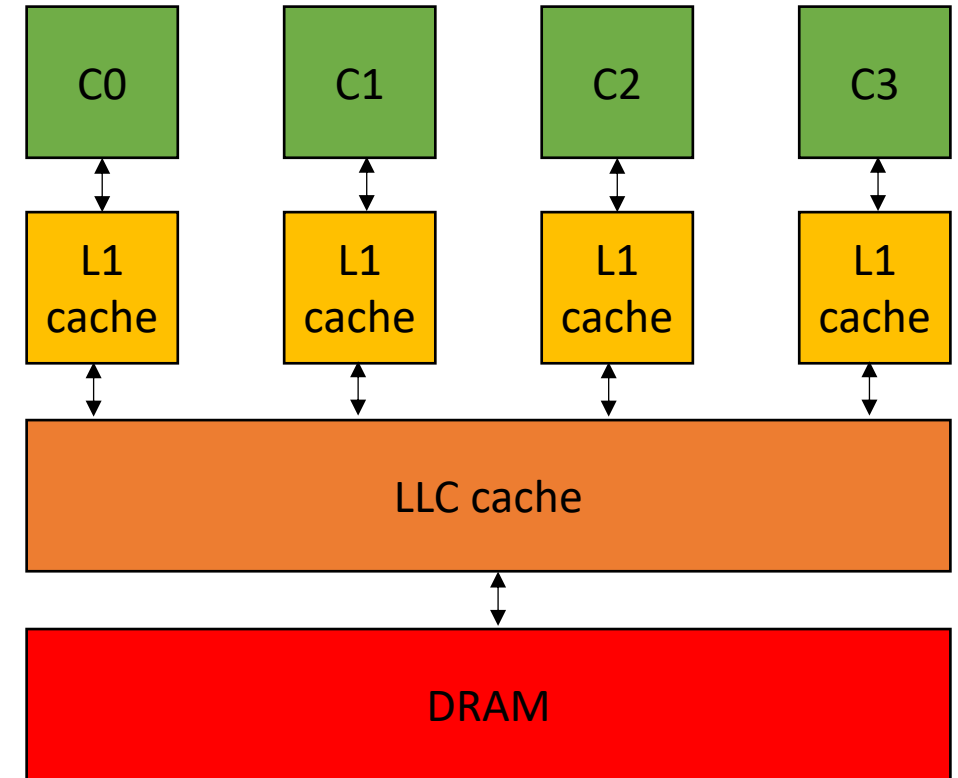


CSE113: Parallel Programming

Jan. 5, 2022

- **Topic:** Architecture and Compiler Overview

- Programming Language to ISA compilation
- 3-address code
- multiprocessors
- memory hierarchy



Announcements

- Still remote
 - Due to the late night storm and early class
 - Plan on Friday in person (I'll let you know by Thursday afternoon if not)
 - I hope everyone is safe!

Asynchronous Forums

- Piazza is setup, I sent out an invite link on canvas.
 - Please join! There will be important announcements there that you will be responsible for
 - 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

- **Tyler:**

- Thursday from 3:00 - 5:00 PM
- Hybrid (remote or in person)
- Room E2 233
- I'll have mine tomorrow. Feel free to swing by and say hi!

- We will have everyone else's hours posted soon

- Monday at the latest

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.

Quiz

- I hope everyone got the quiz for last time done; it was great reading all of your responses!
- There is a quiz for today, due by the start of next lecture, please do it!









Getting to know your classmates

What year are you in your studies?

Jr.	13 respondents	13 %	
Sr.	88 respondents	86 %	
Grad Student	1 respondent	1 %	

Getting to know your classmates

Which of the following programming languages/frameworks do you have experience with?

Python	90 respondents	88 %	 ✓
C	96 respondents	94 %	
C++	87 respondents	85 %	
JavaScript	69 respondents	68 %	
GPU Programming	8 respondents	8 %	
Docker	34 respondents	33 %	
Unix command line	88 respondents	86 %	
console text editor (e.g. vim, emacs)	71 respondents	70 %	

Previous experience

- Previous experience with concurrency:
 - Some in CSE 130
 - *You're going to love C++ threads!*
 - we're going to discuss implementations of primitives, e.g. locks
- Most had no experience!
 - will be helpful for when you take CSE 130

Getting to know your classmates

- We have a great group!
- Hobbies
 - Cycling, surfing, cocktails, music, cooking and more!
- CS topics people are interested in:
 - AI
 - Video games
 - Edge computing
 - Still exploring...

parallel programming applies to lots of these
CS interests!

Some examples

Self driving cars:

- Requires a reaction speed of 1.6s
- How to make faster?
 - Algorithms
 - interconnects
 - **cores**



Nvidia's embedded device has increased from 256, to 384 to 512 cores

Some examples

Just because something is parallel doesn't mean it will go fast!

Some examples

Some things are easy to make fast



pretty straight
forward computation
for brightening

(do every pixel in parallel,
easy to make go fast!)

image processing example

Some examples

Other applications are harder to make go fast

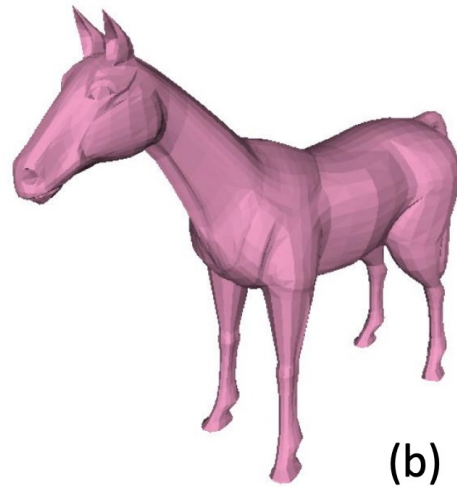
This computation is known as the “Local Laplacian Filter”. Requires visiting all pixels 99 times



*simple parallelism is 2x slower than
finely tuned parallelism*

Some examples

But we need to be careful! Parallel programming is full of tricky corner cases!



parallel programming concepts apply to lots of other hobbies as well!

Cooking

Simple soup recipe

Chop Carrots



Chop Potatoes



Combine and cook



Cooking

How to cook with
one person?



time

Simple soup recipe

Chop Carrots



Chop Potatoes



Combine and cook



Cooking

How to cook with two people?



time

Simple soup recipe

Chop Carrots



Chop Potatoes



Combine and cook



Cooking

How to cook with two people?



time

Simple soup recipe

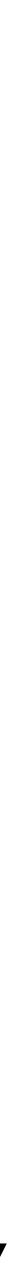
Chop Carrots

Chop Potatoes

Combine and cook



What is the fastest this could possibly go?



To the lecture!

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

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

In a perfect world...

- Programming languages provide an abstraction

Programmer: Writes Code



Hardware Designer: Makes Chips



In a perfect world...

- Programming languages provide an abstraction

*Separation of concerns allows
incredible progress*

Programmer: Writes Code



modern software:
~4.8 million lines of code
(Chromium)

Hardware Designer: Makes Chips



modern chip:
~16 billion transistors
(Apple M1)

In a perfect world...

- Programming languages provide an abstraction

Programmer: Writes Code



Hardware Designer: Makes Chips



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

In a perfect world...

- Historically this worked well



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

In a perfect world...

- Historically this worked well



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz



In a perfect world...

- Historically this worked well

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



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz



In a perfect world...

- Historically this worked well

- Dennard's scaling:

- Computer speed doubles every 1.5 years.



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz



2007

2.1 GHz



In a perfect world...

- Historically this worked well

- Dennard's scaling:

- Computer speed doubles every 1.5 years.



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2003

700 MHz



*3x increase
over 4 years*

2007

2.1 GHz



In a perfect world...

- Historically this worked well



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

- Programming languages also evolved:
 - Garbage Collection
 - Memory Safety
 - Runtimes

However...

These trends slowed down in ~2007



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

However...

These trends slowed down in ~2007



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz



However...

These trends slowed down in ~2007



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

2007
2.1 GHz



2017
2.5 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



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

Reexamining the stack



The negotiators:
Specifications
Compiles
Runtimes
Interpreters

Optimized and designed over decades for single core.

Parallel programming breaks down these abstractions

Performance - e.g., memory contention

Safety - how to reason about shared data

Reexamining the stack

- Nowadays



To efficiently program parallel architectures, developers looking past the negotiators and more directly at hardware

Reexamining the stack

- Nowadays

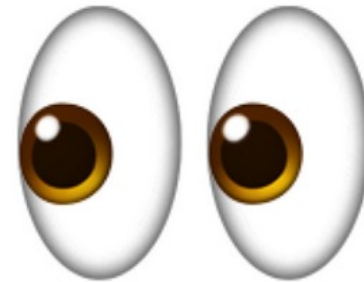
We're going to pick a language that allows reasoning about how it is executed on the hardware



Reexamining the stack

- Nowadays

Heavy runtime
(GC, JIT) makes it
hard to reason
about
performance on
hardware



Reexamining the stack










- Nowadays



often intuitive mappings to assembly
lean runtime



Modern trends

Jan 2023	Jan 2022	Change	Programming Language	Ratings	Change
1	1		 Python	16.36%	+2.78%
2	2		 C	16.26%	+3.82%
3	4	▲	 C++	12.91%	+4.62%
4	3	▼	 Java	12.21%	+1.55%
5	5		 C#	5.73%	+0.05%
6	6		 Visual Basic	4.64%	-0.10%
7	7		 JavaScript	2.87%	+0.78%
8	9	▲	 SQL	2.50%	+0.70%
9	8	▼	 Assembly language	1.60%	-0.25%

source: Tiobe index

Reasons for C's popularity

- There have always been reasons to program close to the hardware
 - Embedded systems
 - parallelism
 - diversity of architecture (especially recently)
- C/C++ has a massive ecosystem, large and active community. It can keep up with hardware trends and allows extremely efficient code to be written while keeping a manageable level of abstraction

C/++ is not perfect

- **Downsides:** Security issues, bugs, pointers, complicated specification
- designing a fast, and safe programming language is *difficult*. Very much an open problem. Many of you may be working on it in your career.
- Rust seems like an interesting development. Not yet to the place where I see it being viable to teach.
 - currently ranked 18 (moving up!)
 - It's a lot to learn a new language and parallelism in one quarter...

Python?

- Great language for scripting
 - We will use it to automate experiments in this class
- The GIL (global interpreter lock) restricts parallelism significantly.
 - makes the language safe
- TensorFlow and Pytorch?
 - wrappers around low-level kernels that execute outside of the python interpreter

Lecture Schedule

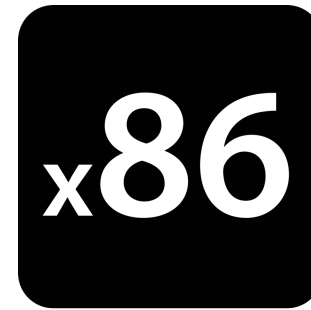
- Overview - why do we need a lecture on compilation and architecture?
- **Compilation** - *How do we translate a program from a human-accessible language to a language that the processor understands*
- Architecture - How do processors execute programs?
- Example

Compilation:

Language



ISA



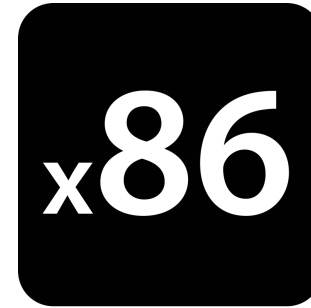
Compilation:

Language



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

ISA



Compilation:

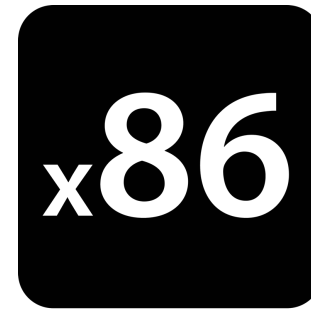
Language



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

*If we didn't have
computers, would this
mean anything?*

ISA



Compilation:

Language



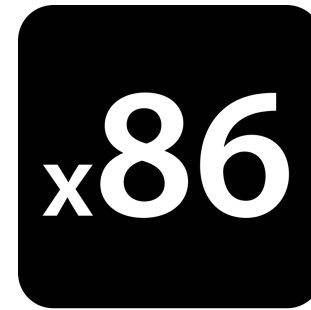
```
int add(int a, int b) {  
    return a + b;  
}
```

Officially defined by the specification

ISO standard: costs \$200

~1400 pages

ISA



Compilation:

Language



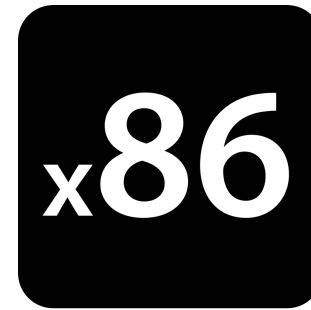
```
int add(int a, int b) {  
    return a + b;  
}
```

Officially defined by the specification

ISO standard: costs \$200

~1400 pages

ISA



official specification

Intel provides a specification: *free*

2200 pages

Compilation:

Language



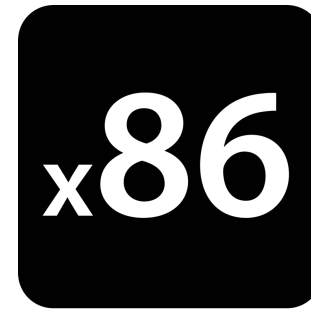
```
int add(int a, int b) {  
    return a + b;  
}
```

Officially defined by the specification

ISO standard: costs \$200

~1400 pages

ISA



???

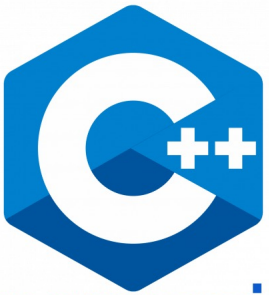
official specification

Intel provides a specification: *free*

2200 pages

Compilation:

Language



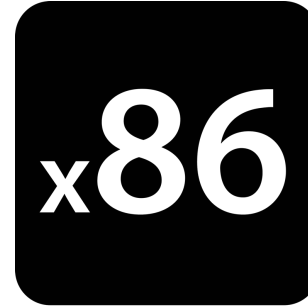
Programming

```
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
```

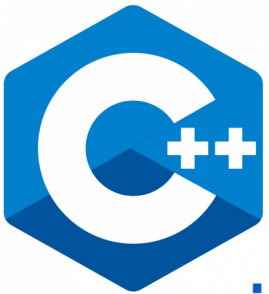
official specification

Intel provides a specification: *free*

2200 pages

Compilation:

Language



Programming

```
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

How about a more complicated program?

Quadratic formula

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

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)
```

