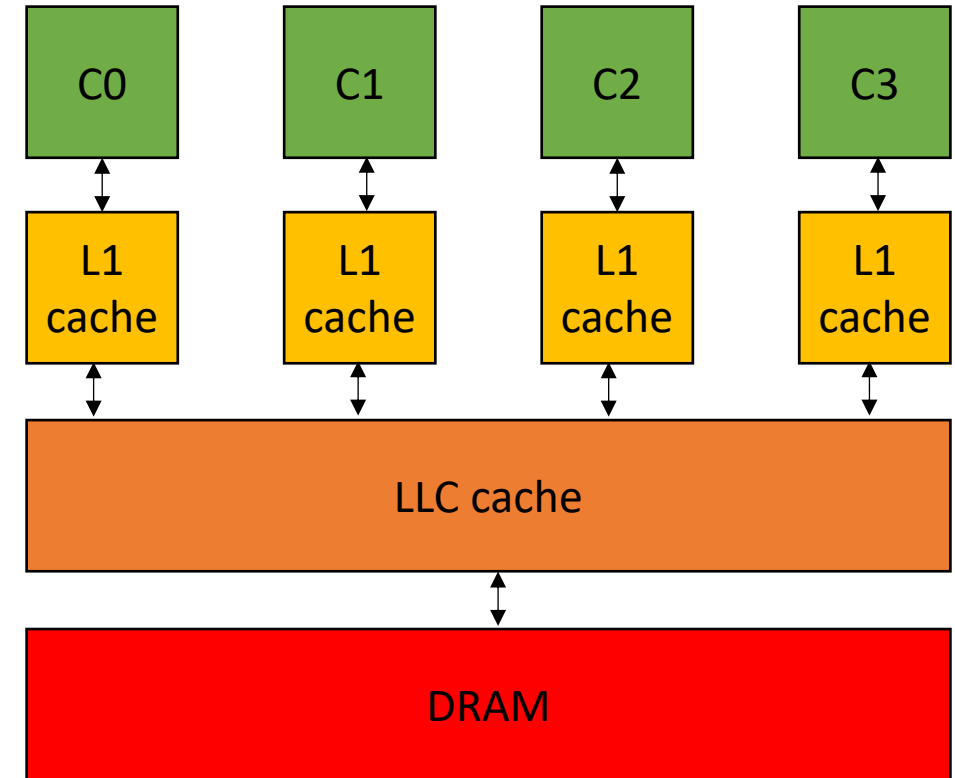


# CSE113: Parallel Programming

Jan. 7, 2022

- **Topic:** Architecture and Compiler Overview

- Cache associativity
- Cache coherence
- False sharing



# Announcements

Another 2 weeks remote ☹️

- We will:
  - continue making lectures available async (you still need to do the quiz!)
  - make the lectures downloadable
  - anything else we can do?

# Asynchronous Forums

- Piazza is setup, Reese sent an announcement with a link
  - We will moderate and try to answer questions within 24 hours
- Unofficial discord:
  - we're trusting you to moderate
  - be nice
  - don't cheat

# Office hours

- **Reese:**

- Wednesday from 2:30 - 4:30 PM
- Hybrid (remote or in person)

- **Sanya:**

- Monday from 4:00 - 5:00 PM
- Friday from 3:30 - 4:30 PM
- ***Asynchronous until Jan. 10!***
- primarily in person (when we can return)

- **Tim:**

- Tuesdays from 2:00 - 3:00 PM
- Thursdays from 2:00 - 3:00 PM
- primarily remote

- **Tyler:**

- Thursday from 3:00 - 5:00 PM
- Hybrid (remote or in person)
- Room E2 233

***Website is updated!***

# Homeworks

- Homework 1 will posted by the end of today
- Due in 2 weeks (Jan 21)
- It must run in the docker and adhere to the directory structure outlined in the assignment. We will provide a script to help you verify this.

# What you can get started with

Instructions here:

<https://sorensenucsc.github.io/CSE113-wi2022/homework-setup.html>

# Homework schedule

After Monday you should be able to do part 1

After Wednesday you should be able to do part 2

After Friday you should be able to do part 3

*TAs and tutors have been instructed not to answer questions on parts that we haven't gone over in class yet.*

# Final TODOs on our end

- Find rooms for in-person office hours
  - Less urgent now
- Post list of department resources for additional machines
  - Linux servers
  - Hummingbird (completely optional)
  - UCSC Unix (only 2 cores)
  - Please give us feedback on these resources!



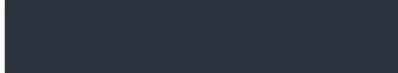
# Today's Quiz

- Normally we'll do quizzes at the beginning of class
- Remember, they are not graded, but please actually do your best
- With these classes being asynchronous, we'll release the quiz after class (2:30 PM) and have it due at midnight tomorrow

Previous quiz


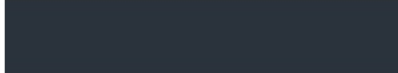
# Some answers

Changing a program from using 1 thread to using 2 threads will always provide a performance improvement

True	3 respondents	5 %	
False	63 respondents	95 %	

# Some answers

Changing a program from using 1 thread to using 2 threads will always provide a performance improvement

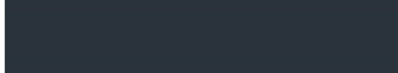
True	3 respondents	5 %	
False	63 respondents	95 %	

## False:

Thread overhead?  
Memory thrashing?  
Sequential Programs?  
Thread vs. Core?

# Some answers

Changing a program from using 1 thread to using 2 threads will always provide a performance improvement

True	3 respondents	5 %	
False	63 respondents	95 %	

## False:

Thread overhead?  
Memory thrashing?  
Sequential Programs?  
Thread vs. Core?

## True:

Intuitively this makes sense  
Machines are multicore  
Many applications are event driven  
Many data intensive applications are embarrassingly parallel

# Some answers

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.

True	9 respondents	14 %	 ✓
False	57 respondents	86 %	

# Some answers

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.

True	9 respondents	14 %	 ✓
False	57 respondents	86 %	

## False:

Imperative low-level languages (C, Java) are very difficult to prove safety/performance. Mainstream compilers do not add thread-level parallelism!

# Some answers

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.

True	9 respondents	14 %	
False	57 respondents	86 %	

## False:

Imperative low-level languages (C, Java) are very difficult to prove safety/performance. Mainstream compilers do not add thread-level parallelism!

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
    ...
}
```



# Some answers

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.

True	9 respondents	14 %	 ✓
False	57 respondents	86 %	

## True:

Parallel vs. Threads: compilers will do vectorized operations

Instruction level parallelism

Libraries? e.g. Numpy in Python, ML frameworks

# Thanks!

- Thanks for all the interesting answers on quizzes!

# Review

- Compiler transforms complicated code into simpler instructions (ISA)

# How are complicated expressions executed?

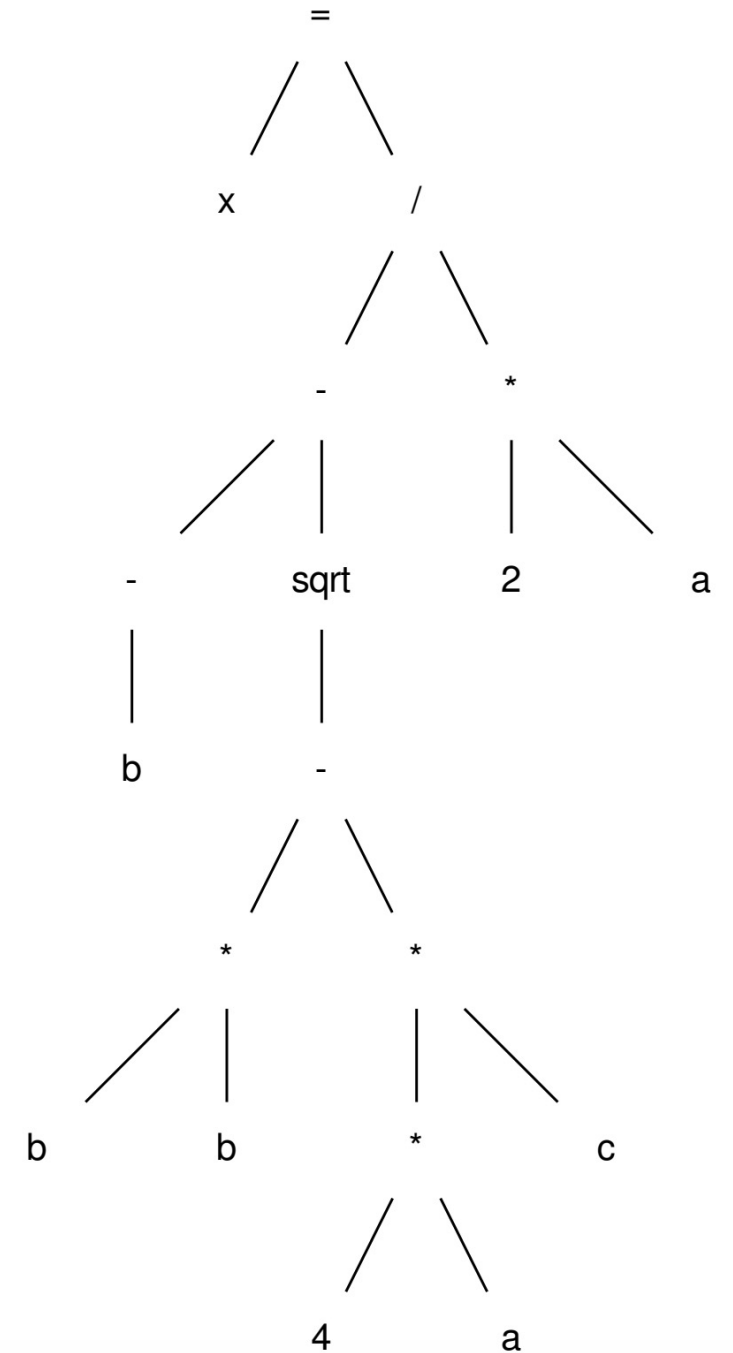
Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

`x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)`

A compiler will turn this into an  
*abstract syntax tree (AST)*



Simplify this code:

post-order traversal, using temporary variables

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

- This is not exactly an ISA
  - unlimited registers
  - not always a 1-1 mapping of instructions.
- but it is much easier to translate to the ISA
- We call this an intermediate representation, or IR
- Examples of IR: LLVM, SPIR-V

# Memory accesses

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

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

*Unless explicitly expressed in the programming language, loads and stores are split into multiple instructions!*

# Review

- Processor executes ISA instructions:
  - Processor can execute multiple threads/processes at the same time



# Core

A core executes a stream  
of sequential ISA instructions

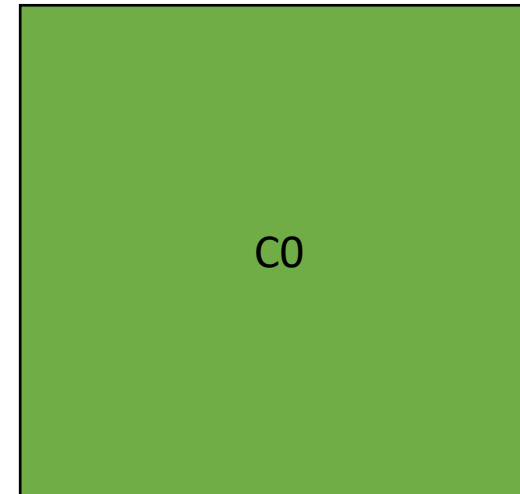
A good mental model executes  
1 ISA instruction per cycle

3 Ghz means 3B cycles per second  
1 ISA instruction takes .33 ns

## Compiled function #0

```
13      movd    eax, xmm0
14      xor     eax, 2147483648
15      movd    xmm0, eax
16      movss   dword ptr [rbp - 16], xmm0
17      movss   xmm0, dword ptr [rbp - 8]
18      mulss   xmm0, dword ptr [rbp - 8]
19      movss   xmm1, dword ptr [rip + .LCPI0_1]
20      mulss   xmm1, dword ptr [rbp - 4]
21      mulss   xmm1, dword ptr [rbp - 12]
22      subss   xmm0, xmm1
23      call    sqrt(float)
24      movaps  xmm1, xmm0
25      movss   xmm0, dword ptr [rbp - 16]
26      subss   xmm0, xmm1
27      movss   xmm1, dword ptr [rip + .LCPI0_0]
28      mulss   xmm1, dword ptr [rbp - 4]
29      divss   xmm0, xmm1
```

Thread 0



Core

# Review

- Processor executes ISA instructions:
  - Processor can execute multiple threads/processes at the same time
  - This is called concurrency, when there is enough resources to execute them simultaneously, then it is called parallelism

# Core

Preemption can occur:

- when a thread executes a long latency instruction
- periodically from the OS to provide fairness
- explicitly using sleep instructions

Compiled function #1

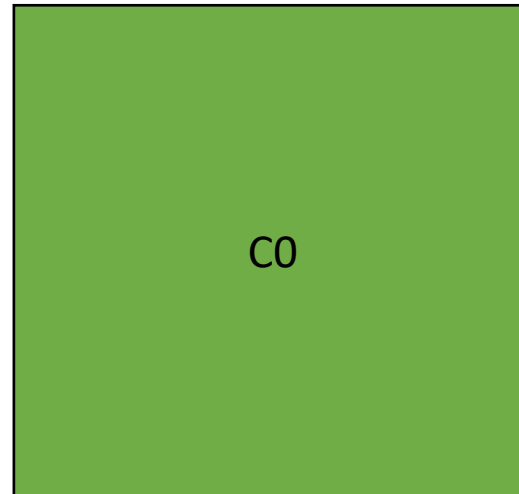
```
movss xmm0, dword ptr [rbp - 8]
mulss xmm0, dword ptr [rbp - 8]
movss xmm1, dword ptr [rip + .LCPI0_1]
mulss xmm1, dword ptr [rbp - 4]
mulss xmm1, dword ptr [rbp - 12]
subss xmm0, xmm1
call sqrt(float)
movaps xmm1, xmm0
movss xmm0, dword ptr [rbp - 16]
subss xmm0, xmm1
movss xmm1, dword ptr [rip + .LCPI0_0]
mulss xmm1, dword ptr [rbp - 4]
divss xmm0, xmm1
add rsp, 16
```

Thread 1

Compiled function #0

```
13 movd eax, xmm0
14 xor eax, 2147483648
15 movd xmm0, eax
16 movss dword ptr [rbp - 16], xmm0
17 movss xmm0, dword ptr [rbp - 8]
18 mulss xmm0, dword ptr [rbp - 8]
19 movss xmm1, dword ptr [rip + .LCPI0_1]
20 mulss xmm1, dword ptr [rbp - 4]
21 mulss xmm1, dword ptr [rbp - 12]
22 subss xmm0, xmm1
23 call sqrt(float)
24 movaps xmm1, xmm0
25 movss xmm0, dword ptr [rbp - 16]
26 subss xmm0, xmm1
27 movss xmm1, dword ptr [rip + .LCPI0_0]
28 mulss xmm1, dword ptr [rbp - 4]
29 divss xmm0, xmm1
```

Thread 2



Core



And place another thread to execute

# Multicores

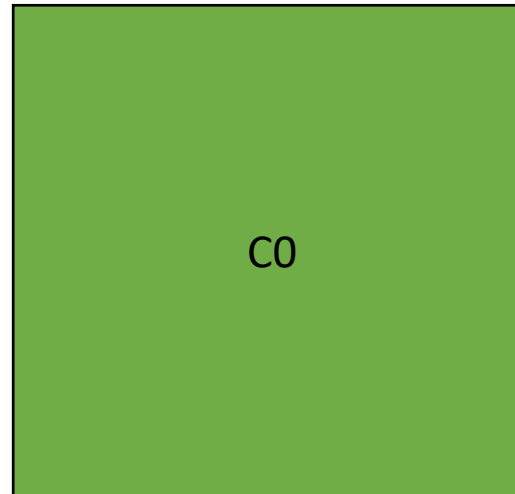
*Threads can execute simultaneously.*

*This is also concurrency. But the simultaneously called parallelism.*

Compiled function #0

```
13      movd    eax, xmm0
14      xor     eax, 2147483648
15      movd    xmm0, eax
16      movss   dword ptr [rbp - 16], xmm0
17      movss   xmm0, dword ptr [rbp - 8]
18      mulss   xmm0, dword ptr [rbp - 8]
19      movss   xmm1, dword ptr [rip + .LCPI0_1]
20      mulss   xmm1, dword ptr [rbp - 4]
21      mulss   xmm1, dword ptr [rbp - 12]
22      subss   xmm0, xmm1
23      call    sqrt(float)
24      movaps  xmm1, xmm0
25      movss   xmm0, dword ptr [rbp - 16]
26      subss   xmm0, xmm1
27      movss   xmm1, dword ptr [rip + .LCPI0_0]
28      mulss   xmm1, dword ptr [rbp - 4]
29      divss   xmm0, xmm1
```

Thread 0

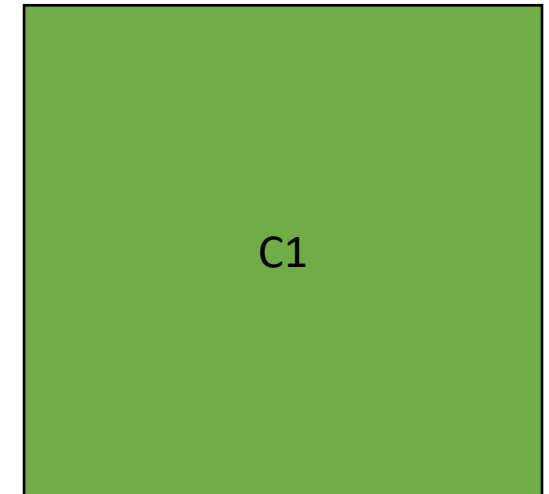


Core

Compiled function #1

```
movss   dword ptr [rsp - 16], xmm0
movss   xmm0, dword ptr [rbp - 8]
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call     sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
add     rsp, 16
```

Thread 1



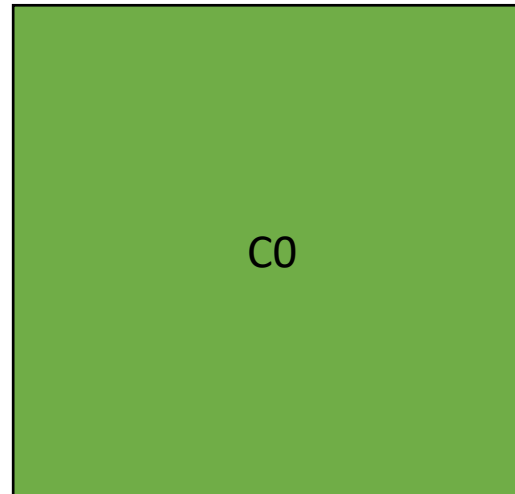
Core

# Multicores

Compiled function #0

```
13    movd    eax, xmm0
14    xor     eax, 2147483648
15    movd    xmm0, eax
16    movss   dword ptr [rbp - 16], xmm0
17    movss   xmm0, dword ptr [rbp - 8]
18    mulss   xmm0, dword ptr [rbp - 8]
19    movss   xmm1, dword ptr [rip + .LCPI0_1]
20    mulss   xmm1, dword ptr [rbp - 4]
21    mulss   xmm1, dword ptr [rbp - 12]
22    subss   xmm0, xmm1
23    call    sqrt(float)
24    movaps  xmm1, xmm0
25    movss   xmm0, dword ptr [rbp - 16]
26    subss   xmm0, xmm1
27    movss   xmm1, dword ptr [rip + .LCPI0_0]
28    mulss   xmm1, dword ptr [rbp - 4]
29    divss   xmm0, xmm1
```

Thread 0

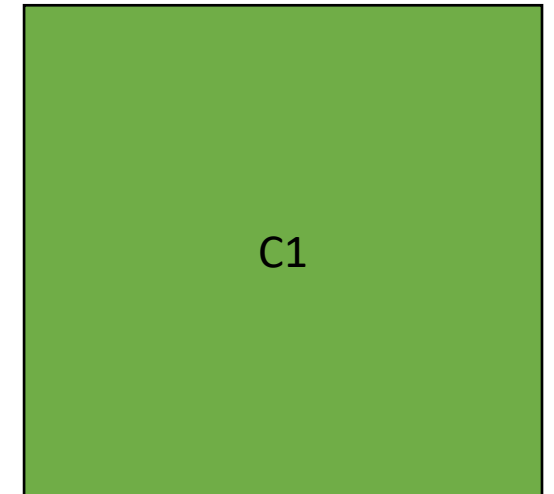


Core

Compiled function #1

