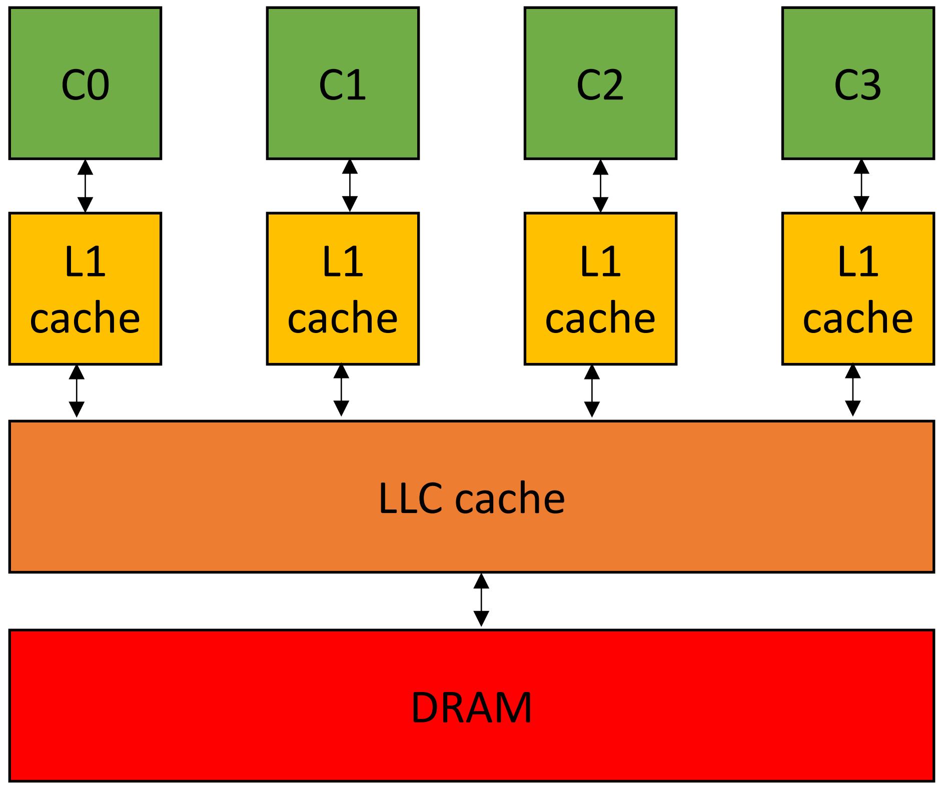
CSE113: Parallel Programming

Jan. 13, 2023

- **Topic**: Architecture and Compiler Overview
  - Cache associativity
  - Cache coherence
  - False sharing



Announcements

Parallel programming video game user study

Quick announcement from Jennifer Villareale

Asynchronous Forums

- Piazza is setup
- Unofficial discord:
  - we’re trusting you to moderate
  - be nice
  - don’t cheat

Office hours

- **Tyler:**
  - Thursday from 3:00 - 5:00 PM
  - Hybrid (remote or in person)
  - Room E2 233
- **Everyone else: coming by monday**

Homework 1

- Homework 1: Planned to be released on Monday (by midnight)
    - Might be delayed due to the new setup, but we are trying our hardest!
    - Due on Thursday Jan. 26 (or 10 days after it is released)
- We will be using a queue system for the submission. If everyone does their work at the same time (e.g. the night of the deadline), it will take longer to get the automatic feedback!
- You can do a lot of the work locally and just use the server to get results for your report.

What you can get started with: Docker

Instructions here:

https://sorensenucsc.github.io/CSE113-wi2023/docker-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.

Note on quizes

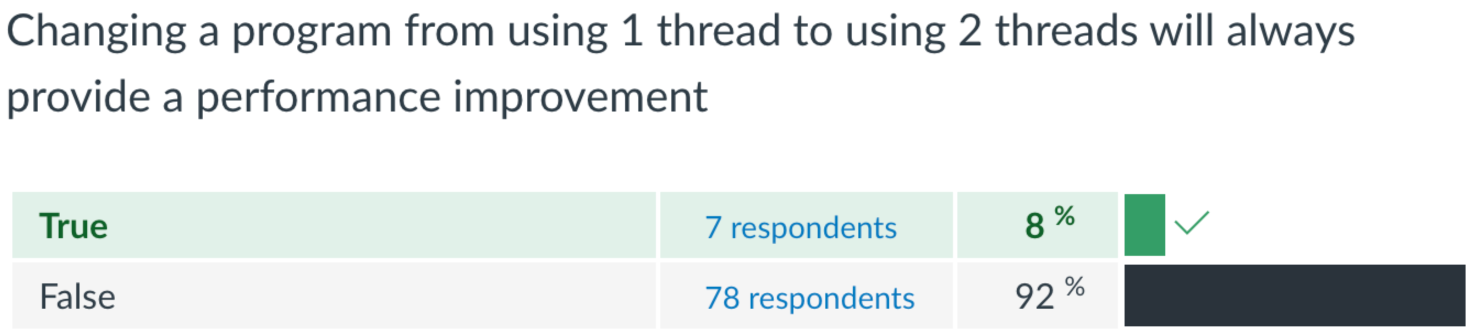
Note about quizzes and attendance:

- You have 3 “free” quizzes that you can miss
- You can miss them for any reason
  - Sick
  - need a break
  - family emergency
- I will not be excusing quizzes
  - If there is an emergency, then you should use one of your ”free” quizzes
  - If you run out of “free” quizzes then it is going to start impacting your grade

Quiz Review

- Everyone has at least 2 cores in their machine ☺
- 20% of people have an M1 mac:
  - This is a really impressive feat by Apple!

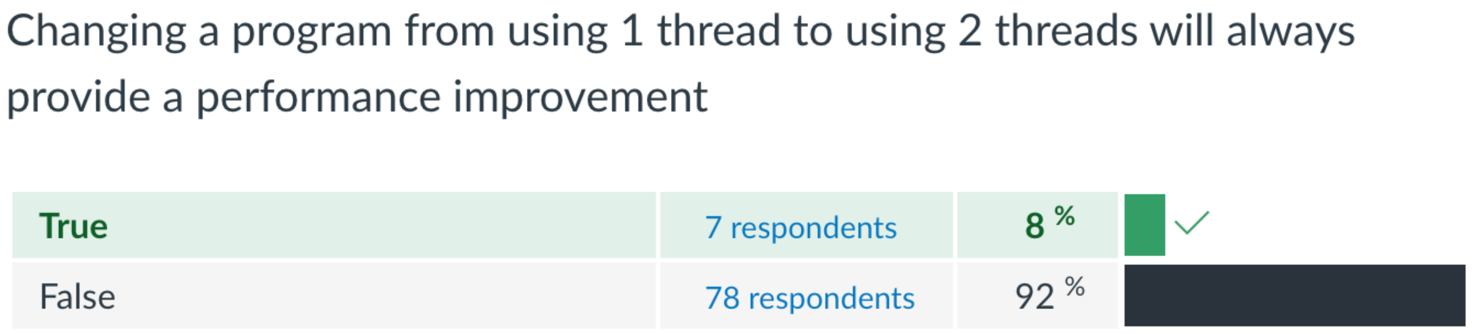
Some answers



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

True	7 respondents	**8 %**	✓
False	78 respondents	92 %	

Some answers

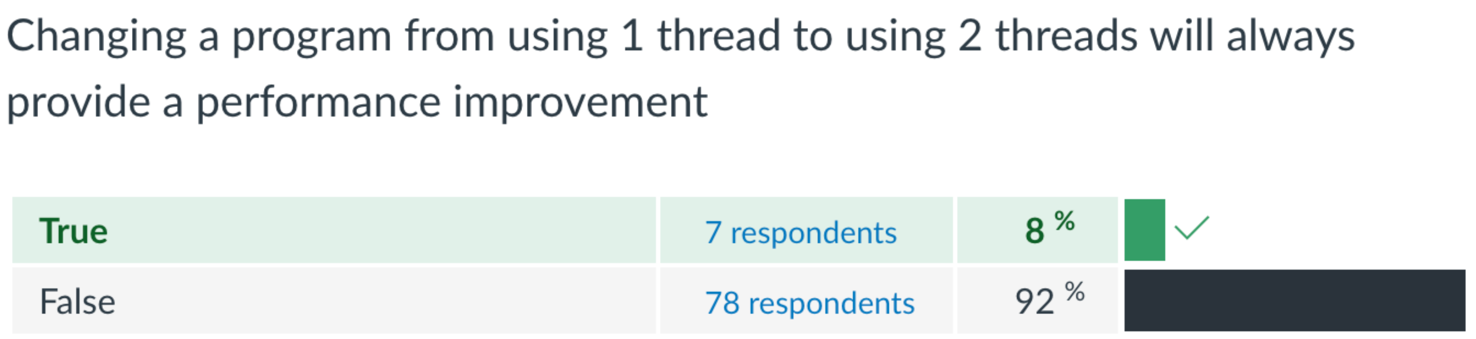


Changing a program from using 1 thread to using 2 threads will always provide a performance improvement

True	7 respondents	8 %	🟩 ✓
False	78 respondents	92 %	■

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

Some answers



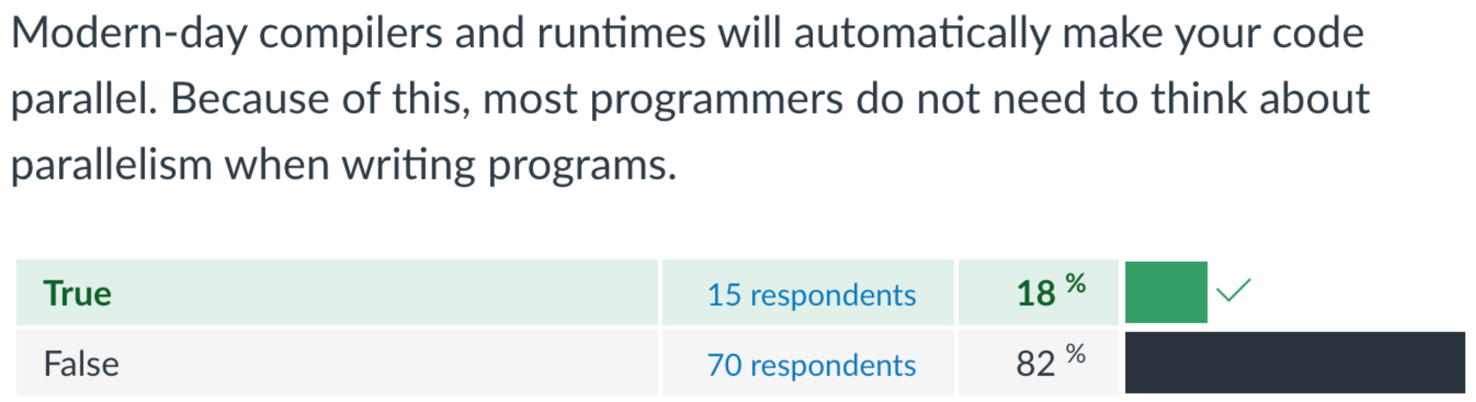
Changing a program from using 1 thread to using 2 threads will always provide a performance improvement

True	7 respondents	**8 %**	✓
False	78 respondents	92 %	

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

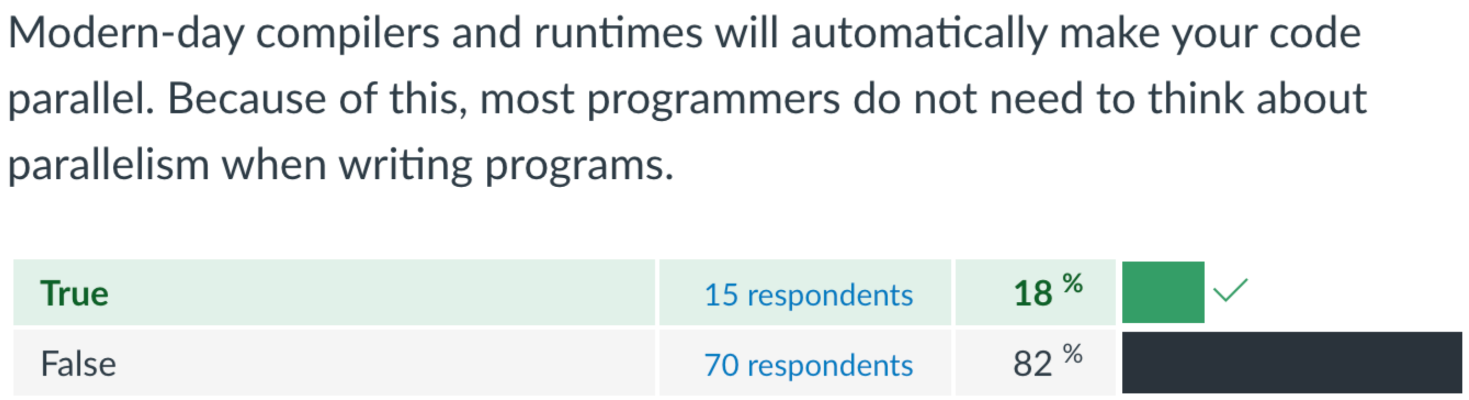
Some answers



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	15 respondents	**18 %**	🟩 ✓
False	70 respondents	82 %	■

Some answers

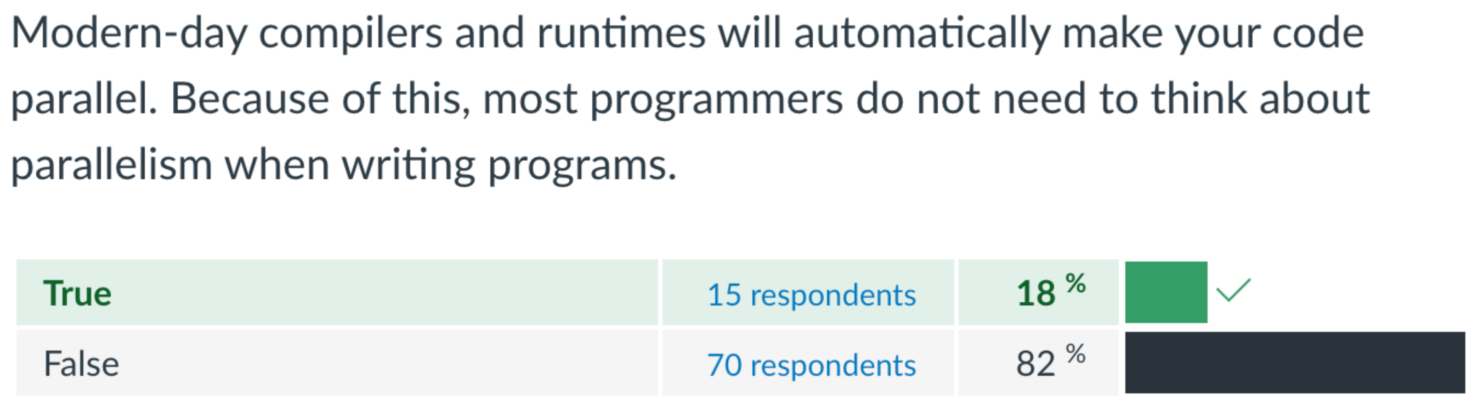


Modern-day compilers and runtimes will automatically make your code parallel. Because of this, most programmers do not need to think about parallelism when writing programs.

True	15 respondents	**18 %**	🟩 ✓
False	70 respondents	82 %	■

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

Some answers



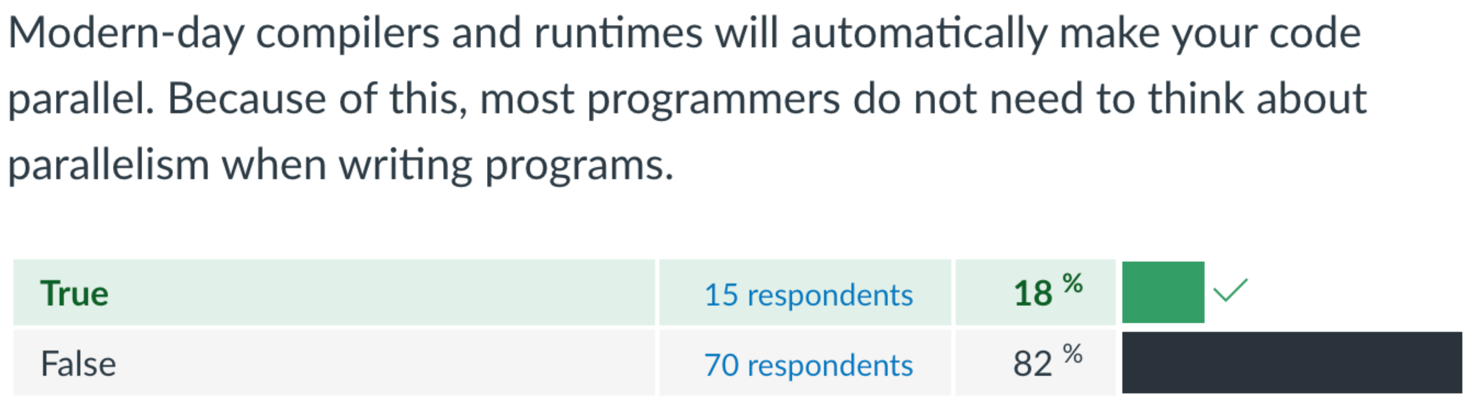
Modern-day compilers and runtimes will automatically make your code parallel. Because of this, most programmers do not need to think about parallelism when writing programs.

True	15 respondents	18 %	✓
False	70 respondents	82 %	

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	15 respondents	**18 %**	🟩 ✓
False	70 respondents	82 %	■

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)
```

gets compiled into:

```
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;
```

- A computer core executes roughly 1 instruction per cycle

Memory accesses

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

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

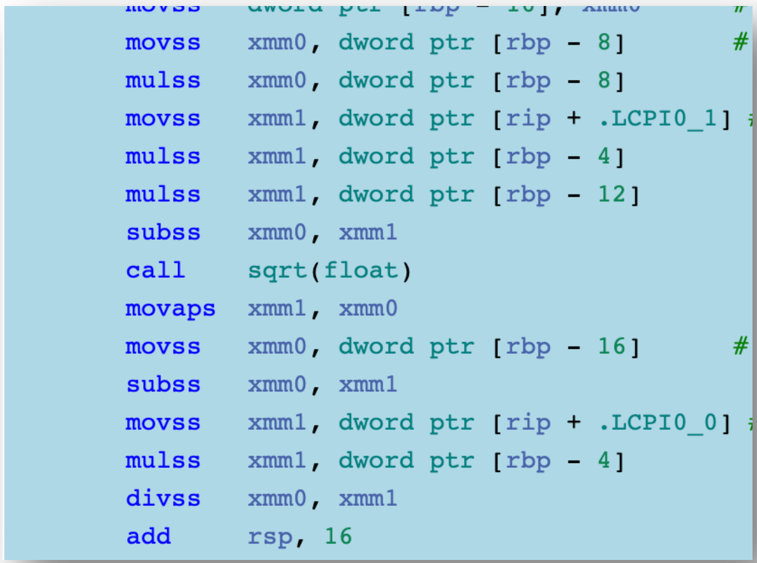
Unless explicitly expressed in the programming language, loads and stores are split into multiple instructions!

Review

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

Core

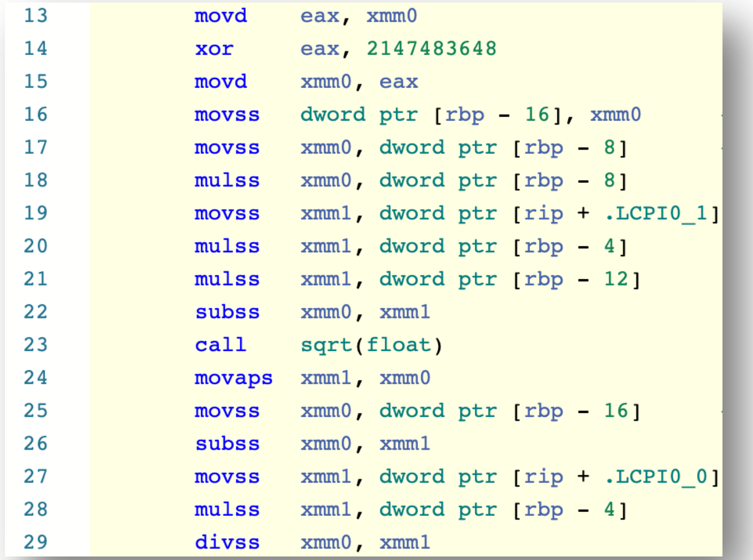
Compiled function #1



```
movss   dword ptr [rbp - 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

Compiled function #0

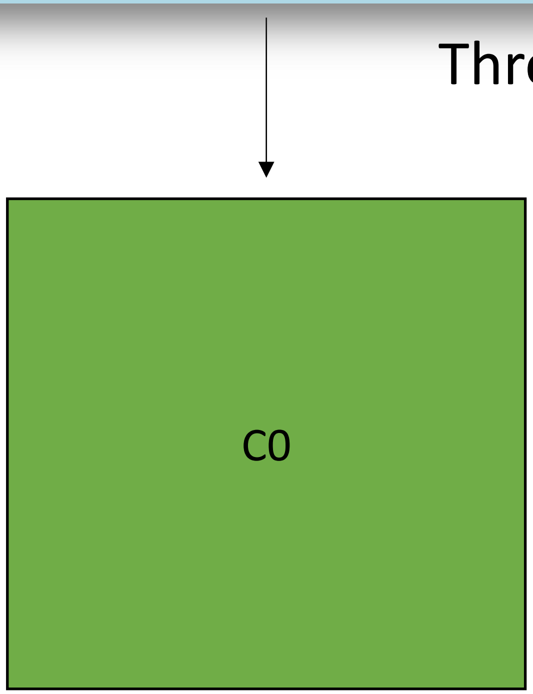


```
movd    eax, xmm0
xor     eax, 2147483648
movd    xmm0, eax
movss   dword ptr [rbp - 16], xmm0
movss   xmm0, dword ptr [rbp - 8]
mulss   xmm0, dword ptr [rbp - 8]
movss   xmm1, dword ptr [rip + .LCPI0_1]
mulss   xmm1, dword ptr [rbp - 4]
mulss   xmm1, dword ptr [rbp - 12]
subss   xmm0, xmm1
call    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
```

Thread 0

Preemption can occur:

