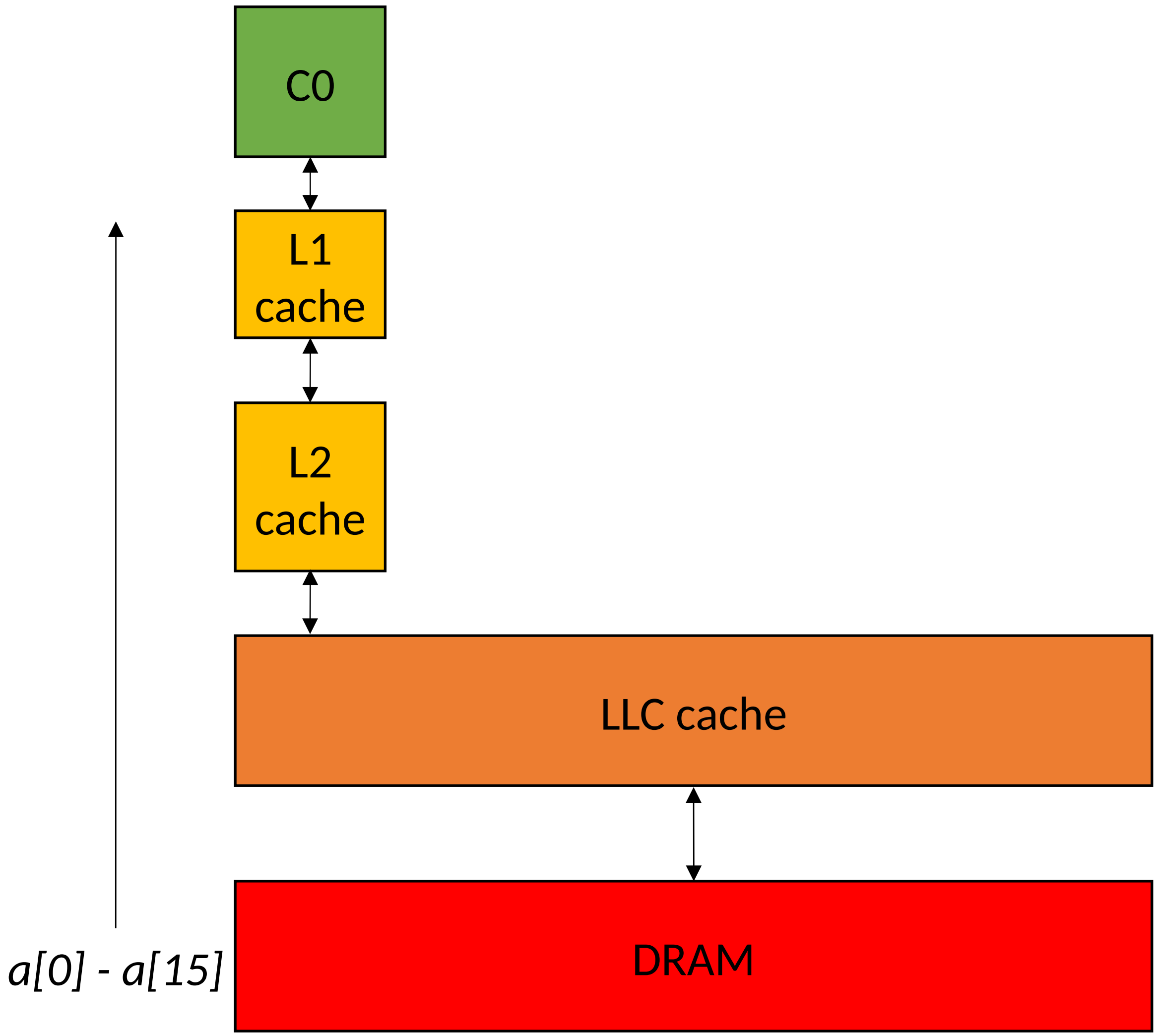
Caches

```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```



Caches

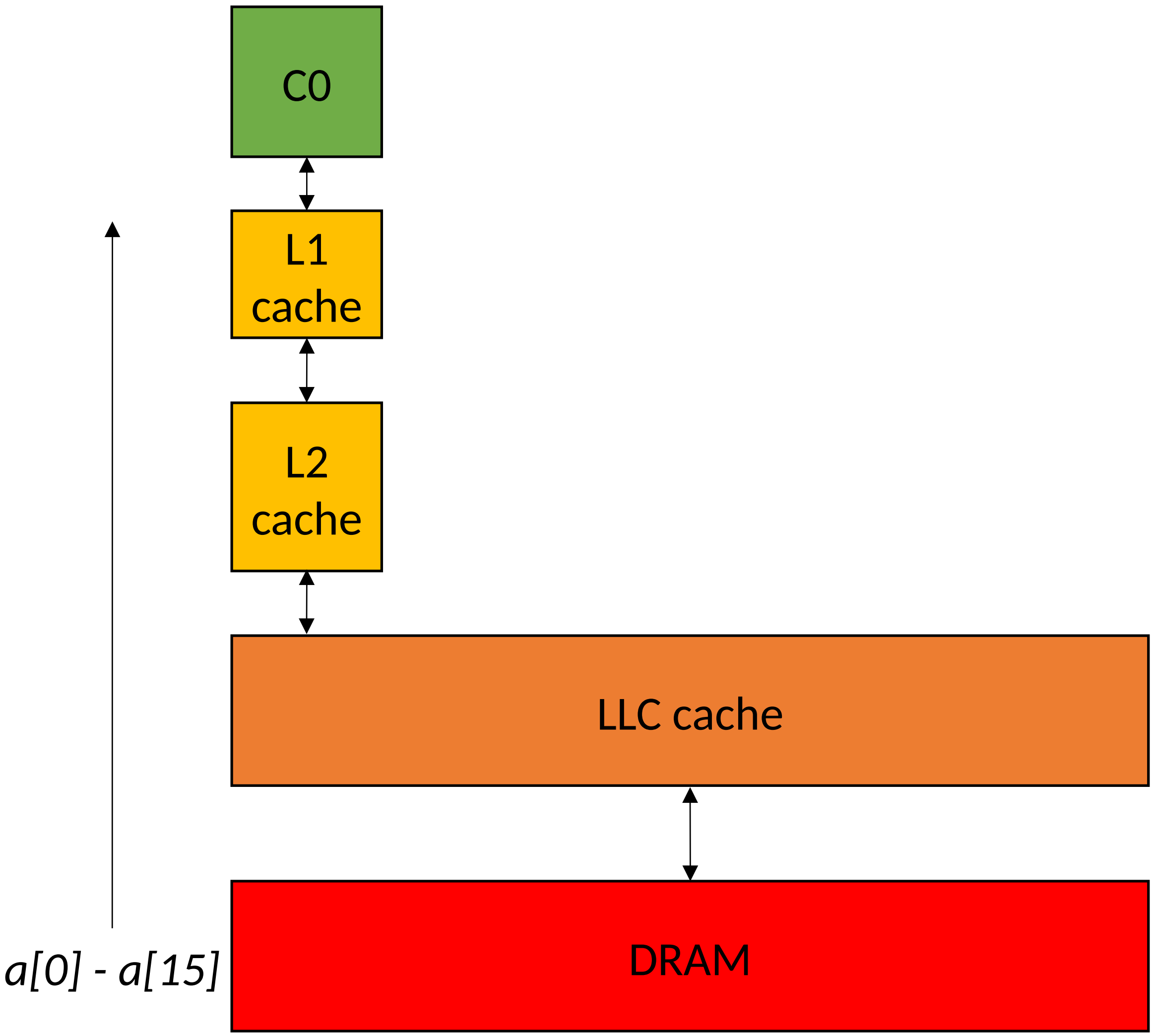
```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```



Caches

```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```

will be a hit because we've loaded a[0] cache line

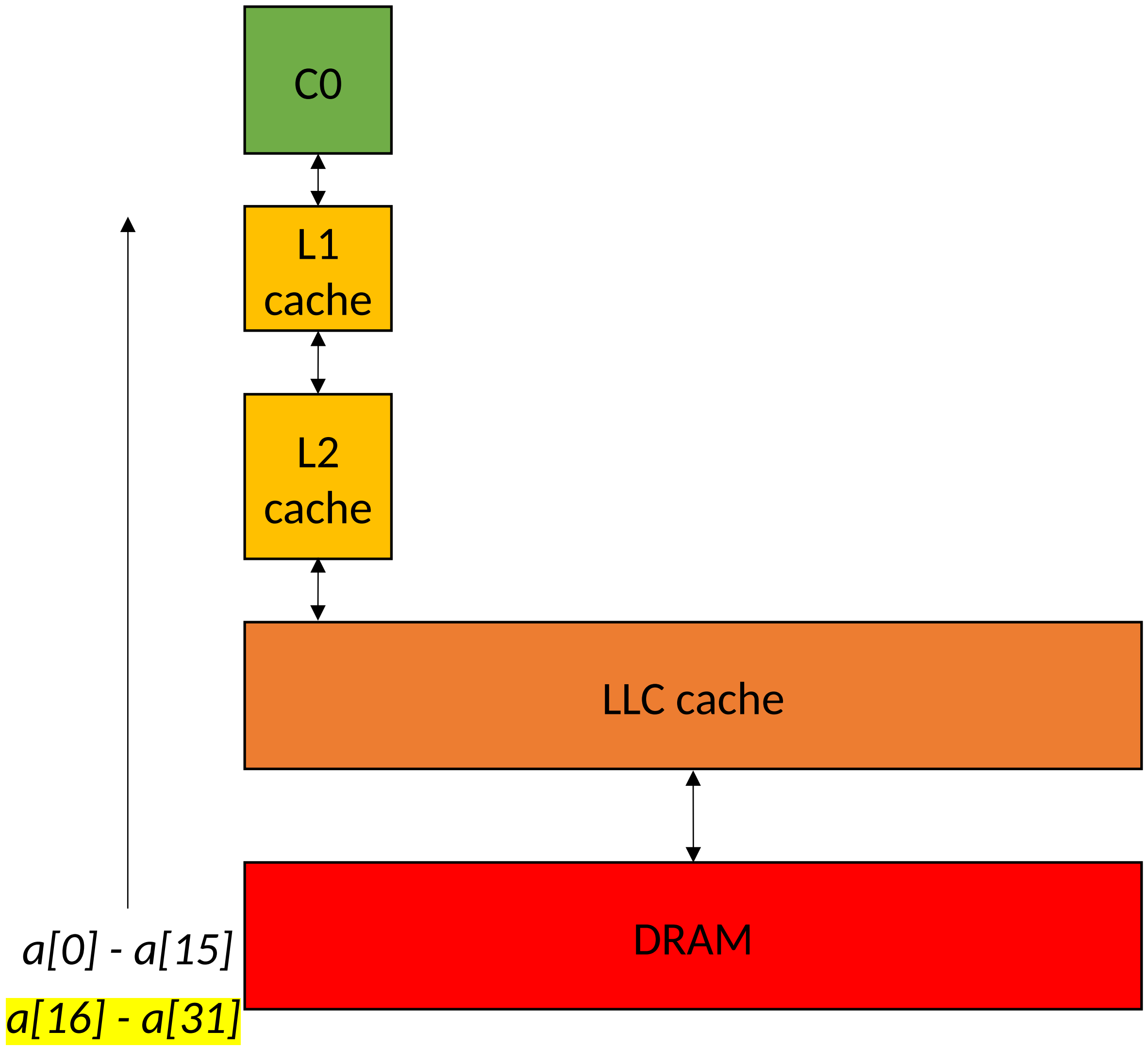


Assume a[0] is not in the cache

Caches

```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```

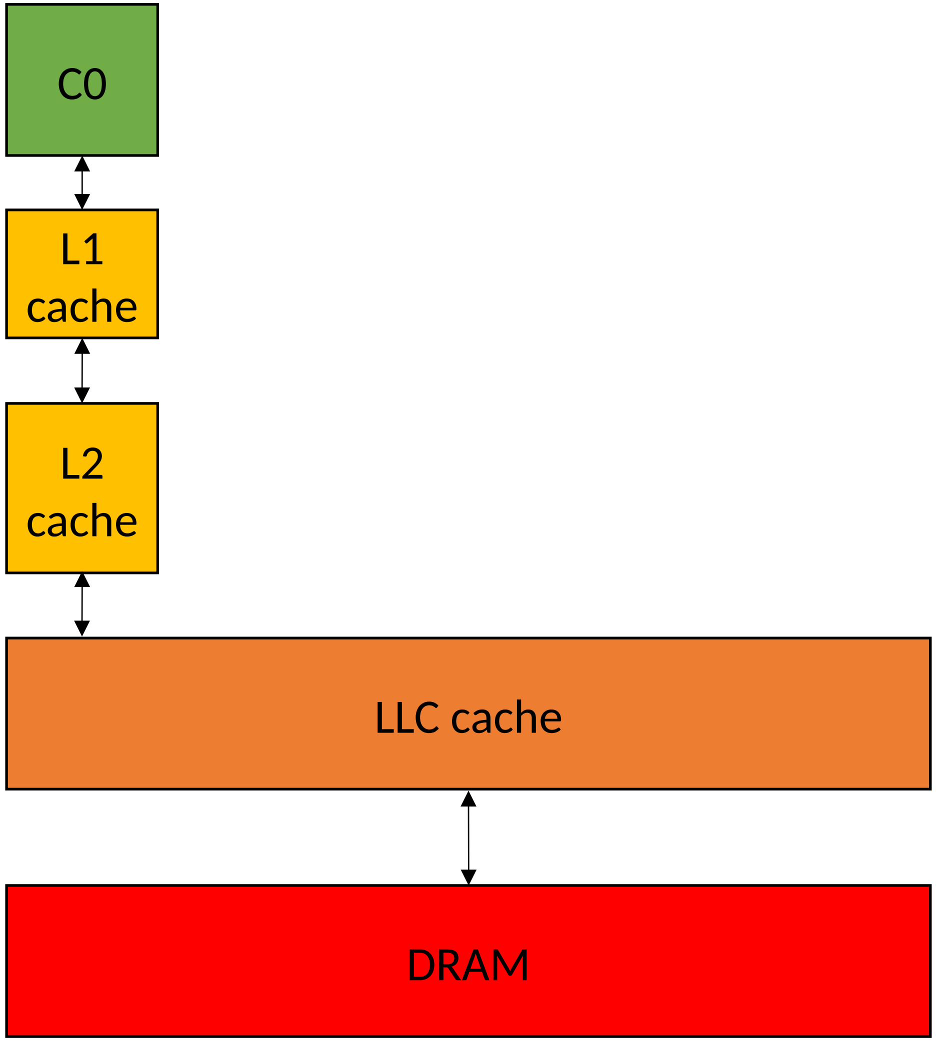
Miss



Assume a[0] is not in the cache

Cache alignment

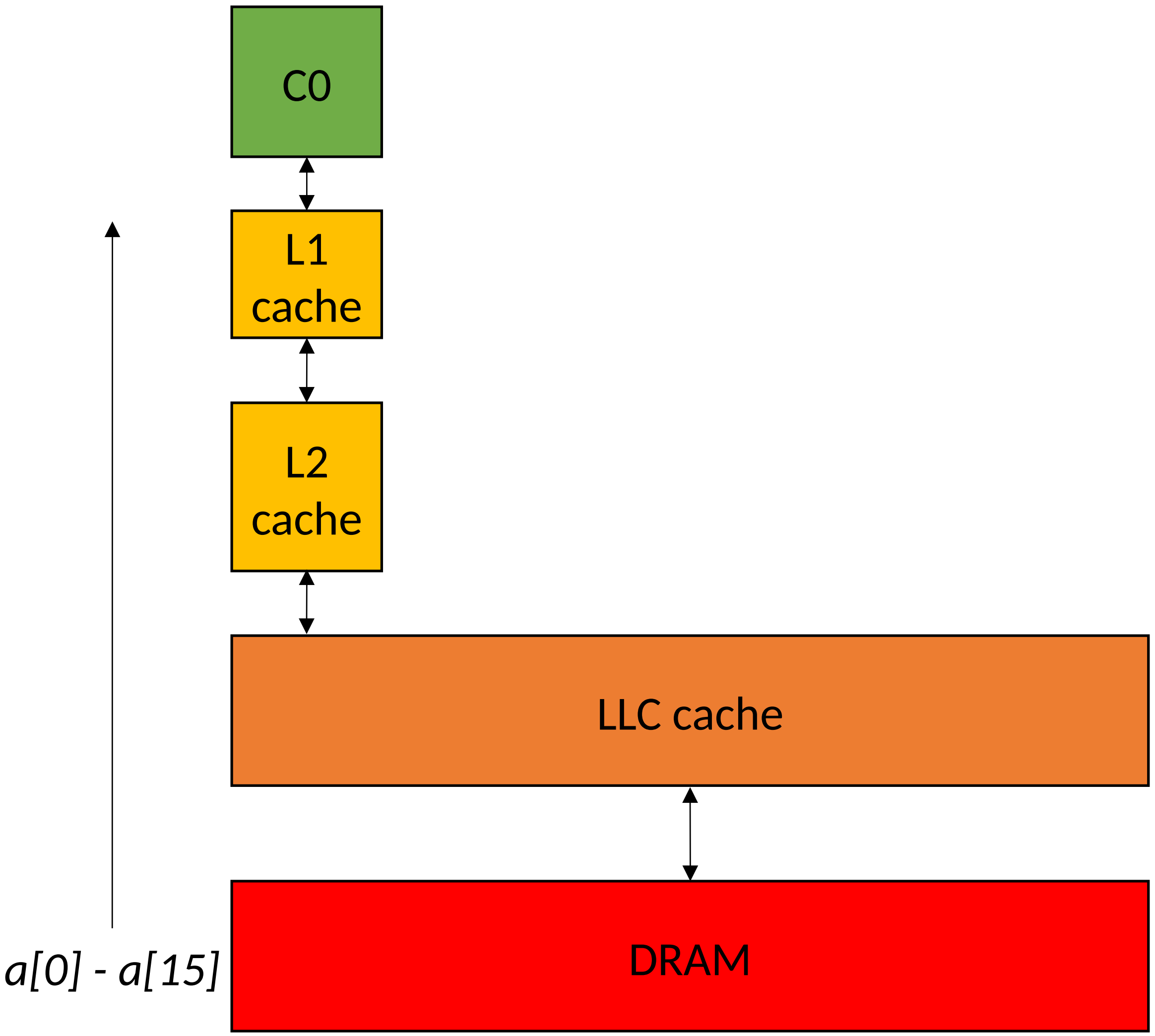
```
int increment_several(int *b) {
    b[0]++;
    b[15]++;
}

int foo(int *a) {
    increment_several(&(a[8]))
}
```



Assume a[0] is not in the cache

Cache alignment

```
int increment_several(int *b) {
    b[0]++;
    b[15]++;
}

int foo(int *a) {
    increment_several(&(a[8]))
}
```

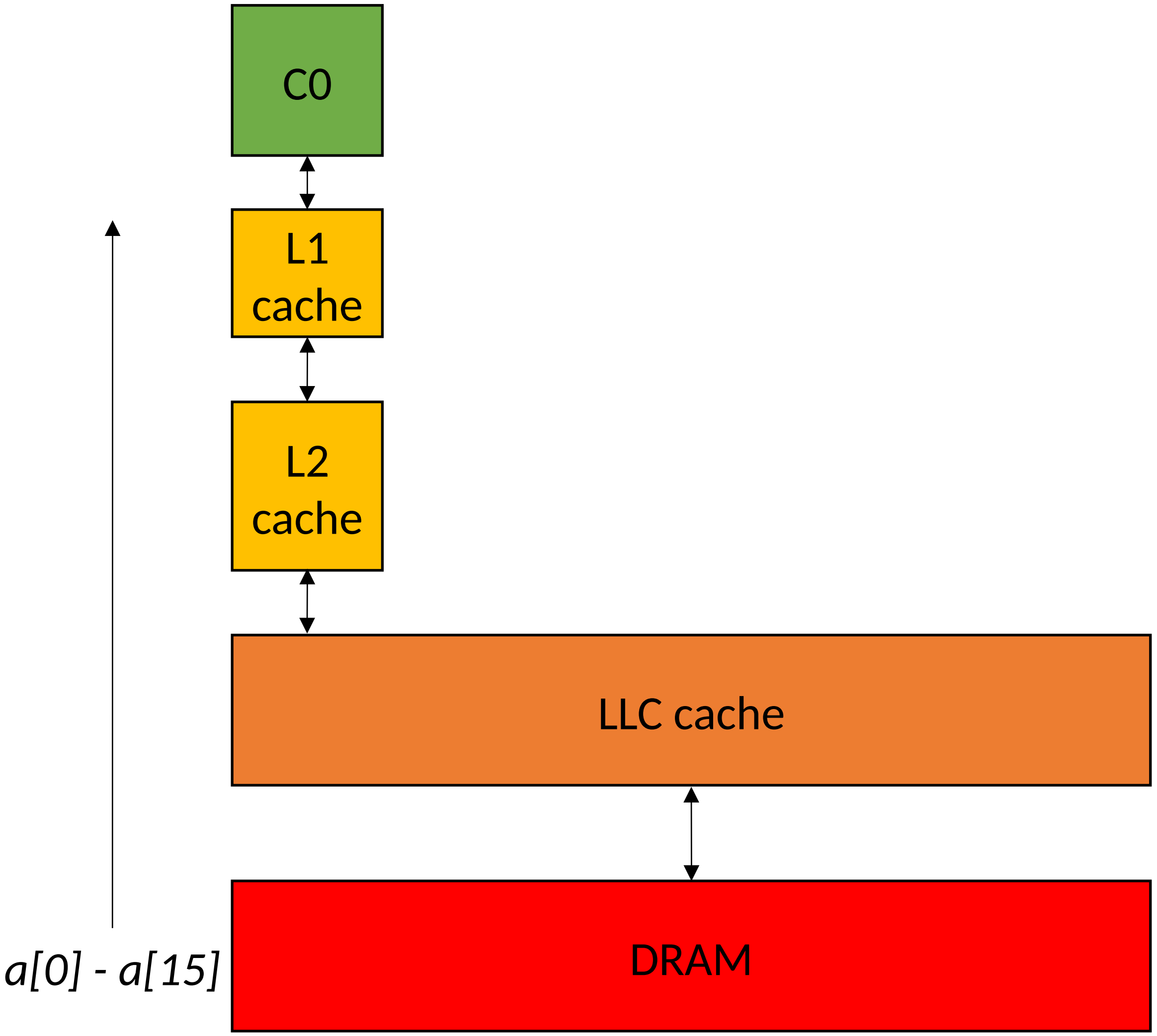


Assume a[0] is not in the cache

Cache alignment

```
int increment_several(int *b) {
    b[0]++;
    b[15]++;
}

int foo(int *a) {
    increment_several(&(a[8]))
}
```