```
movss   dword ptr [rip + .LCPI0_1], xmm0
movss   xmm0, dword ptr [rbp - 8]
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    sqrt(float)
movaps  xmm1, xmm0
movss   xmm0, dword ptr [rbp - 16]
subss   xmm0, xmm1
movss   xmm1, dword ptr [rip + .LCPI0_0]
mulss   xmm1, dword ptr [rbp - 4]
divss   xmm0, xmm1
add     rsp, 16
```

Thread 1



Core

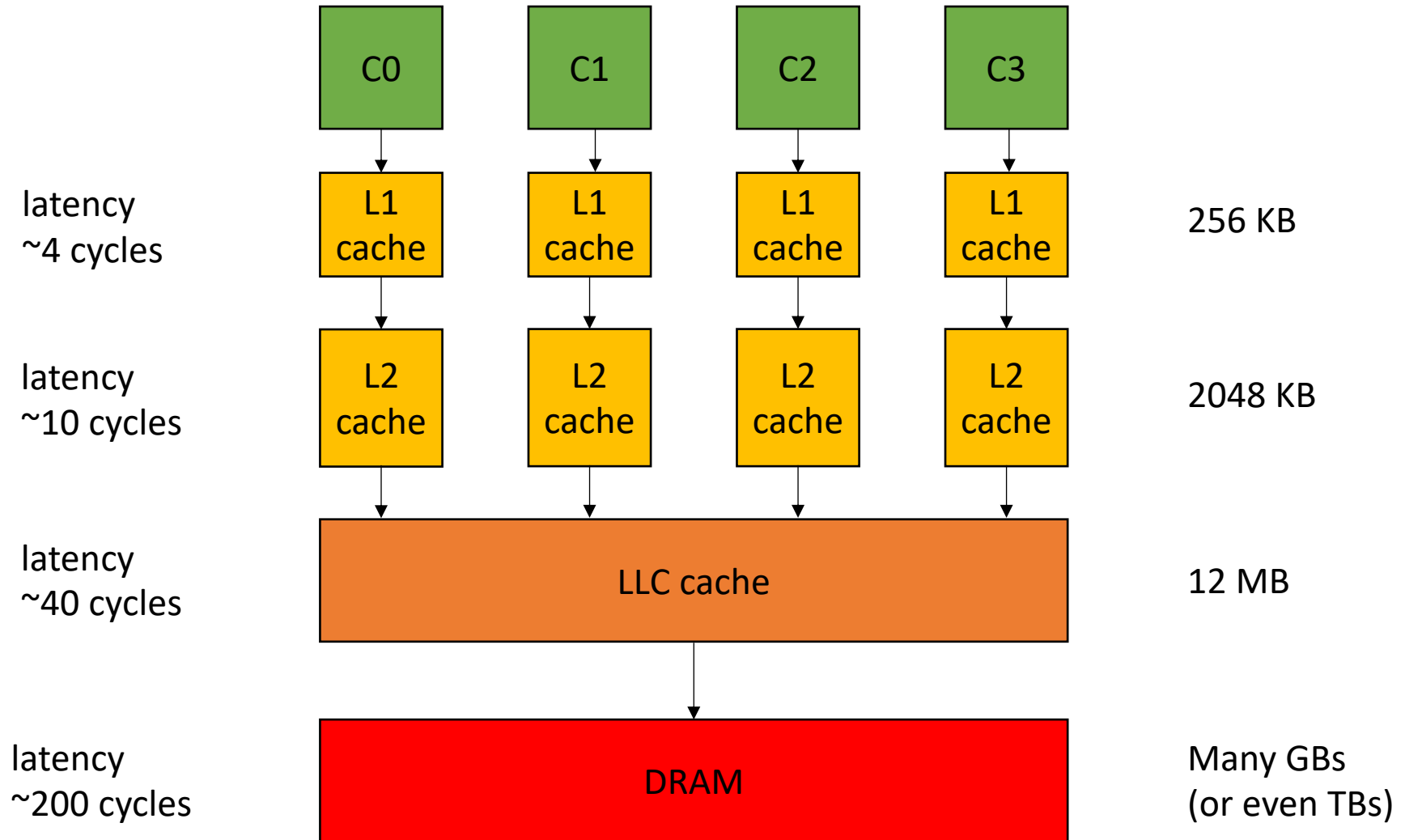
This is fine if threads are independent:  
e.g. running Chrome and Spotify at the  
same time.

If threads need to cooperate to run  
the program, then they need to communicate  
through memory

# Review

- Caches make memory accesses faster

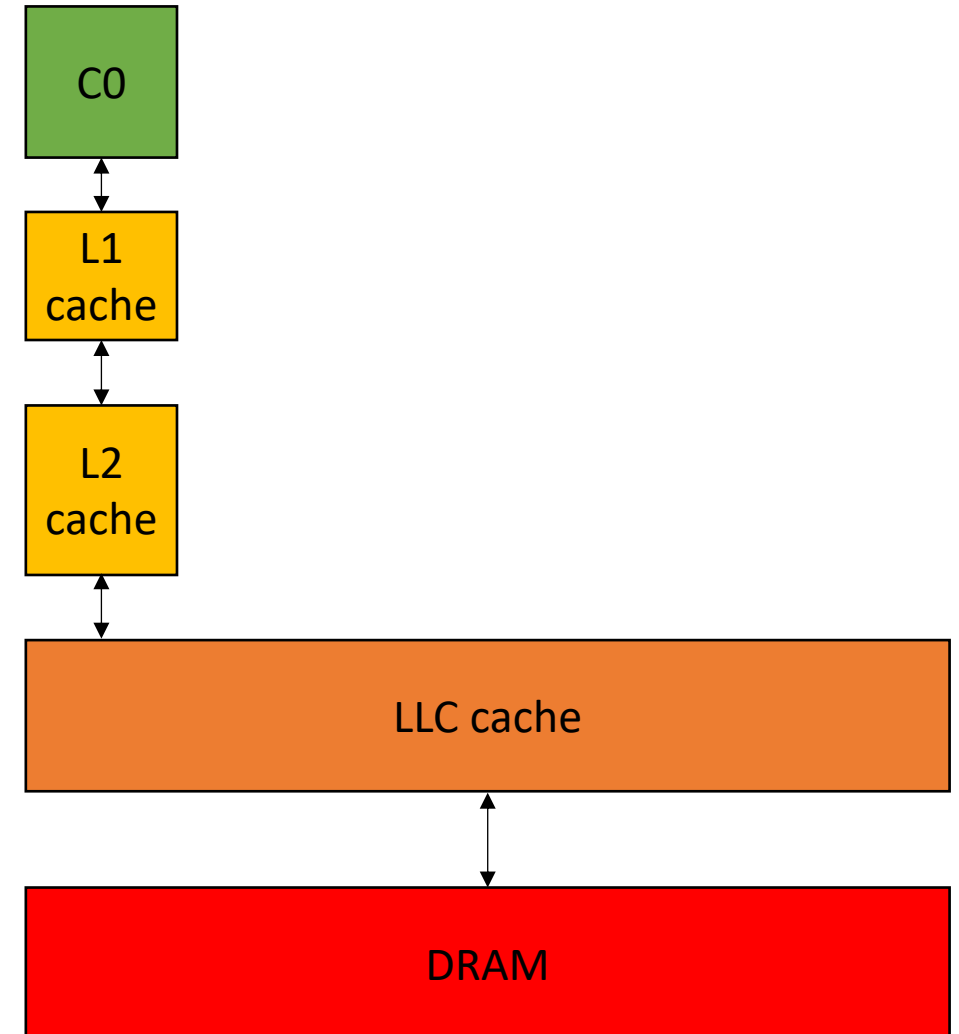
# Caches



# Caches

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

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





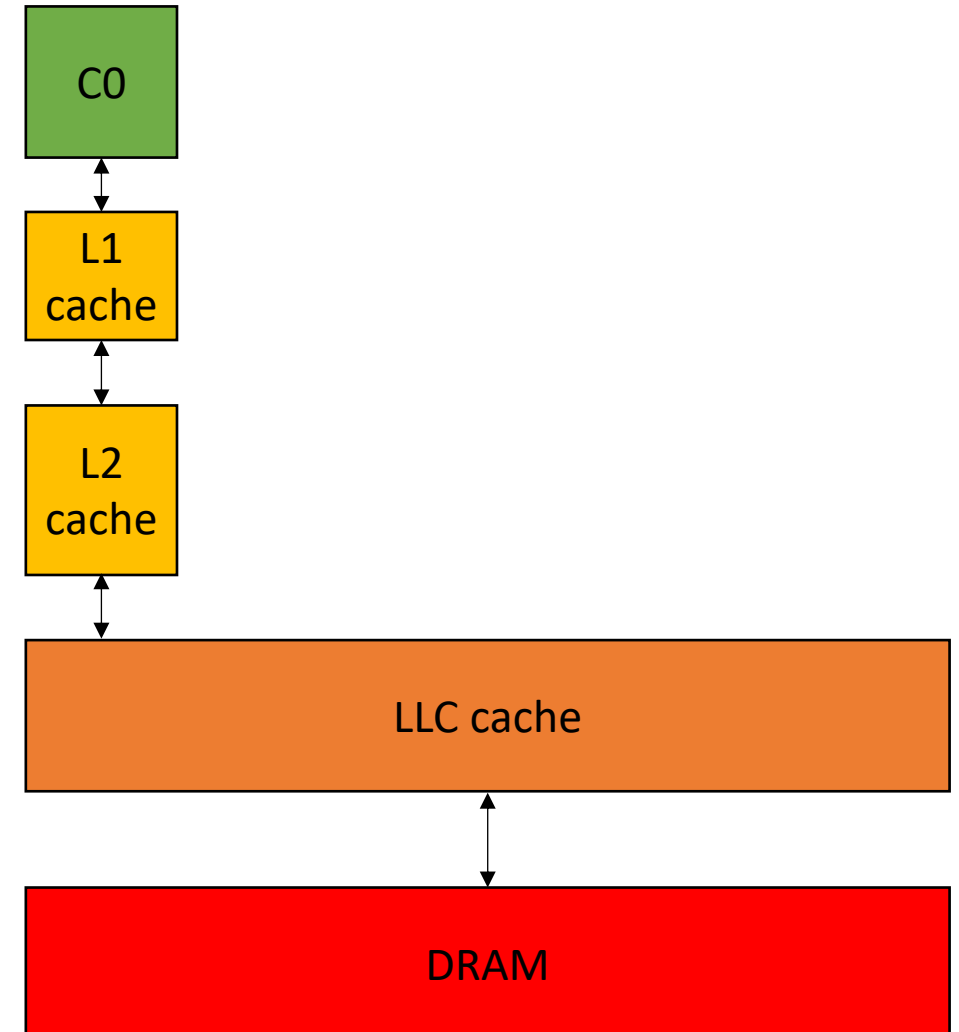
# Caches

```
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!*



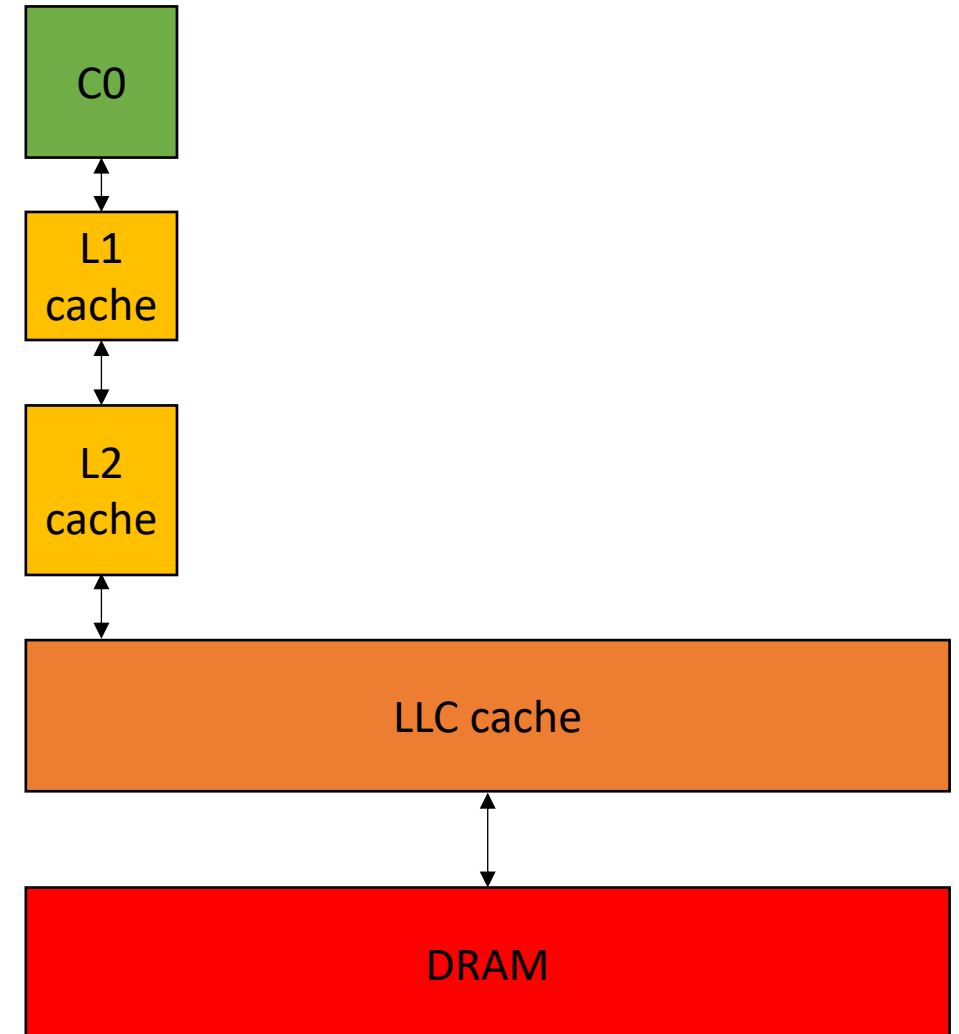
# Caches

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

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

4 cycles

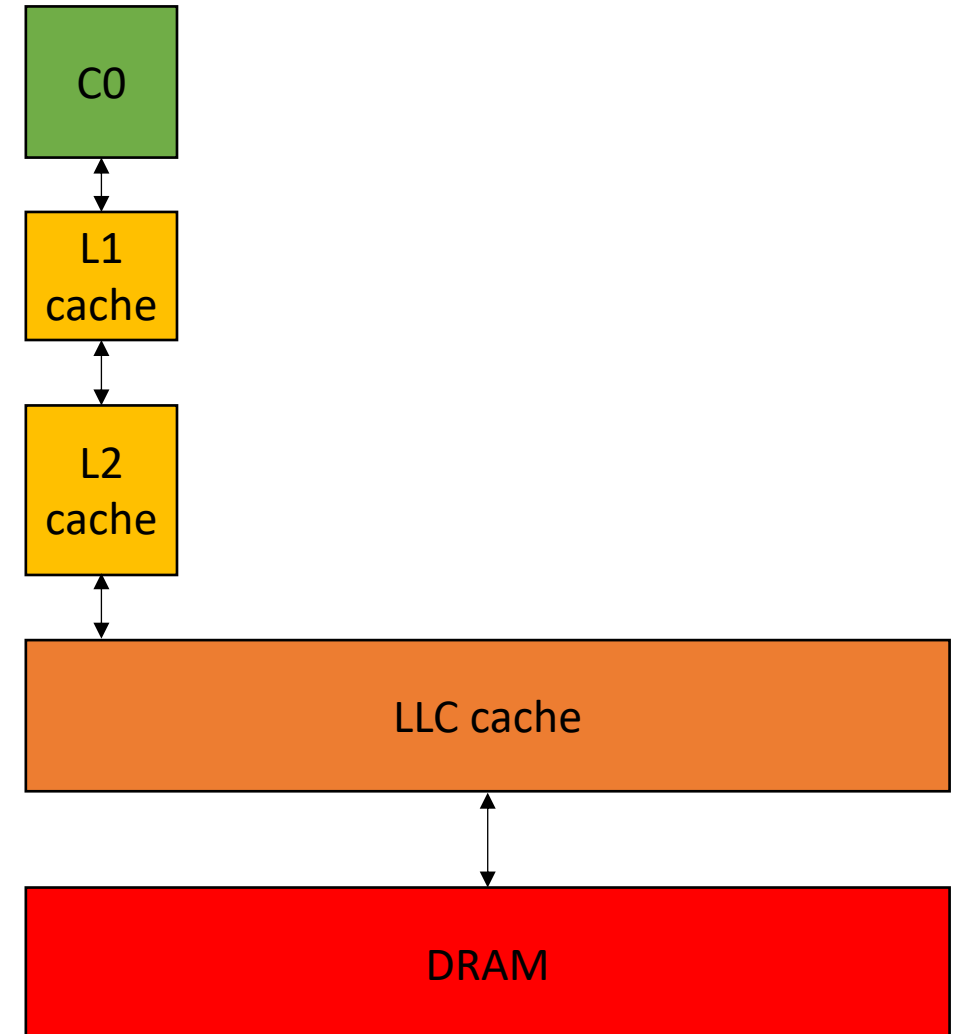
1 cycles



# 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

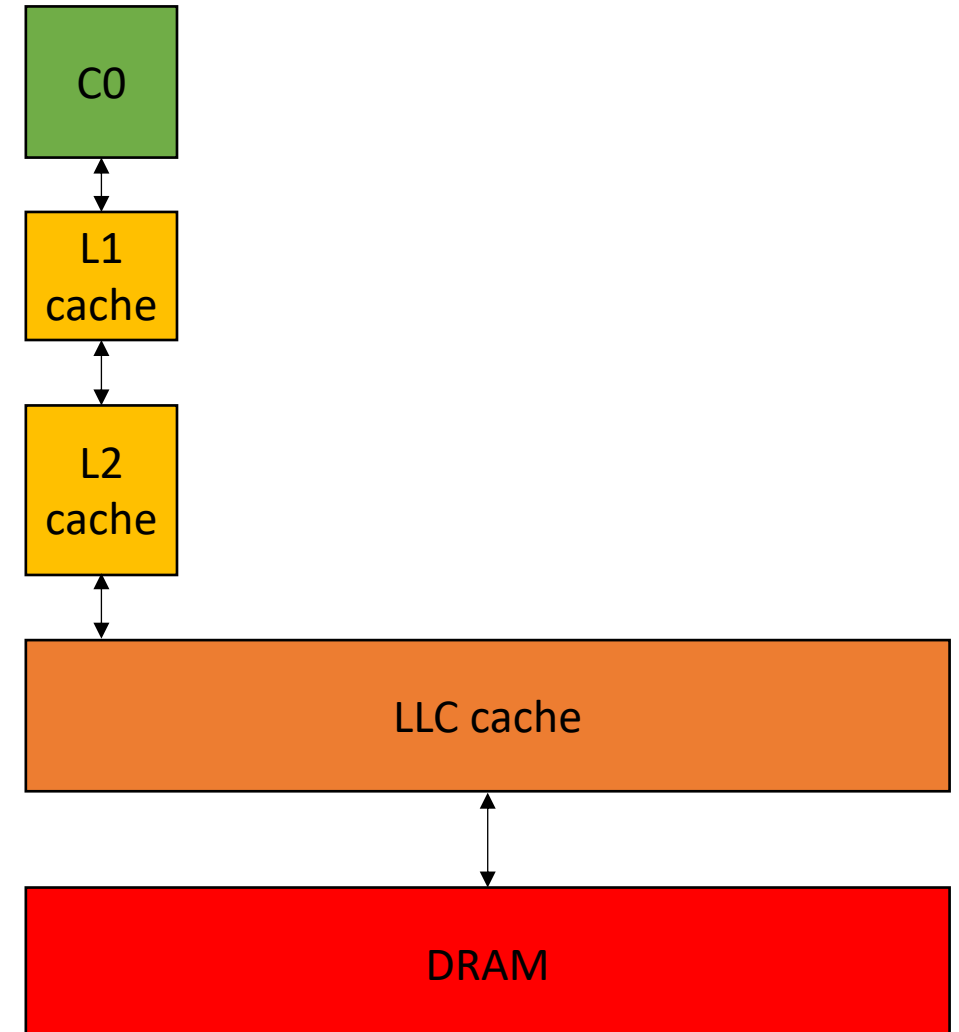


# Caches

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

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

**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]++;  
}
```

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

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

# Passing pointers

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

*pass pointer directly through*

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

*pass an offset of 8*

```
int foo2(int *a) {  
    increment_several(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*



On to the lecture!