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

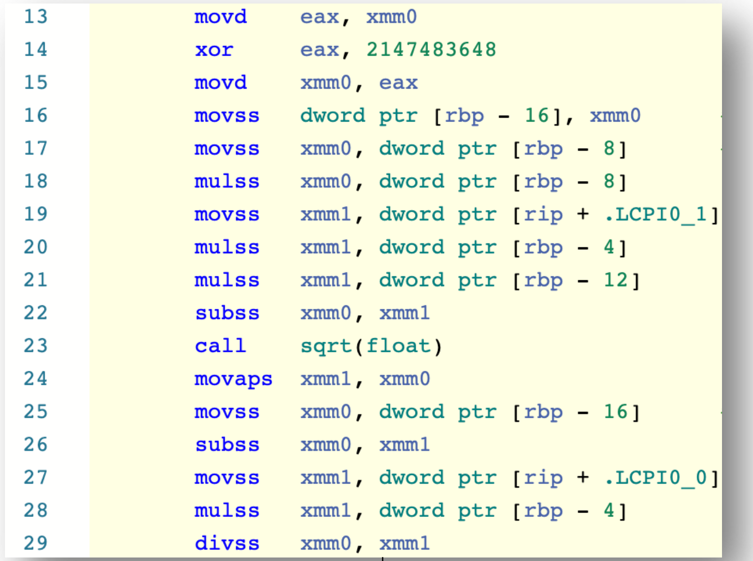


Core

And place another thread to execute

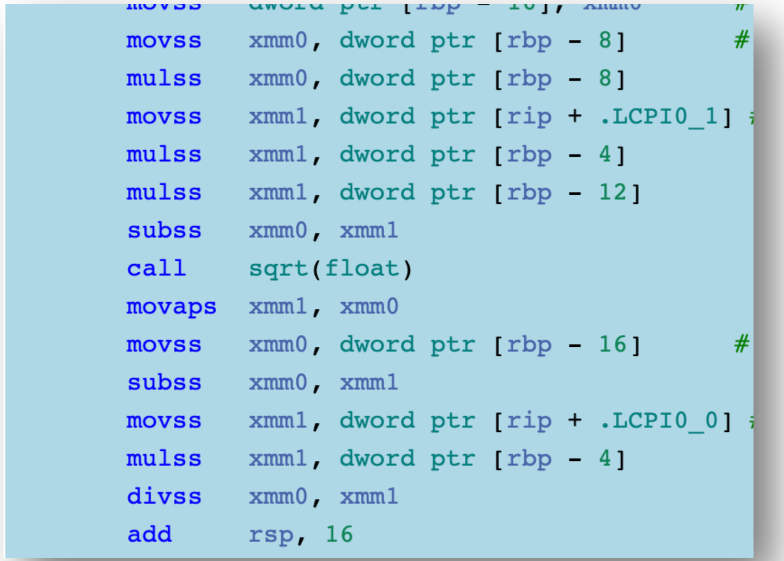
Multicores

Compiled function #0

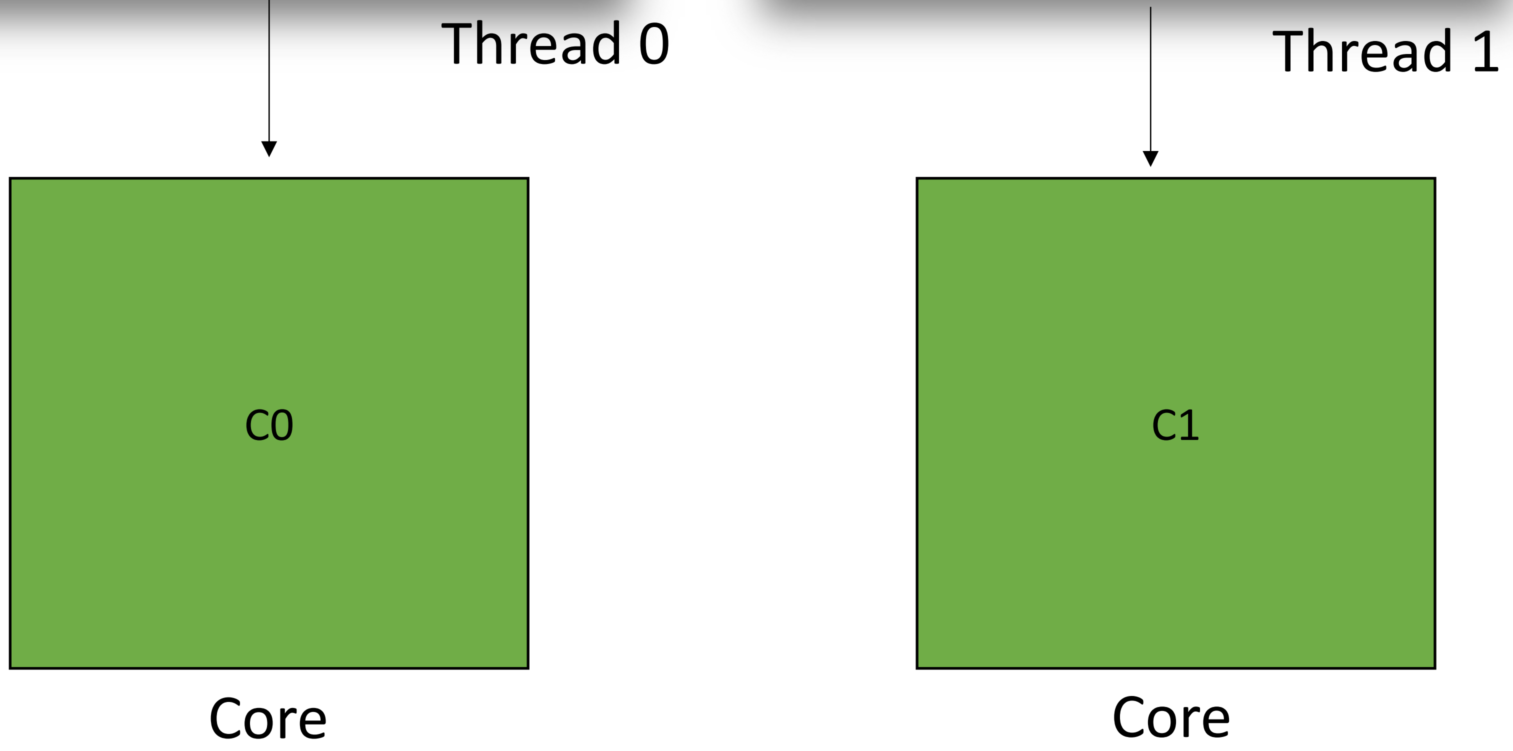


```
        movd    eax, xmm0
        xor     eax, 2147483648
        movd    xmm0, eax
        movss   dword ptr [rbp - 16], xmm0
        movss   xmm0, dword ptr [rbp - 8]
        mulss   xmm0, dword ptr [rbp - 8]
        movss   xmm1, dword ptr [rip + .LCPI0_1]
        mulss   xmm1, dword ptr [rbp - 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
```

Compiled function #1



```
        movss   dword ptr [rbp - 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
```



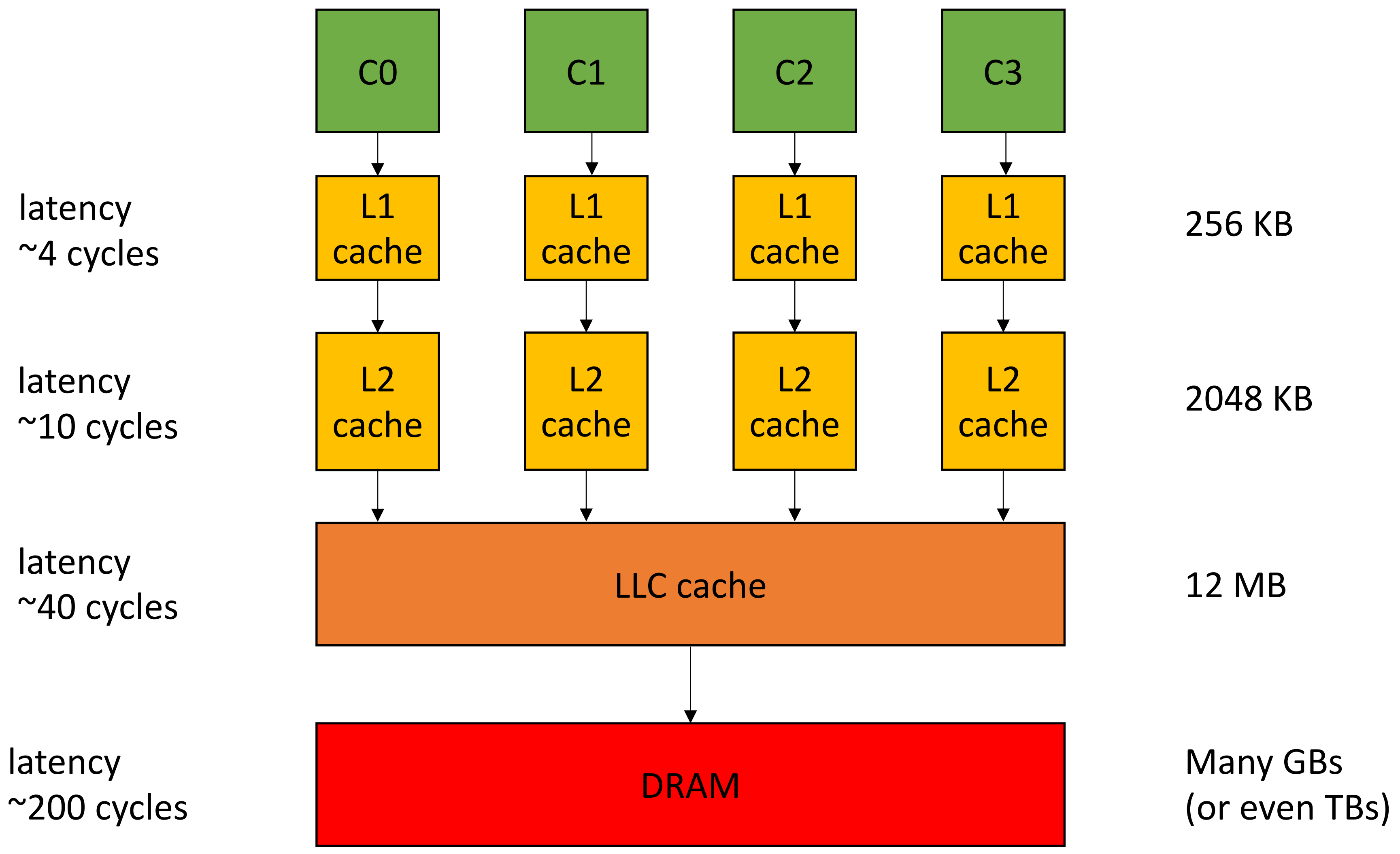
Threads can execute simultaneously.

This is also concurrency. But the simultaneously called parallelism.

Review

- Caches make memory accesses faster

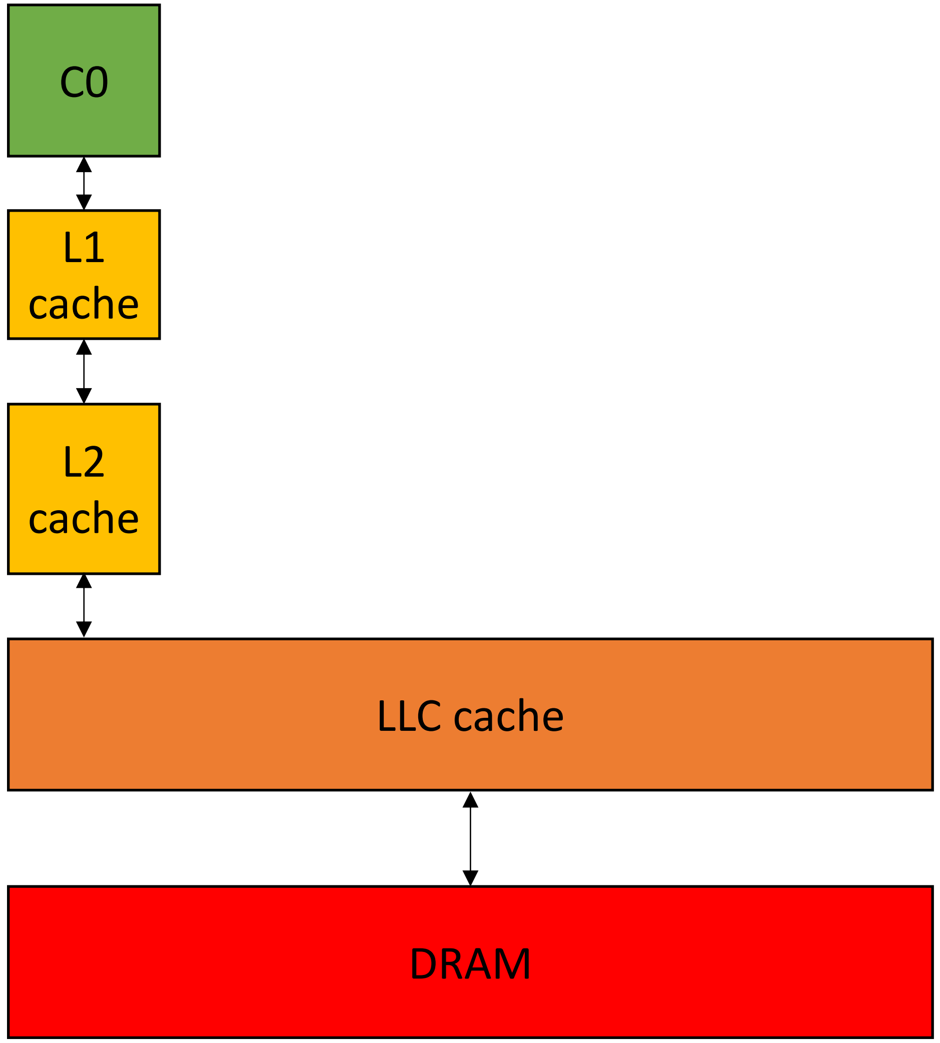
Caches


C0
C1
C2
C3
latency
~4 cycles
L1
cache
256 KB
latency
~10 cycles
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
```



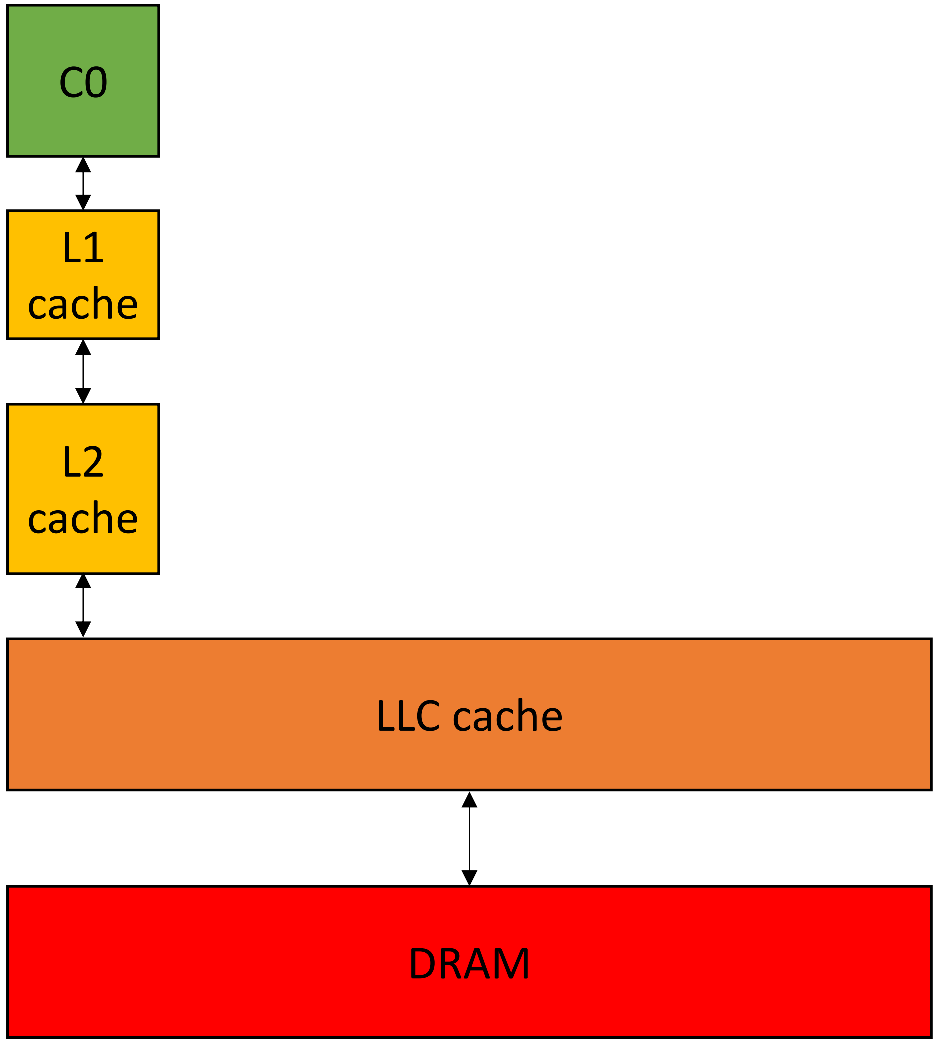
Caches

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

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

4 cycles

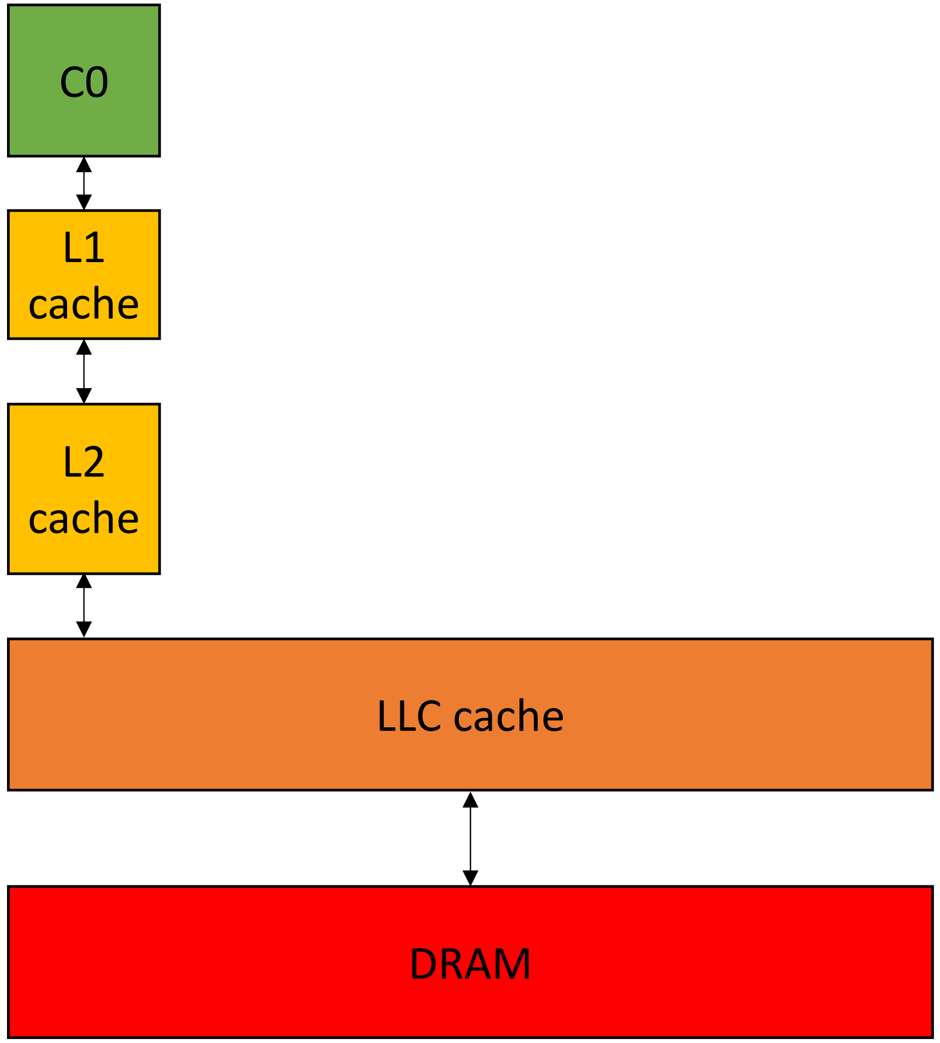
Assuming the value is in the cache!



Caches

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

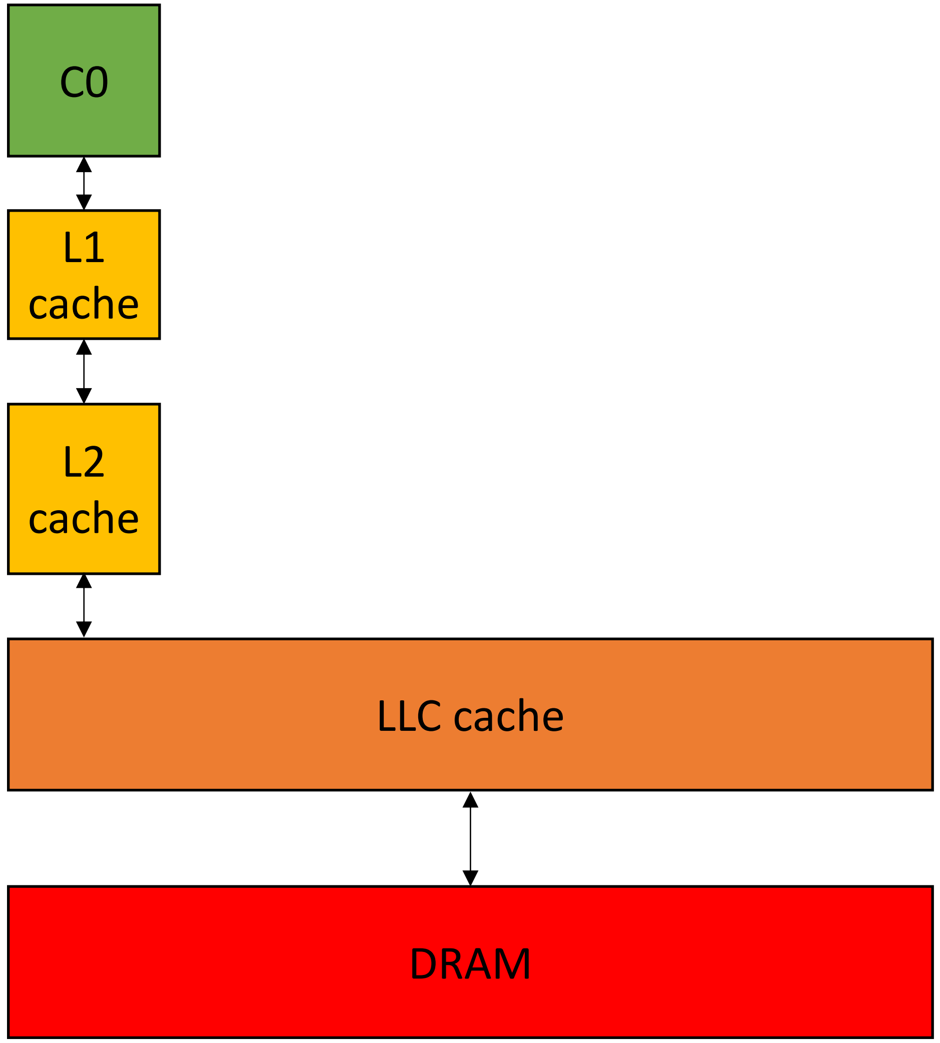
```
%5 = load i32, i32* %4          4 cycles
%6 = add nsw i32 %5, 1          1 cycles
store i32 %6, i32* %4
```



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
```