This loads a[8]

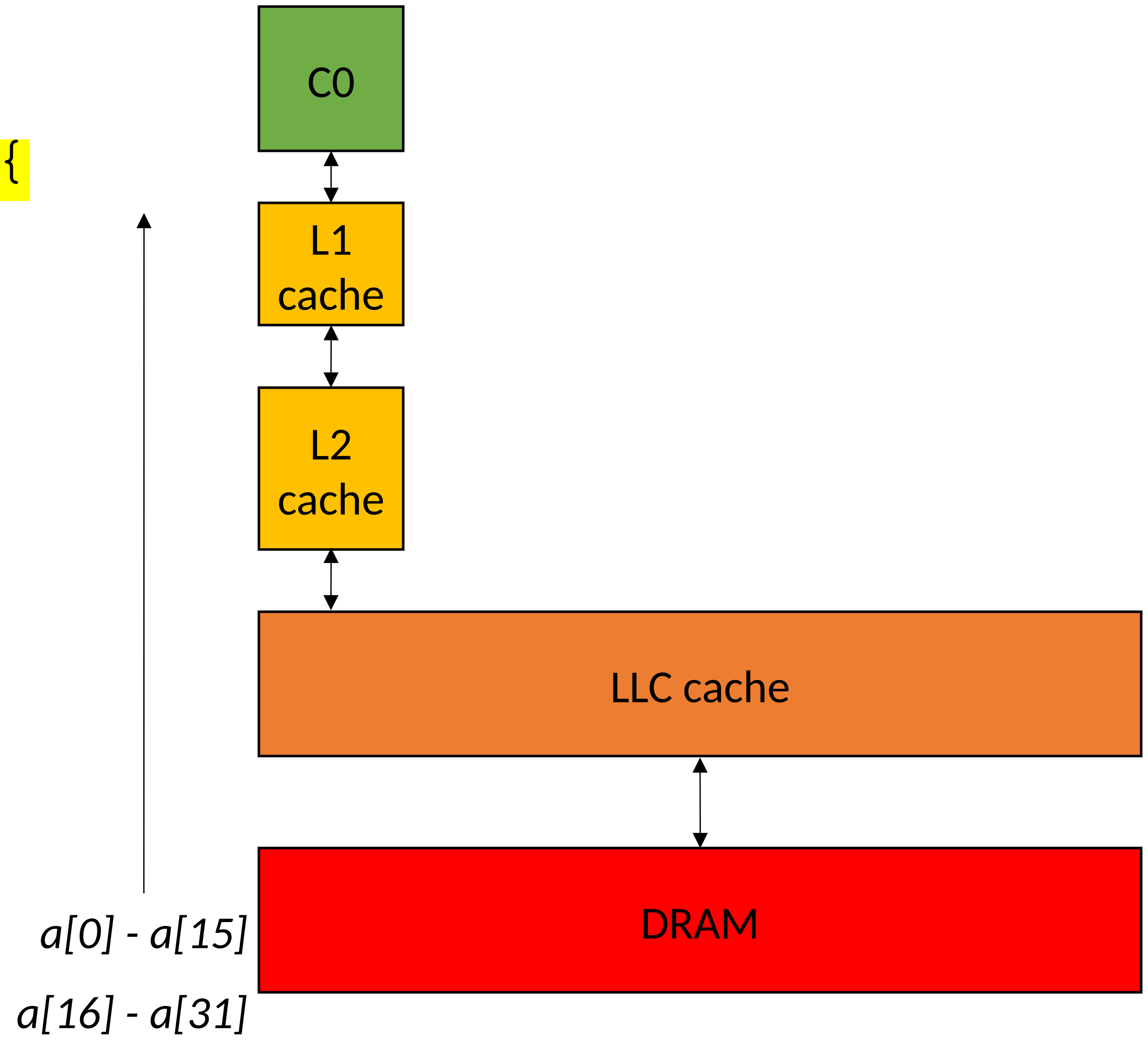


Assume a[0] is not in the cache

Cache alignment

```
int increment_several(int *b) {
    b[0]++;
    b[15]++;
}

int foo(int *a) {
    increment_several(&(a[8]))
}
```

This loads a[8]
This loads a[23], a miss!



Cache alignment

- Malloc typically returns a pointer with “good” alignment.
  - System specific, but will be aligned at least to a cache line, more likely a page

- For very low-level programming you can use special aligned malloc functions

- Prefetchers will also help for many applications (e.g. streaming)

Cache alignment

- Malloc typically returns a pointer with “good” alignment.
  - System specific, but will be aligned at least to a cache line, more likely a page

- For very low-level programming you can use special aligned malloc functions

- Prefetchers will also help for many applications (e.g. streaming)

```
for (int i = 0; i < 100; i++) {
  a[i] += b[i];
}
```

prefetcher will start collecting consecutive data in the cache if it detects patterns like this.

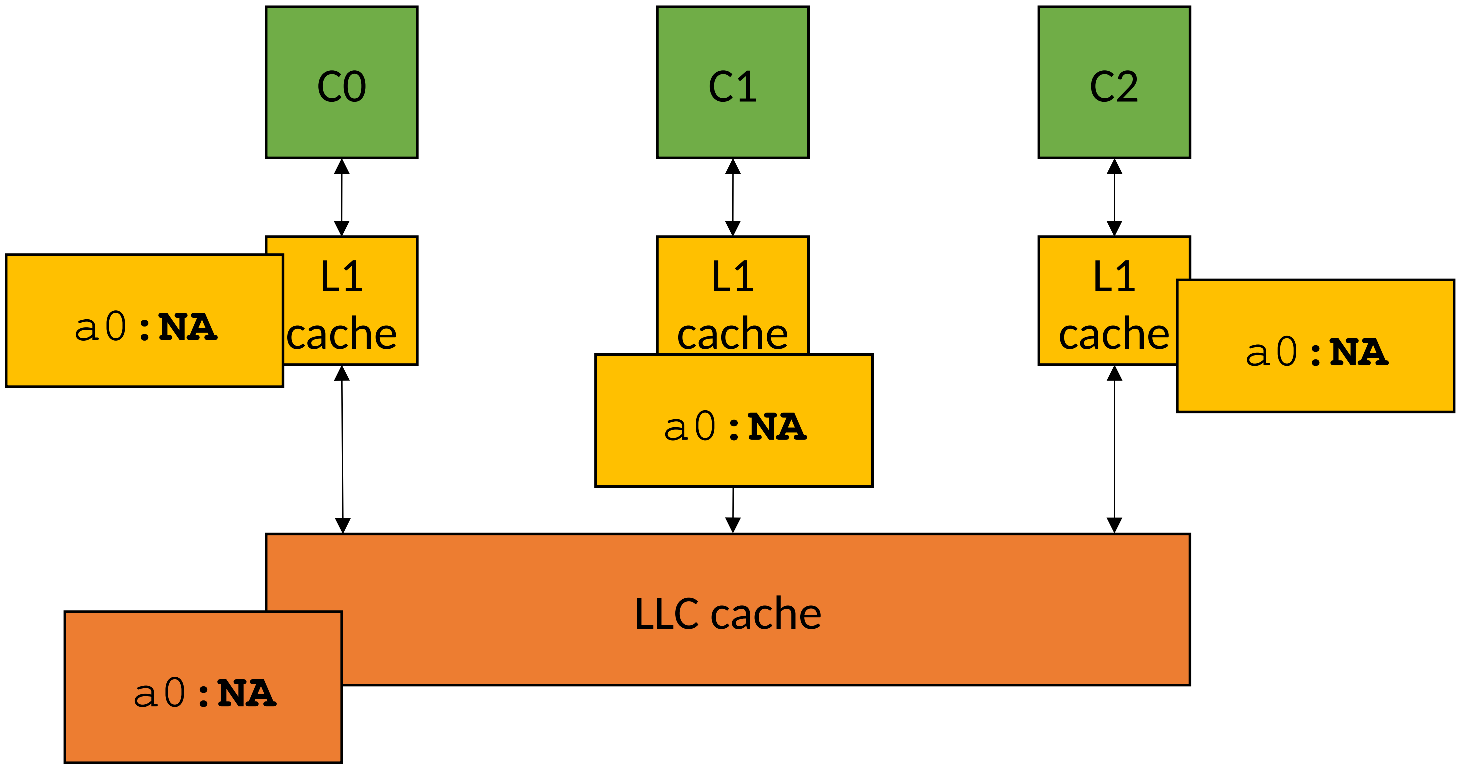
Cache coherence

Cache coherence

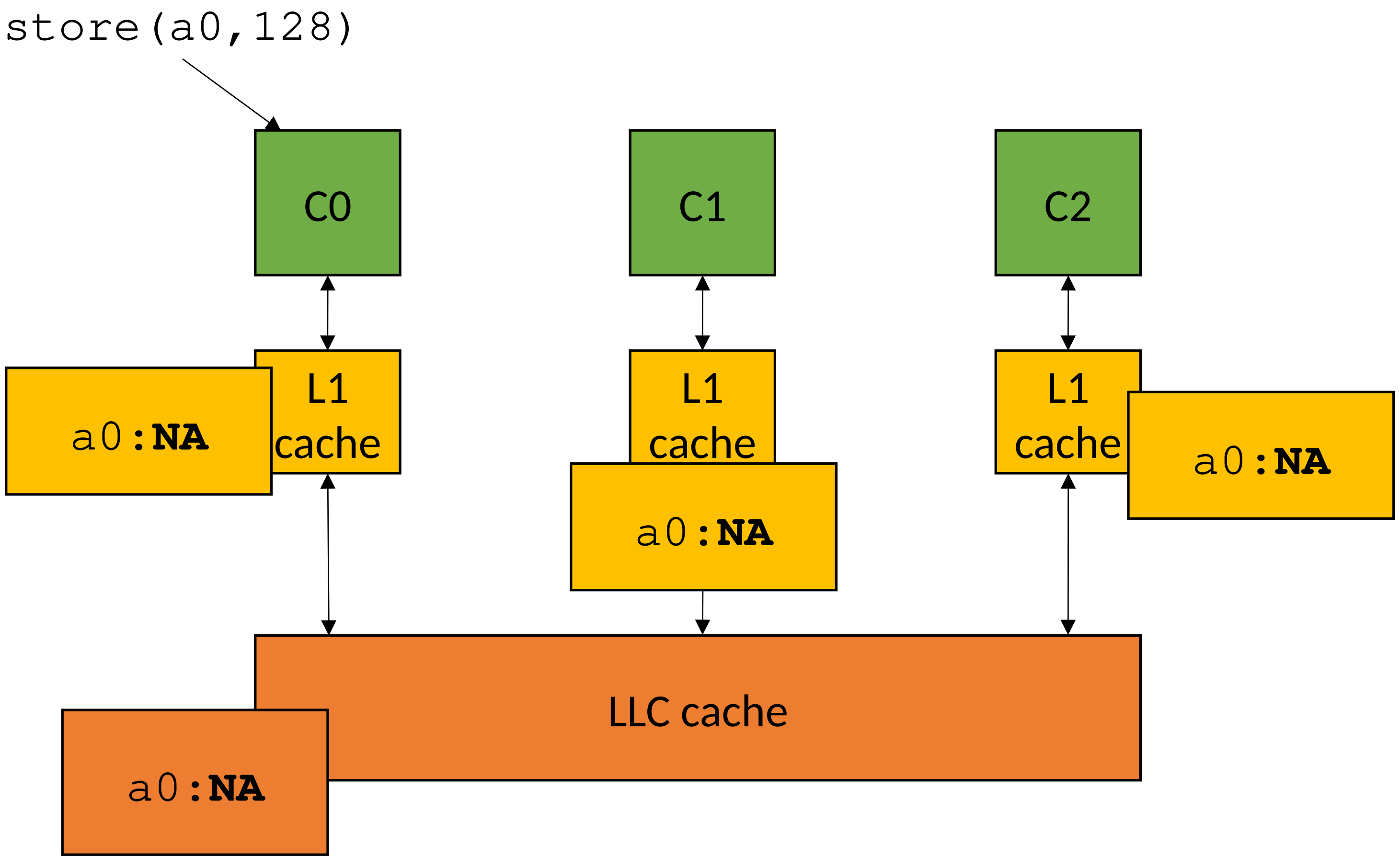
How to manage multiple values for the same address in the system?

simplified view for illustration: L1 cache and LLC

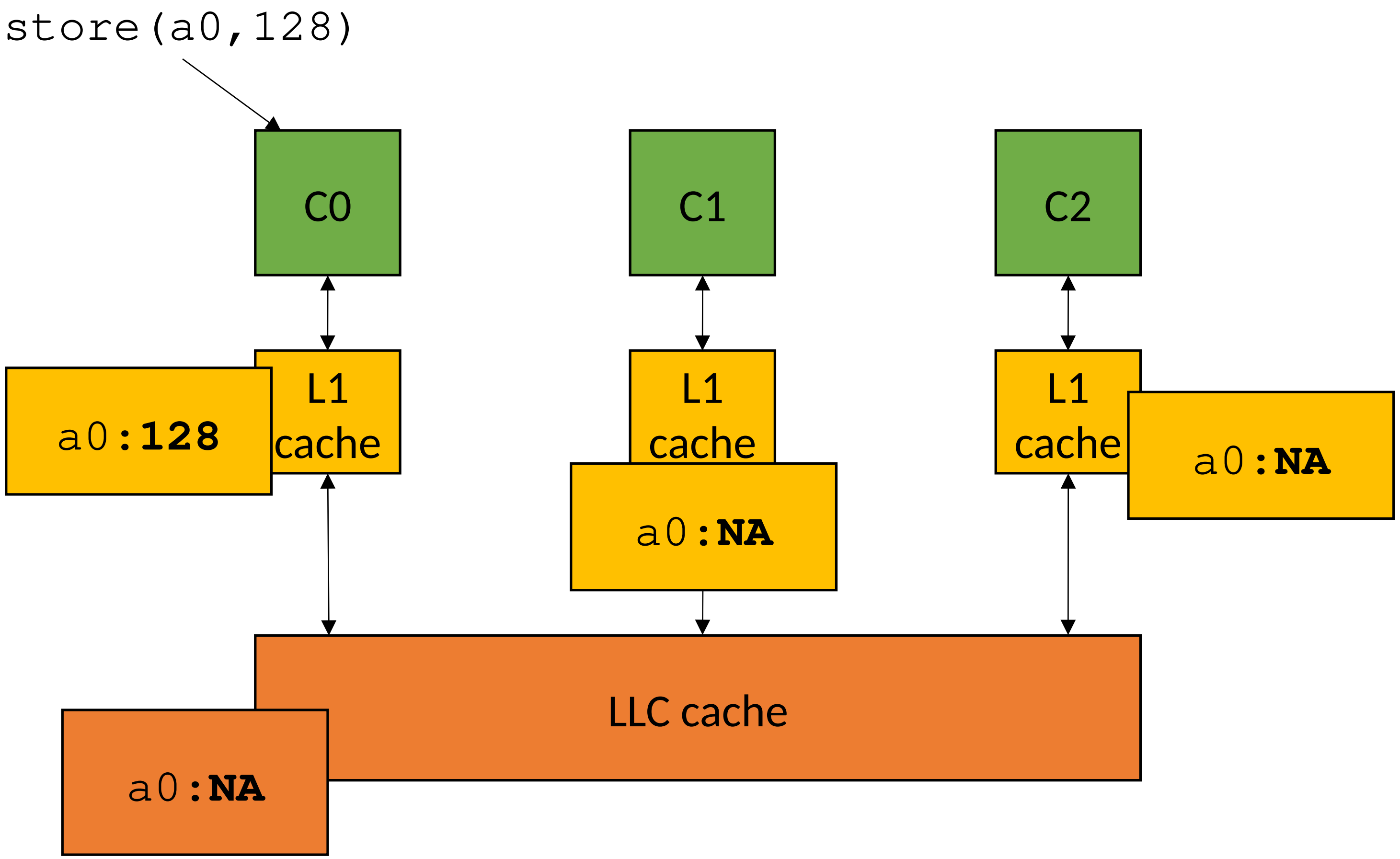
Consider 3 cores accessing the same memory location