# Lecture Schedule

Architecture continued:

- Cache lines
- Cache replacement policy
- Cache coherence
- False sharing

# Cache lines

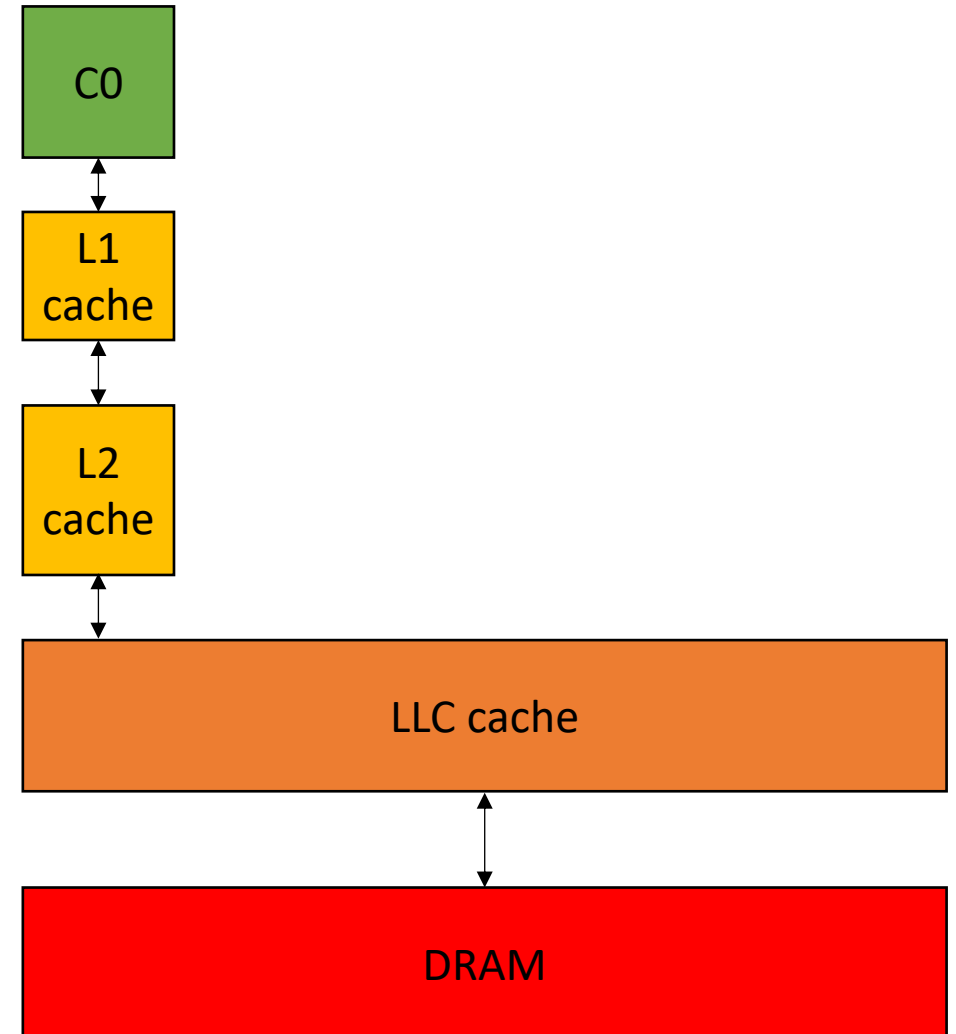
- Cache line size for x86: 64 bytes:
  - 64 chars
  - 32 shorts
  - 16 float or int
  - 8 double or long
  - 4 long long

# Caches

```
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*



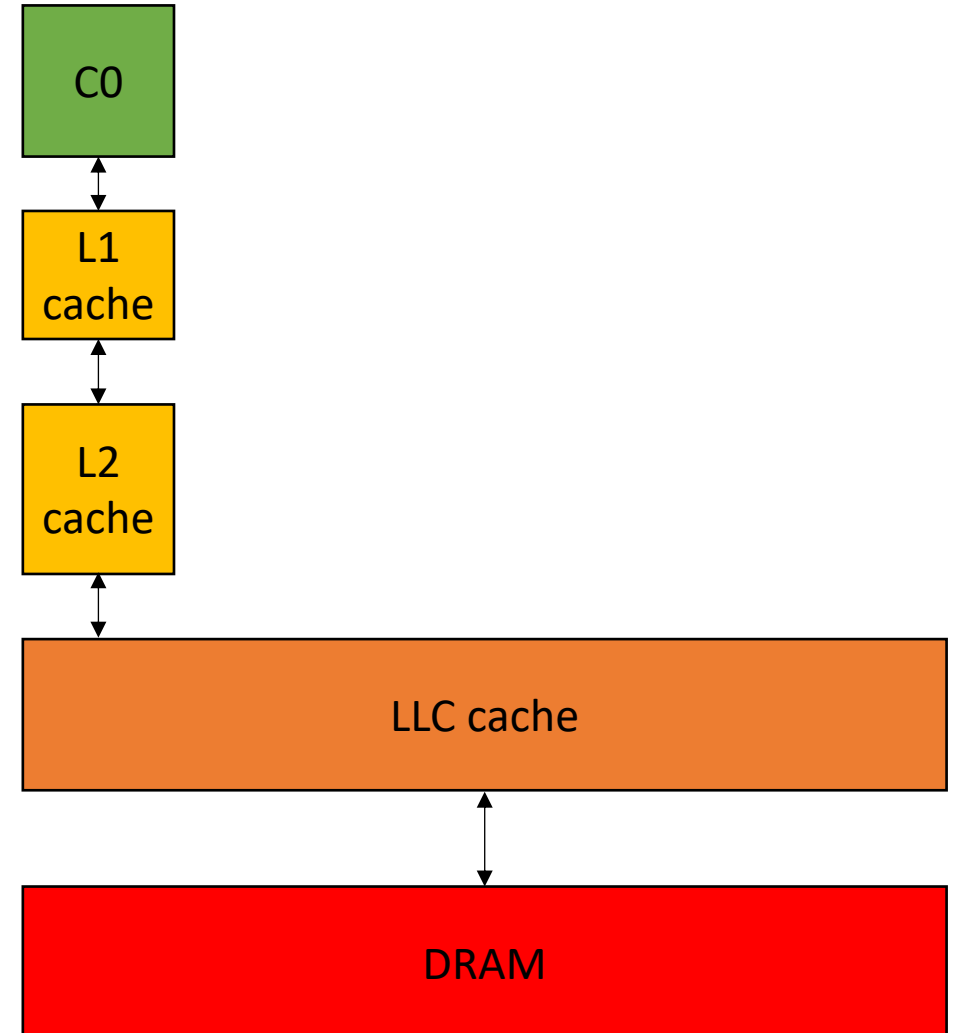
# Caches

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

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

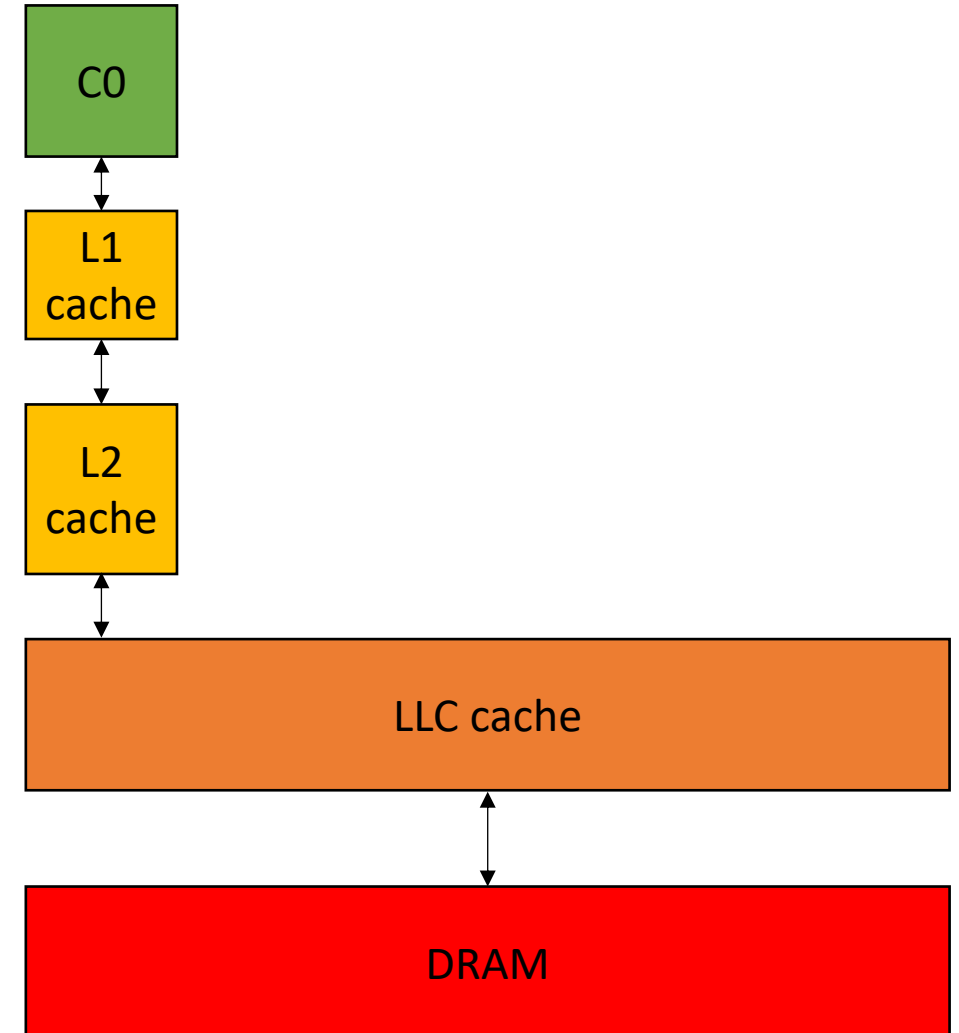
$a[0] - a[15]$

*Assume  $a[0]$  is not in the cache*



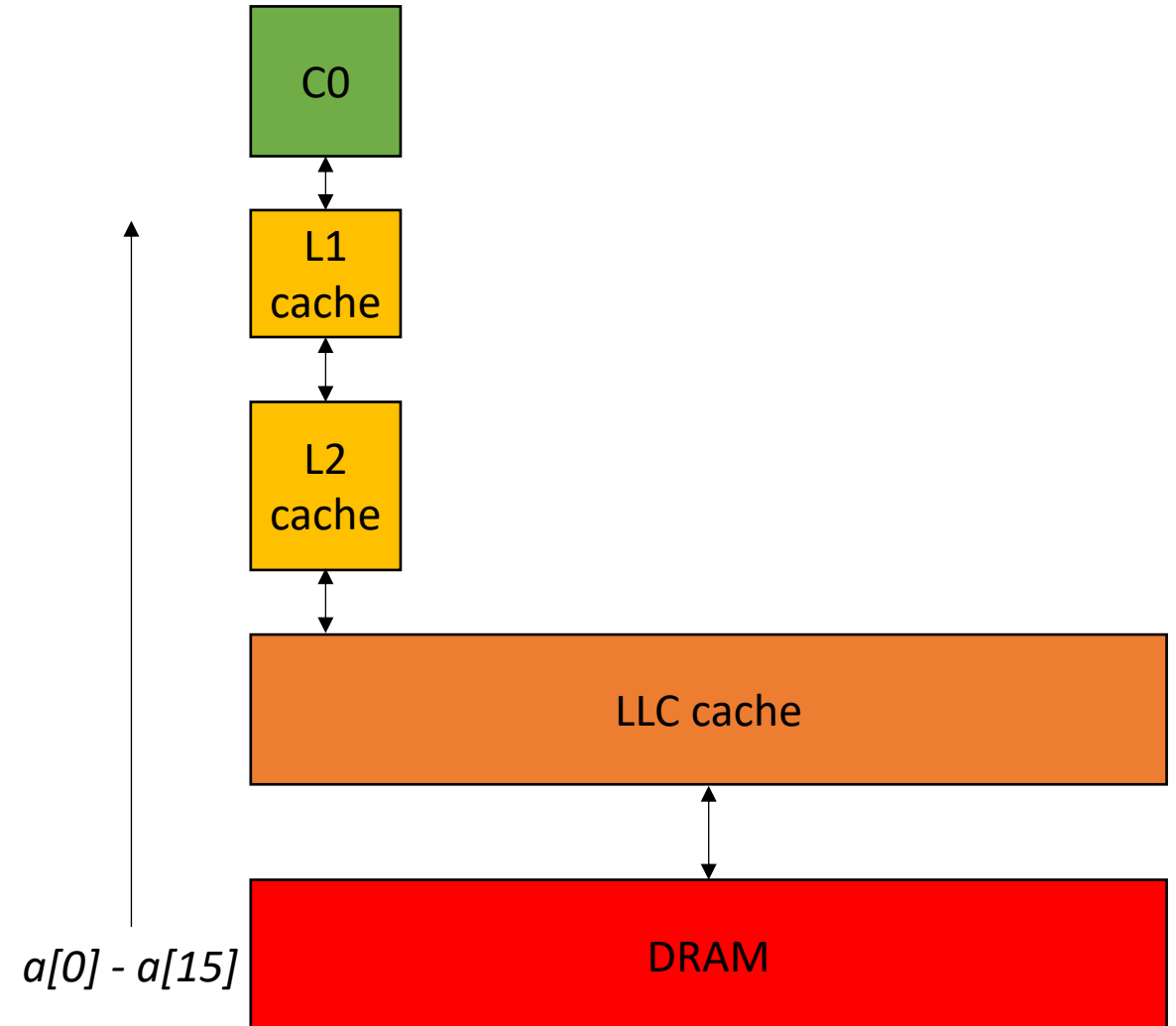
# Caches

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



# Caches

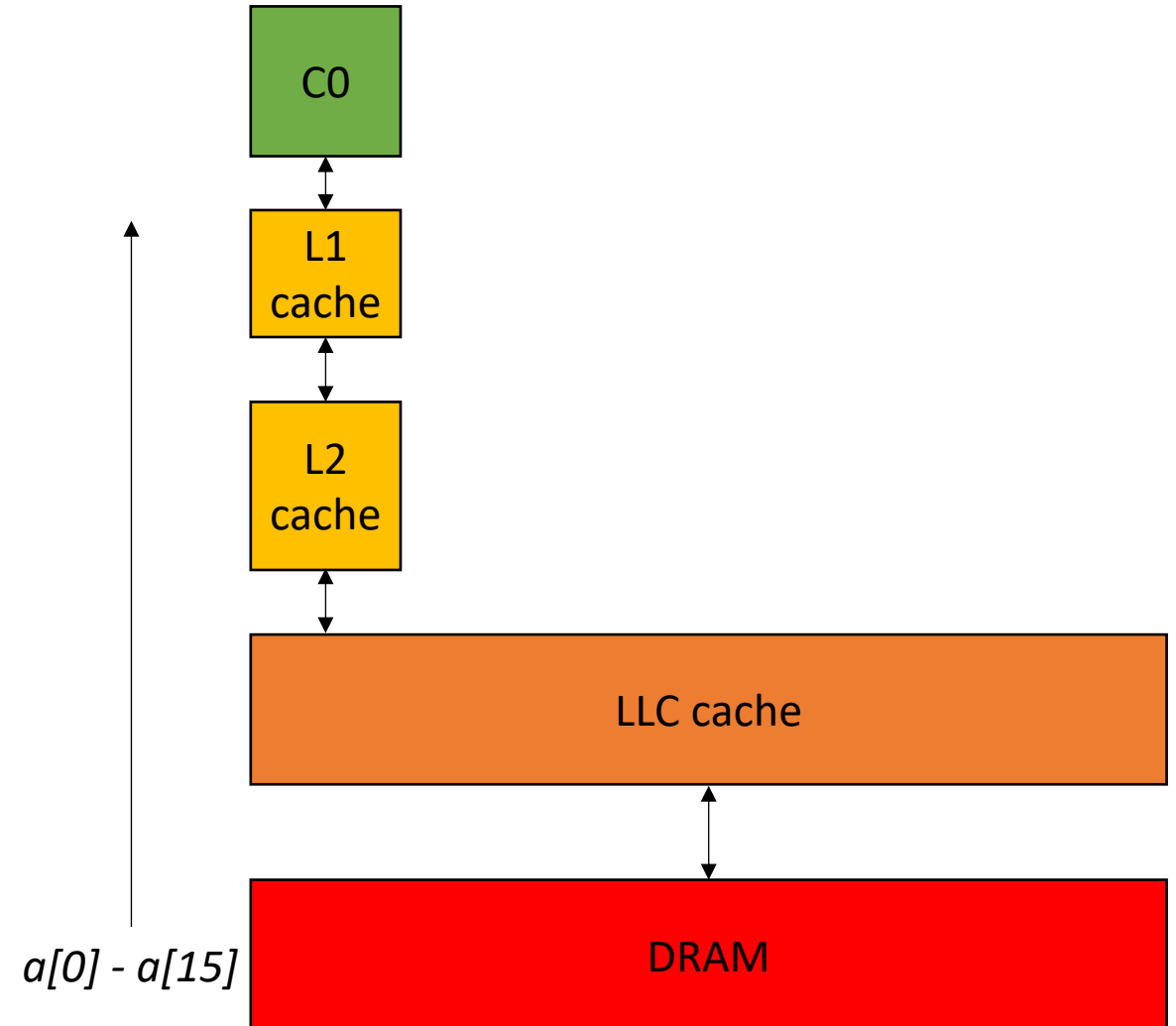
```
int increment_several(int *a) {  
    a[0]++;  
    a[15]++;  
    a[16]++;  
}
```



# Caches

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

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



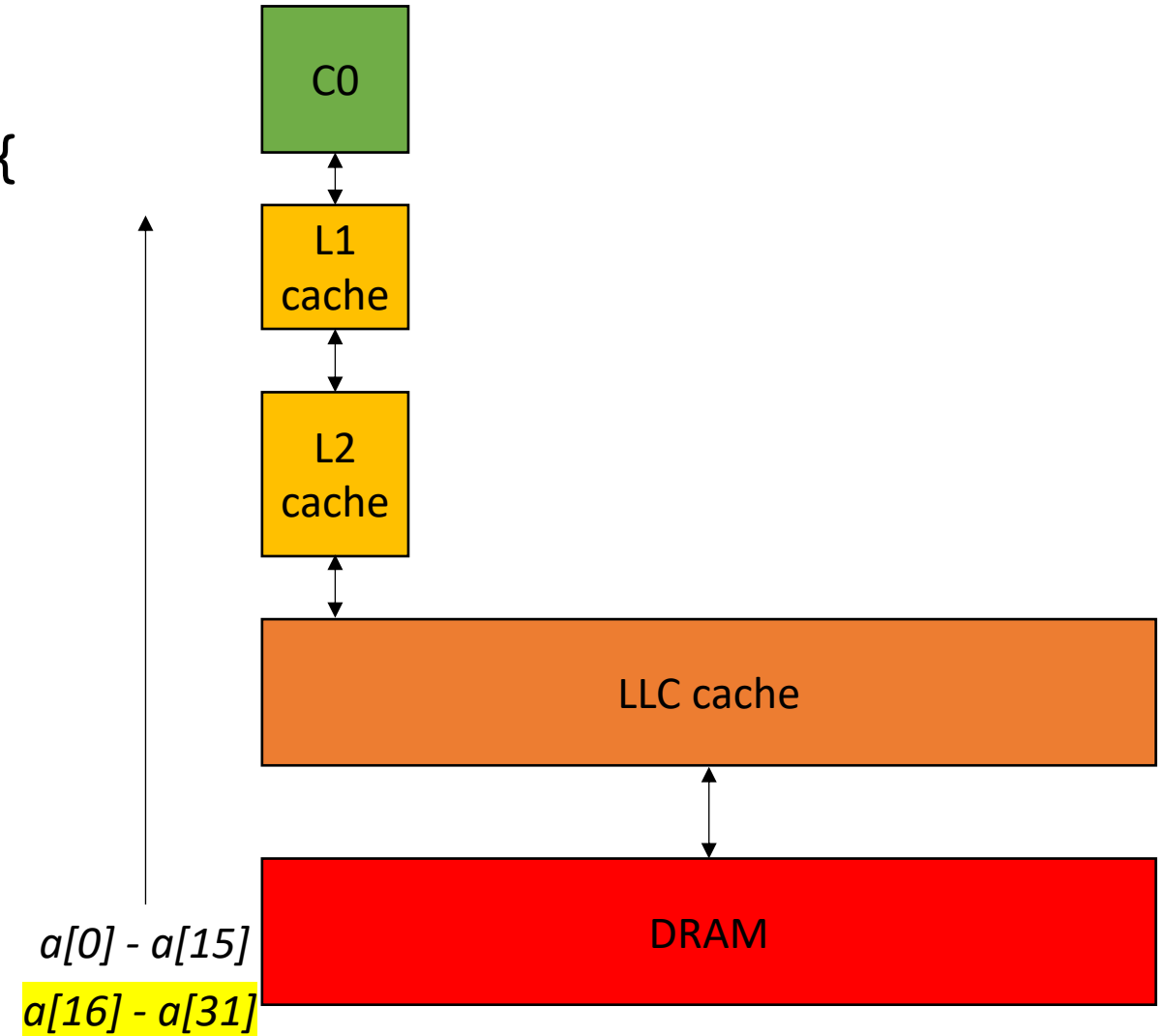


# Caches

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

*Miss*

*Assume a[0] is not in the cache*

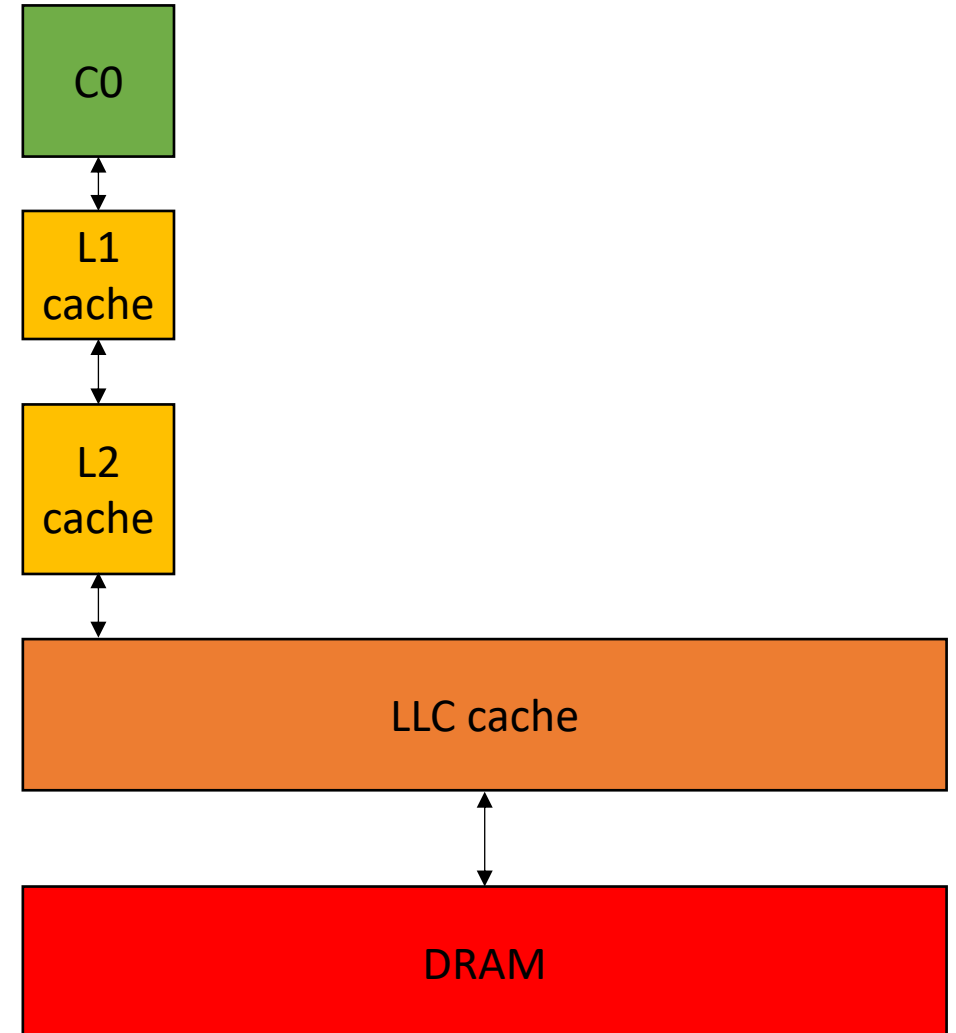


# Cache alignment