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)
```



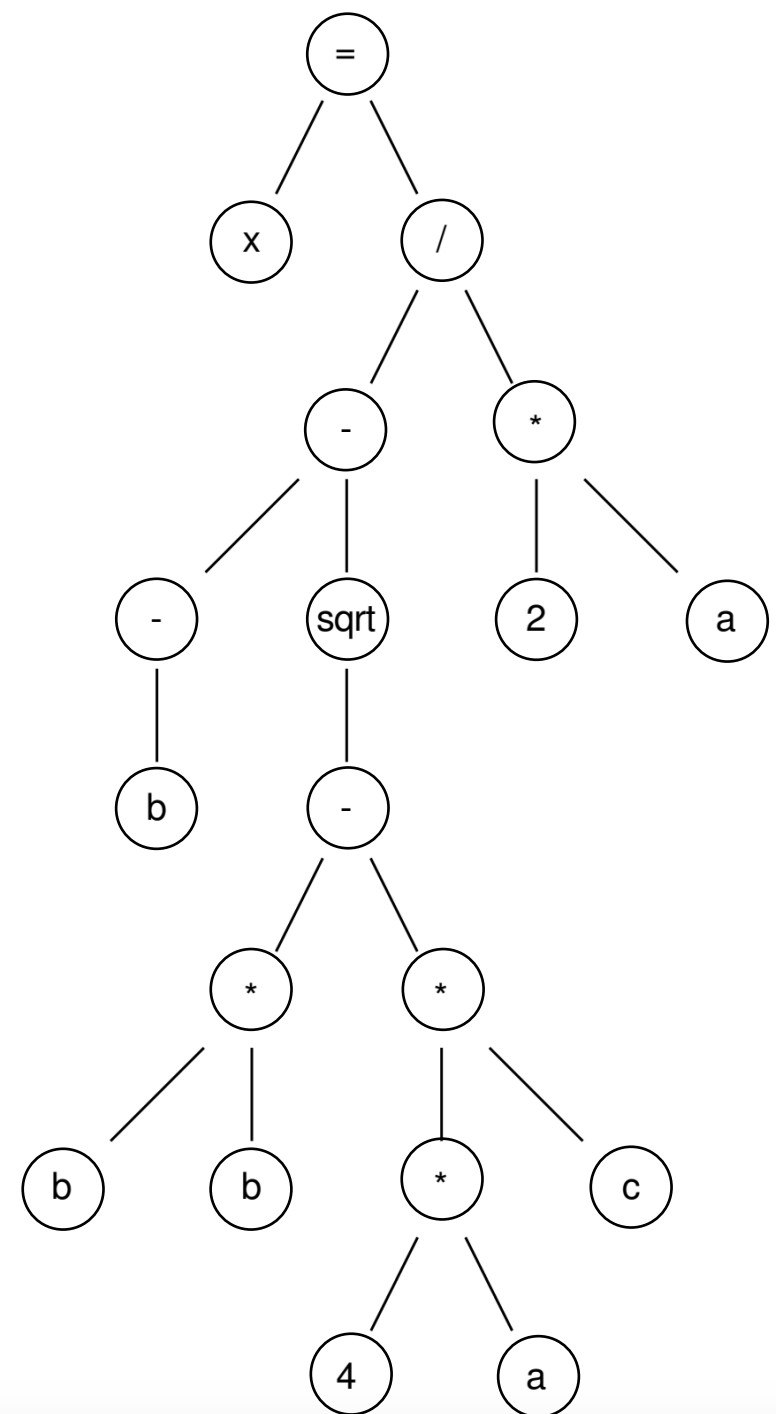
official specification

Intel provides a specification: *free*
2200 pages

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

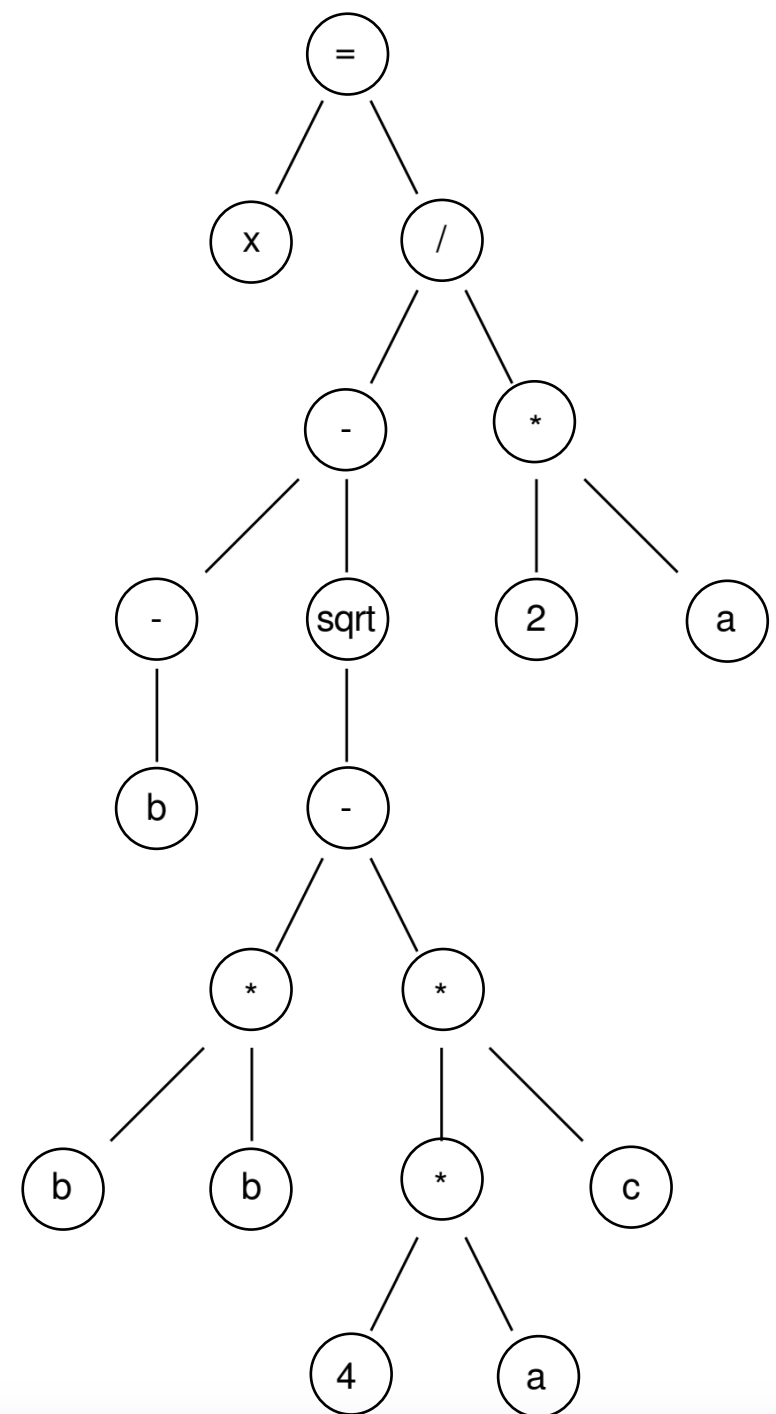
$$x = (-b - \text{sqrt}(b*b - 4 * a * c)) / (2*a)$$

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



Simplify this code:

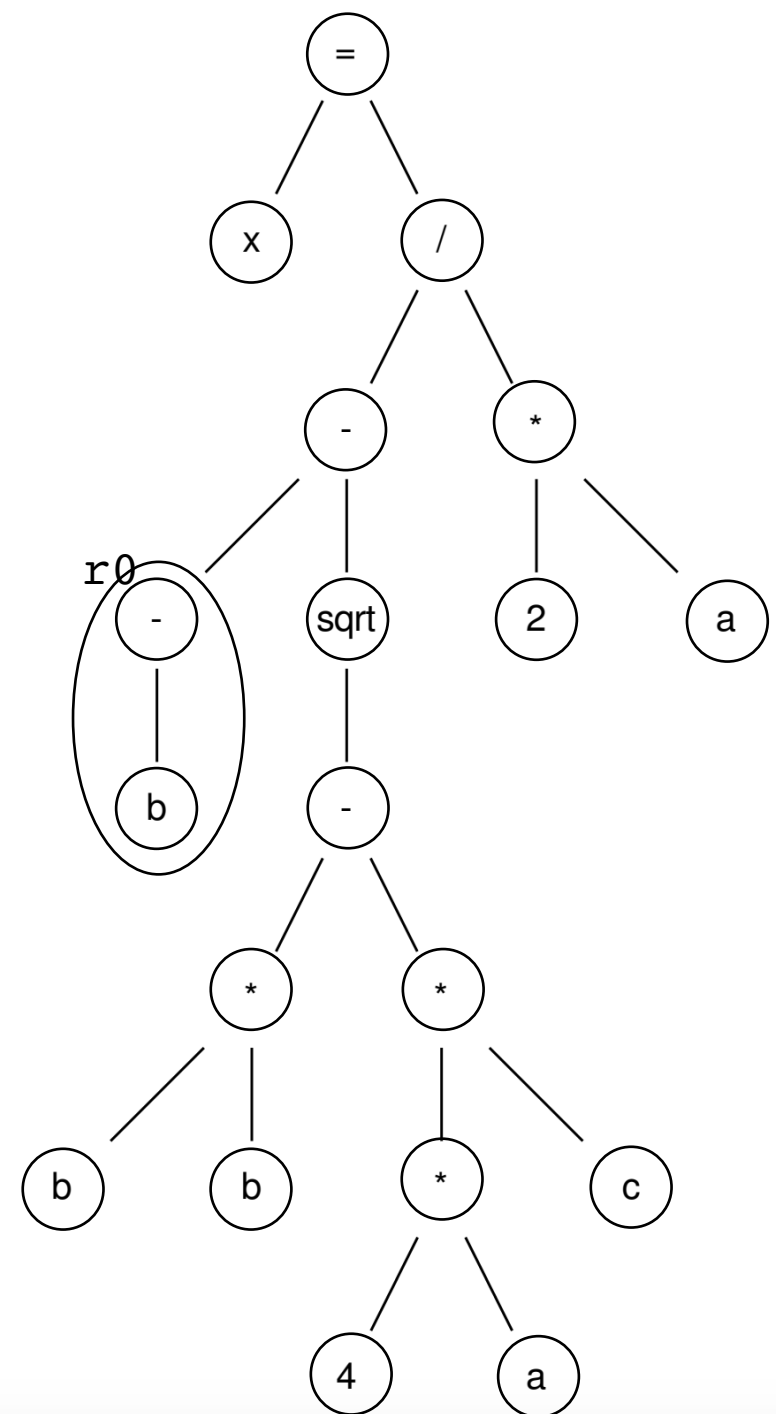
post-order traversal, using temporary variables



Simplify this code:

post-order traversal, using temporary variables

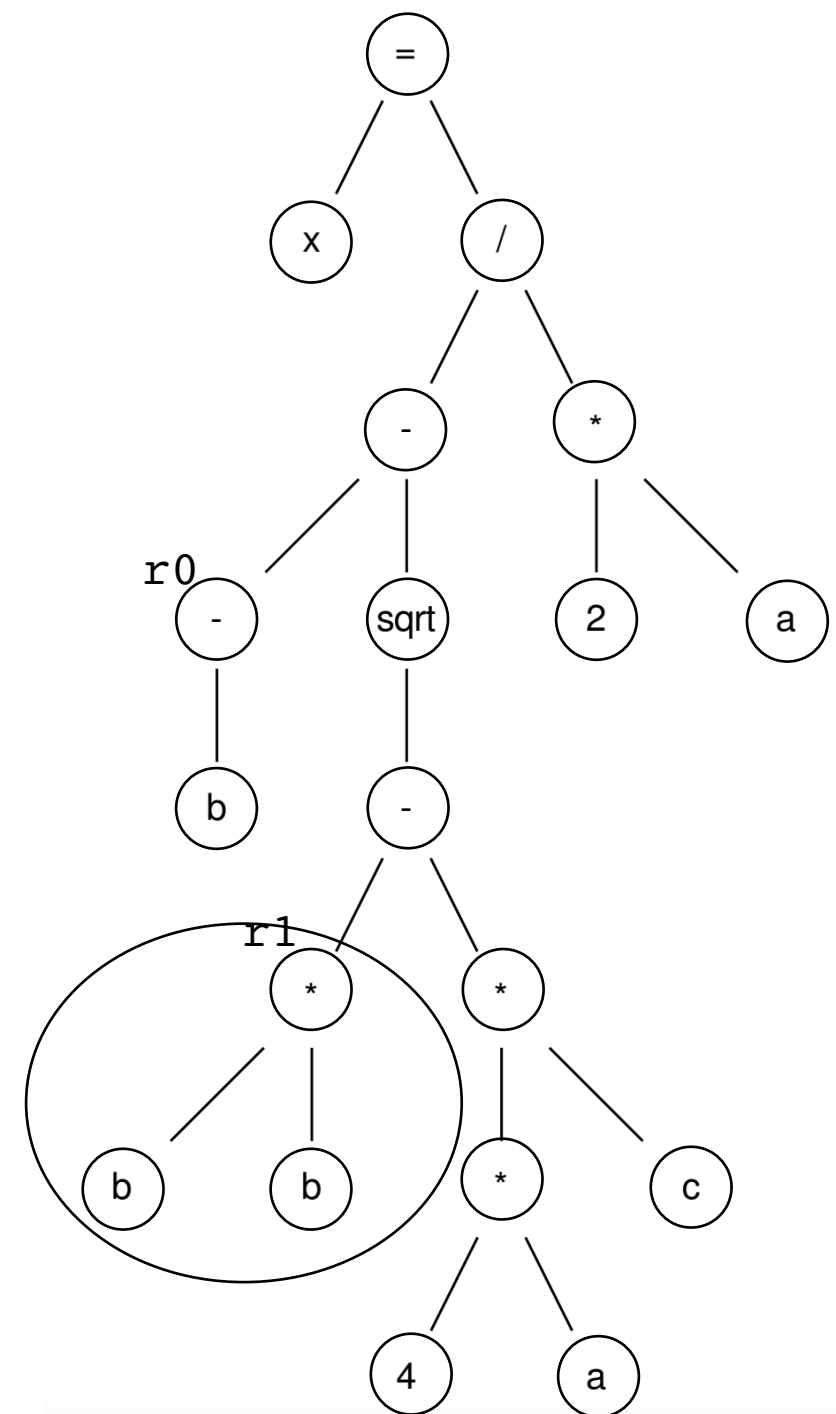
```
r0 = neg(b);
```



Simplify this code:

post-order traversal, using temporary variables

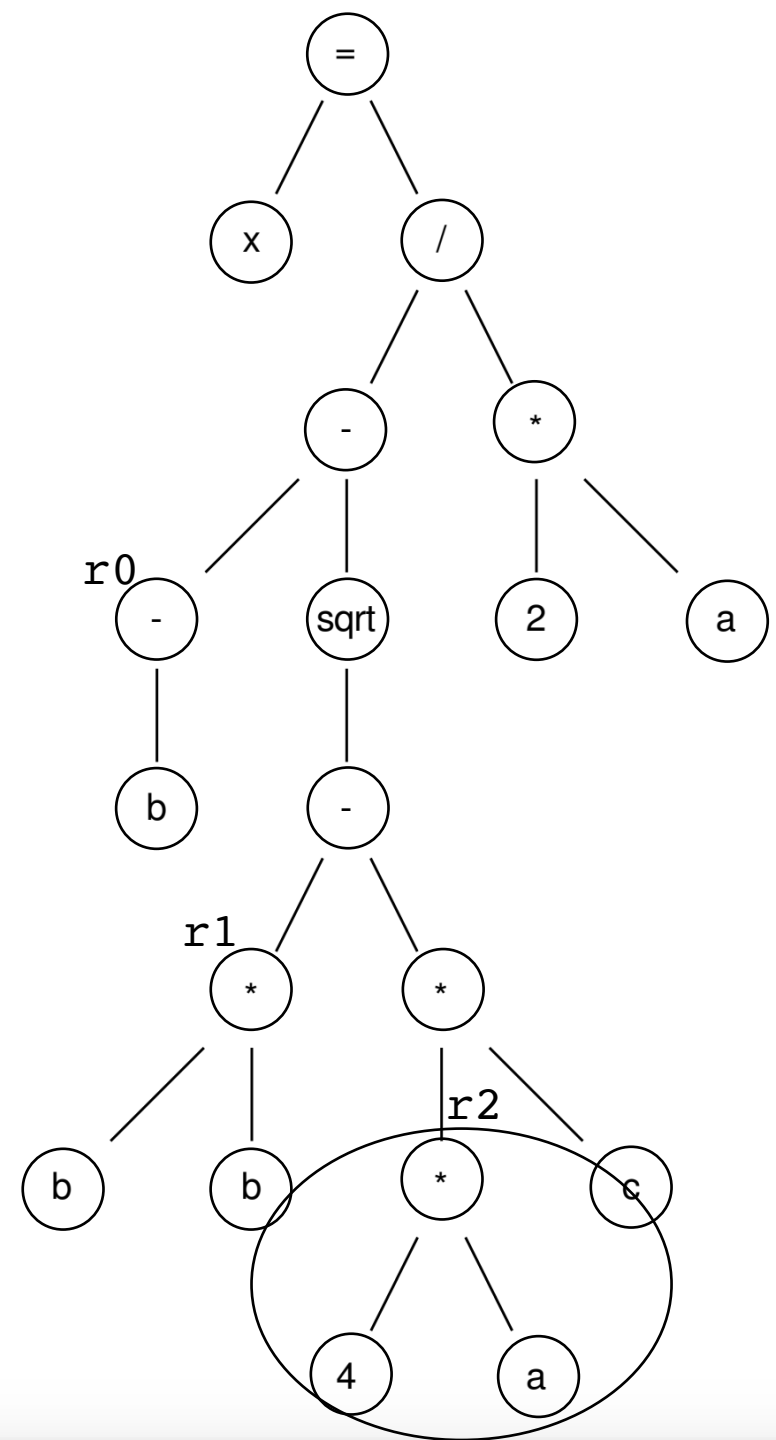
```
r0 = neg(b);  
r1 = b * b;
```



Simplify this code:

post-order traversal, using temporary variables

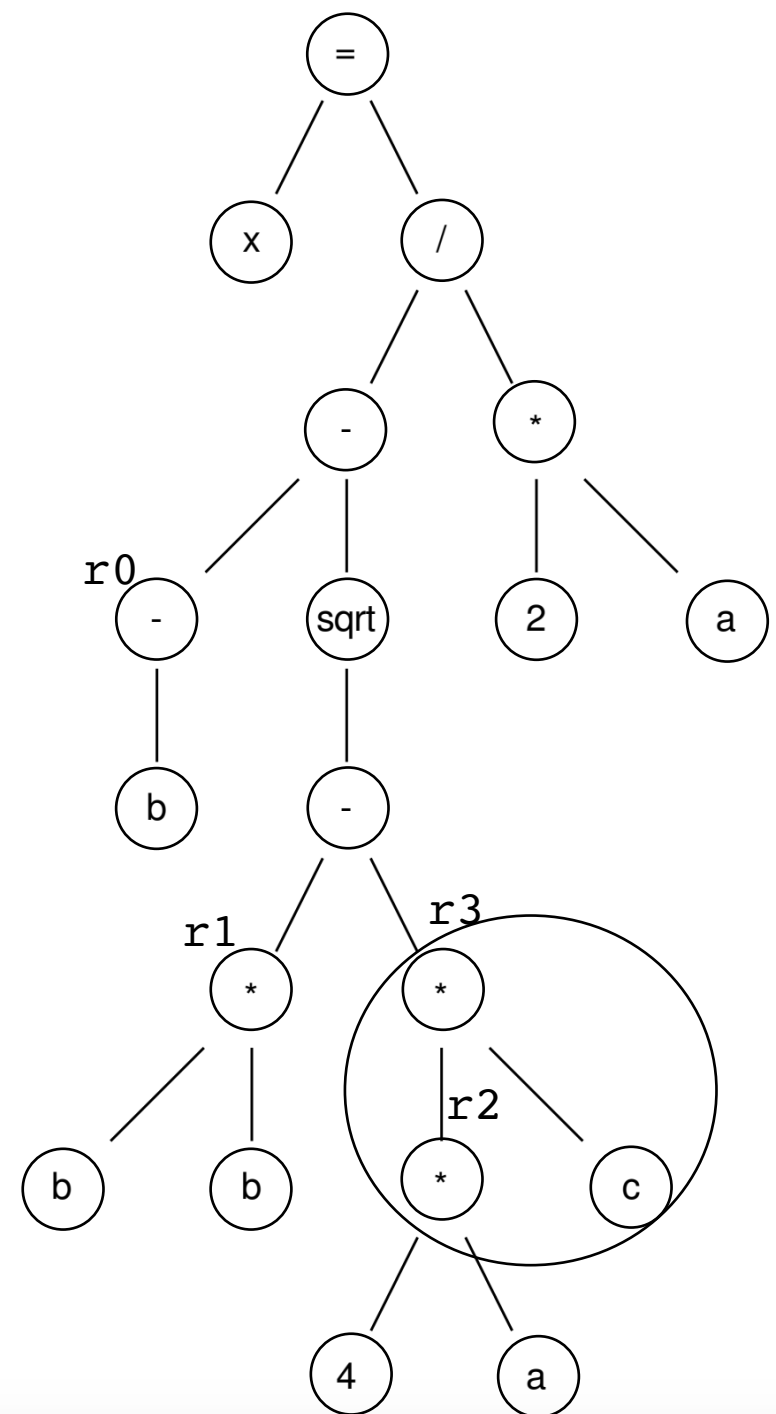
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;
```