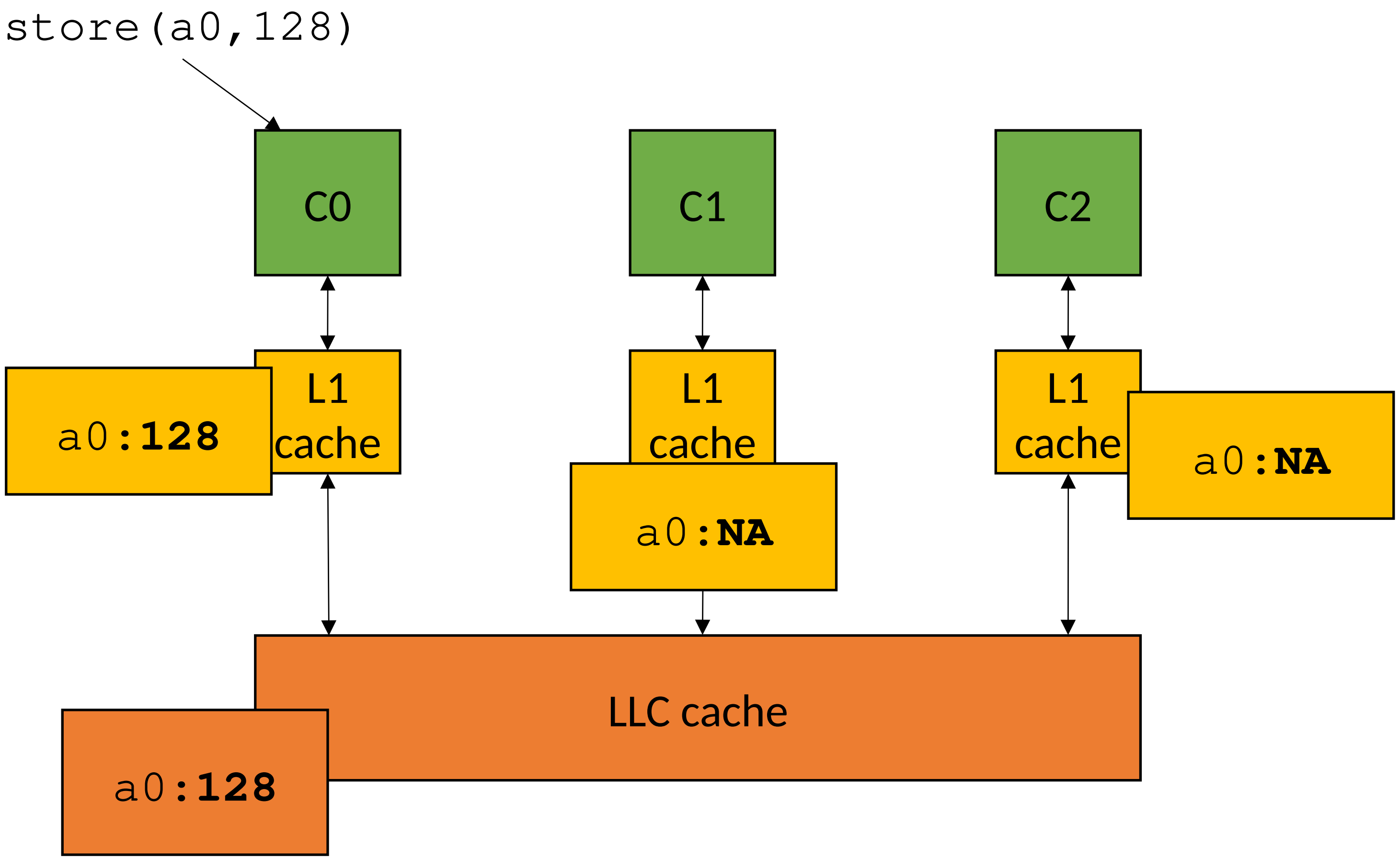
Cache coherence



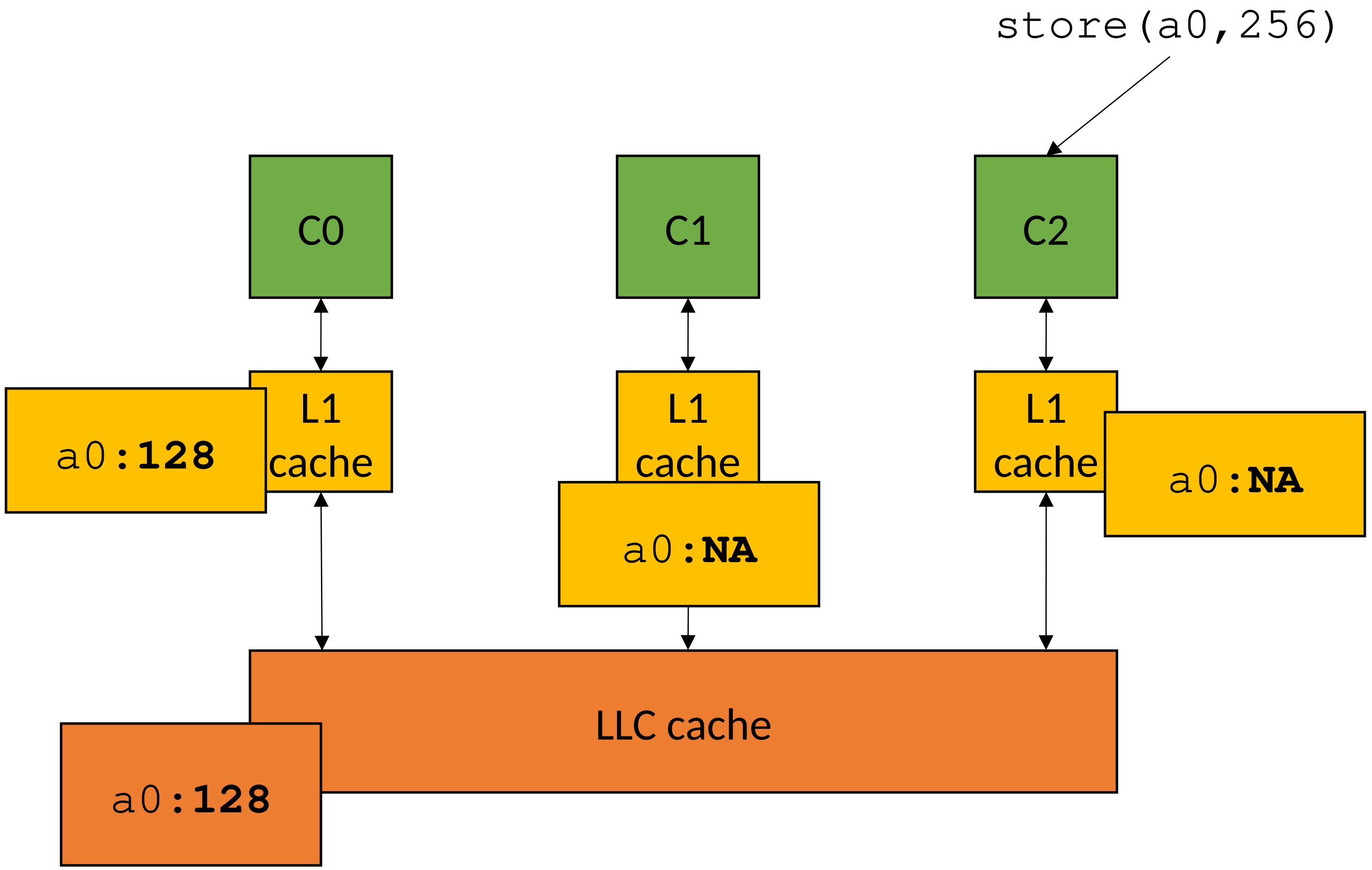
Cache coherence



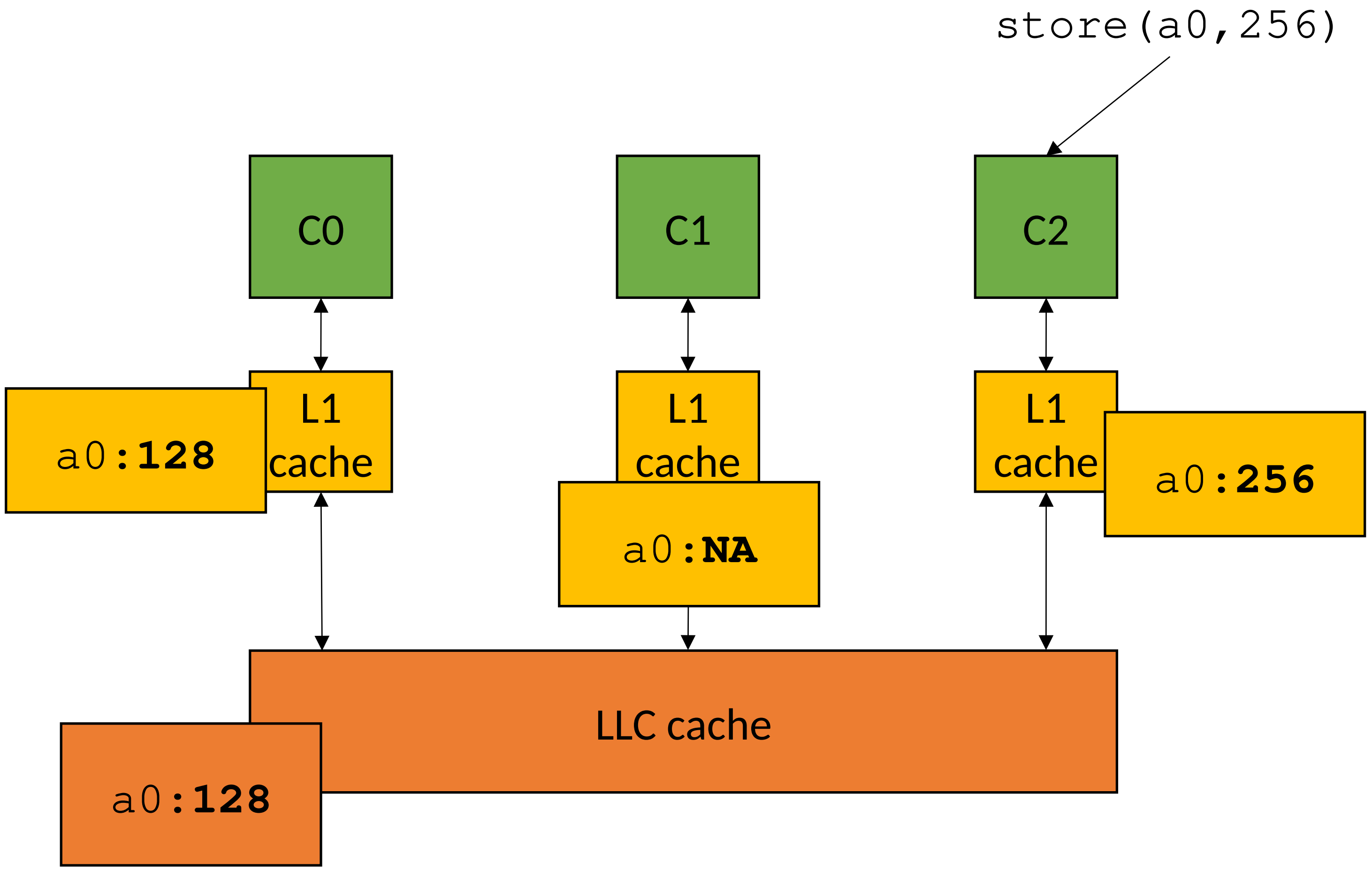
Cache coherence



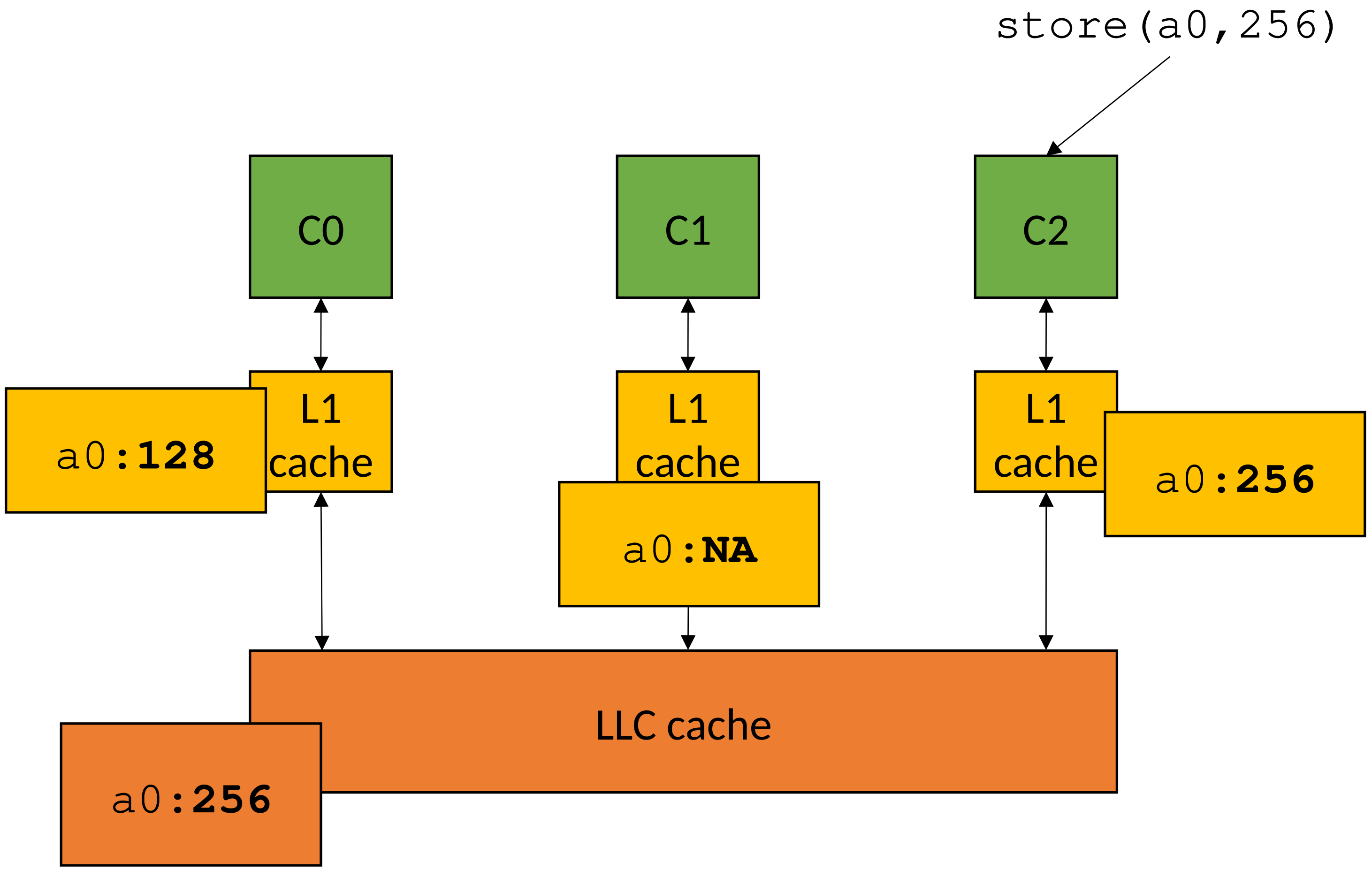
Cache coherence



Cache coherence

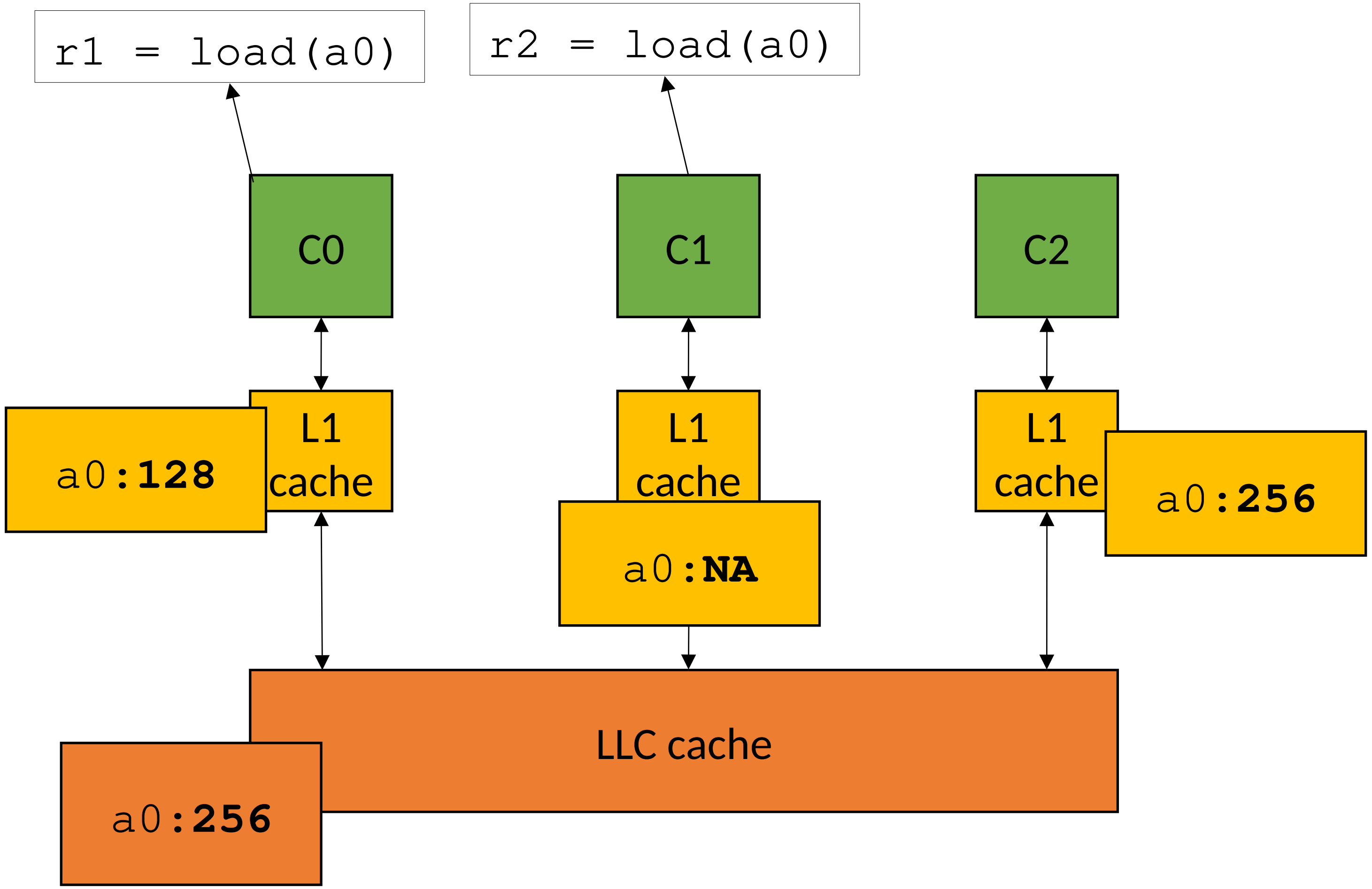


Cache coherence

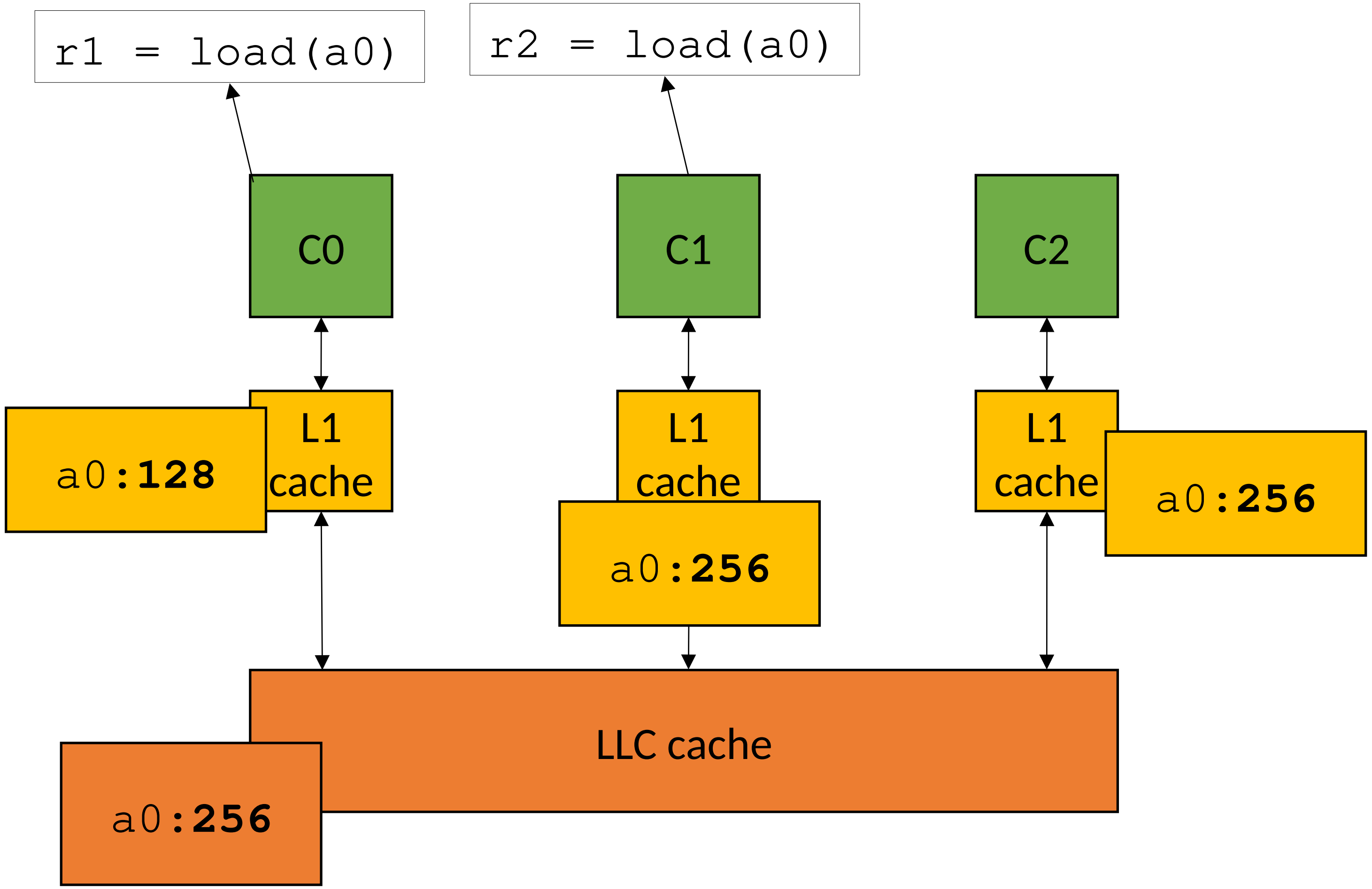


Cache coherence

in parallel

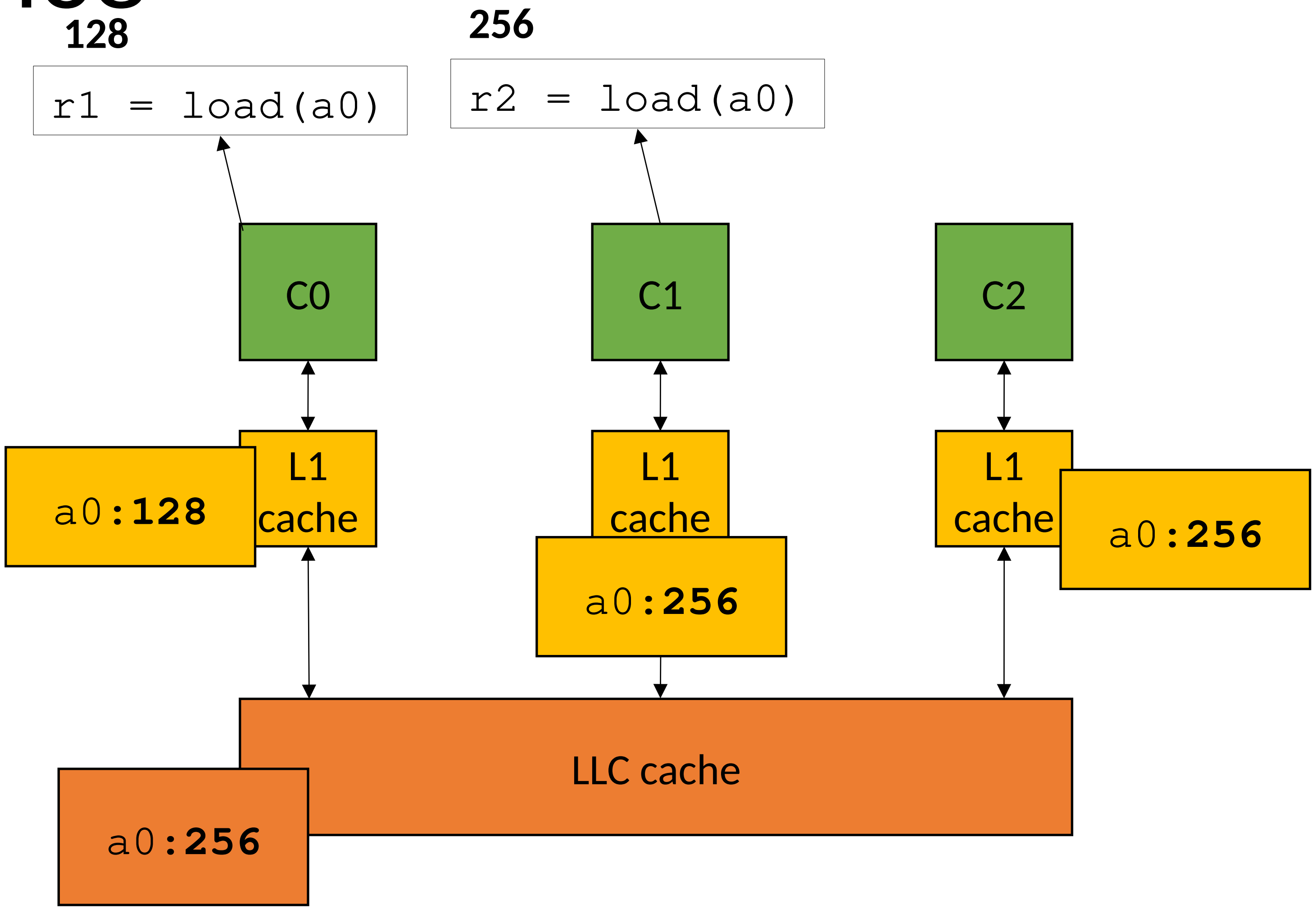


Cache coherence



Cache coherence

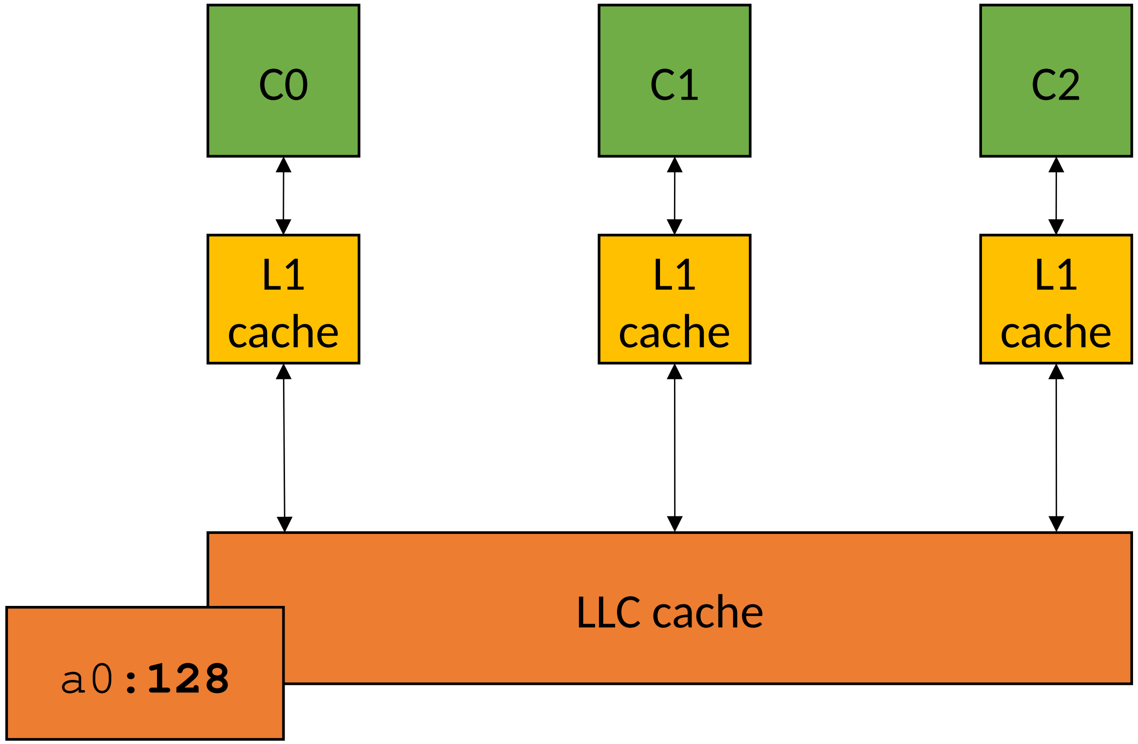
Incoherent view of values!



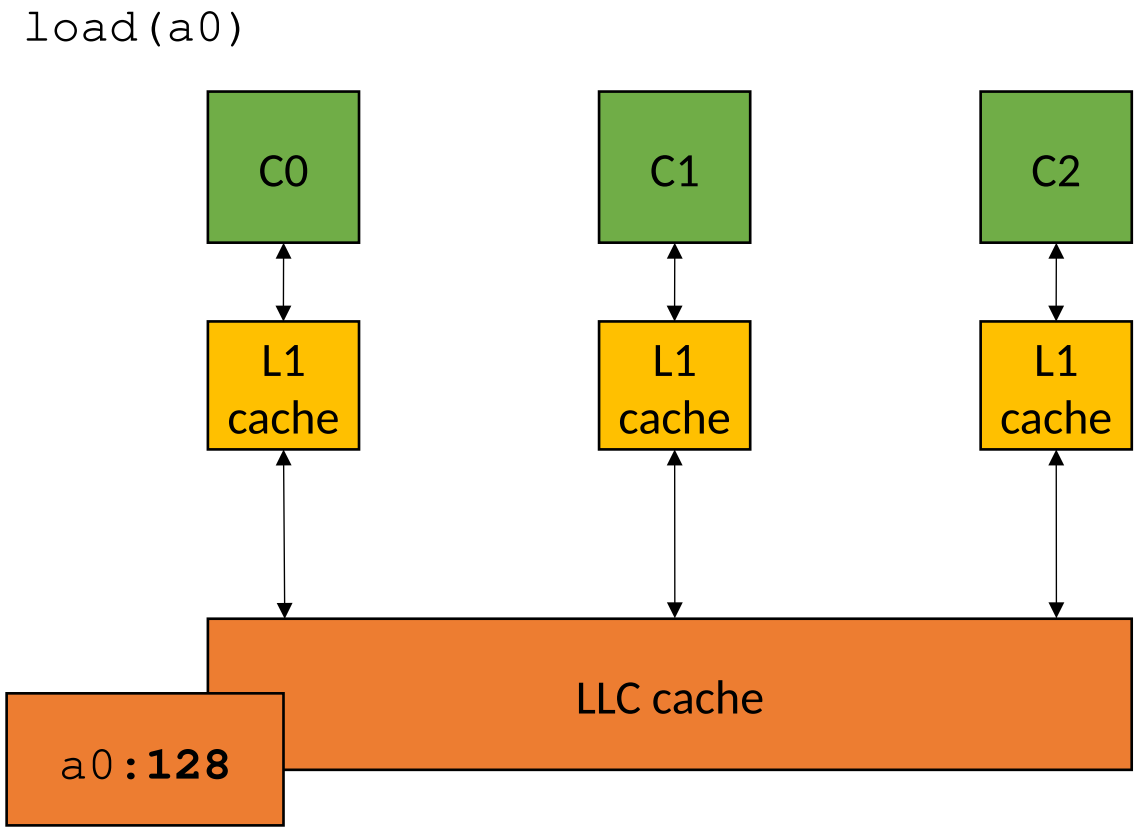
Cache coherence

- MESI protocol
- Cache line can be in 1 of 4 states:
