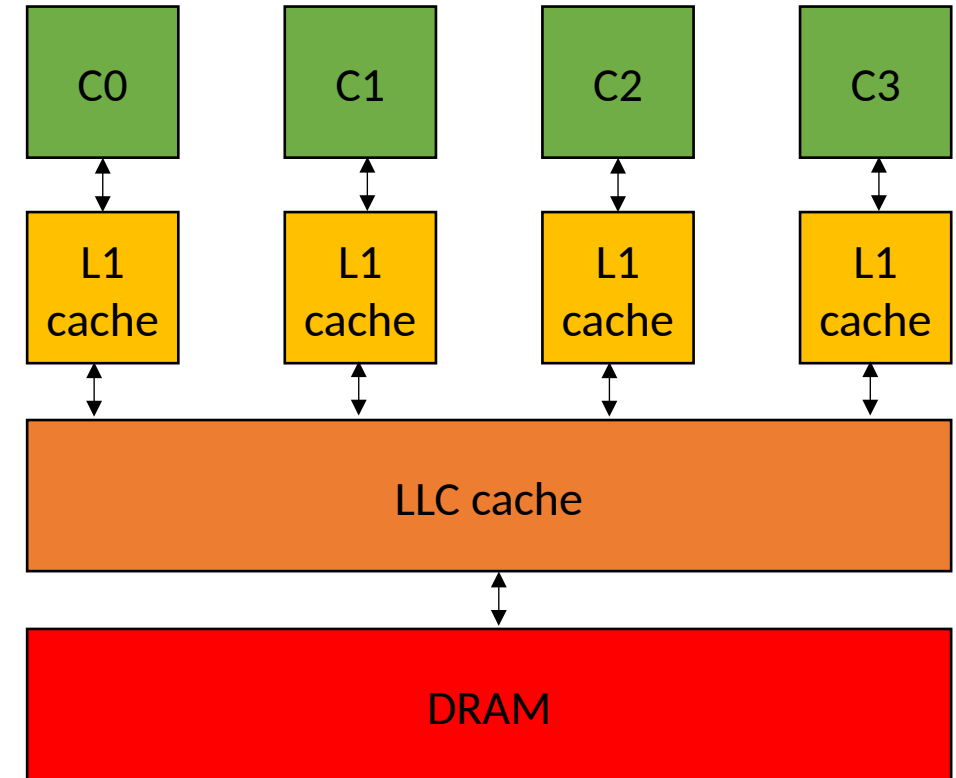# CSE113: Parallel Programming

- **Topic**: Architecture and Compiler Overview
  - Programming Language to ISA compilation
  - 3-address code
  - multiprocessors
  - memory hierarchy

Announcements

- Homework 1 released tonight (or Thursday)!
  - A week to do it
  - 3 free late days

- It will utilize github classroom and docker. There is a tutorial assignment. Please do it! (not graded, but you are expected to know it)

- Solutions require a design doc.
  - Not harshly graded but liable to lose points for low-effort
  - Forces you to think about your solution before you start

Announcements

- Instrcutor and TAs: Office hours announced on the webpage
- Tutors: Office hours announced by Thursday.

Quiz – Getting to know your classmates

What year are you in your studies?

Jr.	29 respondents	**32** %	✓
Sr.	63 respondents	68 %	
Grad Student		0 %	

Quiz – Getting to know your classmates

Python	90 respondents	**98** %	
C	91 respondents	99 %	
C++	78 respondents	85 %	
JavaScript	56 respondents	61 %	
GPU Programming	5 respondents	5 %	
Docker	32 respondents	35 %	
Unix command line	85 respondents	92 %	
console text editor (e.g. vim, emacs)	64 respondents	70 %	

Review

In a perfect world...

- Historically this worked well

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

- Dennard's scaling:
  - Computer speed doubles every 1.5 years.

2003

700 MHz

3x *increase*
over 4 years

2007

2.1 GHz

However...

These trends slowed down in ~2007

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz

1.2x increase
over 10 years

2017
2.5 GHz

2 cores

4 cores

Compiler refresher

Compilation:

x86

Language

```cpp
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

```
add(int, int): # @add(int, int)
push rbp
mov rbp, rsp
mov dword ptr [rbp - 4], edi
mov dword ptr [rbp - 8], esi
mov eax, dword ptr [rbp - 4]
add eax, dword ptr [rbp - 8]
pop rbp
ret
```

official specification

Intel provides a specification: *free*

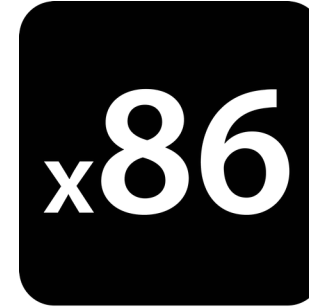2200 pages

Compilation:

Language


```cpp
int add(int a, int b) {
    return a + b;
}
```

Officially defined by the specification
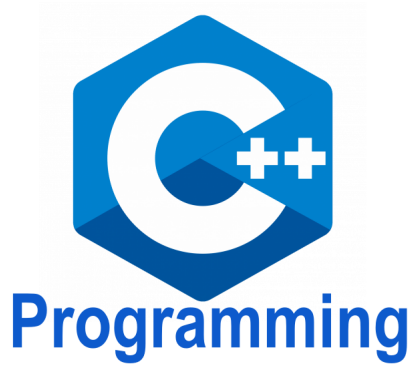
ISO standard: costs $200

~1400 pages


```
add(int, int):
sub sp, sp, #16
str w0, [sp, #12]
str w1, [sp, #8]
ldr w8, [sp, #12]
ldr w9, [sp, #8]
add w0, w8, w9
add sp, sp, #16
ret
```

How about a more complicated program?

Quadratic formula

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

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

x86

official specification
Intel provides a specification: *free*
2200 pages

There is not an ISA instruction that combines all these instructions!

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!

New material – Instruction Level Parallelism

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?

Instruction-level Parallelism (ILP)

- What type of instructions can be done in parallel?

two instructions can be executed in
parallel if they are independent

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)

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;
```

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

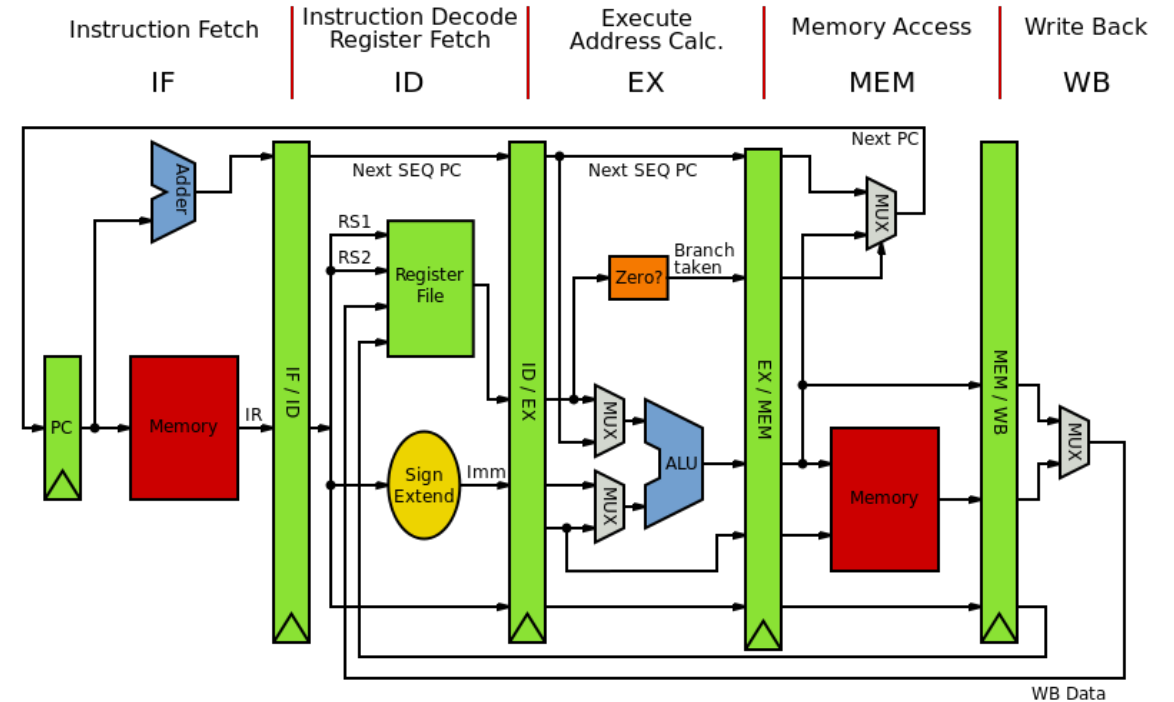*(assume all letter variables are registers)*

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
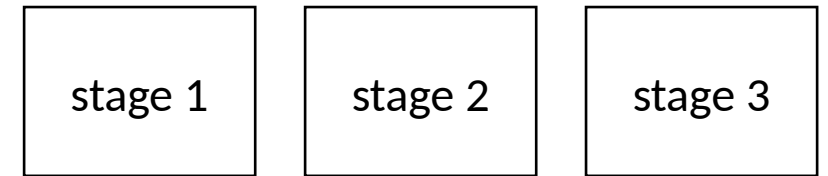 - **Dependencies stall** pipeline

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

Pipeline

- Pipeline parallelism

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

```
instr1;
instr2;
instr3;
```

| stage 1 | stage 2 | stage 3 |

Pipeline

- Pipeline parallelism

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

```
instr1;
```

```
instr2;
instr3;
```
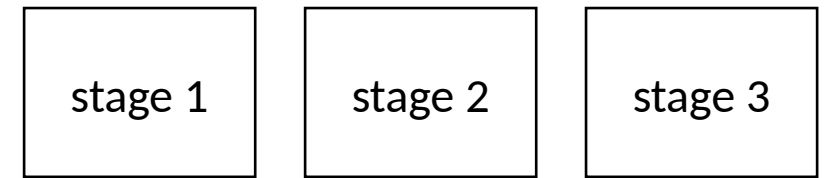
| stage 1 | stage 2 | stage 3 |

Pipeline

- Pipeline parallelism

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

stage 1	stage 2	stage 3

`instr2;`   `instr1;`

`instr3;`

Pipeline

- Pipeline parallelism

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

| stage 1 | stage 2 | stage 3 |

`instr3;`   `instr2;`   `instr1;`

Pipeline

- Pipeline parallelism

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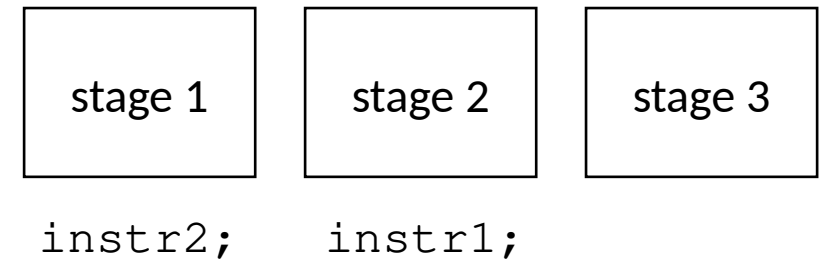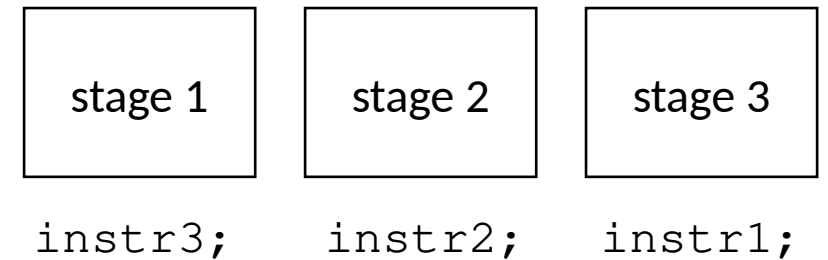 - N instructions can be in-flight
  - Dependencies stall pipeline

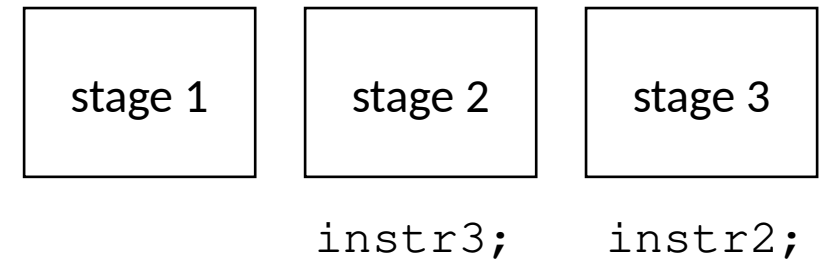| stage 1	stage 2	stage 3

```
         instr3;   instr2;
```

Pipeline

- Pipeline parallelism

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

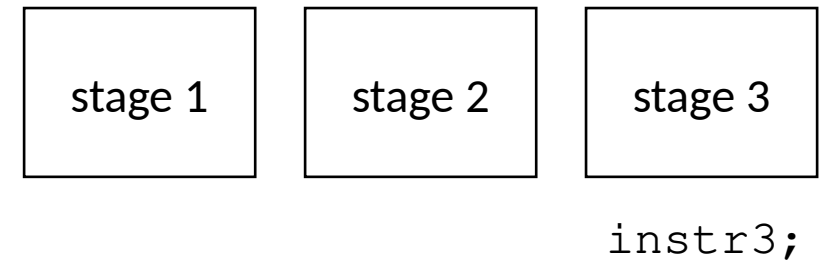| stage 1	stage 2	stage 3

`instr3;`

Pipeline

- Pipeline parallelism

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

| stage 1 | stage 2 | stage 3 |

6 cycles for 3 independent instructions
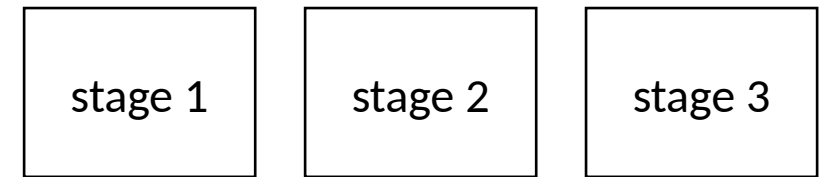
Converges 1 instruction per cycle

Pipeline

- Pipeline parallelism

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

stage 1	stage 2	stage 3

```
instr1;
instr2;
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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

stage 1	stage 2	stage 3