```
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*

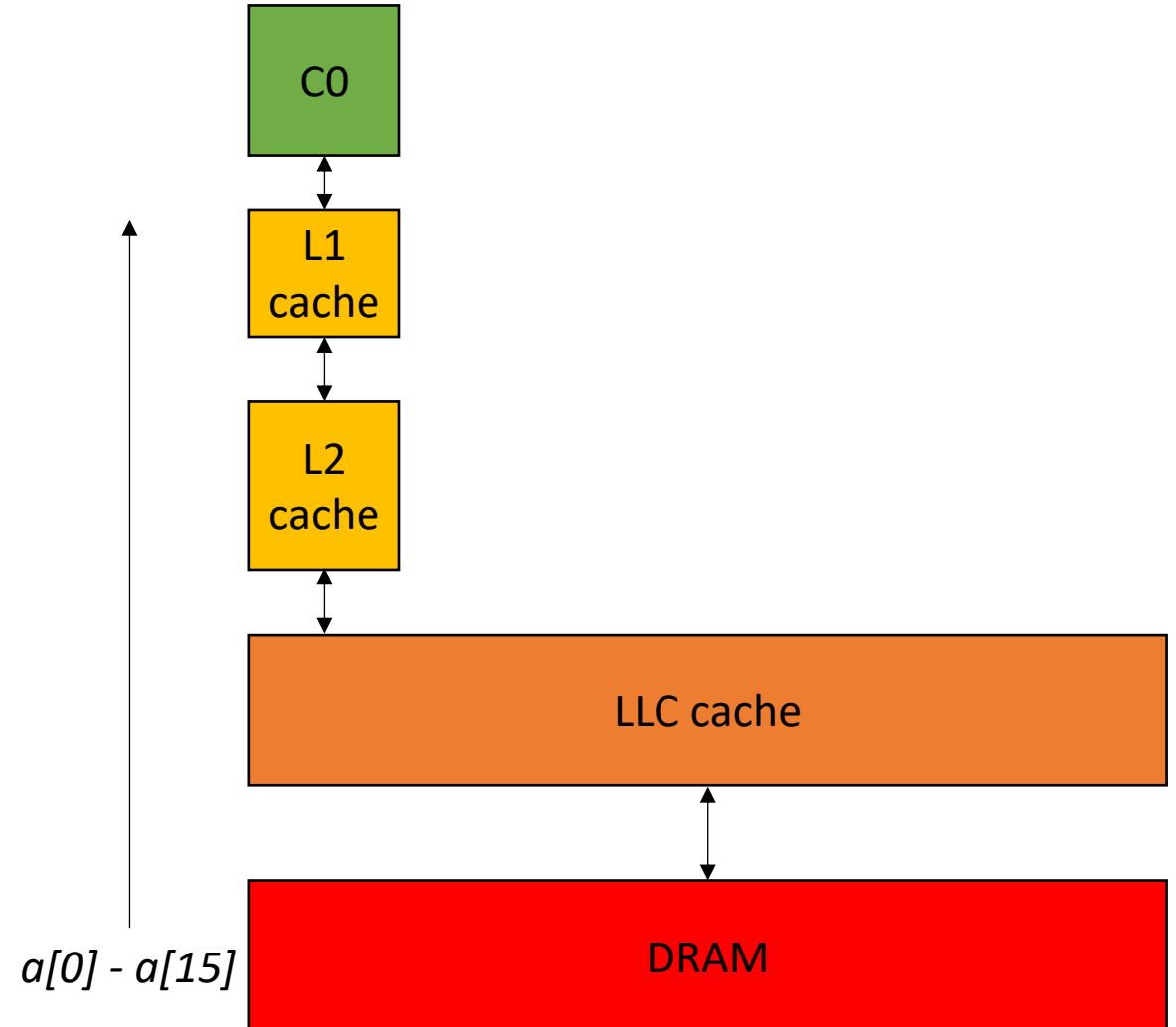


# Cache alignment

```
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*



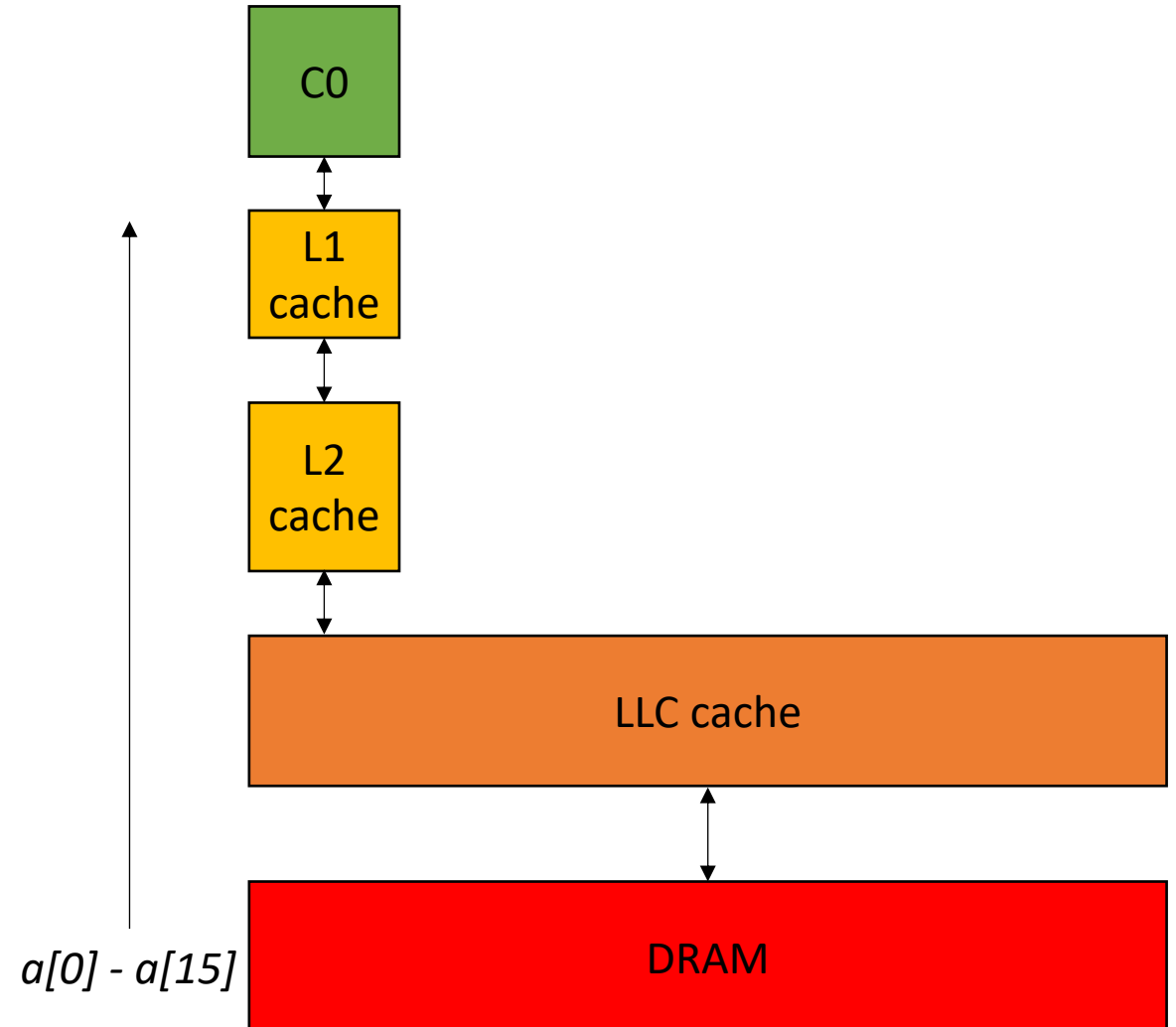
# Cache alignment

```
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

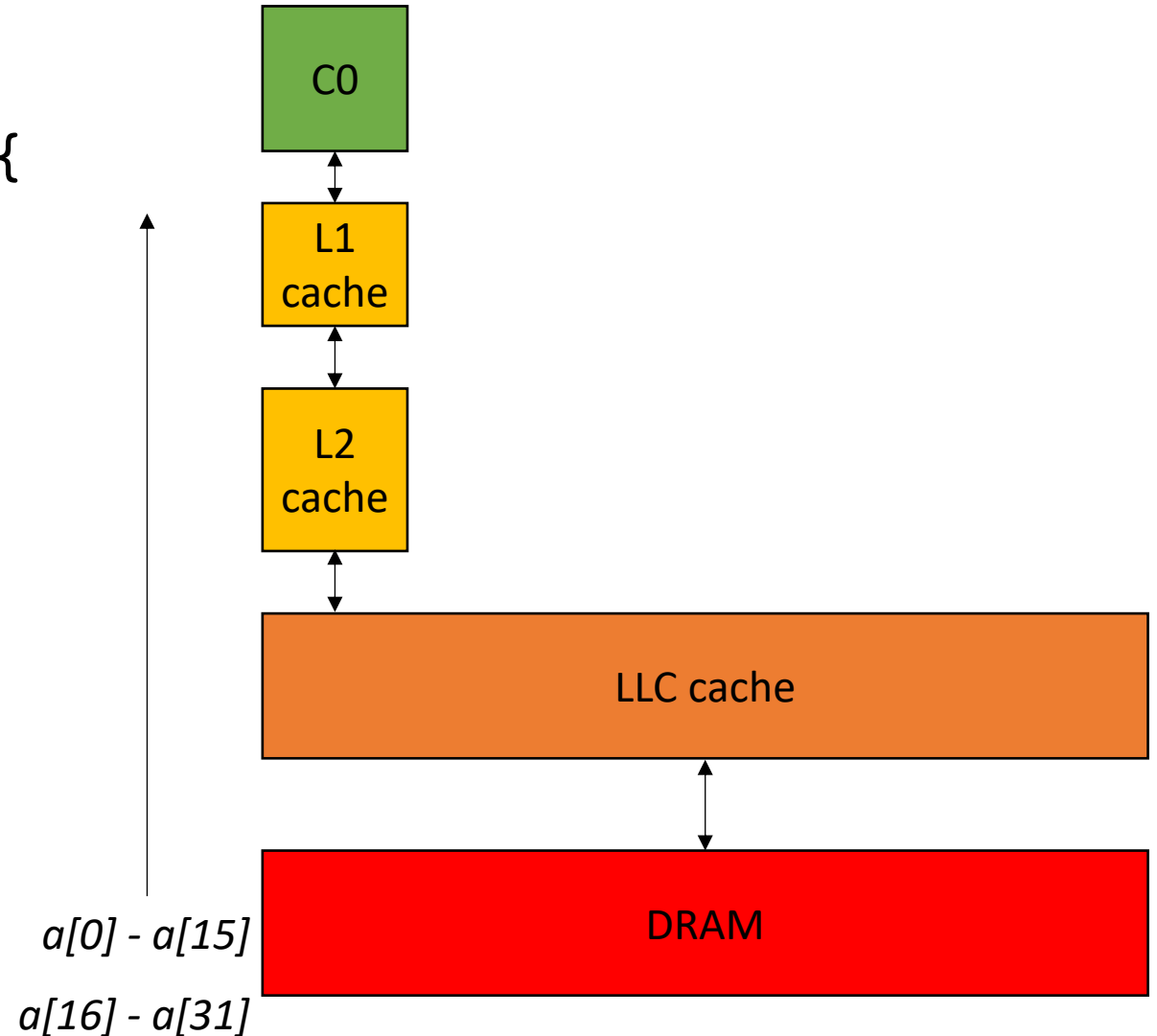
```
int increment_several(int *b) {  
    b[0]++;  
    b[15]++;  
}
```

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

This loads a[8]

This loads a[23], a miss!

*Assume a[0] is not in the cache*



# 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 organization



# Cache organization

In this illustration, box is a cache line.

Assume we read only addresses that start a cache line

Cache is size 6 \* 64 bytes

Memory is size 18 \* 64 bytes

## Cache

value

--	--	--	--	--	--

address

--	--	--	--	--	--

## Memory

value

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

address

0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440
------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value

--	--	--	--	--	--

address

--	--	--	--	--	--

## Memory

value

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

address

0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440
------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value

--	--	--	--	--	--

address

--	--	--	--	--	--

Example: Read address 0x00

## Memory

value

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

address

0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440
------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0					
address	0x00					

Example: Read address 0x00

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value

0					
---	--	--	--	--	--

address

0x00					
------	--	--	--	--	--

Example: Read address 0x1C0

## Memory

value

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

address

0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440
------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7				
address	0x00	0x1C0				

Example: Read address 0x80

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7				
address	0x00	0x1C0				

Example: Read address 0x80

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x80

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440



# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x1C0

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x1C0

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x1C0

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x180

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Example: Read address 0x180

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

**Cache**

*evict!*

value

		7	2			
--	--	---	---	--	--	--

address

	0x1C0	0x80			
--	-------	------	--	--	--

Example: Read address 0x180

**Memory**

value

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

address

0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440
------	------	------	------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value		7	2			
address		0x1C0	0x80			

Example: Read address 0x180

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**Direct mapped:** every memory location can go exactly one place in the cache.

cache block location =  $(\text{address}/64) \% (\text{cache size})$

## Cache

value	6	7	2			
address	0x180	0x1C0	0x80			

Example: Read address 0x180

*We had to evict even though there was room in the cache!*

## Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440



# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value			
address			

*set 1*

value			
address			

*set 2*

Read 0x00  
Read 0x1C0  
Read 0x40

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value			
address			

set 1

value			
address			

set 2

Read 0x00  
Read 0x1C0  
Read 0x40

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value			
address			

set 2

Read 0x00  
Read 0x1C0  
Read 0x40

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value			
address			

set 2

Read 0x180

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value			
address			

set 2

Read 0x180

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value			
address			

set 2

Read 0x180

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

Read 0x180

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440



# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

Read 0x300

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

Read 0x300

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	0	7	2
address	0x00	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

Read 0x300

Evict the “least recently used” value

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

Cache

value		7	2
address		0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Read 0x300

Evict the “least recently used” value

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

**N-way Associative:** every memory location can go N places in the cache.

cache block location  $(\text{address}/64) \% (\text{cache size} / N)$

Cache will make an “intelligent” decision on which value to evict

Cache

value	12	7	2
address	0x300	0x1C0	0x80

set 1

value	6		
address	0x180		

set 2

Read 0x300

Evict the “least recently used” value

example 2-way associative

Memory

value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

# Cache organization

- Why aren't caches fully associative?

# Cache organization

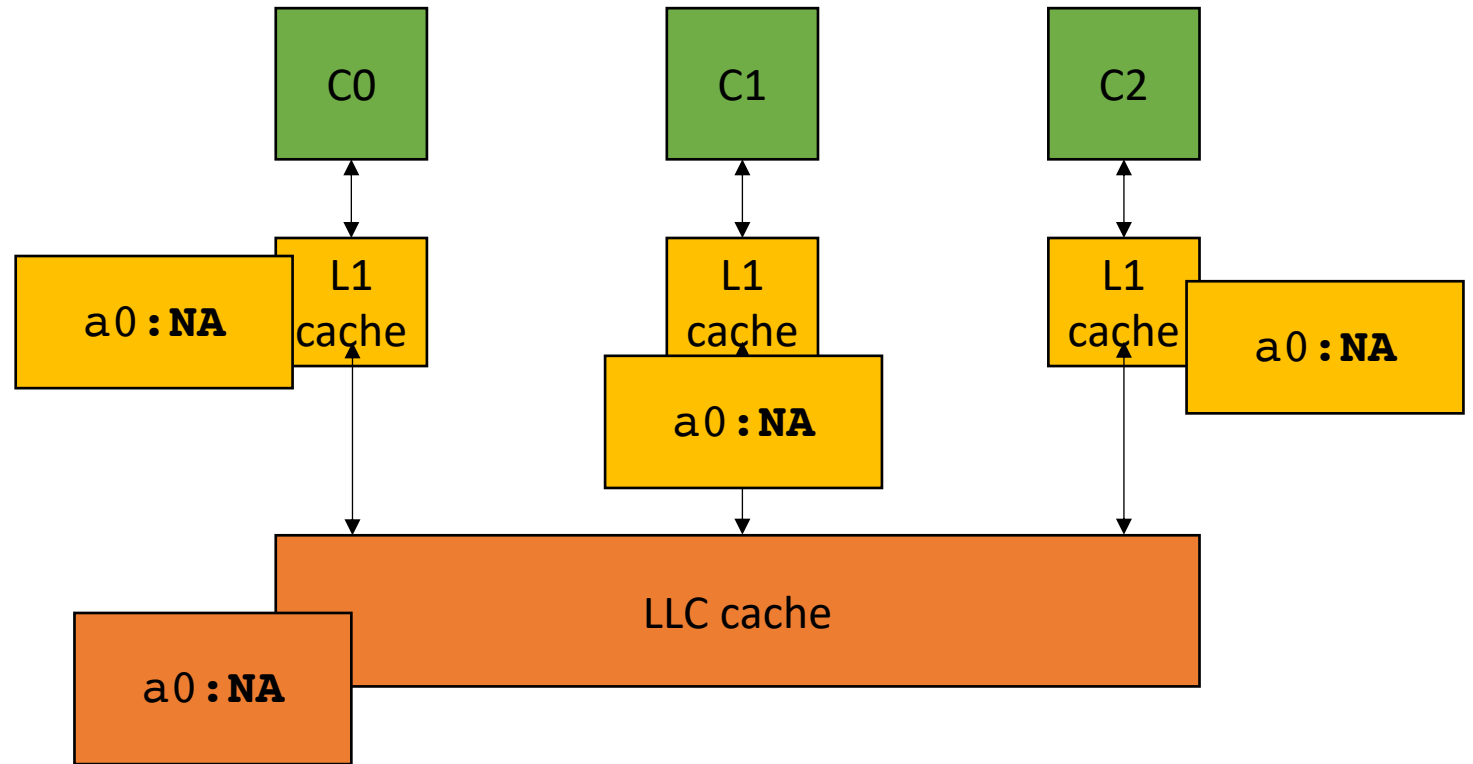
- For Intel Processors:
  - **L1** 8-way associative
  - **L2** 4-way associative
  - **L3** 12-way associative

# Cache coherence

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

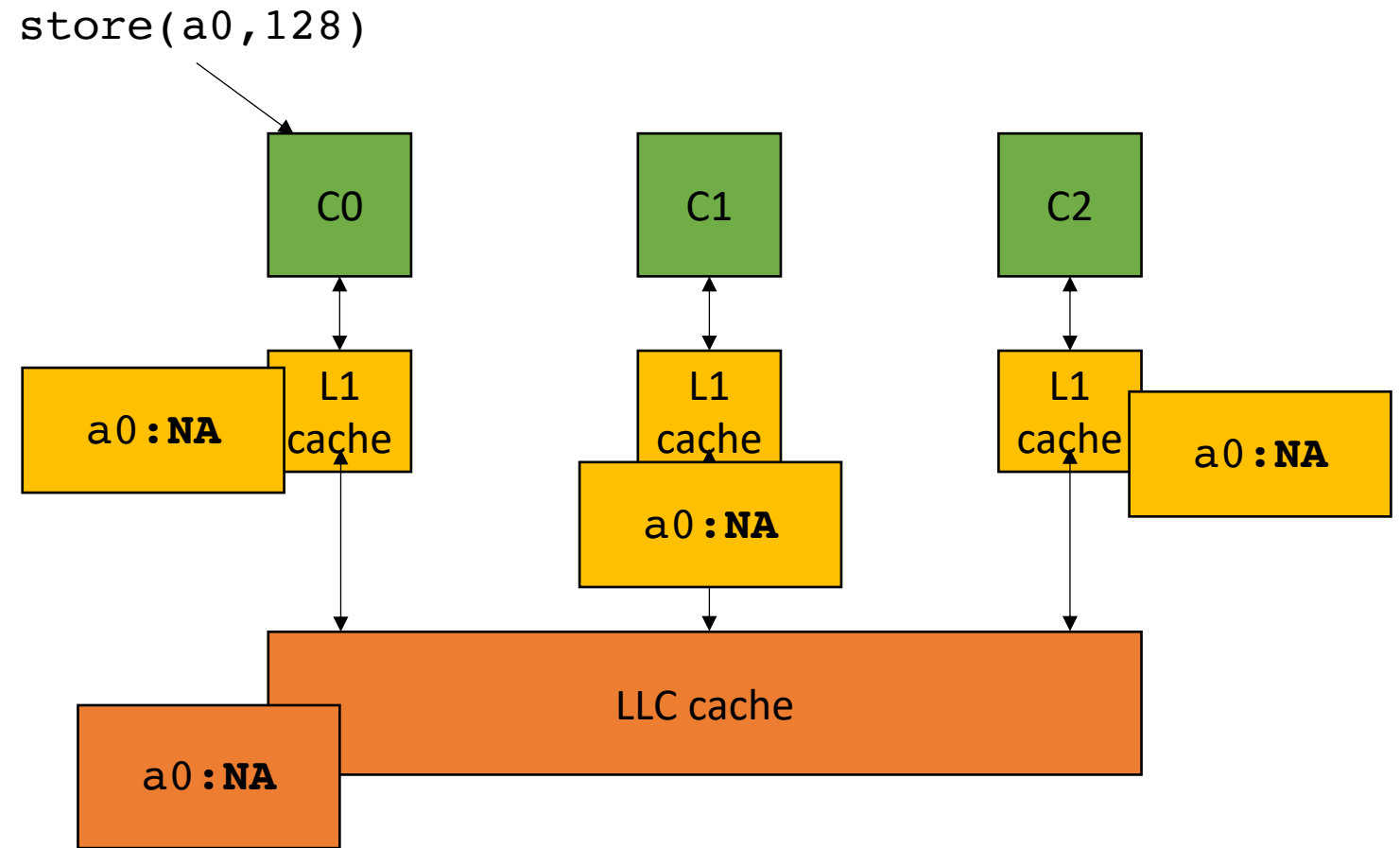
simplified view for illustration:  
L1 cache and LLC