  - **Modified** - the cache contains a modified value and it must be written back to the lower level cache
  - **Exclusive** - only 1 cache has a copy of the value
  - **Shared** - more than 1 cache contains the value, they must all agree on the value
  - **Invalid** - the data is stale and a new value must be fetched from a lower level cache

Cache coherence

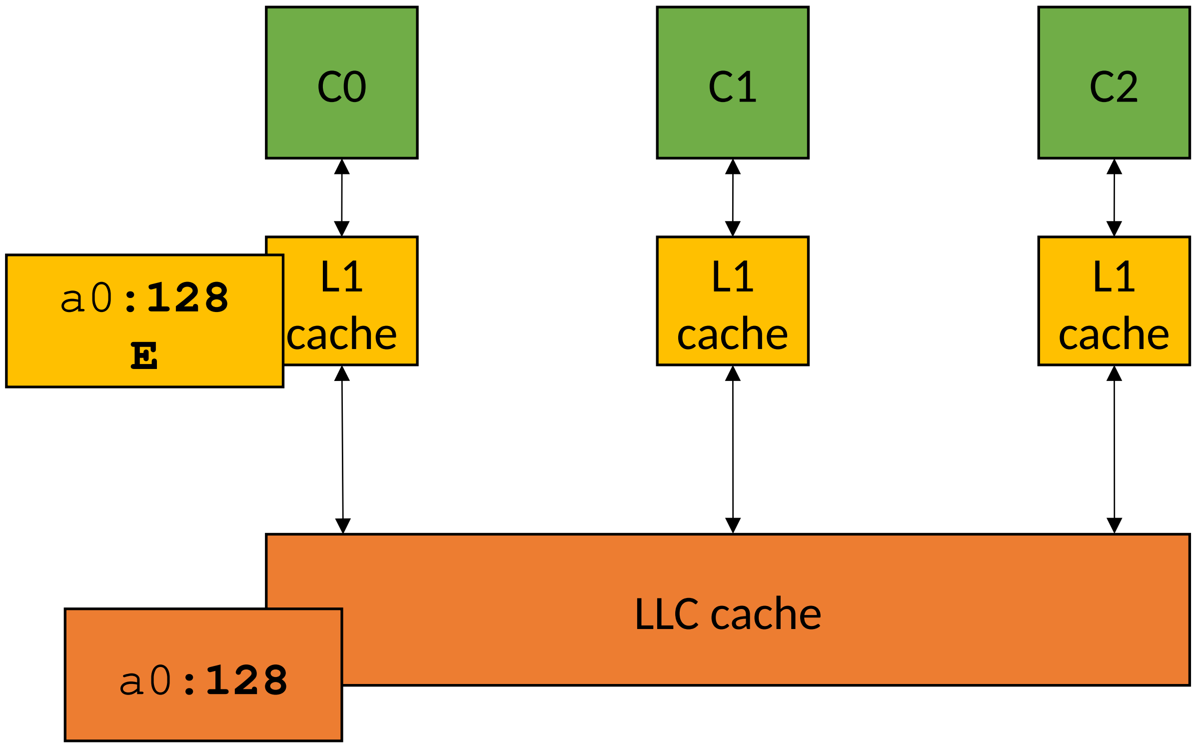


Cache coherence

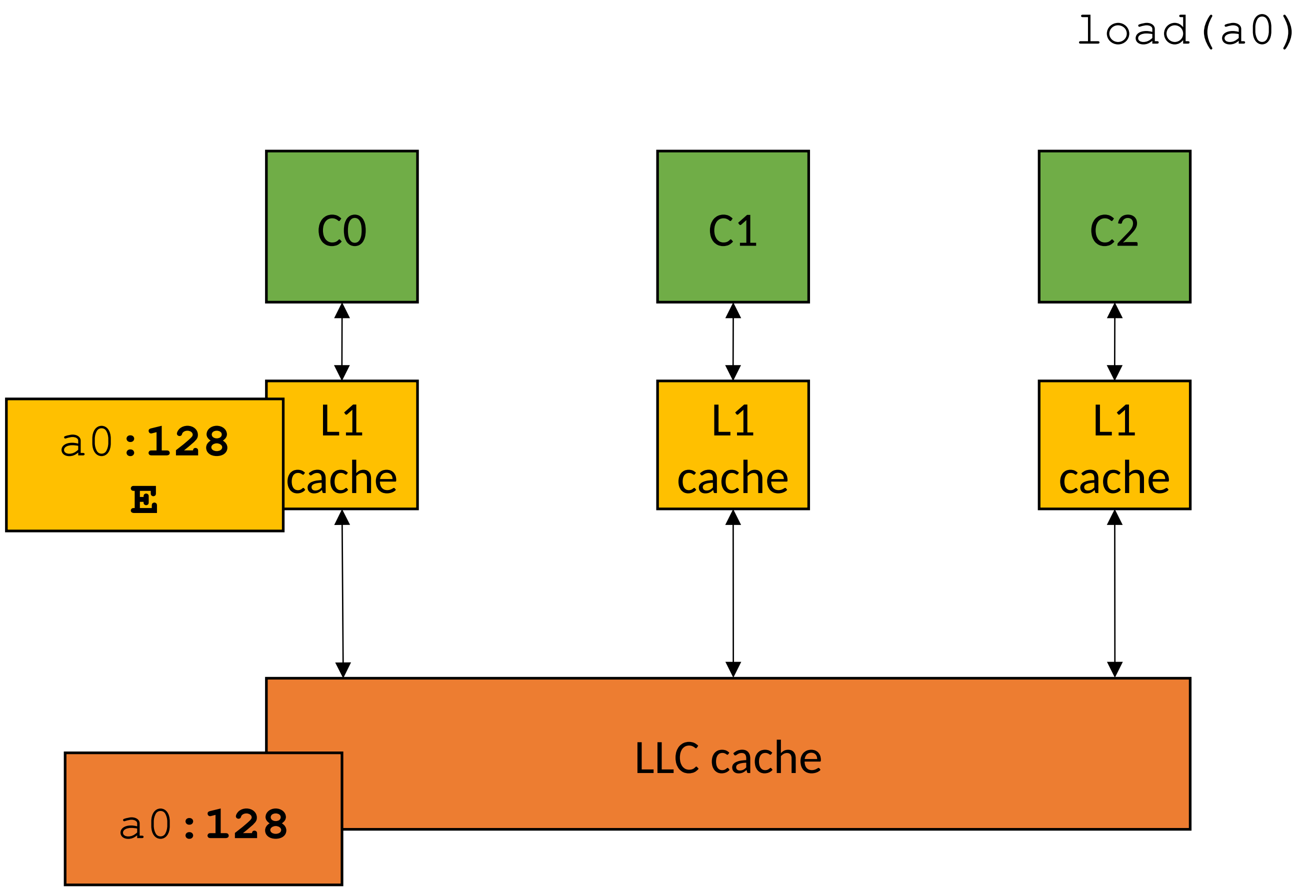


Cache coherence

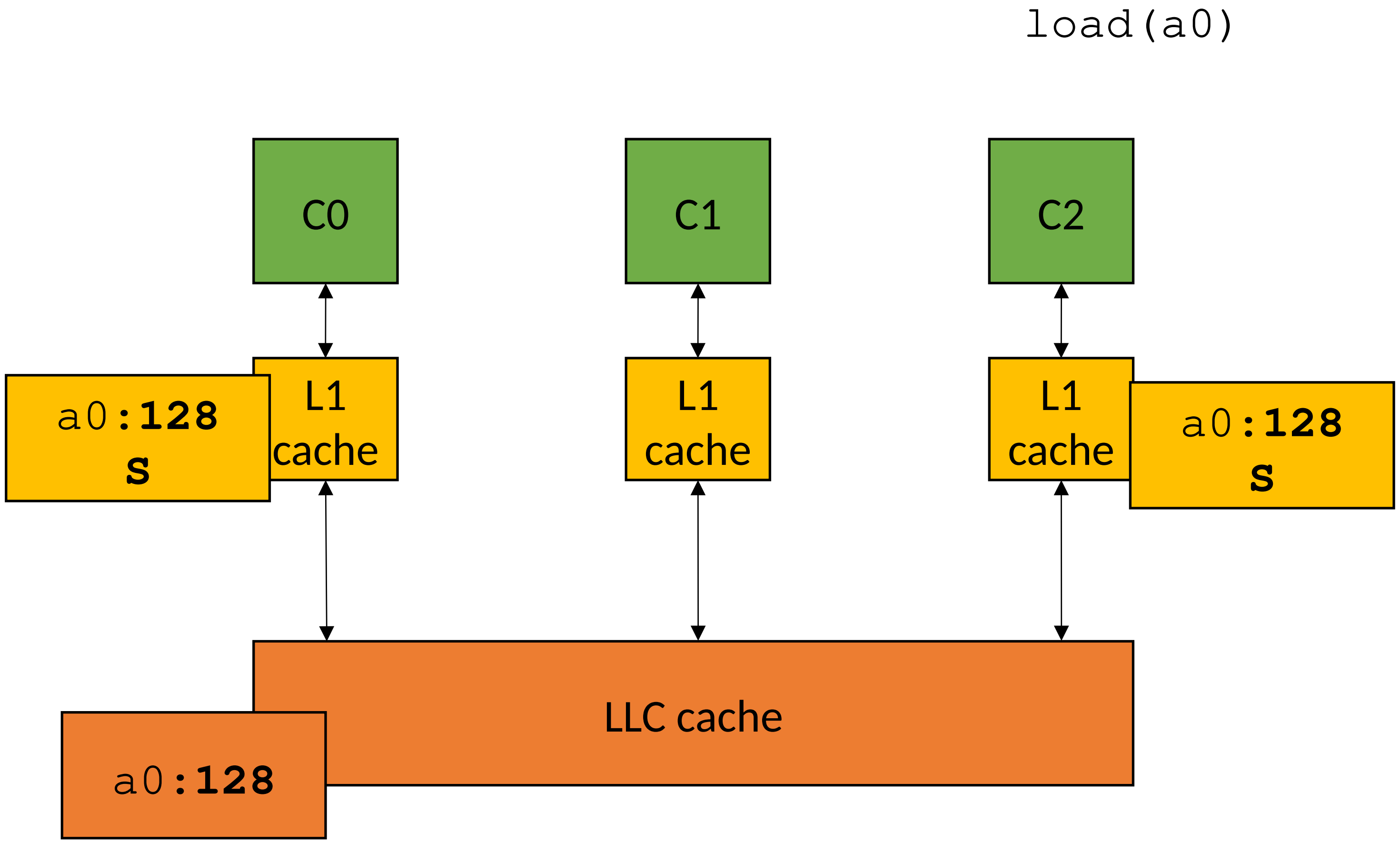
Exclusive states are clean: they match main memory



Cache coherence

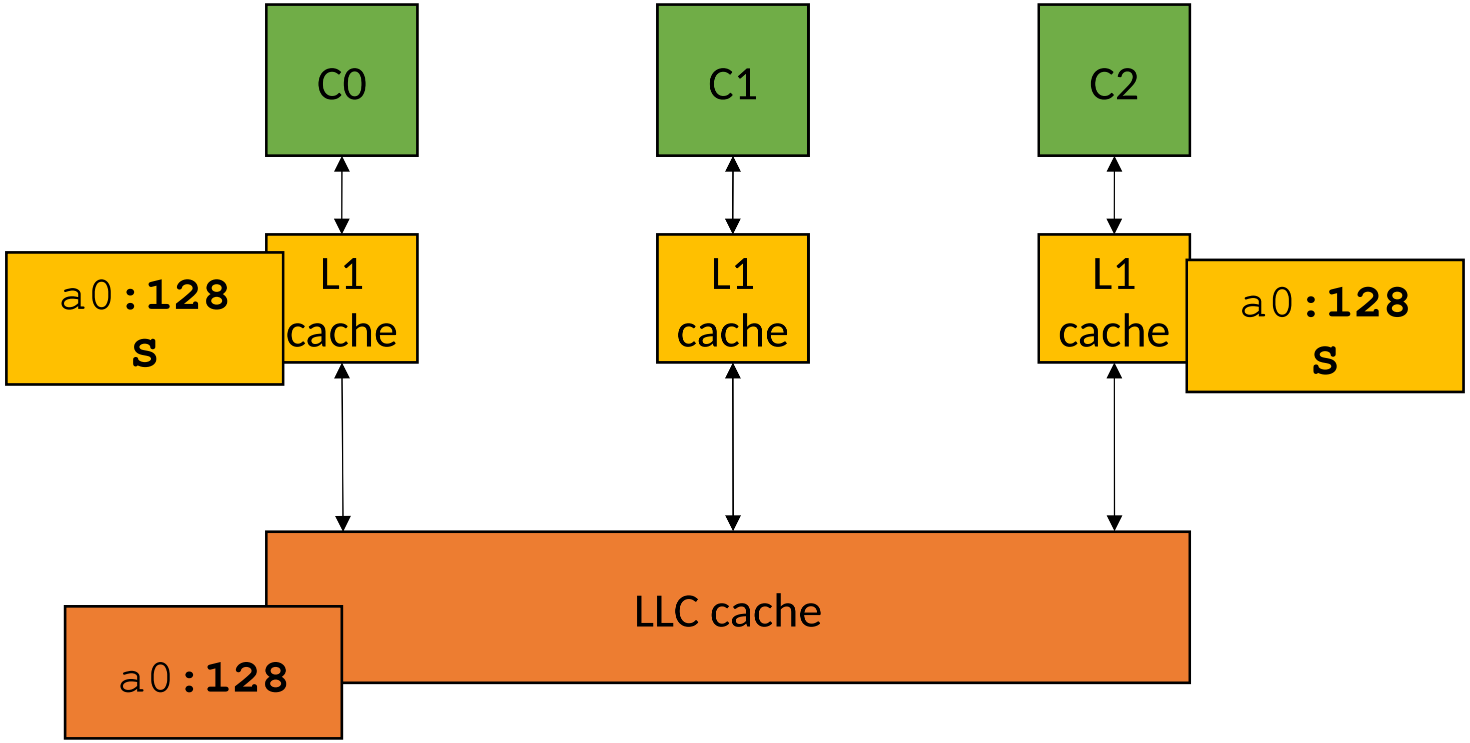


Cache coherence



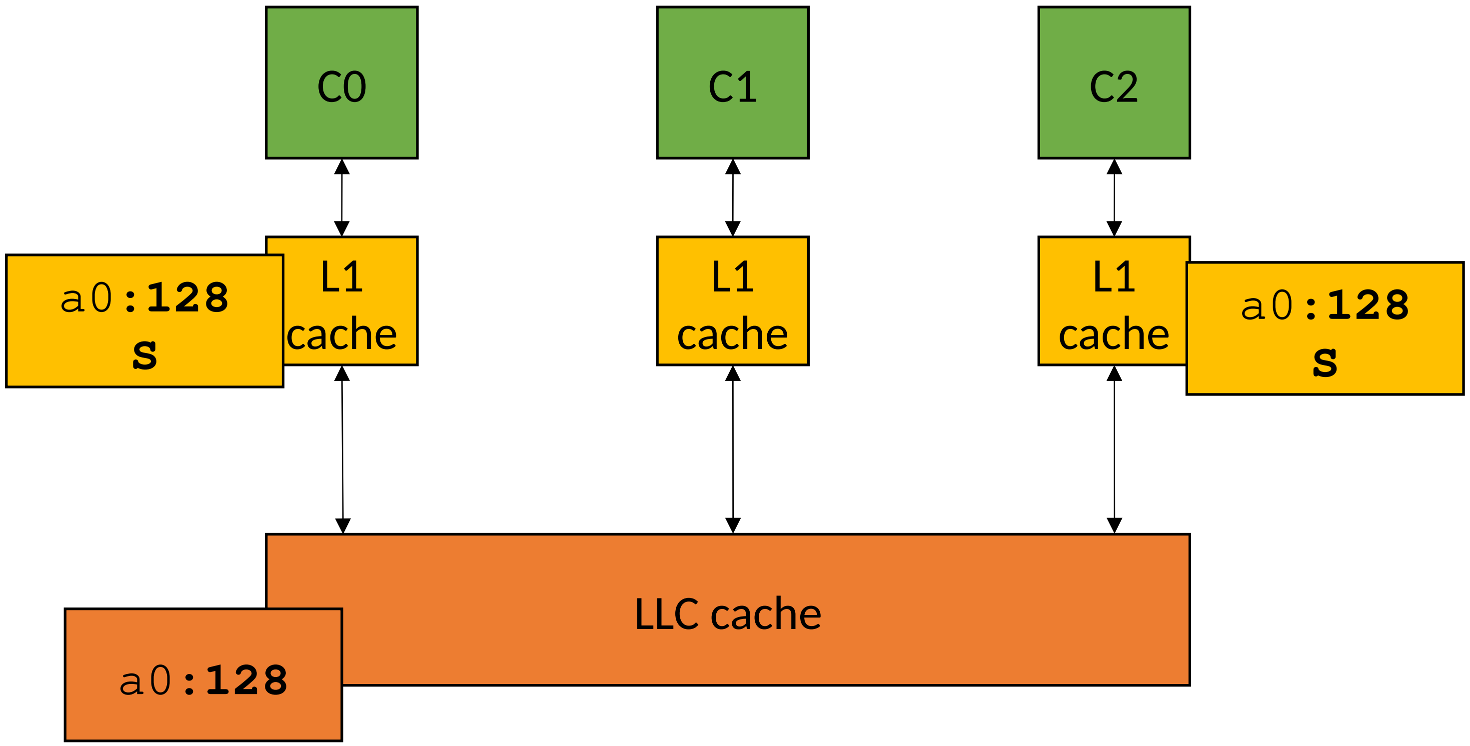
Shared states are clean: they match main memory

