CSE113: Introduction to Parallel and Concurrent Programming

UCSC CSE

https://techwireasia.com/2020/06/japans-fugaku-is-the-worlds-fastest-supercomputer/
https://www.lenovo.com/
https://www.apple.com

Enrollment

- Class is full with a large waitlist.

- In the past I have opened the waitlist, with the condition that we've had at least 1 TA per every 40 students.
- We have 1 TA for every 60 students as it stands right now.

- Because of this, I may not be able to let any more students into the class. It is already a high ratio and I may not be able to make it higher.

- I will provide accommodations for those that get in late. So please stay on the waitlist, but no need to keep coming to class, etc.

- I'm happy to let as many people audit the class as want to.

Hello!

- Professor Mohsen Lesani (he/him)

  Associate Professor

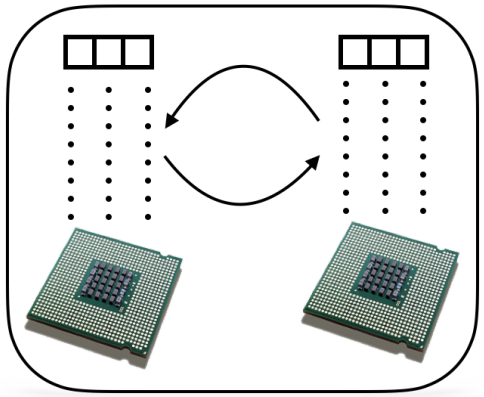
- Previously
  - Associate Professor at UCR
  - Postdoc at MIT
  - PhD at UCLA

- https://mohsenlesani.github.io/

Research Interests

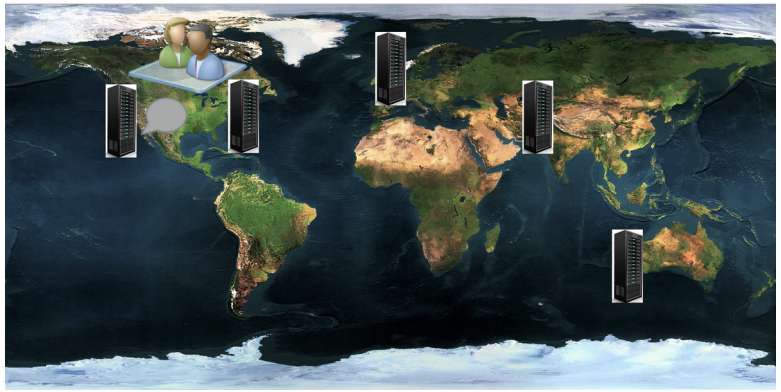
PhD: UCLA

Transactional Memory and Concurrent Data Structures



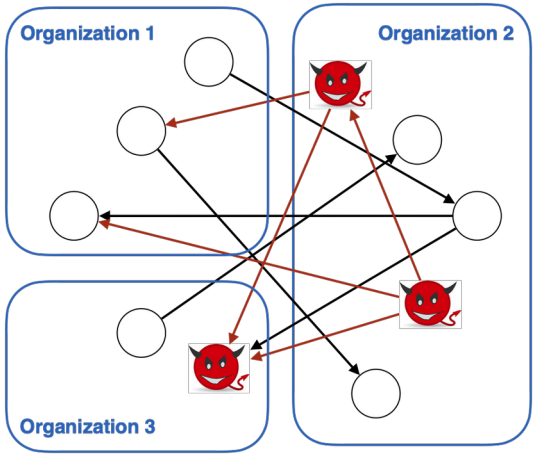
Postdoc: MIT

Distributed Data Stores



UCR

Synthesis and Verification of Reliable and Secure Distributed Systems



Research Interests

- Secure distributed systems
  - Heterogeneous Quorum Systems
  - Resilient Partitioning and Replication
  - Blockchain cross-chain transactions
- Automatic analysis and synthesis of distributed data stores
  - Automatic Synthesis of Replicated systems
  - Integrity, Convergence and Recency
  - RDMA and FPGA
- Verification of distributed systems
  - Interactive Theorem proving
  - Semantics and Program Logics
- Data analytics
  - Declarative Graph Analytics
- Concurrent Systems
  - Automatic Fence Insertion
  - Verification of Concurrent Data Structures

Concurrency and Parallelism is everywhere!

Fujitsu SC at Riken (Japan)

7.6M cores

Consumer Laptop

2-16 cores

Mobile Phone

2-8 cores

Watches?

1 core

BUT

still need to worry about concurrency!

https://techwireasia.com/2020/06/japans-fugaku-is-the-worlds-fastest-supercomputer/
https://www.lenovo.com/
https://www.apple.com

Concurrency and Parallelism is everywhere!

In many cases you won't know what hardware you are programming for

web apps

Android apps

You still need to worry about concurrency!

https://techwireasia.com/2020/06/japans-fugaku-is-the-worlds-fastest-supercomputer/
https://www.lenovo.com/
https://www.apple.com

People have a variety of interests

- What are some of yours?
- Non CS topics?
- CS topics?

Parallel programming concepts 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

source: IEEESpectrum

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

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Some examples

Other applications are harder to make go fast

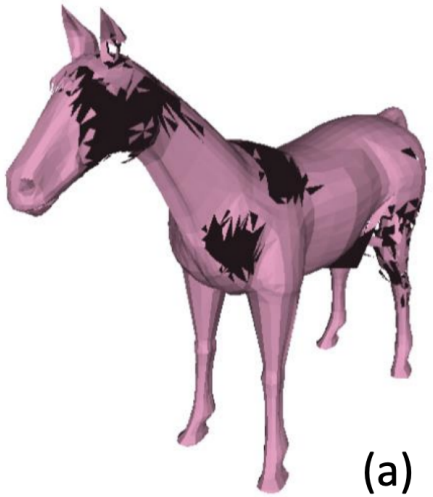
This computation is known as the “Local Laplacian Filter”. Requires visiting all pixels 99 times

simple parallelism is 2x slower than finely tuned parallelism

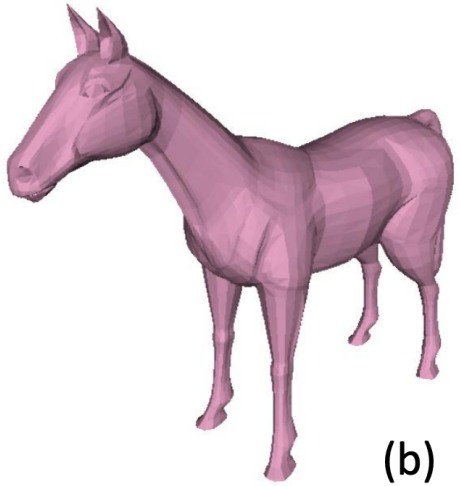
from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Some examples

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



(a)



(b)

(c)

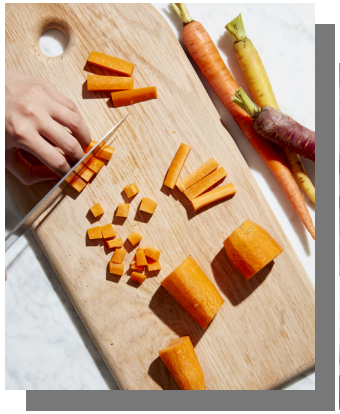
(d)

Parallel programming concepts apply to lots of other non-CS topics 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

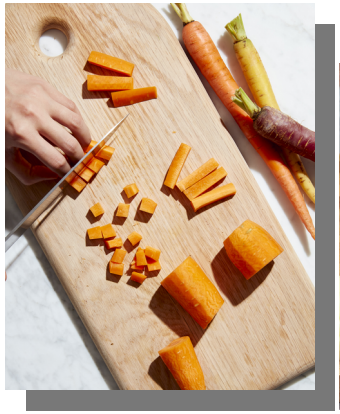
Cooking

How to cook with two people?

time

Simple soup recipe

Chop Carrots



Chop Potatoes

Combine and cook

Cooking

How to cook with two people?

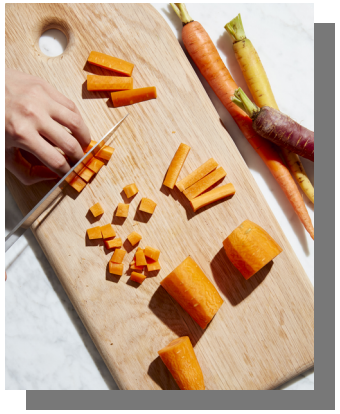
time

Simple soup recipe

Chop Carrots

Chop Potatoes

Combine and cook



Learning Objectives

- Foundations of concurrent/parallel computing
  - **Concepts**, not languages/frameworks!
  - Allows you to pick up future new languages and frameworks quickly

- Think in concurrency
  - understand common synchronization idioms and their performance characterizations
  - efficiently (and safely) utilize modern systems

- Shared memory concurrency
  - Many concepts apply to other domains, but likely have different performance characteristics (e.g. distributed systems)
  - Emphasis on thread cooperation

Class information

- Important to go over class organization and structure
  - You want to know this information.
  - All information is also on the class website that we keep updating.

- Future lectures will be more visual

Today’s Schedule

- Class Structure
- Class Contents
- Assignments, Tests, Grades
  - Special note on new AI tools

Class size

- This is a large class: 120 students!
- We have help:
  - 2 grad TAs:
  - Undergrad graders/tutors
- We will need lots of organization and structure to make it through the quarter smoothly.
- We are continuing to develop new class material, structure, and frameworks to aid with scaling. I appreciate your understanding and patience.

Teaching Staff Introductions

- Grad TAs:
  - Jessica Dagostini
    - PhD student working on HPC applications
    - 3nd time TA'ing this class
  - Gurpreet Dhillon
    - MS student working on GPU synchronization

Teaching Staff Introductions

- Undergrad tutors and graders
    - Working on this! Please let me know if you have anyone in mind!

Teaching Staff Introductions

- They are all awesome! And they are passionate about parallel programming! Please utilize their office hours and tutoring as much as possible!

- They will be your primary point of contact for technical help throughout the quarter

Class Resources

- **Public**: https://mohsenlesani.github.io/slugcse113/
  - Slides, schedule, resources

- **Private**: Canvas
  - Homeworks, grades, exams, announcements, recorded lectures, zoom links

- **Private Class forum:**
  - Piazza - invite link incoming

Required Background

- **CSE 12**
  - Assembly and hardware
- **CSE 101**
  - Data structure specifications (Queues and Stacks)
  - Reasoning about algorithms (Space and time complexity)
- **CSE 120 (recommended)**
  - Caches

If you did not have architecture, please consult the architecture reference on webpage!