Consider 3 cores accessing the same memory location

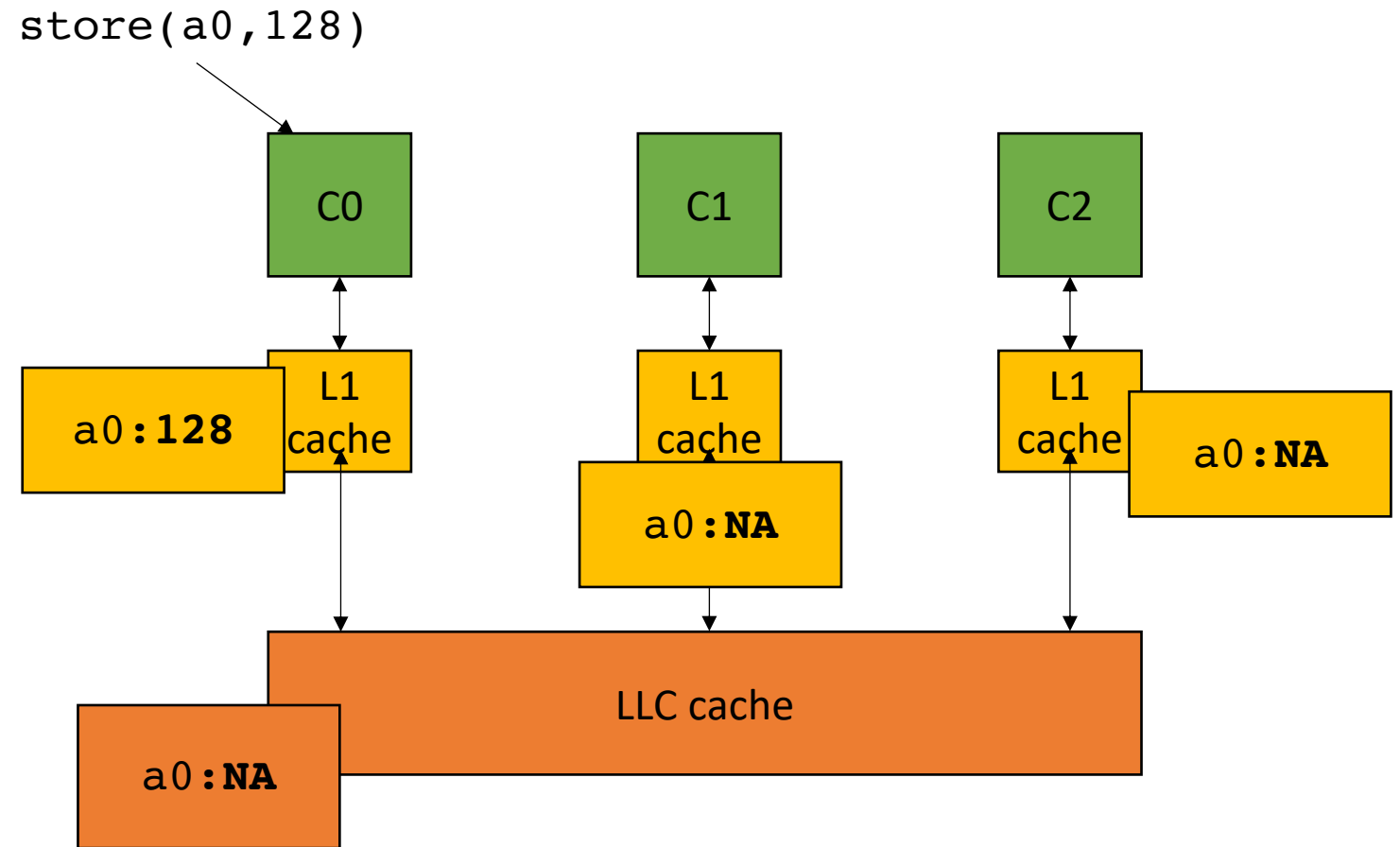




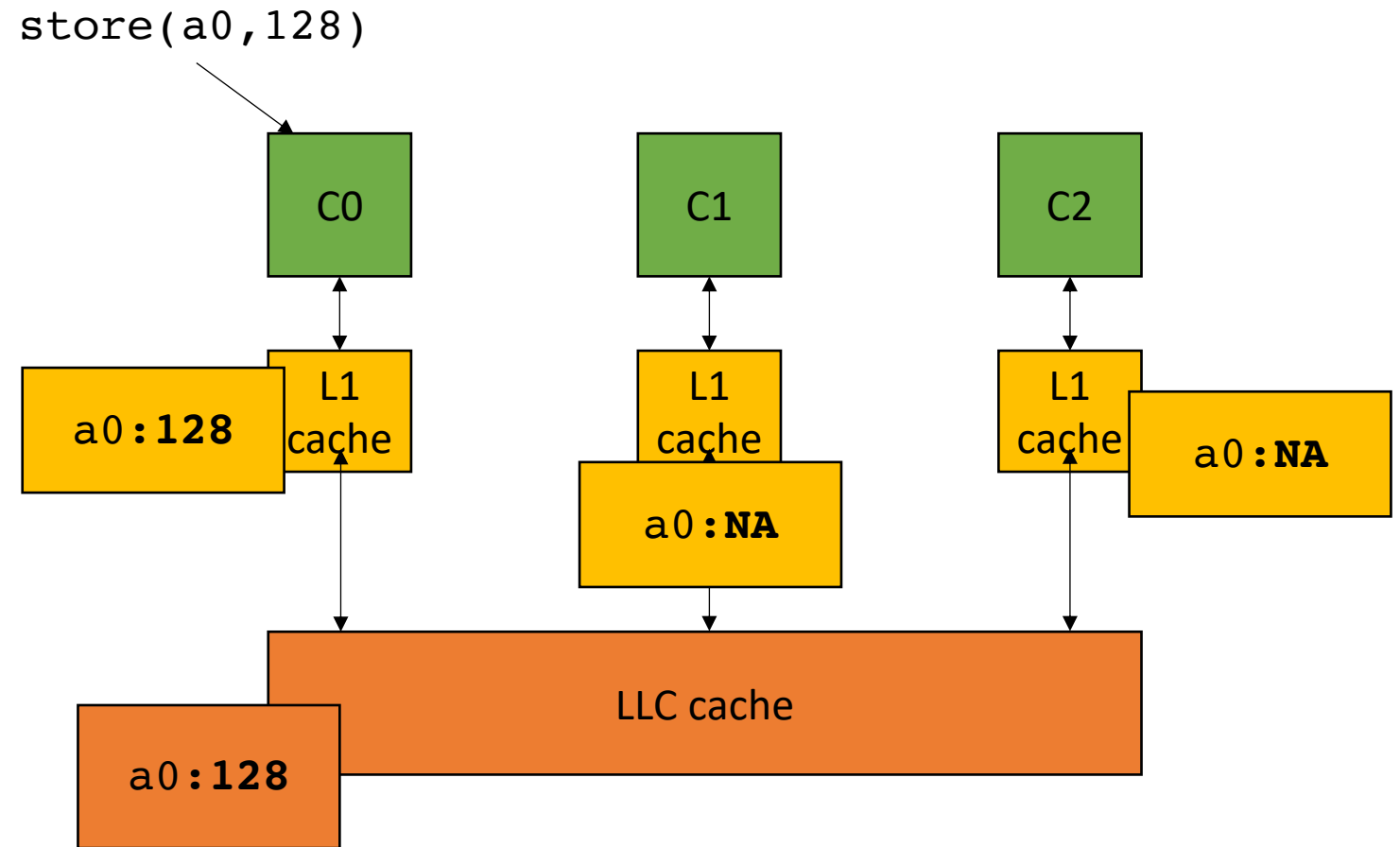
# Cache coherence



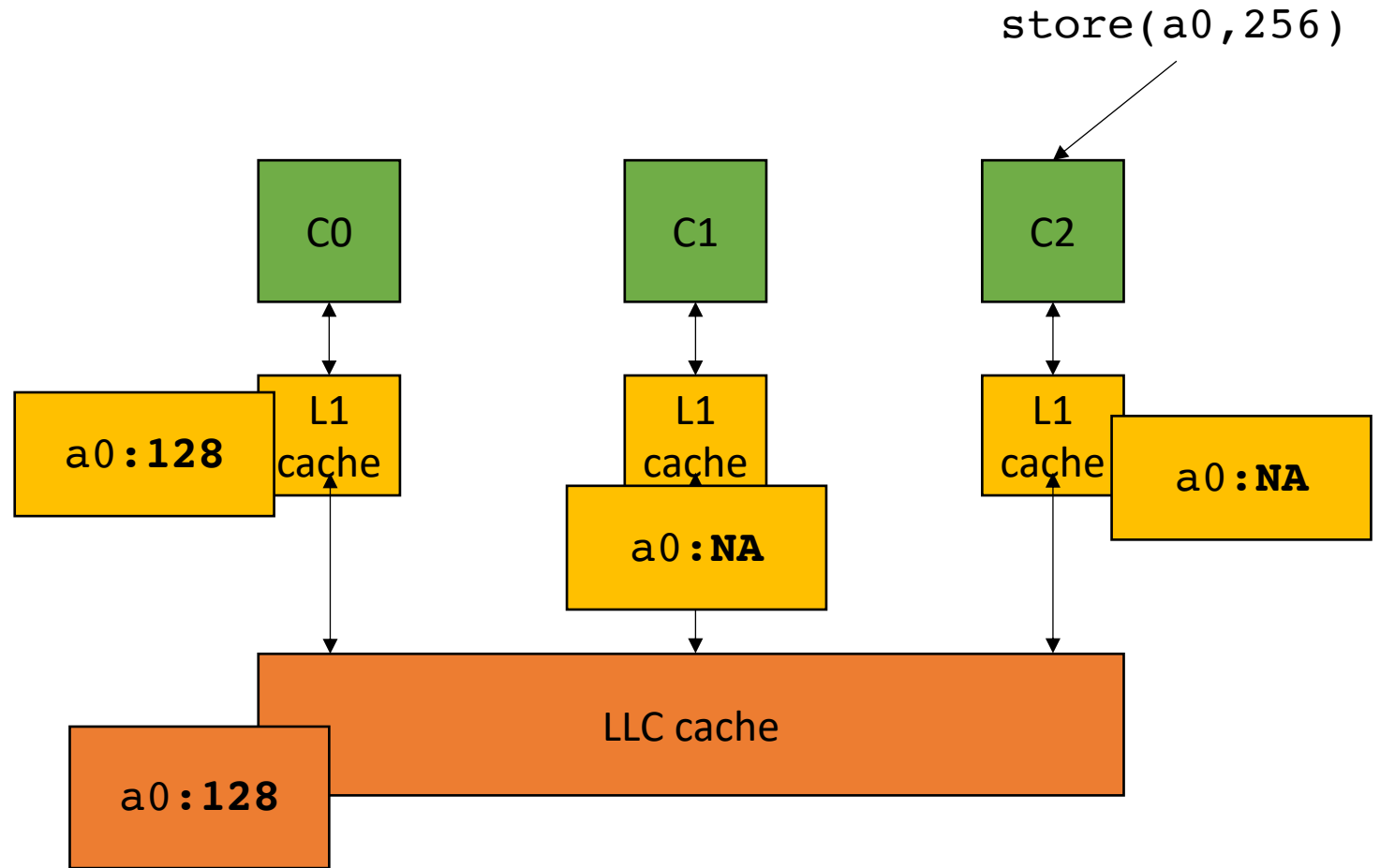
# Cache coherence



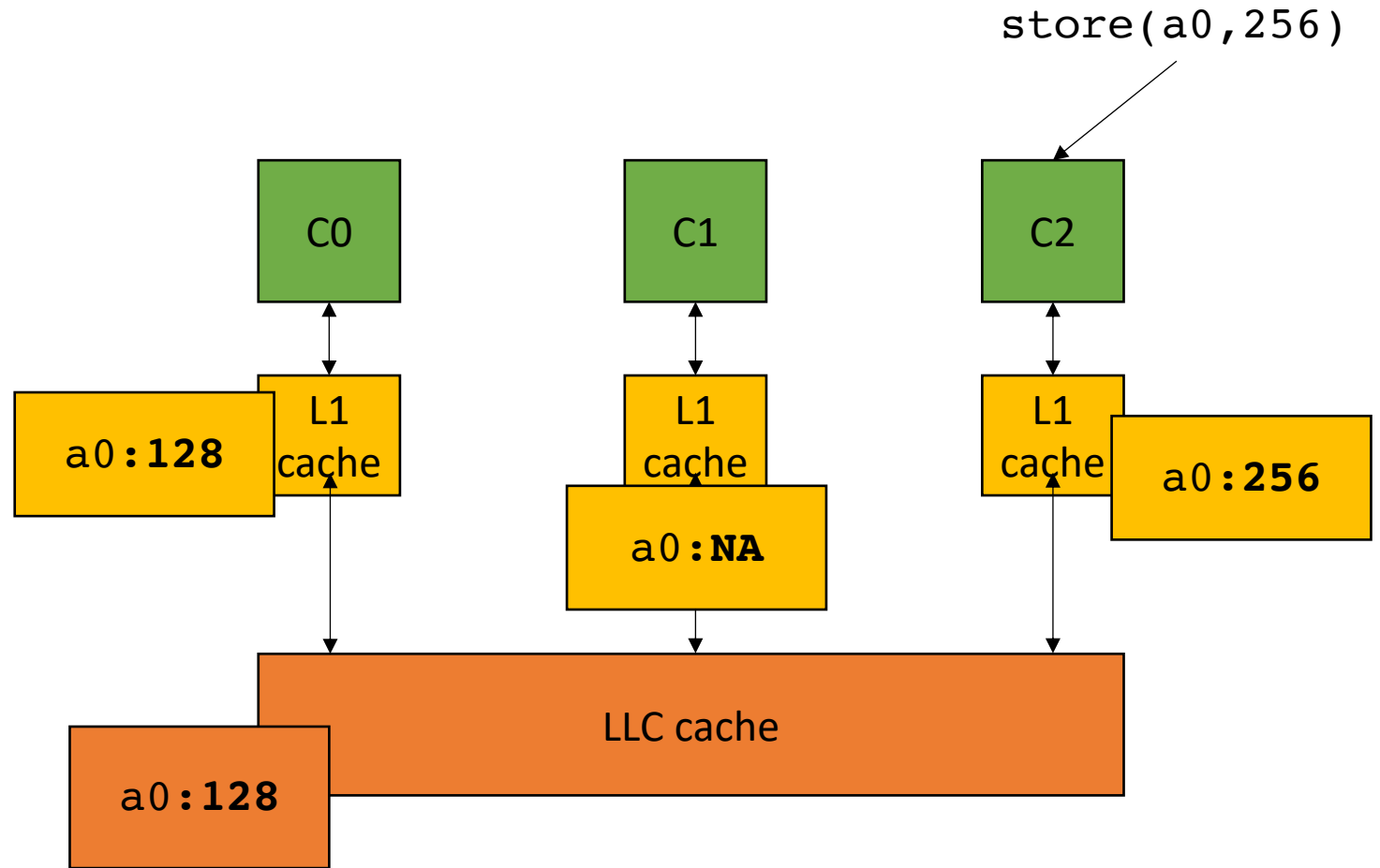
# Cache coherence



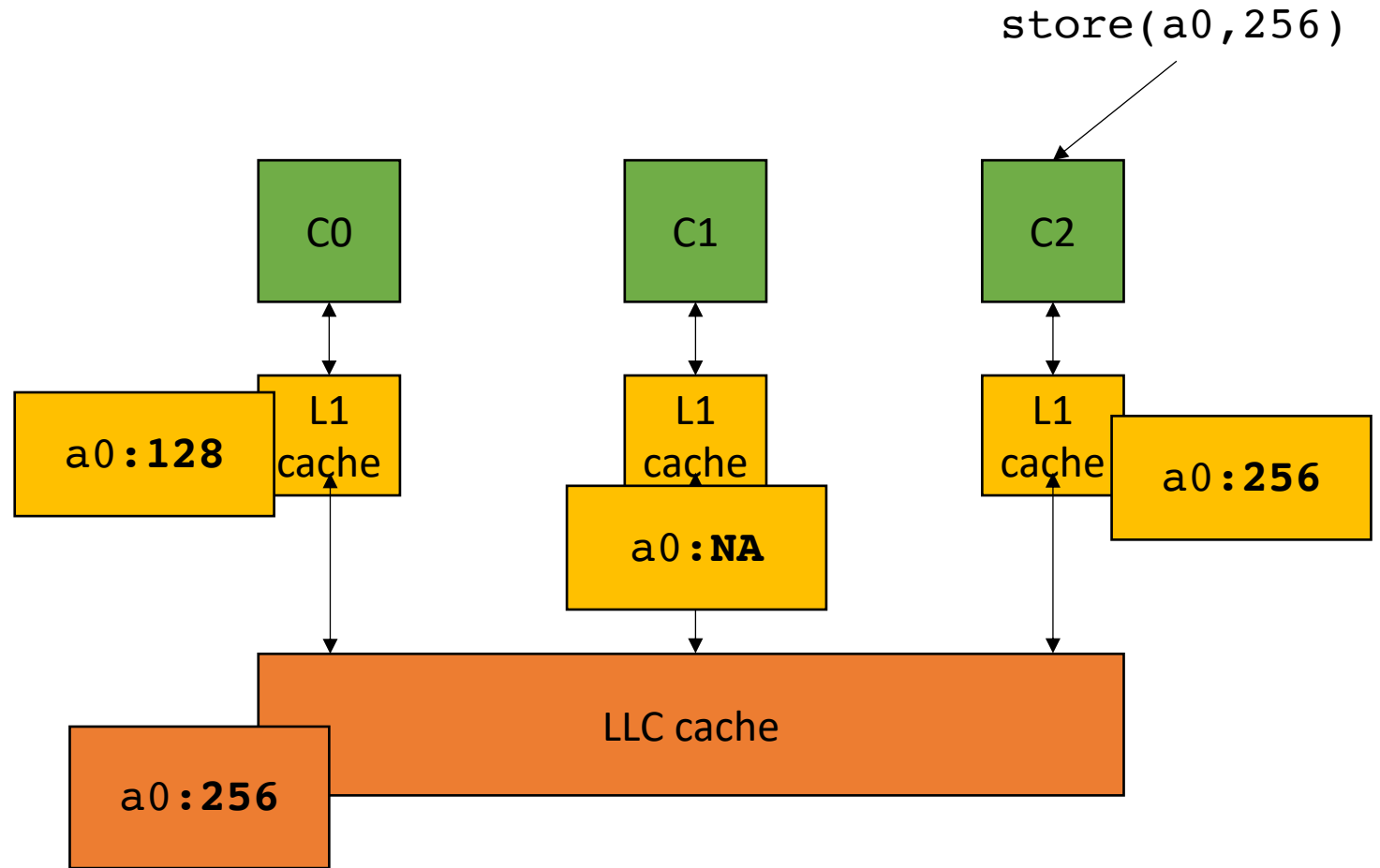
# Cache coherence



# Cache coherence

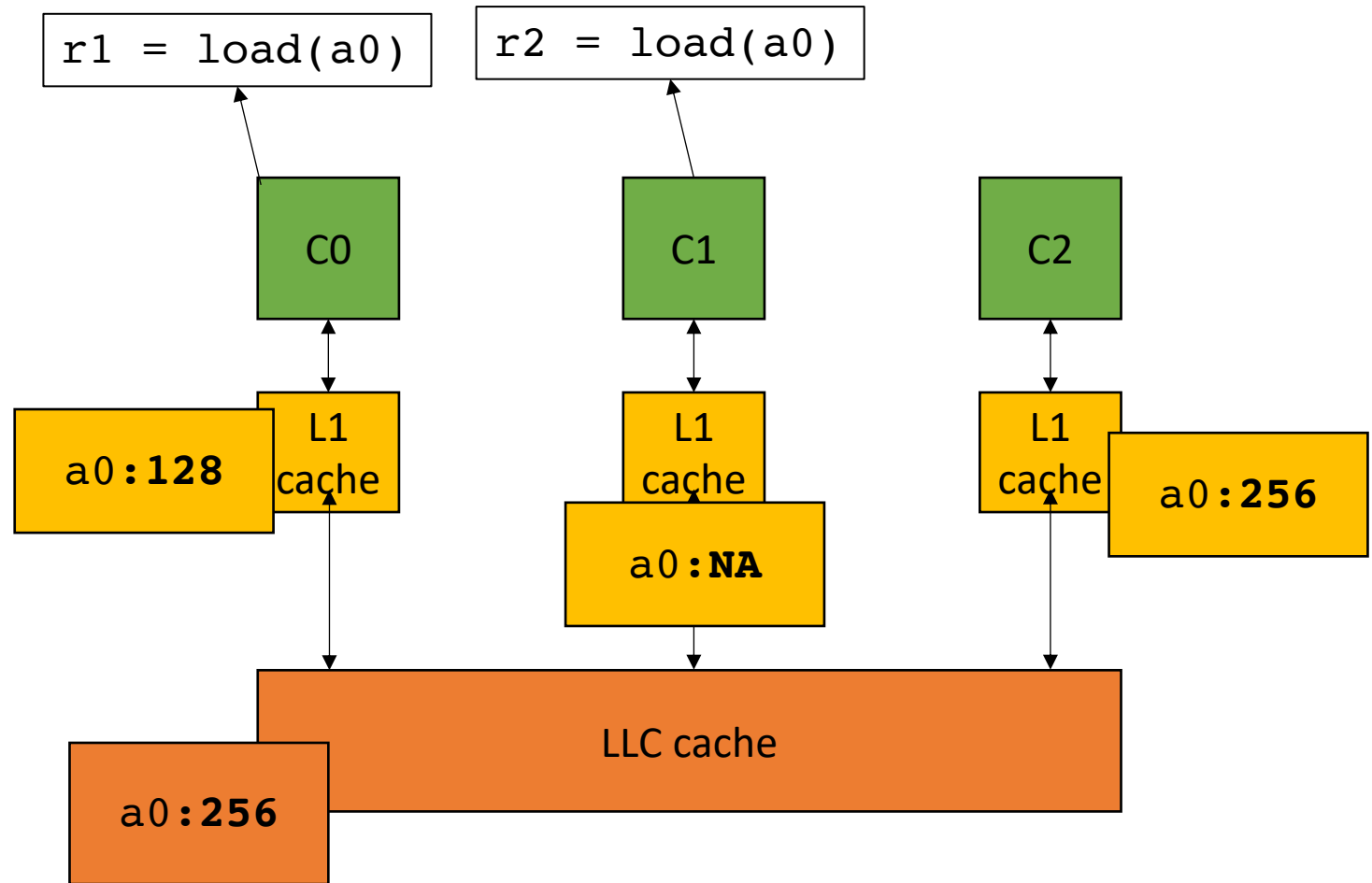


# Cache coherence

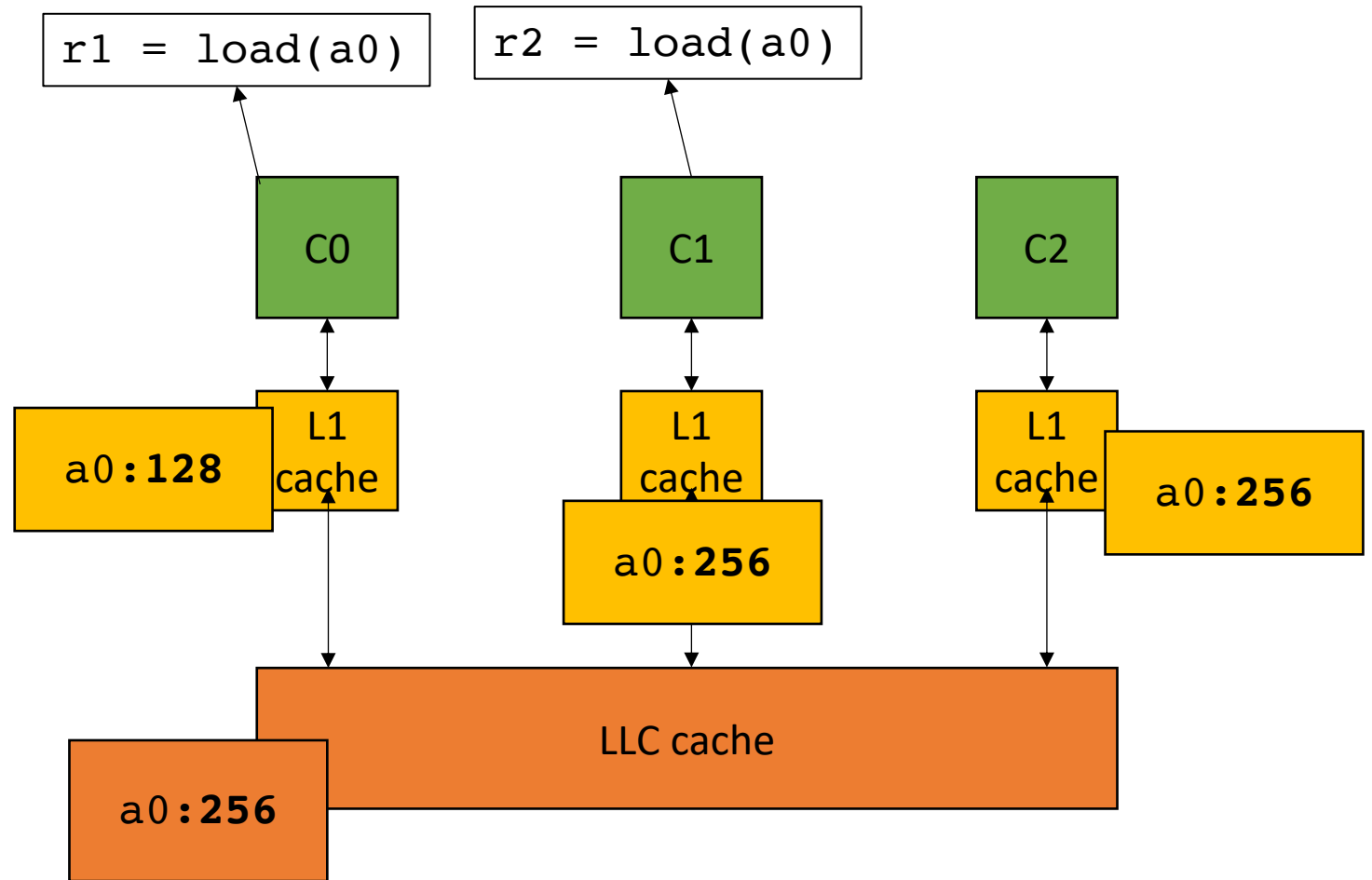


# Cache coherence

*in parallel*



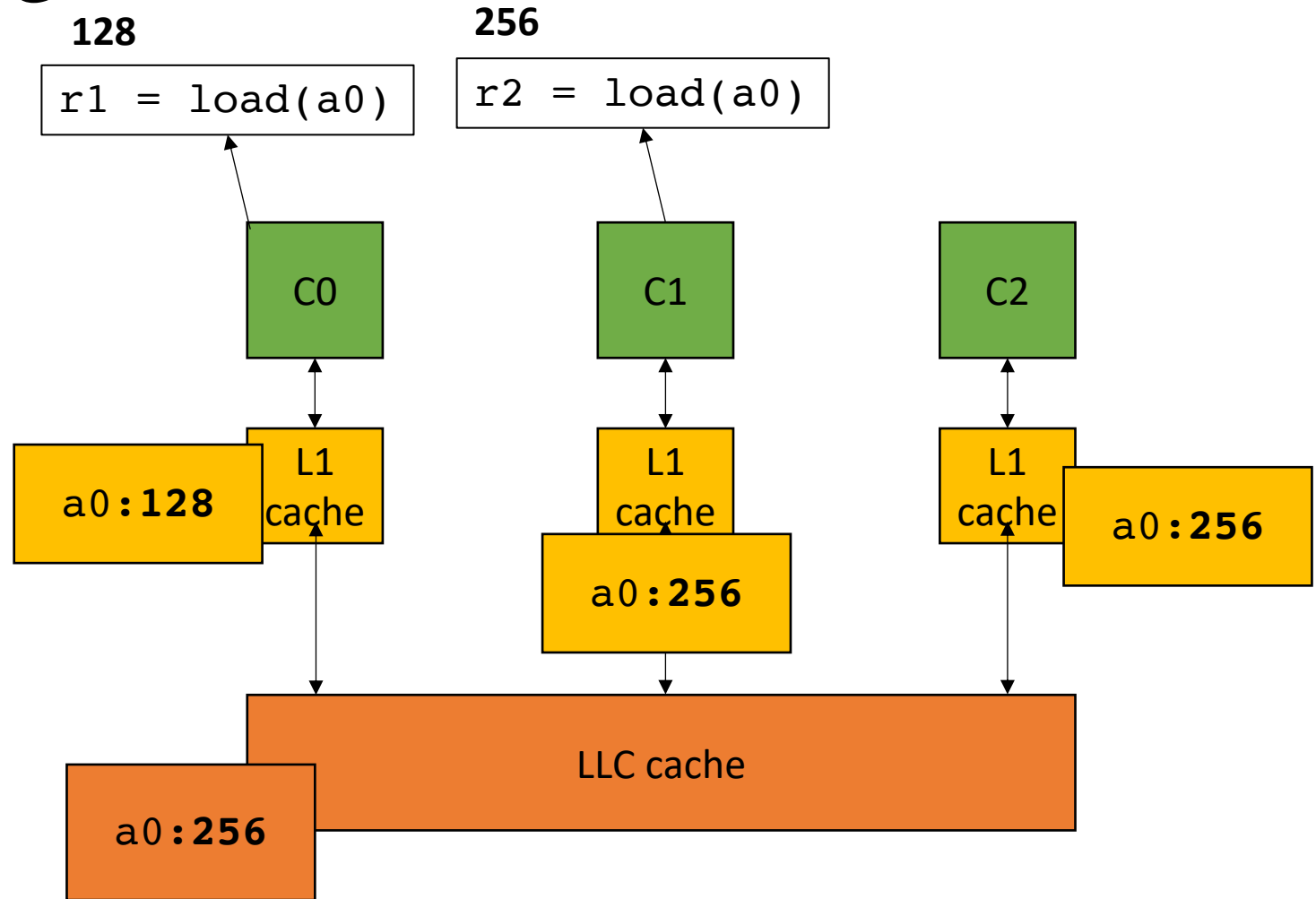
# Cache coherence





# Cache coherence

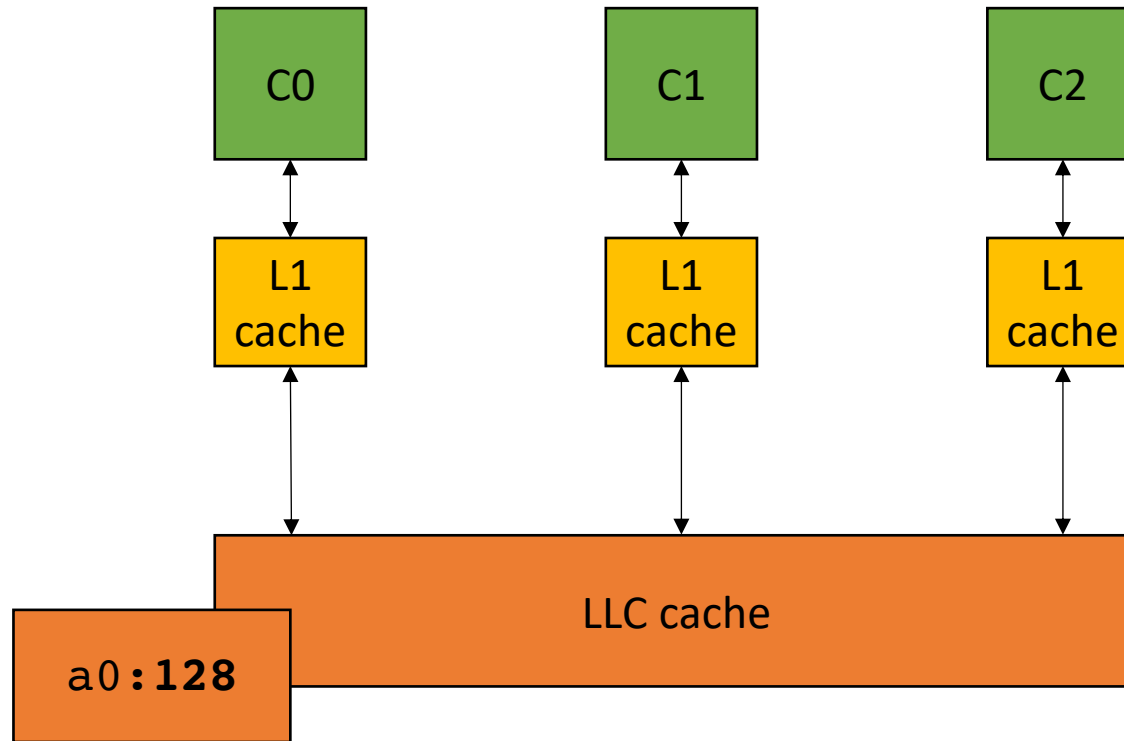