Simplify this code:

post-order traversal, using temporary variables

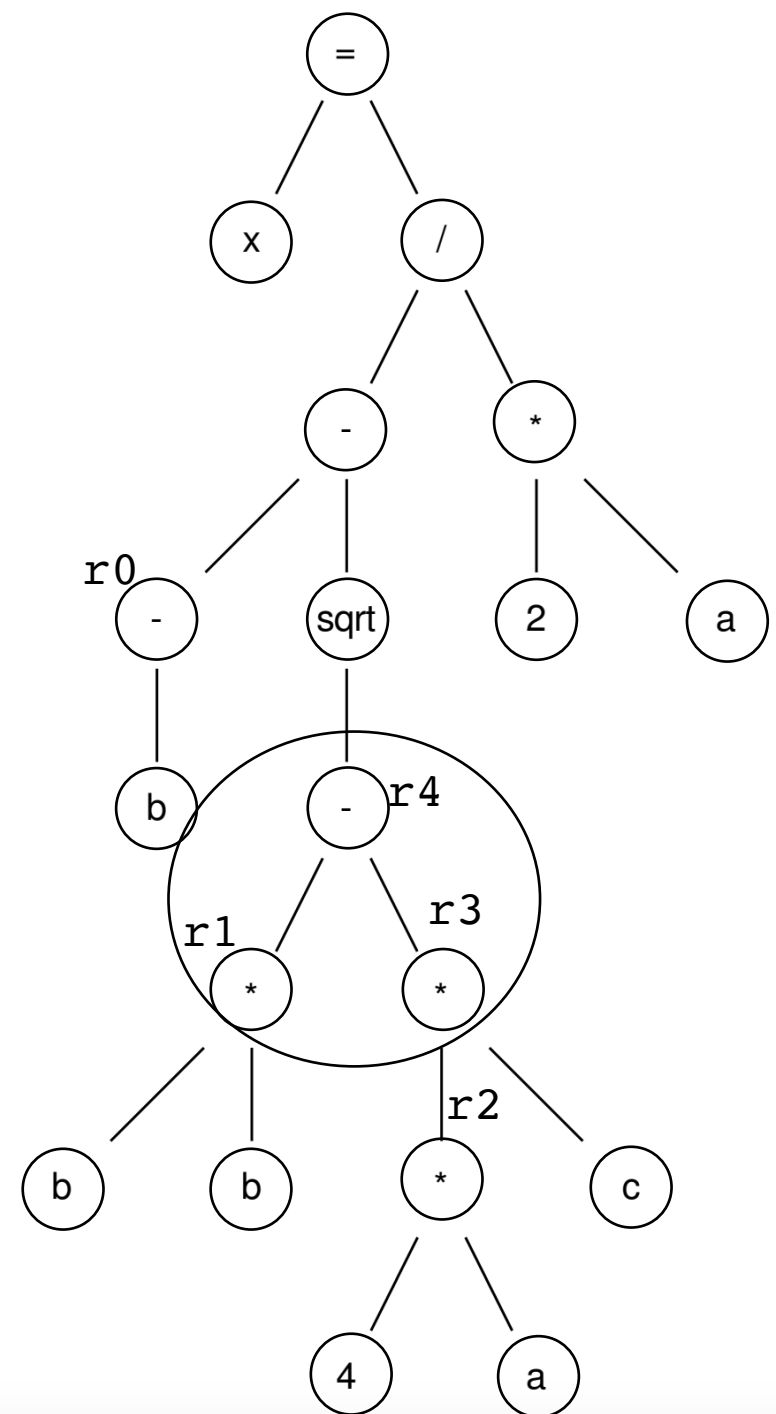
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;
```



Simplify this code:

post-order traversal, using temporary variables

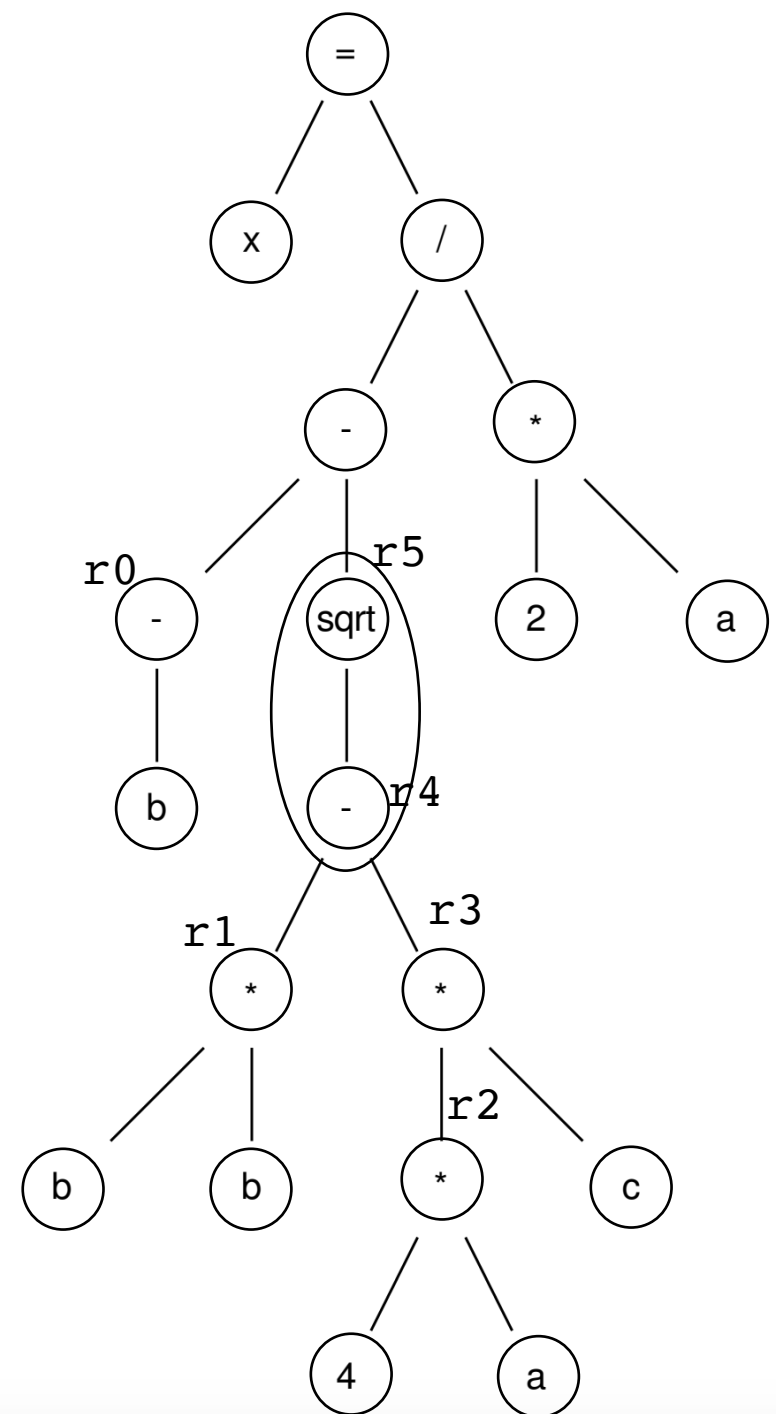
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;
```



Simplify this code:

post-order traversal, using temporary variables

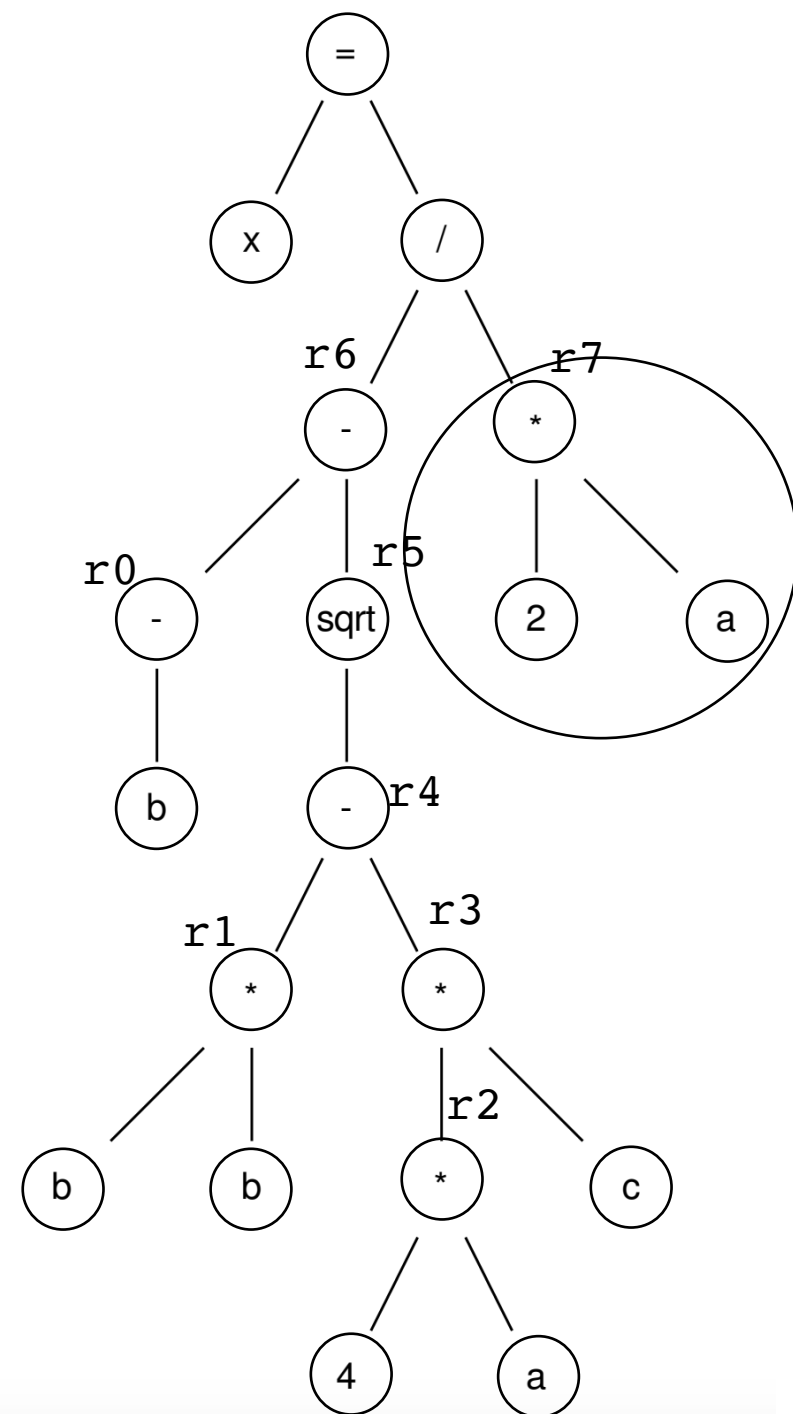
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);
```



Simplify this code:

post-order traversal, using temporary variables

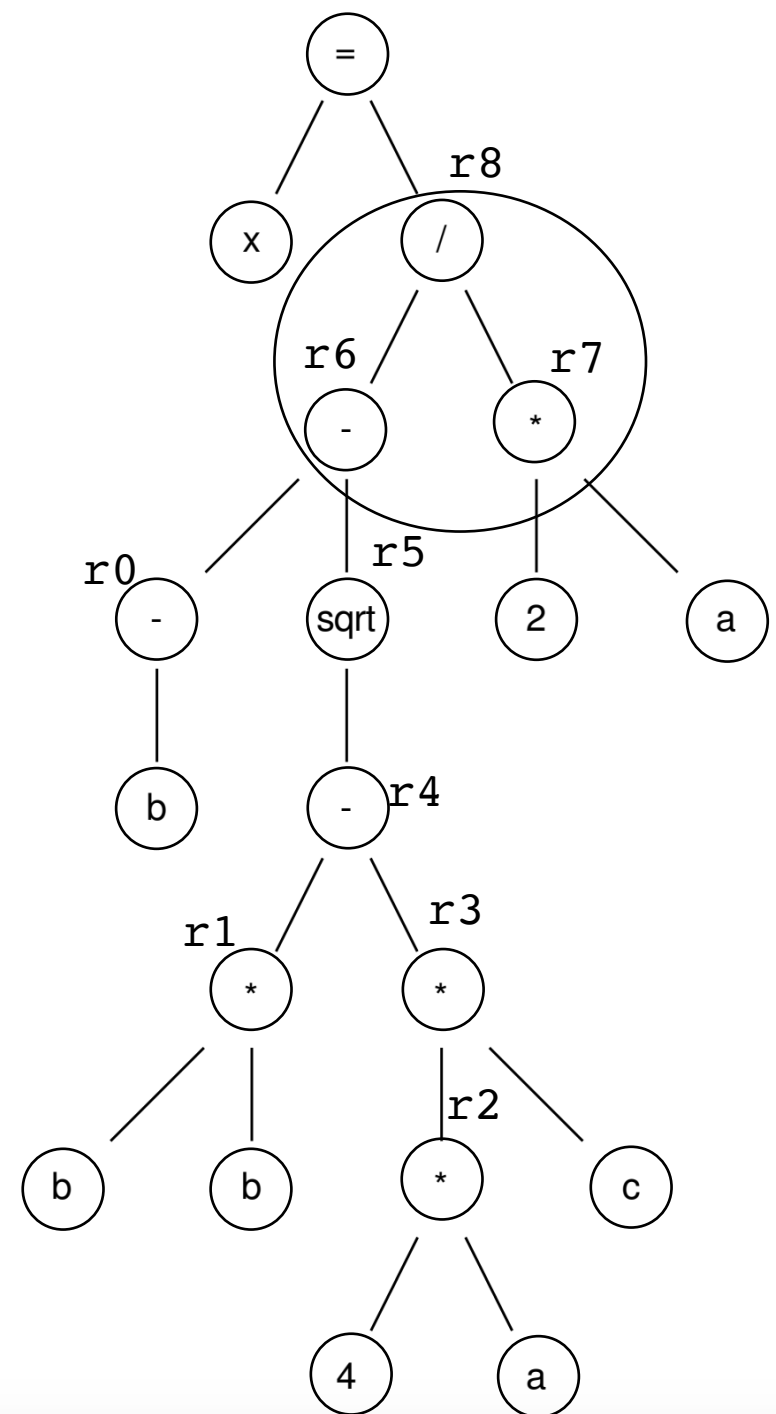
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;
```