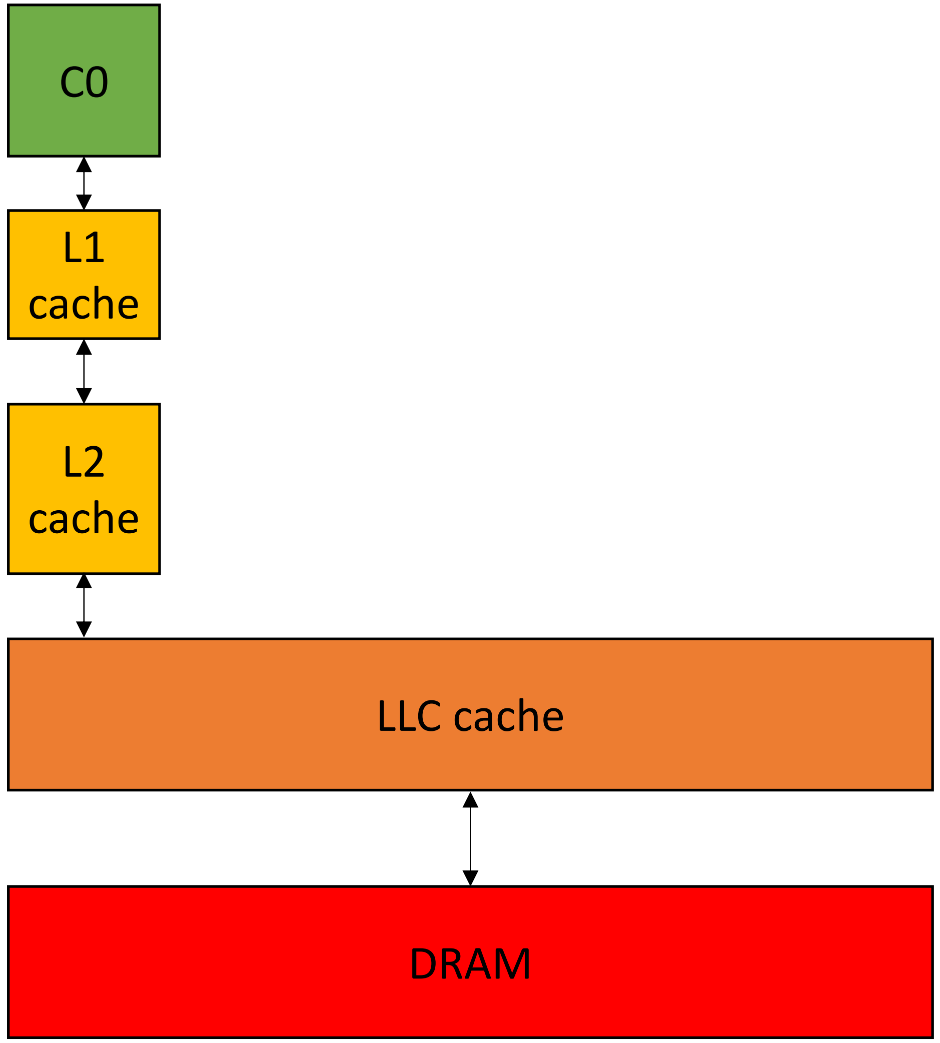


Caches

```
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]++;
}
```

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_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

Cache lines

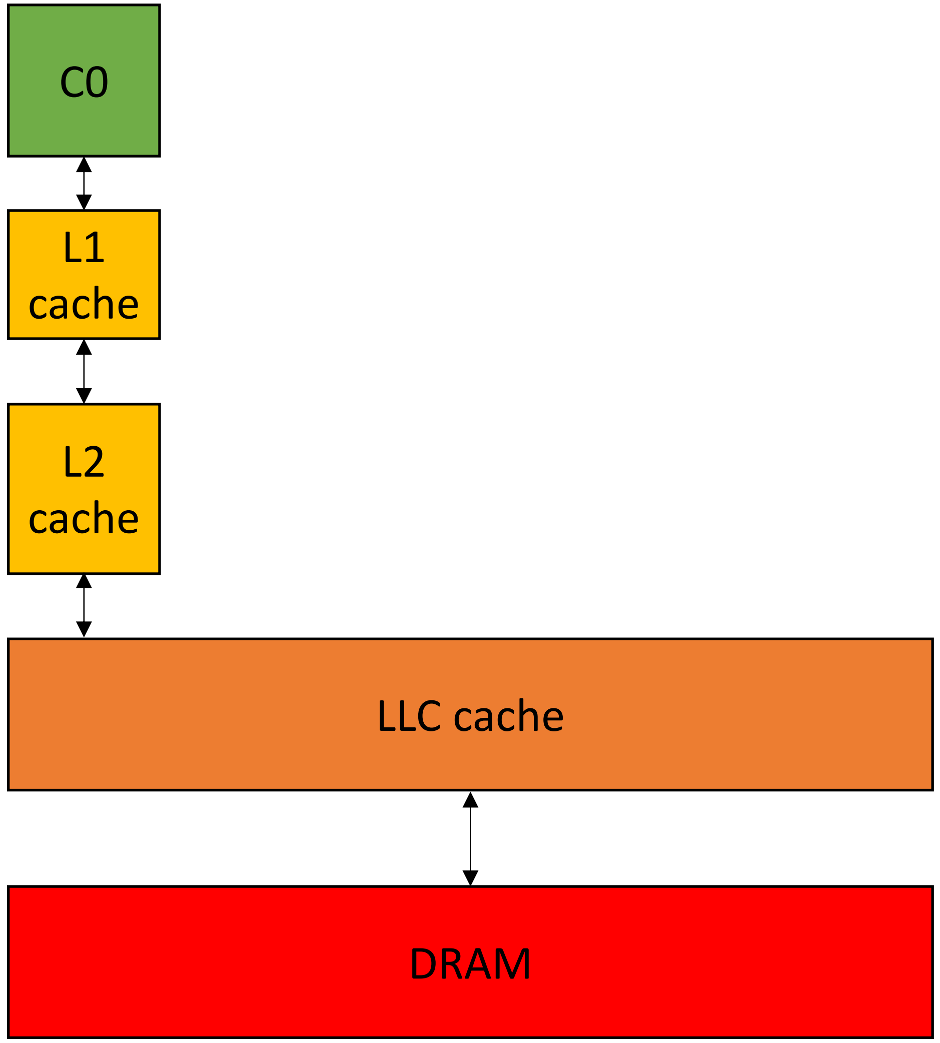
- Cache line size for x86: 64 bytes:
  - 64 chars
  - 32 shorts
  - 16 float or int
  - 8 double or long

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
```

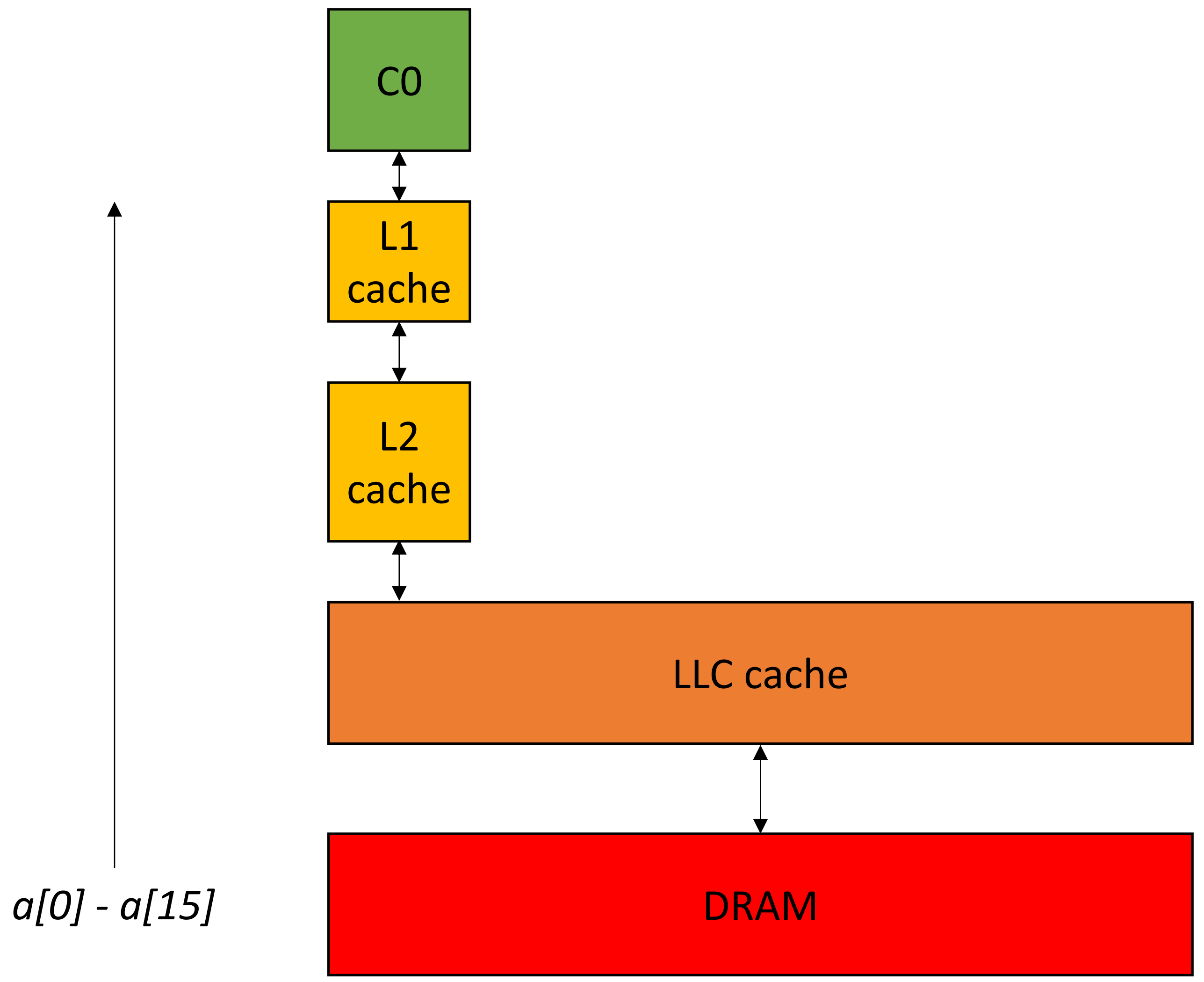


Assume a[0] is not in the cache

Caches

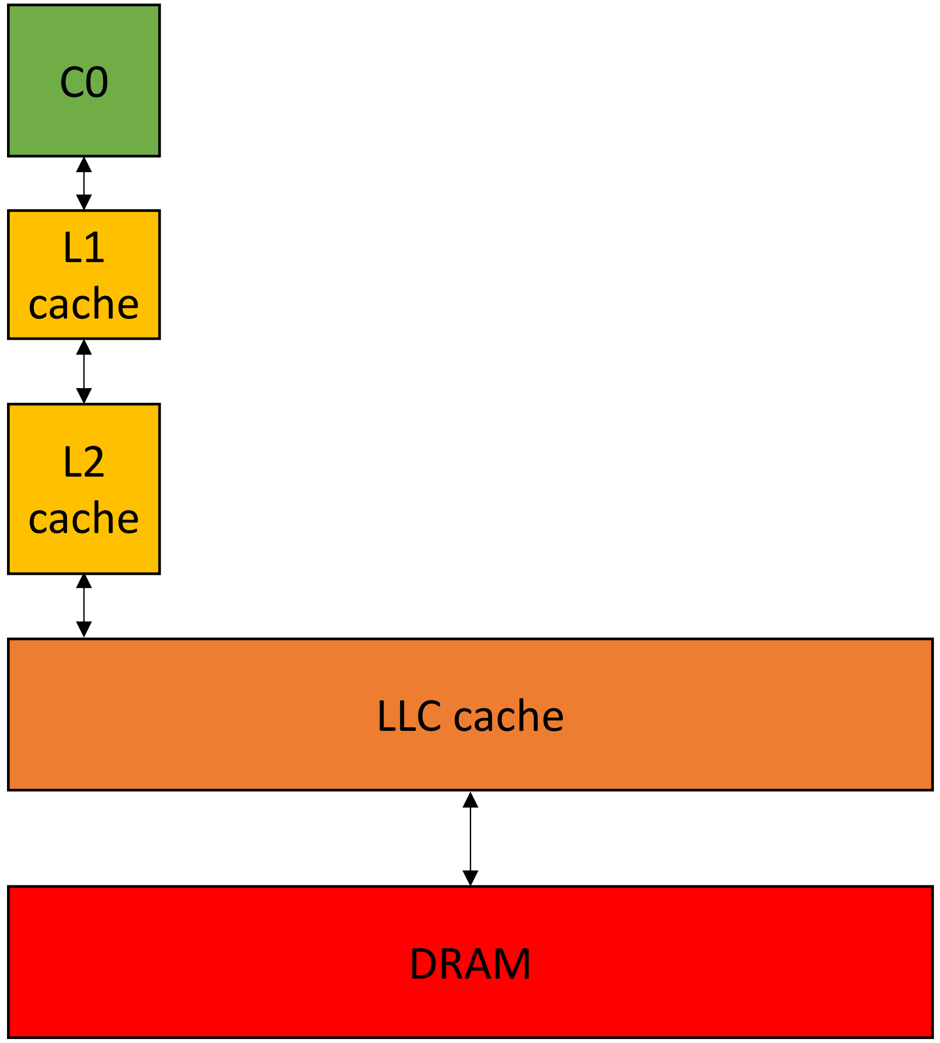
```
int increment(int *a) {
    a[0]++;
}
```

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



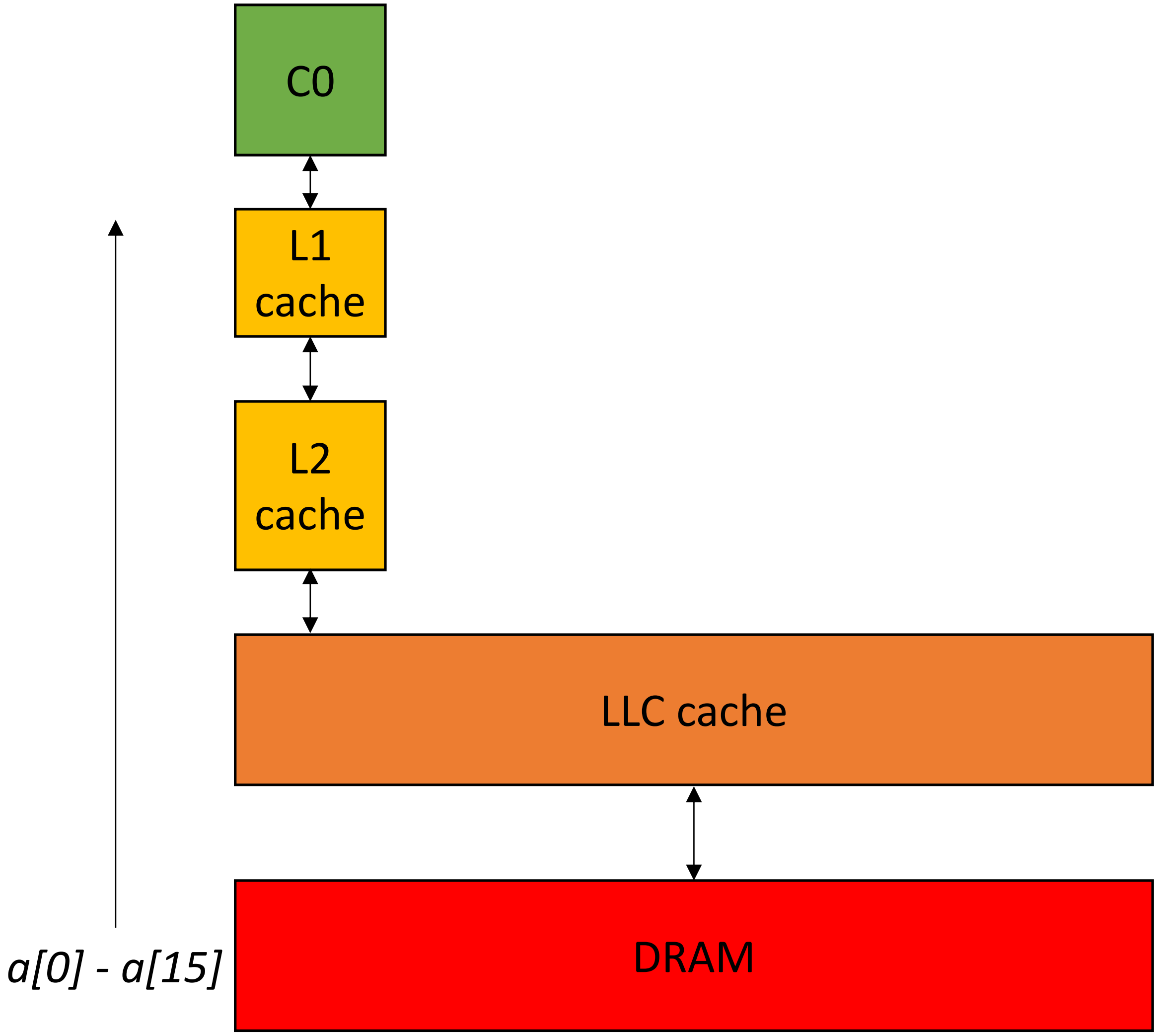
Caches

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



Caches

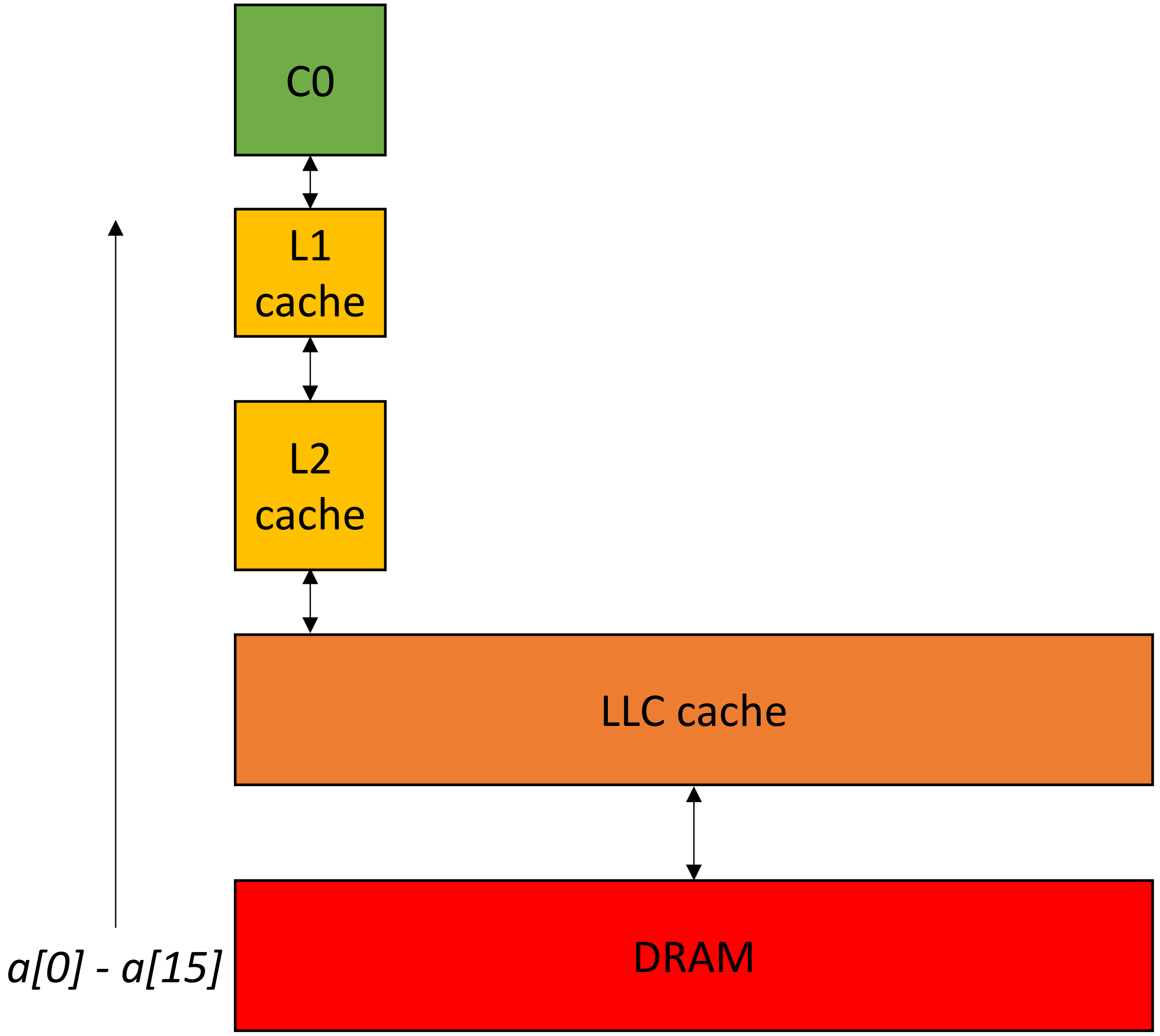
```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```