```
instr1;
```

```
instr2;
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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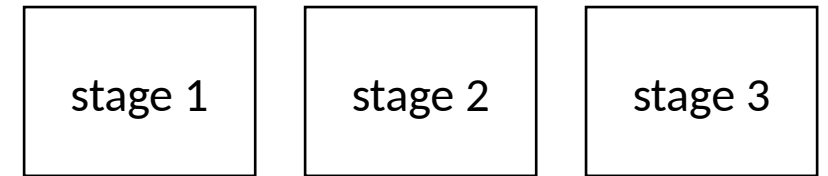 - Dependencies stall pipeline

| stage 1 | stage 2 | stage 3 |

```
instr1;
```

```
instr2;
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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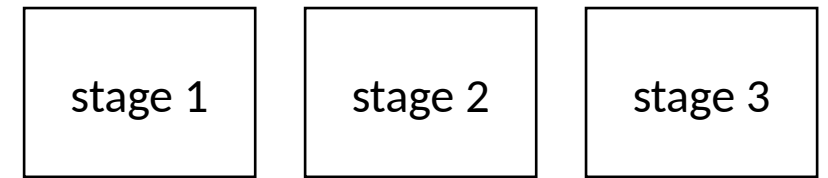 - Dependencies stall pipeline

stage 1	stage 2	stage 3

```
                                    instr1;
```

```
instr2;
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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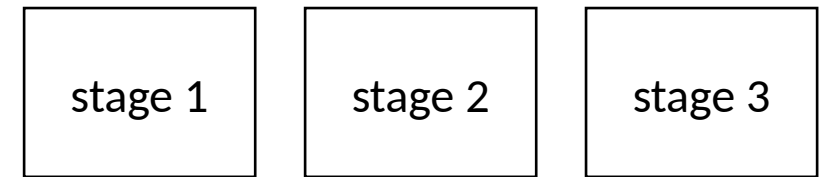 - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3

```
instr2;
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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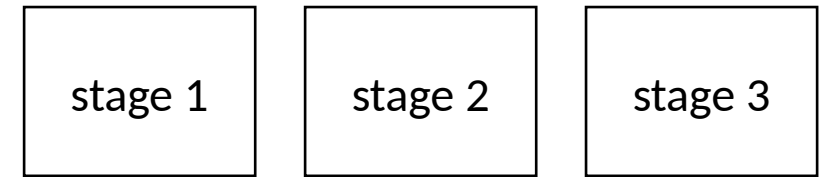 - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3

```
instr2;
```

```
instr3;
```

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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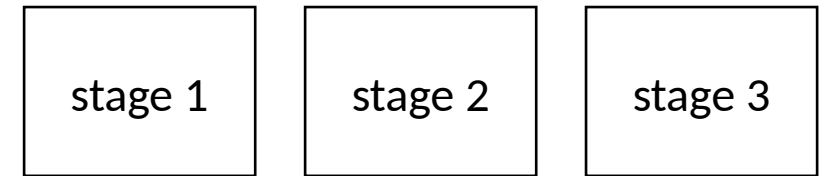 - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3

`instr2;`

`instr3;`

and so on…

*What if the instructions
depend on each other?*

Pipeline

- Pipeline parallelism

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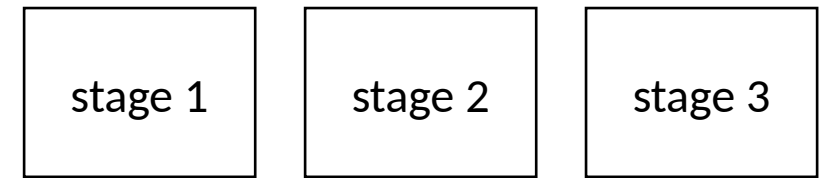 - Dependencies stall pipeline

| stage 1 | stage 2 | stage 3 |

What if the instructions depend on each other?

9 cycles for 3 instructions

converges to 3 cycles per instruction

Pipeline

- Pipeline parallelism

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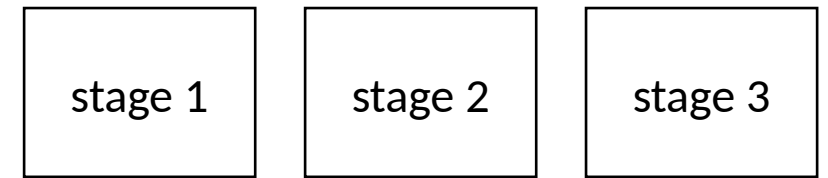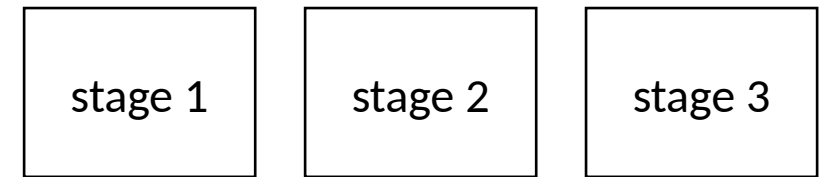 - Dependencies stall pipeline

stage 1	stage 2	stage 3

```
instr1;
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

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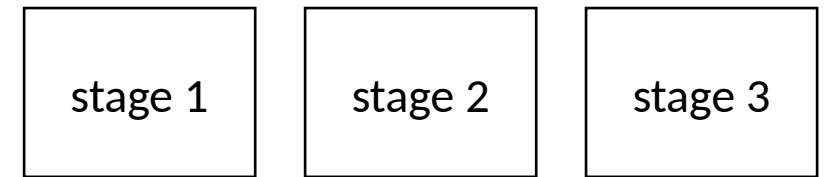   - N instructions can be in-flight
    - Dependencies stall pipeline

stage 1	stage 2	stage 3

`instr1;`

```
instrX0;
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

*If there are non-dependent
instructions from other places in
the program that we can interleave
then we can get back performance!*

Pipeline

- Pipeline parallelism

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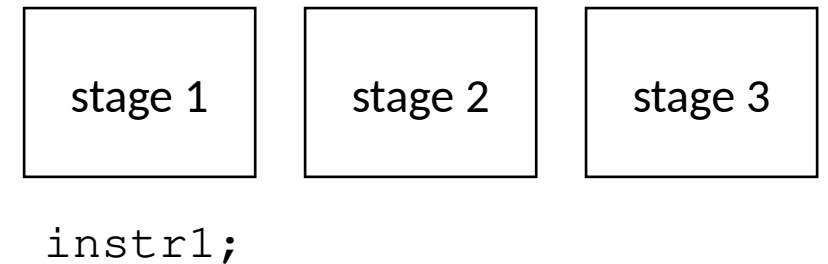   - N instructions can be in-flight
    - Dependencies stall pipeline

stage 1	stage 2	stage 3

`instrX0;`   `instr1;`

```
instrX1;
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

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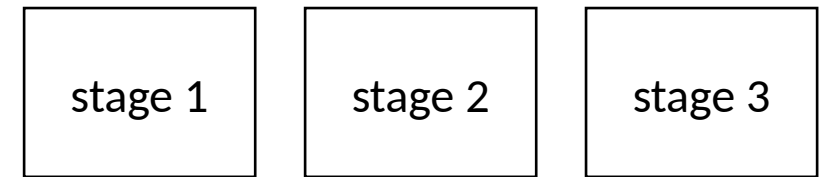 - N instructions can be in-flight
  - Dependencies stall pipeline

stage 1	stage 2	stage 3

`instrX1;`   `instrX0;`   `instr1;`

```
instr2;
instrX2;
instrX3;
instr3;
```

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

Pipeline

- Pipeline parallelism

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

stage 1	stage 2	stage 3

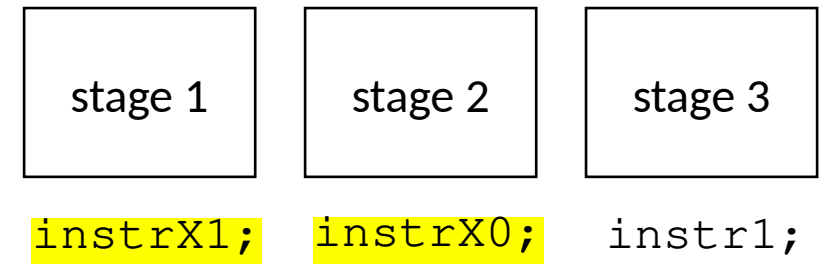```
instr2;   instrX1;   instrX0;
```

```
instrX2;
instrX3;
instr3;
```

*If there are non-dependent
instructions from other places in
the program that we can interleave
then we can get back performance!*

Pipeline

- Pipeline parallelism

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

stage 1	stage 2	stage 3

`instr2;`   `instrX1;`   `instrX0;`

`instrX2;`
`instrX3;`
`instr3;`

and so on…

We converge to 1 cycle per instruction again!

If there are non-dependent instructions from other places in the program that we can interleave then we can get back performance!

How can hardware execute ILP?

- Executing multiple instructions at once:

- Very Long Instruction Word (VLIW) architecture
    - Multiple instructions are combined into one by the compiler

- Superscalar architecture:
    - Several sequential operations are issued in parallel

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;
```

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

Independent Instructions

- Out-of-order execution
  - Hardware looks ahead for independent instructions
  - Hardware delays dependent instructions

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)

What does this mean for us?

- We should have an abstract performance model for instruction scheduling (the order of instructions)

- Try not to place dependent instructions in sequence

- Many times the compiler will help us here, but sometimes it cannot!

Two techniques to optimize for ILP

- Independent for loops (loop unrolling)

- Reduction for loops (loop unrolling)

What is loop unrolling?

```
for (int i = 0;   i < 12;   i++) {
  a[i] = b[i] + c[i];

}
```

Can we unroll this loop?
Data and control dependencies

```
for (int i = 0;   i < 6;   i+=2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

Using Loop Unrolling to Exploit ILP

- for loops with independent chains of computation

```
for (int i = 0; i < SIZE; i++) {
    SEQ(i);
}
```

where: `SEQ(i) = instr1;`   and let instr(N) depends on instr(N-1)
                   `instr2;`
                   `...`
                   `a[i] = instrN;`

loops only write to memory
addressed by the loop variable

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

Saves one addition and one comparison per loop, but doesn't help with ILP

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

Let green highlights indicate instructions from iteration `i`.

Let blue highlights indicate instructions from iteration `i + 1`.

Using Loop Unrolling to Exploit ILP

- Simple loop unrolling:

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

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

Using Loop Unrolling to Exploit ILP

- This is what you are doing in part 1 of homework 1

- You are playing the role of a compiler unrolling loops

- Your "compiler" is written in Python. You print out C++ code

- You the code is parameterized by dependency chain and by unroll factor

Loop Unrolling for Reduction Loops

- Prior approach examined loops with independent iterations and chains of dependent computations

- Now we will look at reduction loops:
  - Entire computation is dependent
  - Typically short bodies (addition, multiplication, max, min)

1	2	3	4	5	6

addition: 21

max: 6

min: 1

Loop Unrolling for Reduction Loops

- Simple implementation:

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

| 1 | 2 | 3 | 4 | 5 | 6 |

```
1 + 2 + 3 + 4 + 5 + 6
```

Loop Unrolling for Reduction Loops

- Simple implementation:

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

What is associativity?

1	2	3	4	5	6

```
(1 + 2 + 3) + (4 + 5 + 6)
```

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:

| 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:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
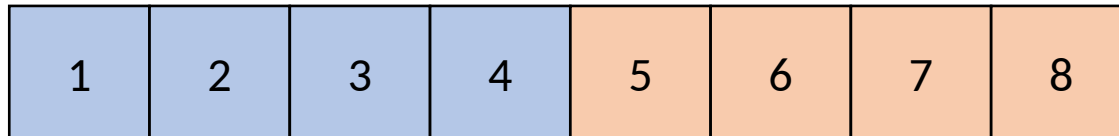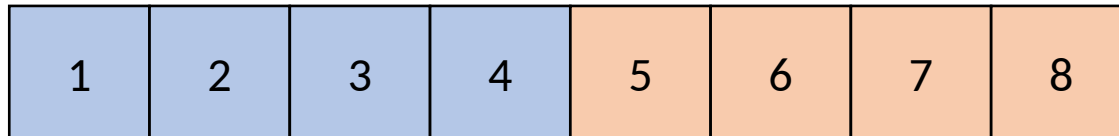
Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

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

10	2	3	4	26	6	7	8

Do addition reduction in base memory location

Loop Unrolling for Reduction Loops

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

10	2	3	4	26	6	7	8

Add together base locations

Loop Unrolling for Reduction Loops

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

10	2	3	4	26	6	7	8

Add together base locations

Loop Unrolling for Reduction Loops

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

36	2	3	4	26	6	7	8

Add together base locations

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])
```

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])
```

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])
```

independent instructions can be done in parallel!

Watch out!

- Our abstraction: separate dependent instructions as far as possible

- Pros:
  - Simple

- Cons:
  - Can lead to register spilling, causing expensive loads

consider `instr1` and `instr2` have a data dependence, and `instrX's` are independent

```
instr1;
instrX0;
instrX1;
...
Instr2;
```

independent instructions. If they overwrite the register storing `instr1's` result, then it will have to be stored to memory and retrieved before `instr2`

Watch out!

- Our abstraction: separate dependent instructions as far as possible

- Pros:
    - Simple

- Cons:
    - Can lead to register spilling, causing expensive loads

Solutions include using a **resource model** to guide the topological ordering. Highly architecture dependent. Compiler algorithms become more expensive.

Consider timing the compile time in your homework assignment.