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;
```



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


```
1 // Type your code here, or load an example.
2 float sqrt(float x);
3
4 float add(float a, float b, float c) {
5     return (-b - sqrt(b*b - 4 * a * c)) / (2*a);
6
7 }
8
```

C program

```
A Output... Filter... Libraries + Add new... Add tool...
1
2 define dso_local float @_Z3addfff(float %0, float %1, float %2) #0 !dbg !
3     %4 = alloca float, align 4
4     %5 = alloca float, align 4
5     %6 = alloca float, align 4
6     store float %0, float* %4, align 4
7     call void @llvm.dbg.declare(metadata float* %4, metadata !12, metadata
8     store float %1, float* %5, align 4
9     call void @llvm.dbg.declare(metadata float* %5, metadata !14, metadata
10    store float %2, float* %6, align 4
11    call void @llvm.dbg.declare(metadata float* %6, metadata !16, metadata
12    %7 = load float, float* %5, align 4, !dbg !18
13    %8 = fneg float %7, !dbg !19
14    %9 = load float, float* %5, align 4, !dbg !20
15    %10 = load float, float* %5, align 4, !dbg !21
16    %11 = fmul float %9, %10, !dbg !22
17    %12 = load float, float* %4, align 4, !dbg !23
18    %13 = fmul float 4.000000e+00, %12, !dbg !24
19    %14 = load float, float* %6, align 4, !dbg !25
20    %15 = fmul float %13, %14, !dbg !26
21    %16 = fsub float %11, %15, !dbg !27
22    %17 = call float @_Z4sqrtf(float %16), !dbg !28
23    %18 = fsub float %8, %17, !dbg !29
24    %19 = load float, float* %4, align 4, !dbg !30
25    %20 = fmul float 2.000000e+00, %19, !dbg !31
26    %21 = fdiv float %18, %20, !dbg !32
27    ret float %21, !dbg !33
28 }
```

llvm IR

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!

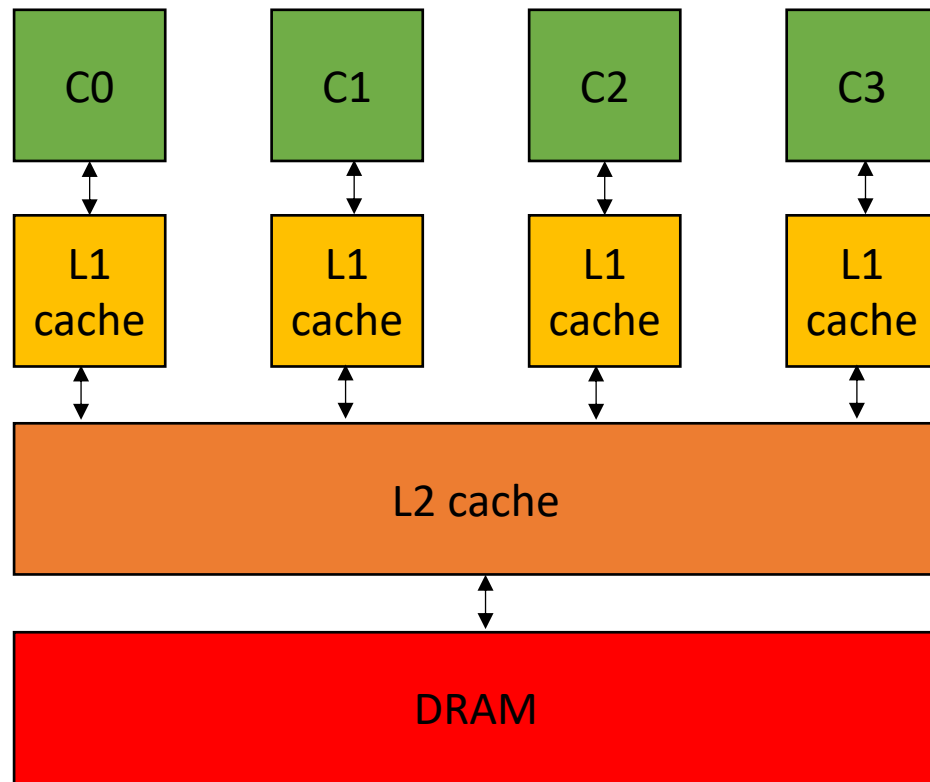
Zoom out

- This can be a lot if you don't have a compiler background; don't feel overwhelmed!
- To be successful in this class, you don't need to be an expert on compilation, ISAs, or IRs.
- The important thing is to have a mental model of how your complex code is broken down into instructions that are executed on hardware, especially loads and stores

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

Architecture visual



Core

A core executes a stream
of sequential ISA instructions

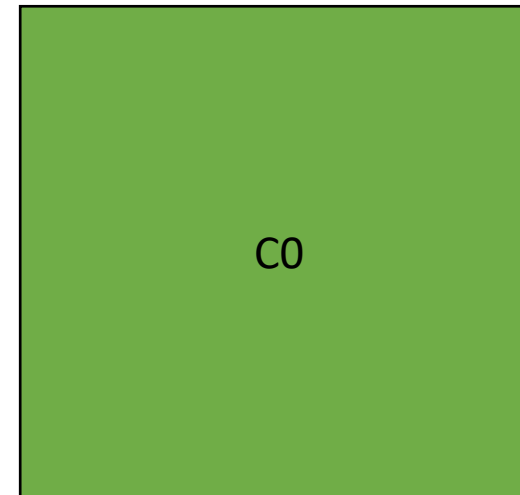
A good mental model executes
1 ISA instruction per cycle

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

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
```

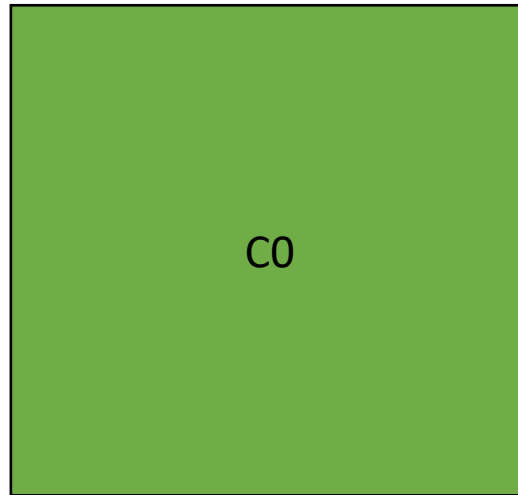
Thread 0

Compiled function #1

```
movss  xmm0, dword ptr [rbp - 16]
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

Sometimes multiple programs want to share the same core.



Core

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
```

Compiled function #1

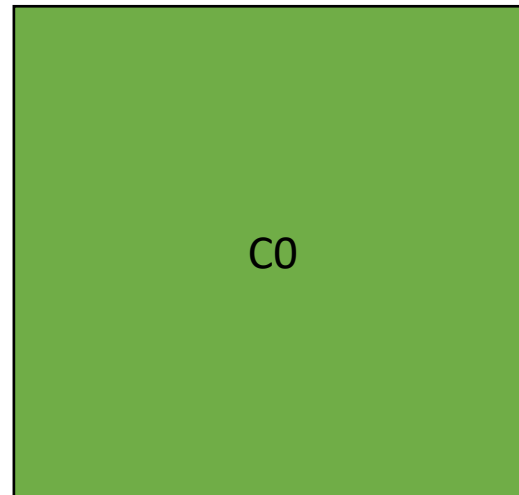
```
movss  xmm0, dword ptr [rbp - 16]
movss  xmm0, dword ptr [rbp - 8]
mulss  xmm0, dword ptr [rbp - 8]
movss  xmm1, dword ptr [rip + .LCPI0_1]
mulss  xmm1, dword ptr [rbp - 4]
mulss  xmm1, dword ptr [rbp - 12]
subss  xmm0, xmm1
call   sqrt(float)
movaps xmm1, xmm0
movss  xmm0, dword ptr [rbp - 16]
subss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

Sometimes multiple programs want to share the same core.



Thread 0

Thread 1



Core



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

Core

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
```

Compiled function #0

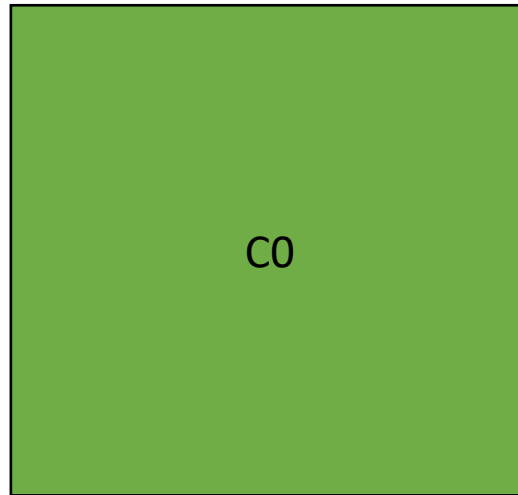
```
13 movd eax, xmm0
14 xor eax, 2147483648
15 movd xmm0, eax
16 movss dword ptr [rbp - 16], xmm0
17 movss xmm0, dword ptr [rbp - 8]
18 mulss xmm0, dword ptr [rbp - 8]
19 movss xmm1, dword ptr [rip + .LCPI0_1]
20 mulss xmm1, dword ptr [rbp - 4]
21 mulss xmm1, dword ptr [rbp - 12]
22 subss xmm0, xmm1
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
```

Sometimes multiple programs want to share the same core.

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

Thread 1

Thread 0



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

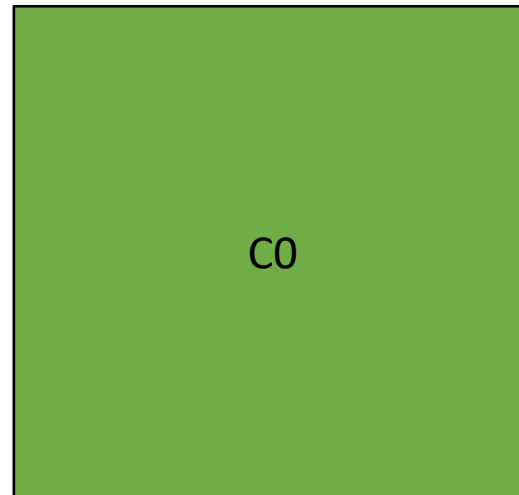
```
movss xmm0, dword ptr [rbp - 8] #
mulss xmm0, dword ptr [rbp - 8]
movss xmm1, dword ptr [rip + .LCPI0_1]
mulss xmm1, dword ptr [rbp - 4]
mulss xmm1, dword ptr [rbp - 12]
subss xmm0, xmm1
call sqrt(float)
movaps xmm1, xmm0
movss xmm0, dword ptr [rbp - 16] #
subss xmm0, xmm1
movss xmm1, dword ptr [rip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

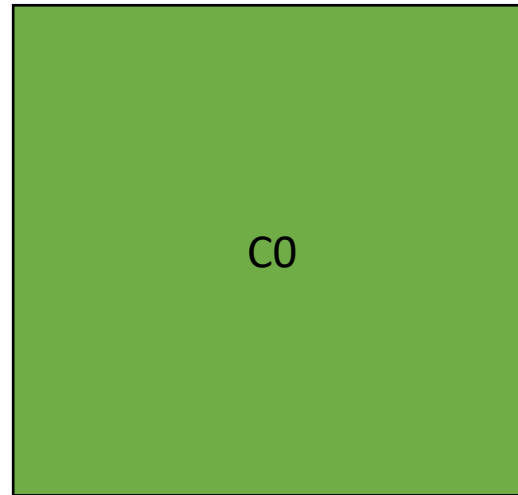
Threads can execute simultaneously (at the same time) if there enough resources.

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

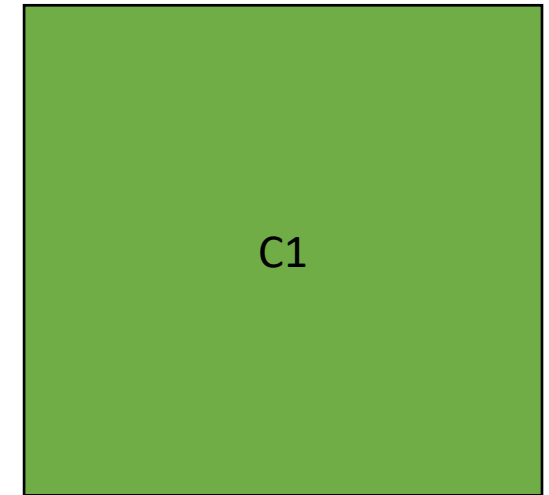


Core

Compiled function #1