Caches

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

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

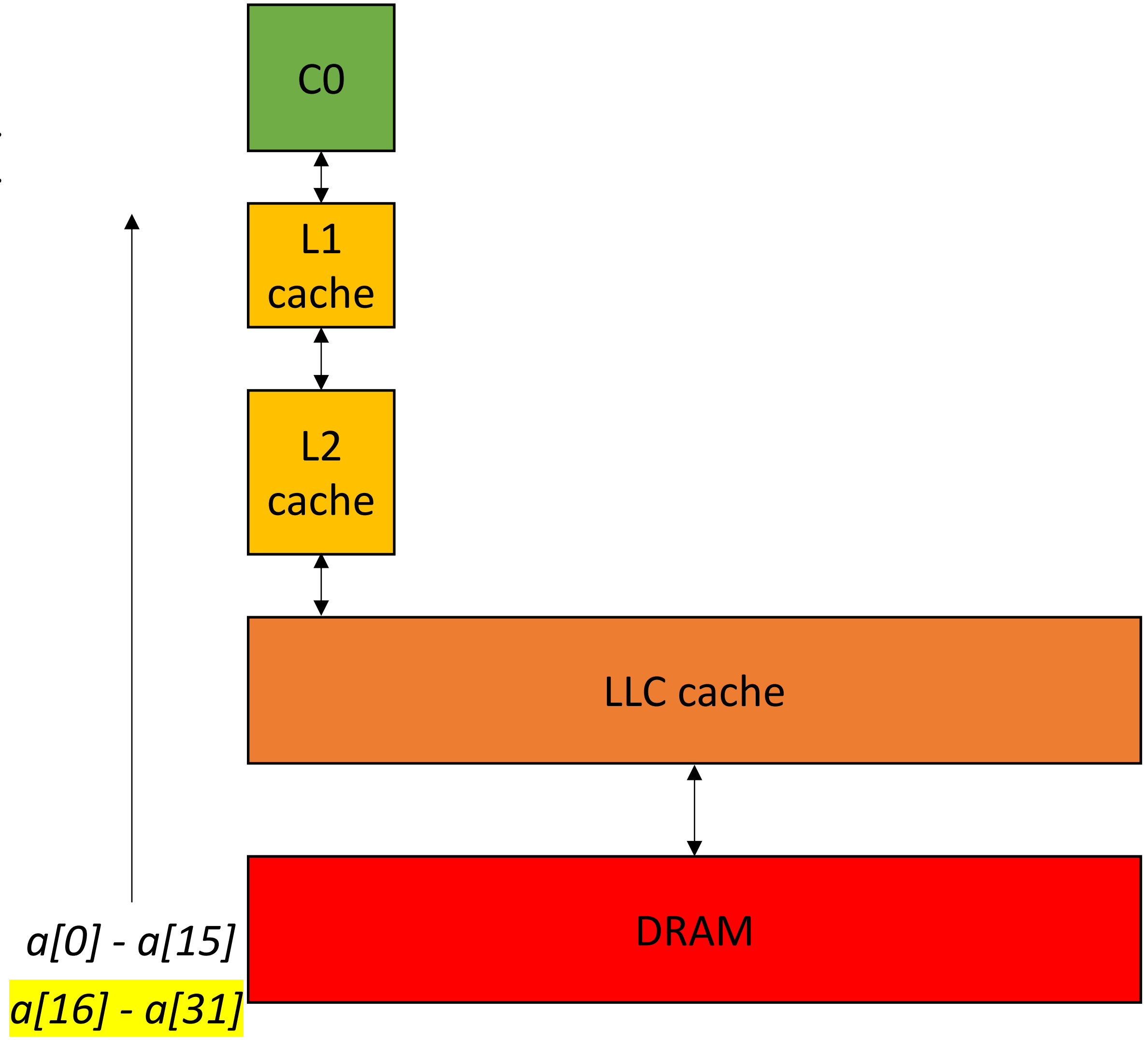


Assume a[0] is not in the cache

Caches

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

Miss

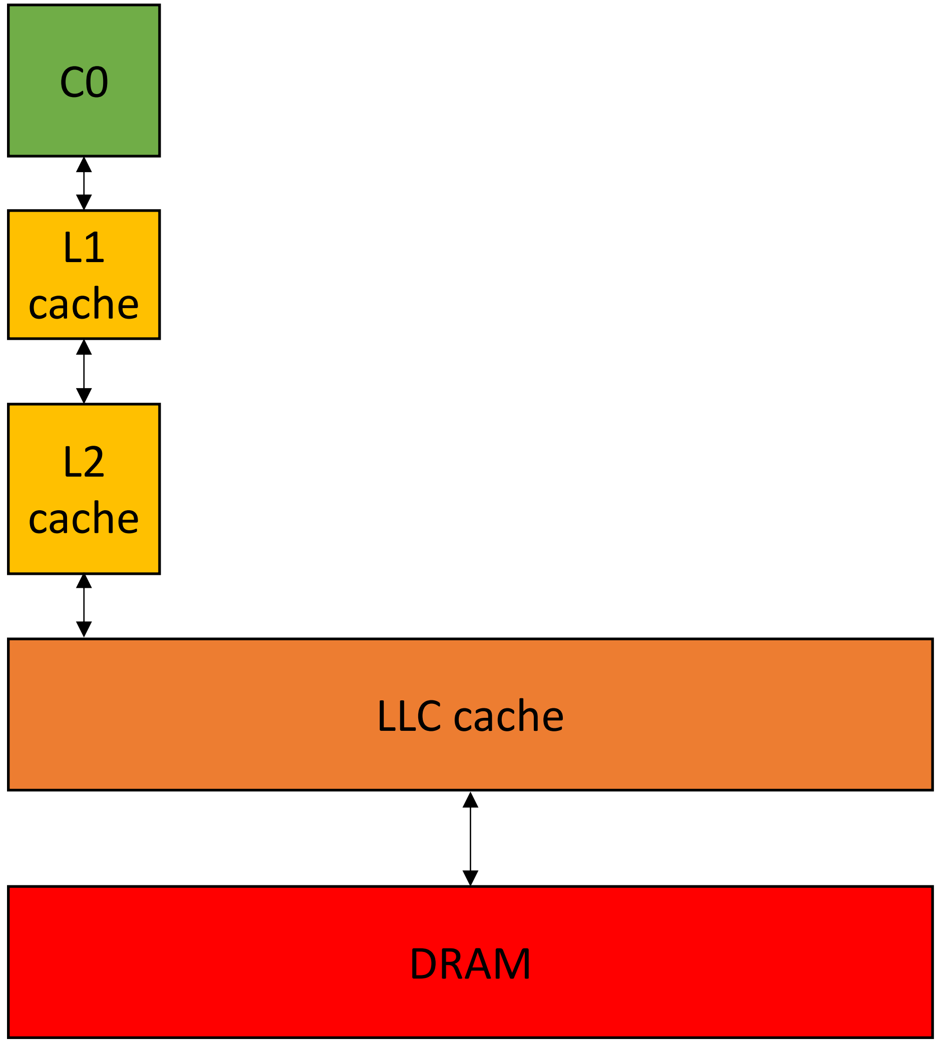


Assume a[0] is not in the cache

Cache alignment

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

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

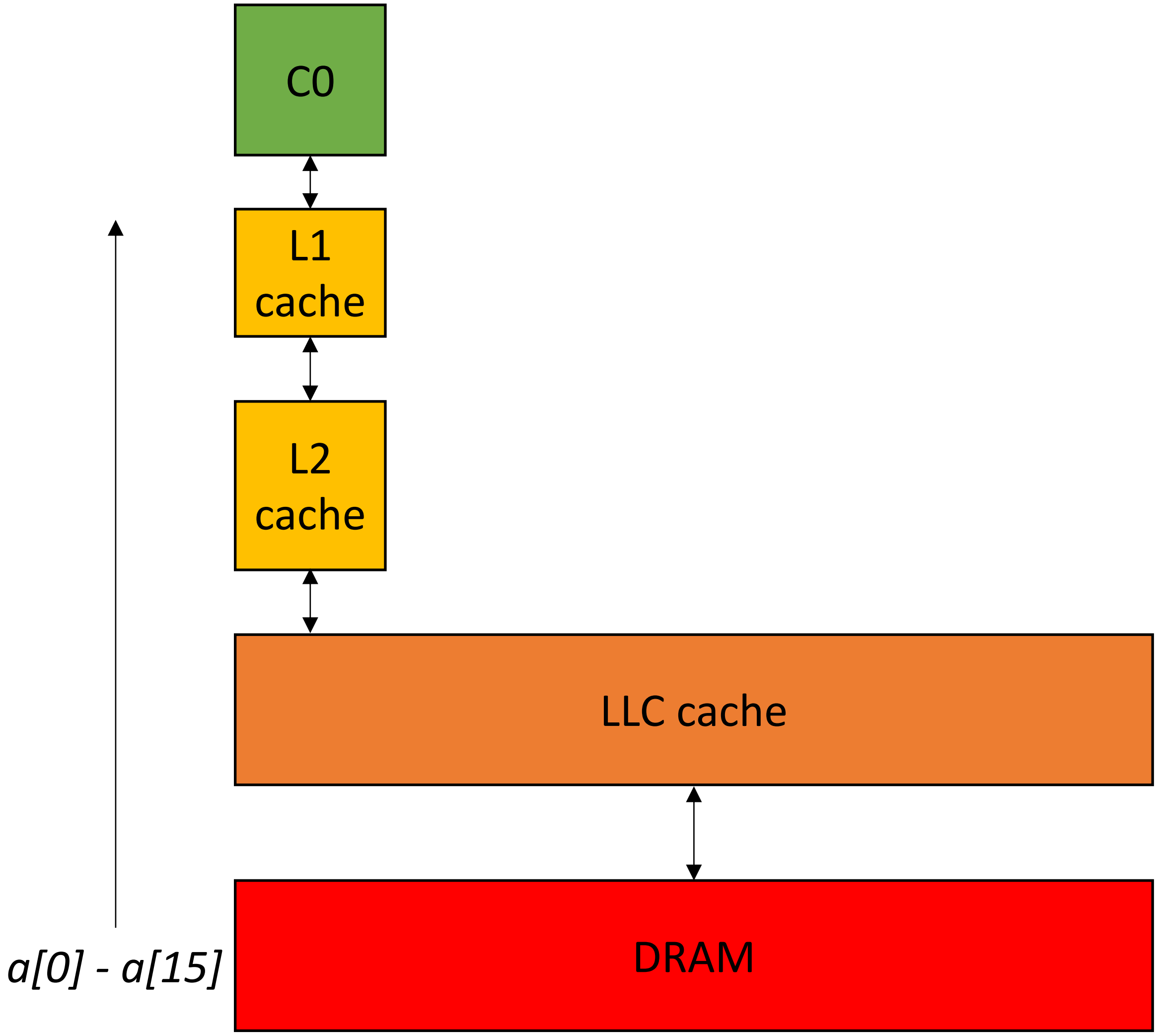


Assume a[0] is not in the cache

Cache alignment

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

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



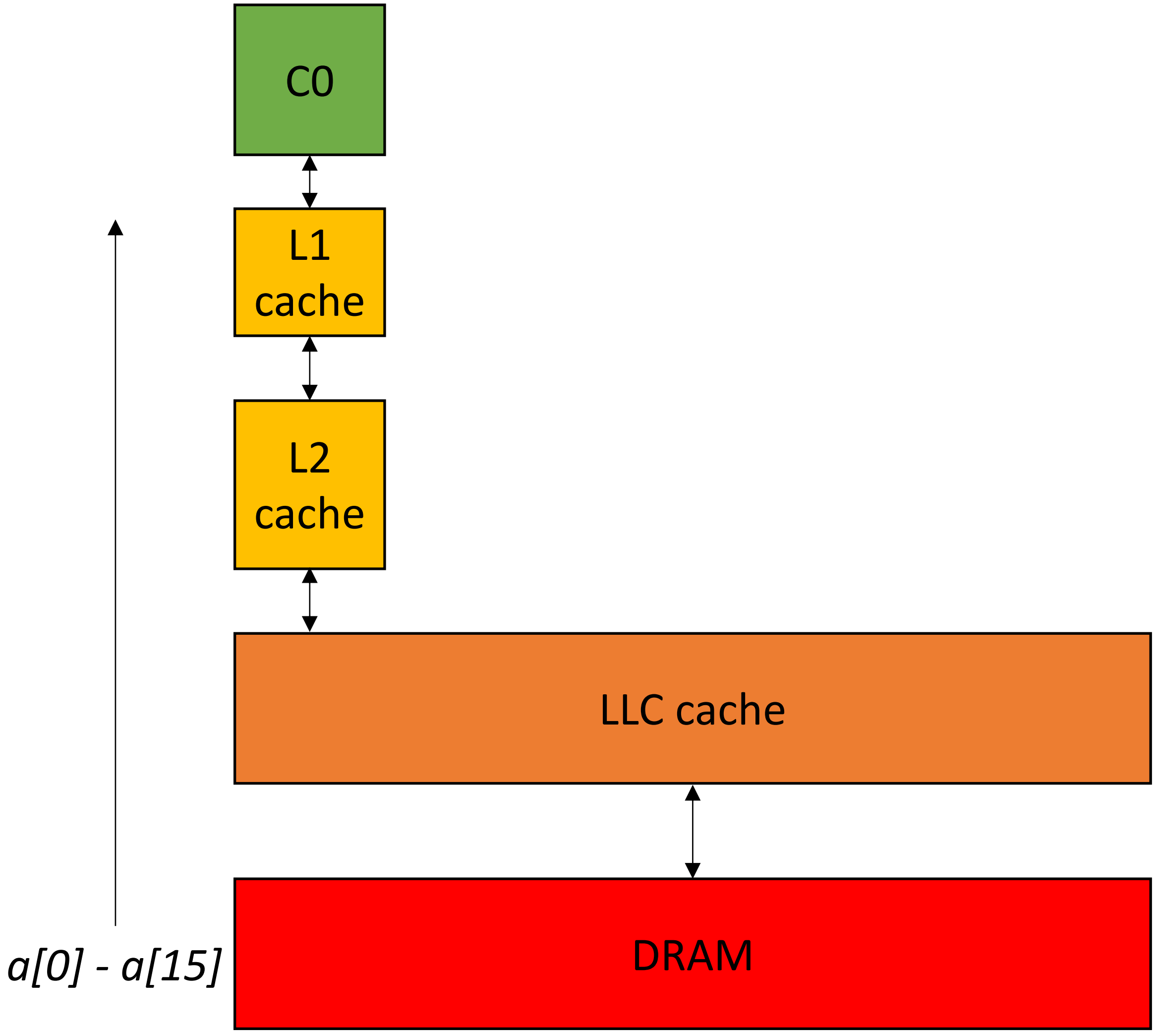
Assume a[0] is not in the cache

Cache alignment

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

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

This loads a[8]



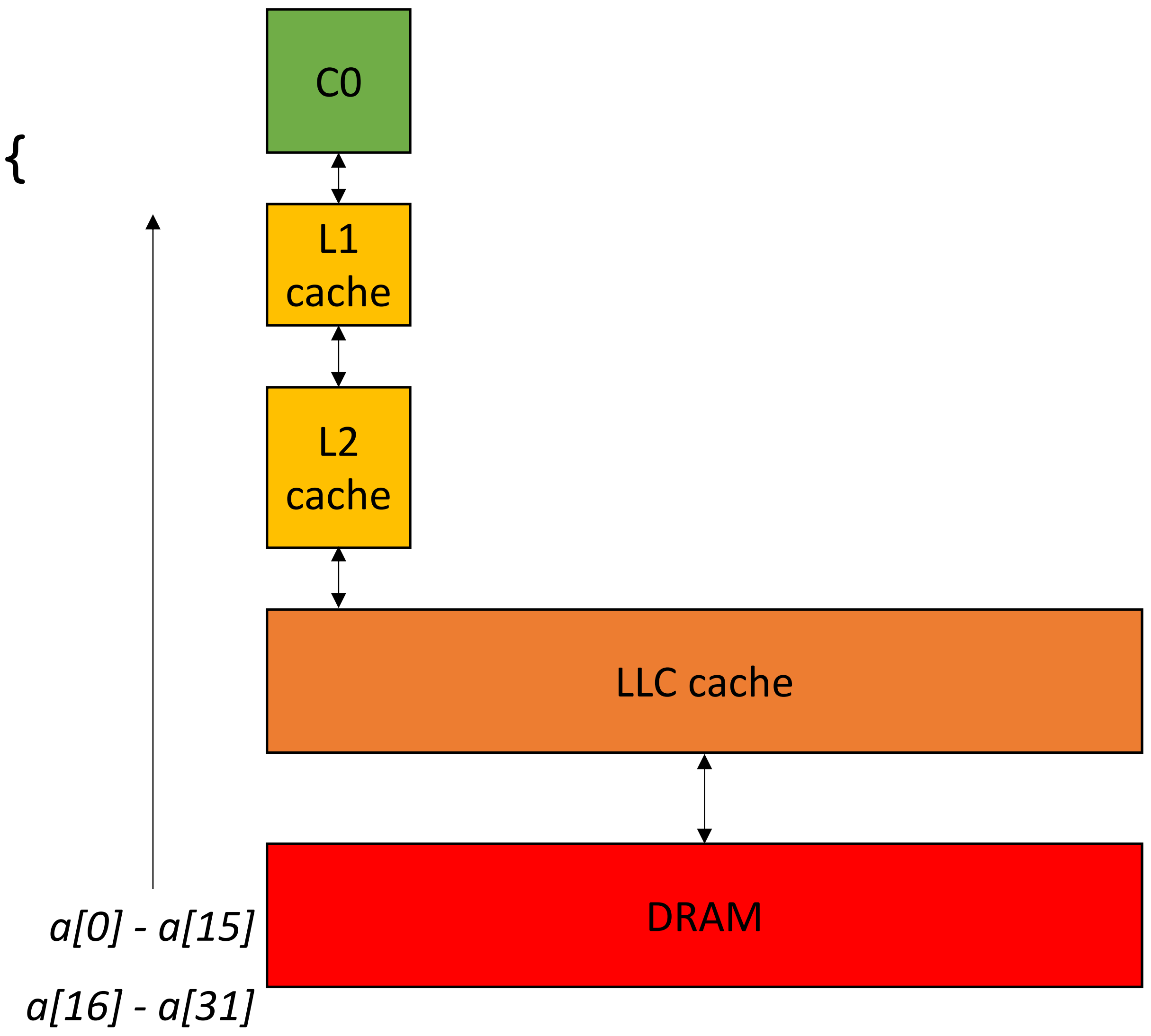
Assume a[0] is not in the cache

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 organization

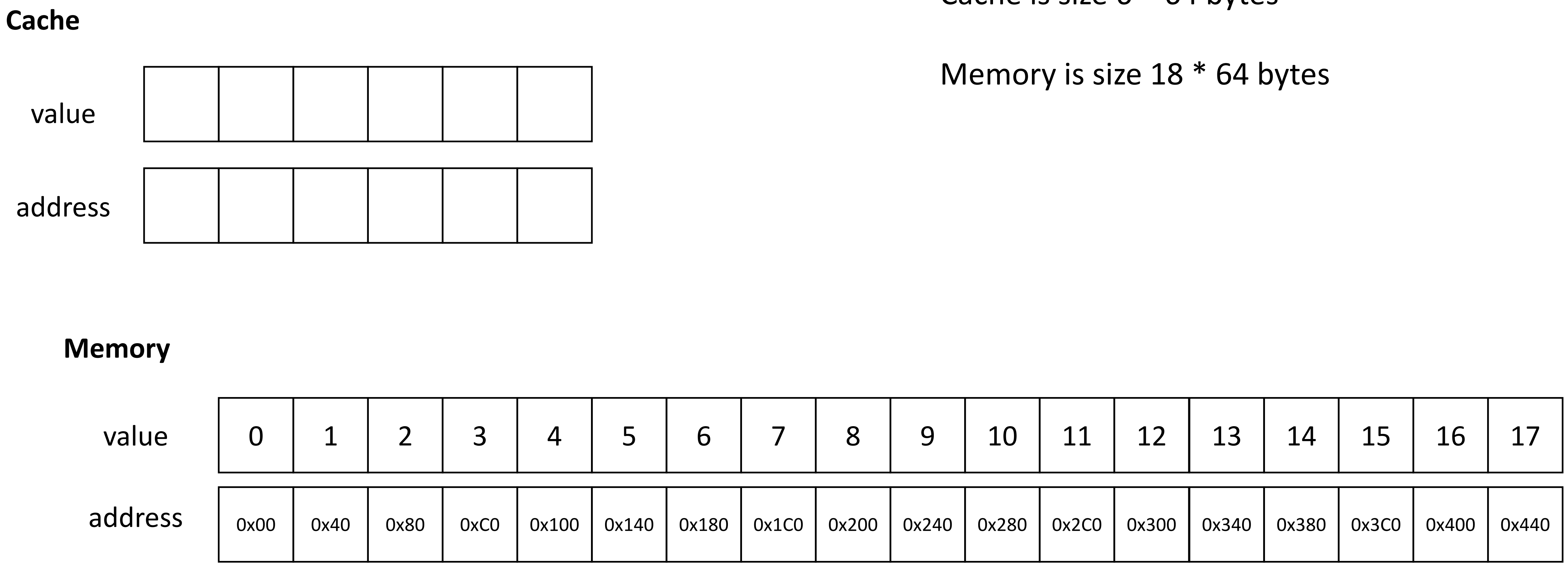
Cache organization

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 organization

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

cache block location = (address/64) % (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 = (address/64) % (cache size)

Example: Read address 0x00

Cache

value						
address						

Memory

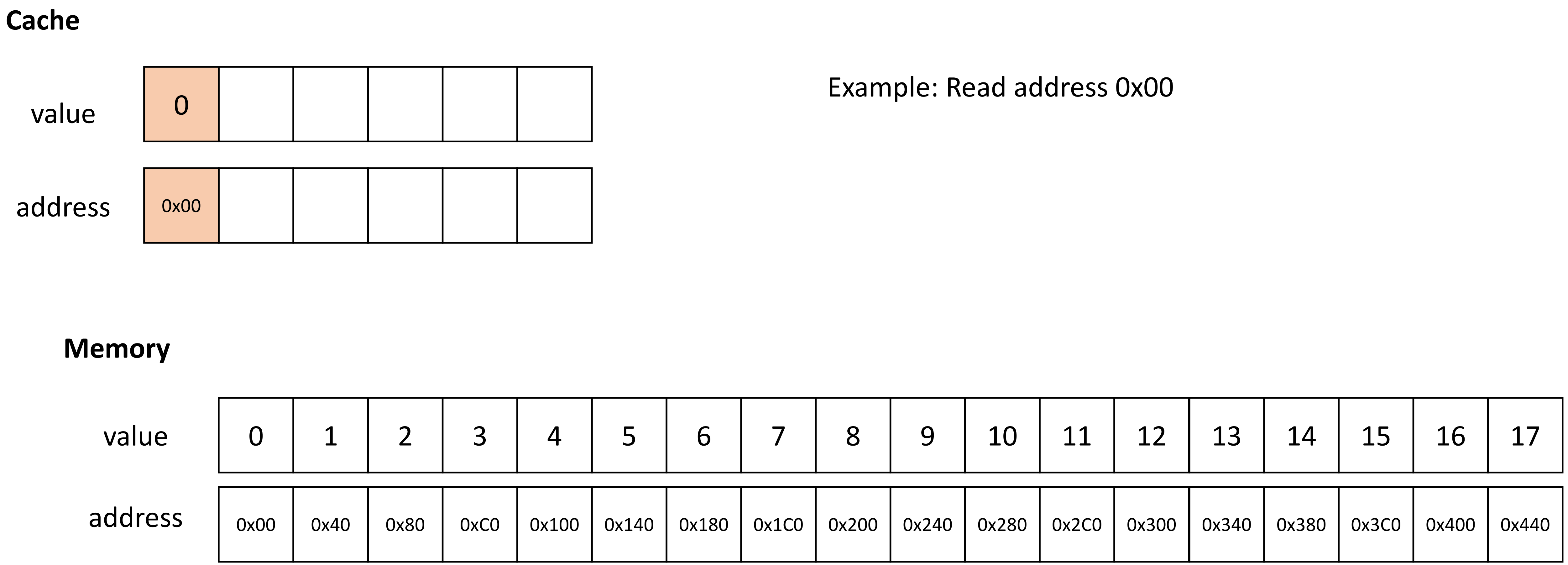
value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

Cache organization

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

cache block location = (address/64) % (cache size)

Example: Read address 0x00



Cache organization

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

cache block location = (address/64) % (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 = (address/64) % (cache size)

Example: Read address 0x80

Cache

value	0	7				
address	0x00	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 = (address/64) % (cache size)

Example: Read address 0x80

Cache

value	0	7				
address	0x00	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 = (address/64) % (cache size)

Example: Read address 0x80

Cache

value	0	7	2			
address	0x00	0x1C0	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 = (address/64) % (cache size)

Example: Read address 0x1C0

Cache

value	0	7	2			
address	0x00	0x1C0	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 = (address/64) % (cache size)

Example: Read address 0x1C0

Cache

value	0	7	2			
address	0x00	0x1C0	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 = (address/64) % (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 = (address/64) % (cache size)

Example: Read address 0x180

Cache

value	0	7	2			
address	0x00	0x1C0	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 = (address/64) % (cache size)

Example: Read address 0x180

Cache

value	0	7	2			
address	0x00	0x1C0	0x80			

Memory

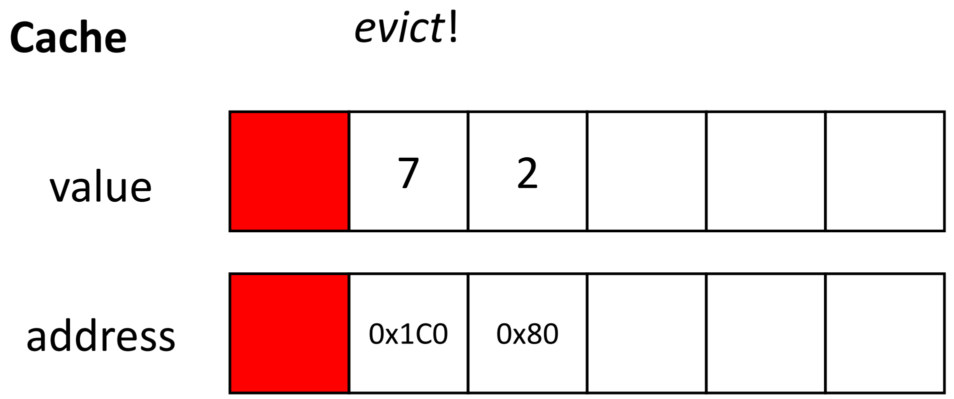
value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