Memory hierarchy overview
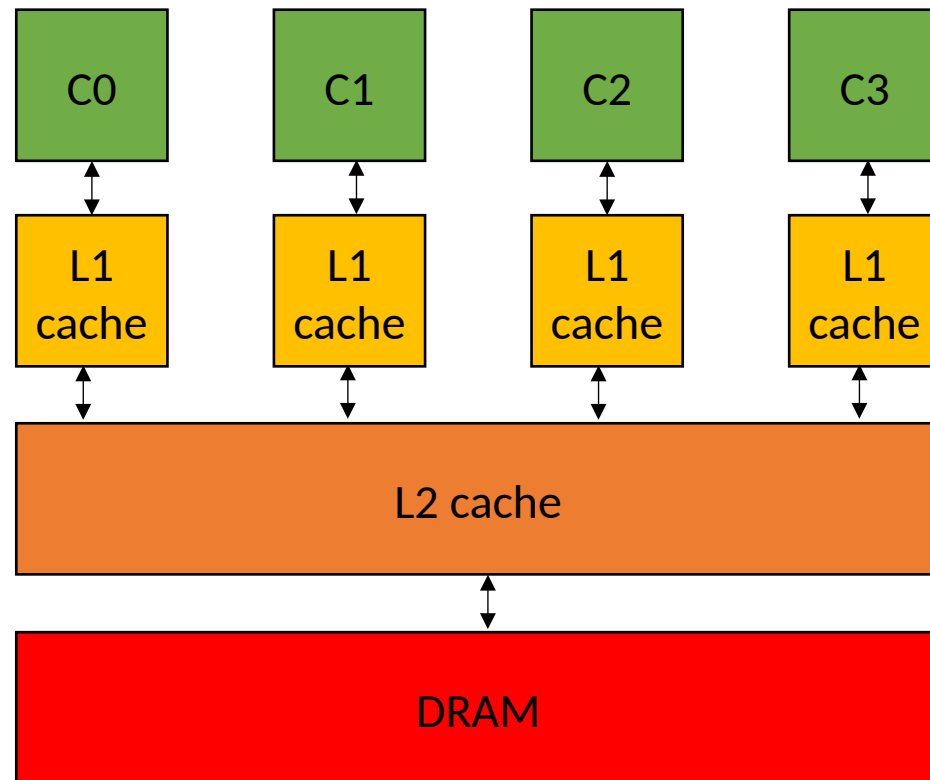
Core

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

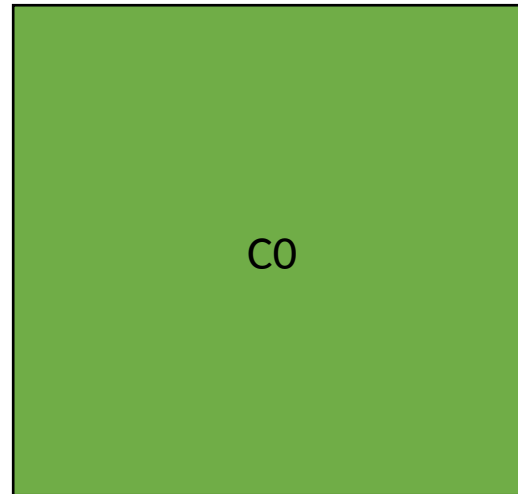Compiled function #0

```
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

C0

Core

Core

Sometimes multiple programs want to share the same core.

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
```
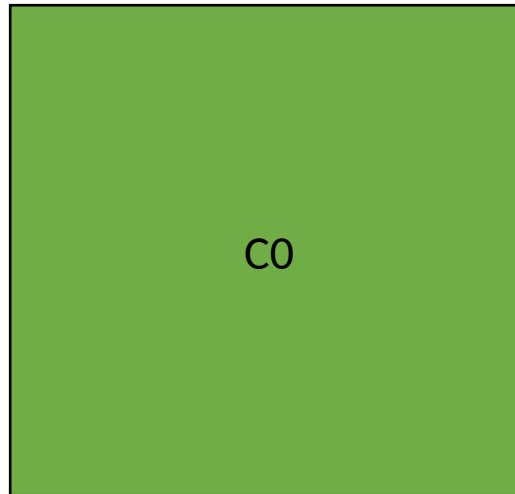
Thread 0

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

C0

Core

Core

Sometimes multiple programs want to share the same core.
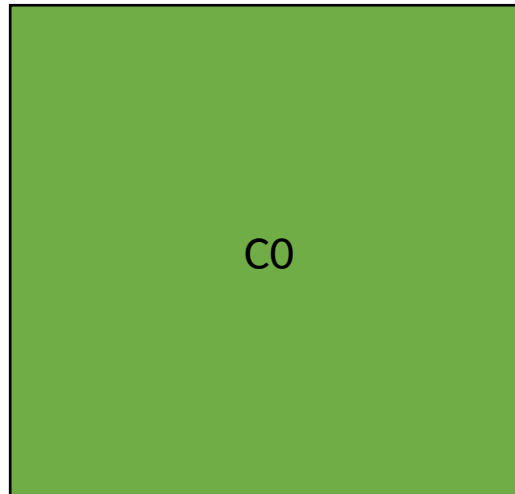
Compiled function #0

```
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

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

X

C0

Core

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

Core

Sometimes multiple programs want to share the same core.

This is called concurrency: multiple threads taking turns executing on the same hardware resource

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

```
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

C0

Core

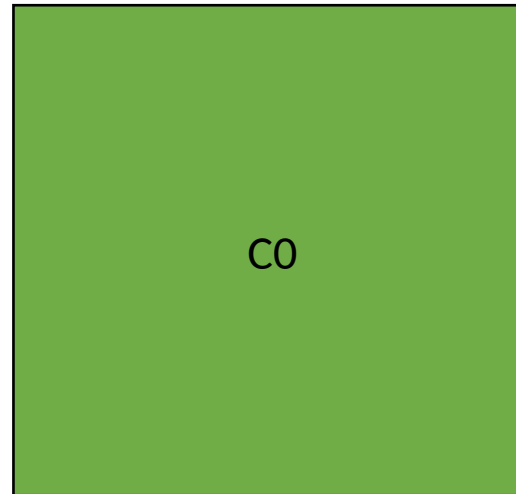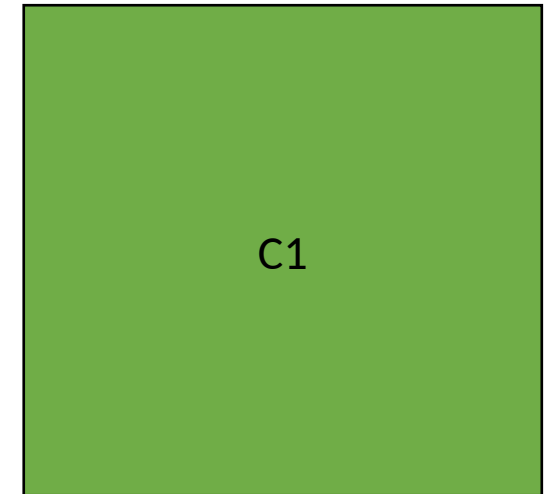And place another thread to execute

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    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
```
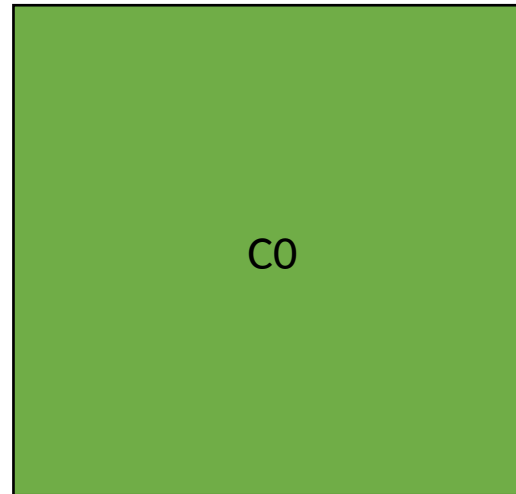
Thread 1

Compiled function #0

```
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
```
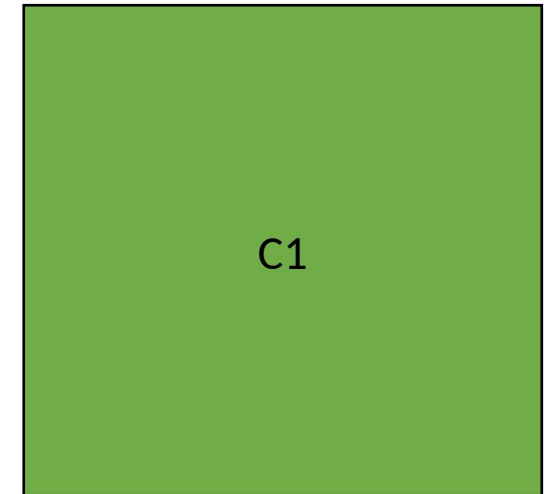
Thread 0

C0

Core

And place another thread to execute

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

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
```
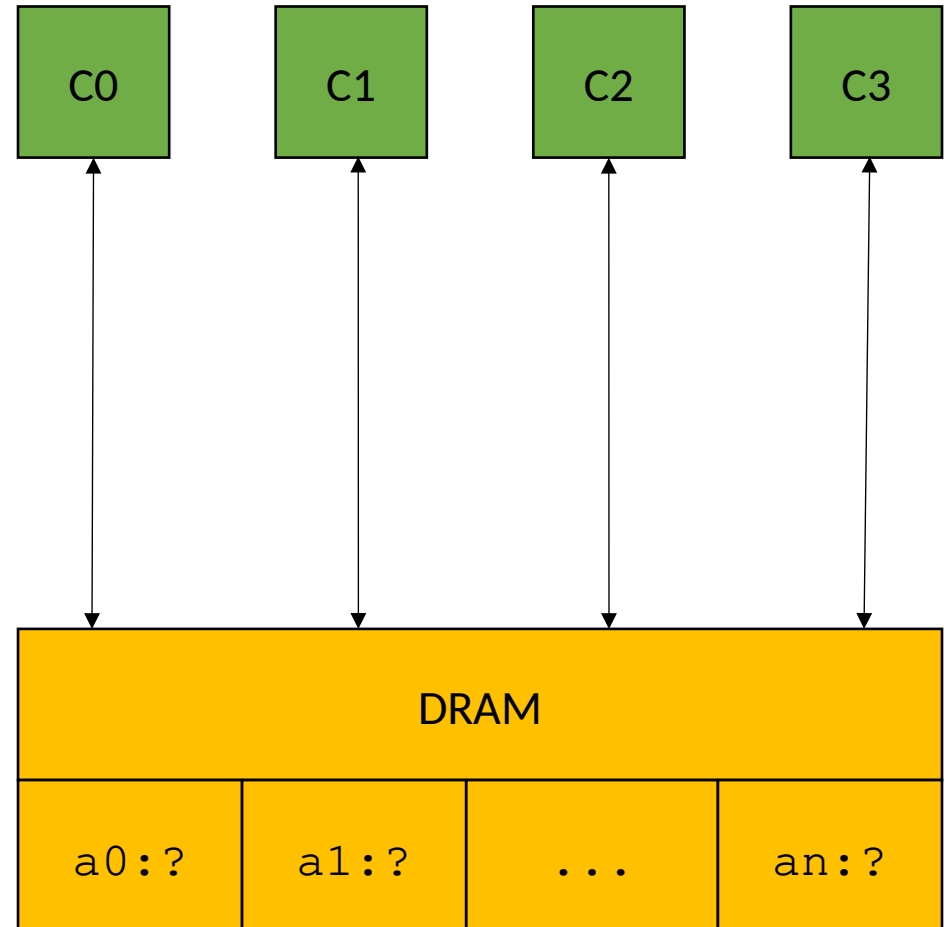
Thread 1

Threads can execute simultaneously (at the same time) if there enough resources.

This is also concurrency. But when they execute at the same time, its called: parallelism.