```
movss  dword ptr [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



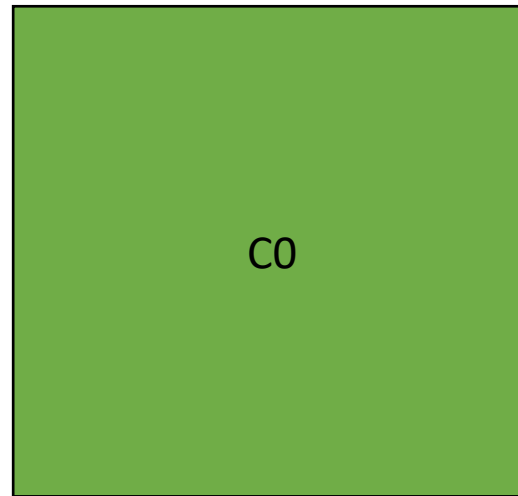
Core

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

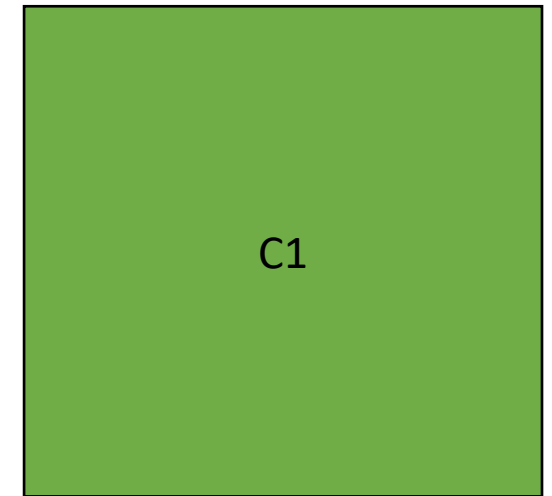


Core

Compiled function #1

```
movss  dword ptr [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



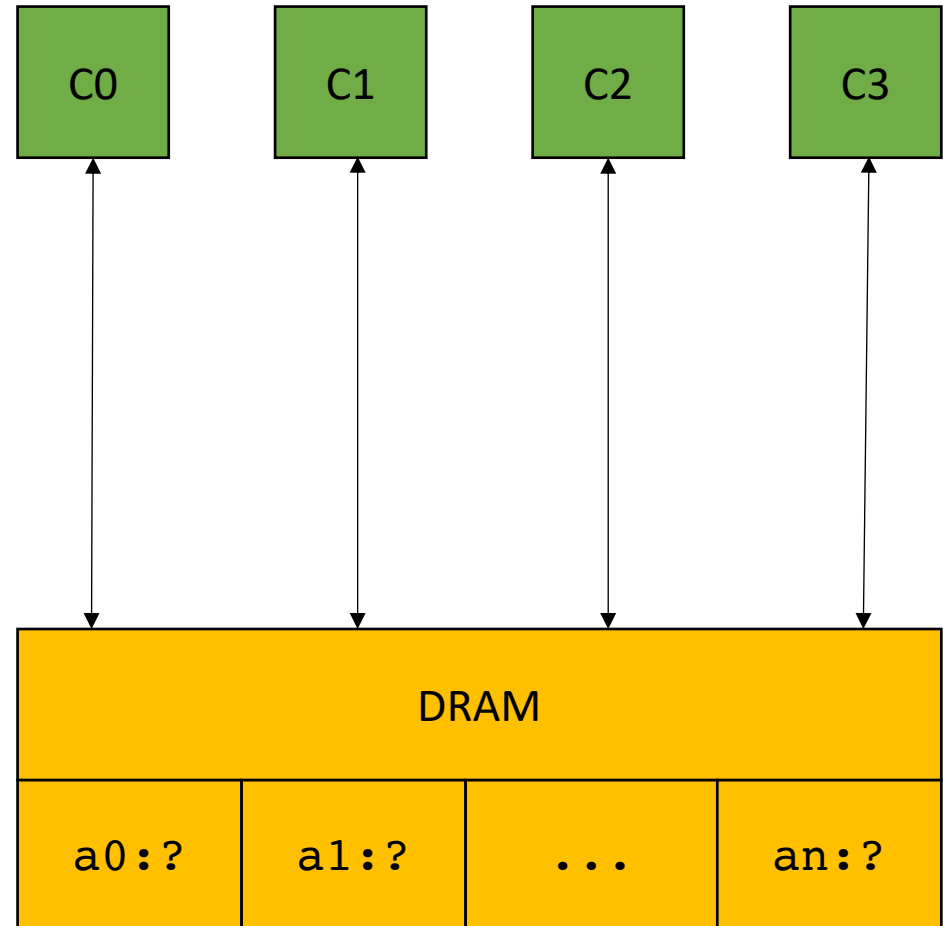
Core

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

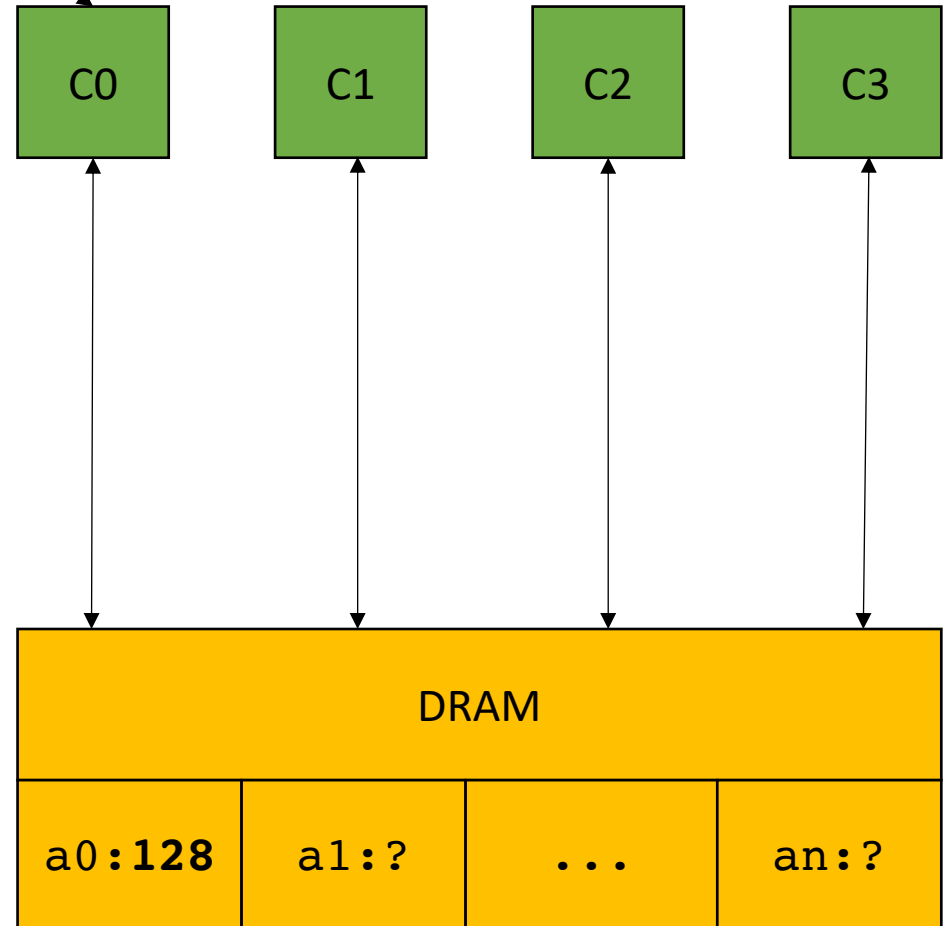
Main memory

`store(a0, 128)`

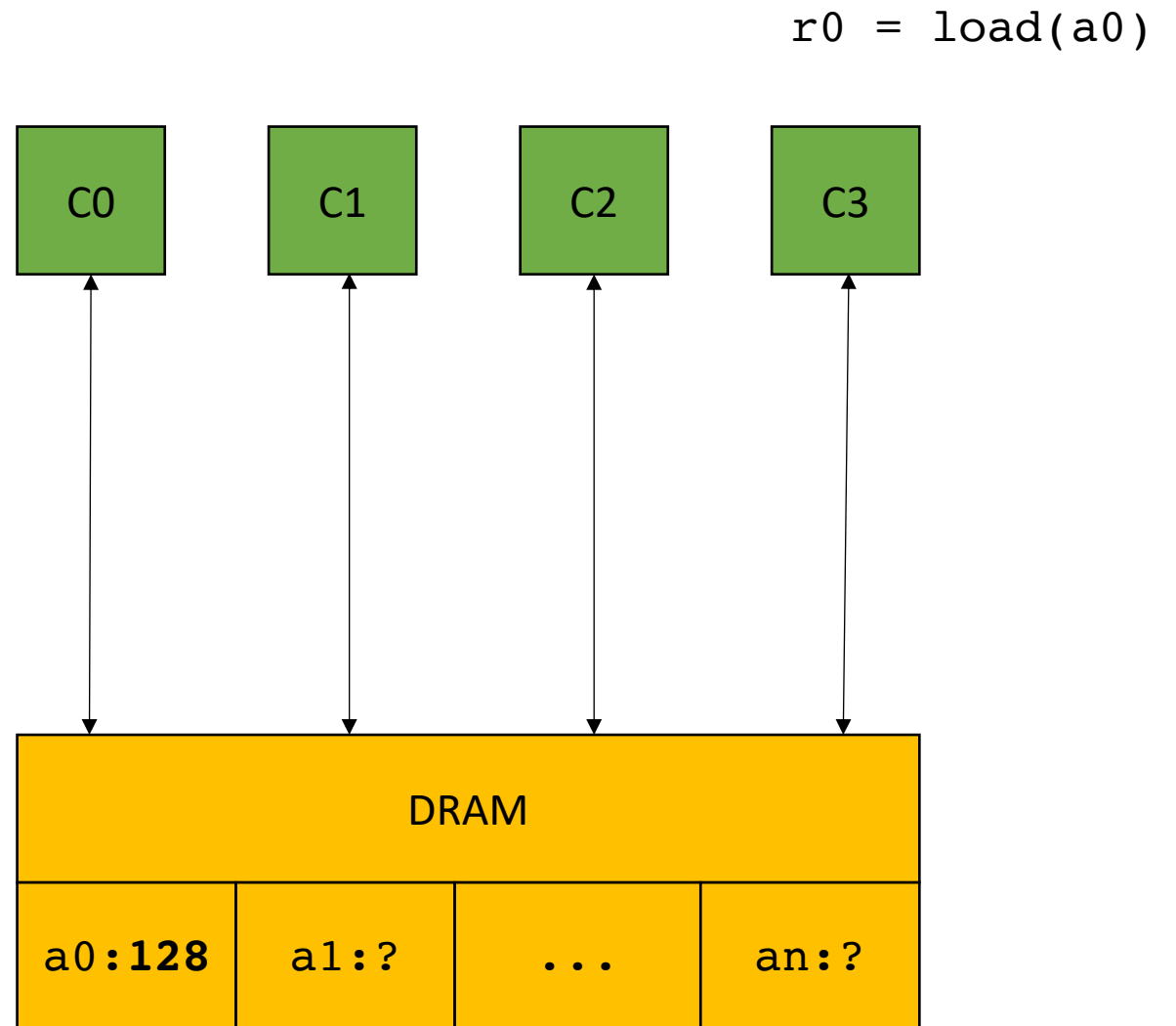


Main memory

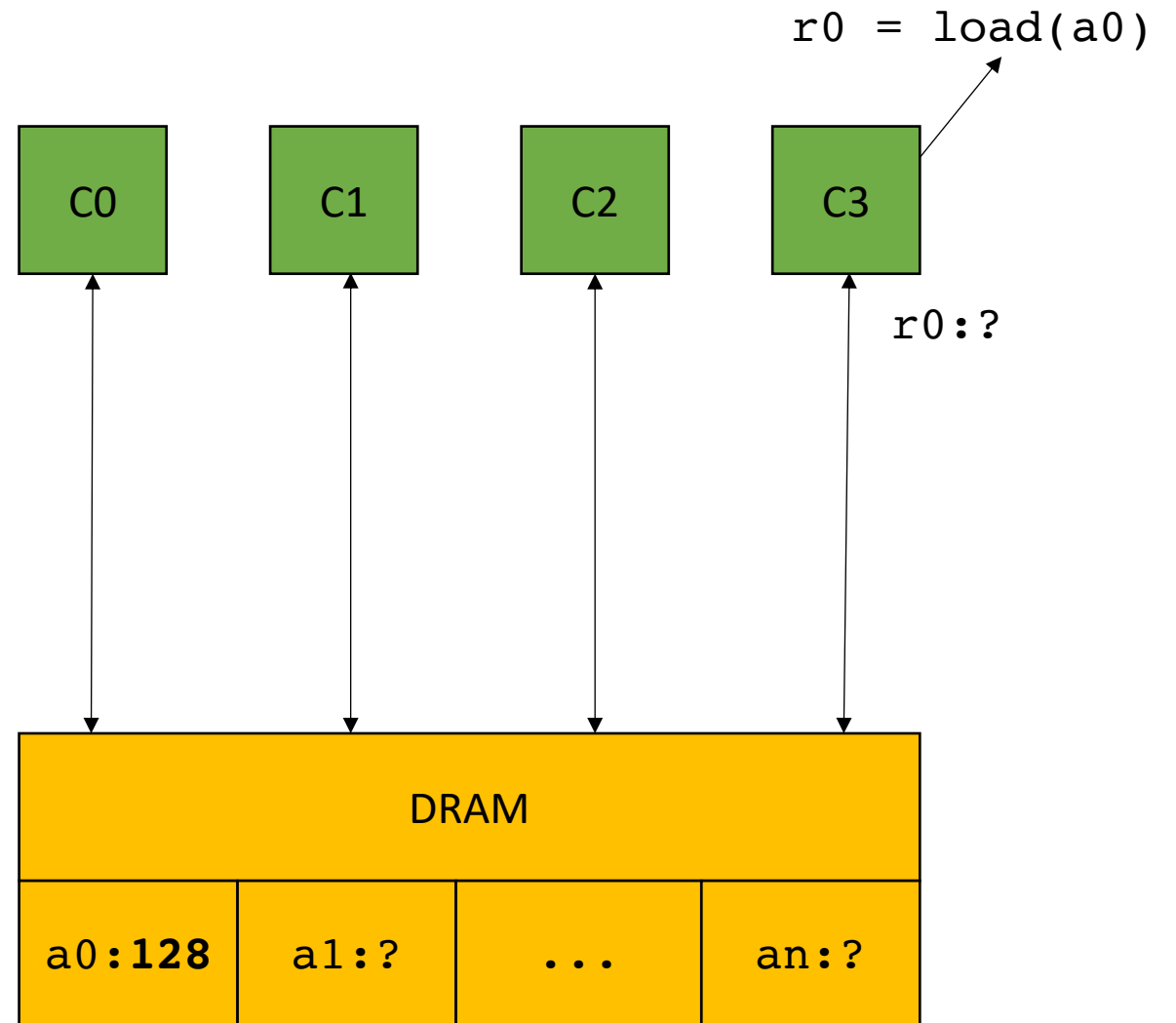
`store(a0, 128)`



Main memory

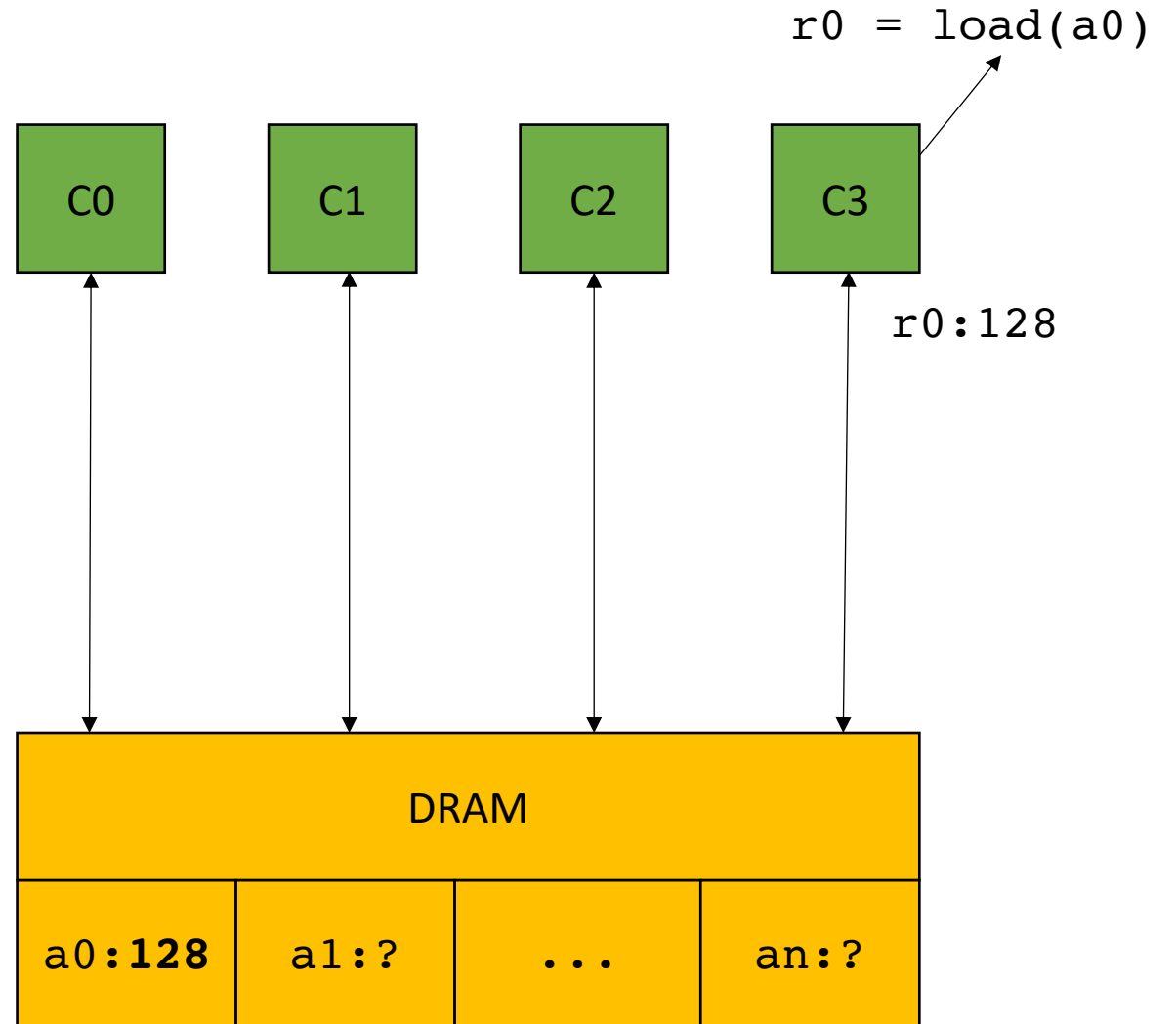


Main memory



Main memory

Problem solved!
Threads can communicate!

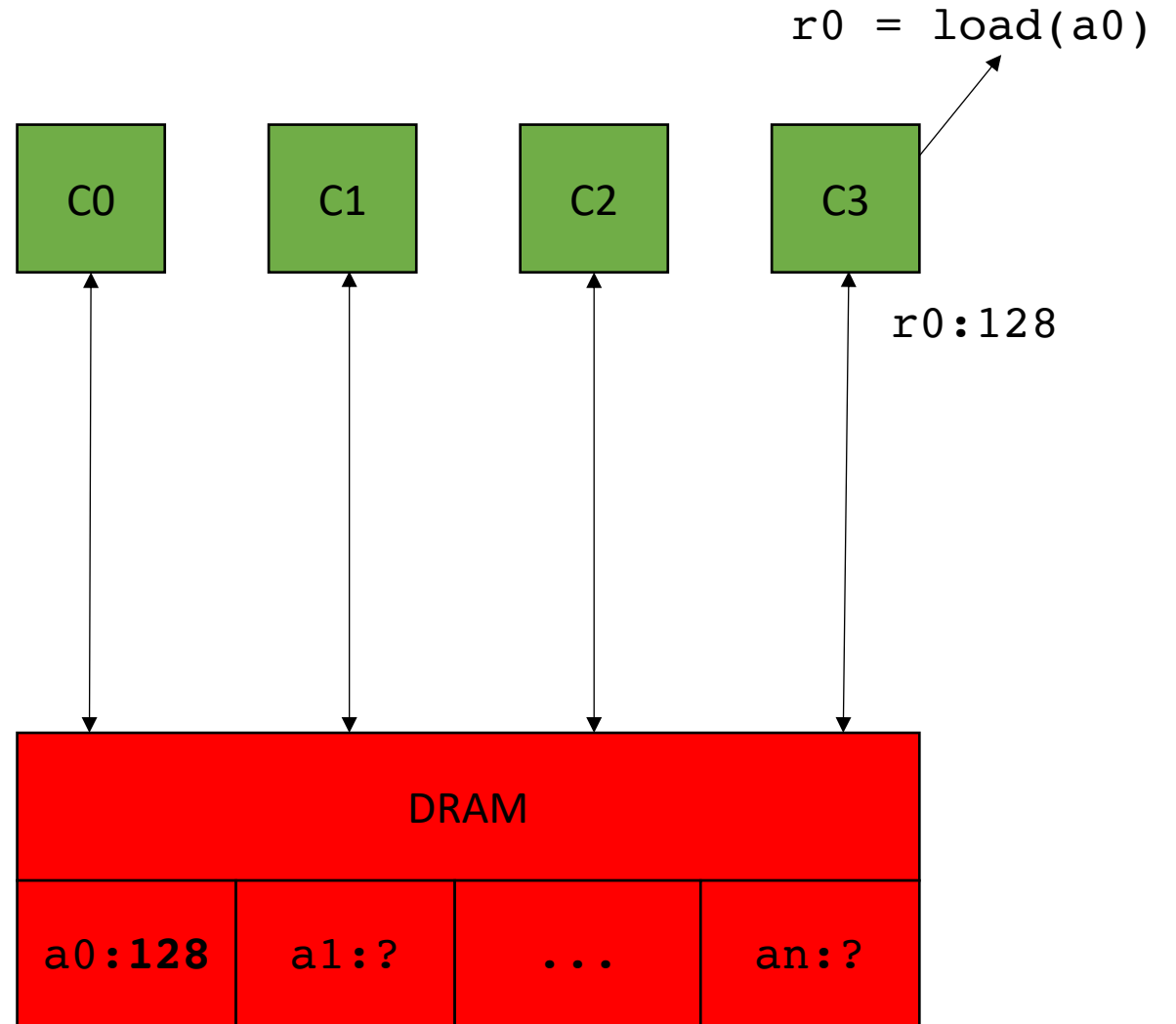


Main memory

Problem solved!

Threads can communicate!

reading a value takes ~200 cycles



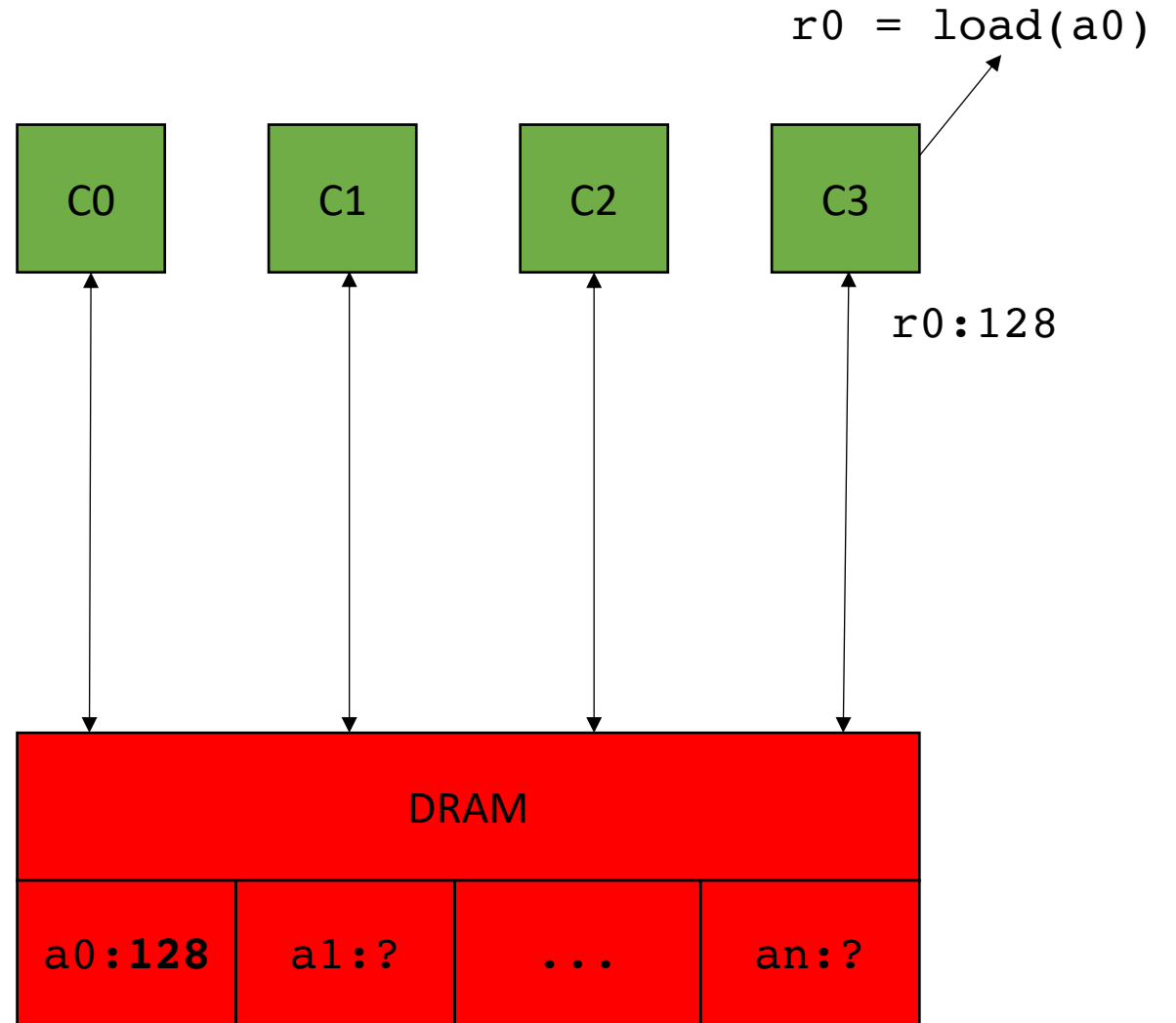
Main memory

Problem solved!

Threads can communicate!

reading a value takes ~200 cycles