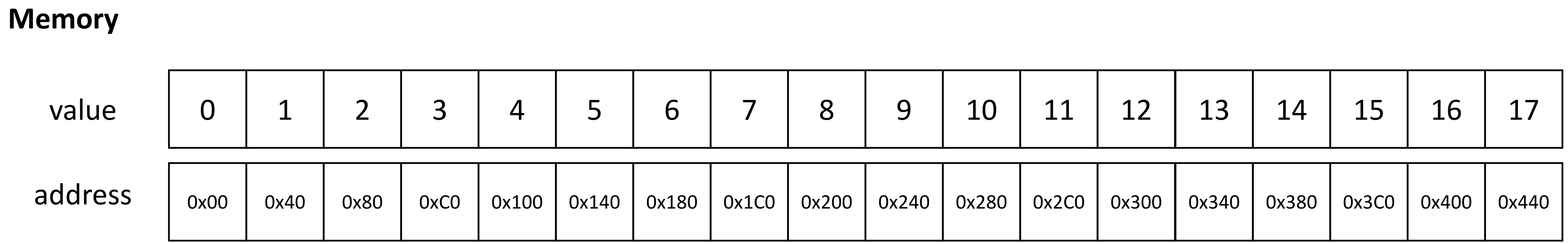
Cache organization

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

cache block location = (address/64) % (cache size)

Example: Read address 0x180





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 = (address/64) % (cache size)

Example: Read address 0x180

Cache

value		7	2			
address		0x1C0	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 = (address/64) % (cache size)

Example: Read address 0x180

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

Cache

value	6	7	2			
address	0x180	0x1C0	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

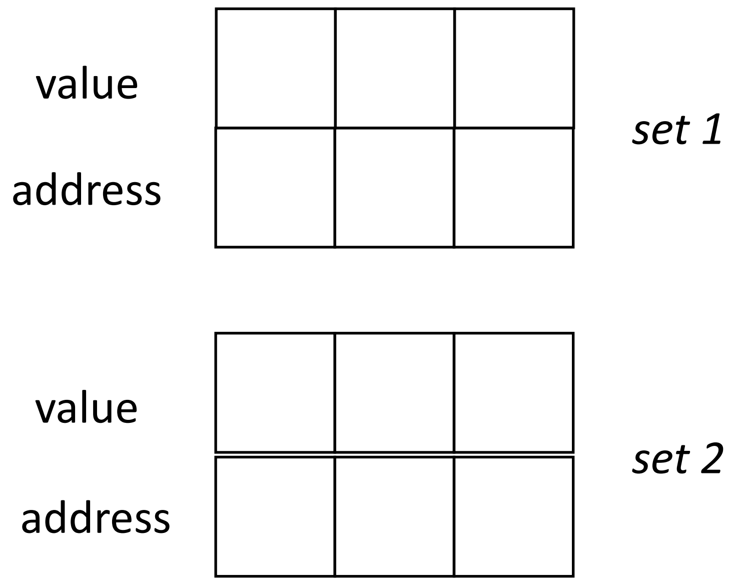
Cache organization

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

cache block location (address/64) % (cache size / N)

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

Cache



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

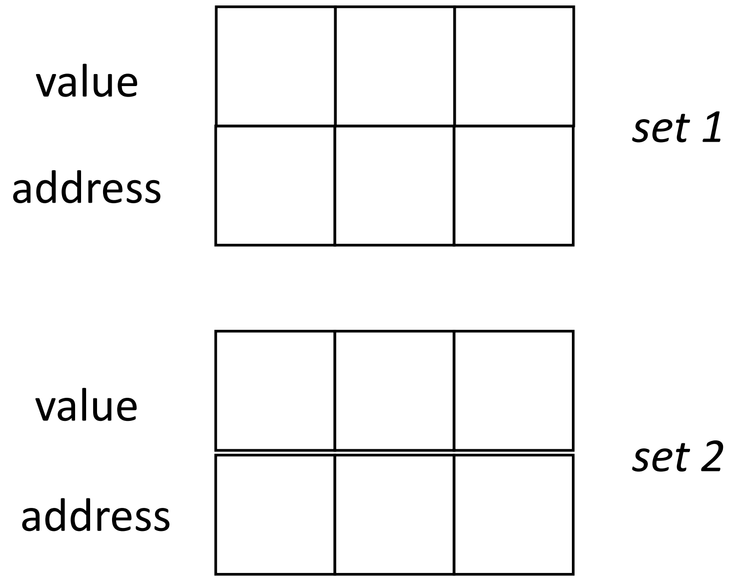
Cache organization

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

cache block location (address/64) % (cache size / N)

Cache will make an "intelligent" decision on which value to evict

Cache



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 (address/64) % (cache size / N)

Cache will make an "intelligent" decision on which value to evict

Cache

value	0	7	2	*set 1*
address	0x00	0x1C0	0x80	

value				*set 2*
address				

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 (address/64) % (cache size / N)

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

Cache

set 1

value	0	7	2
address	0x00	0x1C0	0x80

set 2

value			
address			

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 (address/64) % (cache size / N)

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

Cache

value	0	7	2	*set 1*
address	0x00	0x1C0	0x80	

value				*set 2*
address				

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 (address/64) % (cache size / N)

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

Cache

value	0	7	2	*set 1*
address	0x00	0x1C0	0x80	

value	▨			*set 2*
address	▨			

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 (address/64) % (cache size / N)

Cache will make an "intelligent" decision on which value to evict

Cache

set 1

value	0	7	2
address	0x00	0x1C0	0x80

set 2

value	6		
address	0x180		

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 (address/64) % (cache size / N)

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

Cache

value	0	7	2	*set 1*
address	0x00	0x1C0	0x80	

value	6			*set 2*
address	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 (address/64) % (cache size / N)

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

Cache

value	0	7	2	*set 1*
address	0x00	0x1C0	0x80	

value	6			*set 2*
address	0x180			

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 (address/64) % (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

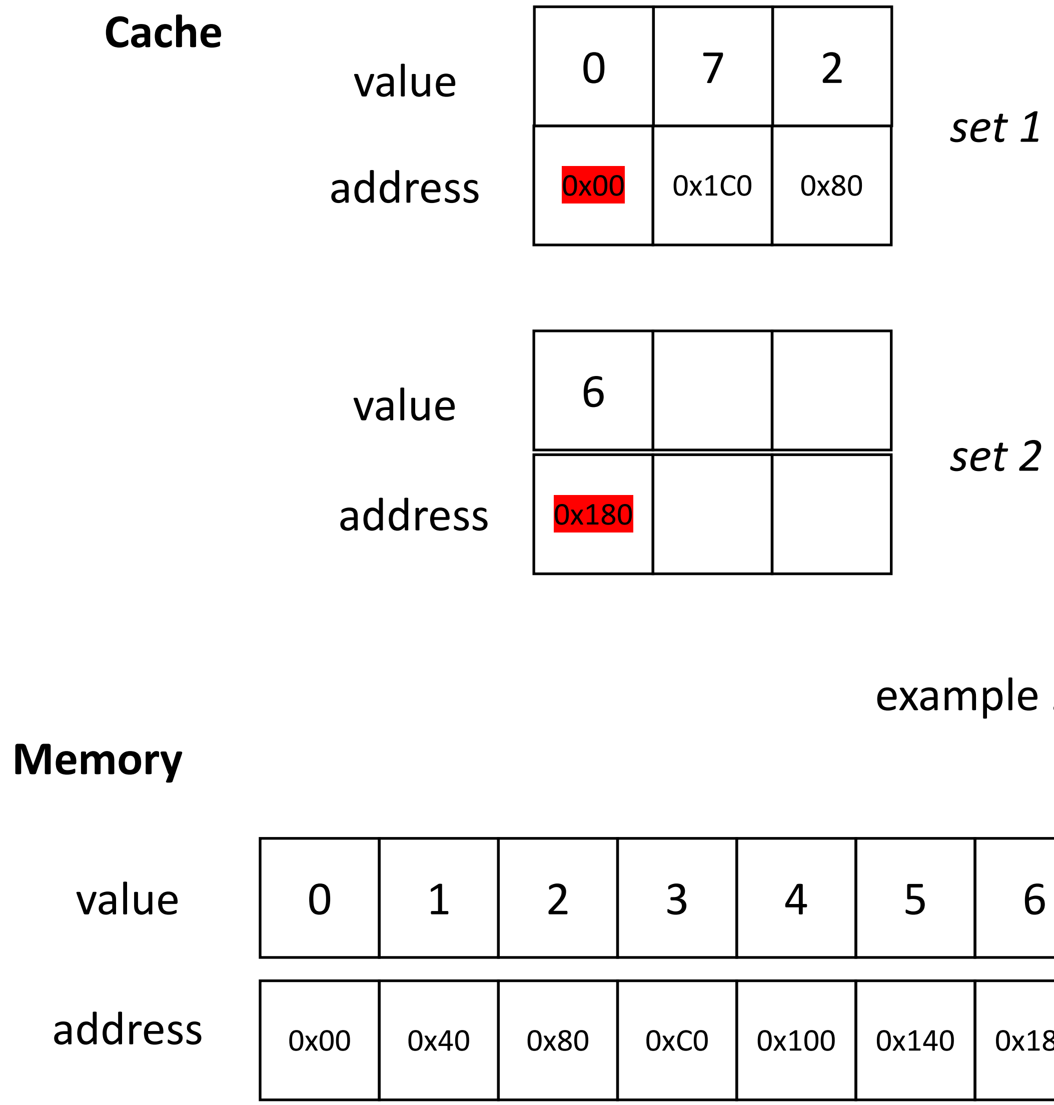
value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	0x00	0x40	0x80	0xC0	0x100	0x140	0x180	0x1C0	0x200	0x240	0x280	0x2C0	0x300	0x340	0x380	0x3C0	0x400	0x440

Cache organization

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

cache block location (address/64) % (cache size / N)

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



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

Read 0x300

Evict the “least recently used” value

Cache organization

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

cache block location (address/64) % (cache size / N)

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

Cache

value	🟥	7	2	*set 1*
address	🟥	0x1C0	0x80	
value	6			*set 2*
address	0x180			

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 (address/64) % (cache size / N)

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

Cache

value	12	7	2	*set 1*
address	0x300	0x1C0	0x80	

value	6			*set 2*
address	0x180			

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

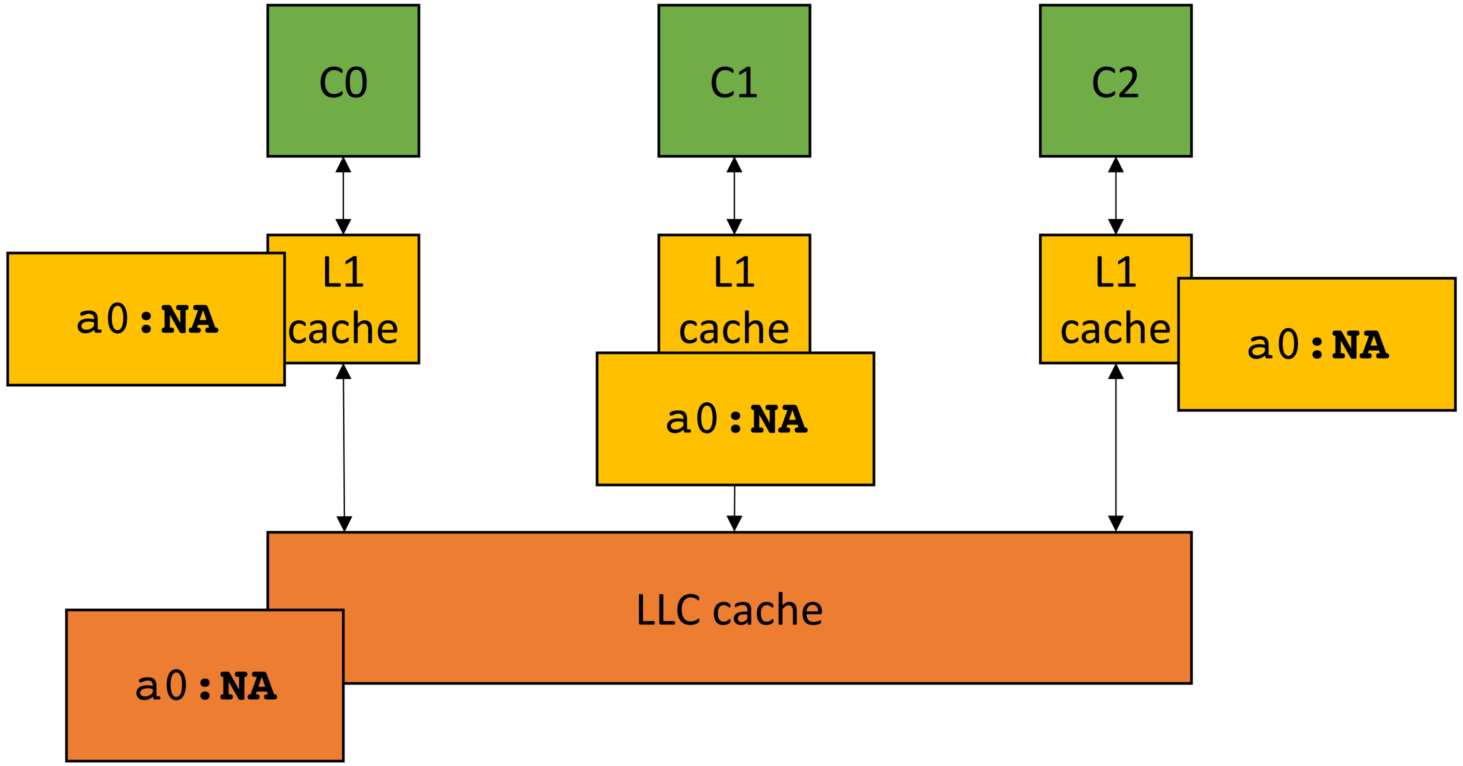
- For Intel Processors:
  - **L1** 8-way associative
  - **L2** 4-way associative
  - **L3** 12-way associative

Cache coherence

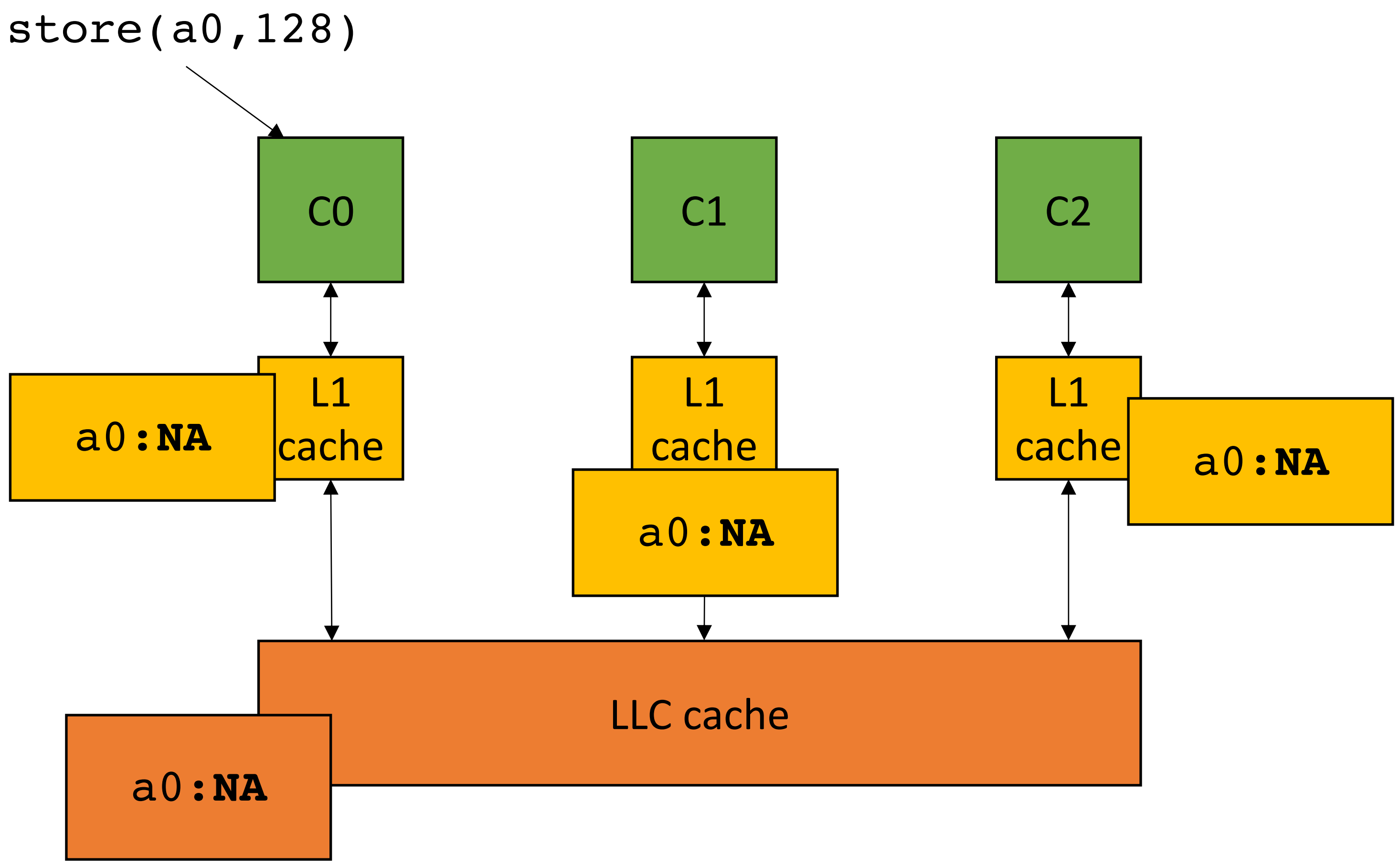
How to manage multiple values for the same address in the system?

simplified view for illustration: L1 cache and LLC

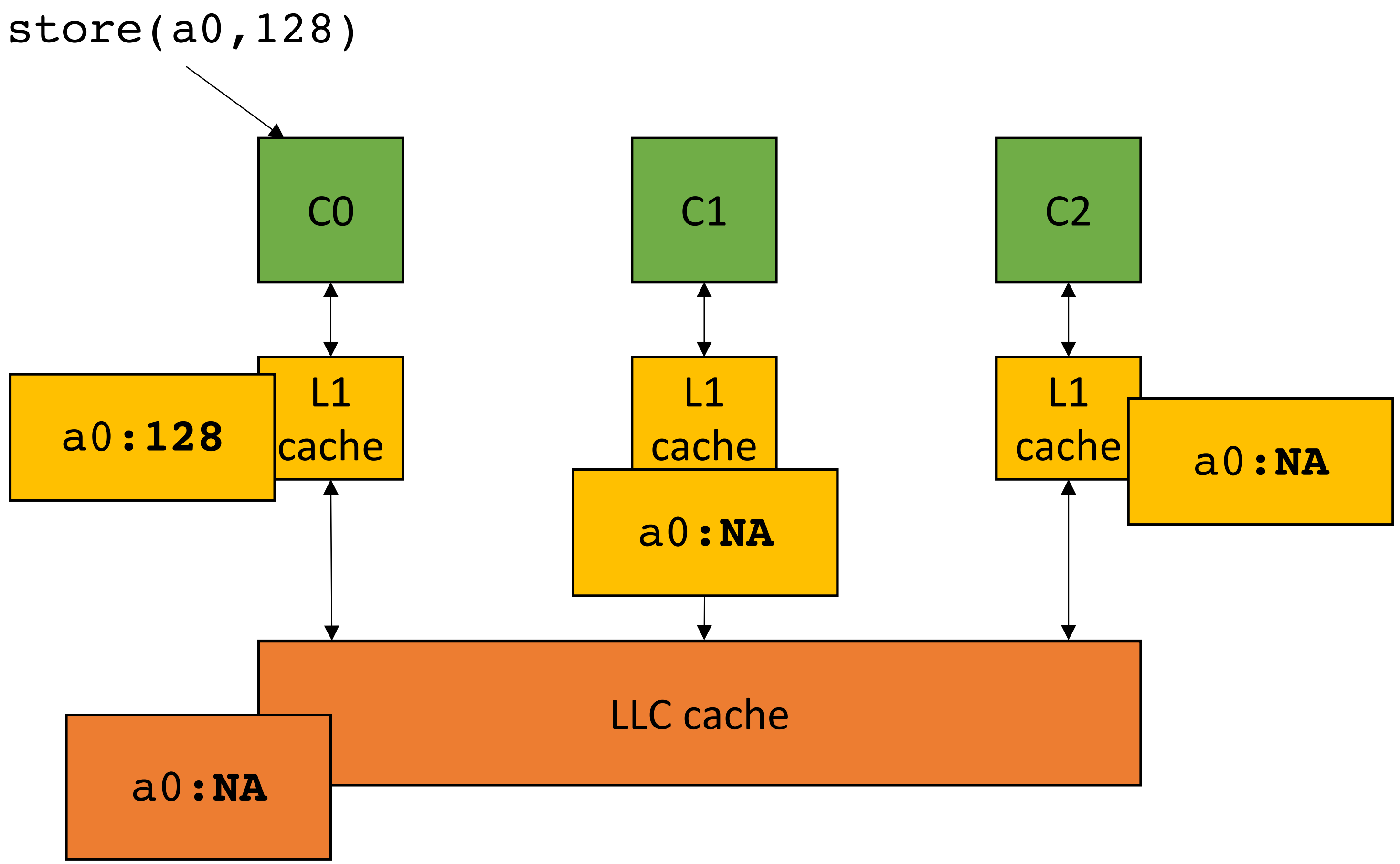
Consider 3 cores accessing the same memory location