Incoherent view of values!



# Cache coherence

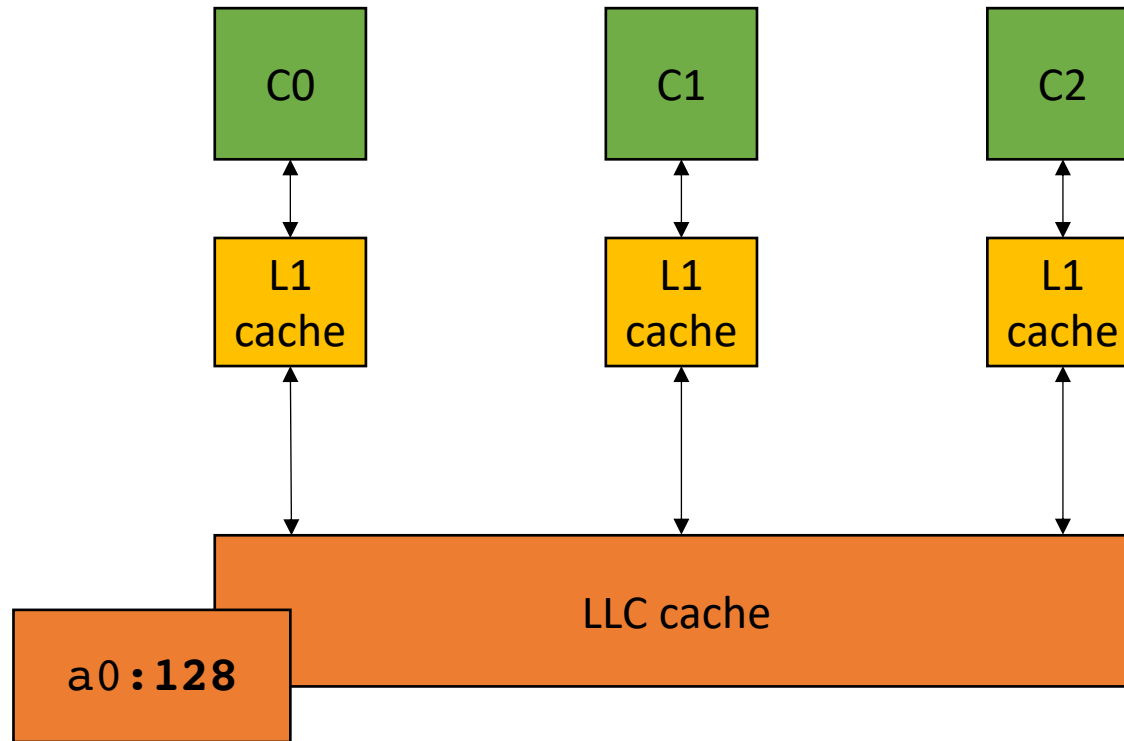
- MESI protocol
- Cache line can be in 1 of 4 states:
  - **Modified** - the cache contains a modified value and it must be written back to the lower level cache
  - **Exclusive** - only 1 cache has a copy of the value
  - **Shared** - more than 1 cache contains the value, they must all agree on the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

# Cache coherence

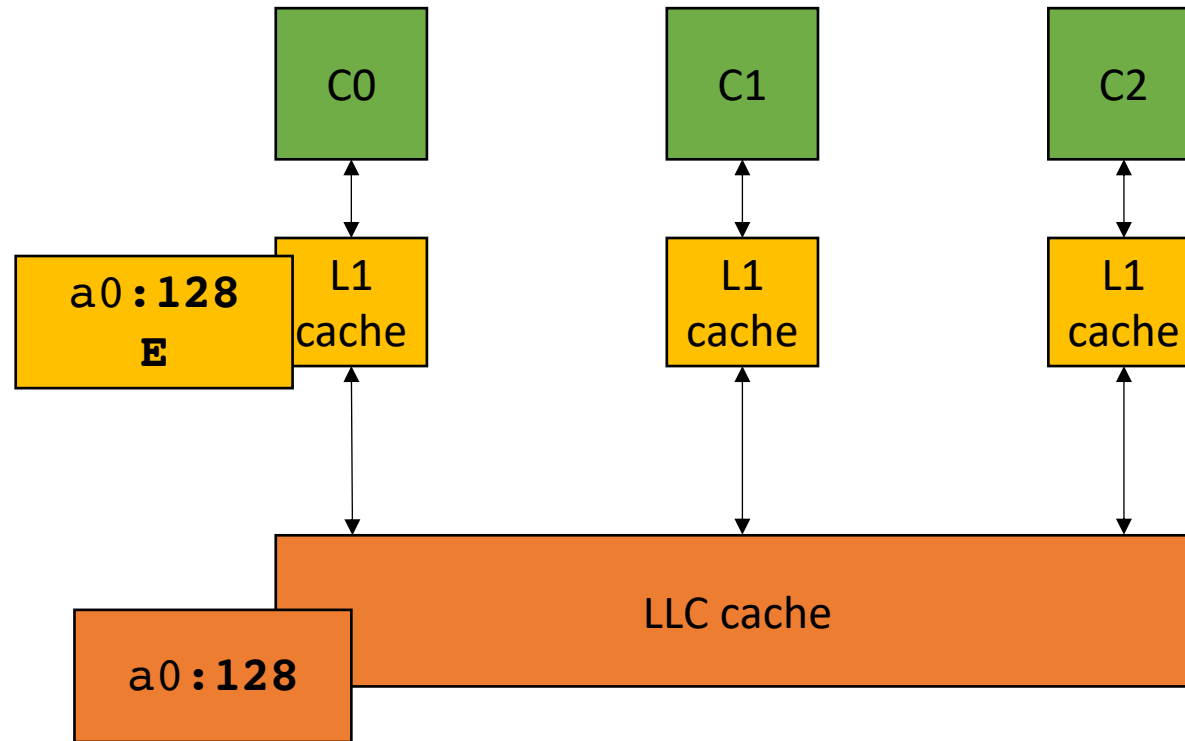


# Cache coherence

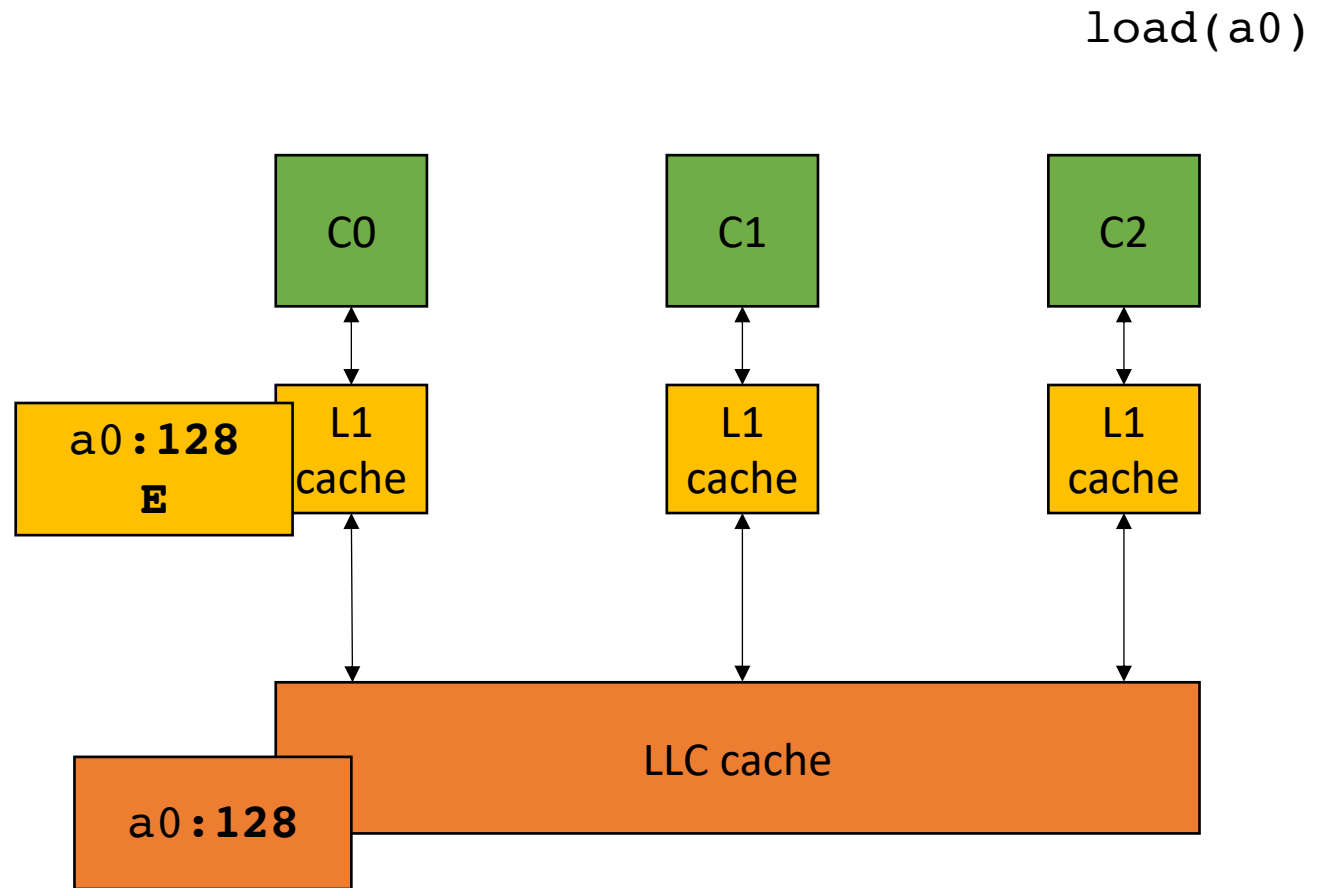
load(a0)