Useful Background

- **CSE 13S**
  - C programming and unix command line
- **CSE 130**
  - Basic Concurrency

Required Skills

Because this is an upper division class, I do expect a general CS foundation. For the homeworks, I will assume that you are:

- comfortable using a linux command-line
- programming in a high-level language (e.g. Python)
- programming in a low-level language (e.g. C)
- a high-level understanding of computer architecture
- a basic ability to use Github
- a basic ability to use Docker

Class Format

- Tuesdays Thursdays 01:30 - 03:05pm**: 95 minutes**
  - Class is generally structured as follows:
    - Announcements (homework assignments, etc.)
    - Quiz review
    - Previous lecture review
    - New material

- Please be engaged and participate in class! questions, comments, corrections, etc. are all welcome!

Class format

- This is an in-person synchronous class
  - I expect you to make an effort to attend the synchronous lecture
  - Please participate in class and network with your classmates

Class format

- But please don’t come to class sick!
- I plan to record lectures and release through Yuja.
- Watch the lecture and then do the associated quiz.

Class format

- This class is designed to be a synchronous in-person class. Lecture recordings are not meant to be an equal substitute for live class. They are meant to fill in occasionally.

- If you do not plan to attend the majority of the classes in-person, please talk to advising to find a class better suited for you.
  - Inevitably, students that do not attend lectures do not do as well in the class

- If synchronous attendance drops significantly, then we will **revisit attendance grading and lecture recordings.**

Office Hours

- **My office hours:**
  - 3:10 – 5:10 PM on Thursdays
  - Please be prepared with your questions to make sure the time is available to others as well.

Office Hours

- The TAs and tutors will organize their office hours by the end of this week and we will announce the days/times to the web page.
- We will strive to get a mix of both hybrid and in-person throughout the week

Asynchronous Discussion

- **Piazza**
  - Private message (to teaching staff) technical homework questions, sensitive questions
  - Programming and framework questions (global). Please try to help your peers!
  - Tech news (global)
  - Discussions on class material (global)
  - We will try to answer in 24 hours.

- Email
  - Please try using Piazza. Please send direct emails only for sensitive issues.
  - Do not expect replies off-hours (after 5 pm, weekends, holidays)

Asynchronous Discussion

- **Additional forums**
  - You are welcome to create one yourselves
  - Please make it open and available to all your classmates
  - Please provide sufficient moderation (e.g. be nice to each other!)
  - Do not cheat
  - Please remember that anything that is not in Canvas may not be private

  - If there are issues, please let me or a TA know!

Class Content

Class Content

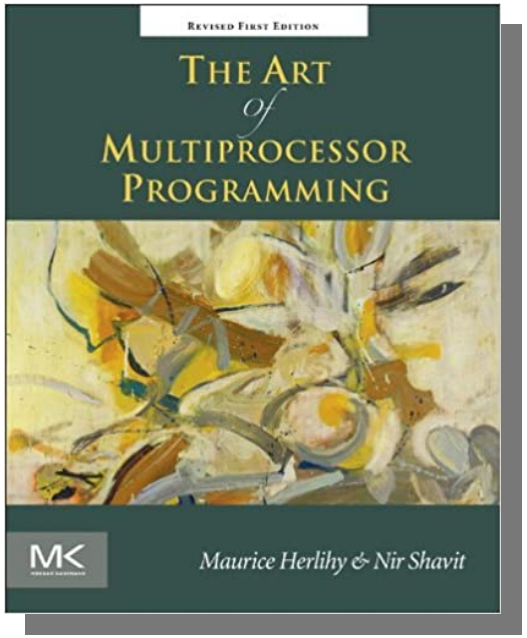
- **20** classes, split into
- **5** modules, so there are
- **~4** classes per module

- **Reference book:**

Available online from the library

Link on the webpage

Book uses Java: we will use C++



Class Content

- **Module 1: Introduction, Background and ILP**

This module will provide an architectural refresher and discuss how modern hardware exploits parallelism within a thread (ILP). We will also introduce threading in C++.

Class Content

- **Module 2: Mutual Exclusion**

This module will discuss the fundamental problem of mutual exclusion. We will discuss the theory behind mutual exclusion, how it is implemented in practice, and specialized mutual exclusion objects.

Class Content

- **Module 3: Concurrent Data Structures**

This module will discuss concurrent objects and how to reason about them. We will discuss several implementations and discuss how it can be used in load balancing and software pipelining.

Class Content

- **Module 4: Parallel Programming on GPGPUs**

This module will discuss general purpose (GP) GPU programming. We will discuss the SIMT programming model, hierarchical execution, and different architectural considerations when optimizing programs.

- *I'm very excited for this module! We want to use a new platform (WebGPU) to allow everyone to program the GPUs on their own machines!*

Class Content

- **Module 5: Advanced topics**

This module will discuss advanced topics, including memory consistency, subtle concurrent data structures, and fairness.

Class Content

- **Schedule:**

https://mohsenlesani.github.io/slugcse113/#schedule

Readings are highly recommended; they will be a useful reference for test studying and homeworks

Slides will be uploaded before lecture

Accessibility

UC Santa Cruz is committed to creating an academic environment that supports its diverse student body. If you are a student with a disability who requires accommodations to achieve equal access in this course, please submit your Accommodation Authorization Letter from the Disability Resource Center (DRC) to me by email, preferably within the first two weeks of the quarter. I would also like us to discuss ways we can ensure your full participation in the course. I encourage all students who may benefit from learning more about DRC services to contact DRC by phone at 831-459-2089 or by email at drc@ucsc.edu.

Assignments and Tests

Assignments

- Five assignments, one assignment per module
- Each homework is worth 10% of your grade (total of 50%)
- Released halfway through the module
- Deadlines announced in the schedule.

- We will try to make homeworks due at midnight. If we receive too many questions off hours, we will move earlier (e.g., 8 PM)
- Please do not expect replies off-hours (after 5 pm, weekends, holidays)

Assignments

- Late Policy:
  - You have 10 days to submit each assignment.
  - Each assignment has 3 days that you can turn in the assignment late with no penalty.
  - No work accepted after the 3 days.

Assignments

- **Format**:
  - Coding assignments in C/++ and Python (and some Javascript/wgsl for module 4)
  - It is recommended that you have access to a machine with at least 4 cores.
  - TAs will provide Docker image and autograding instructions.

  - We aim to use github classroom for submission and automatic feedback.
  - You will be graded on the server feedback rather than the results from your own machine. This is to help provide fair (and scalable) grading across the increasing diversity of devices that everyone has these days. Someone with an Apple M-series processor will get very different results than someone with an Intel X86 processor.
  - *Architectural differences are very interesting to discuss and I hope we can have detailed discussions about how your machine's results differ from the server on Piazza*.

Academic Integrity

For assignments:

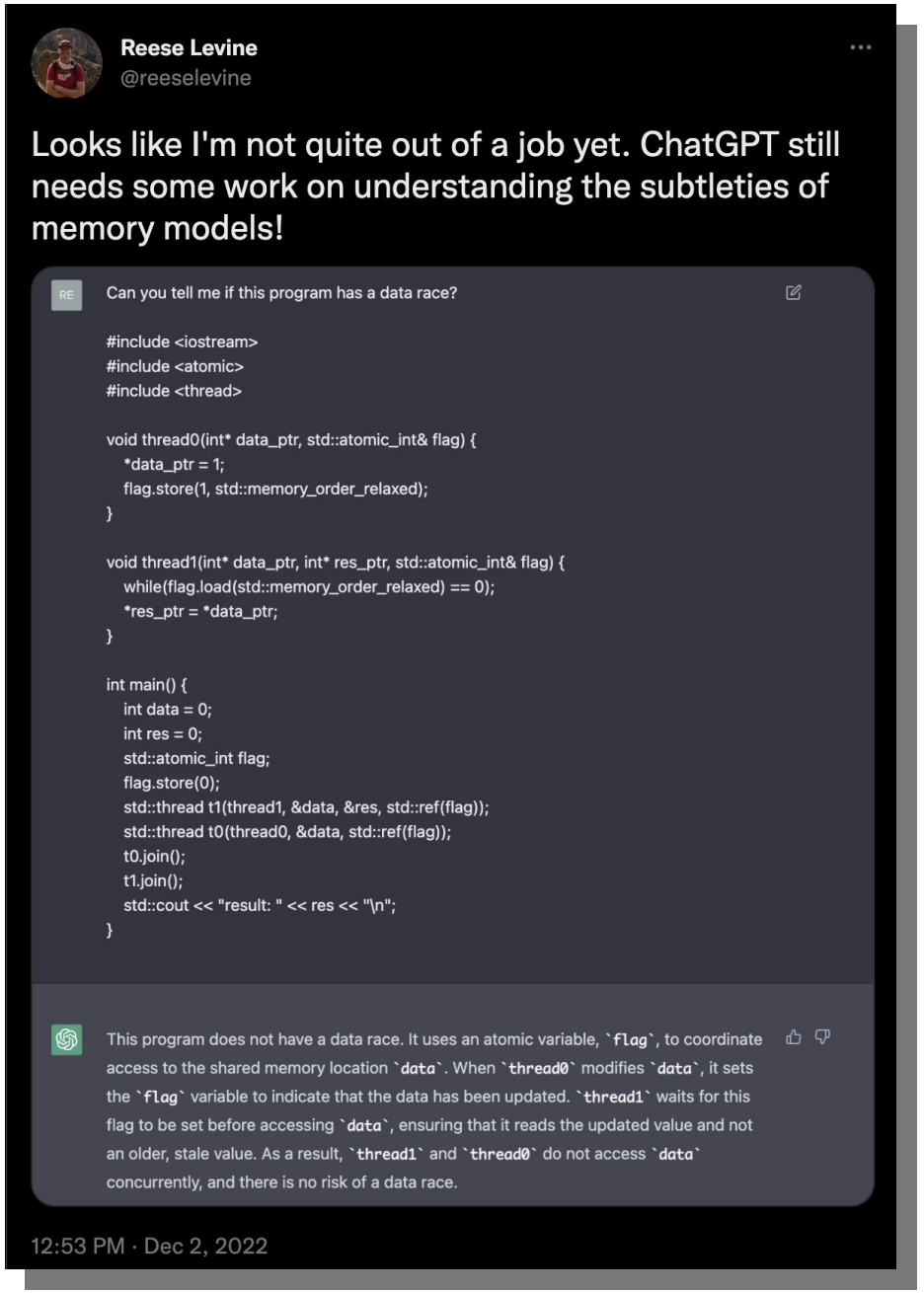
I expect submitted assignments to contain your own original work. You can refer to notes, slides, internet, etc. But do not blindly copy code.