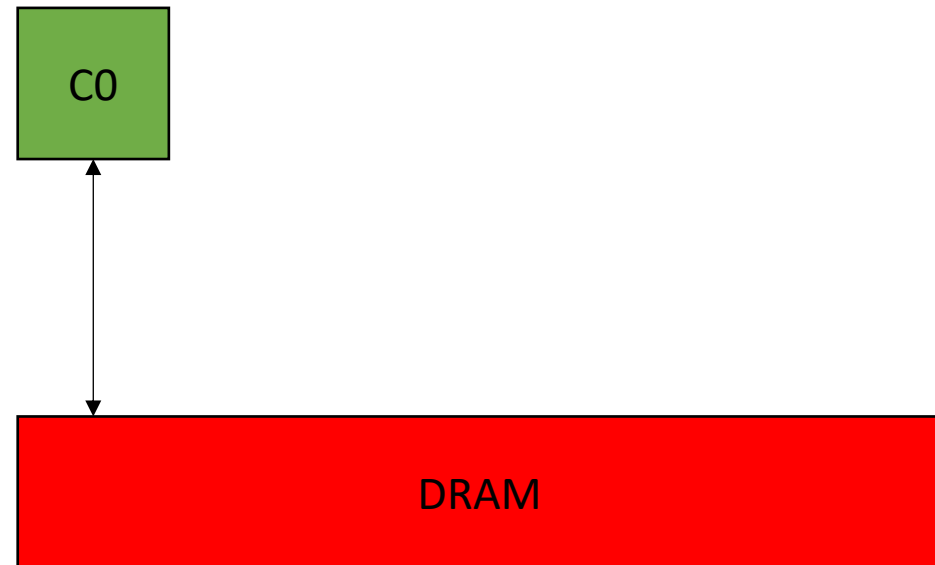
Bad for parallelism, but
also really bad for sequential
code (which we optimized for
decades!)



Main memory

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

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

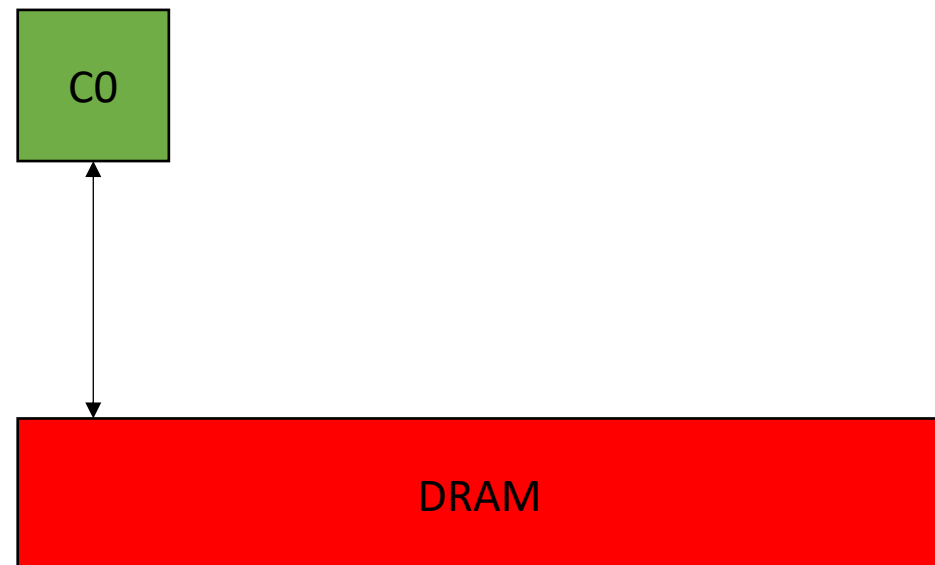


Main memory

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

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

200 cycles



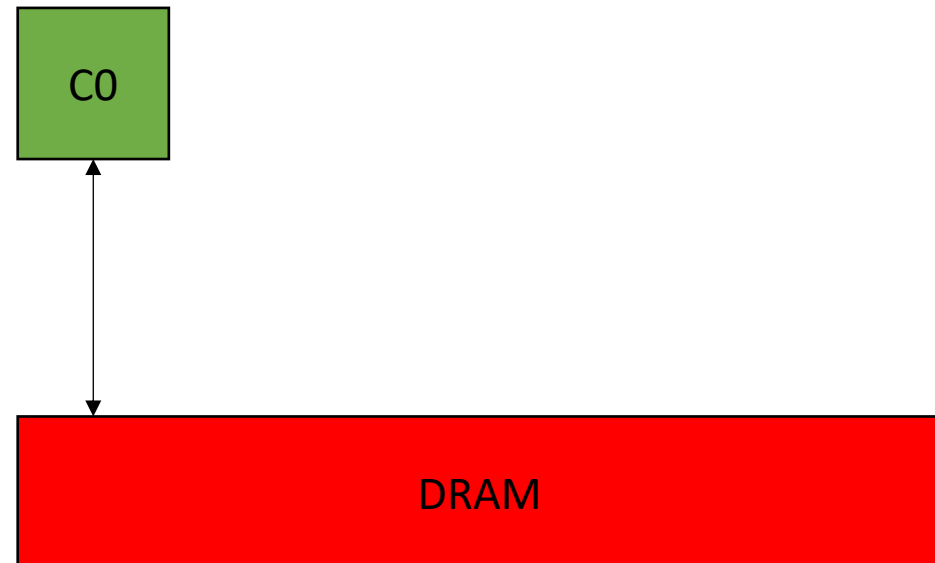
Main memory

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

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

200 cycles

1 cycles



Main memory

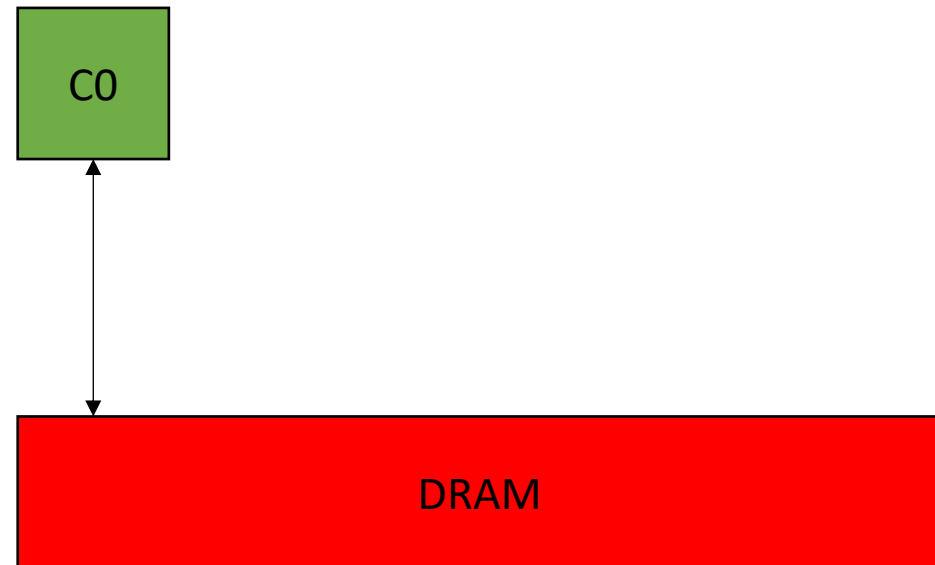
```
int increment(int *a) {  
    a[0]++;  
}
```

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

200 cycles

1 cycles

200 cycles



Main memory

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

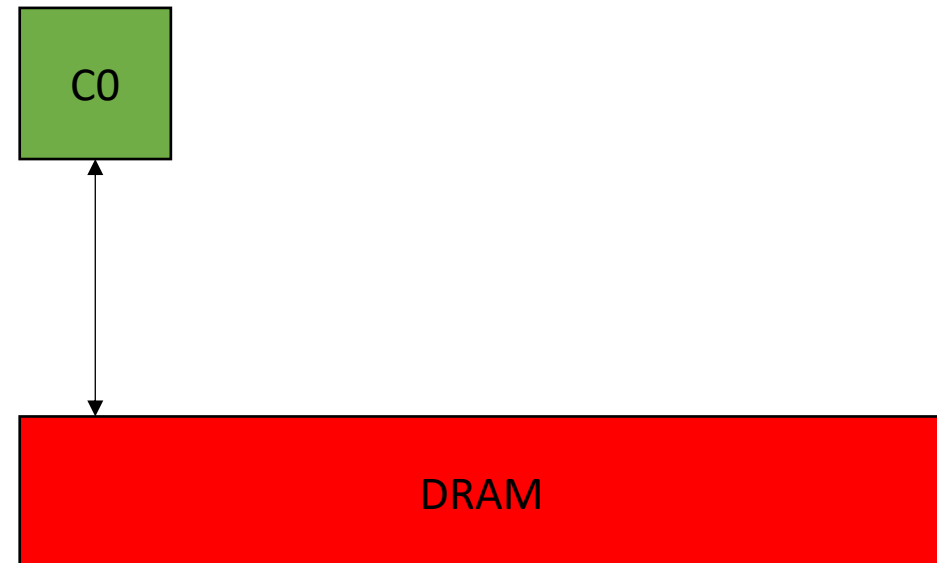
```
%5 = load i32, i32* %4  
%6 = add nsw i32 %5, 1  
store i32 %6, i32* %4
```

200 cycles