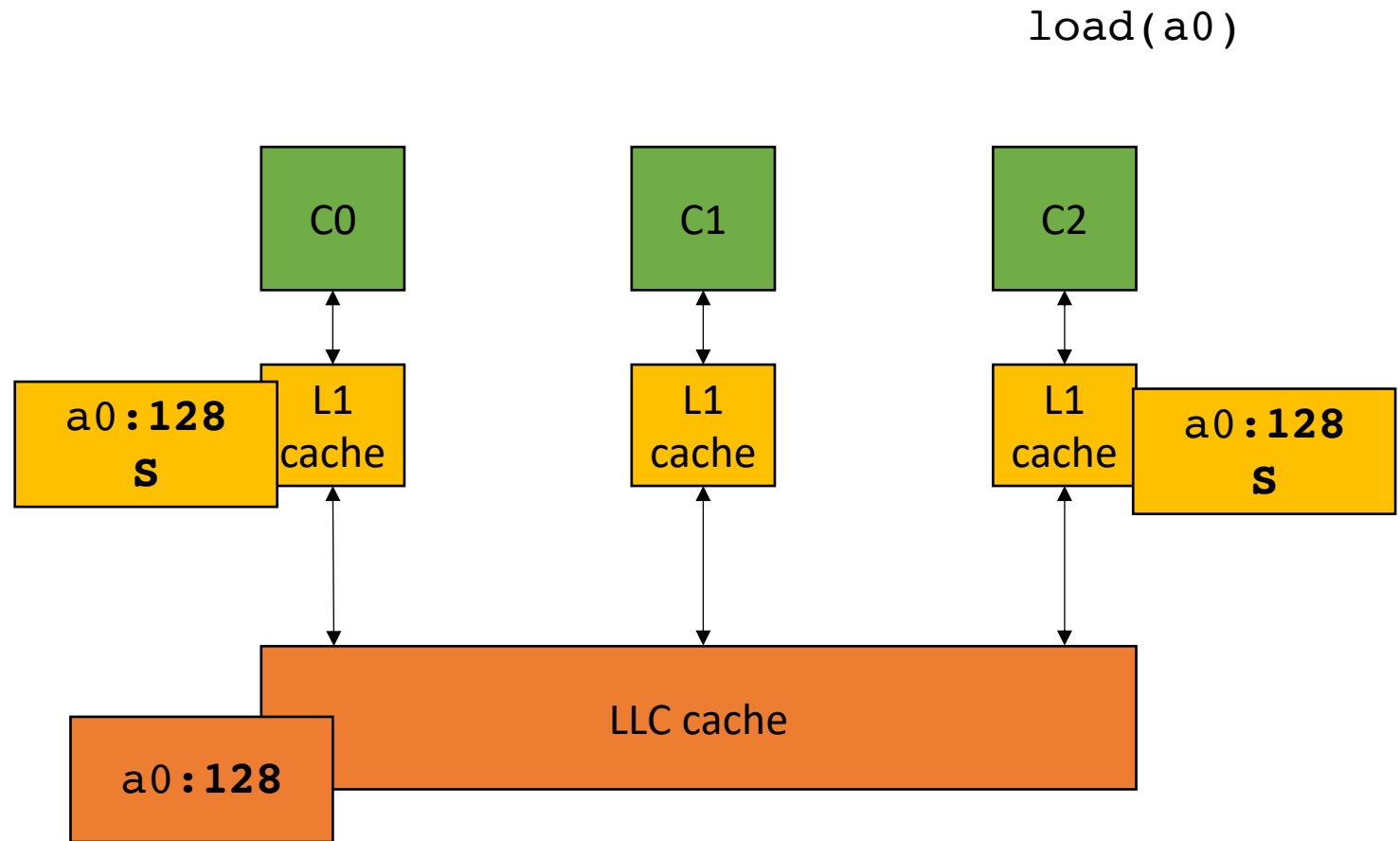
# Cache coherence



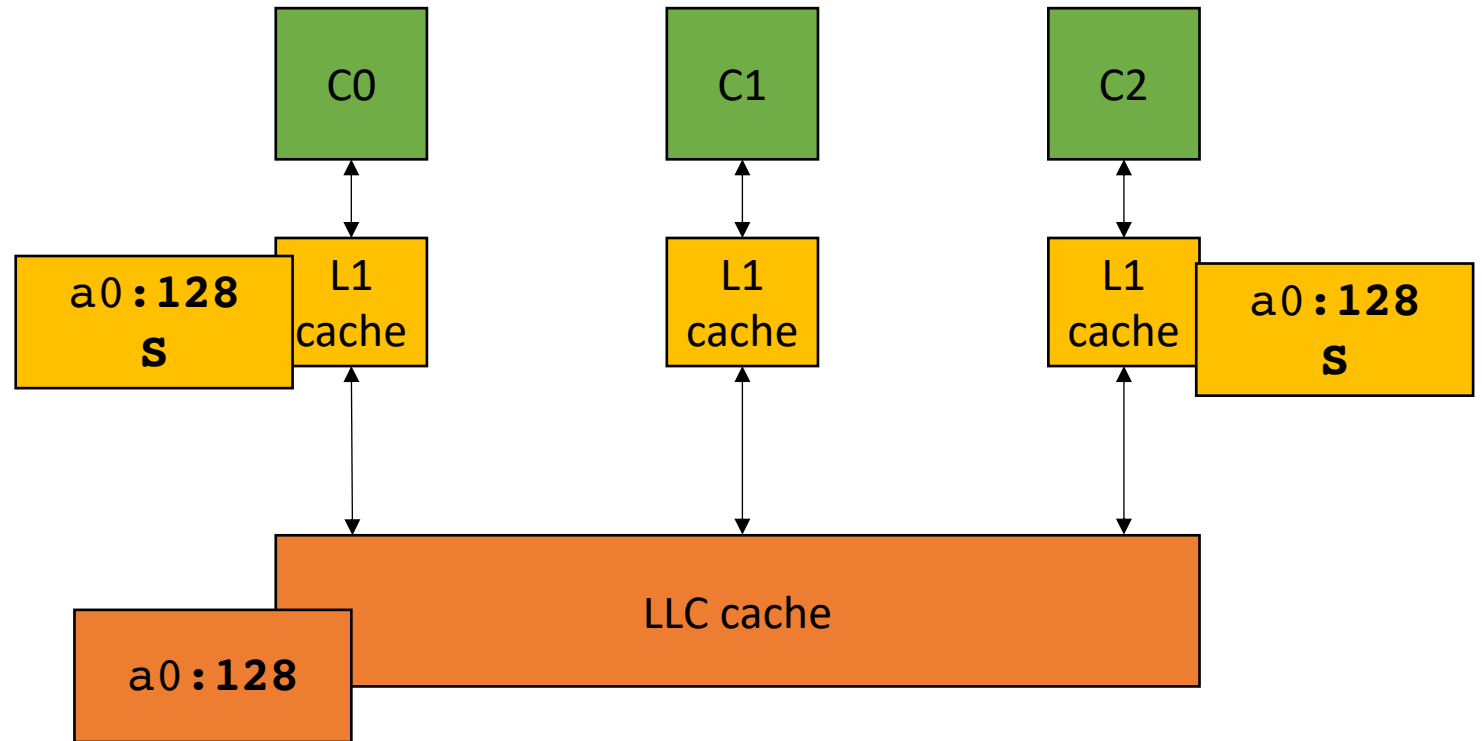
# Cache coherence



# Cache coherence

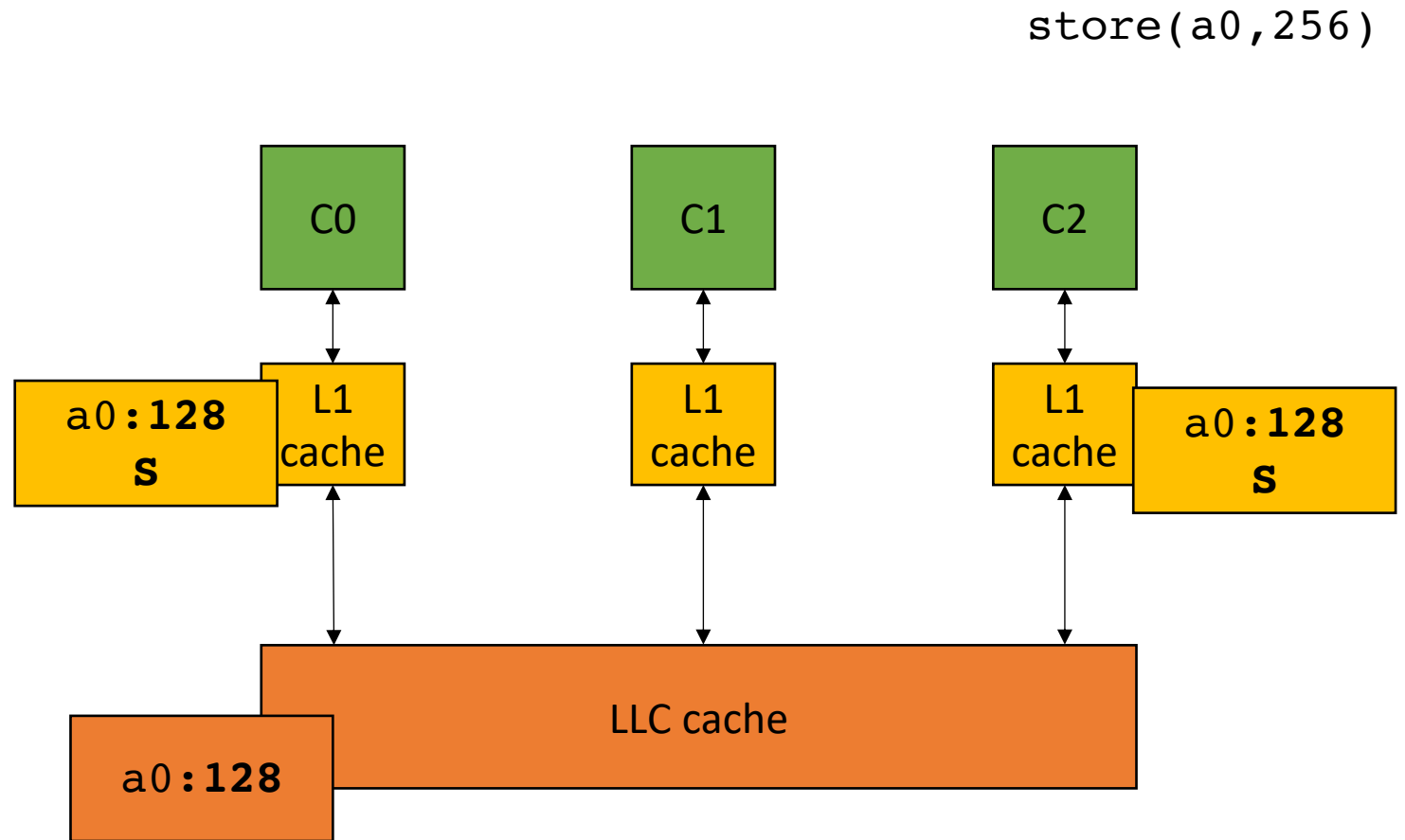


# Cache coherence

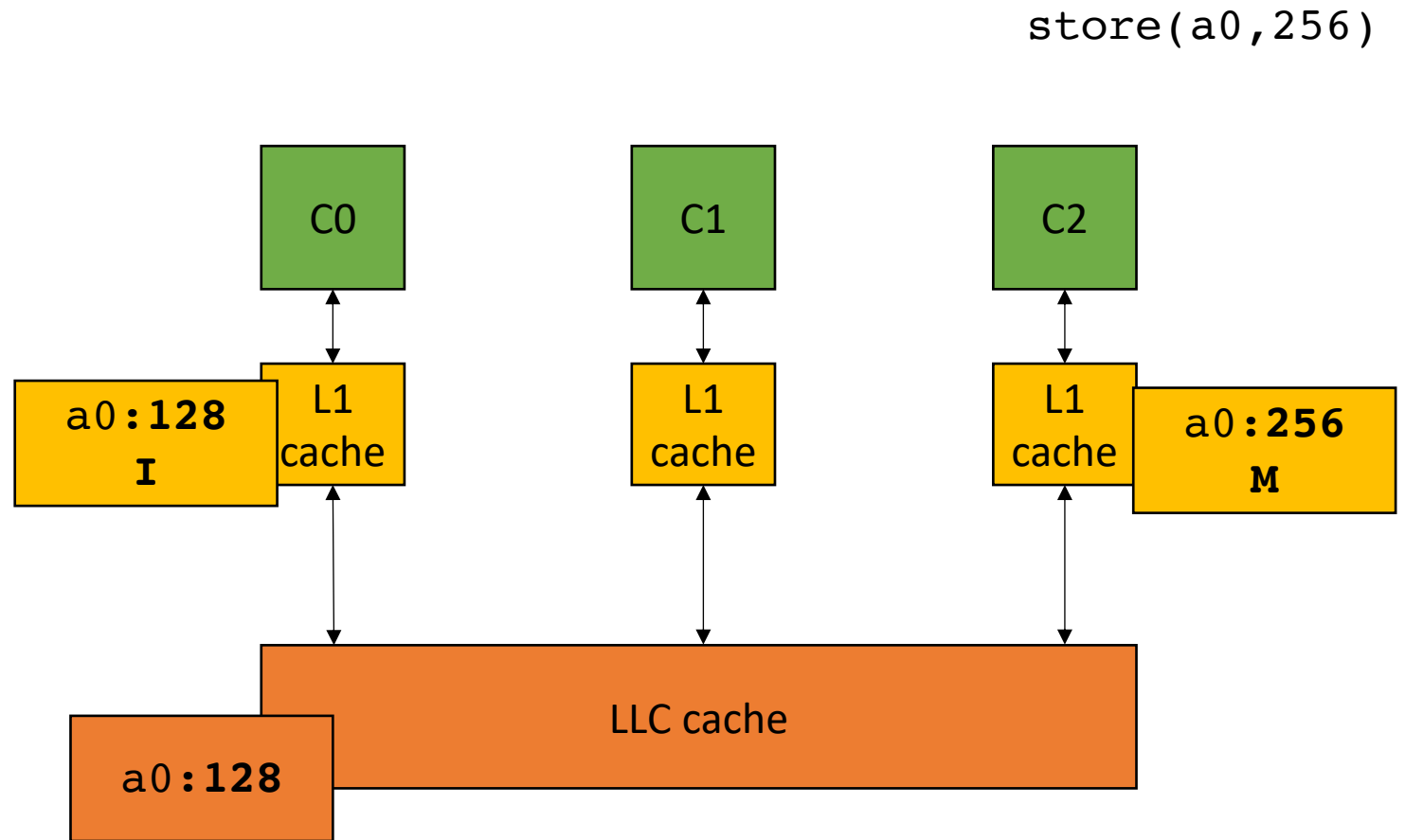




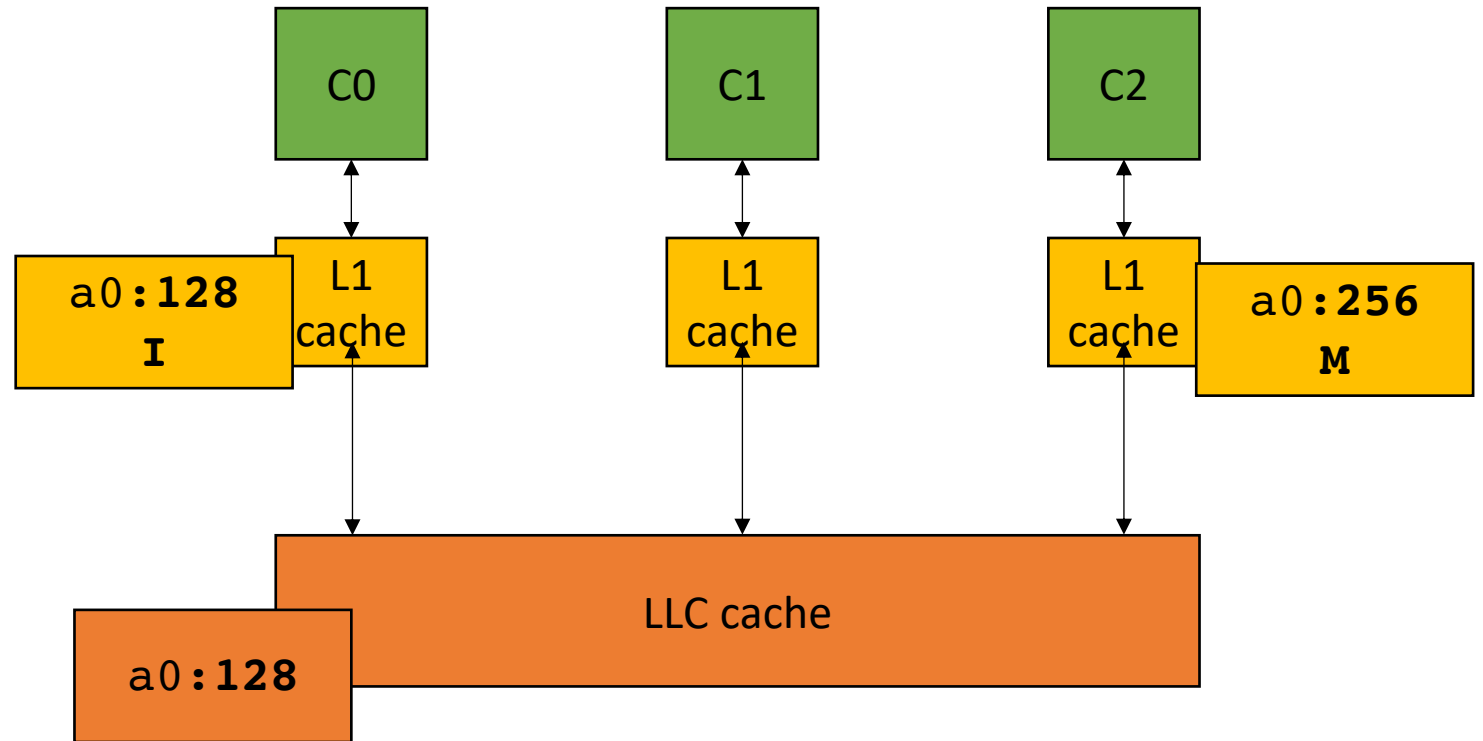
# Cache coherence



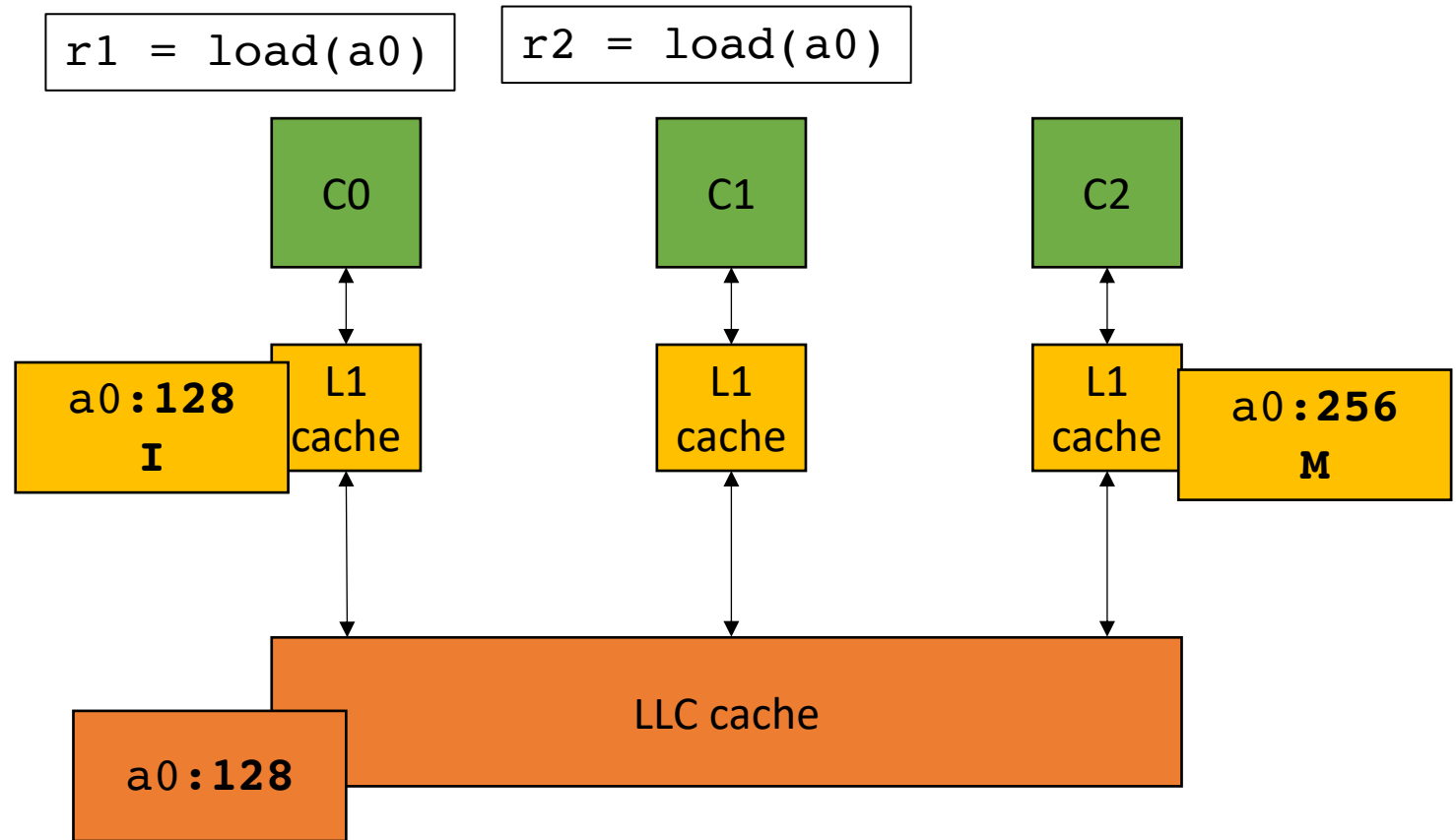
# Cache coherence



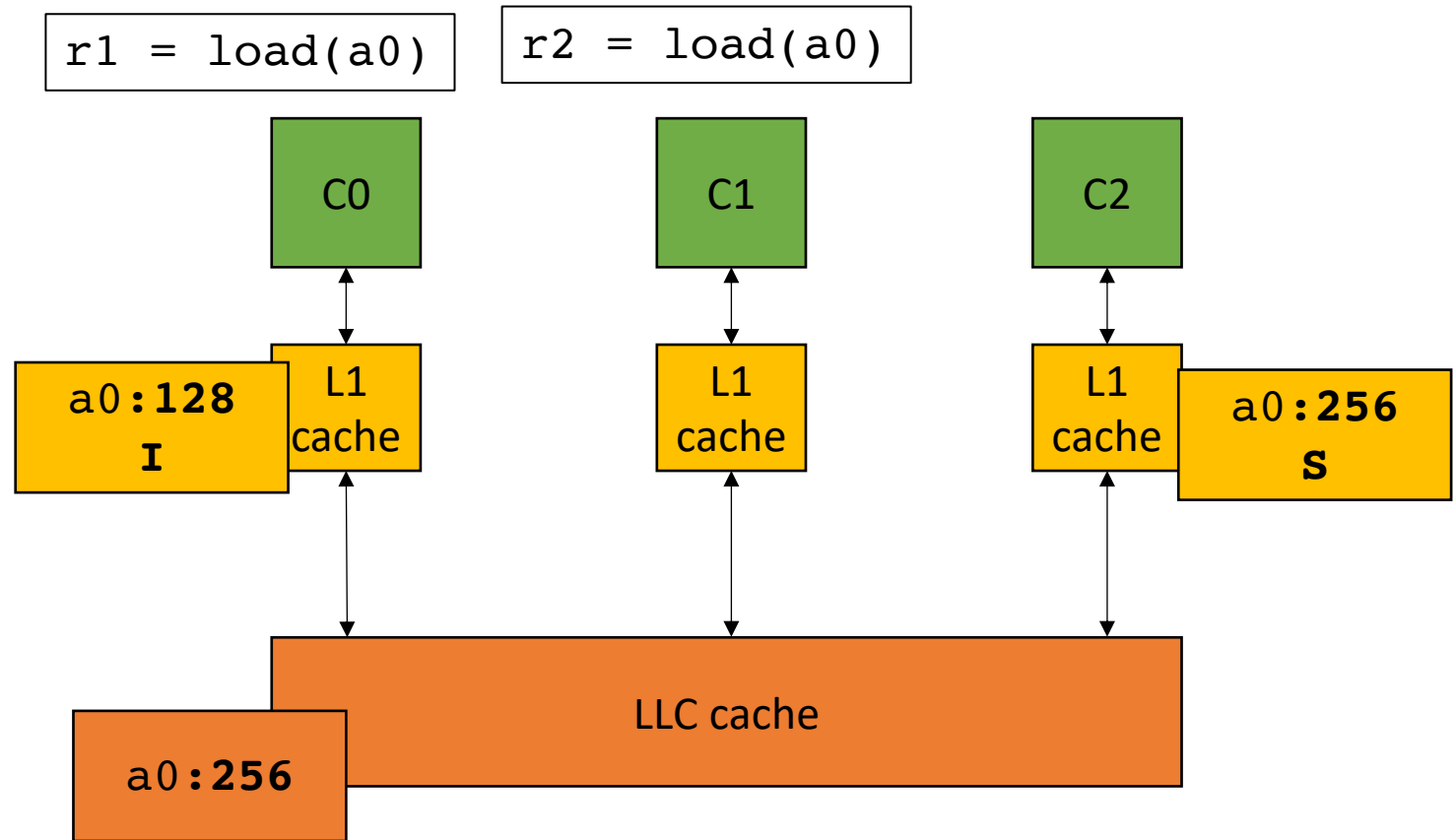
# Cache coherence



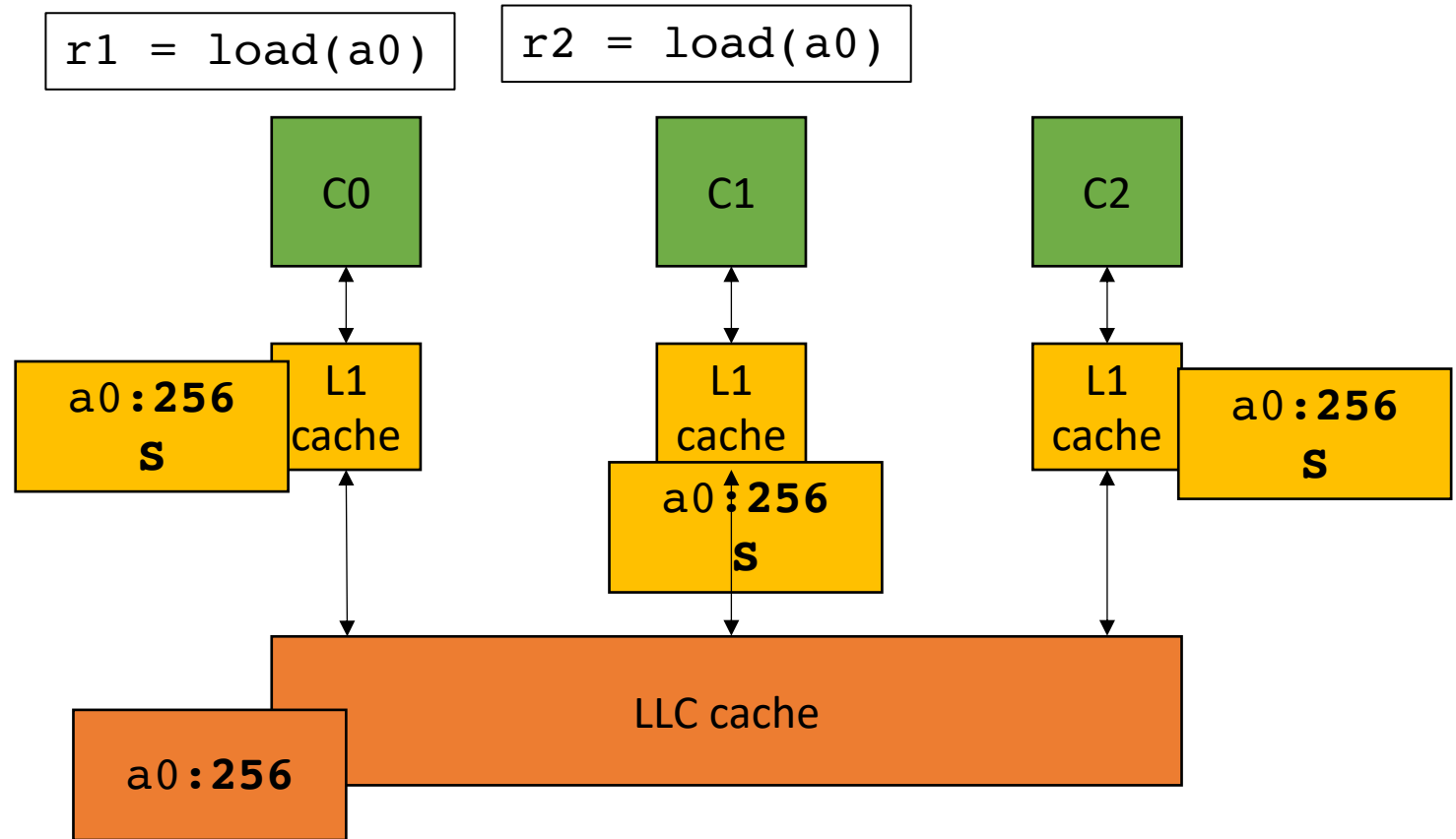
# Cache coherence



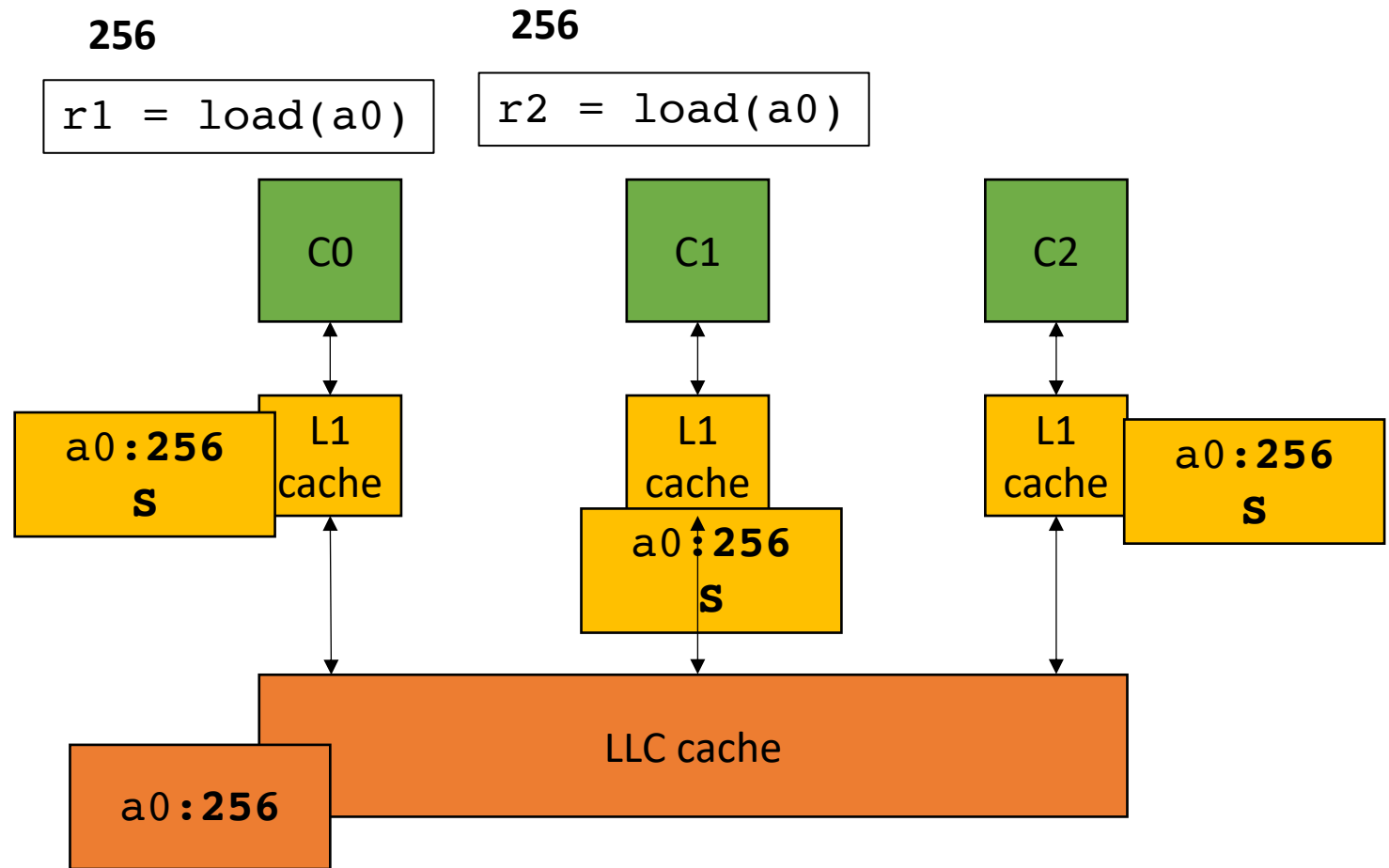
# Cache coherence



# Cache coherence



# Cache coherence



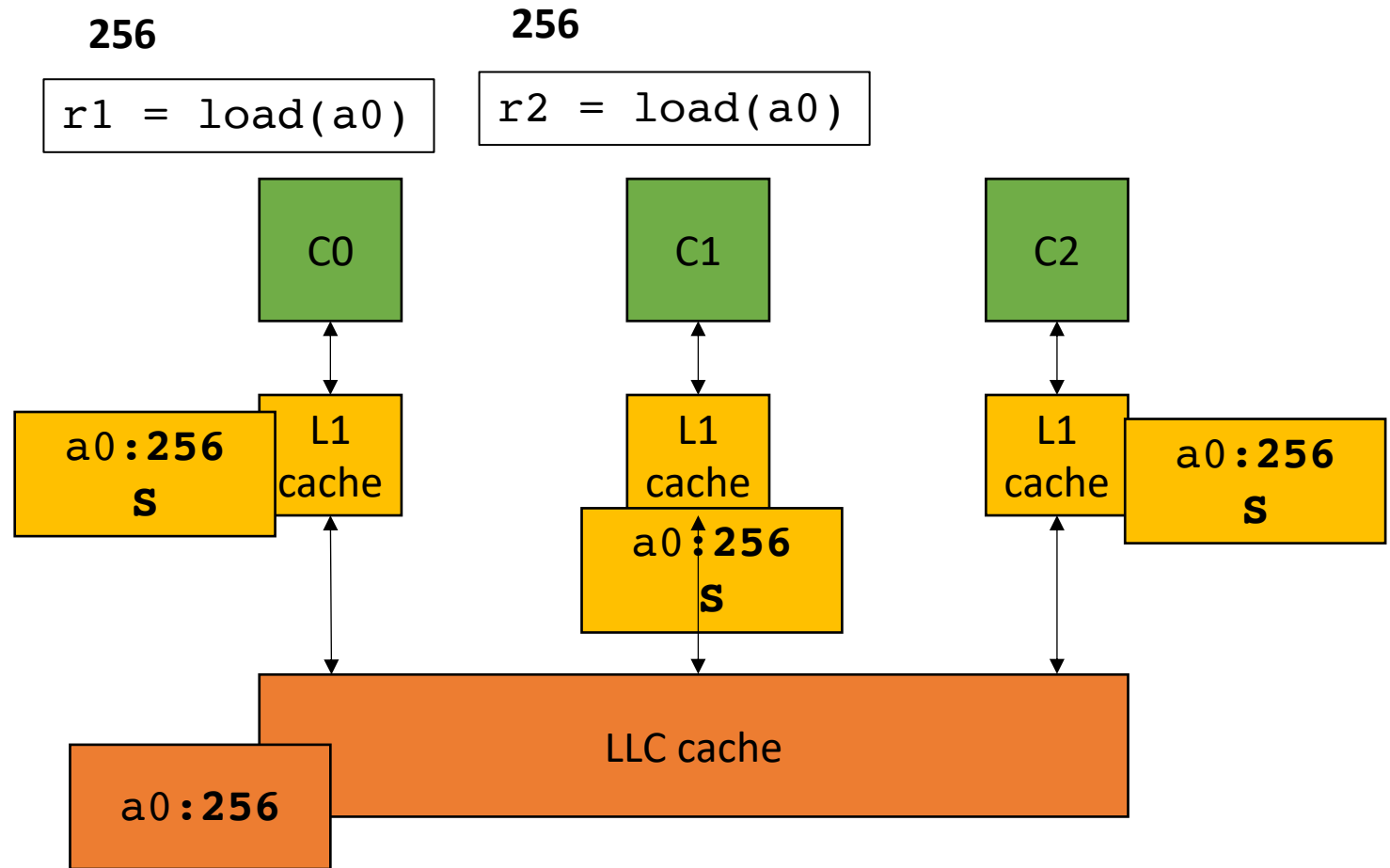
# Cache coherence

## 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





# Lecture Schedule

- Overview - why do we need a lecture on compilation and architecture?
- Compilation - How do we translate a program from a human-accessible language to a language that the processor understands
- Architecture - How do processors execute programs?
- **Example**

# Example

- A function that increments a memory location ITERATION times

```
void repeat_increment(volatile int *a) {  
    for (int i = 0; i < ITERATIONS; i++) {  
        int tmp = *a;  
        tmp += 1;  
        *a = tmp;  
    }  
}
```

# Example

- A function that increments a memory location ITERATION times

*guarantees that memory accesses are not optimized!*

```
void repeat_increment(volatile int *a) {  
    for (int i = 0; i < ITERATIONS; i++) {  
        int tmp = *a;  
        tmp += 1;  
        *a = tmp;  
    }  
}
```

# Example

- A function that increments a memory location ITERATION times