C0

Core

C1

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

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

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

C0

Core

C1

Core

Main memory

`store(a0,128)`
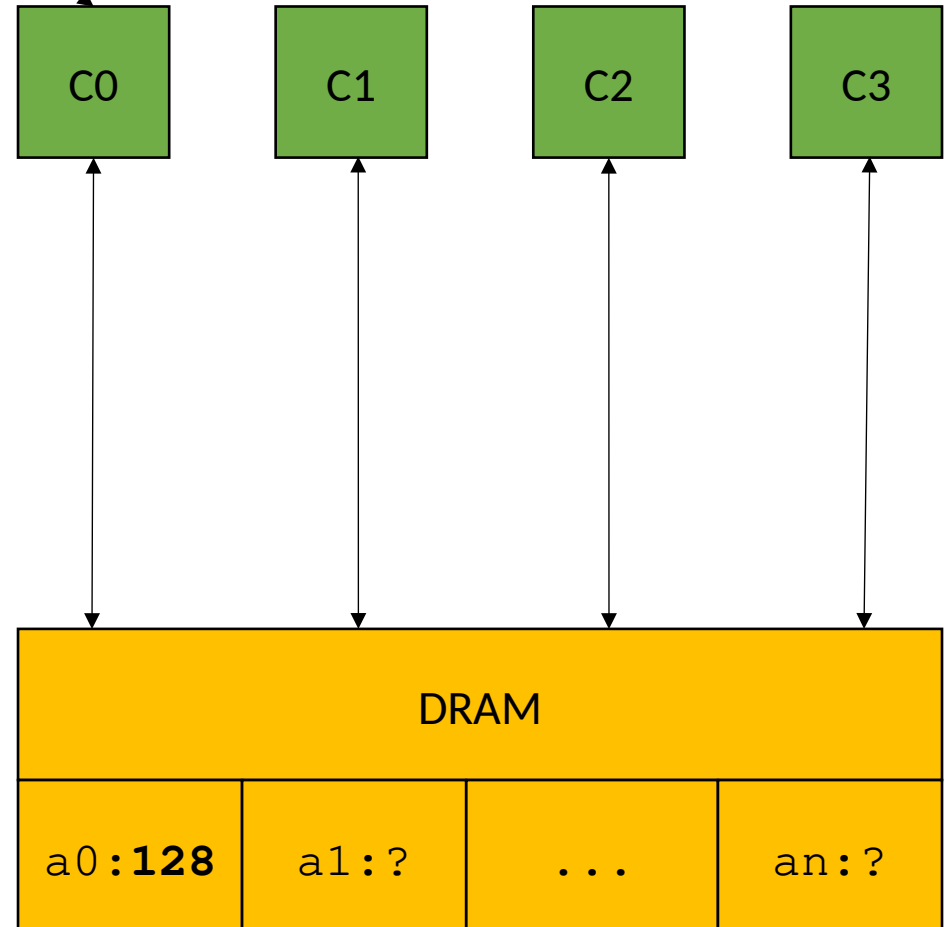
Main memory
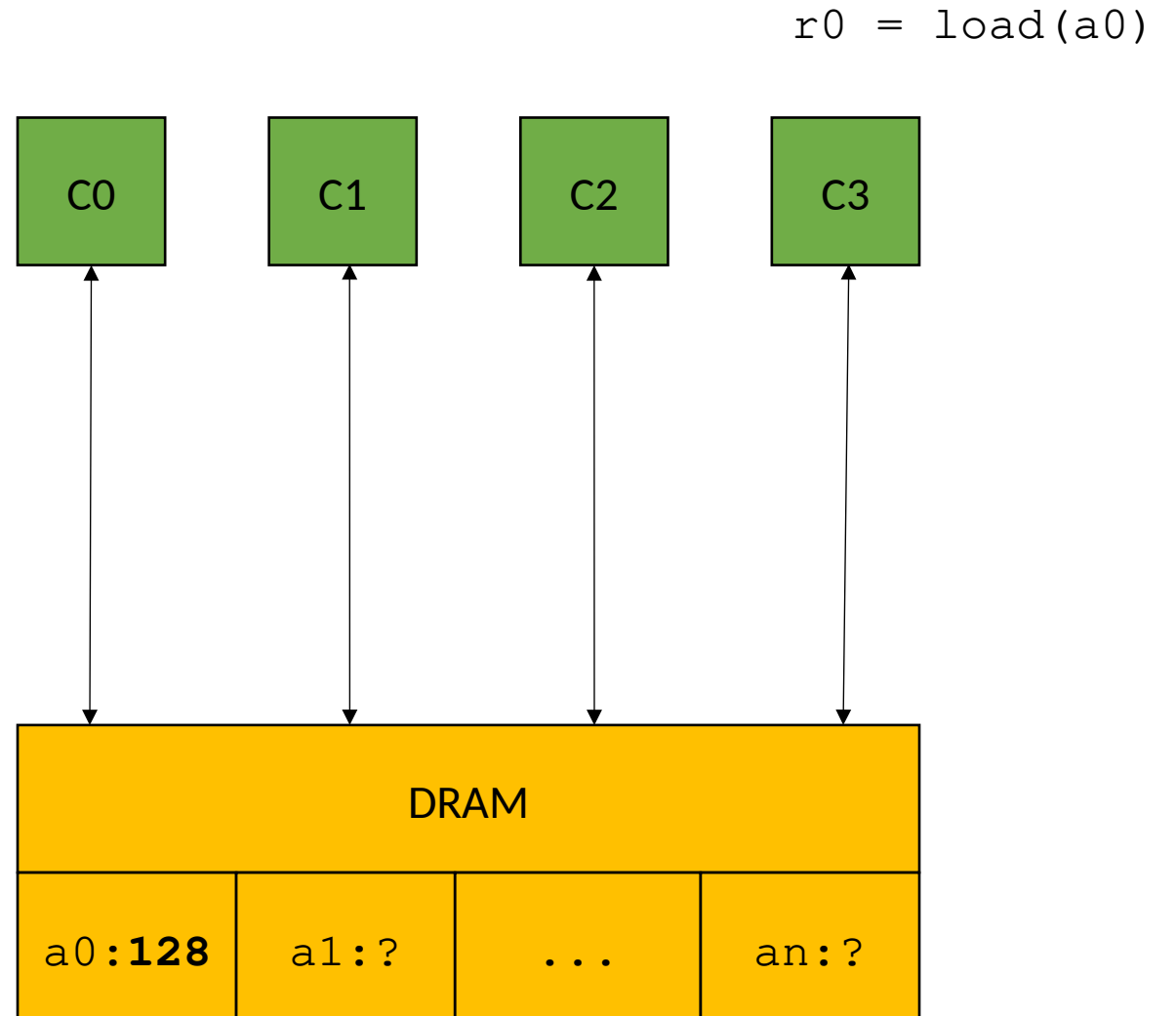
Main memory
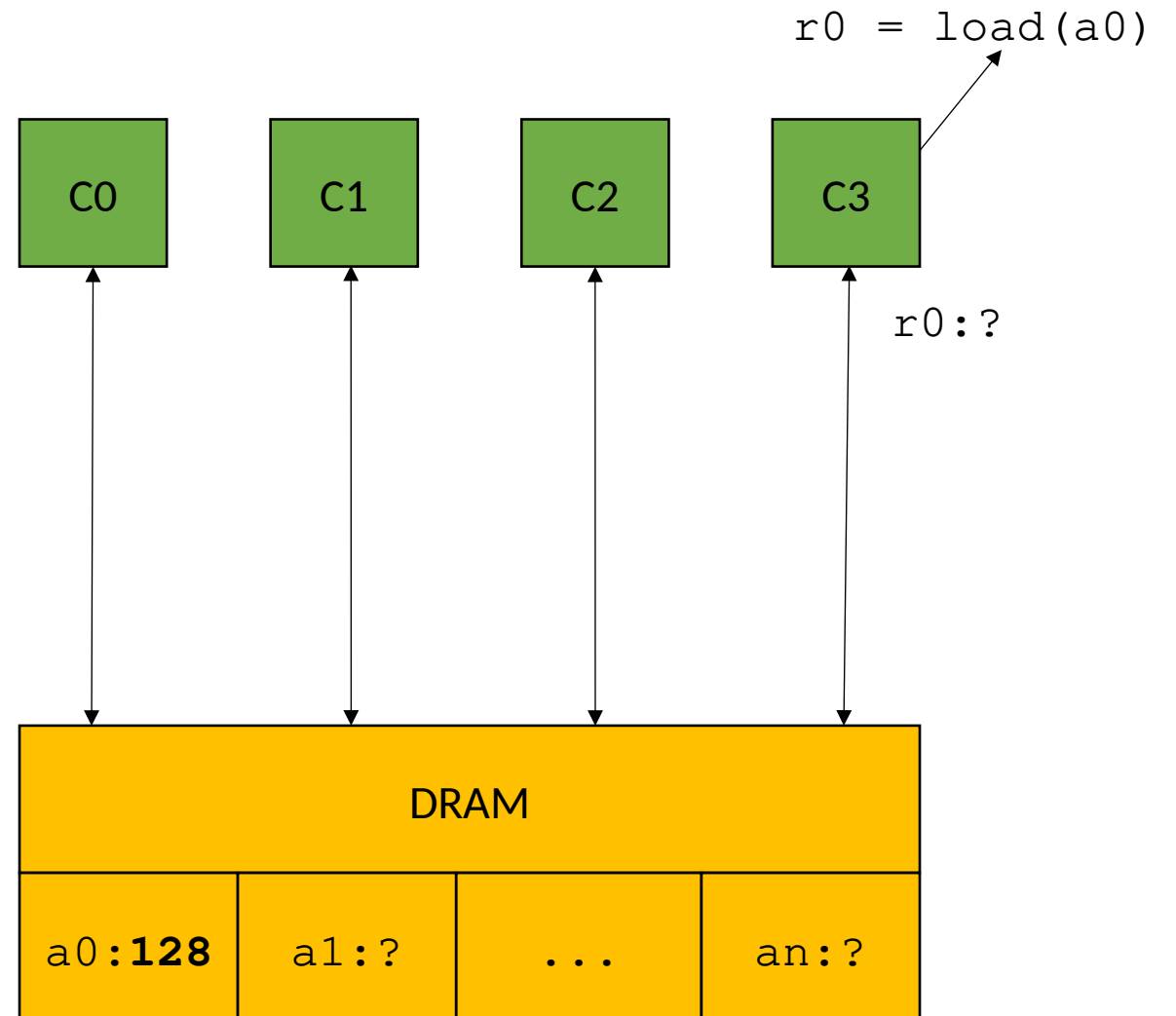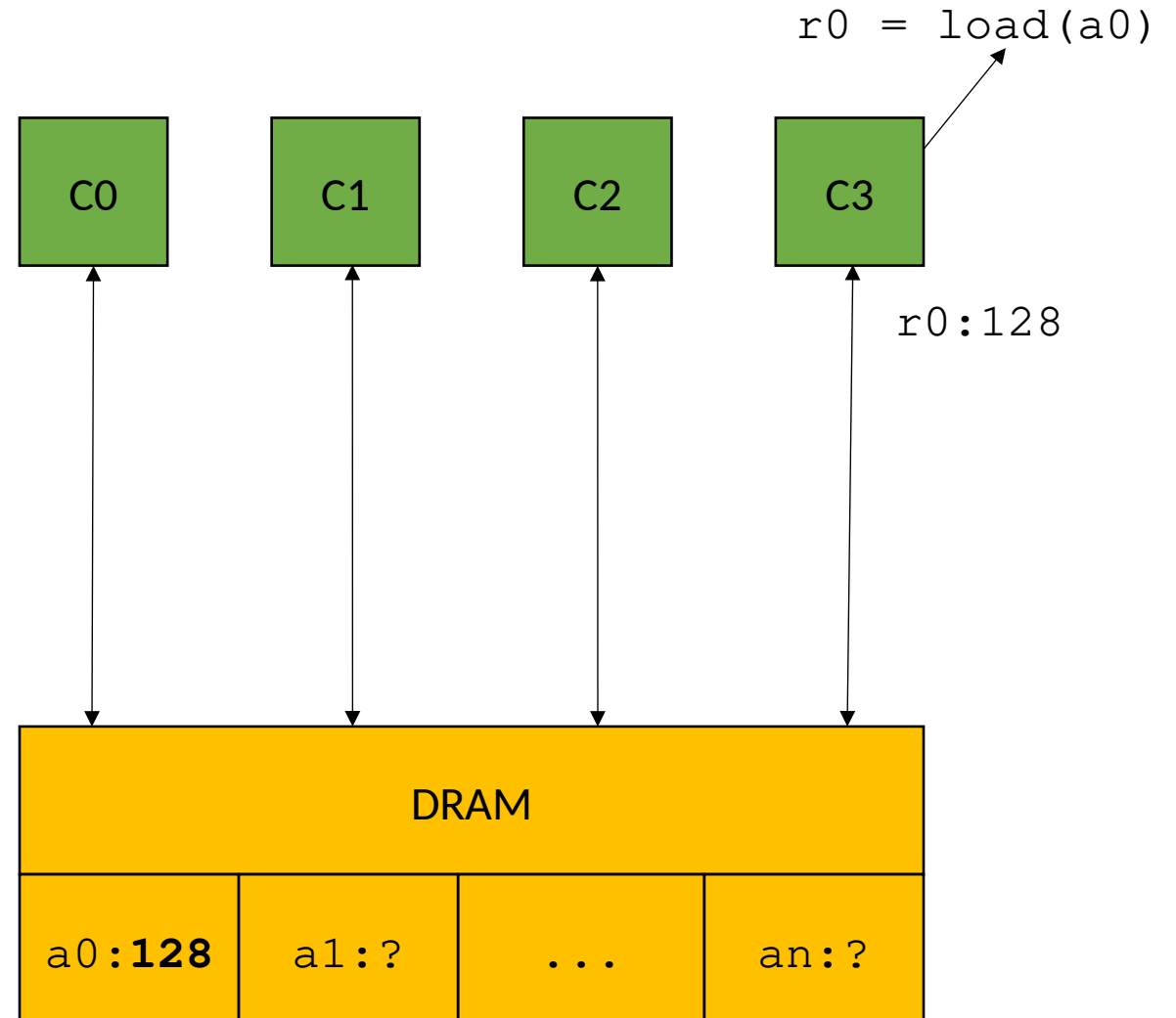
`r0 = load(a0)`

Main memory

r0 = load(a0)

C0   C1   C2   C3

r0:?

DRAM

| a0:**128** | a1:? | ... | an:? |

Main memory

Problem solved!
Threads can communicate!

r0 = load(a0)

C0   C1   C2   C3

r0:128

DRAM

| a0:**128** | a1:? | ... | an:? |

Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

```
r0 = load(a0)
```

C0	C1	C2	C3

`r0:128`

DRAM			
a0:**128**	a1:?	...	an:?
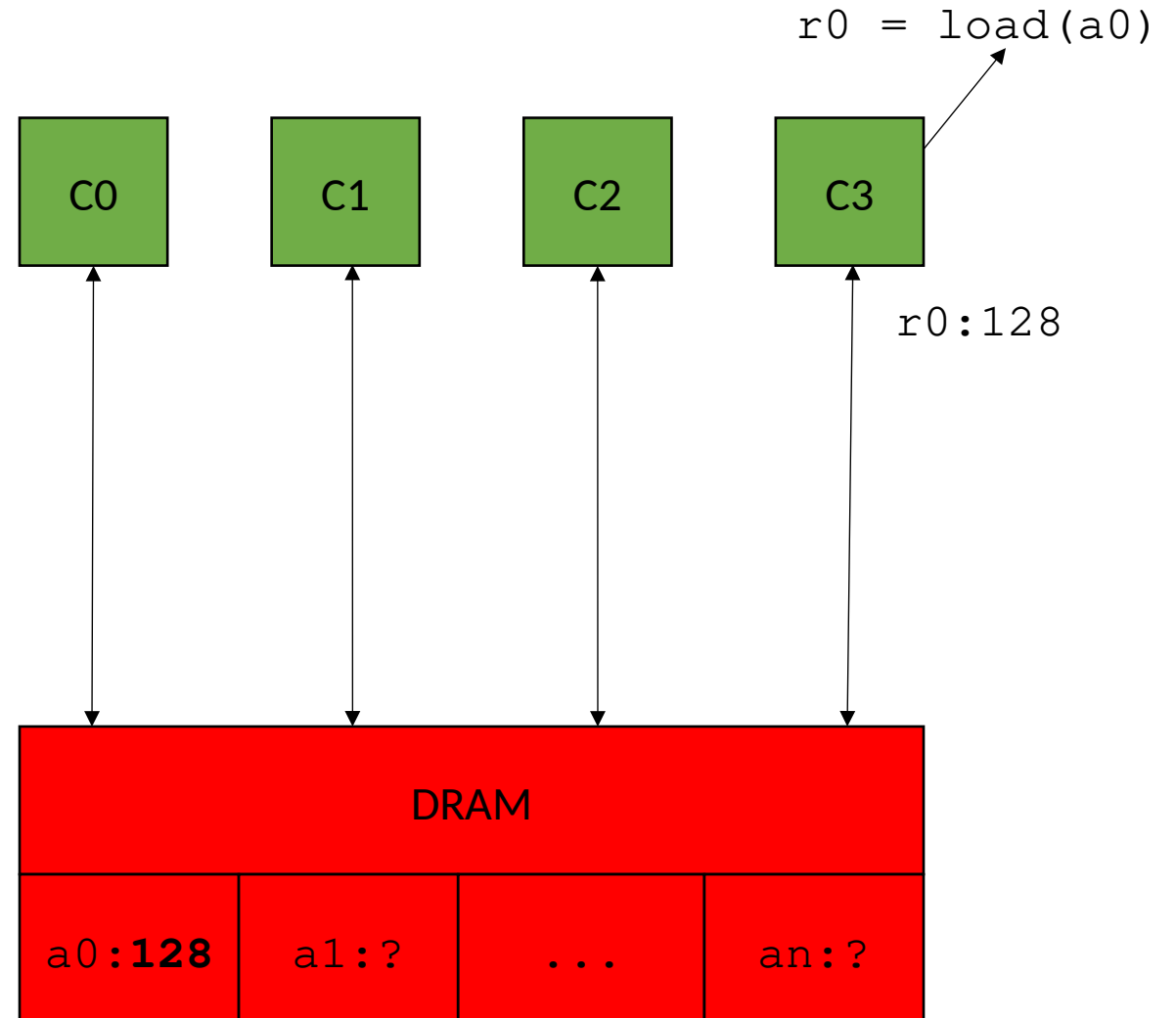
Main memory

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!)

```
r0 = load(a0)
```

| C0 | C1 | C2 | C3 |

`r0:128`

DRAM			
a0:**128**	a1:?	...	an:?