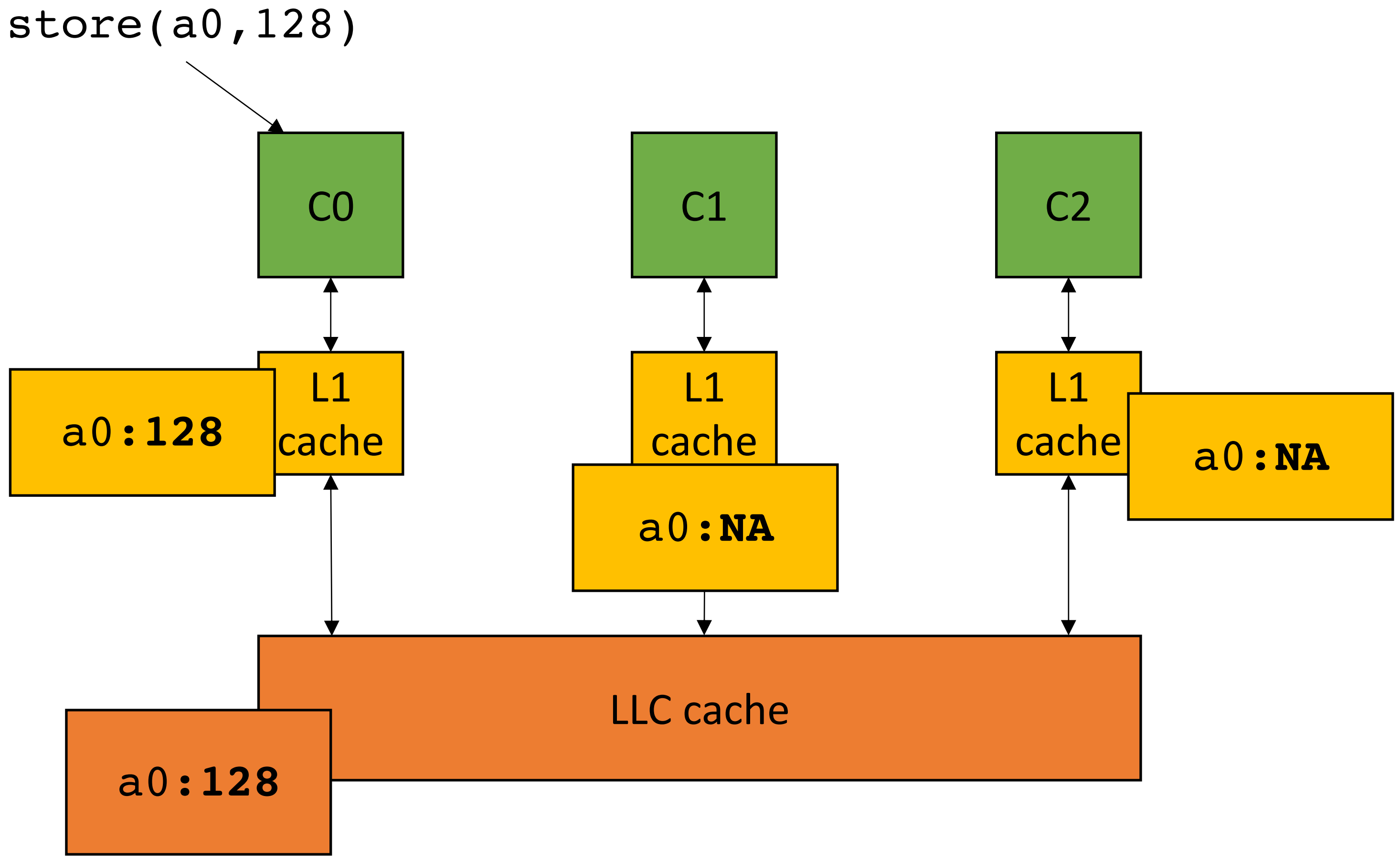
Cache coherence



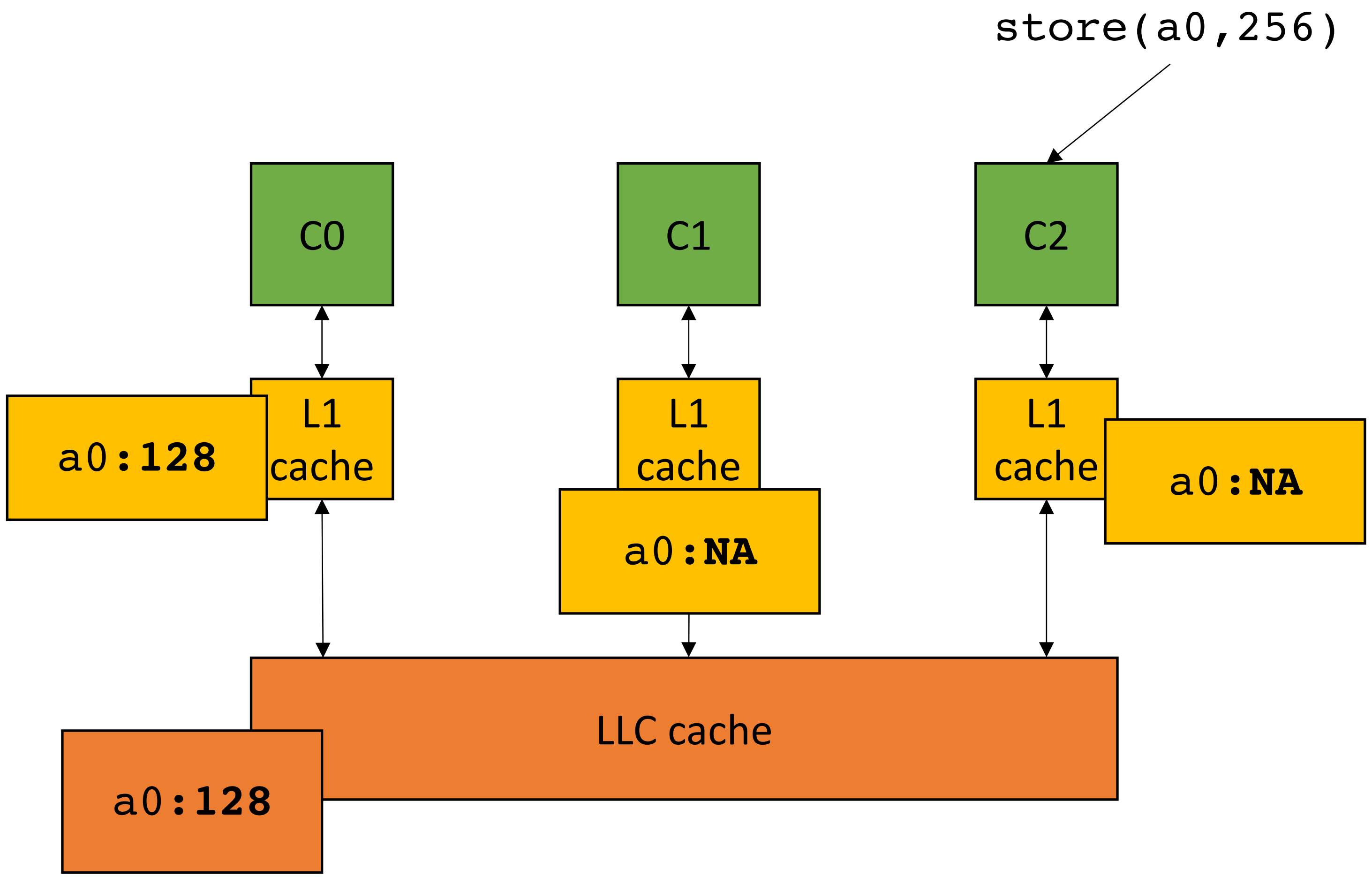
Cache coherence



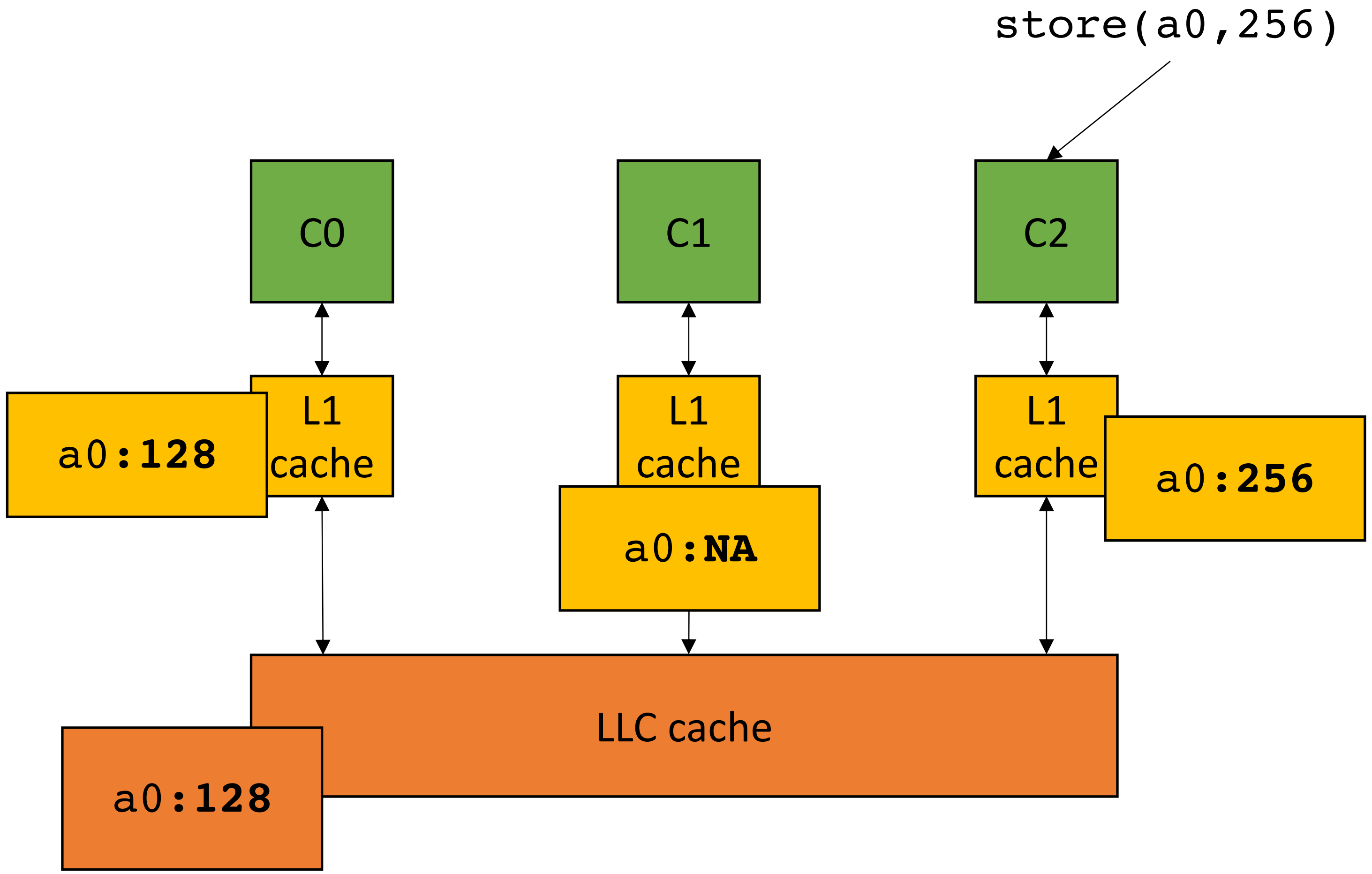
Cache coherence



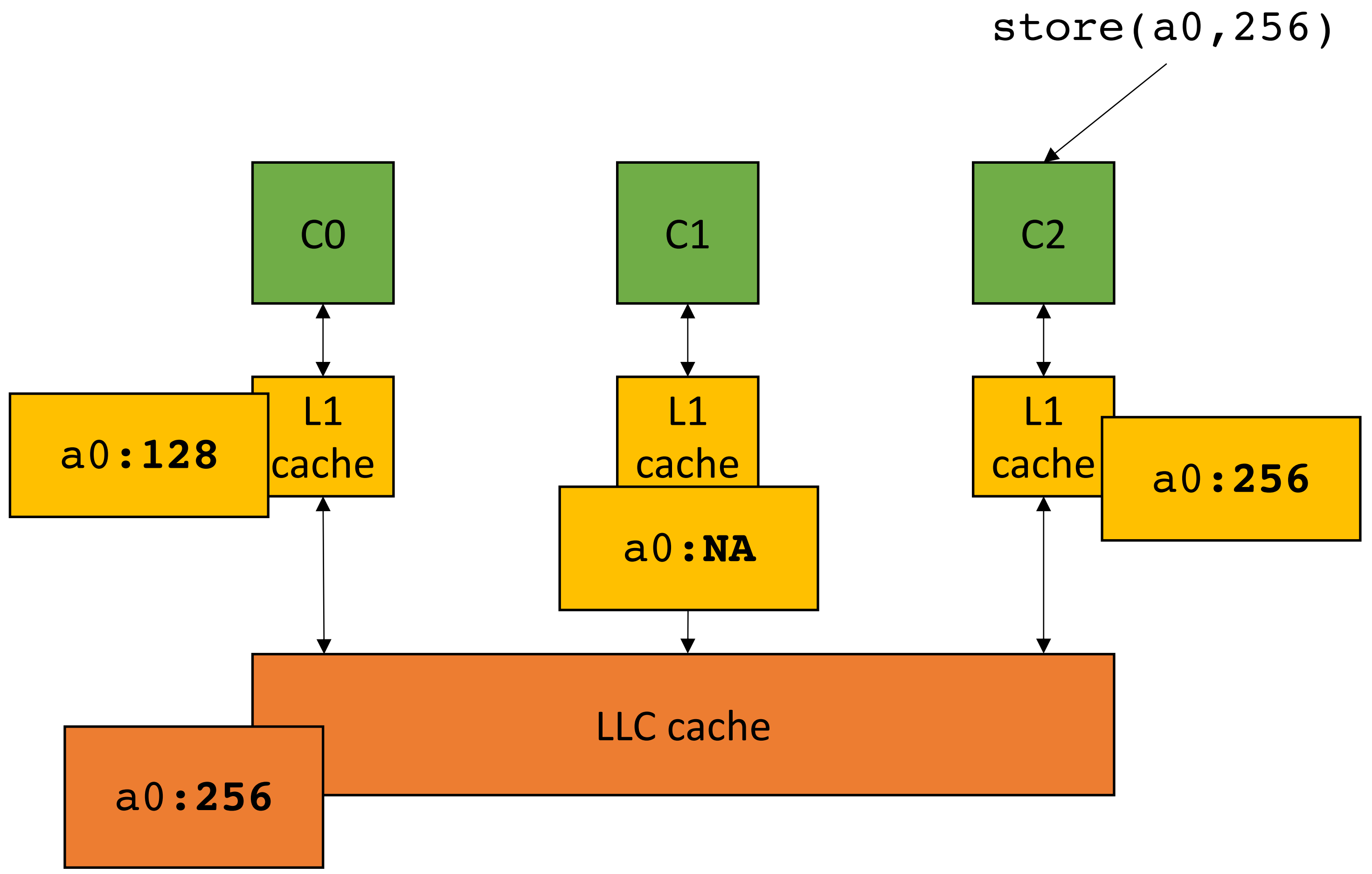
Cache coherence



Cache coherence

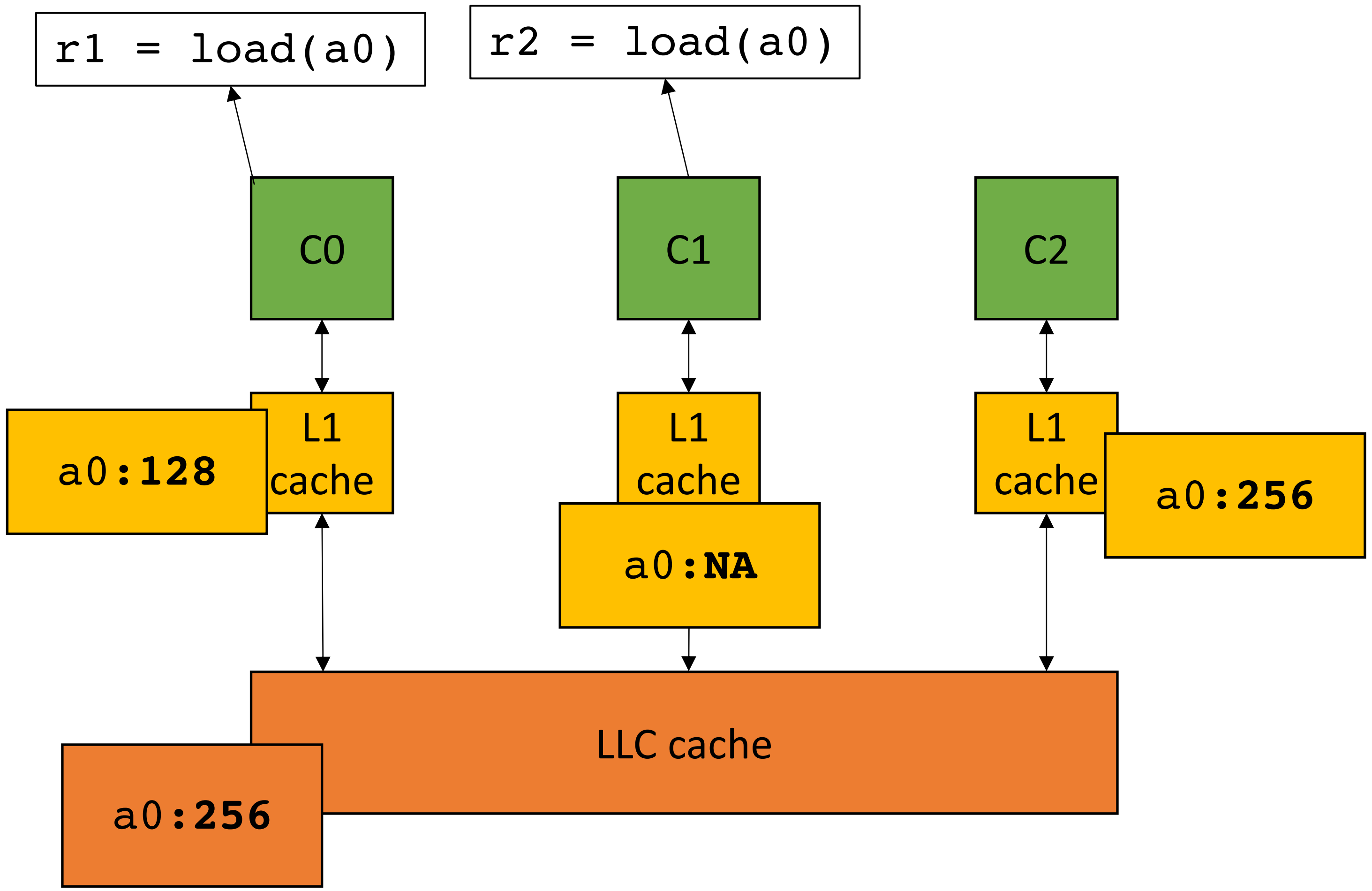


Cache coherence

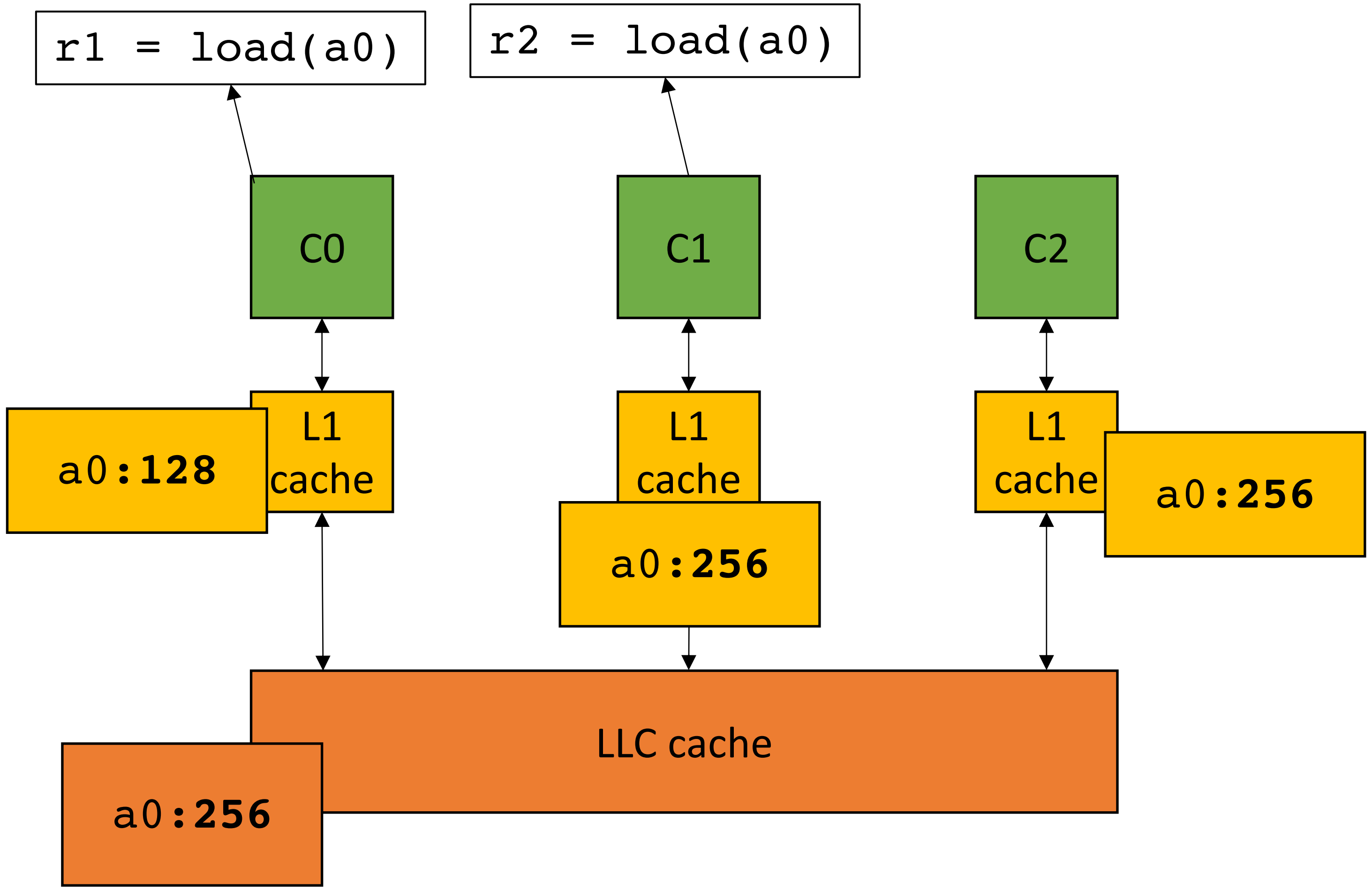


Cache coherence

in parallel

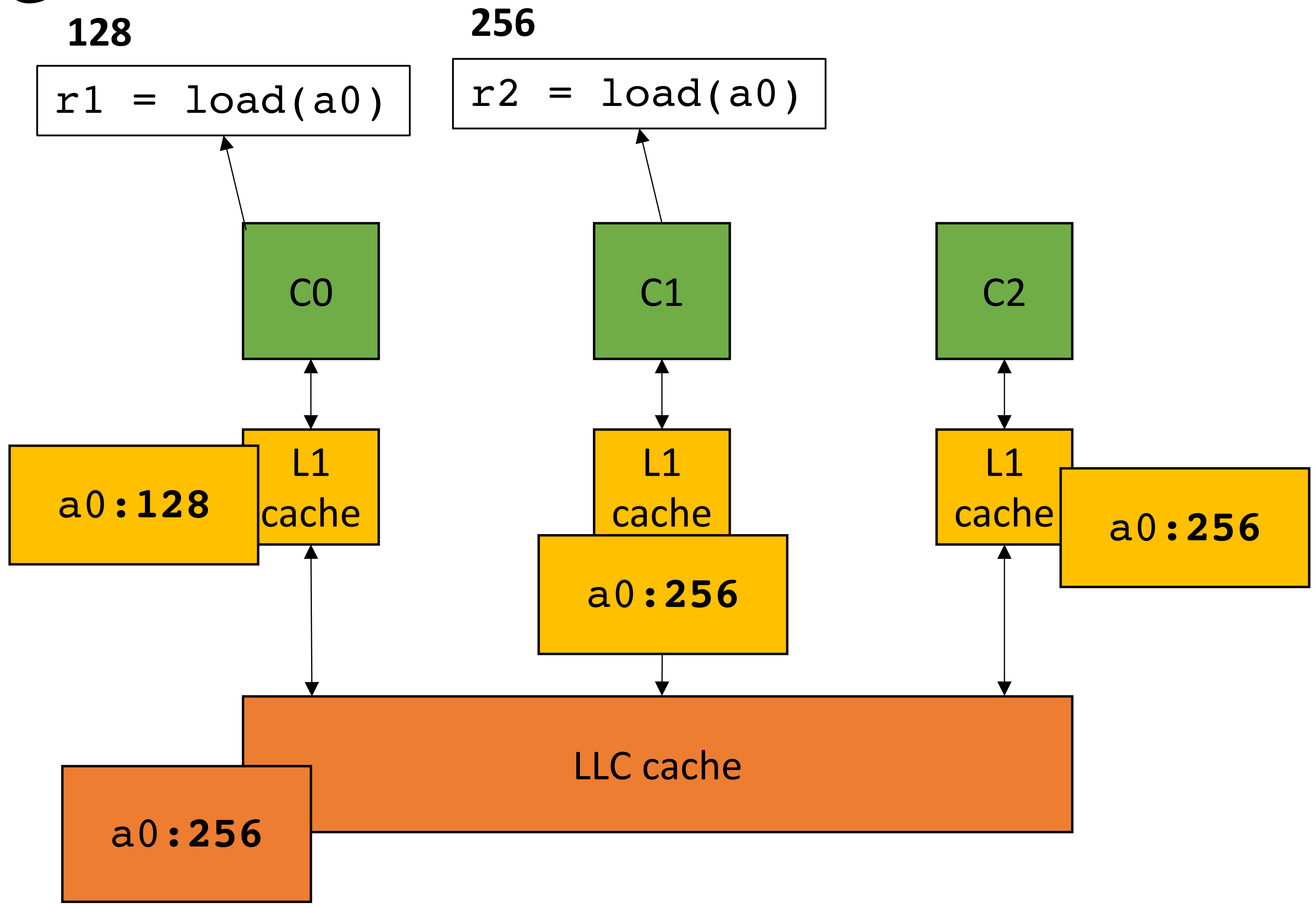


Cache coherence



Cache coherence

Incoherent view of values!