1 cycles

200 cycles

401 cycles



Main memory

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

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

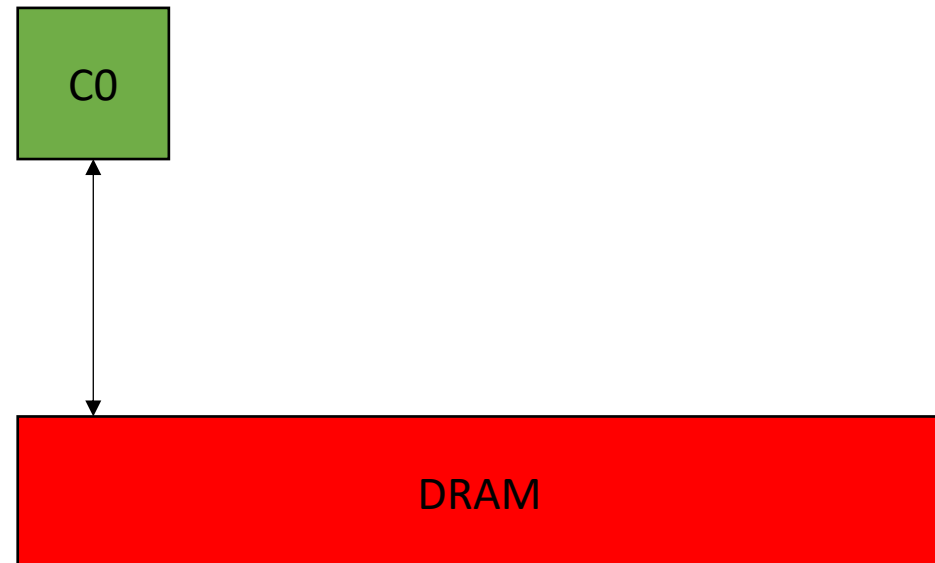
```
%5 = load i32, i32* %4      200 cycles  
%6 = add nsw i32 %5, 1      1 cycles  
store i32 %6, i32* %4      200 cycles
```

200 cycles

1 cycles

200 cycles

401 cycles



Main memory

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

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

40100 cycles!

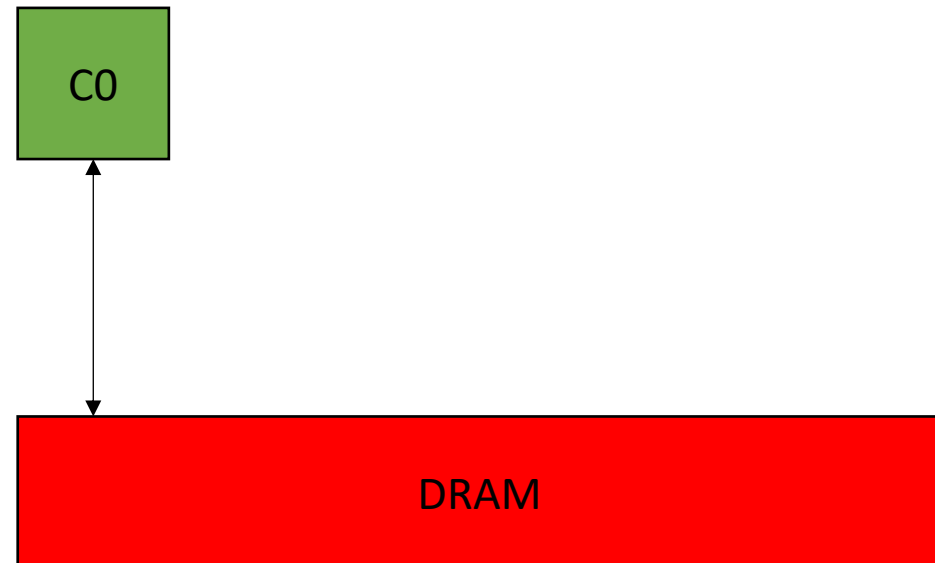
```
%5 = load i32, i32* %4      200 cycles  
%6 = add nsw i32 %5, 1      1 cycles  
store i32 %6, i32* %4      200 cycles
```

200 cycles

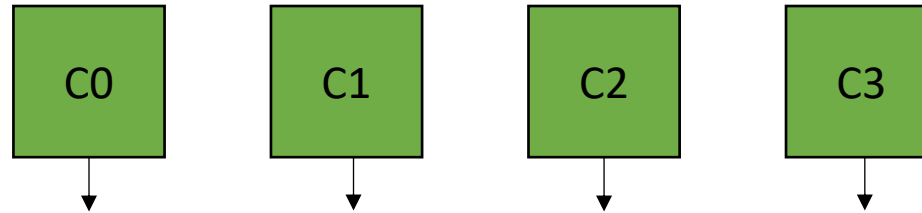
1 cycles

200 cycles

401 cycles



Caches

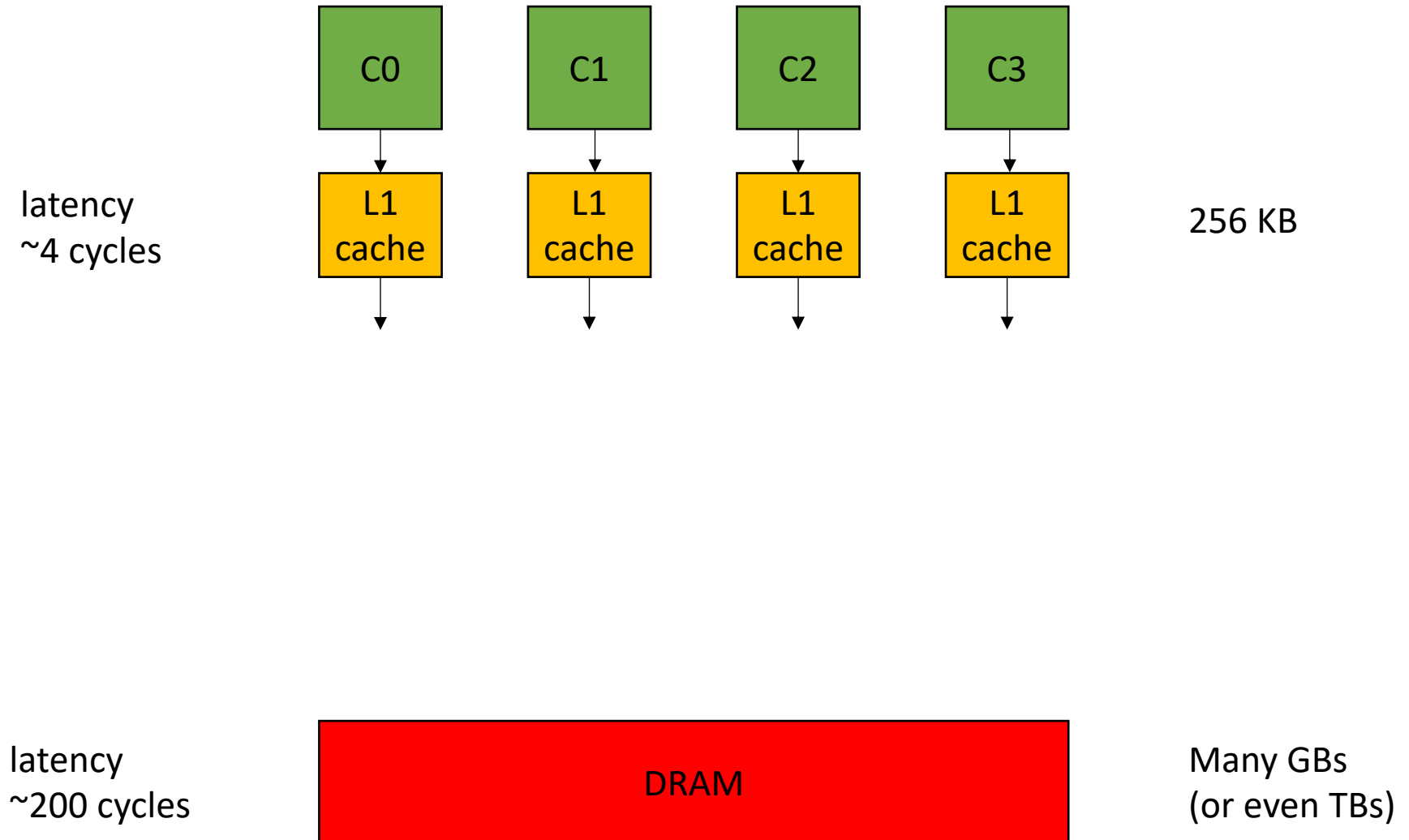


latency
~200 cycles

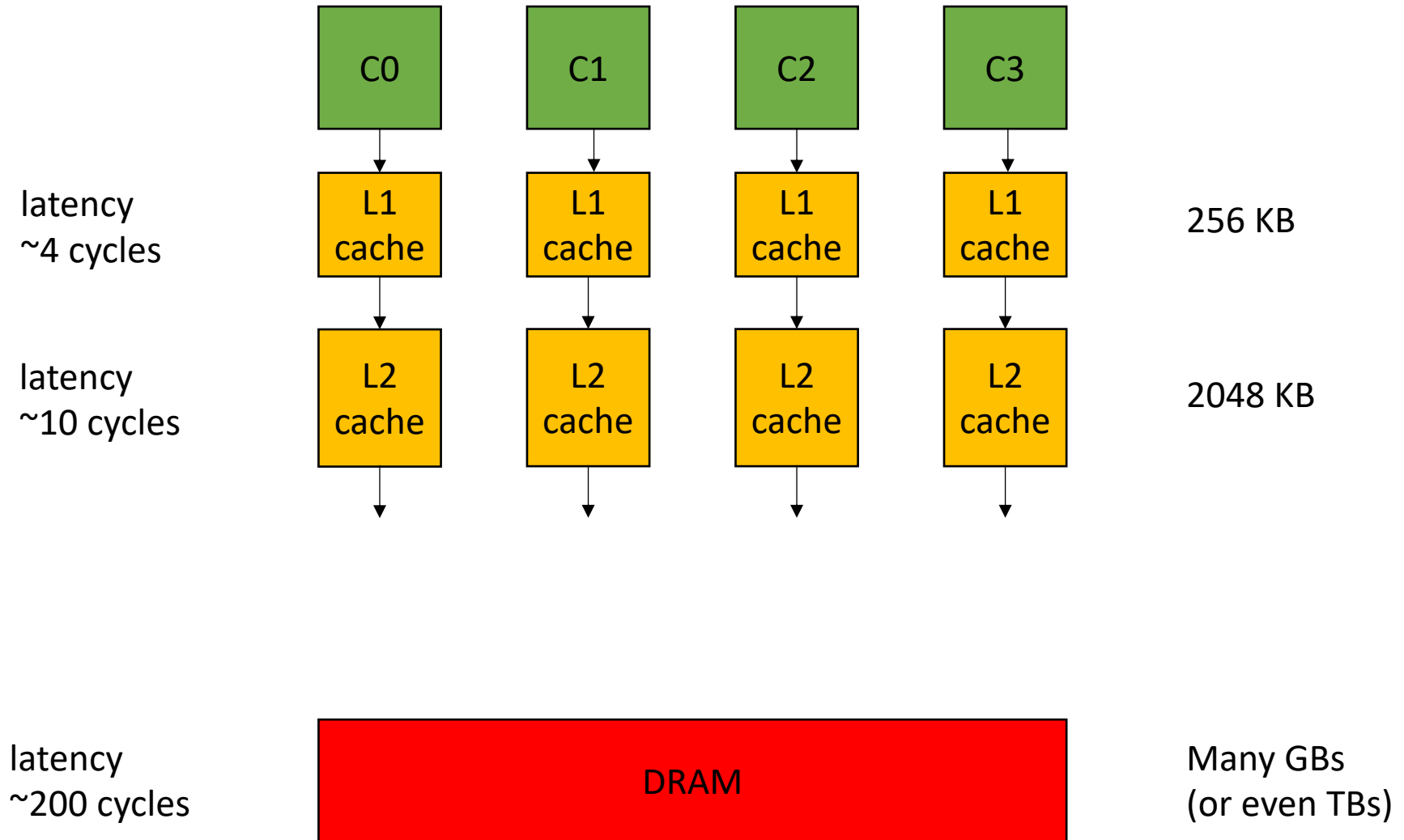
DRAM

Many GBs
(or even TBs)

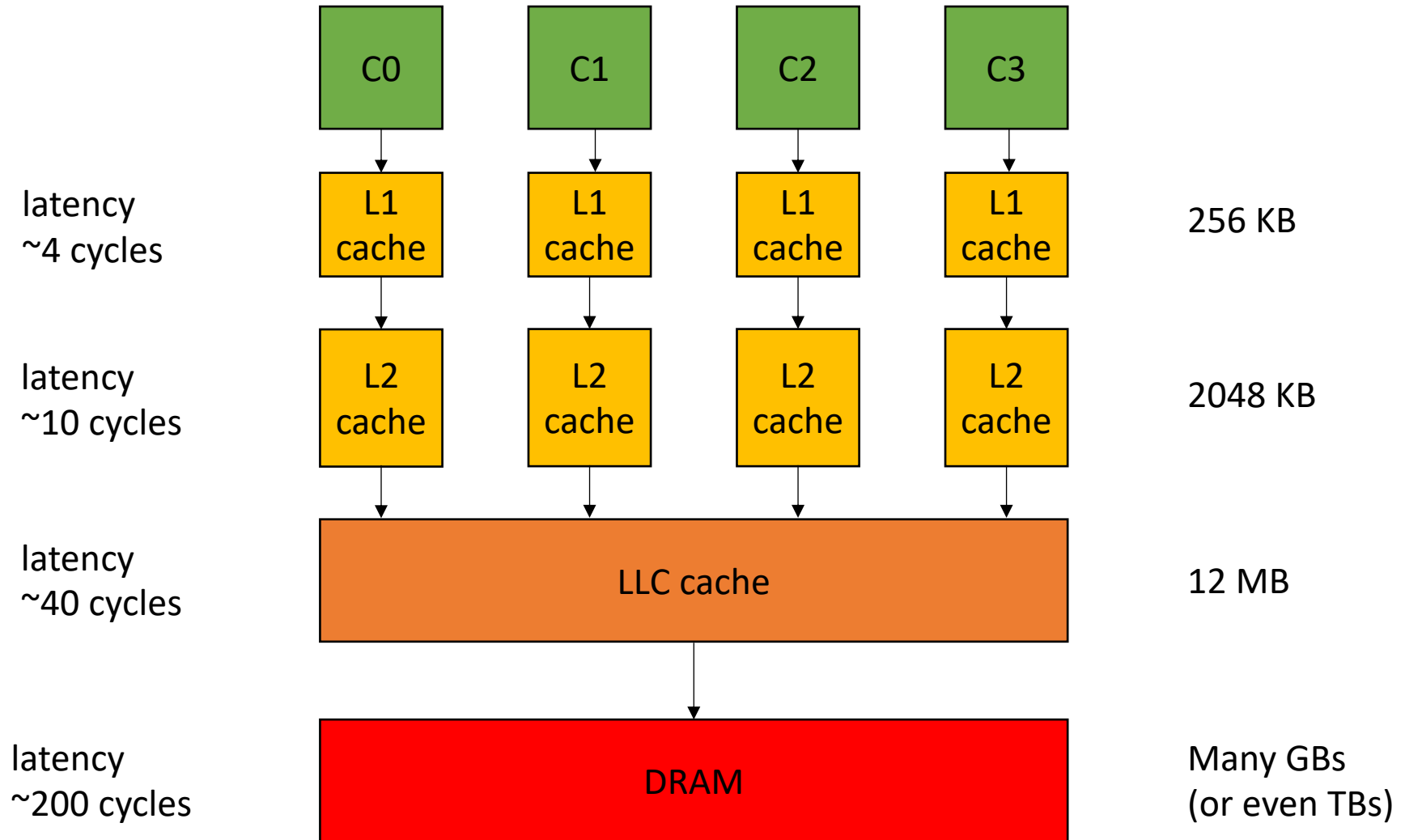
Caches



Caches



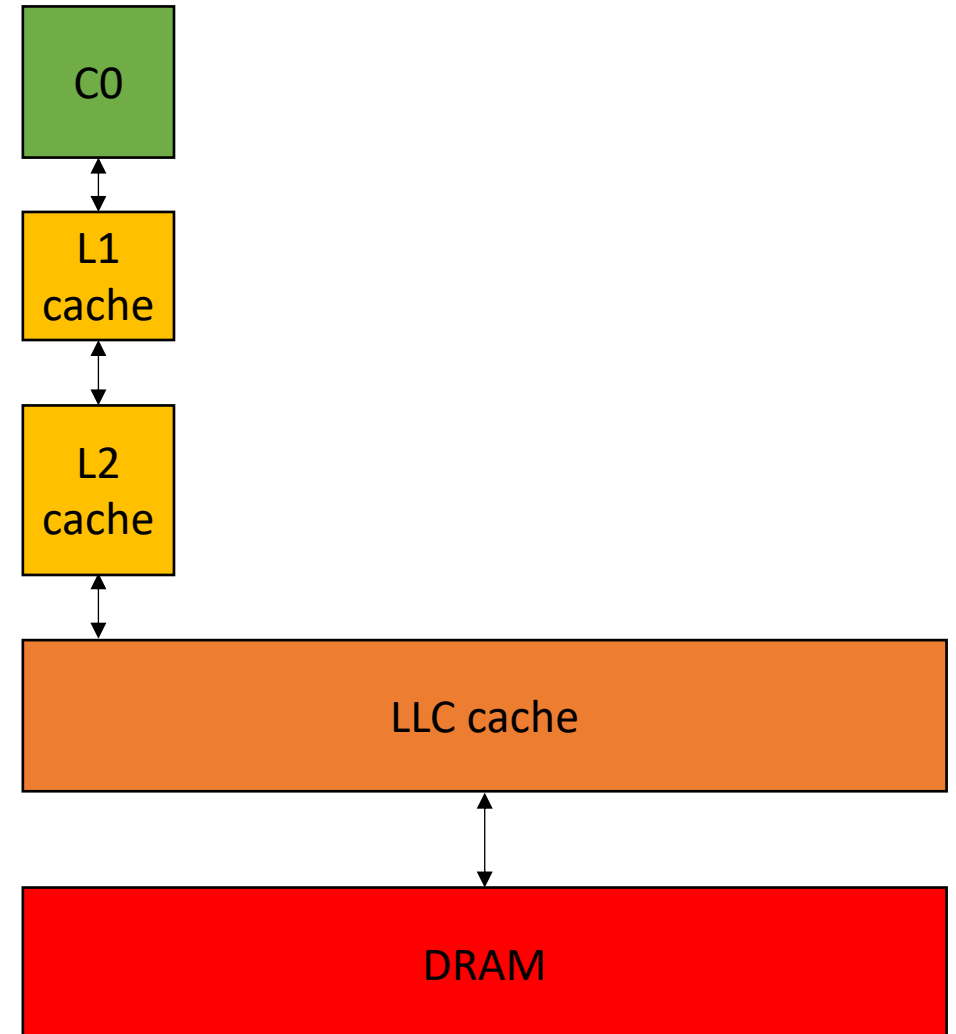
Caches