Main memory

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

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

C0

DRAM

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

C0

DRAM

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

C0

DRAM

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

C0

DRAM

Main memory

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

%5 = load i32, i32* %4   200 cycles

                                1 cycles

%6 = add nsw i32 %5, 1   200 cycles
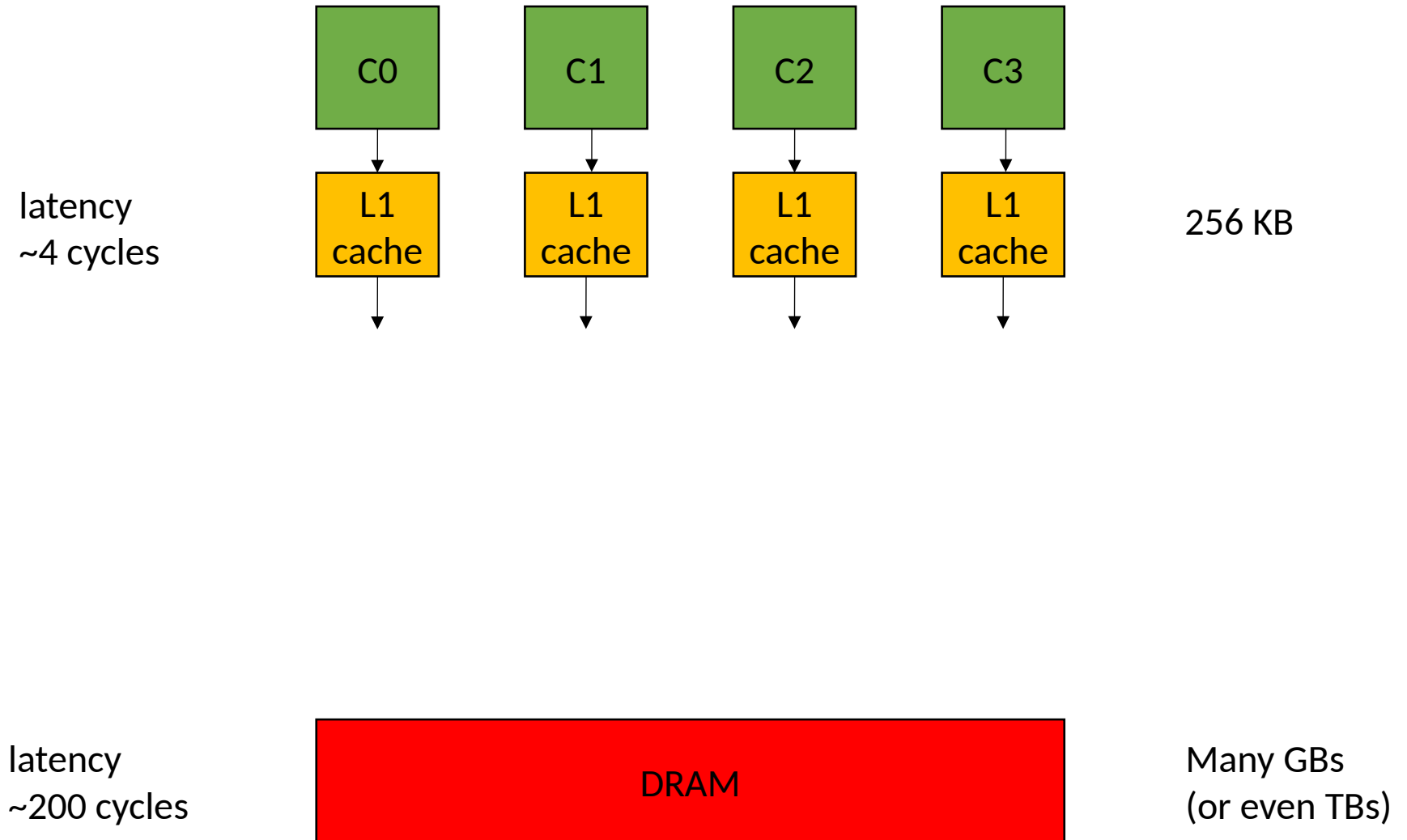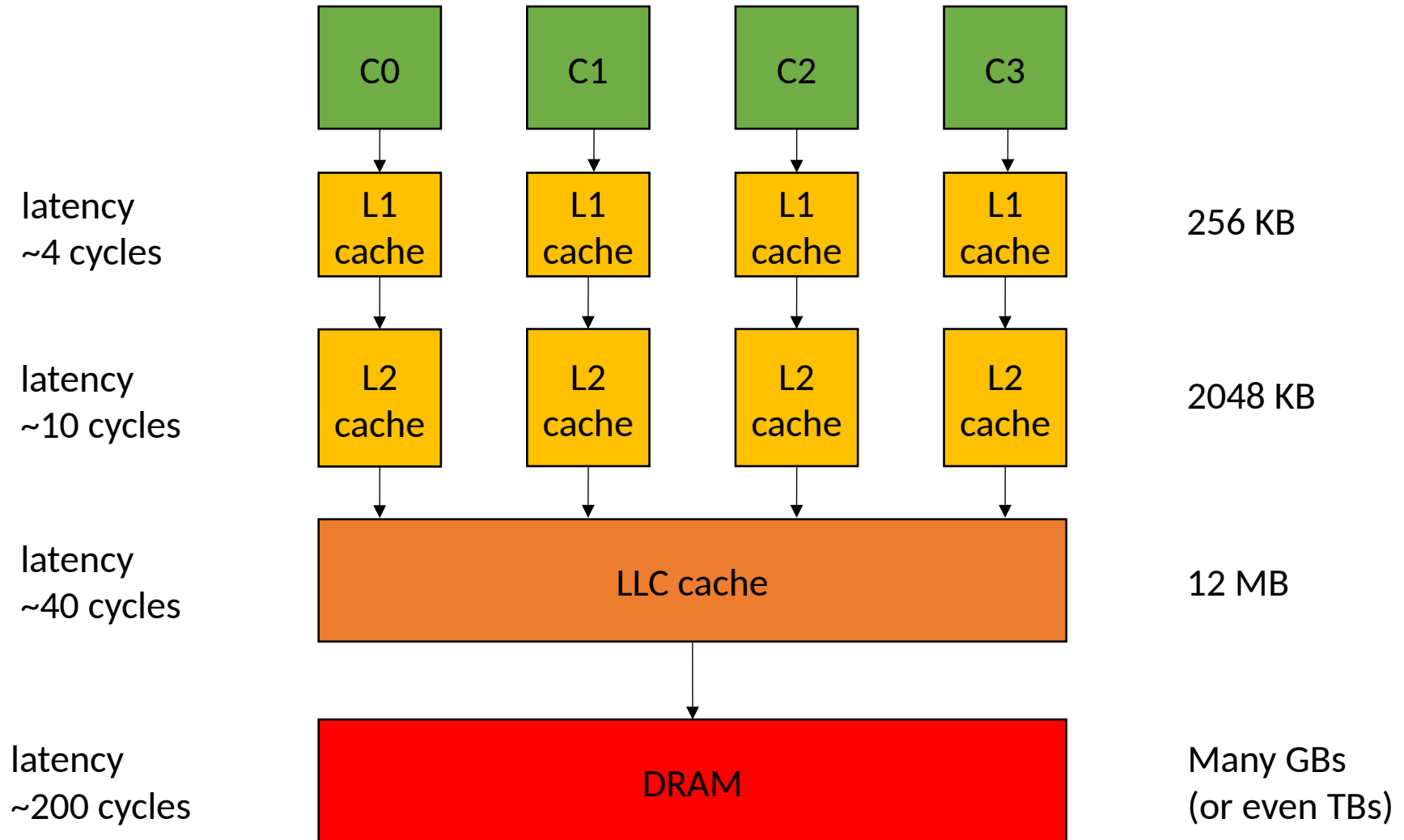
store i32 %6, i32* %4   **401 cycles**

C0

DRAM

Main memory

```
int increment(int *a) {
    a[0]++;
}
```
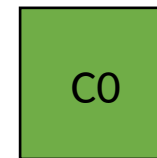
```
int x = 0;
for (int i = 0; i < 100; i++) {
    increment(&x);
}
```

```
%5 = load i32, i32* %4        200 cycles
                               1 cycles
%6 = add nsw i32 %5, 1         200 cycles
store i32 %6, i32* %4          401 cycles
```

C0

DRAM

Main memory

```
int increment(int *a) {
    a[0]++;
}
```
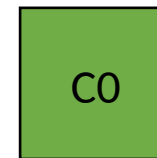
```
int x = 0;
for (int i = 0; i < 100; i++) {
    increment(&x);
}
```

40100 cycles!

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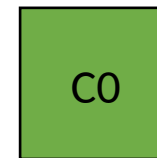
200 cycles

1 cycles

200 cycles

401 cycles

C0

DRAM

Caches

C0 C1 C2 C3

latency
~200 cycles

DRAM

Many GBs
(or even TBs)

Caches

latency
~4 cycles

C0   C1   C2   C3

L1 cache   L1 cache   L1 cache   L1 cache

256 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
~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
```
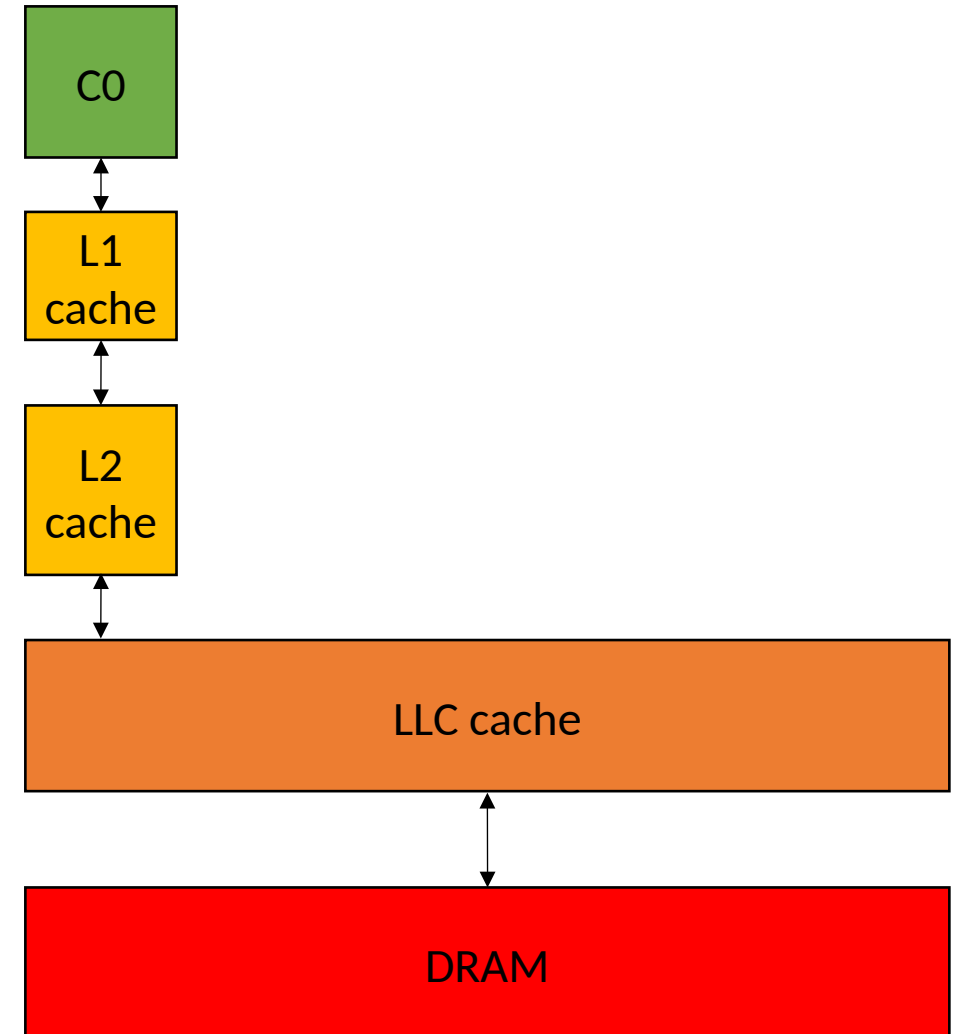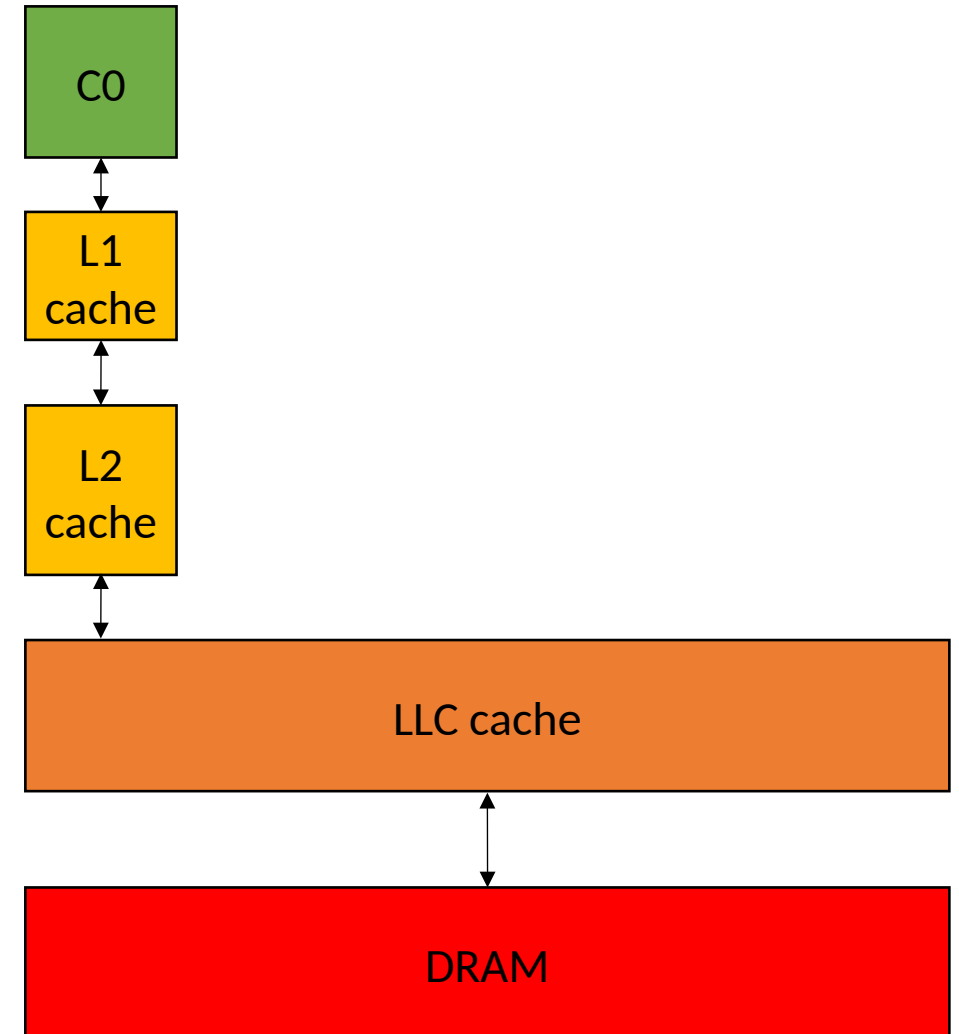
Caches

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