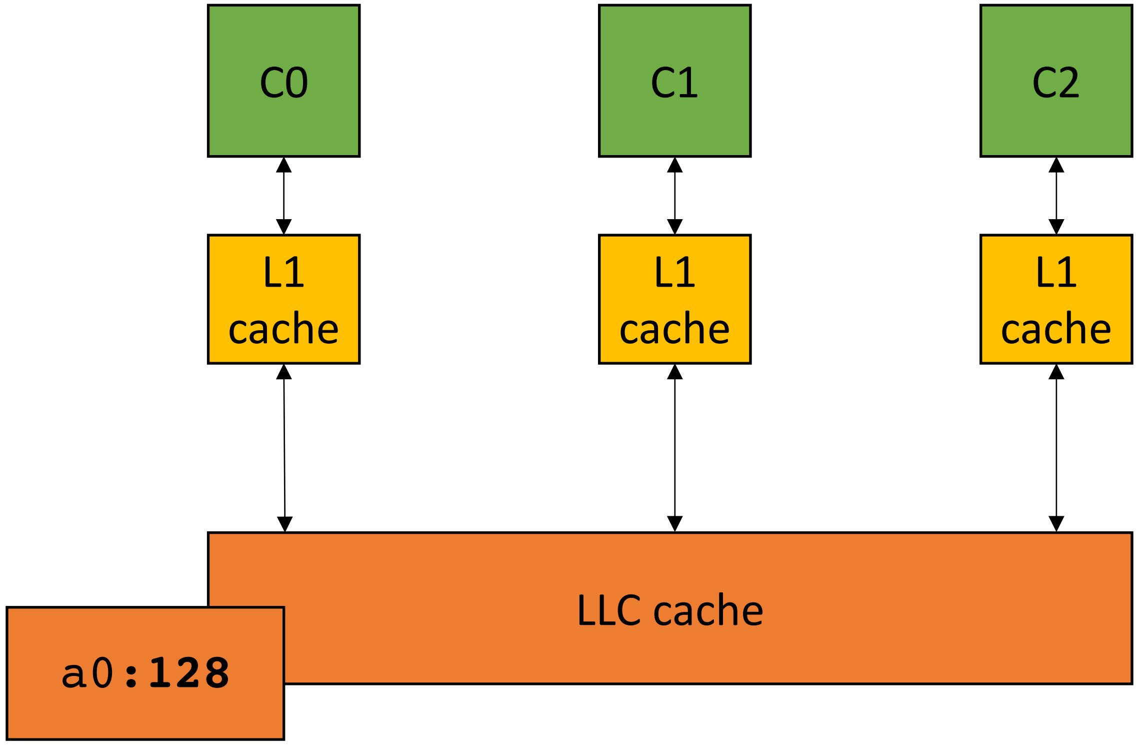


Cache coherence

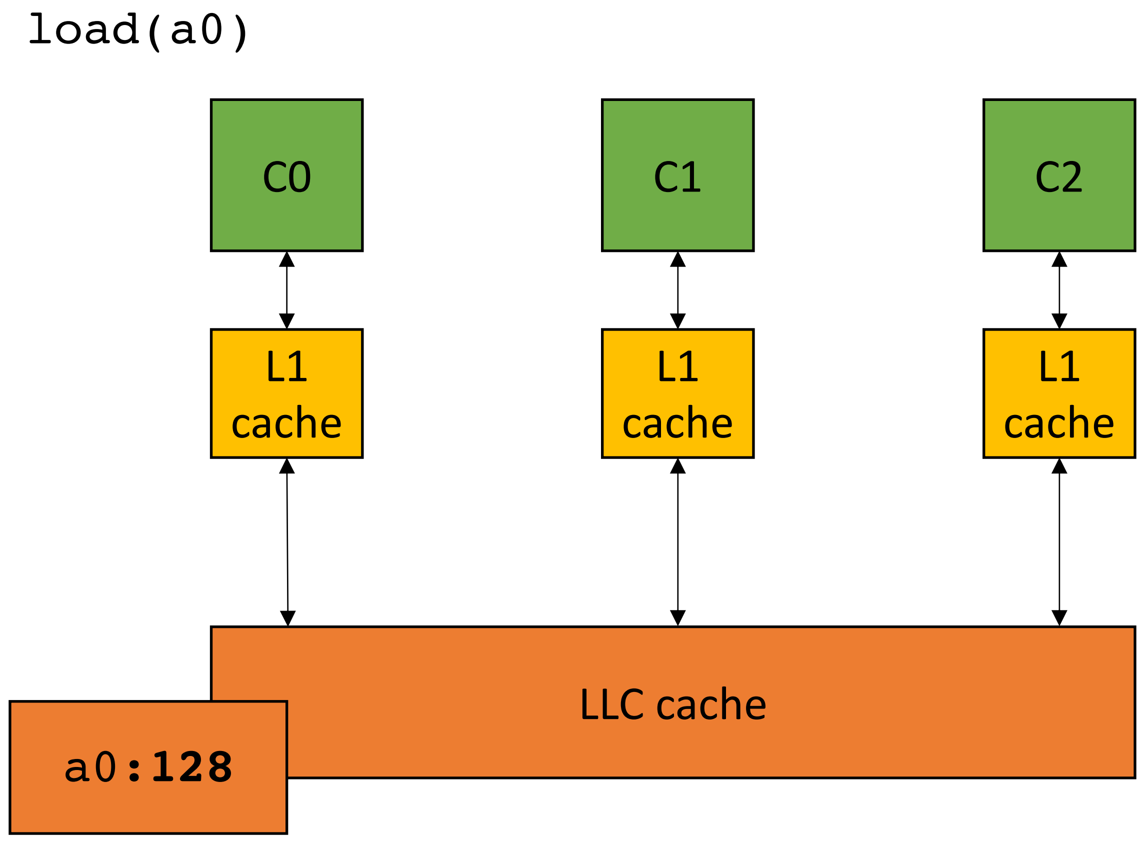
- MESI protocol

- 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

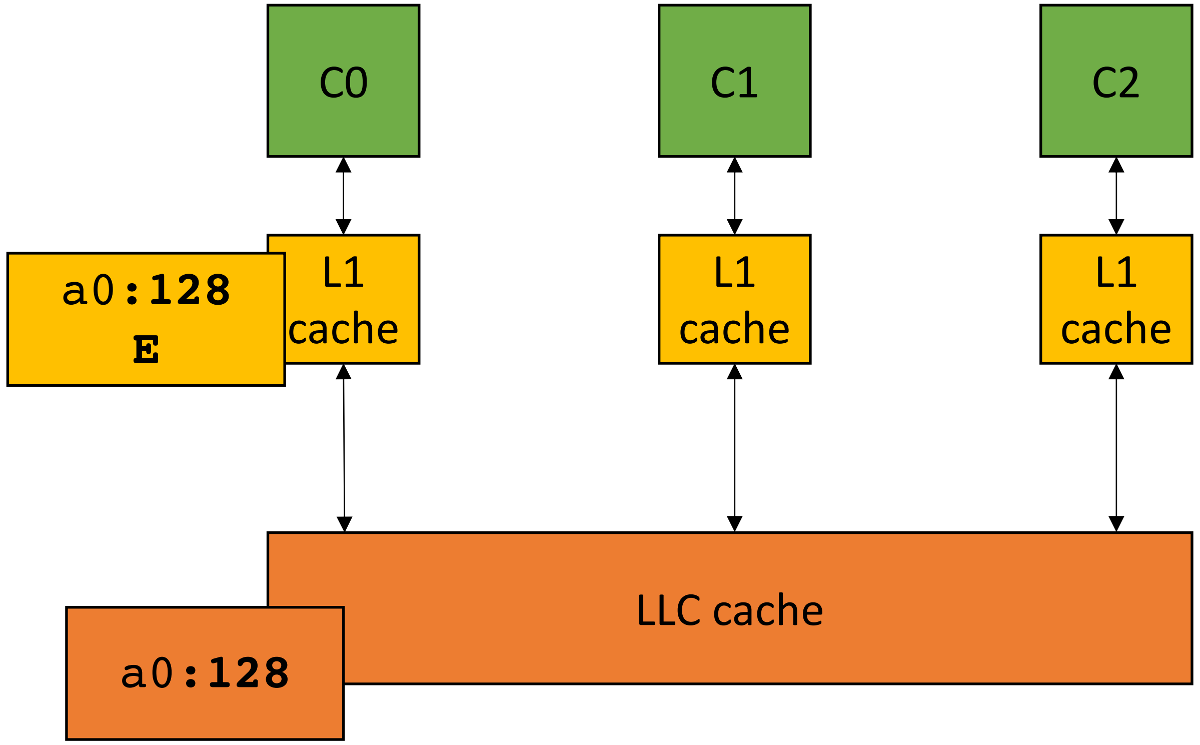


Cache coherence



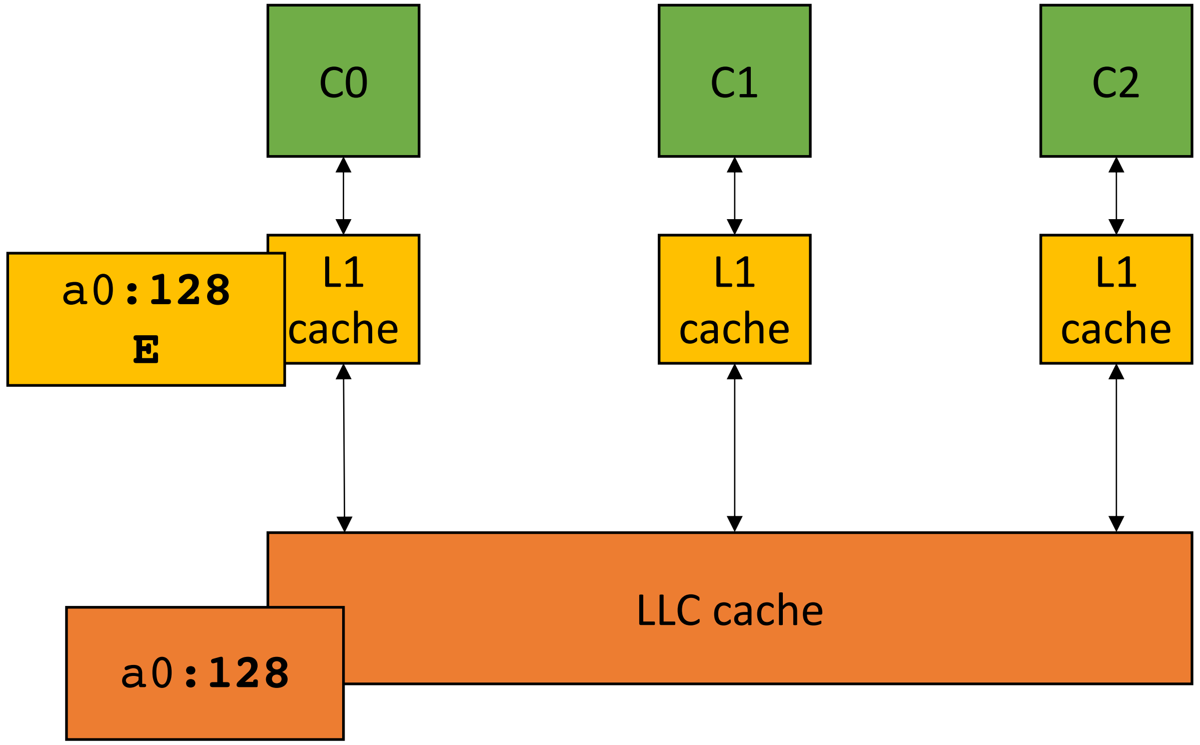
Cache coherence

Exclusive states are clean: they match main memory

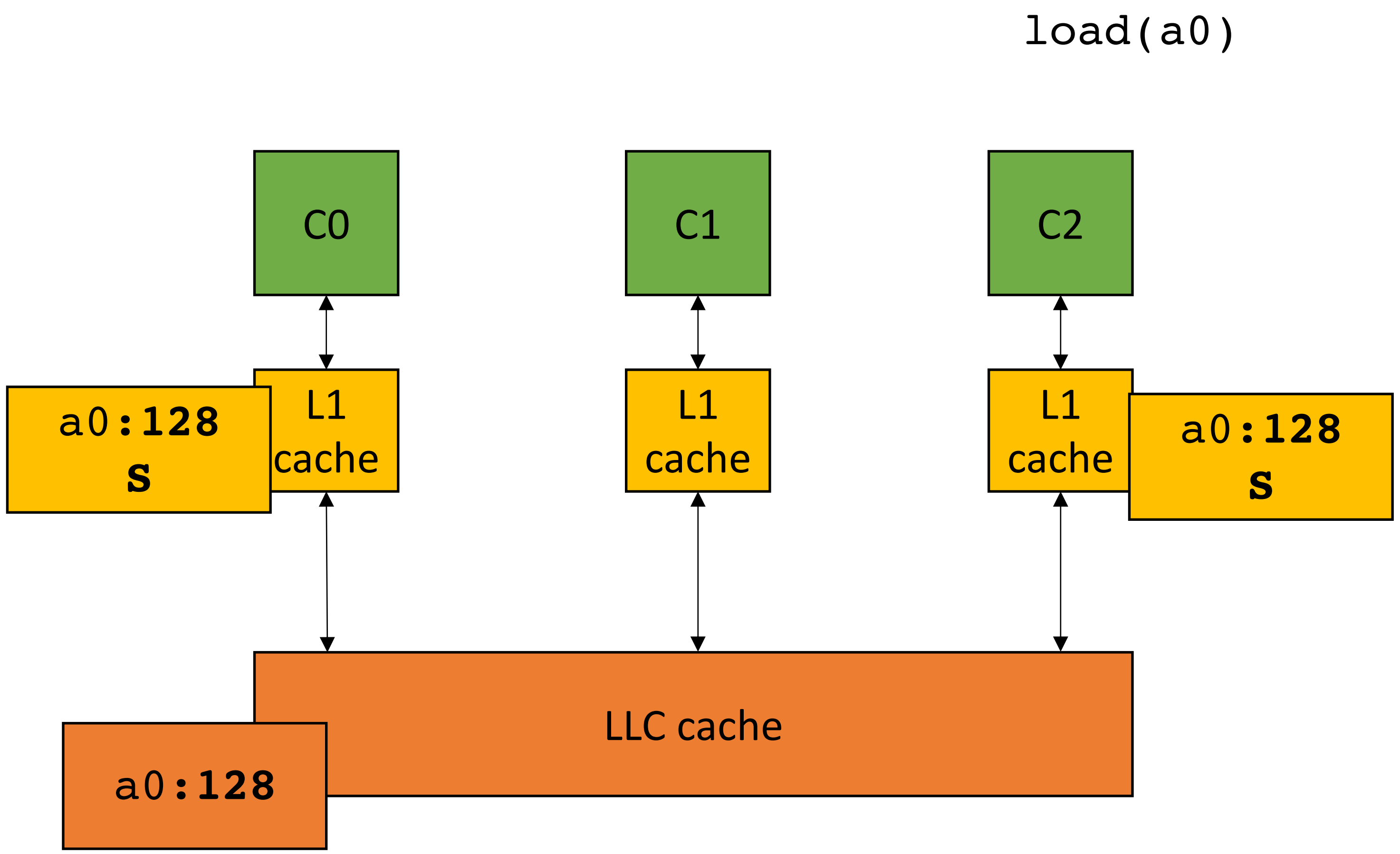


Cache coherence

```
load(a0)
```

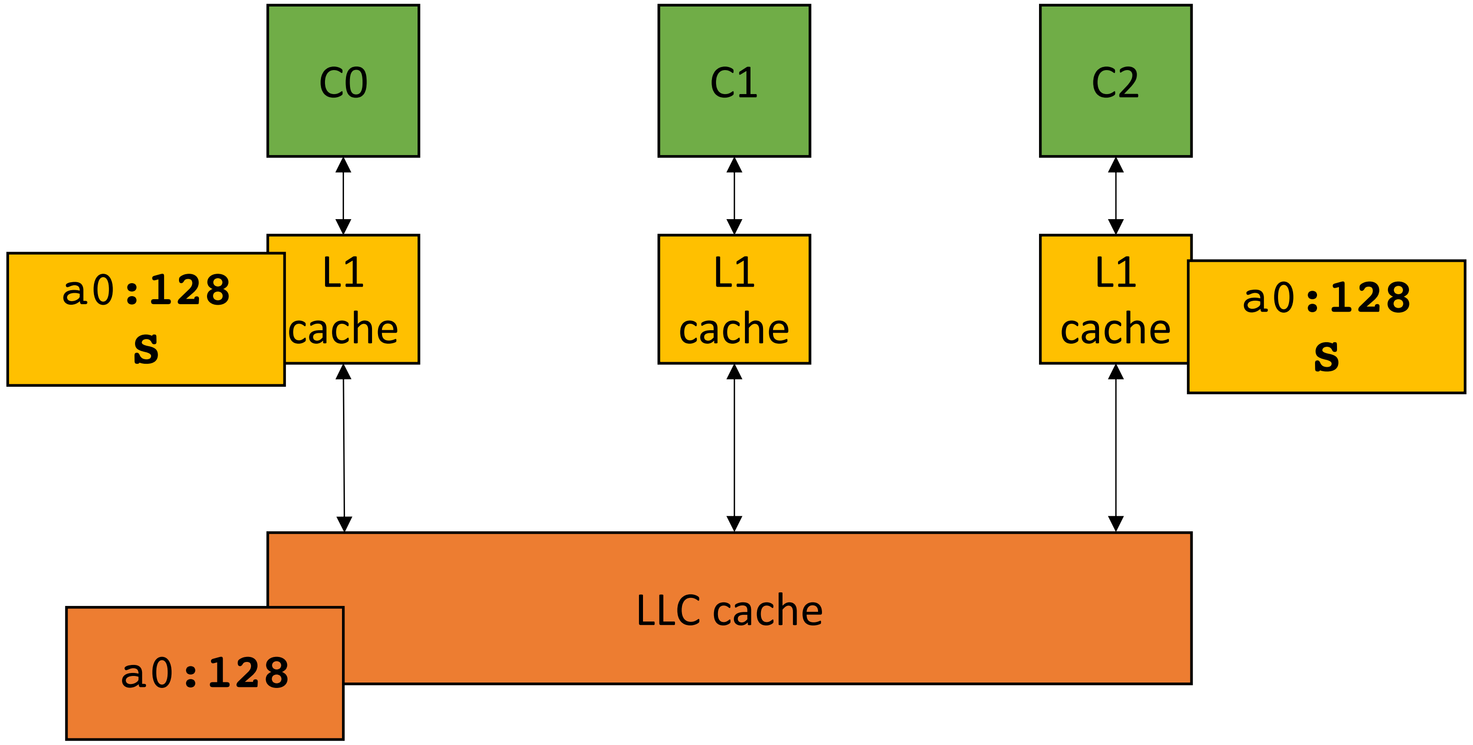


Cache coherence



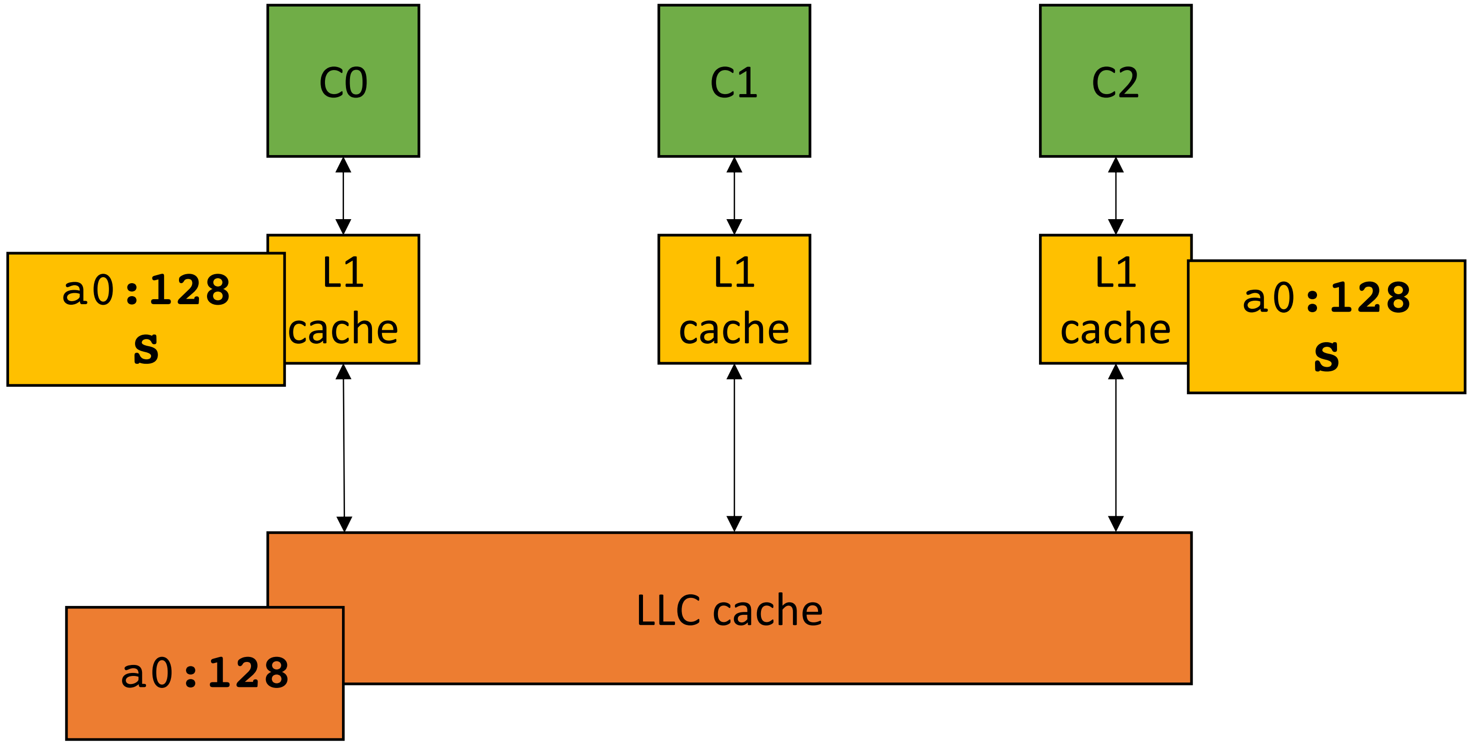
Shared states are clean: they match main memory