Cache coherence



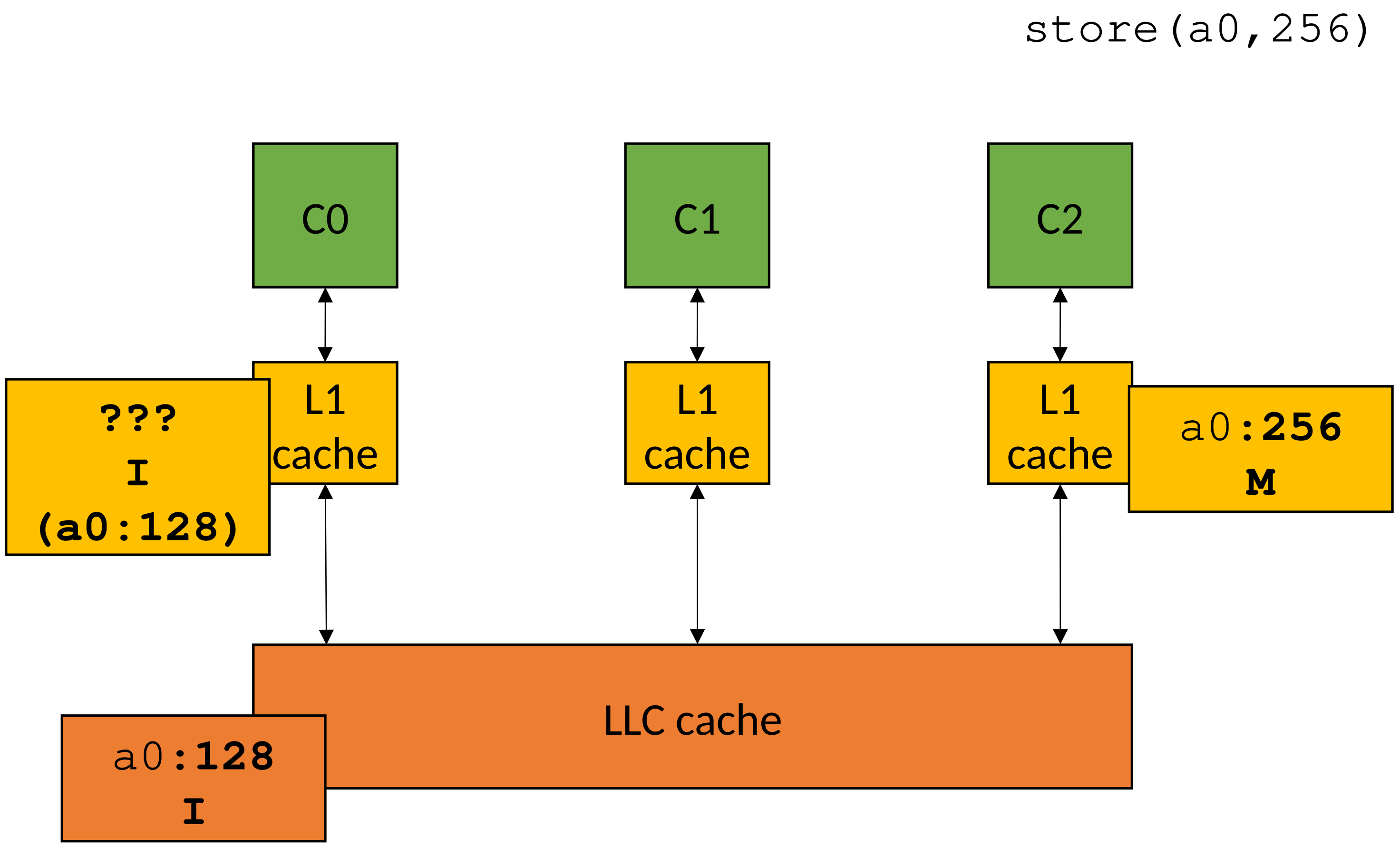
Cache coherence

`store(a0,256)`



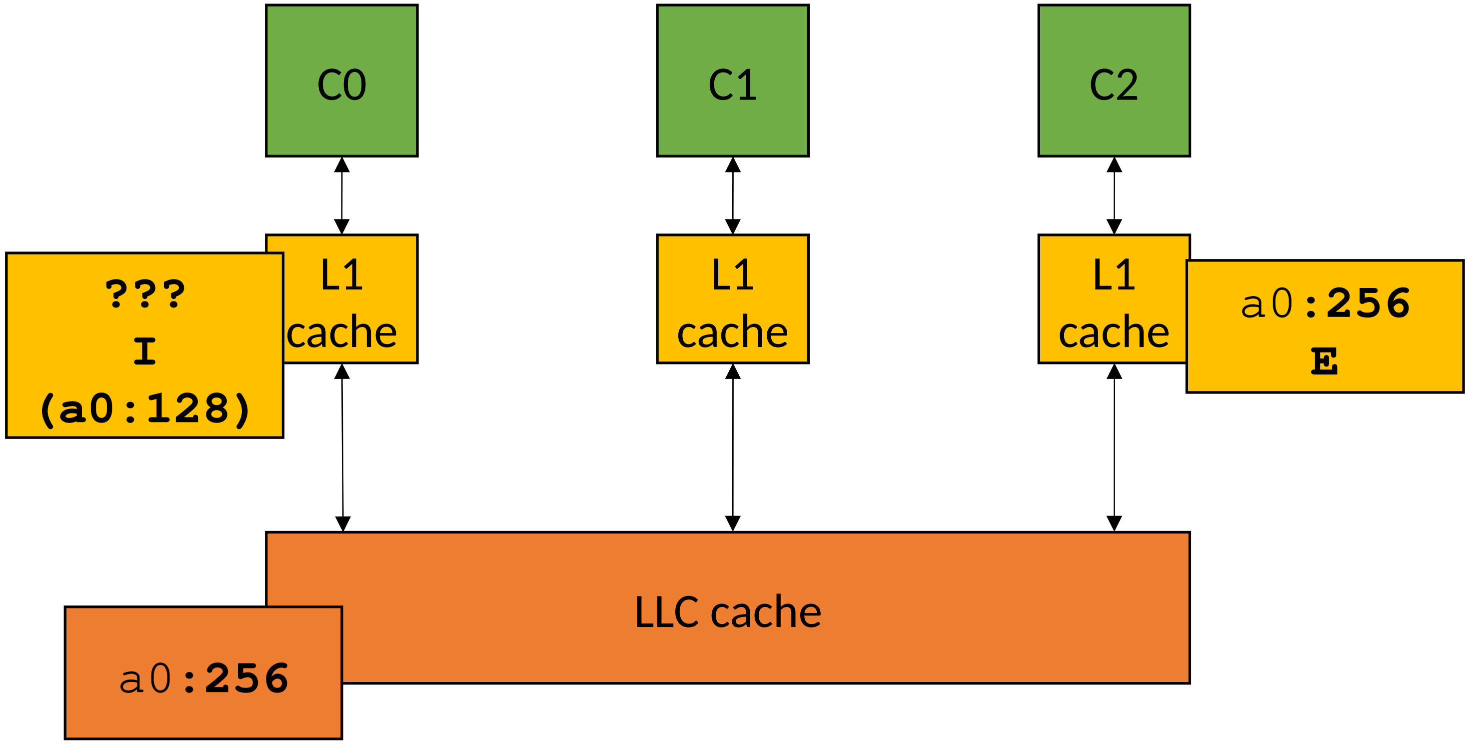
Cache coherence

Modified states are dirty: they don't match main memory

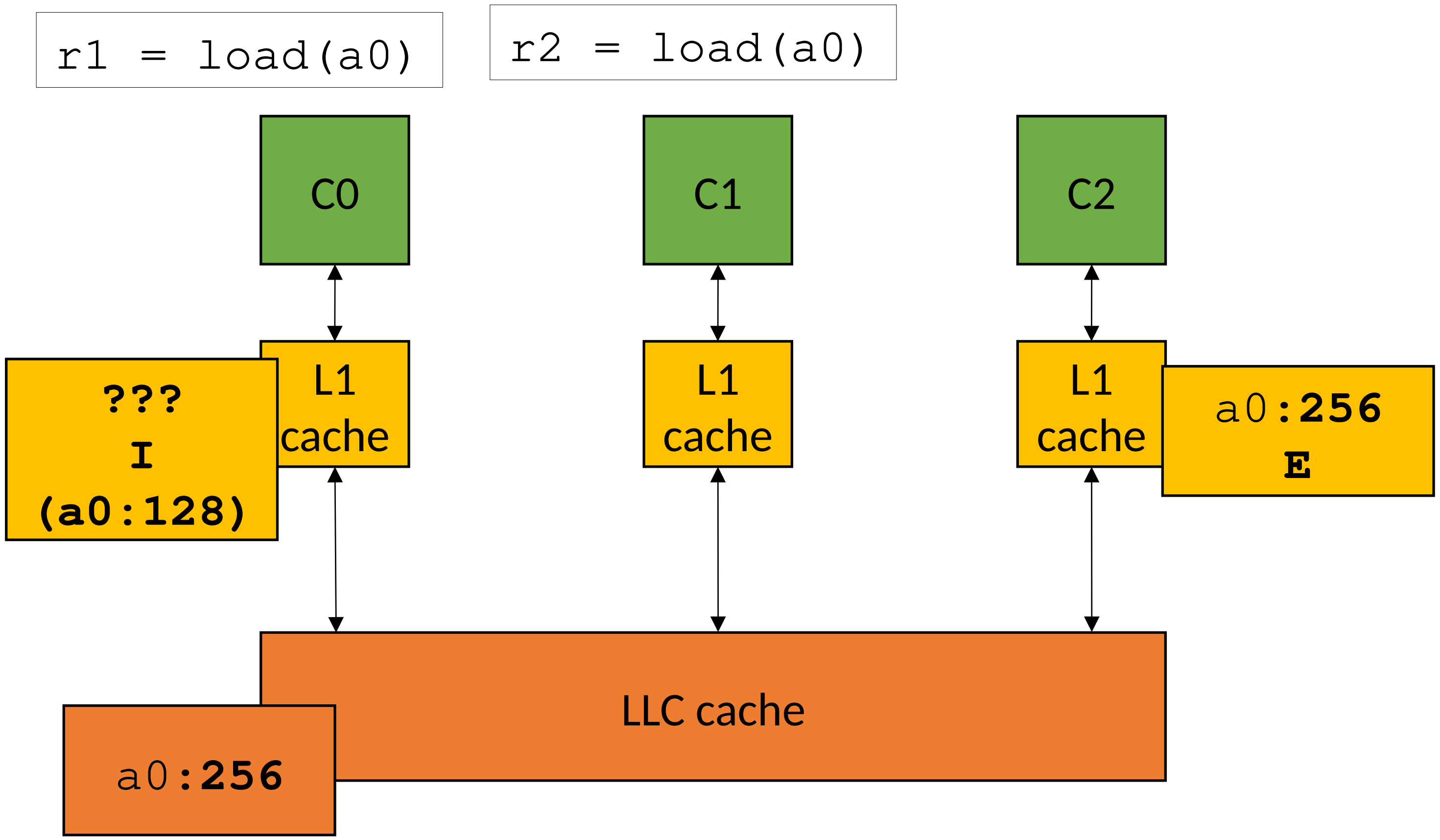


Cache coherence

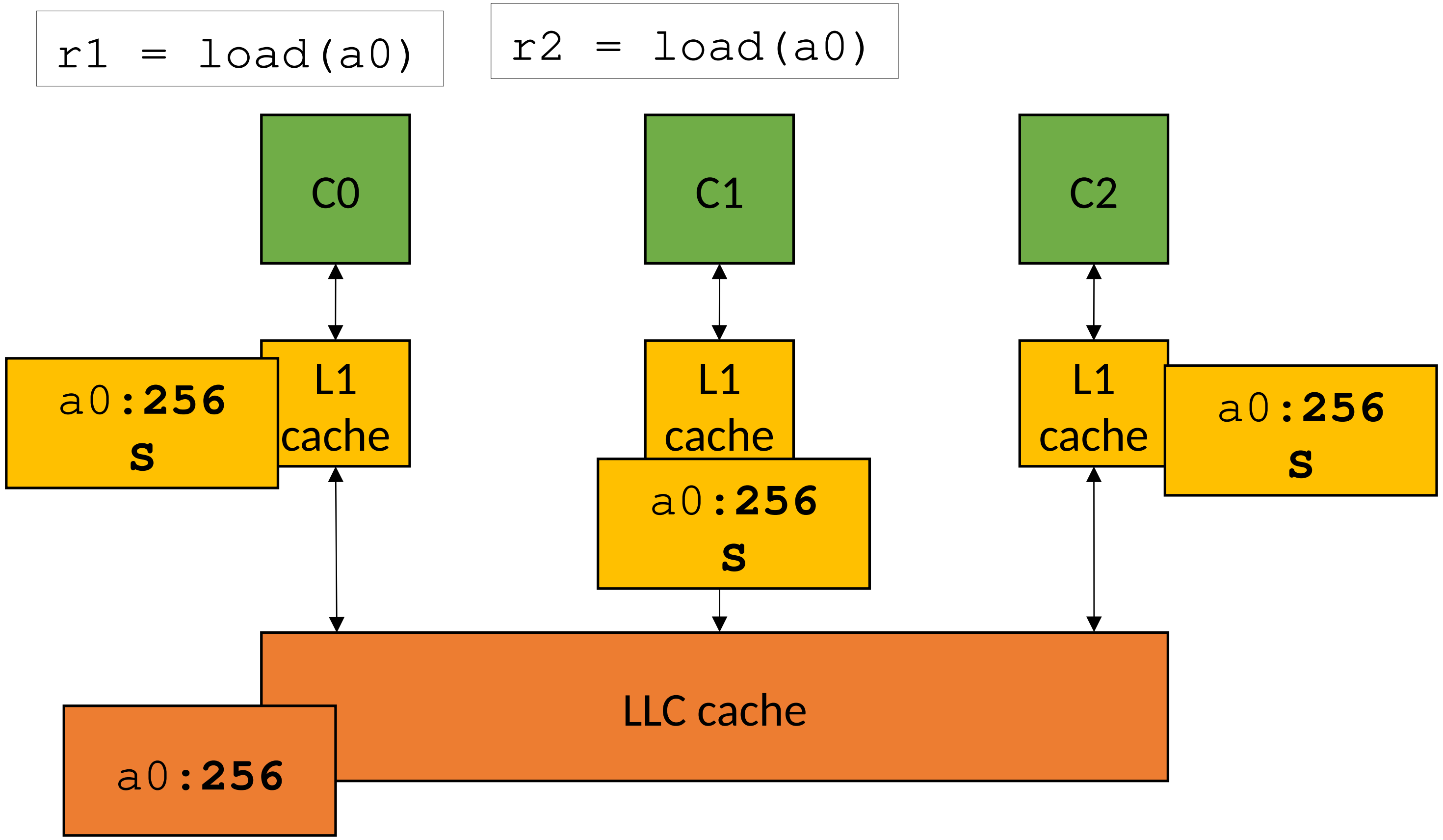
Invalid states
are considered unused



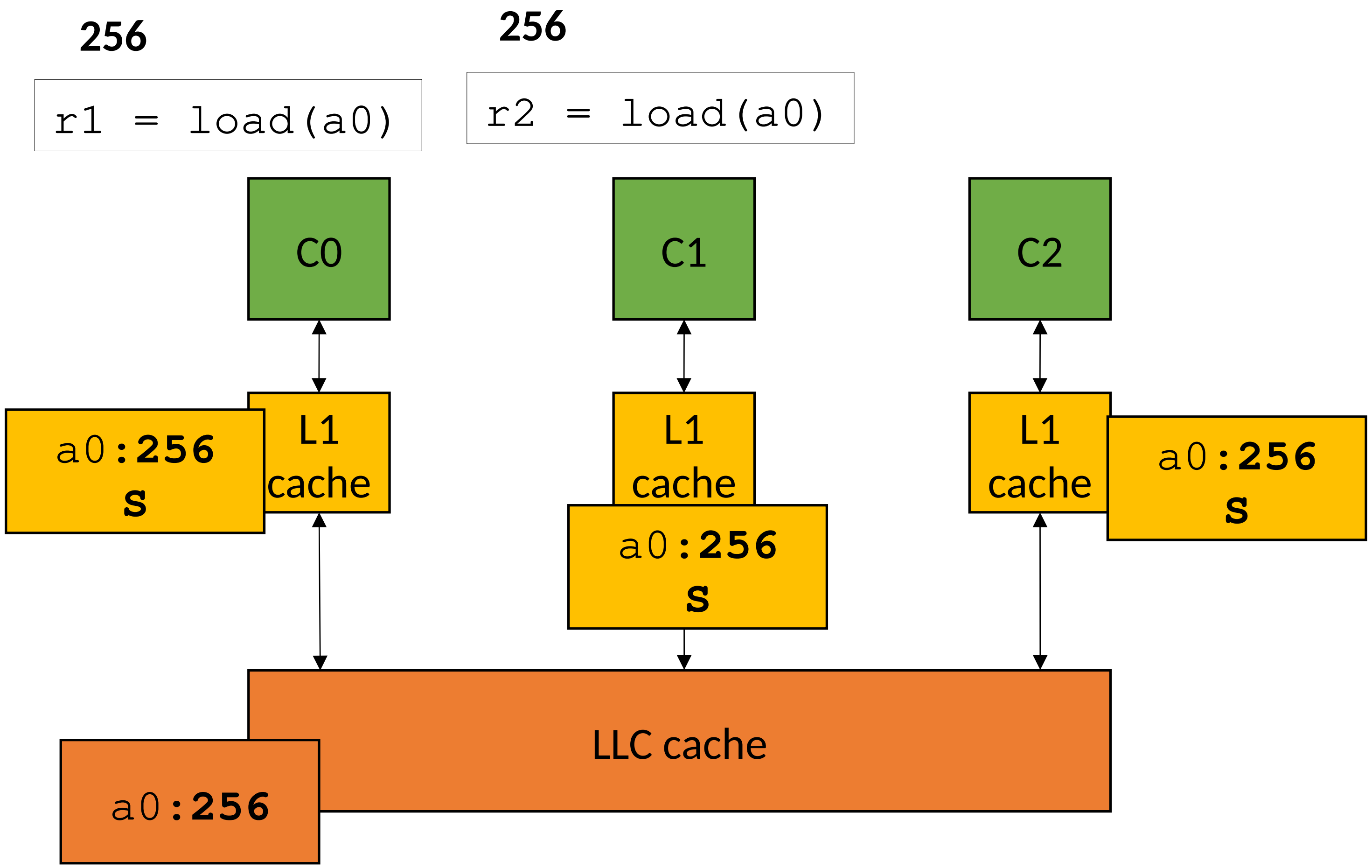
Cache coherence



Cache coherence



Cache coherence



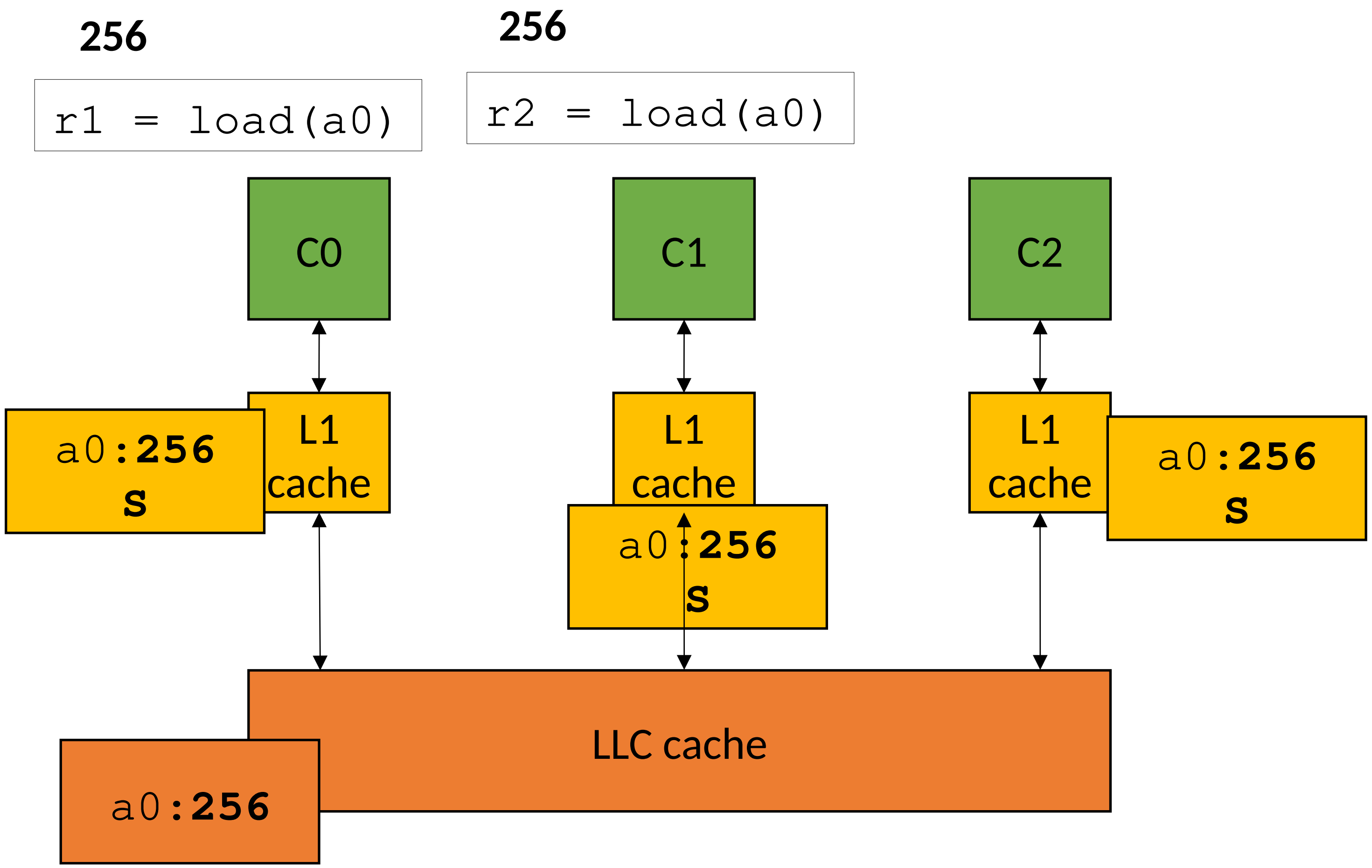
Cache coherence

Takeaways:

Caches must agree on values across cores.

Caches are functionally invisible! Cannot tell with raw input and output

But performance measurements can expose caches, especially if they share the same cache line



C++ Threads

- Introduction
  - Learn as needed throughout class

- Multi-threading officially introduced in C++11
  - only widely available after ~2014
  - official specification
  - cross-platform

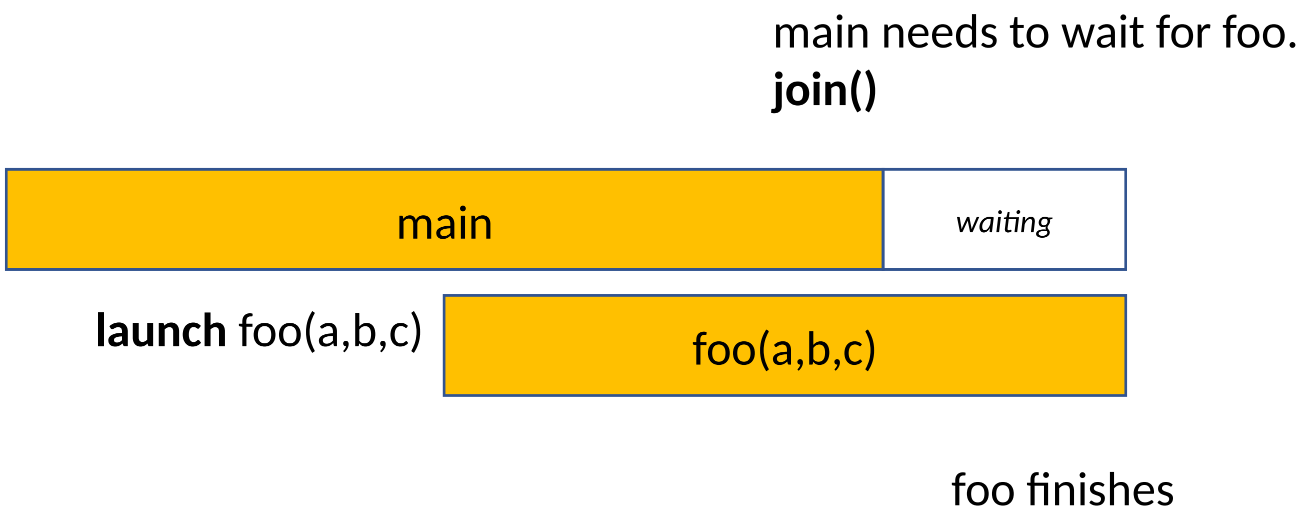
- Before C++ threads
  - pthreads

C++ Threads

- Main idea:
  - run functions concurrently

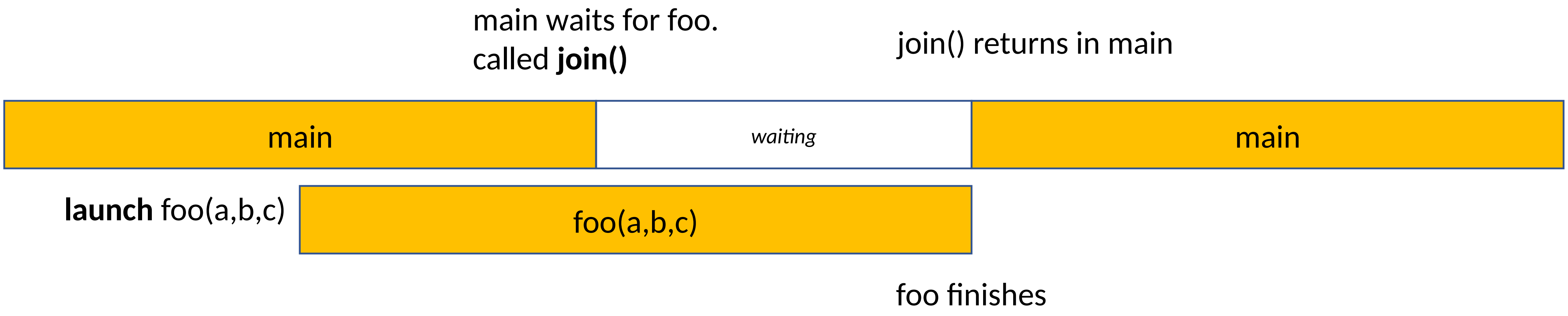
C++ Threads

- Main idea:
  - run functions concurrently

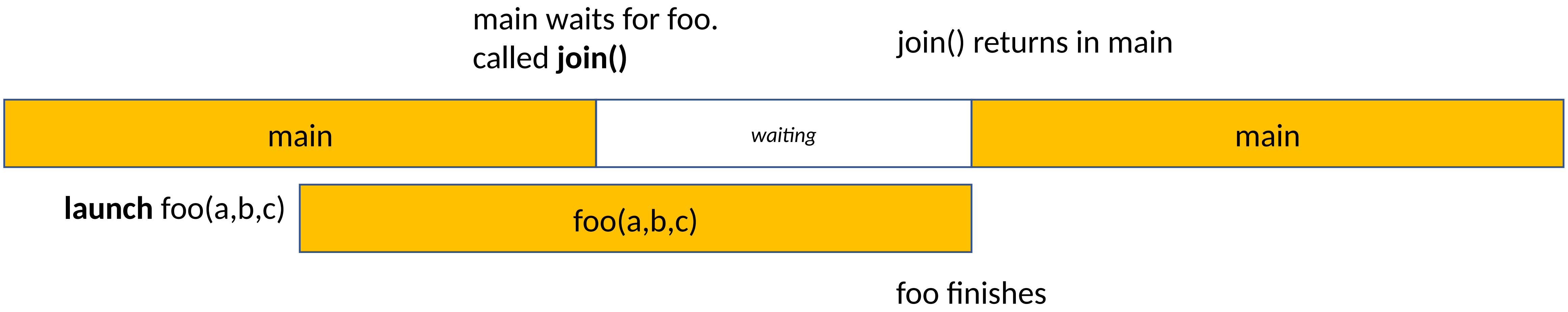


C++ Threads

- Main idea:
  - run functions concurrently

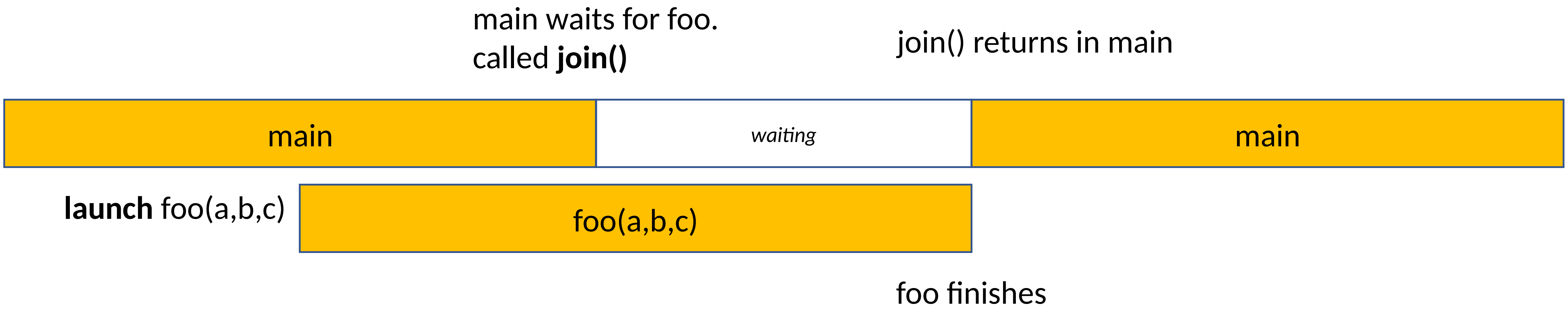


```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```



```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
   // some foo code
}

int main() {
   // some main code
   thread thread_handle (foo,1,2,3);
   // code here runs concurrently with foo
   thread_handle.join();
   return 0;
}
```

header and namespace

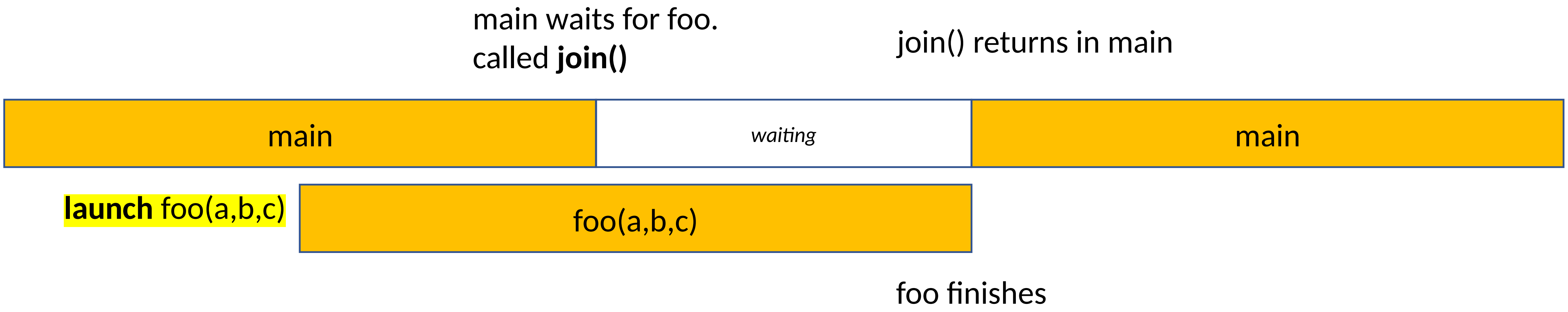


```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

Launches a concurrent thread that executes foo

Stores a handle in thread_handle (don't lose the handle!)

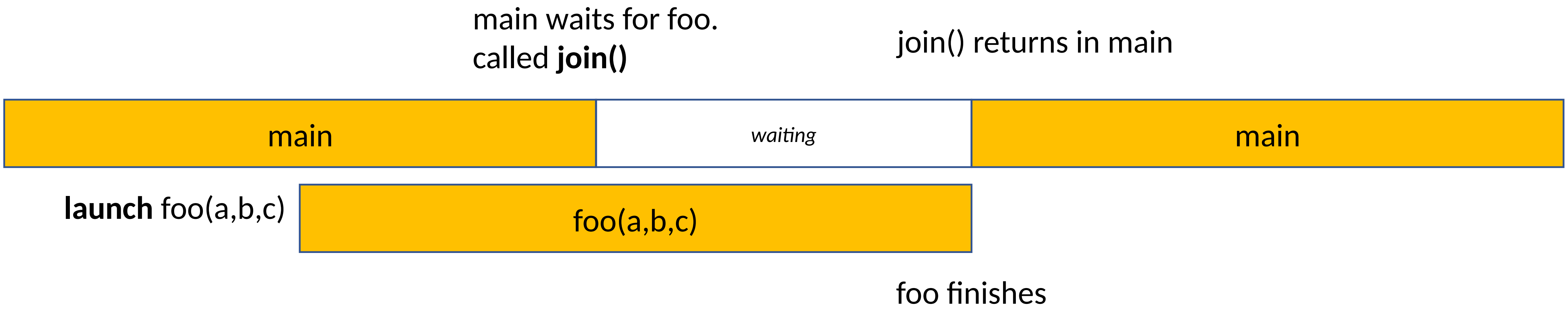
constructor takes in the function, and all arguments



```
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

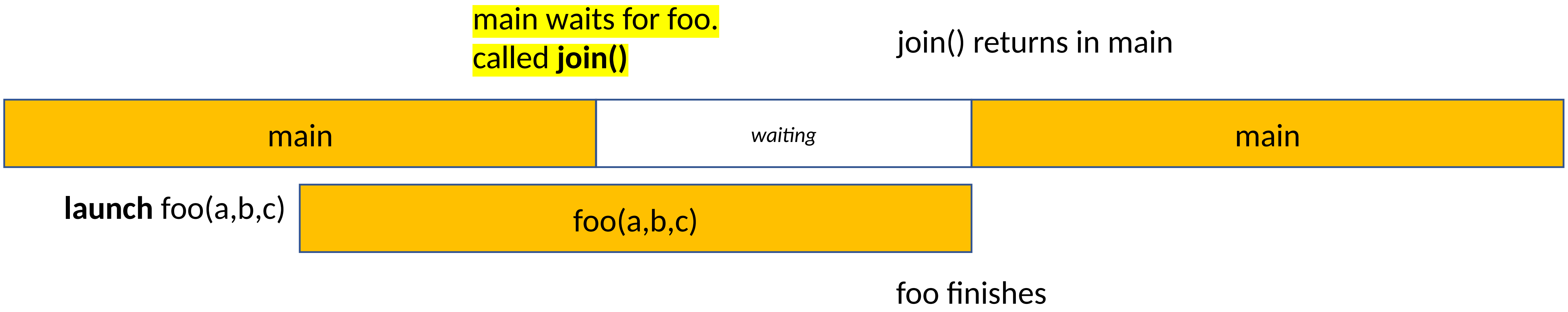
Requires C++14

clang++ -std=c++14 main.cpp



```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