```
%5 = load i32, i32* %4          4 cycles
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```
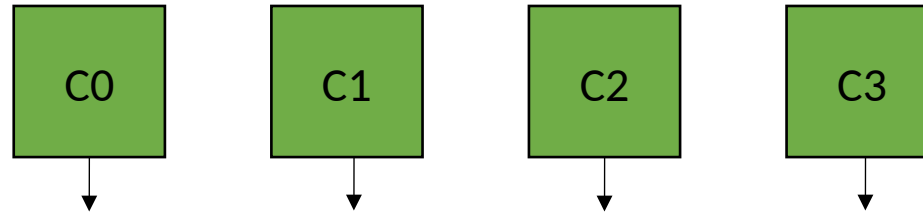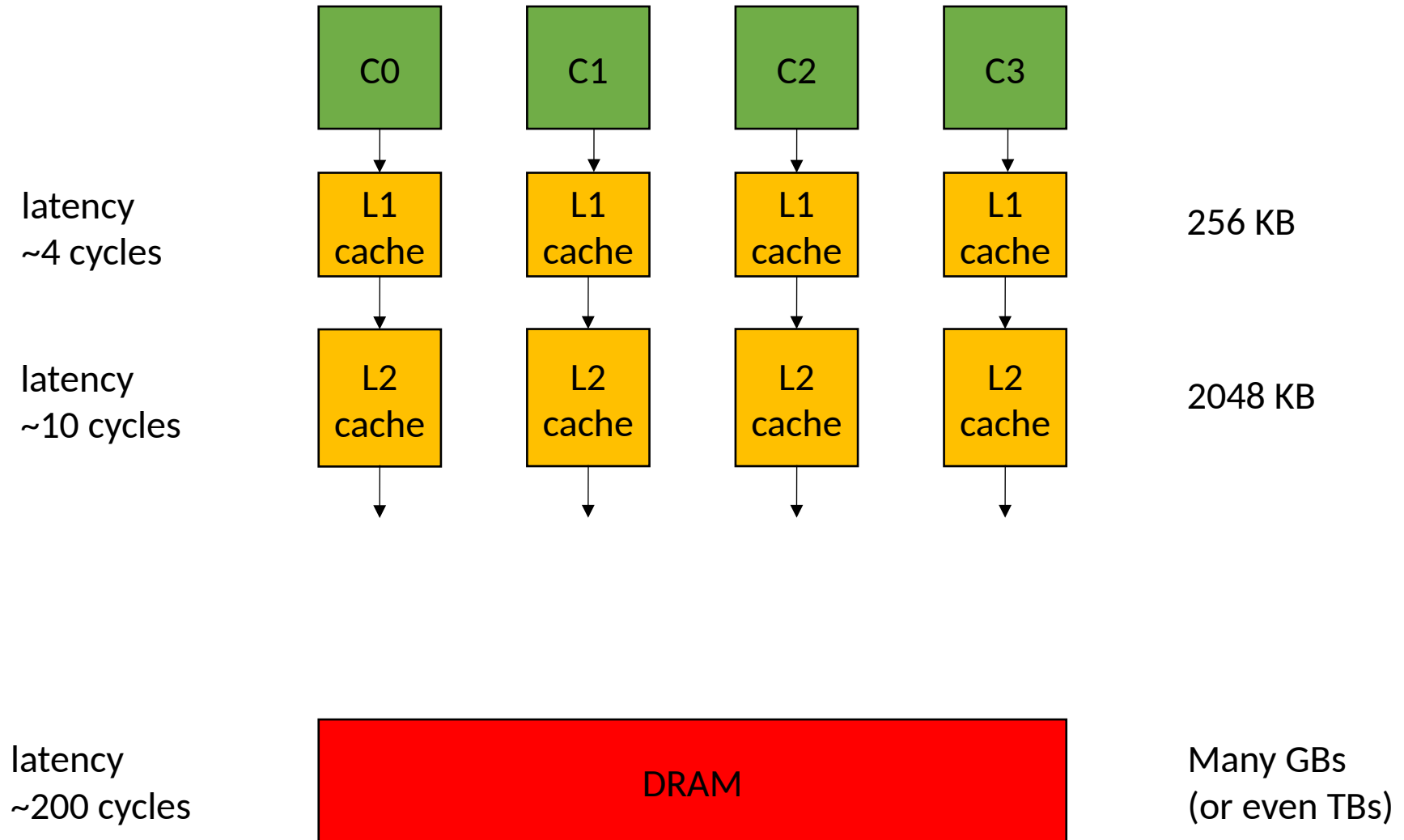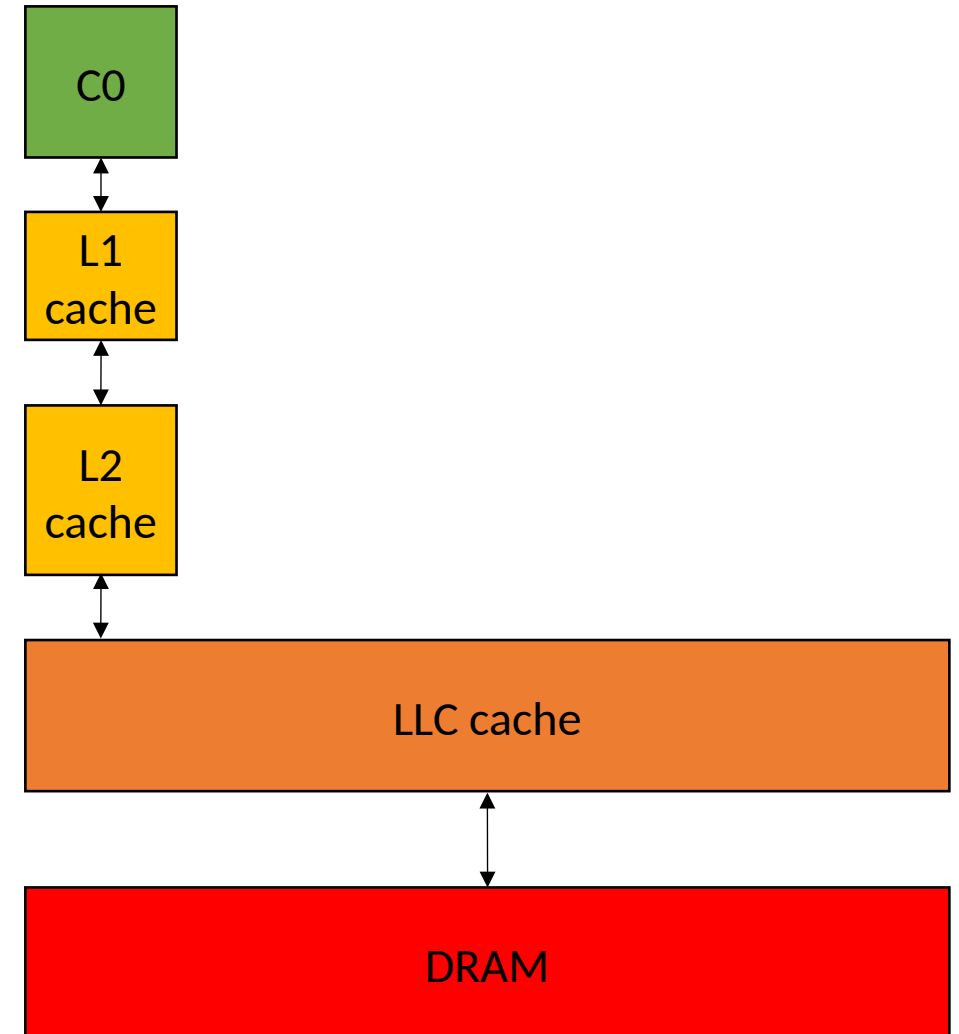
Assuming the value is in the cache!

C0

L1 cache

L2 cache

LLC cache

DRAM

Caches

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

```
%5 = load i32, i32*        4 cycles
%4                          1 cycles
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```
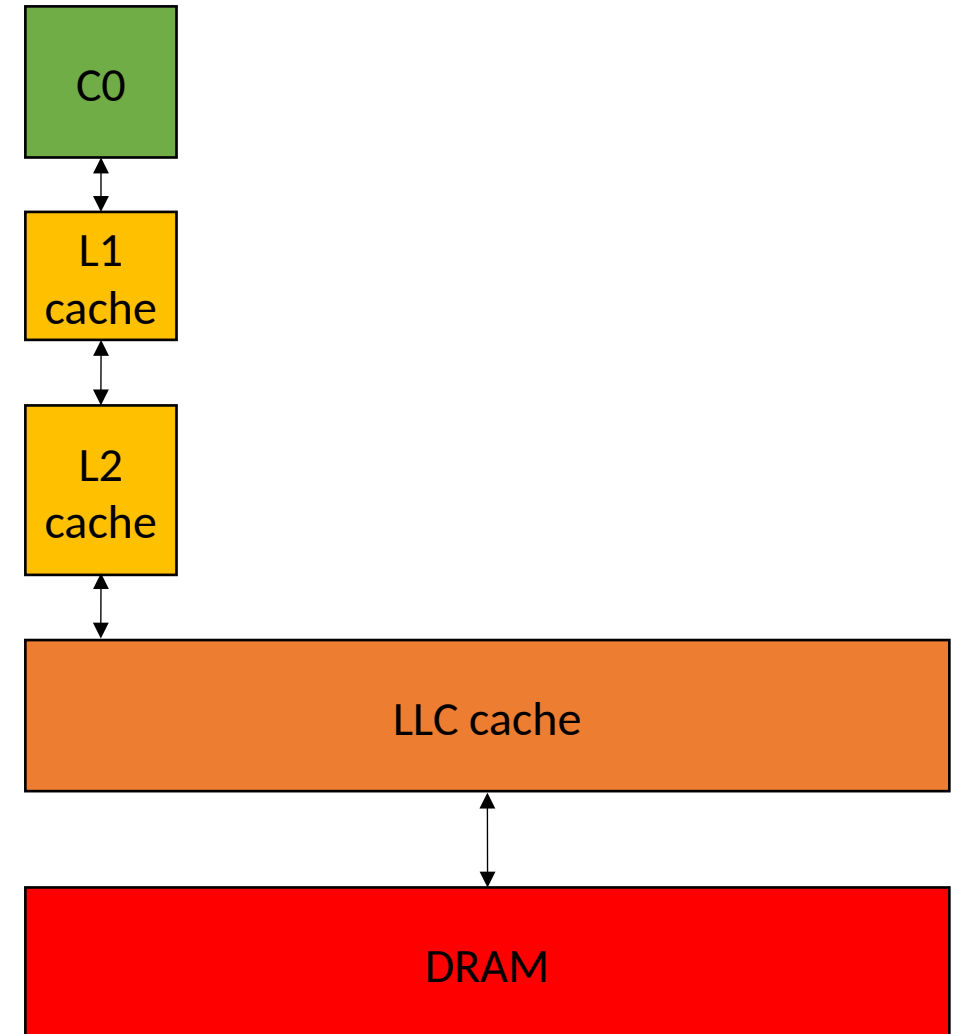
C0

L1 cache

L2 cache

LLC cache

DRAM

Caches

```
int increment(int *a) {
    a[0]++;
}
```
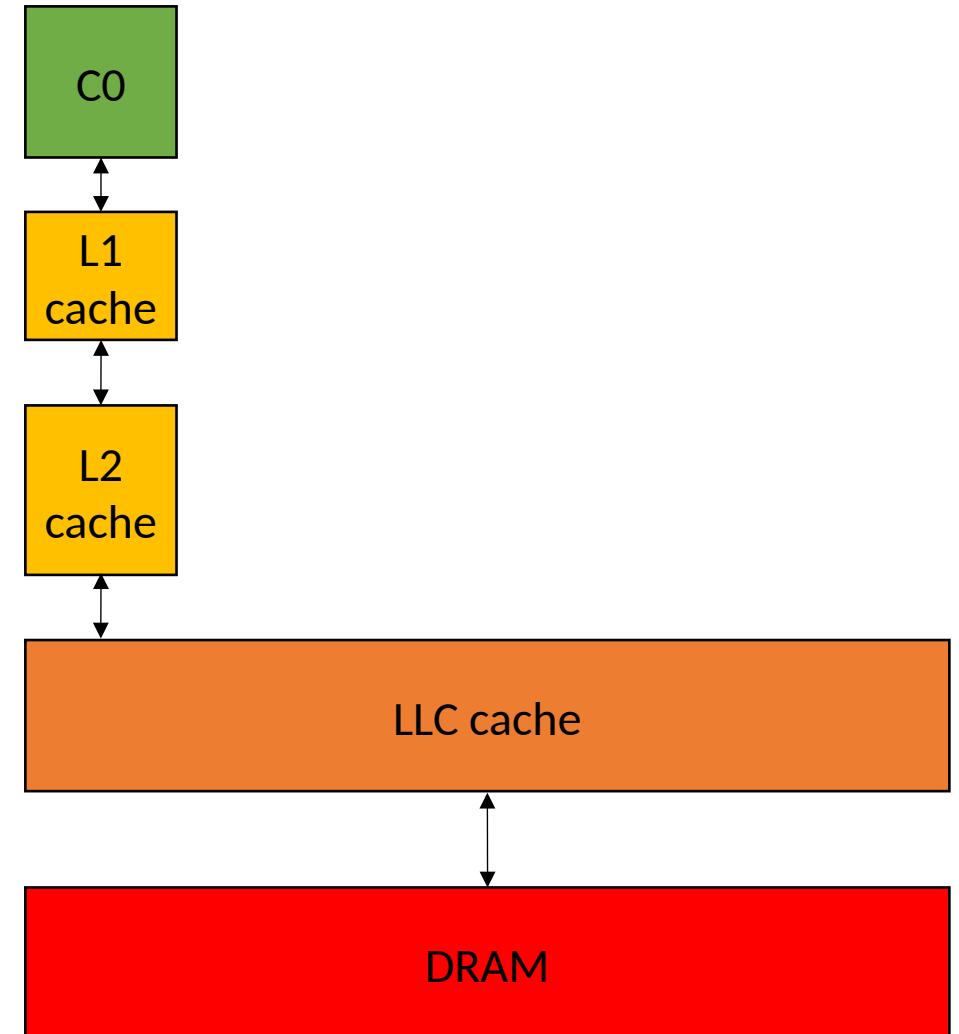
```
%5 = load i32, i32*        4 cycles
%4                         1 cycles
%6 = add nsw i32 %5,       4 cycles
1
store i32 %6, i32* %4
```

Caches

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

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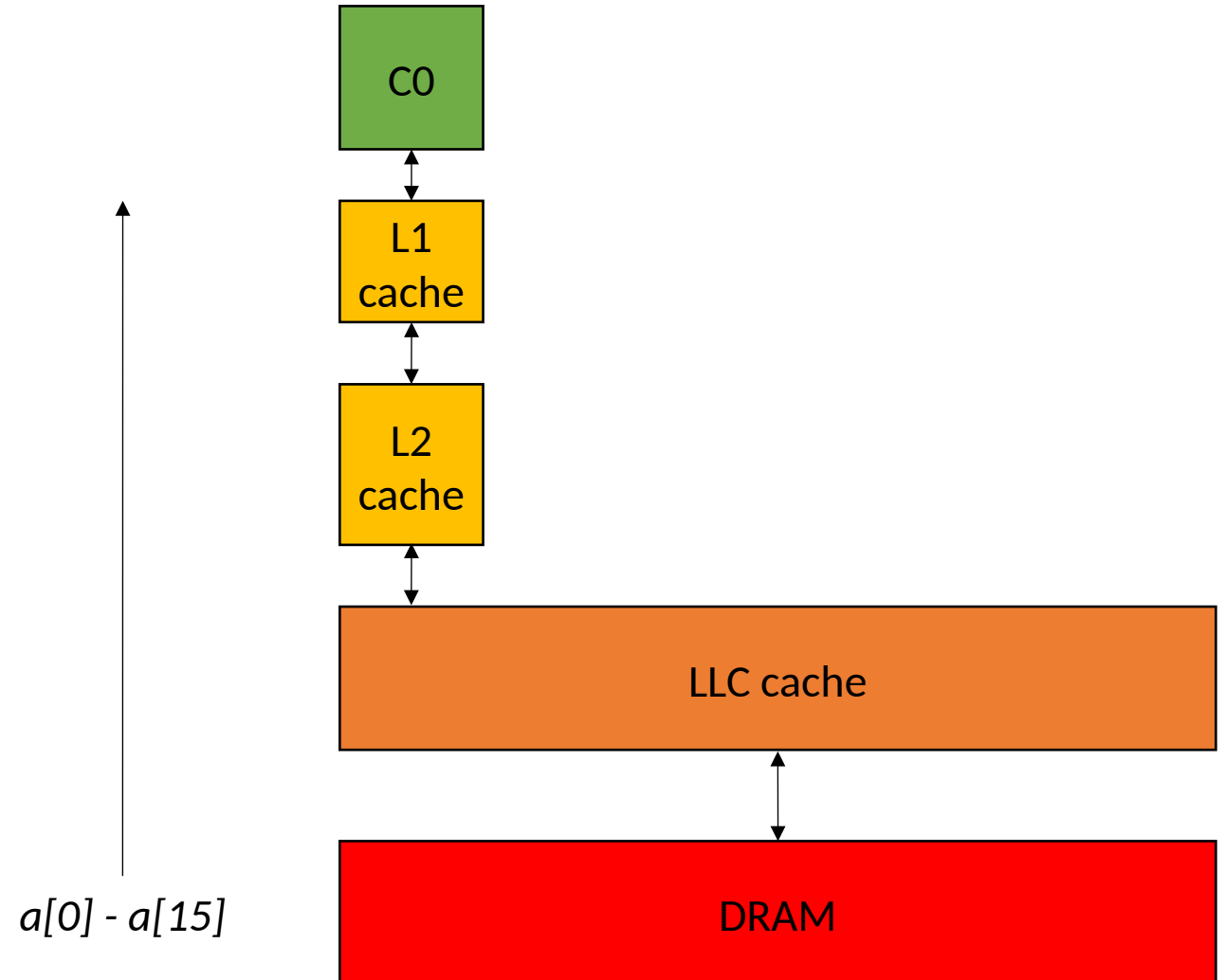
Caches

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