Any part of your submission that is not your original work (e.g. code snippets from the internet) need a citation. My aim is to be lenient with cited code, but we may remove some points based on the extent. A few missing points is better than a referral for academic misconduct.

AI Tools

- Exciting time for AI:
  - Github co-pilot
  - ChatGPT
- Impact on learning objectives is not clear.
- This class has been designed to be taken *without* the use of AI tools.
  - Please do not use them for this class
  - They inhibit you from learning the foundations of parallel programming
  - If we suspect these are being used widely then we may implement random audits.

AI Tools



Reese Levine
@reeselevine

Looks like I'm not quite out of a job yet. ChatGPT still needs some work on understanding the subtleties of memory models!

RE Can you tell me if this program has a data race?

```
#include <iostream>
#include <atomic>
#include <thread>

void thread0(int* data_ptr, std::atomic_int& flag) {
  *data_ptr = 1;
  flag.store(1, std::memory_order_relaxed);
}

void thread1(int* data_ptr, int* res_ptr, std::atomic_int& flag) {
  while(flag.load(std::memory_order_relaxed) == 0);
  *res_ptr = *data_ptr;
}

int main() {
  int data = 0;
  int res = 0;
  std::atomic_int flag;
  flag.store(0);
  std::thread t1(thread1, &data, &res, std::ref(flag));
  std::thread t0(thread0, &data, std::ref(flag));
  t0.join();
  t1.join();
  std::cout << "result: " << res << "\n";
}
```

This program does not have a data race. It uses an atomic variable, `flag`, to coordinate access to the shared memory location `data`. When `thread0` modifies `data`, it sets the `flag` variable to indicate that the data has been updated. `thread1` waits for this flag to be set before accessing `data`, ensuring that it reads the updated value and not an older, stale value. As a result, `thread1` and `thread0` do not access `data` concurrently, and there is no risk of a data race.

12:53 PM · Dec 2, 2022

Discussing results

You cannot share code snippets or discuss coding solutions at a low-level.

However, unless otherwise specified: in the second week of the assignment, you can share local (from your own machine) results with your classmates.

If you collaborate with your classmates, please mention in the submission. Again, a few missing points is better than a misconduct referral.

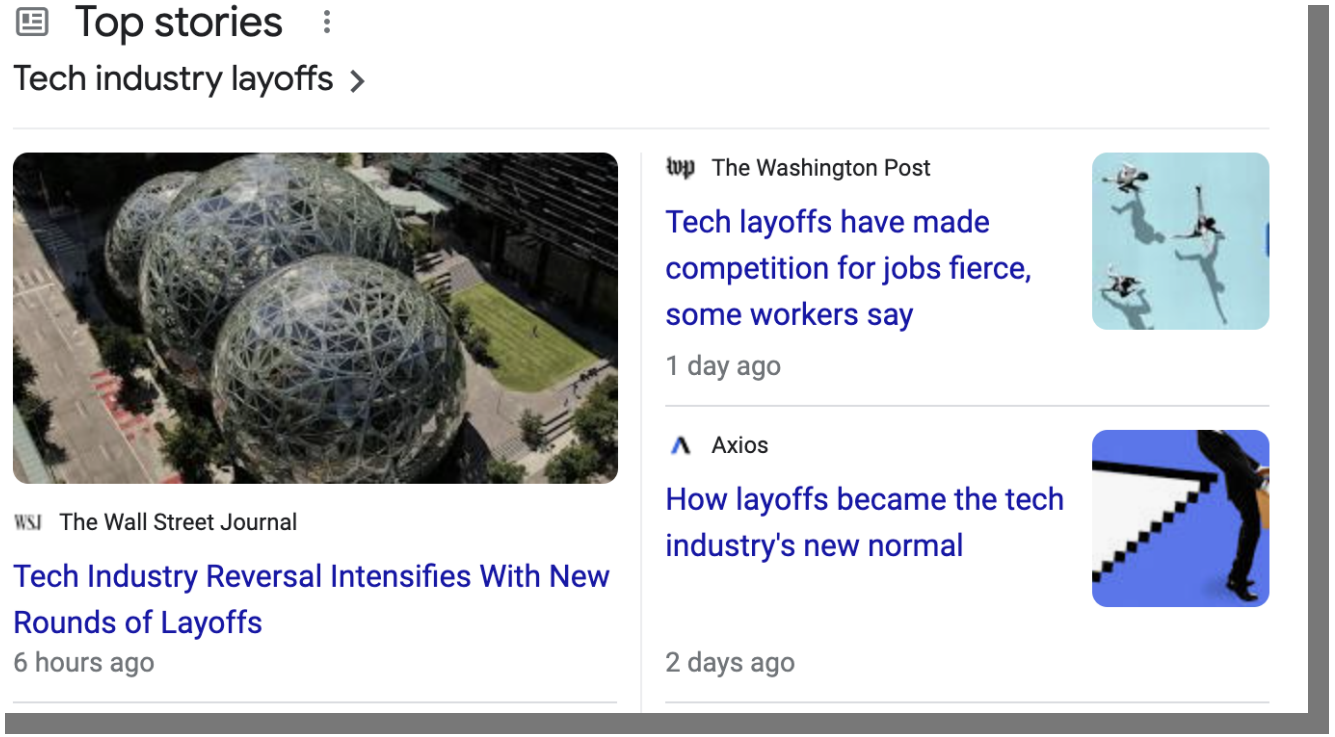
You will have different machines and thus, your results may not align completely: it is interesting to think about why!

Cheating

This class has a zero tolerance policy on cheating. Please don't do it. I would much rather get a hundred emails asking for help than have to refer anyone for academic misconduct.

Cheating harms you: this is the best chance in your career to take the time to really learn the class material. If you do not learn the material you will not be successful in a tech career.

The current economic conditions are volatile for computer science graduates. You will not stand out to a company for having straight As. You will stand out if you can show a deep understanding of complicated CS topics. When you cheat, you deprive yourself of this learning.



Tests

Two tests:

Final and Midterm

Format:

In-person tests

Tests

Midterm

- Data, time will be announced in the schedule. Often halfway through module 3.
- It will be in the class.
- Worth 10% of grade
- *Review slides and readings*

Tests

Final

- Data, time and location will be announced in the schedule.
- Worth 30% of grade
- *Inclusive: slide material from all year, including readings*

Tests

For each test you are allowed 3 pages of notes, front and back, printed or handwritten, you can print slides, etc.

Reviewing Grades

- For assignments and tests:
  - You have 1 week from when the grade is posted to discuss grades with teaching staff

Assignments and Tests

Grade Breakdown:

- 5 homeworks: 50%
- 1 midterm: 10%
- 1 final: 30%
- **attendance/quiz**: 10%

Attendance and Quizzes

- Small canvas “quiz” every lecture - take the quiz to get the daily points
- Quizzes are posted after class and due before the next class. Only submit the quiz once you have watched the lecture (either in person or remotely)! They are meant to test your understanding.
- Quiz answers are not graded! only if you submit it
  - However, low-effort quiz submissions are liable to be failed.
- Some quiz questions do not have a right or wrong answer. They are meant to make you think about the material!
- You can miss up to 3 quizzes without penalty.

Website tour

Final notes

Updates

- There may be updates to HWs and tests
- There may be schedule changes

Thank you!

- I’m happy to have all of you in the class!

- Your experiences and feedback will help shape this class for future students. Email is always open for comments about class material, HW assignments, etc.

Starting on more technical concepts

- **Architecture/Compiler review:**
  - Parallel programming lives at the edge of the software/hardware interface. We will need to understand architecture/compiler basics in order to program efficient and correct programs
  - *Good programming languages for parallel architectures is still an open problem!!*

Architecture and compiler overview

- 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

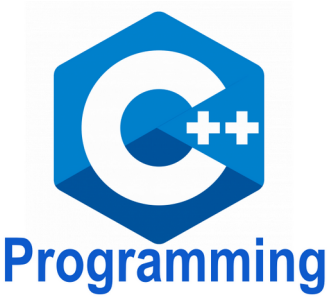
Lecture Schedule

- **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 productivity

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

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

Hardware Designer: Makes Chips

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

The negotiators:
Specifications
Compiles
Runtimes
Interpreters

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

2003

700 MHz

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

2007

2.1 GHz

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

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

2 cores

1.2x increase
over 10 years

2017
2.5 GHz

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 (2024 is basically the same)

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/++
  - has massive ecosystem,
  - has large and active community
  - can keep up with hardware trends and
  - allows extremely efficient code to be written
  - keeping a manageable level of abstraction

C/++ is not perfect

- **Downsides**:
  - out of bound bugs,
  - pointers,
  - security issues,
  - 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 19 (Down from 18 last year, but overall 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

ISA


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

Compilation:

Language

ISA


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

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

Compilation:

Language

ISA


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

Officially defined by the specification

ISO standard: costs $200

~1400 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


```
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


```
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


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


```
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
```