calling join() on the thread handle will cause main to wait for the thread launched with thread_handle to finish.

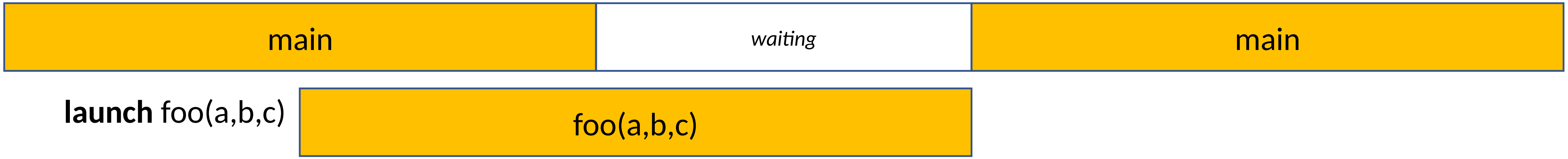


```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

After foo finishes,
main starts executing again

main waits for foo.
called **join()**

join() returns in main



launch foo(a,b,c)

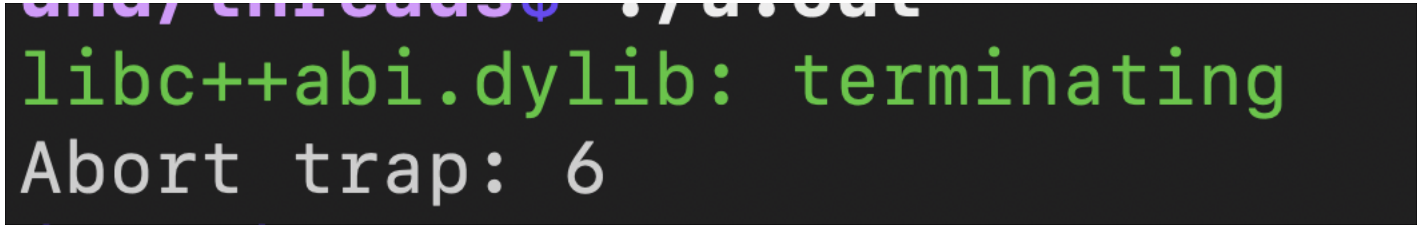
foo finishes

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

What happens if you don't join your threads?

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
   // some foo code
}

int main() {
   // some main code
   thread thread_handle (foo,1,2,3);
   // code here runs concurrently with foo
   thread_handle.join();
   return 0;
}
```

What happens if you don't join your threads?



JOIN YOUR THREADS!!!

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```

return value?

Doesn't have to be void, but it is ignored

how to get values back from threads?

Pass by address (C++ or C)

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  thread_handle.join();
  cout << ret << endl;
  return 0;
}
```

```cpp
#include <thread>
#include <iostream>
using namespace std;

int c;
void foo(int a, int b) {
  // return a + b;
  c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2);
  // code here runs concurrently with foo
  thread_handle.join();
  cout << c << endl;
  return 0;
}
```

Options

global variable
(don’t do this very often!)

```cpp
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if….

```
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
   // return a + b;
   *c = a + b;
}

int main() {
   // some main code
   int ret = 0;
   thread thread_handle (foo,1,2, &ret);
   // code here runs concurrently with foo
   cout << ret << endl;
   thread_handle.join();
   return 0;
}
```

What if....

Undefined behavior!
Cannot access the same values concurrently without protection!

Next module we will talk protection (locks)

SPMD programming model

- Same program, multiple data

- Main idea: many threads execute the same function, but they operate on different data.