```
%5 = load i32, i32* %4
%6 = add nsw i32 %5, 1
store i32 %6, i32* %4
```
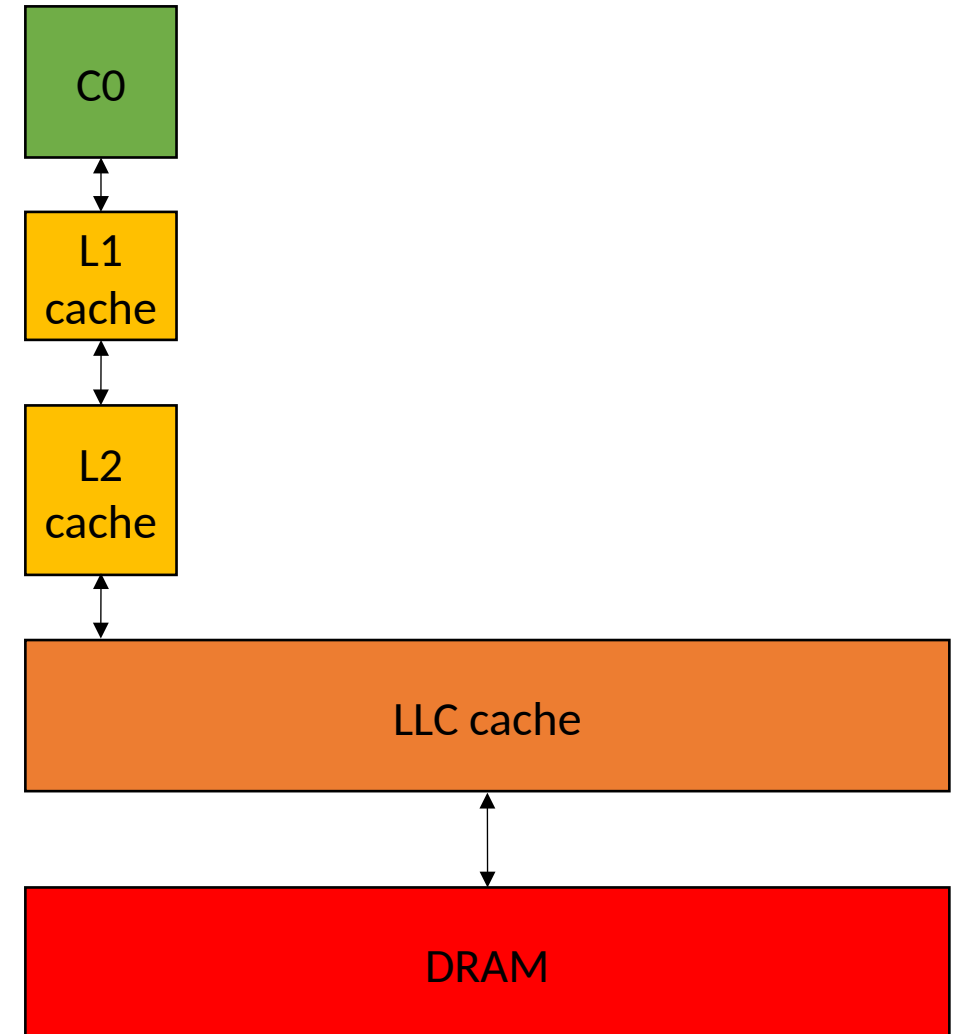
C0

L1 cache

L2 cache

LLC cache

DRAM

Caches

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

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

C0

L1 cache

L2 cache

LLC cache

DRAM

a[0] - a[15]

Caches

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

C0

L1 cache

L2 cache

LLC cache

DRAM

Caches

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

C0

L1 cache

L2 cache

LLC cache

DRAM
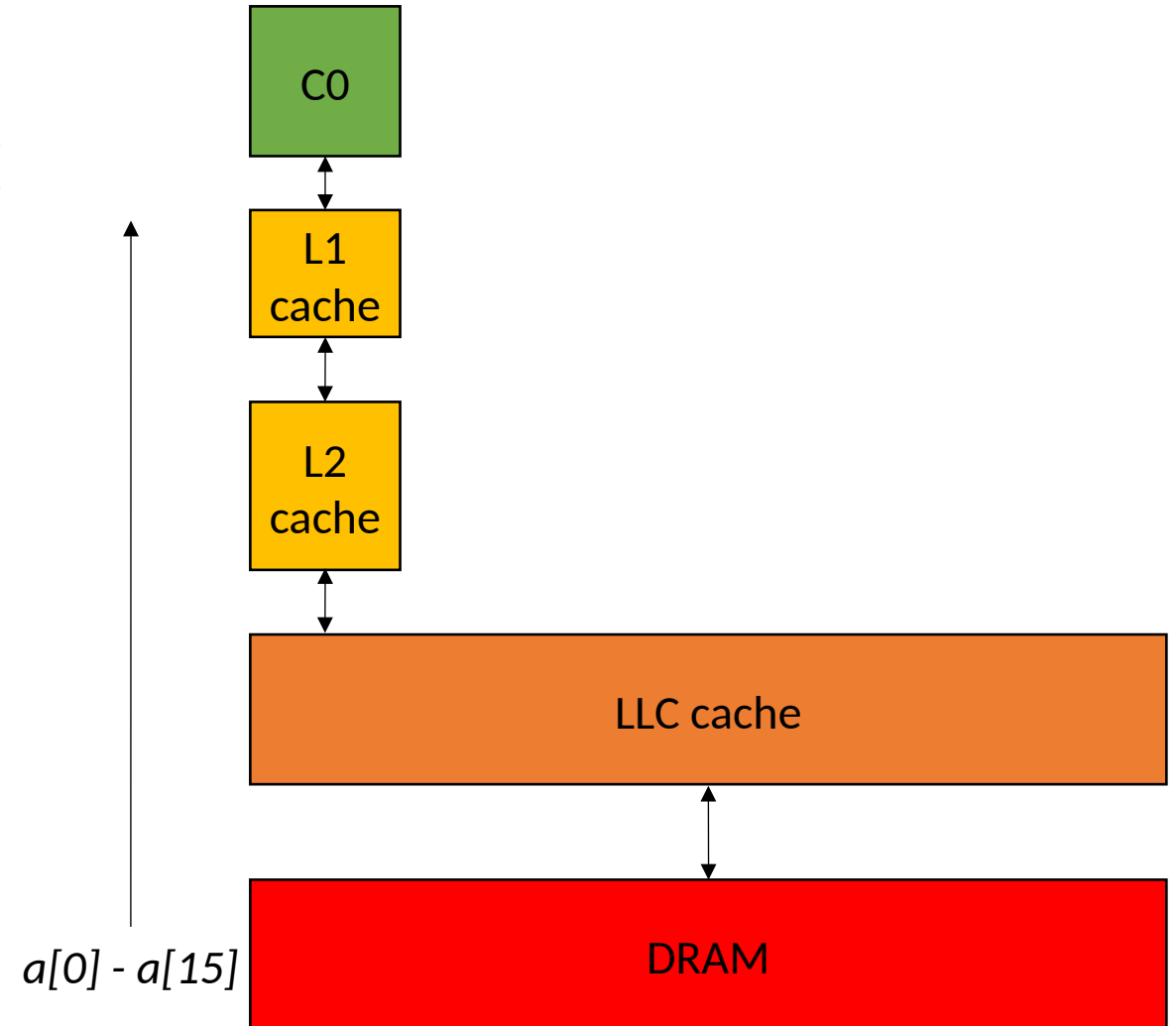
a[0] - a[15]

Caches

```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```
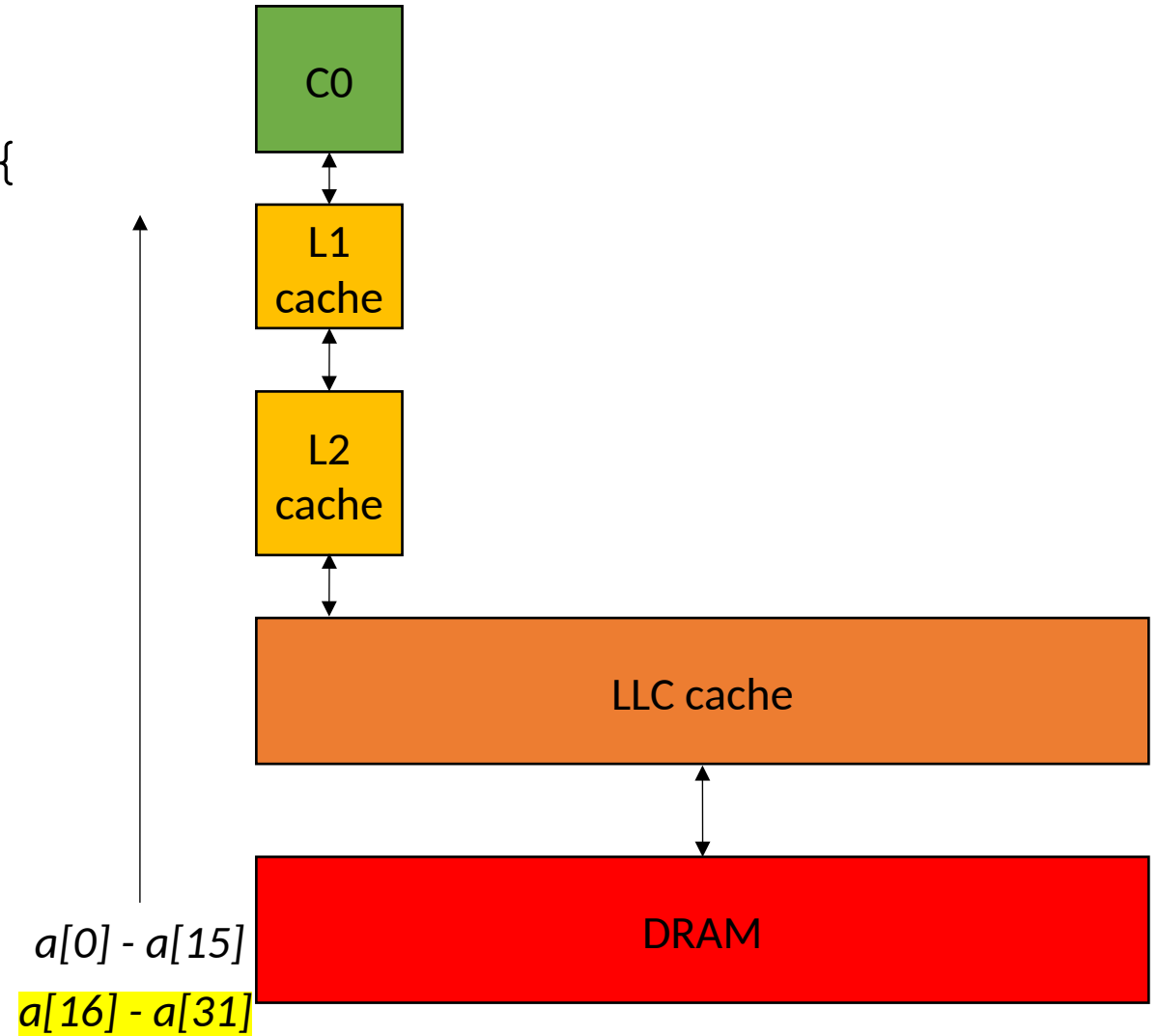
will be a hit because we've loaded a[0] cache line

C0

L1 cache

L2 cache

LLC cache

DRAM

a[0] - a[15]

Caches

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

Miss

C0

L1 cache

L2 cache

LLC cache

DRAM

a[0] - a[15]

a[16] - a[31]