Officially defined by the specification

ISO standard: costs $200

~1400 pages

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

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

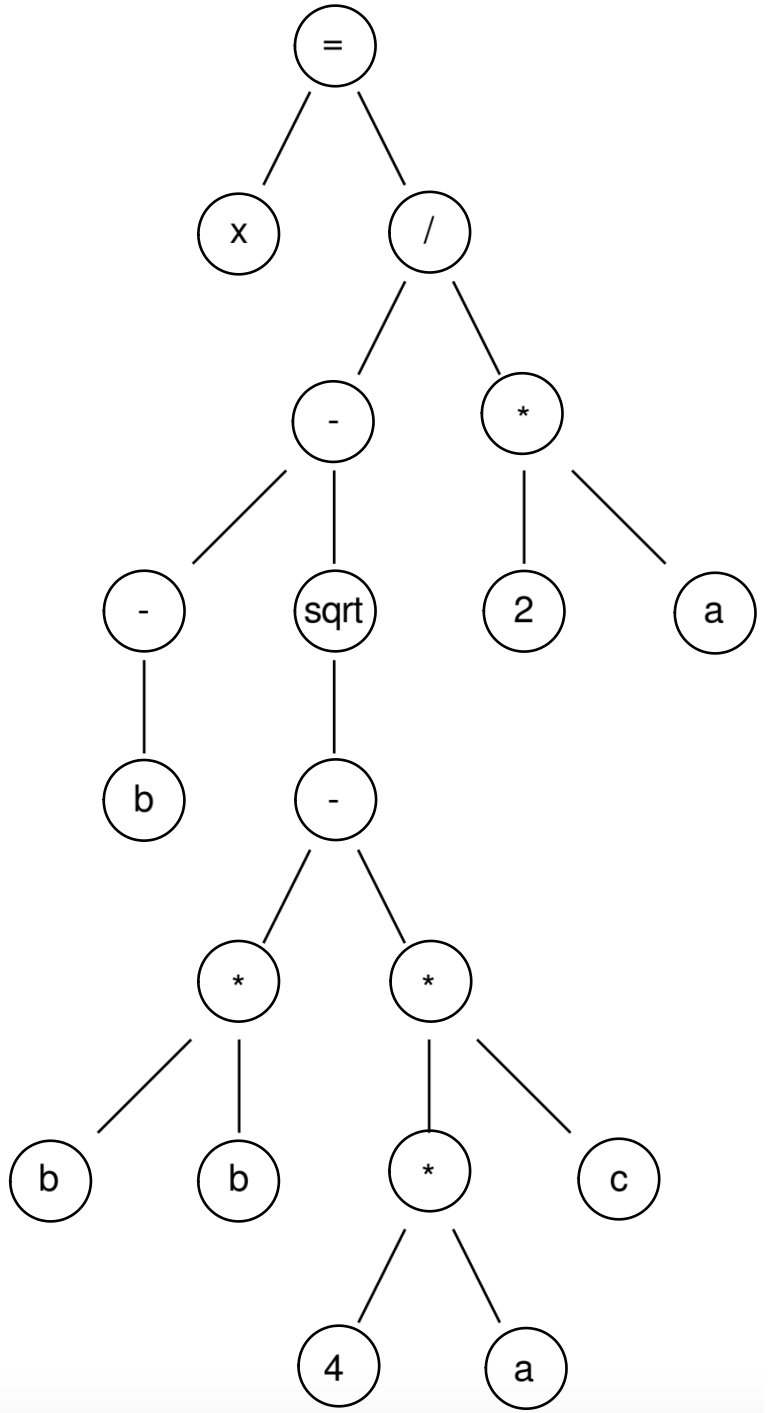
```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

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

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

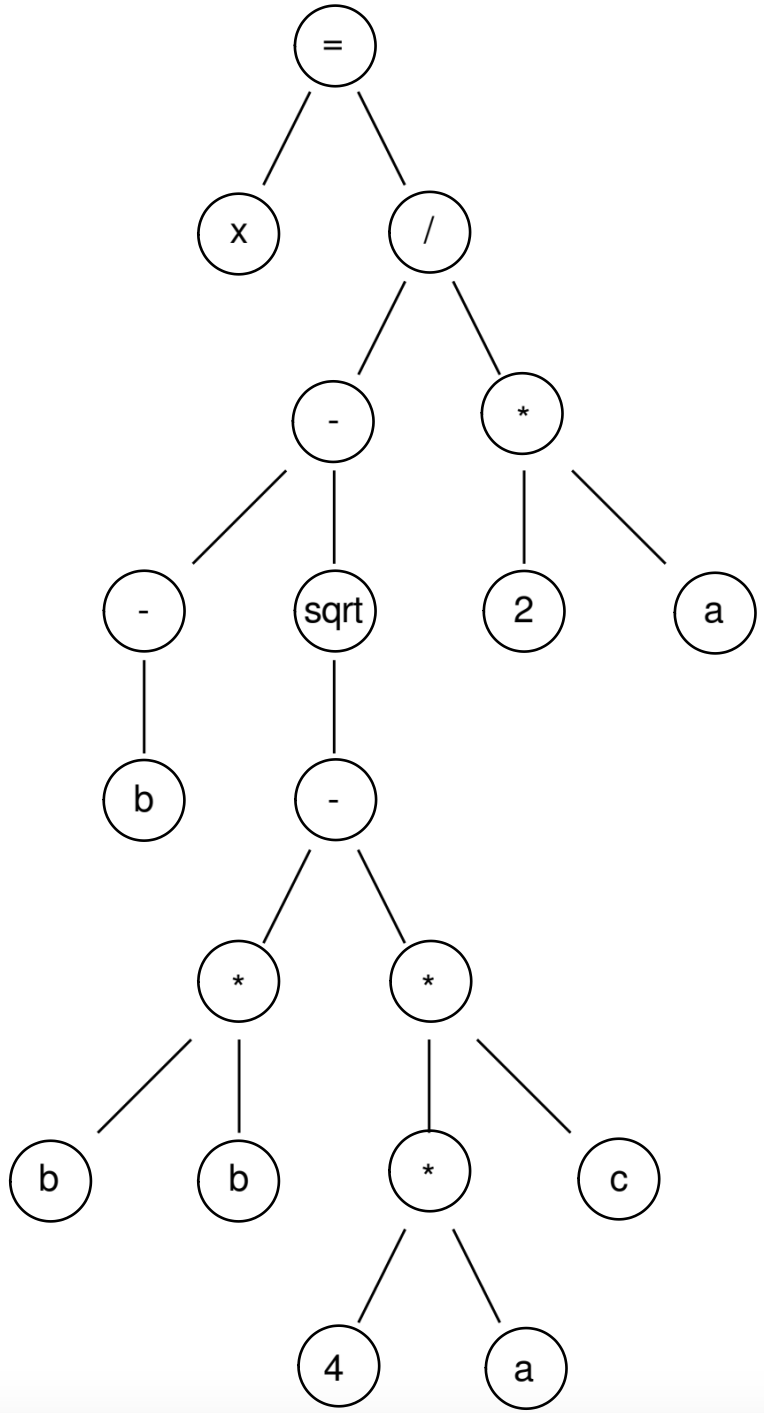
```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

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



Simplify this code:

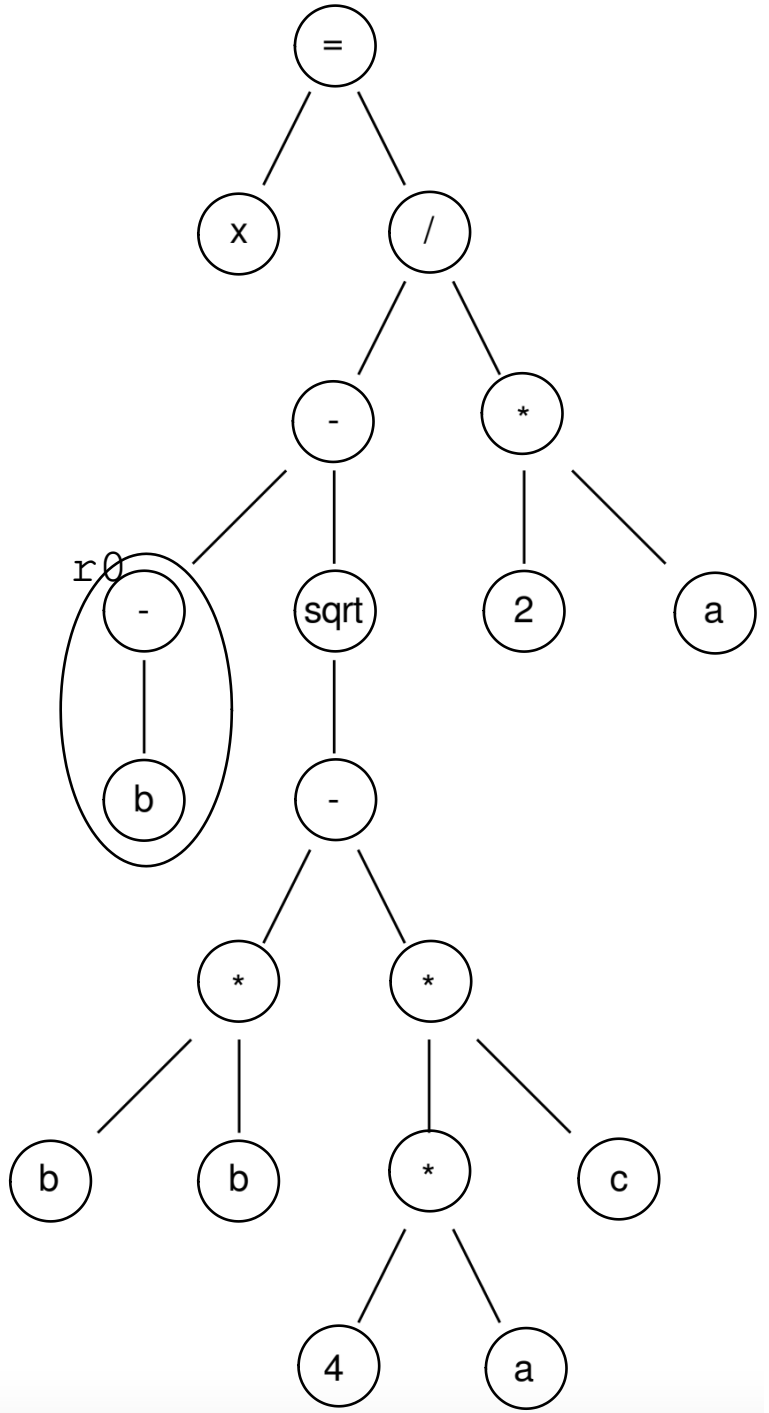
Post-order traversal, using temporary variables