- How do they get different data?
  - each thread can access their own thread id, a contiguous integer starting at 0 up to the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

lets do this in parallel!
each thread increments different
elements in the array

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

The function gets a thread id and the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = 0; i < a_size; i++) {
        a[i]++;
    }
}
```

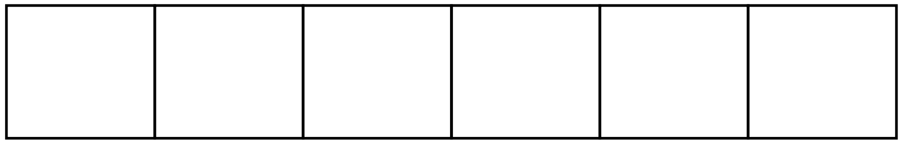
A few options on how to split up the work
lets do round robin

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```



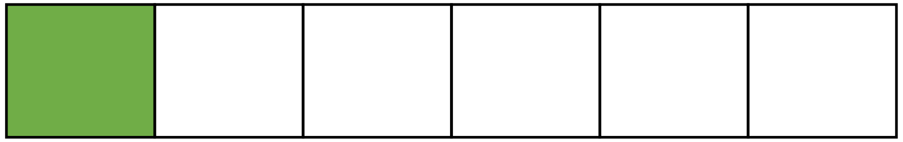
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

iteration 1 computes index 0



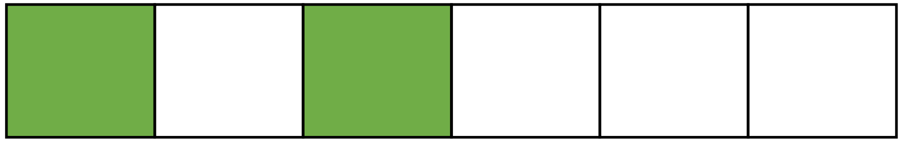
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

iteration 2 computes index 2



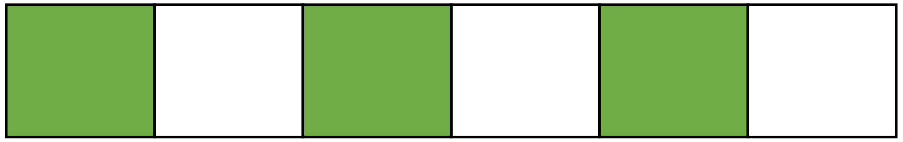
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

iteration 3 computes index 4



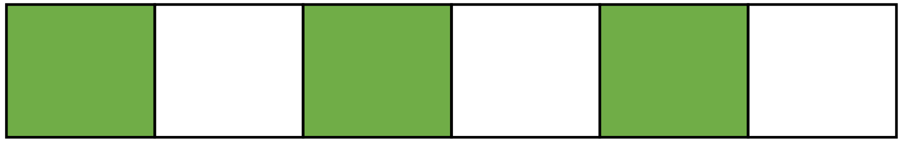
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

switch to thread 1



array a

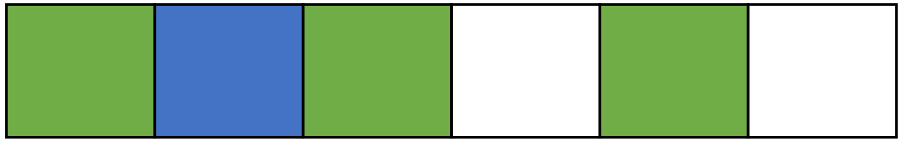
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 1 computes index 1



array a

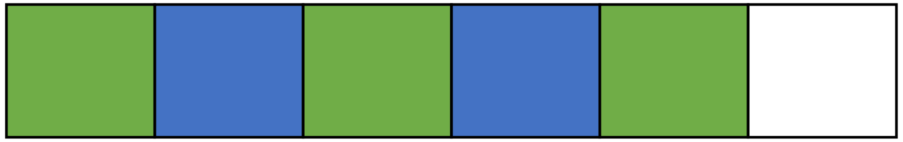
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 2 computes index 3



array a

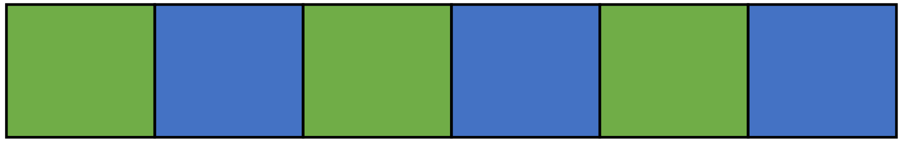
Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

switch to thread 1

iteration 3 computes index 5



array a

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads);


#define THREADS 8
#define A_SIZE 1024
int main() {
  int *a = new int[A_SIZE];
  // initialize a
  thread thread_ar[THREADS];
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i] = thread(increment_array, a, A_SIZE, i, THREADS);
  }
  for (int i = 0; i < THREADS; i++) {
    thread_ar[i].join();
  }
  delete[] a;
  return 0;
}
```

Extra if time

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input*, *produces an output*

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input*, *produces an output*

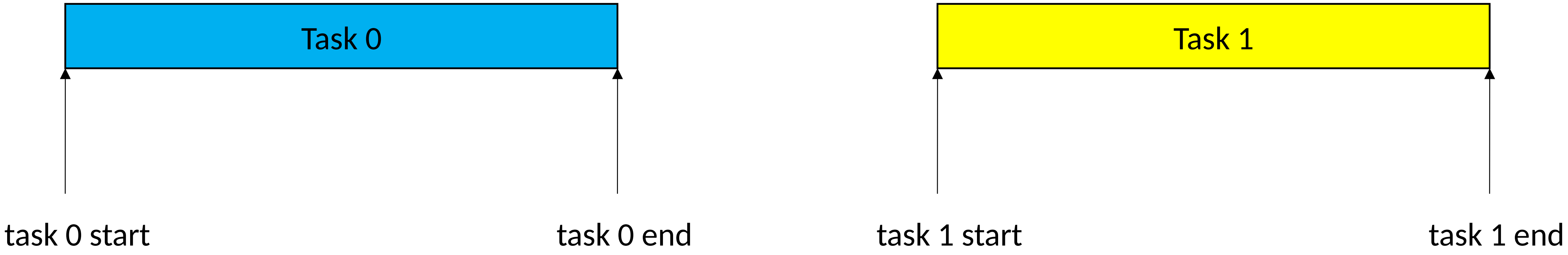
- Concrete tasks:
  - Application (e.g. Spotify and Chrome)
  - Function
  - Loop iterations
  - Individual instructions
  - Circuit level?

coarse

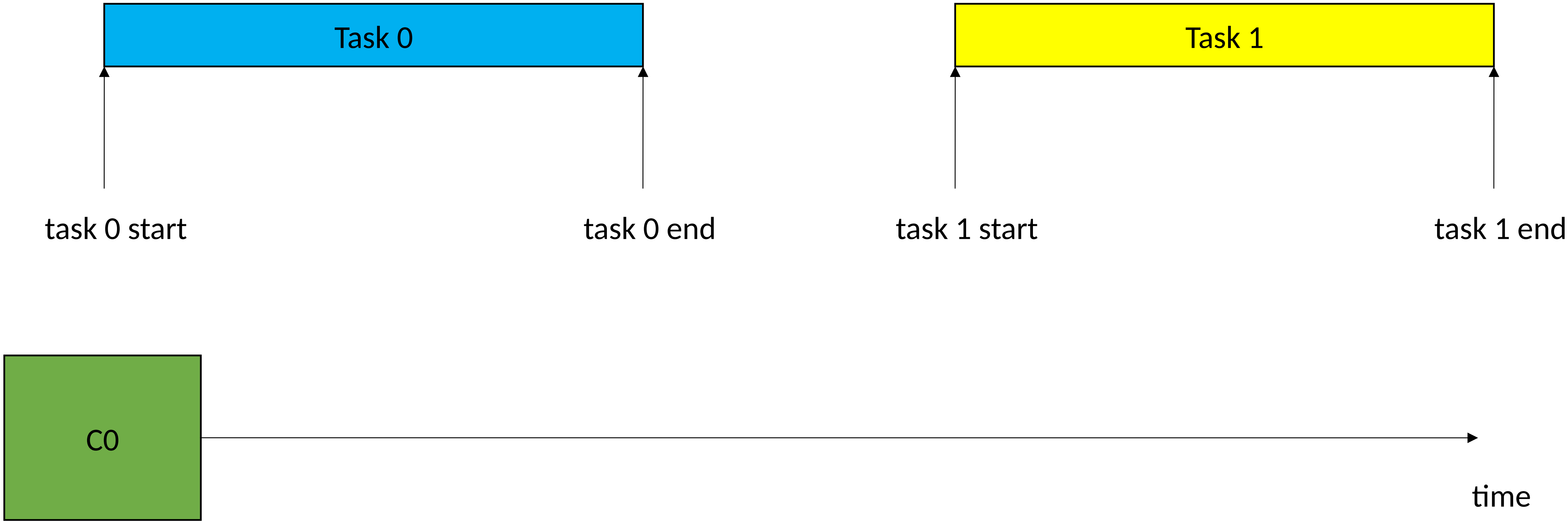
granularity

fine

Concurrency vs. Parallelism

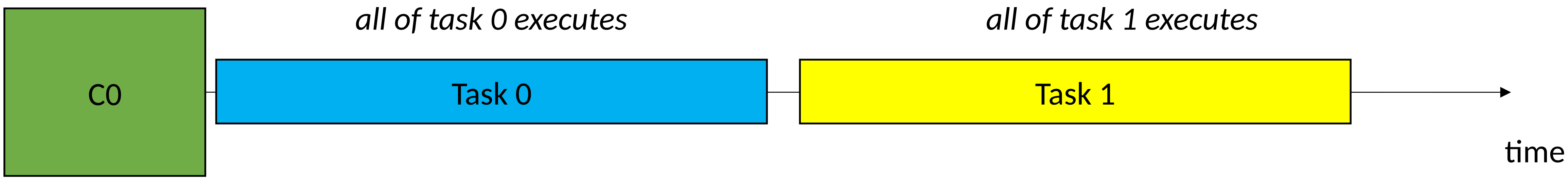


Concurrency vs. Parallelism



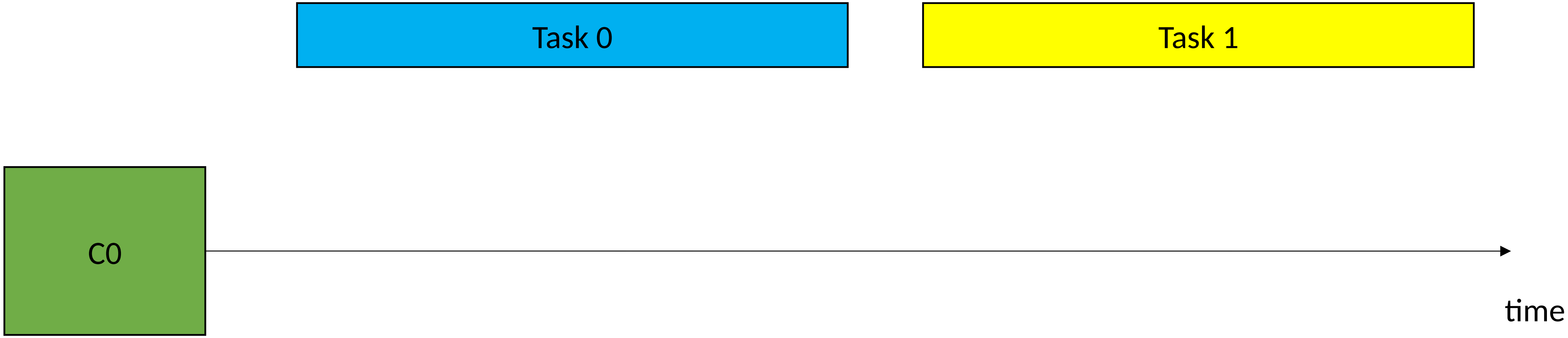
Concurrency vs. Parallelism

Sequential execution
Not concurrent or parallel



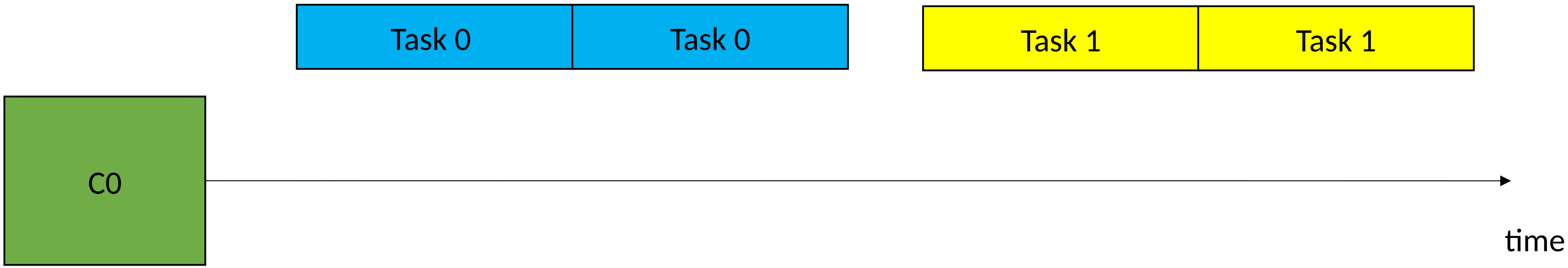
Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

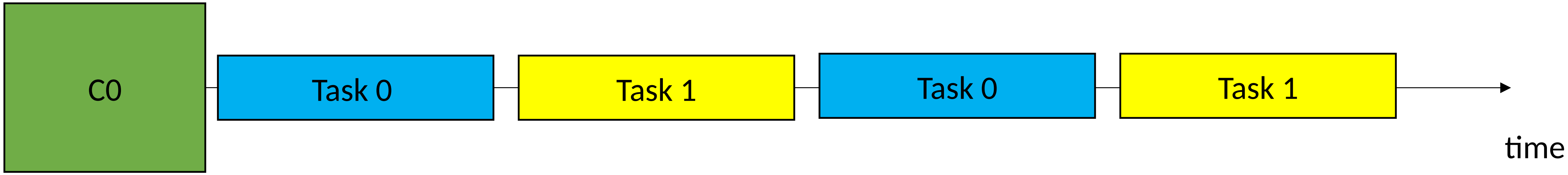
The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)

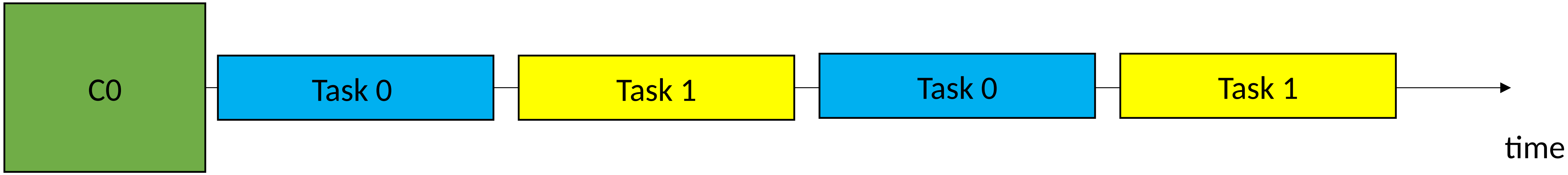
tasks are interleaved on the same processor



Concurrency vs. Parallelism

- Definition:
  - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

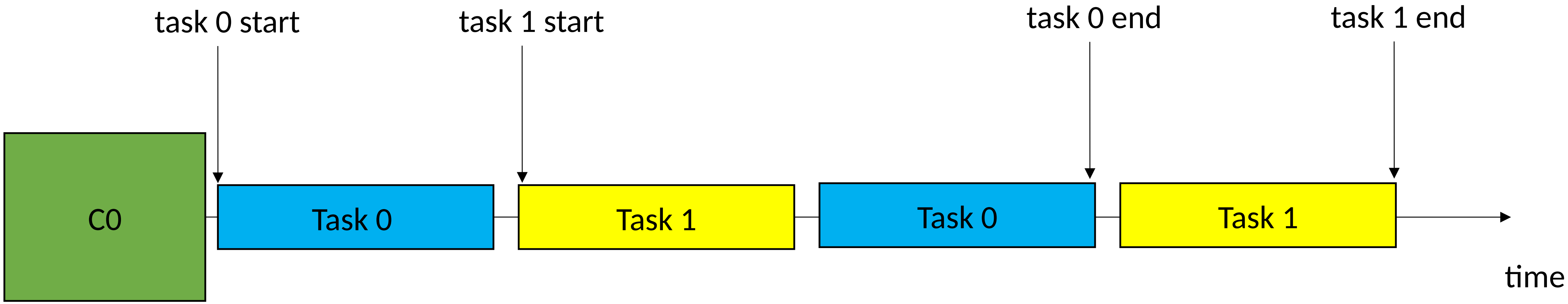
The OS can preempt a thread
(remove it from the hardware resource)



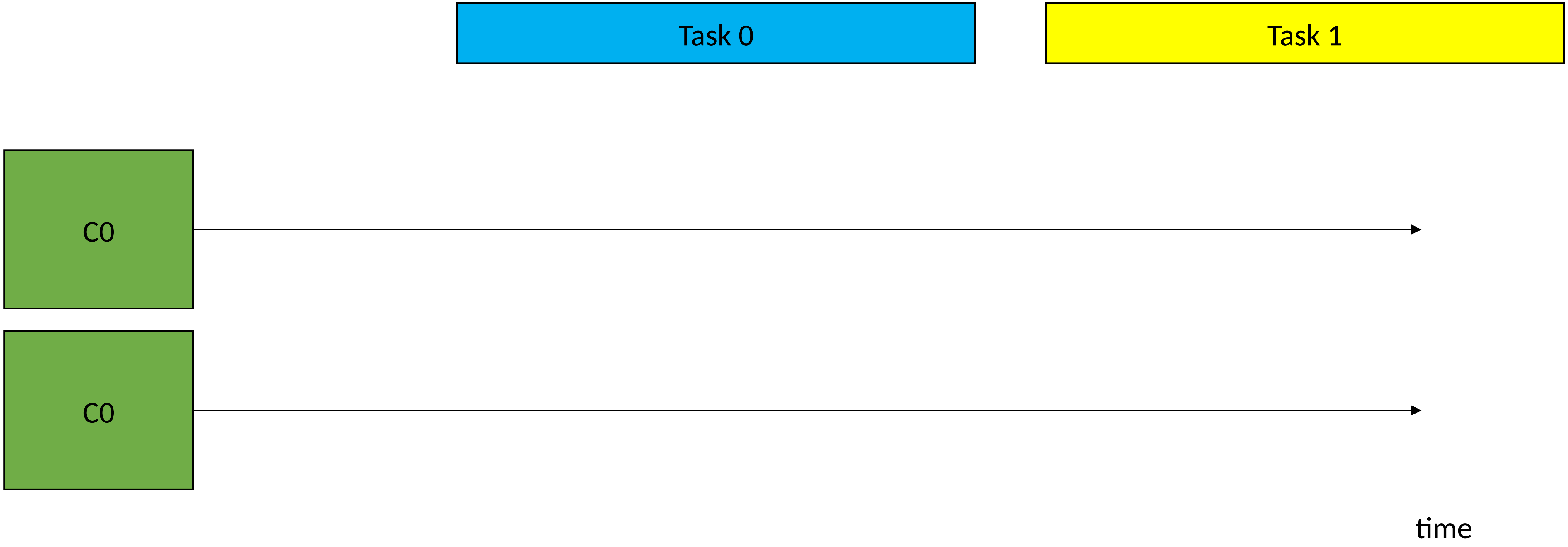
Concurrency vs. Parallelism

- Definition:
  - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

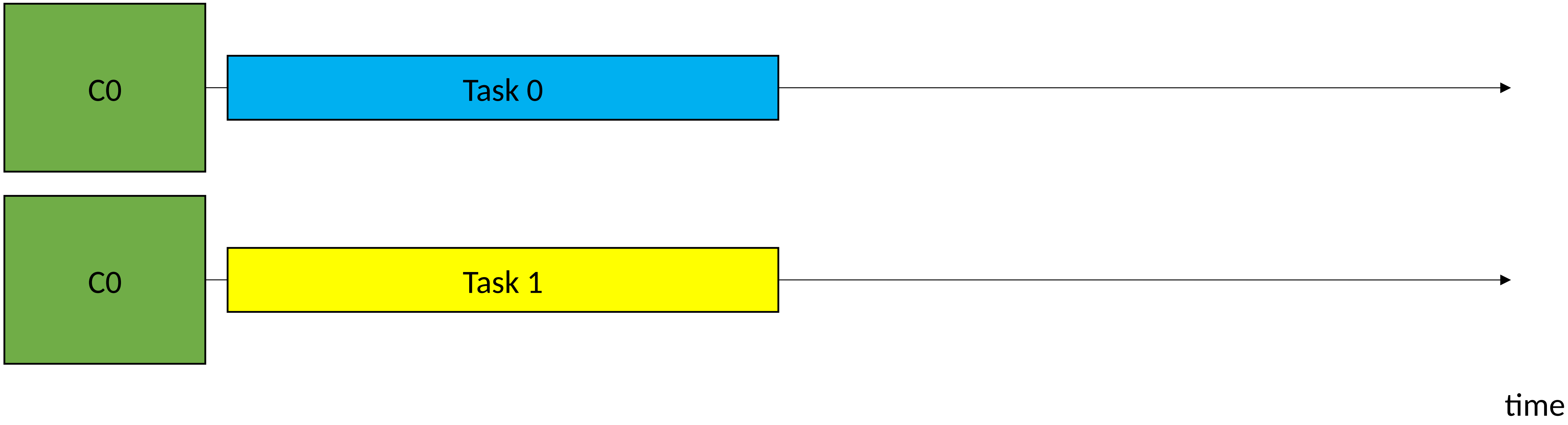
The OS can preempt a thread
(remove it from the hardware resource)