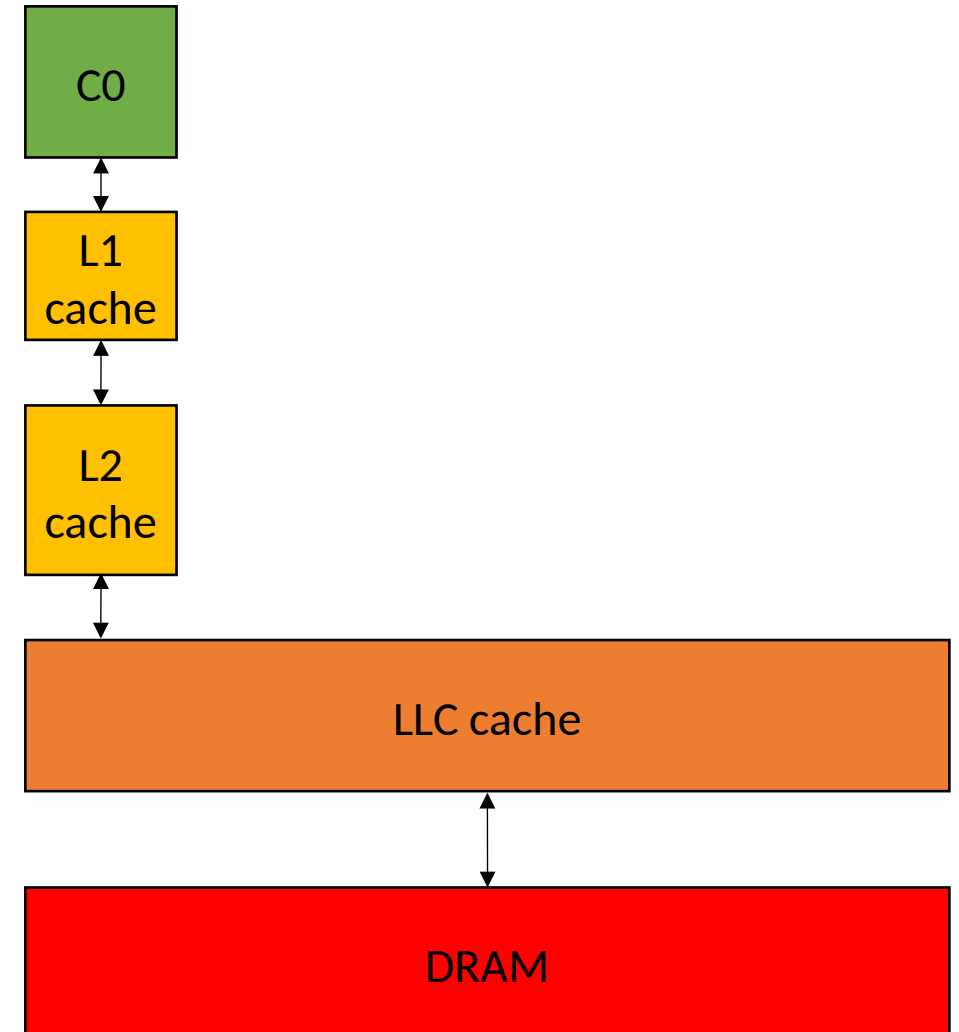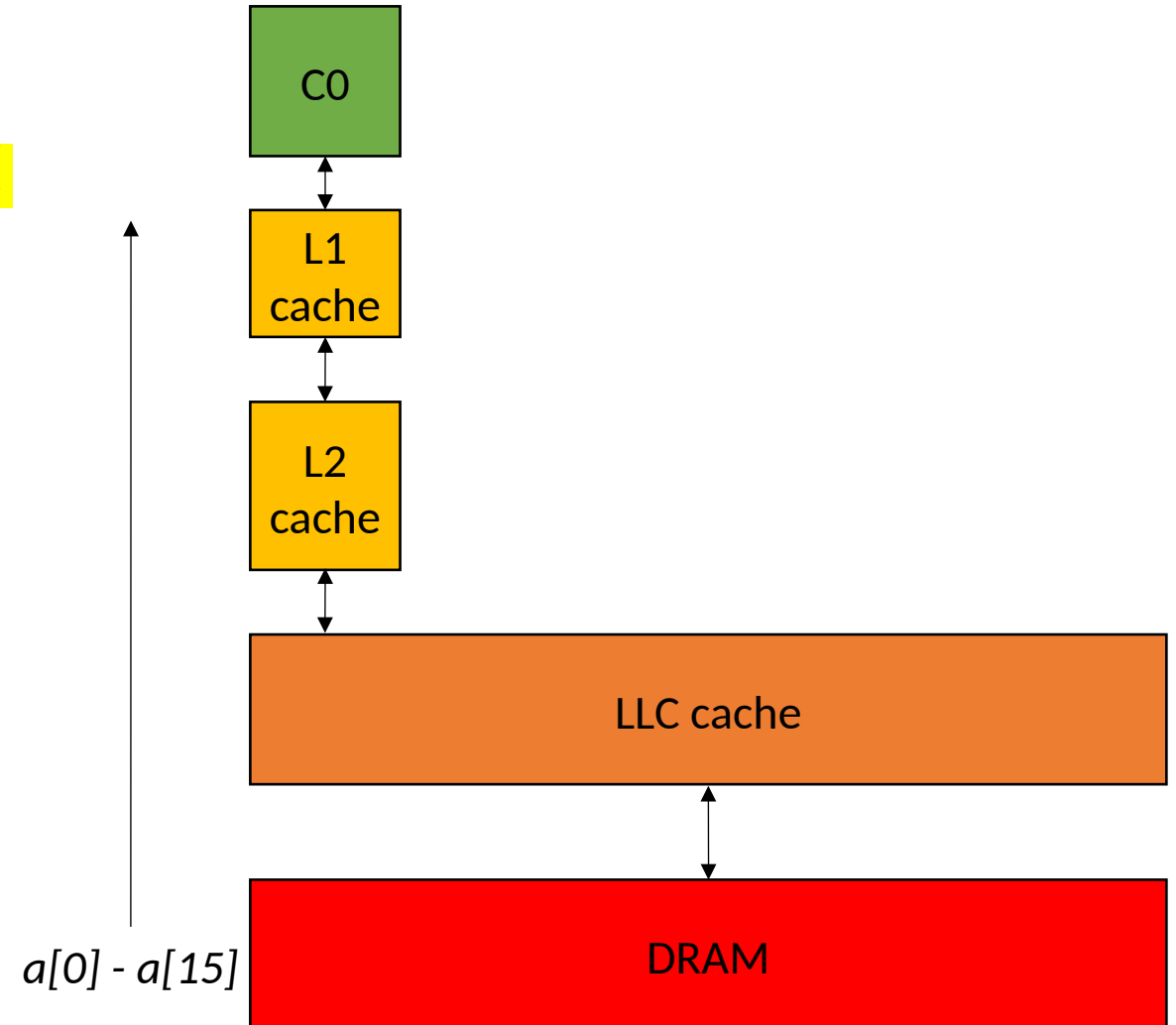# Cache alignment

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

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


C0

L1 cache

L2 cache

LLC cache

DRAM

Cache alignment

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

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

C0

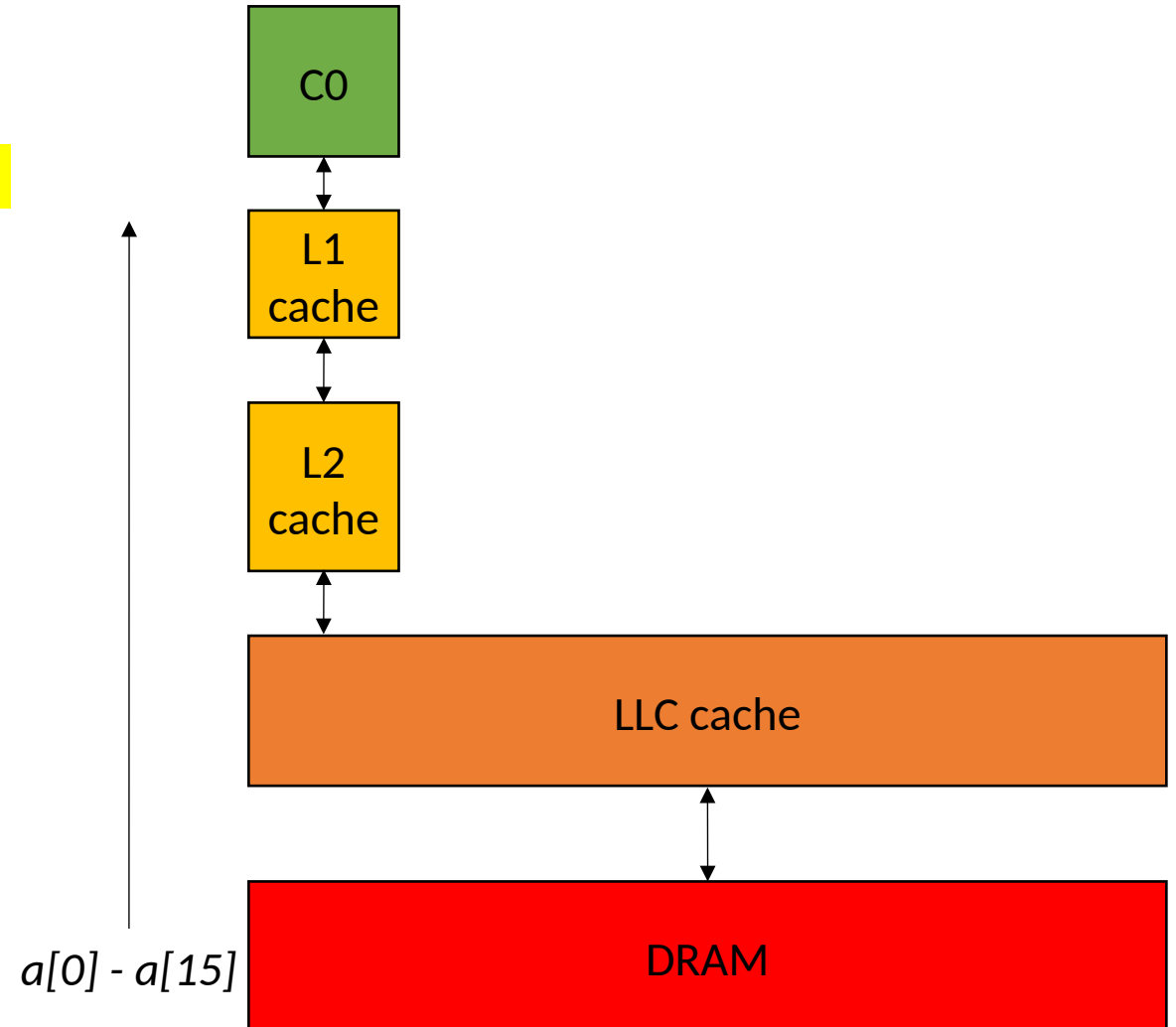L1 cache

L2 cache

LLC cache

DRAM

a[0] - a[15]

Cache alignment

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


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

This loads a[8]

C0

L1 cache

L2 cache

LLC cache

DRAM

a[0] - a[15]

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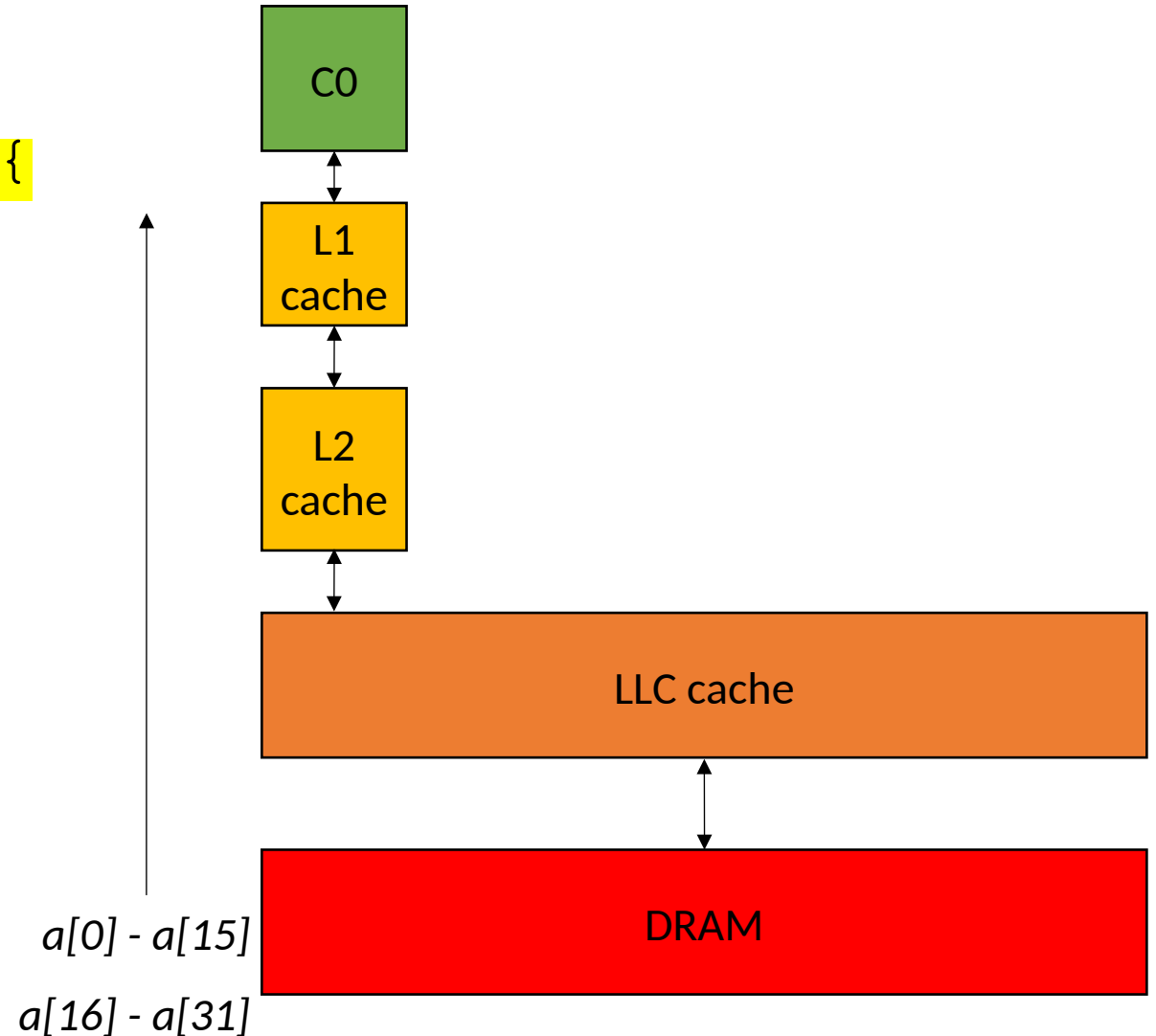
```
int increment_several(int *b) {
    b[0]++;
    b[15]++;
}


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

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

C0

L1 cache

L2 cache

LLC cache

a[0] - a[15]

DRAM

a[16] - a[31]

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.