Cache coherence



Cache coherence

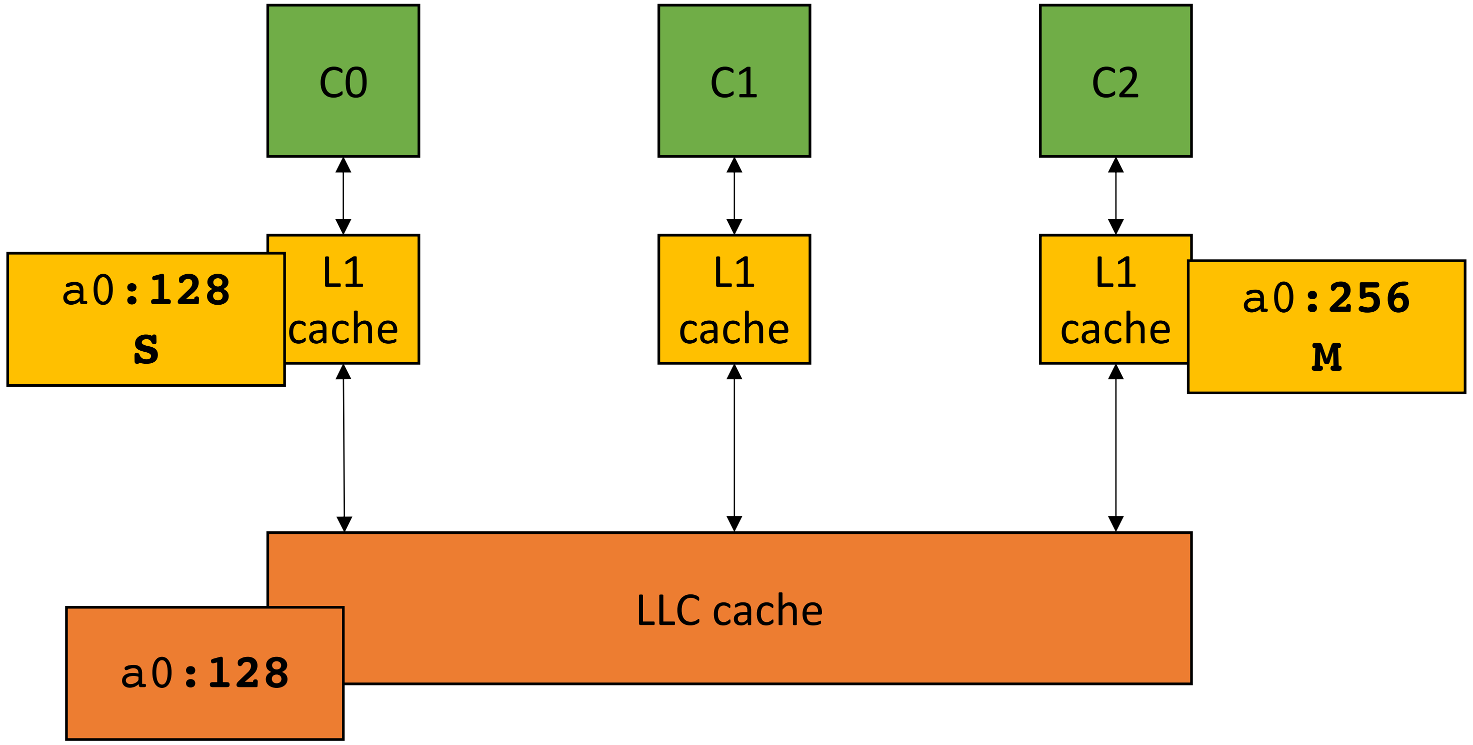
`store(a0,256)`



Cache coherence

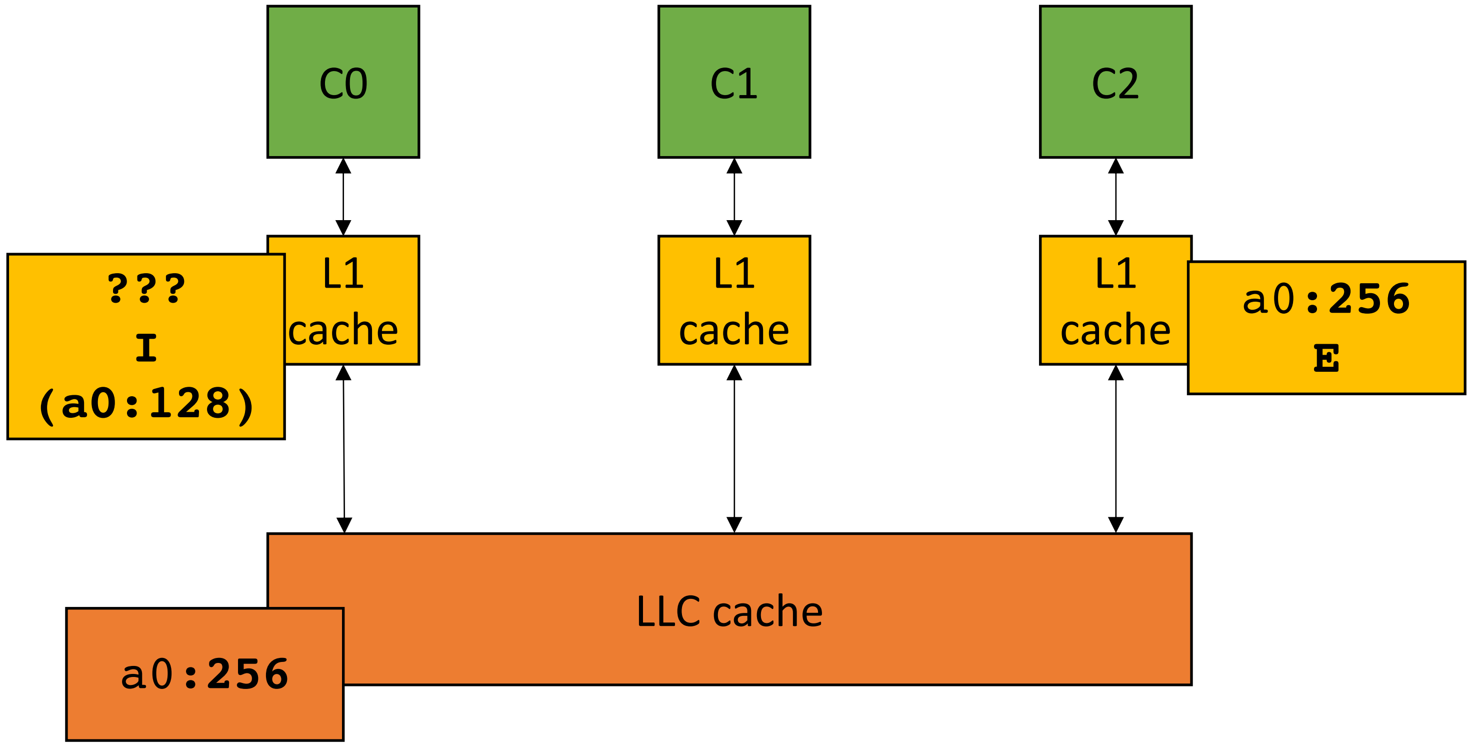
`store(a0,256)`

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

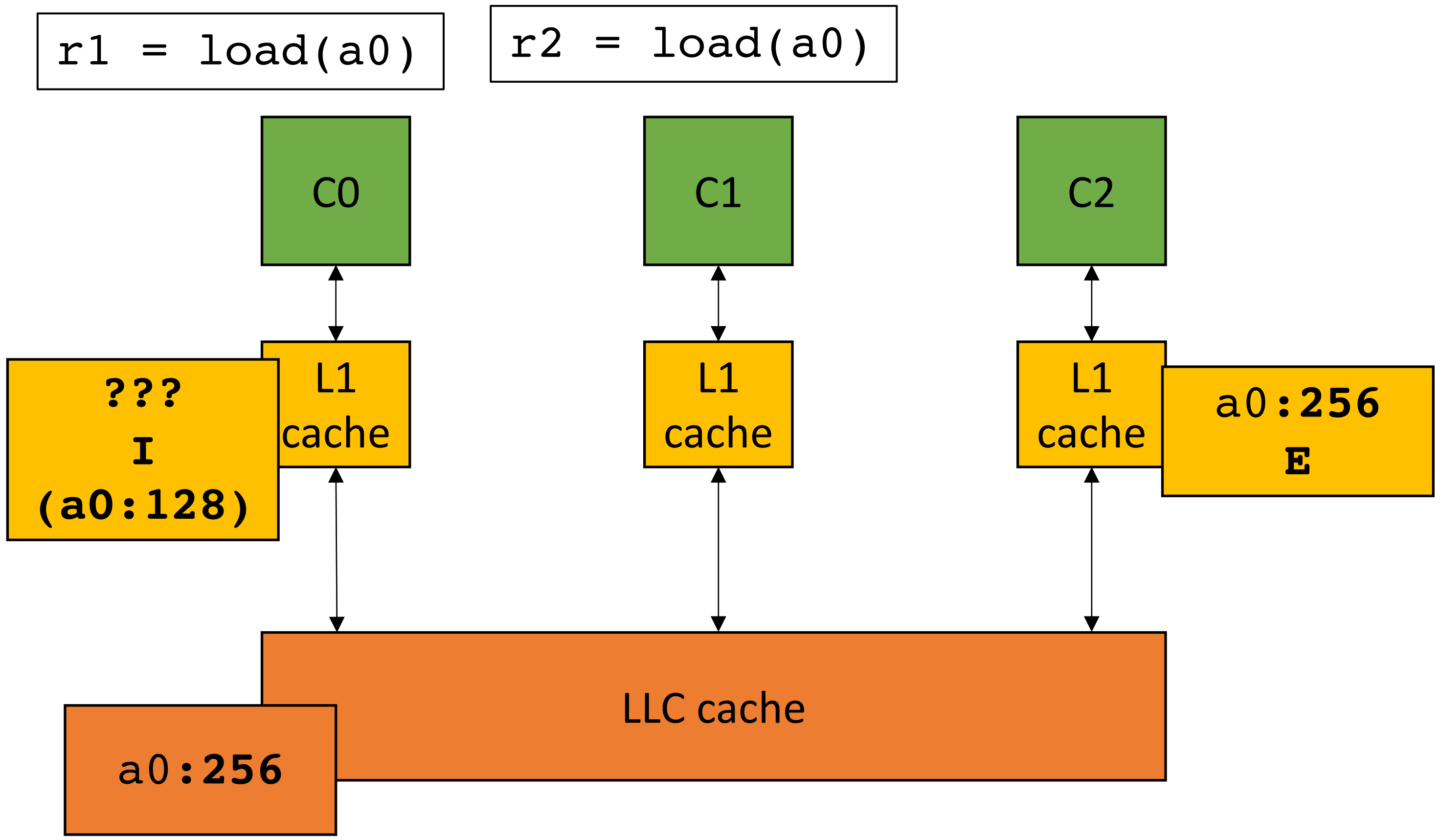


Cache coherence

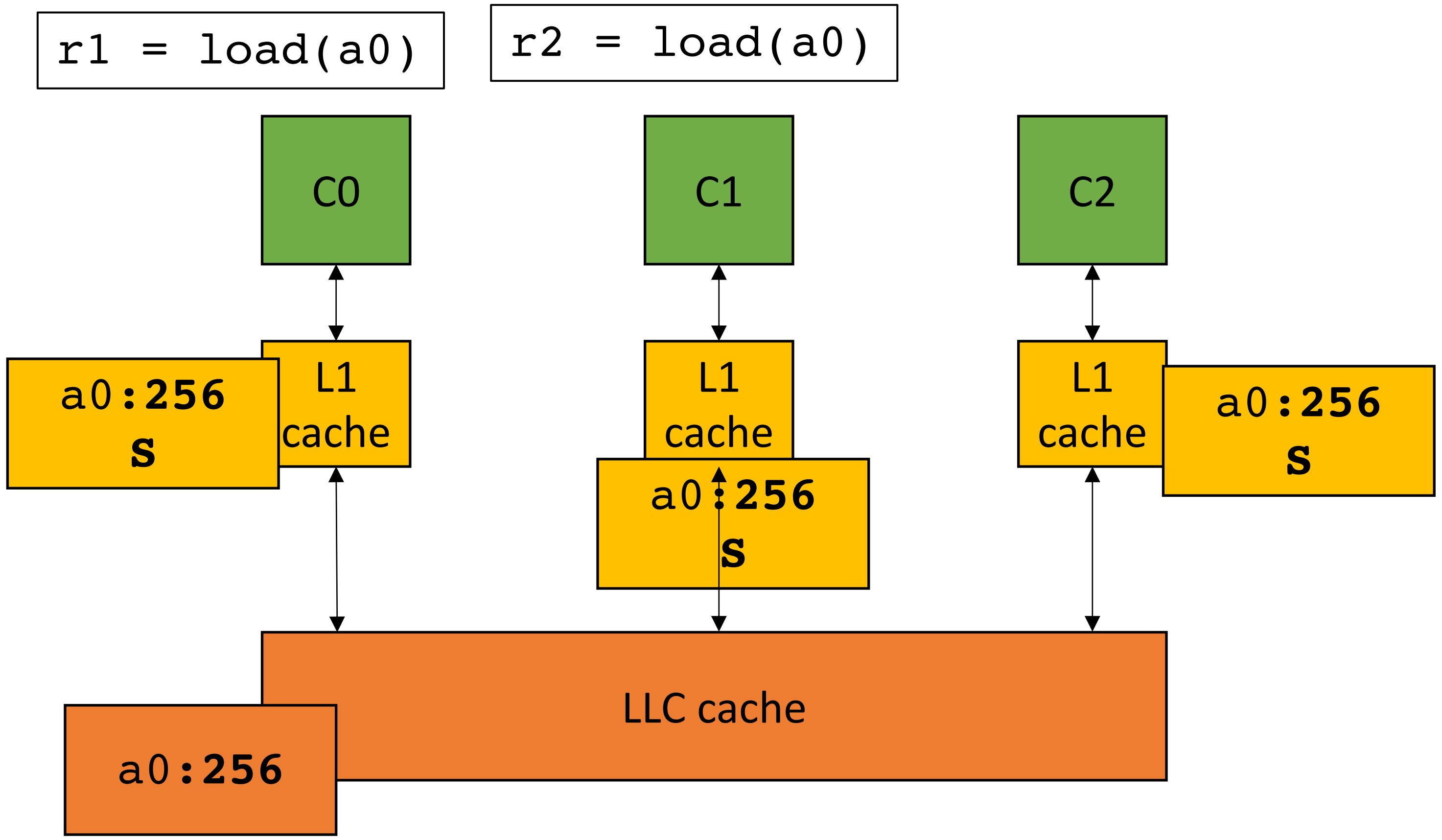
Invalid states
are considered unused



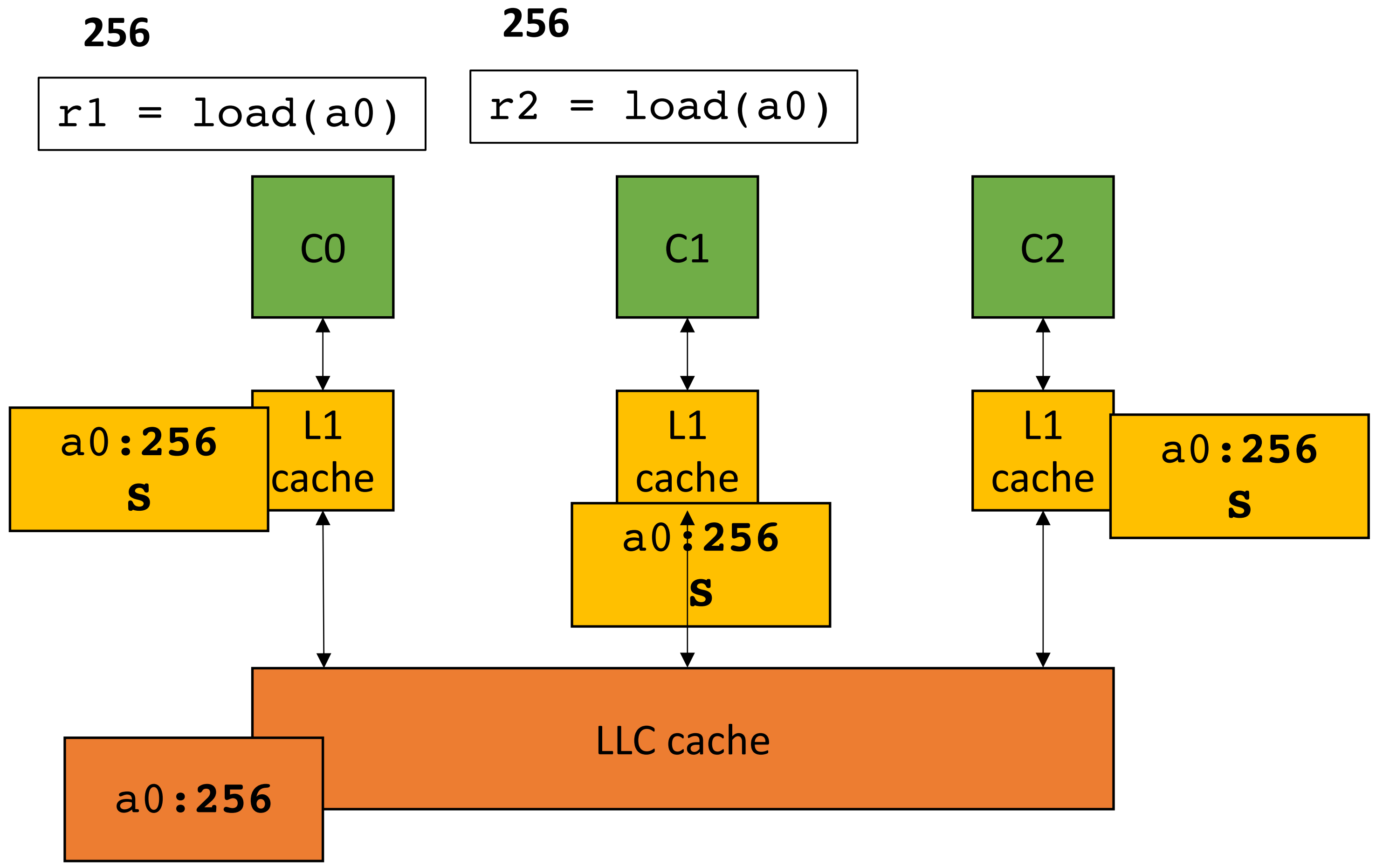
Cache coherence



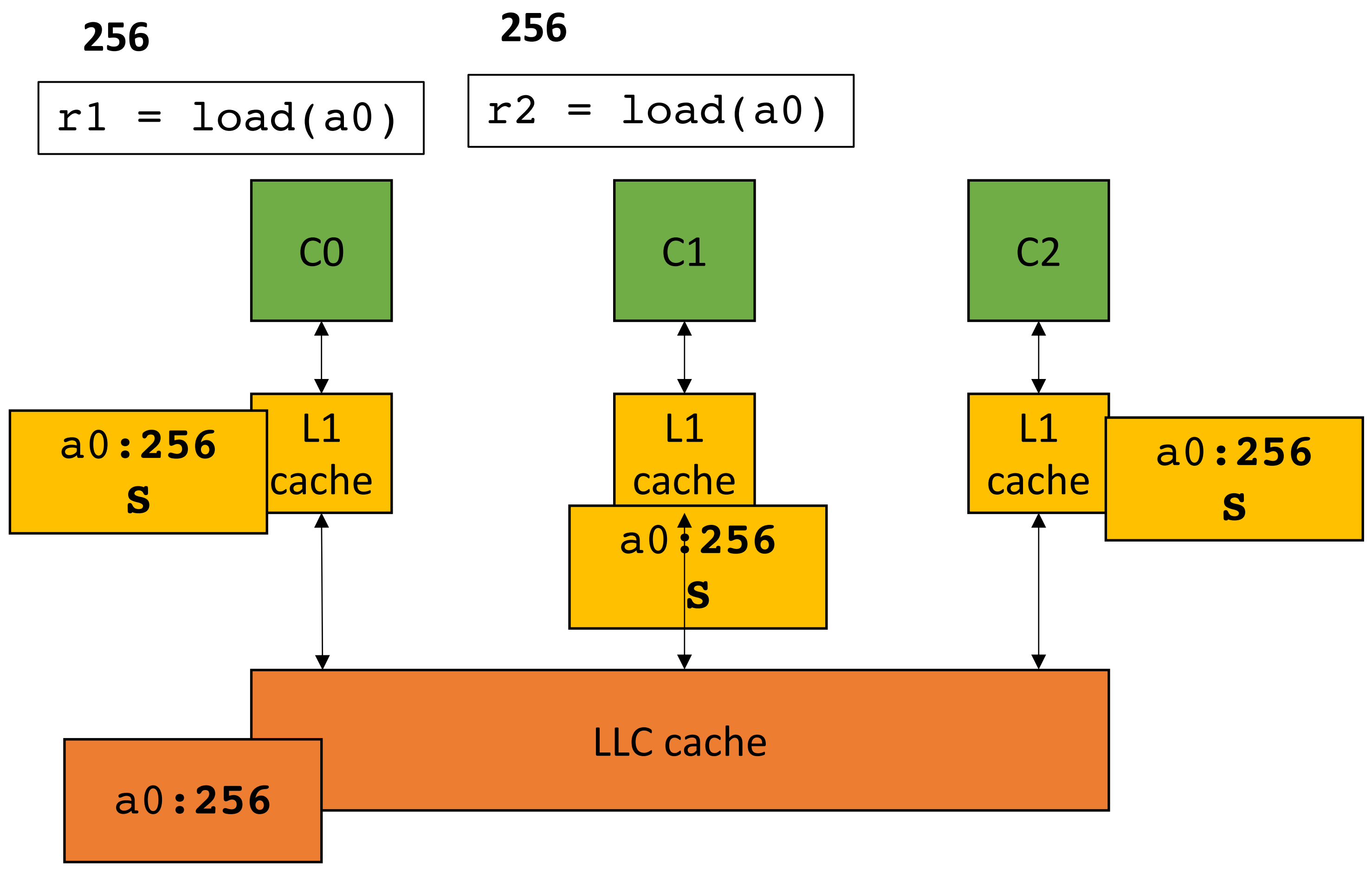
Cache coherence



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

Thank you!

- Remember to do the quiz today!

- Homework will be released on Monday
  - Due in 10 days
  - you can start by getting docker set up

- We will discuss ILP and C++ threads next week

- Have a good weekend: go do something fun!