Caches

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

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



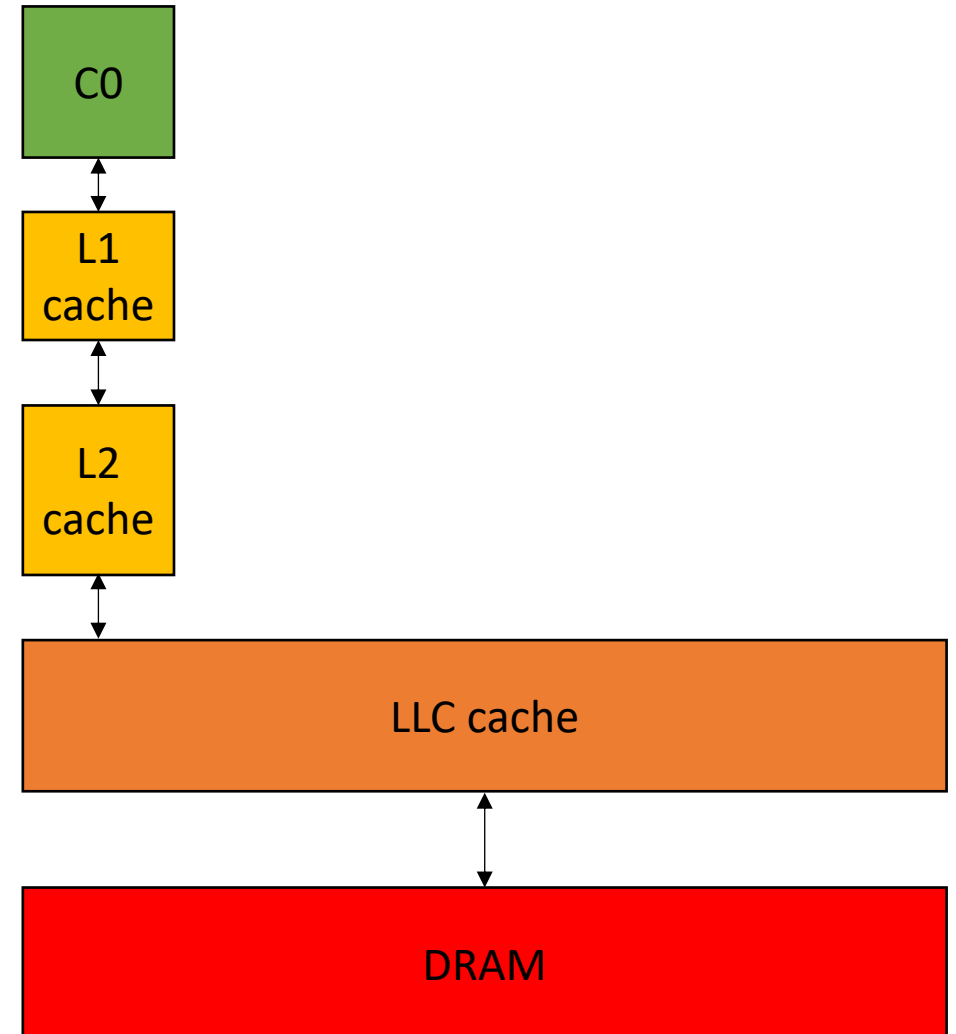
Caches

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

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

4 cycles

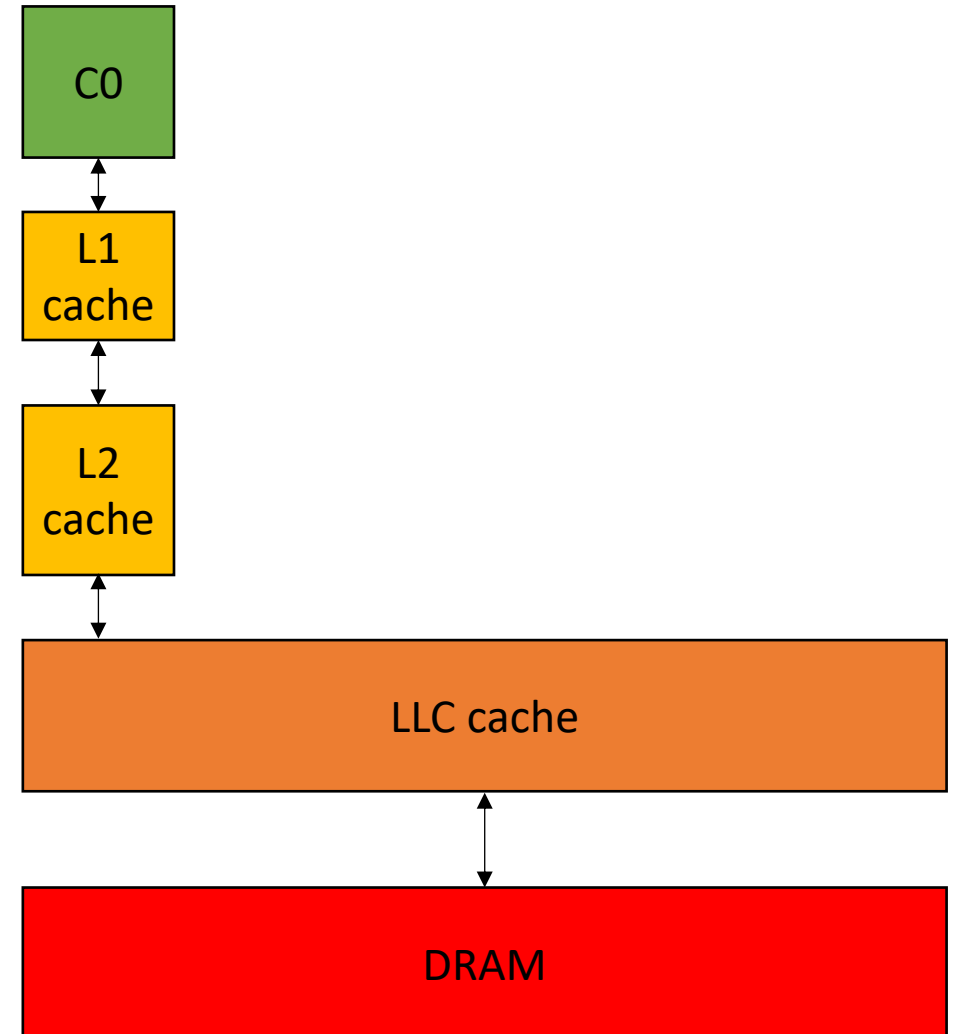
Assuming the value is in the cache!



Caches

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

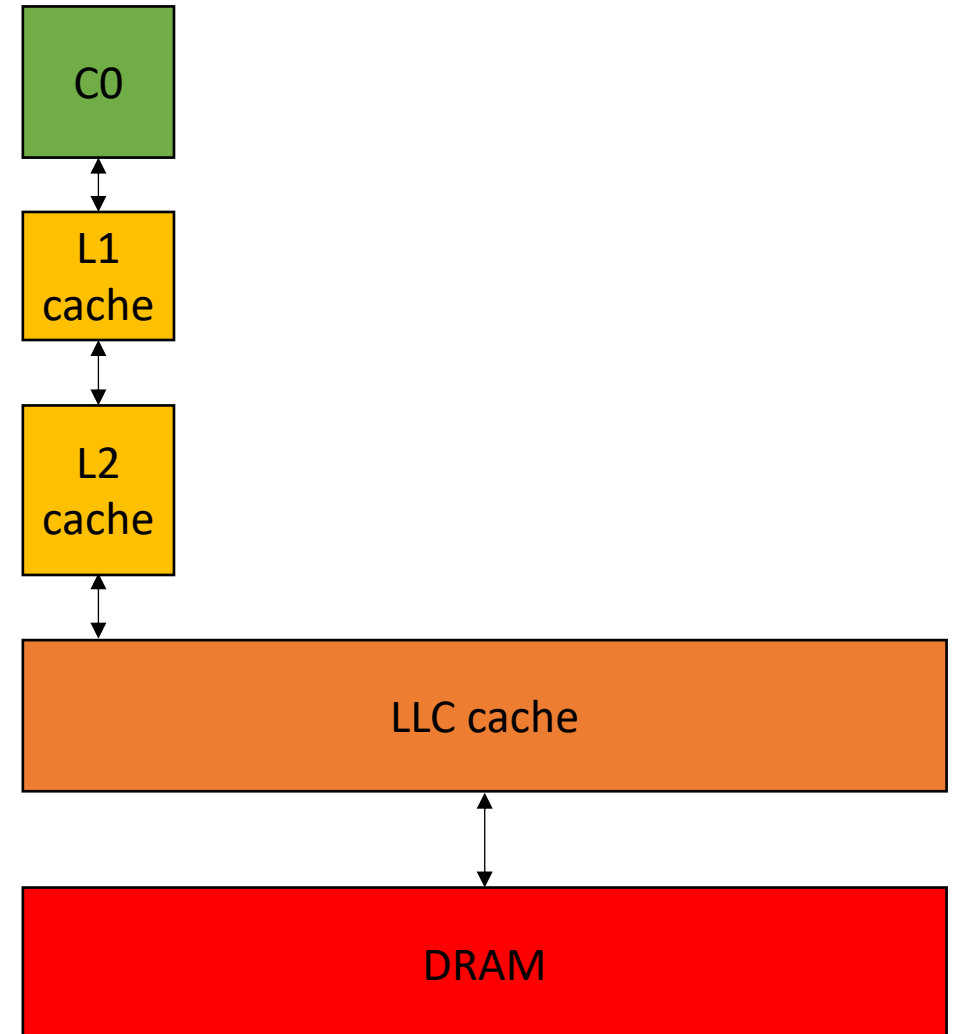
```
%5 = load i32, i32* %4    4 cycles  
%6 = add nsw i32 %5, 1    1 cycles  
store i32 %6, i32* %4
```



Caches

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

<code>%5 = load i32, i32* %4</code>	4 cycles
<code>%6 = add nsw i32 %5, 1</code>	1 cycles
<code>store i32 %6, i32* %4</code>	4 cycles

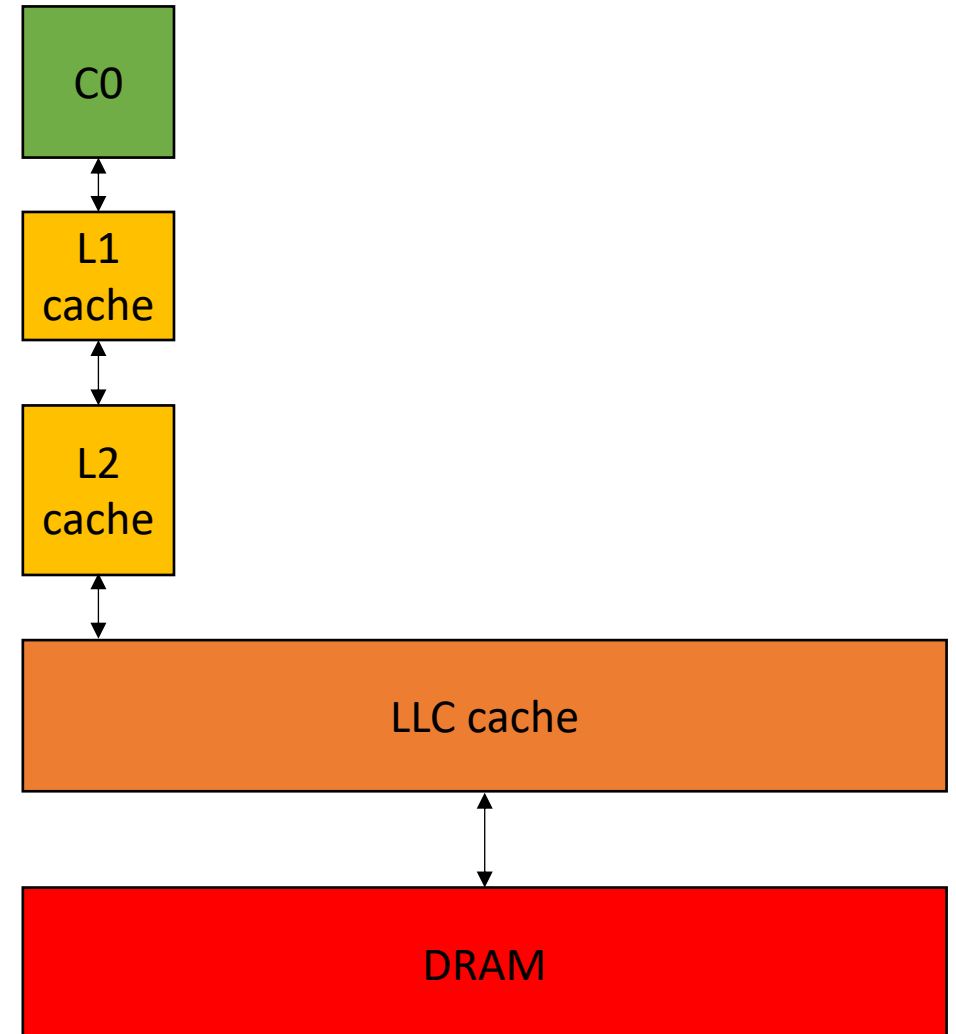


Caches

```
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!



Next lecture

- Cache associativity
 - Cache coherence
 - False Sharing
-
- Homework 1 released on Friday
 - Make sure to do the quiz!!!