Simplify this code:

post-order traversal, using temporary variables

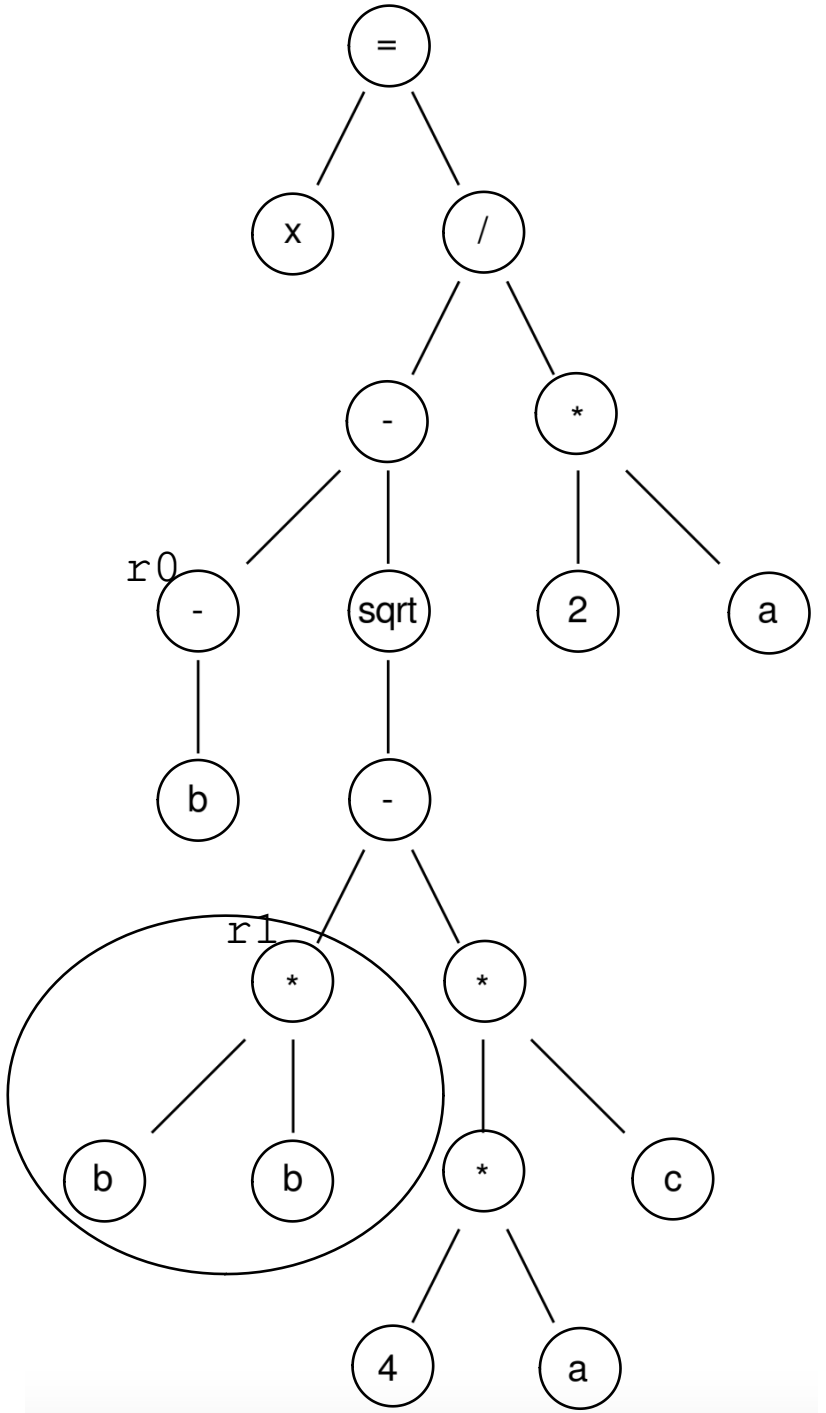
```
r0 = neg(b);
```



Simplify this code:

post-order traversal, using temporary variables

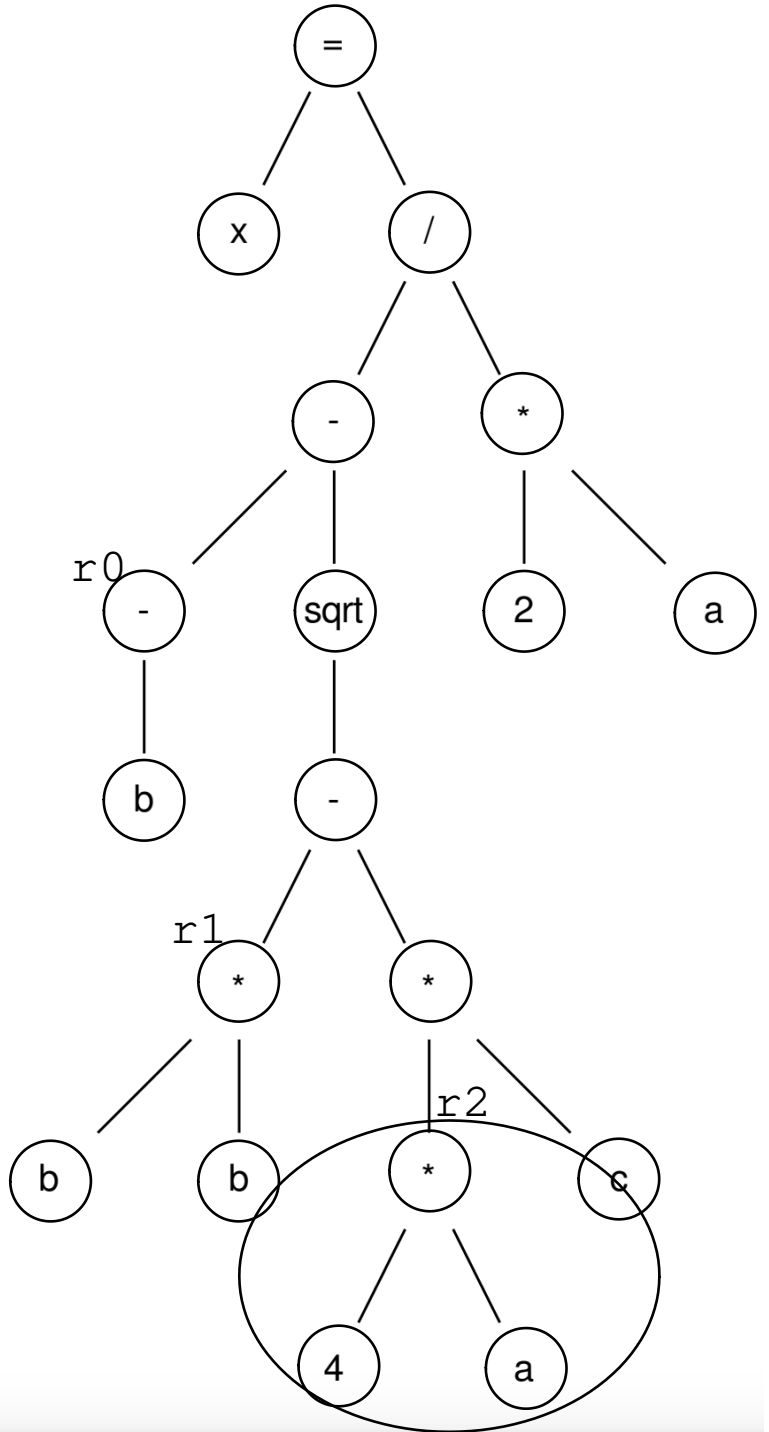
```
r0 = neg(b);
r1 = b * b;
```



Simplify this code:

post-order traversal, using temporary variables

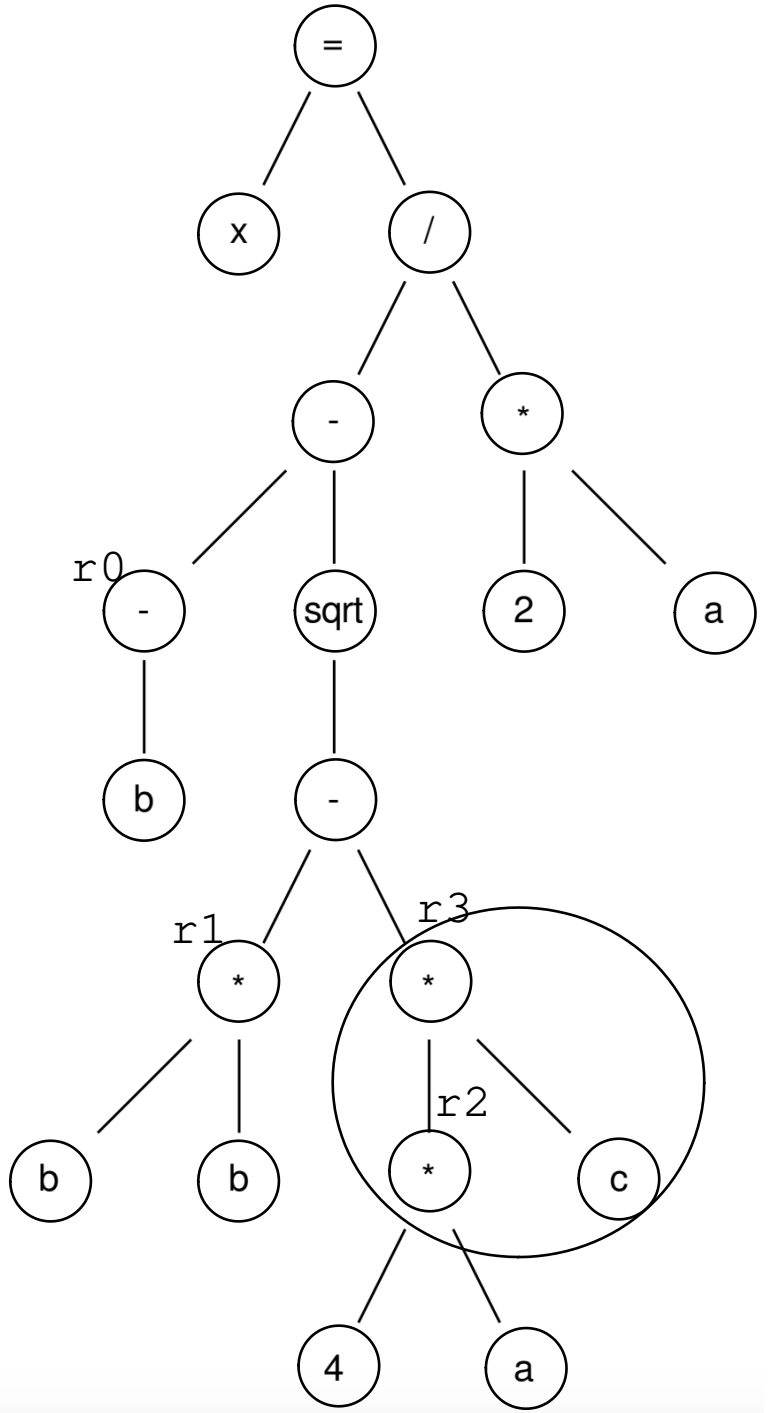
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
```



Simplify this code:

post-order traversal, using temporary variables

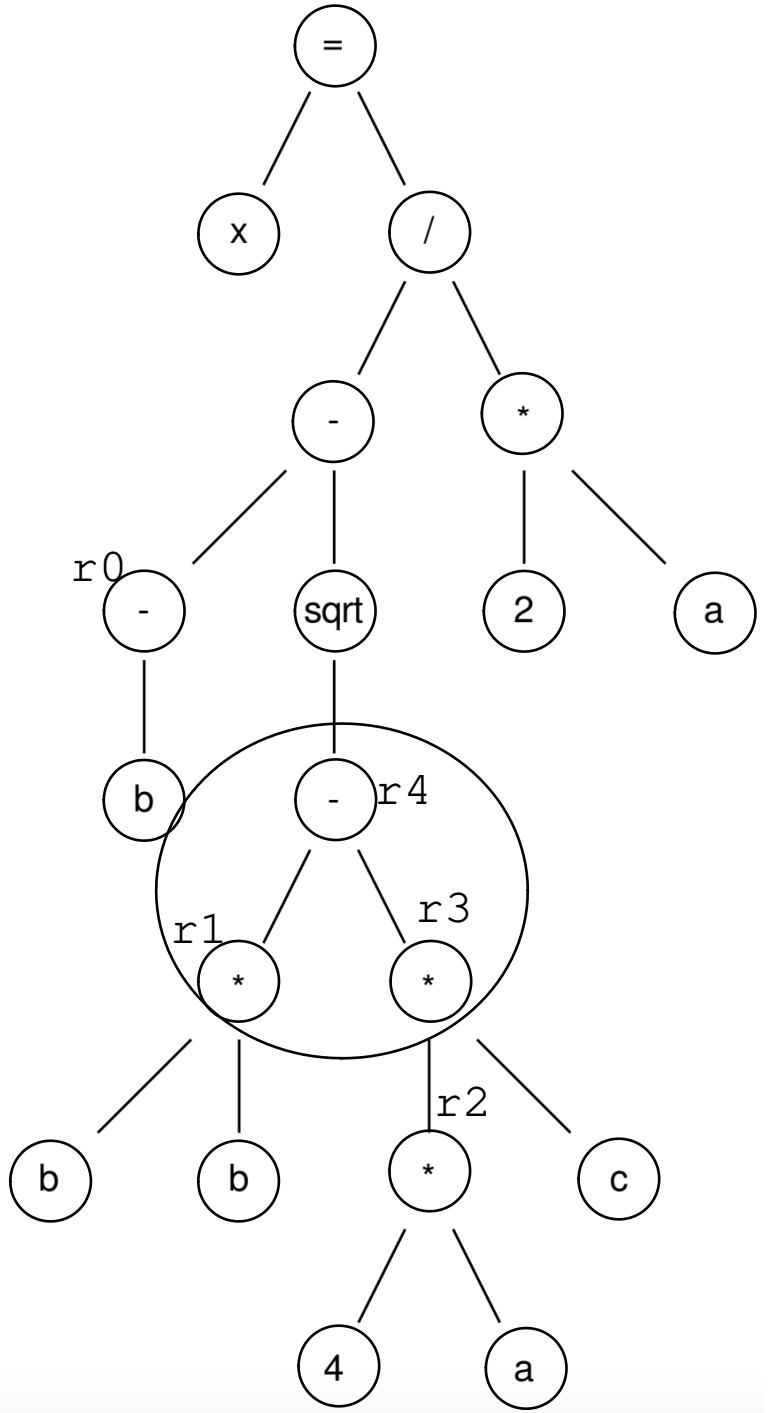
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
```