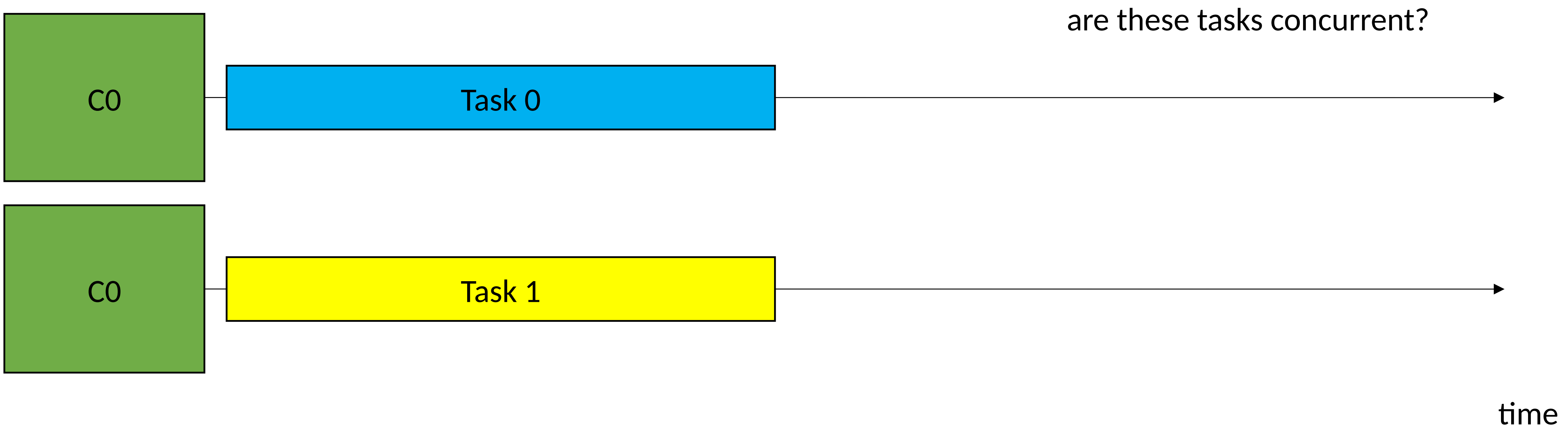
Concurrency vs. Parallelism



Concurrency vs. Parallelism

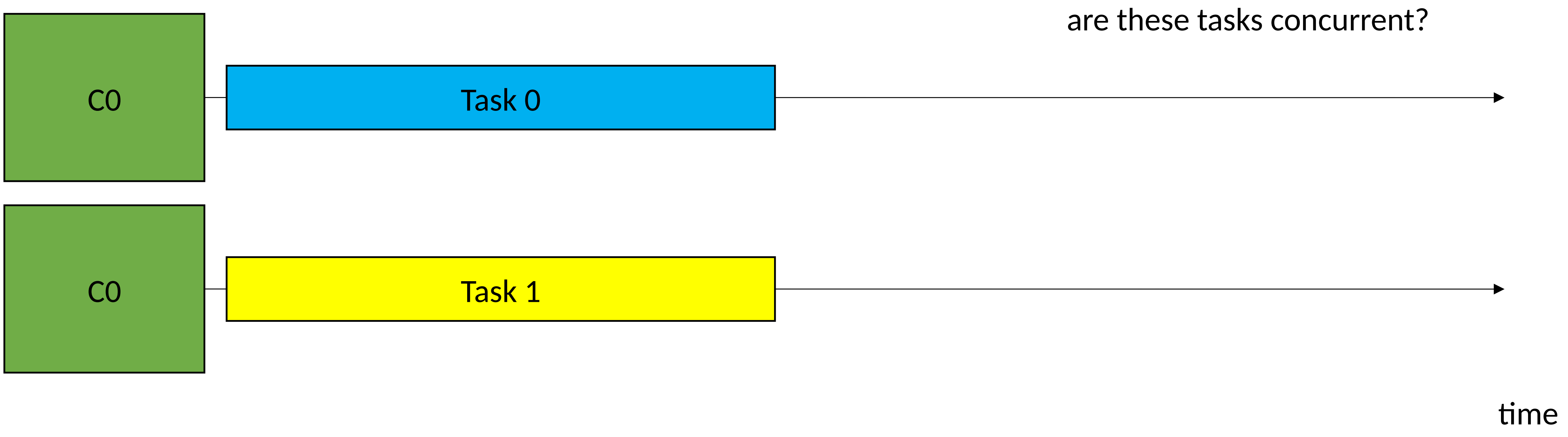


Concurrency vs. Parallelism

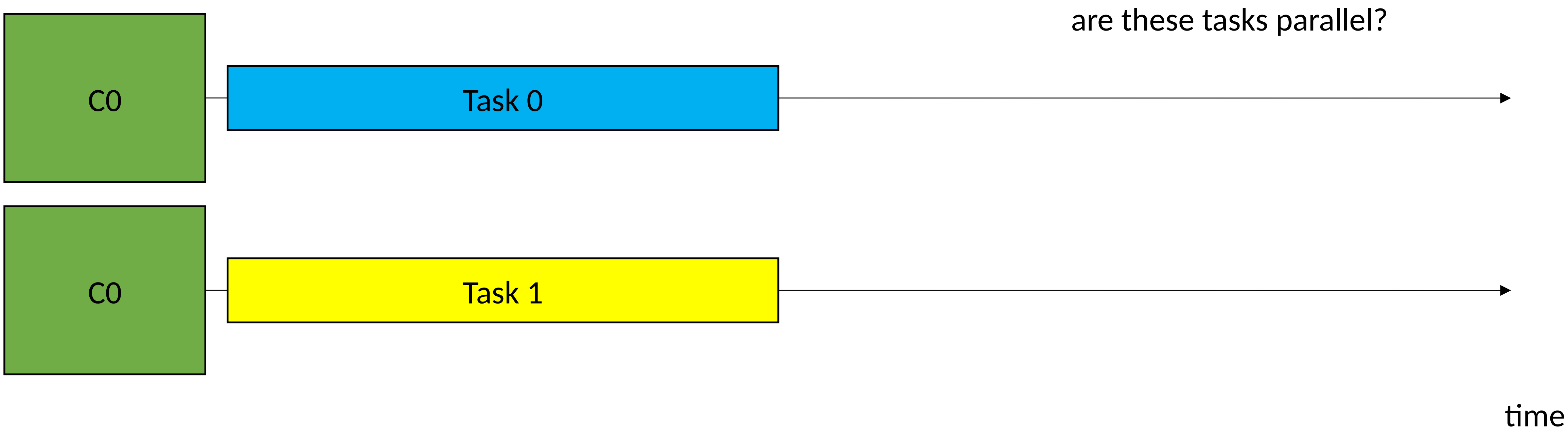


Concurrency vs. Parallelism

- 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

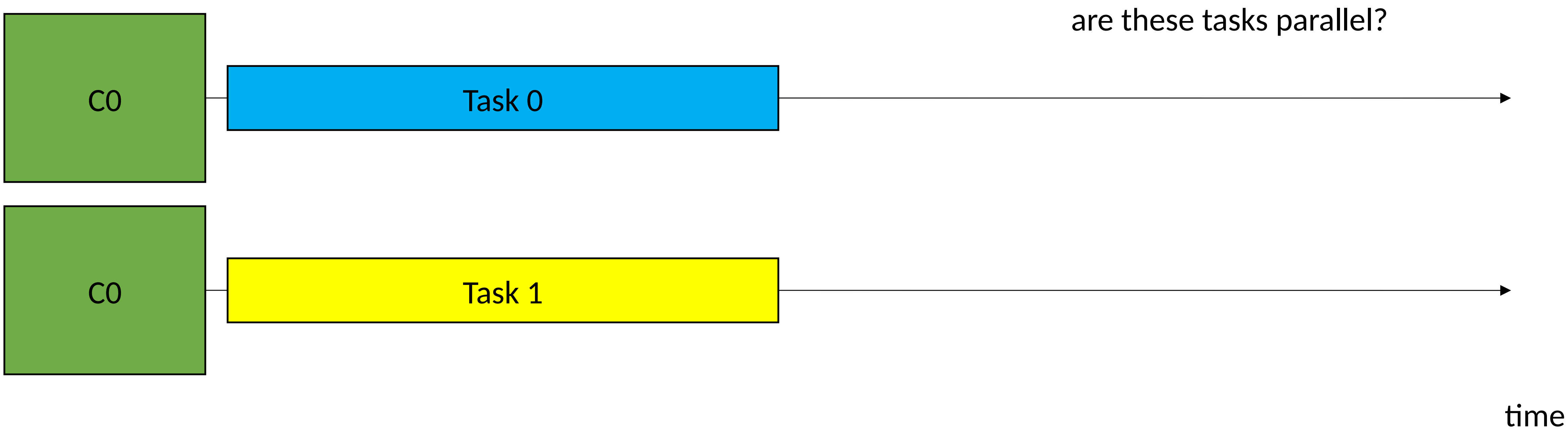


Concurrency vs. Parallelism



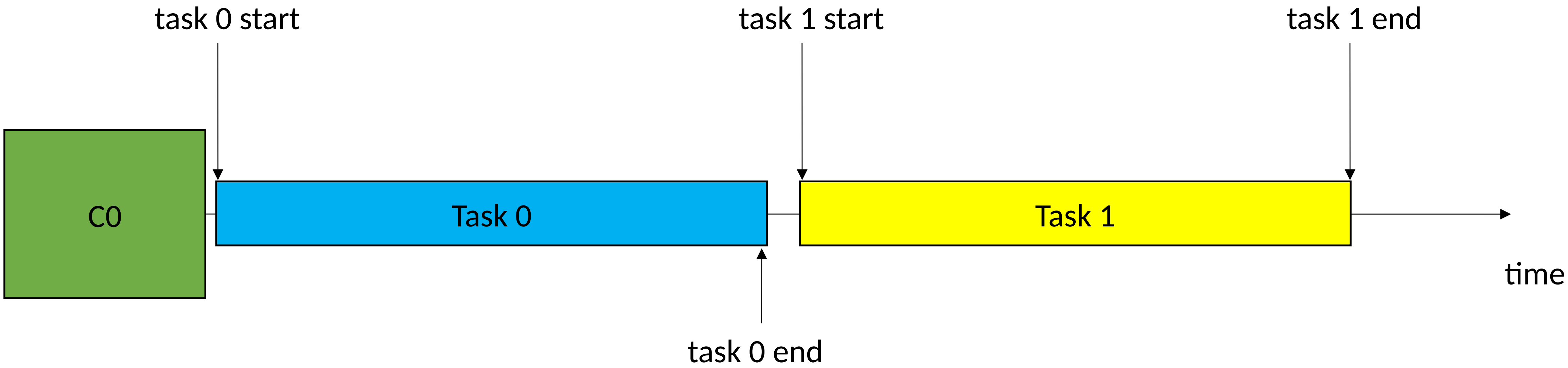
Concurrency vs. Parallelism

- Definition:
  - An execution is **parallel** if there is a point in the execution where computation is happening simultaneously



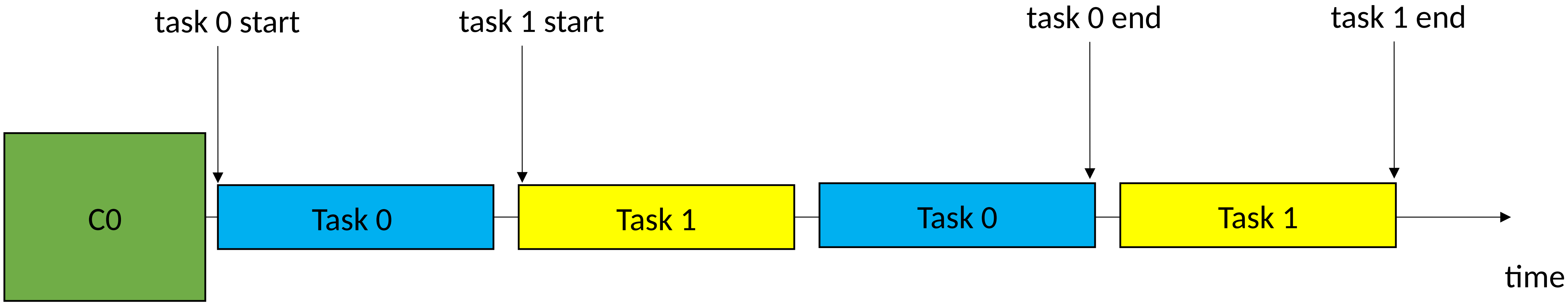
Concurrency vs. Parallelism

- Examples:
  - Neither concurrent or parallel (sequential)



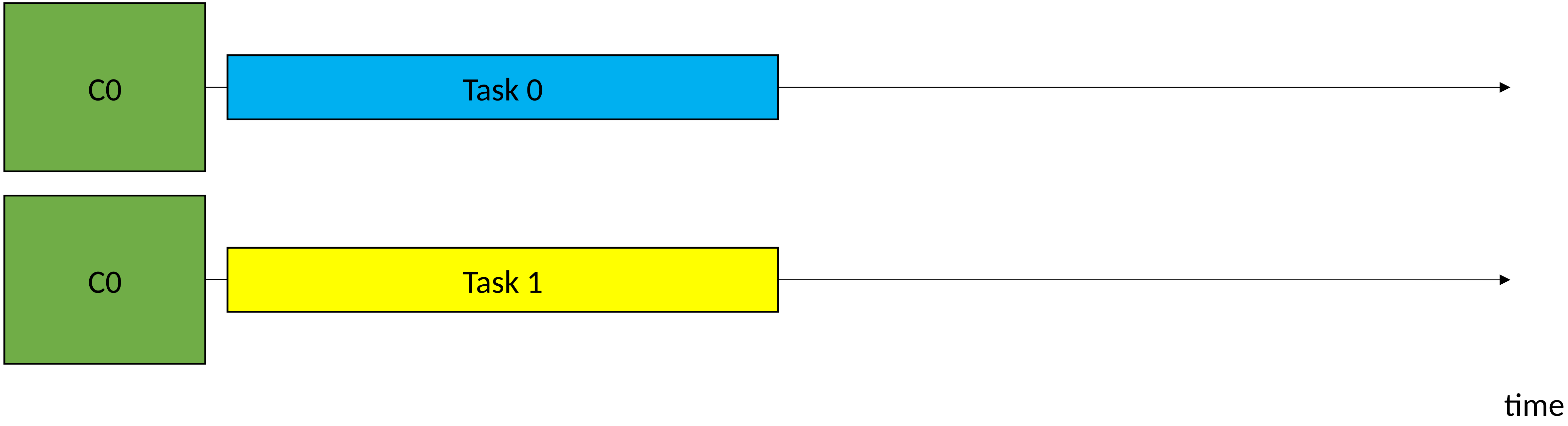
Concurrency vs. Parallelism

- Examples:
  - Concurrent but not parallel



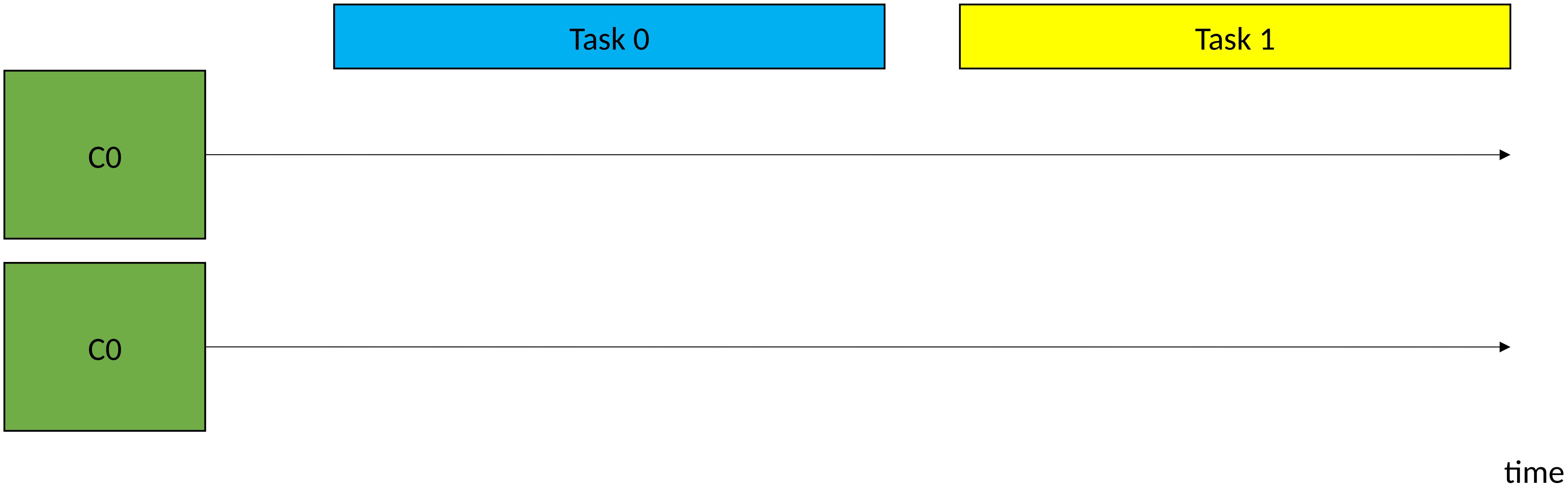
Concurrency vs. Parallelism

- Examples:
  - Parallel and Concurrent



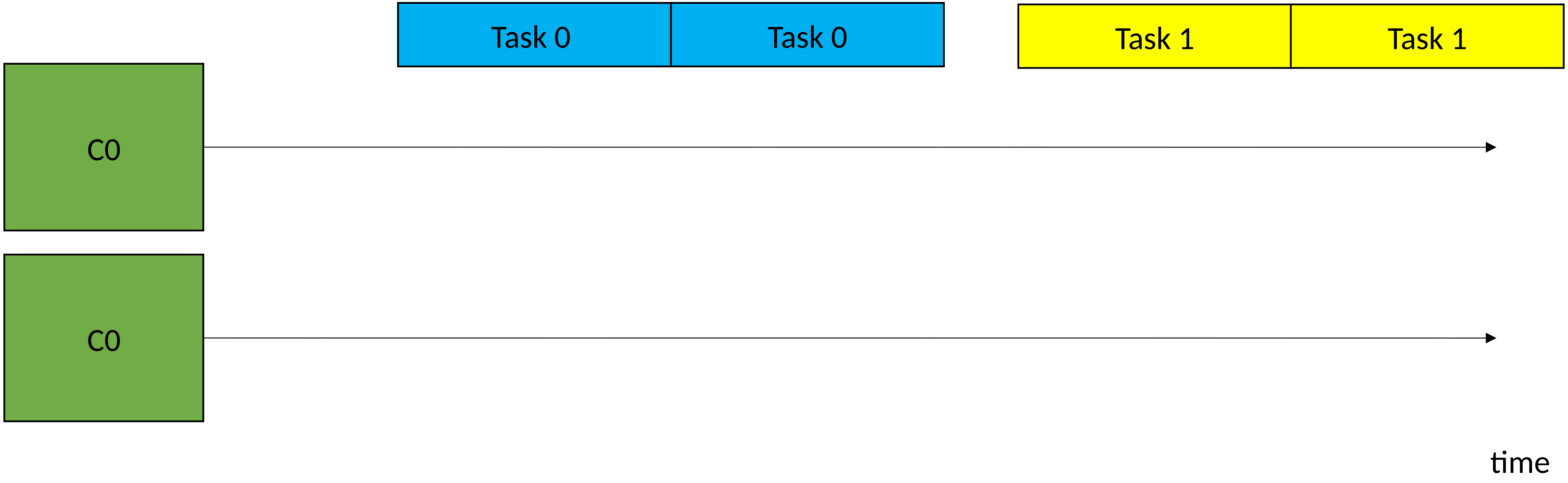
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



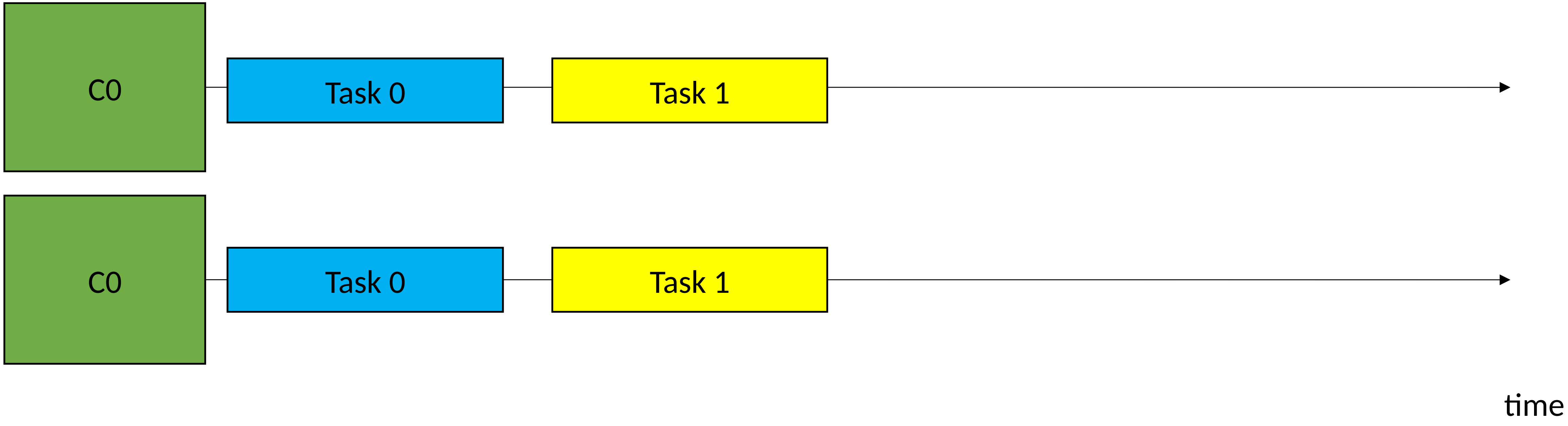
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



Concurrency vs. Parallelism

- Examples:
  - Parallel execution but task 0 and task 1 are not concurrent?



Concurrency vs. Parallelism

- In practice:
  - Terms are often used interchangeably.
  - *Parallel programming* is often used by high performance engineers when discussing using parallelism to accelerate things
  - *Concurrent programming* is used more by interactive applications, e.g. event driven interfaces.