- Do this for 8 elements:
  - Allocate a contiguous array

# Example

- A function that increments a memory location ITERATION times
- Do this for 8 elements:
  - Allocate a contiguous array
- Loop through the 8 elements and increment each one:

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    repeat_increment(a+i);  
}
```

# Example

- We can also do each array element in parallel!

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    repeat_increment(a+i);  
}
```

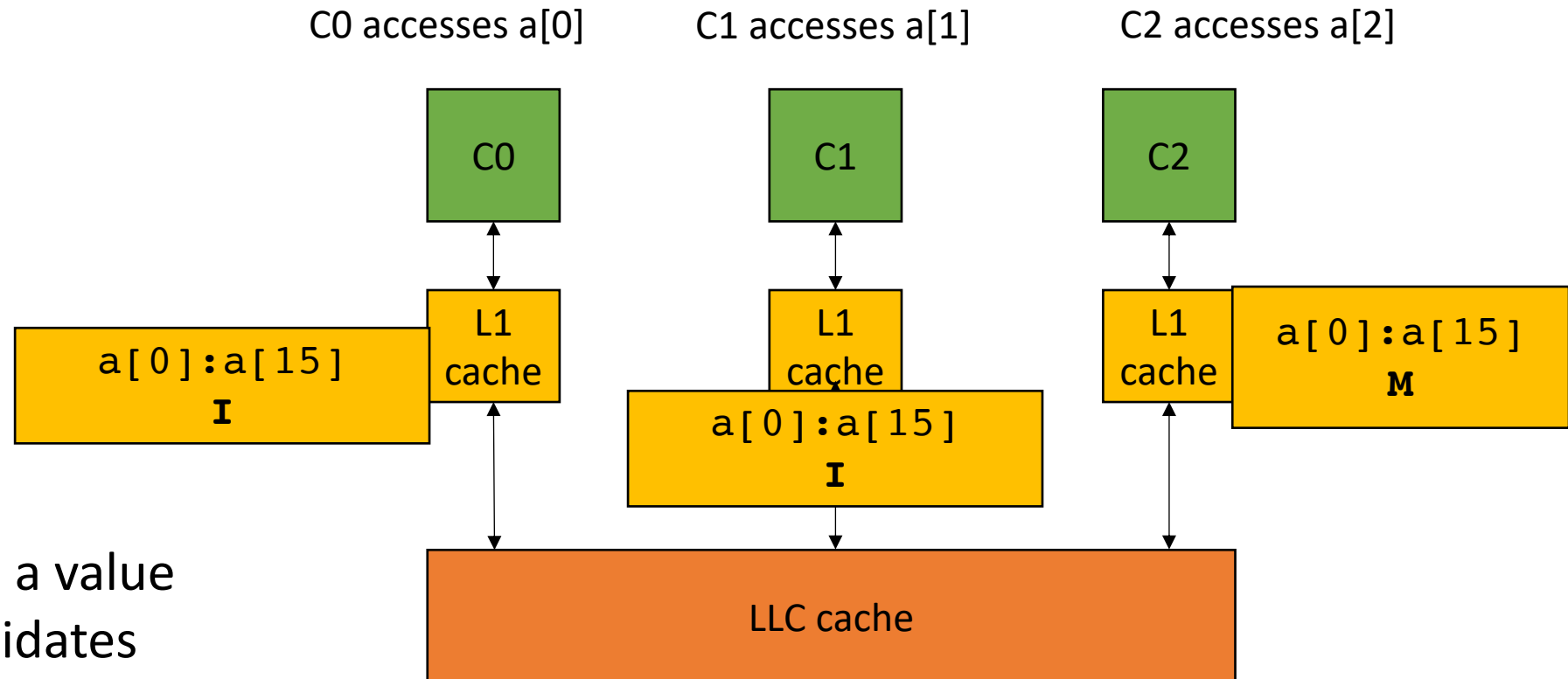
```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    thread(repeat_increment, a+i);  
}
```

*Don't worry, we will go over C++ thread  
in more detail on Thursday*

# Example

- Run example

# What's going on?



when one core modifies a value  
in the cache line, it invalidates  
everyone else's cache line.

This is called ***False Sharing***



Fix?

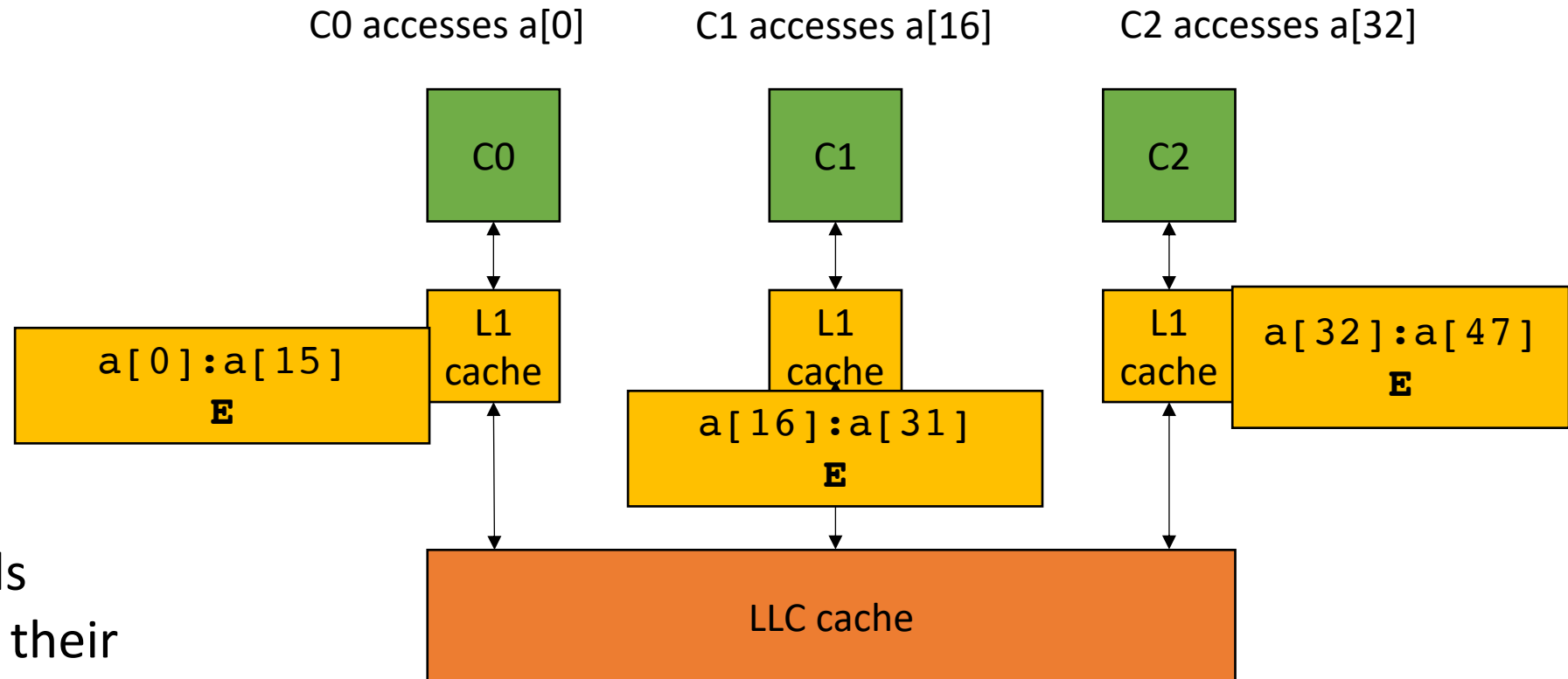
# Fix?

- **Padding:** give each element its own cache line:
  - Recall cache line is size 16 ints, so we will use 16x more memory

```
int a[NUM_ELEMENTS * 16];
```

```
for (int i = 0; i < NUM_ELEMENTS; i++) {  
    thread(repeat_increment, a+(i*16));  
}
```

# What's going on?



With padding, all threads have exclusive access to their lines! No need to trigger invalidations or write-back each operation

# Thank you!

- Remember to do the quiz today!
- Homework will be released by the end of class today
  - Due in two weeks
  - Just work on getting Docker up and going!
- We will discuss ILP and C++ threads next week
- Have a good weekend: go do something fun!