Simplify this code:

post-order traversal, using temporary variables

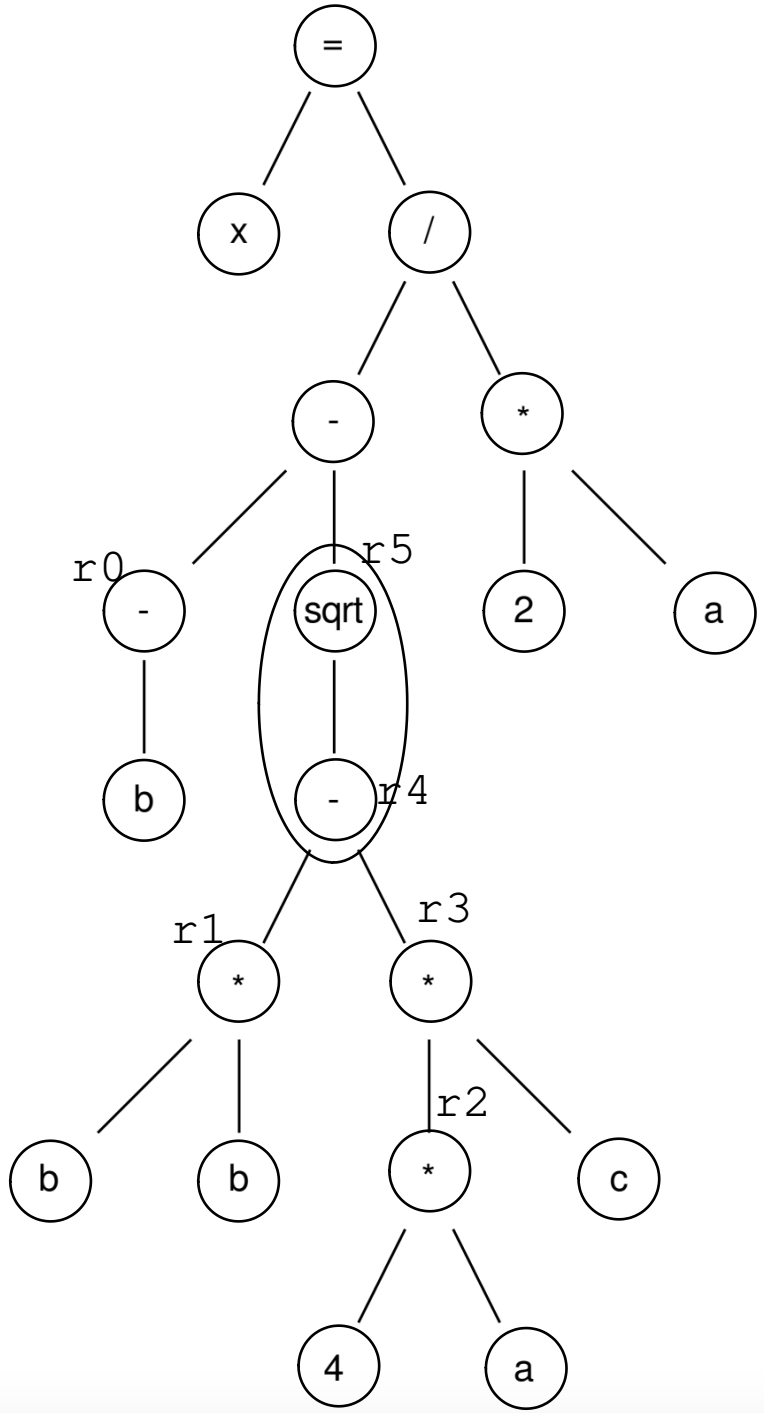
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 - r3;
```



Simplify this code:

post-order traversal, using temporary variables

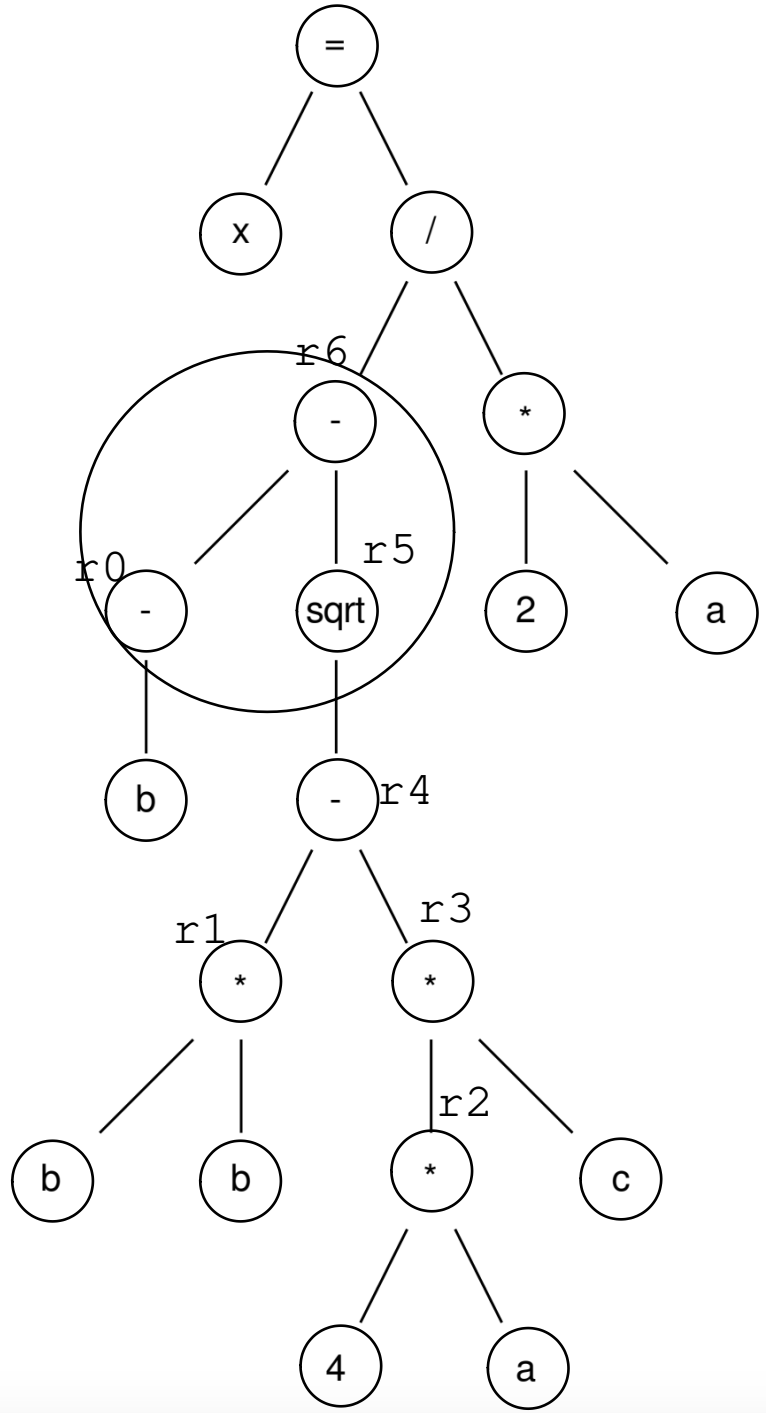
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 - r3;
r5 = sqrt(r4);
```



Simplify this code:

post-order traversal, using temporary variables

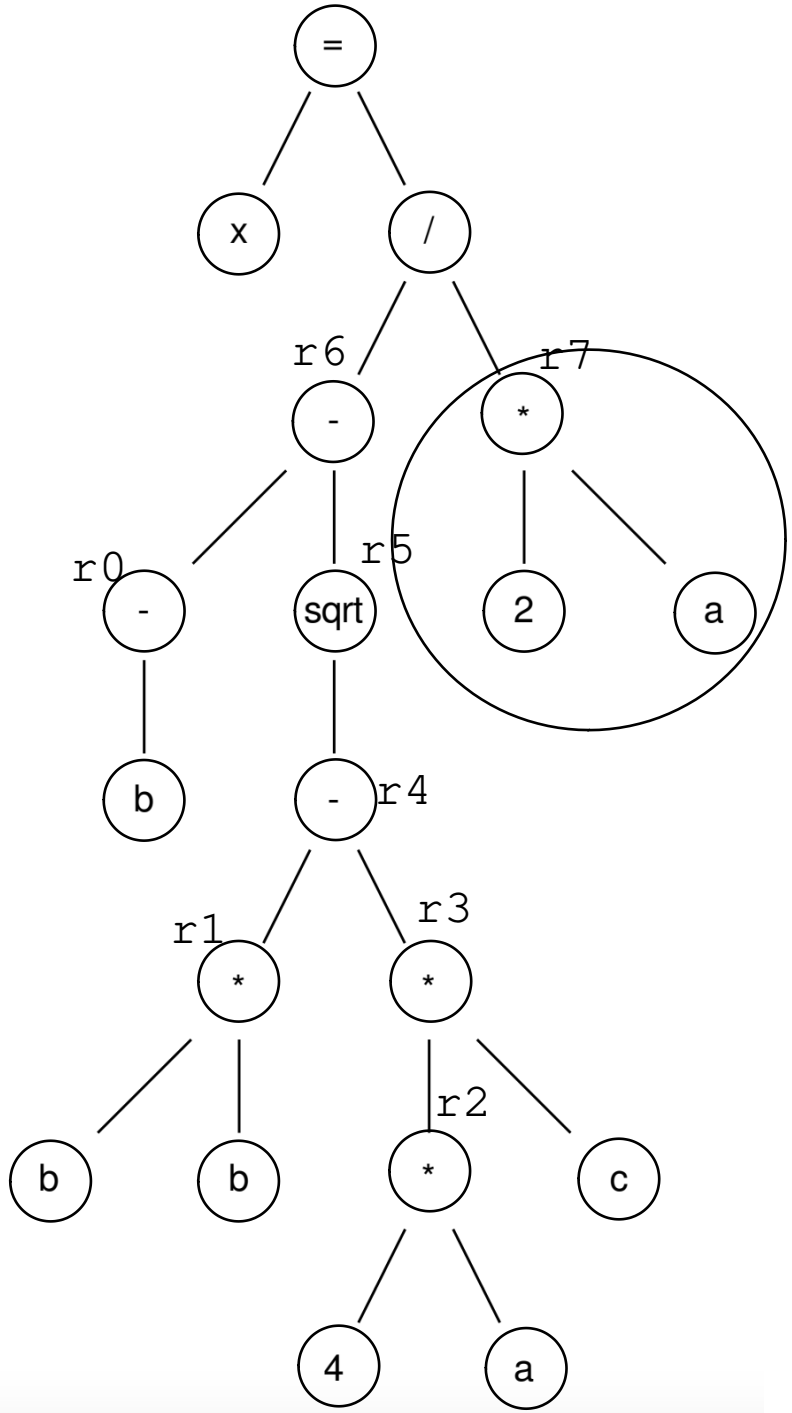
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 - r3;
r5 = sqrt(r4);
r6 = r0 - r5;
```



Simplify this code:

post-order traversal, using temporary variables

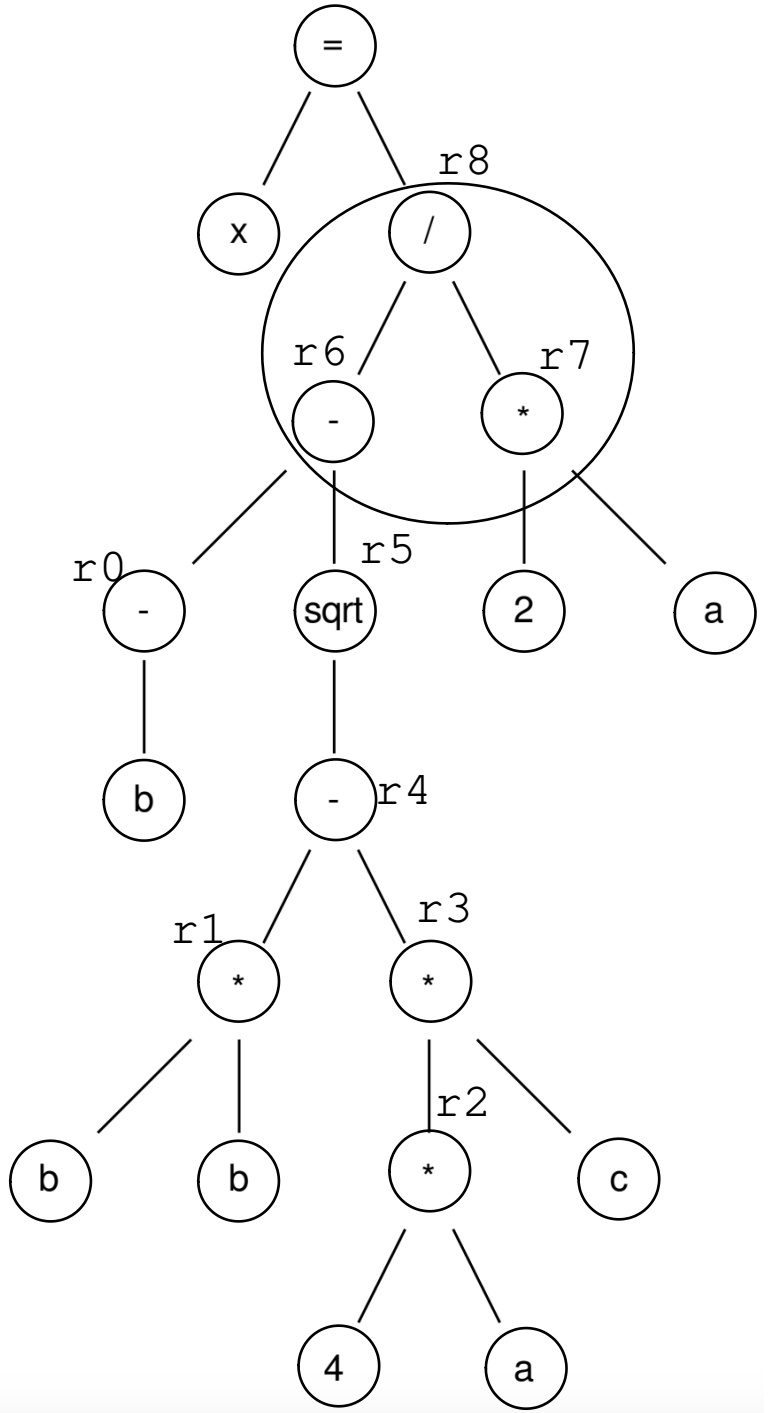
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 - r3;
r5 = sqrt(r4);
r6 = r0 - r5;
r7 = 2 * a;
```



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

```c
// Type your code here, or load an example.
float sqrt(float x);

float add(float a, float b, float c) {
   return (-b - sqrt(b*b - 4 * a * c)) / (2*a);

}

```

A Output... Filter... Libraries Add new... Add tool...

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

C program

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

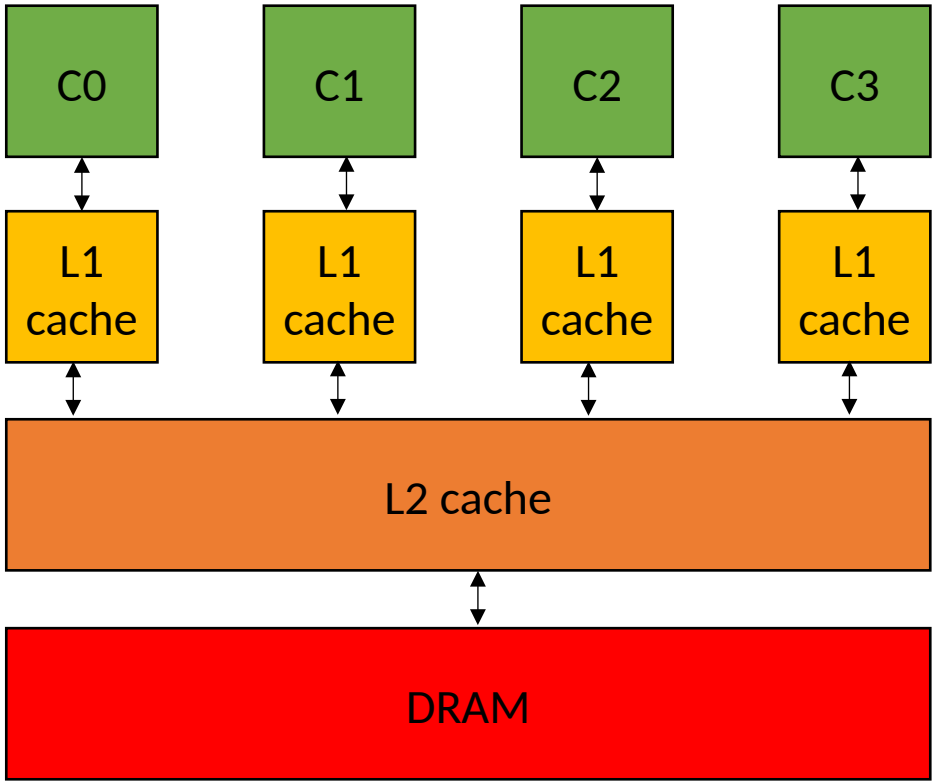
- Overview - why do we need a lecture on compilation and architecture?

- 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

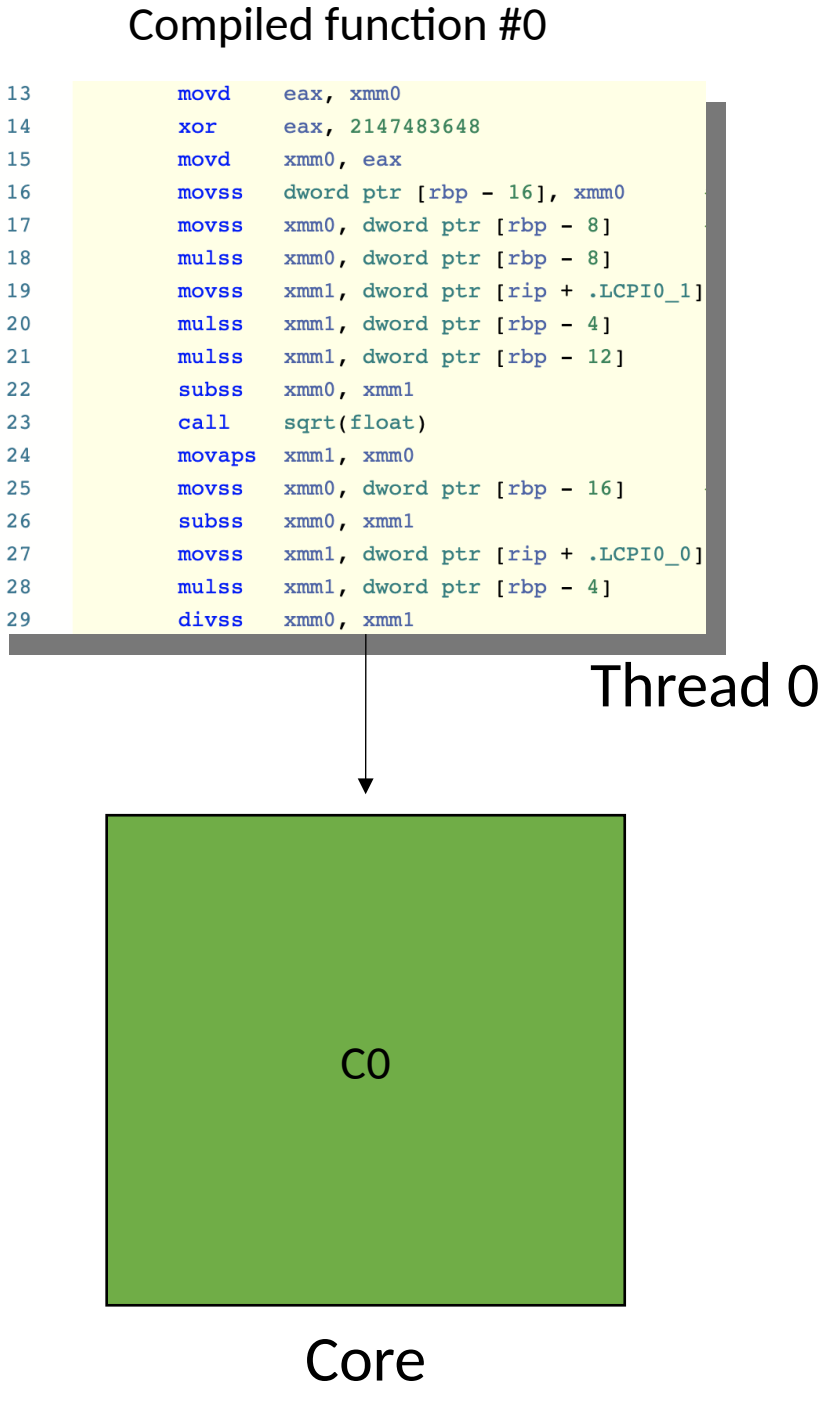


Core

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



Core

Sometimes multiple programs want to share the same core.

Compiled function #0

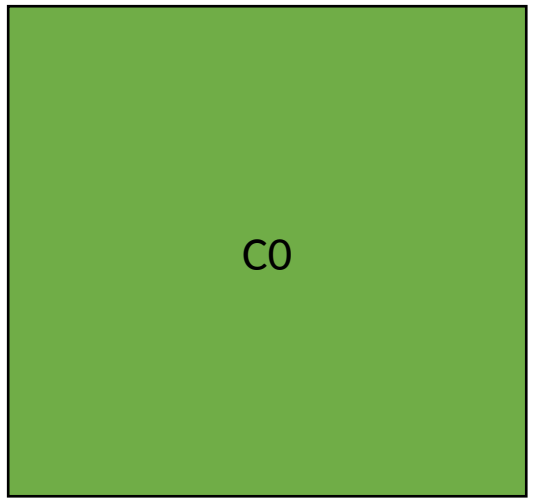
```
13        movd    eax, xmm0
14        xor     eax, 2147483648
15        movd    xmm0, eax
16        movss   dword ptr [rbp - 16], xmm0
17        movss   xmm0, dword ptr [rbp - 8]
18        mulss   xmm0, dword ptr [rbp - 8]
19        movss   xmm1, dword ptr [rip + .LCPI0_1]
20        mulss   xmm1,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

Core

Sometimes multiple programs want to share the same core.

Compiled function #0

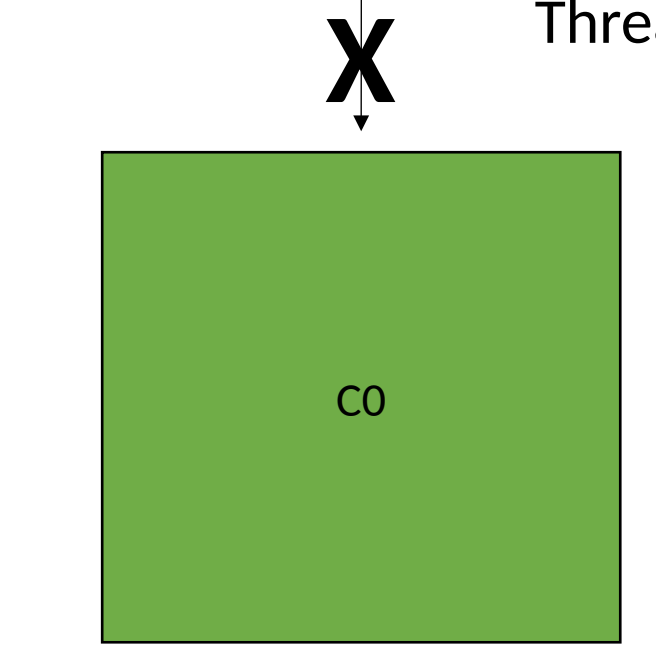
```
13      movd    eax, xmm0
14      xor     eax, 2147483648
15      movd    xmm0, eax
16      movss   dword ptr [rbp - 16], xmm0
17      movss   xmm0, dword ptr [rbp - 8]
18      mulss   xmm0, dword ptr [rbp - 8]
19      movss   xmm1, dword ptr [rip + .LCPI0_1]
20      mulss   xmm1, dword ptr [rbp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

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
```

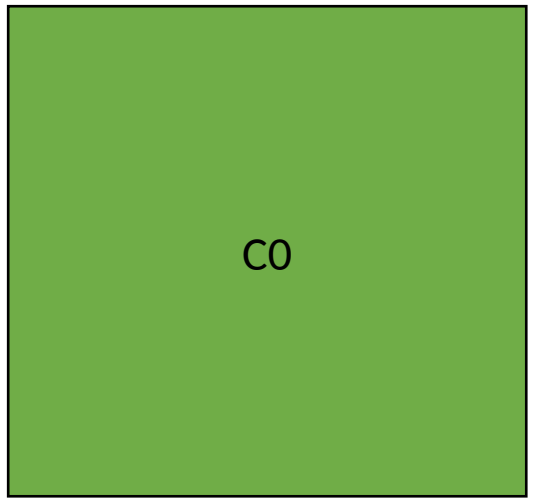
Thread 1

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

Sometimes multiple programs want to share the same core.



Core

And place another thread to execute

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

Thread 1

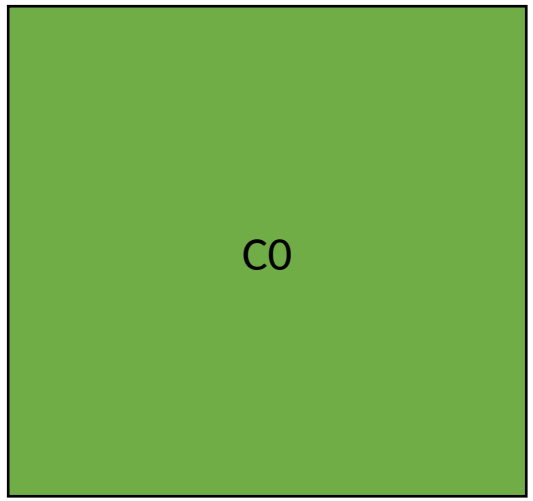
Compiled function #0

```
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

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

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

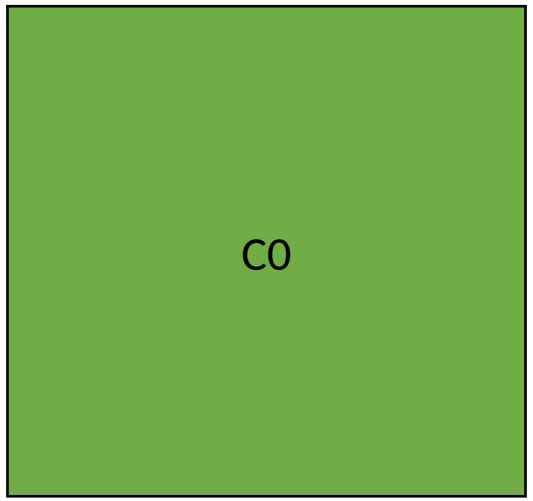
```
13      movd    eax, xmm0
14      xor     eax, 2147483648
15      movd    xmm0, eax
16      movss   dword ptr [rbp - 16], xmm0
17      movss   xmm0, dword ptr [rbp - 8]
18      mulss   xmm0, dword ptr [rbp - 8]
19      movss   xmm1, dword ptr [rip + .LCPI0_1]
20      mulss   xmm1, dword ptr [rbp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

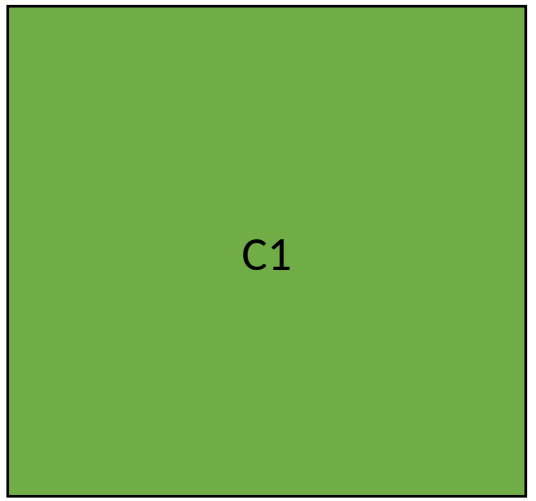
Compiled function #1

```
        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



Core



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

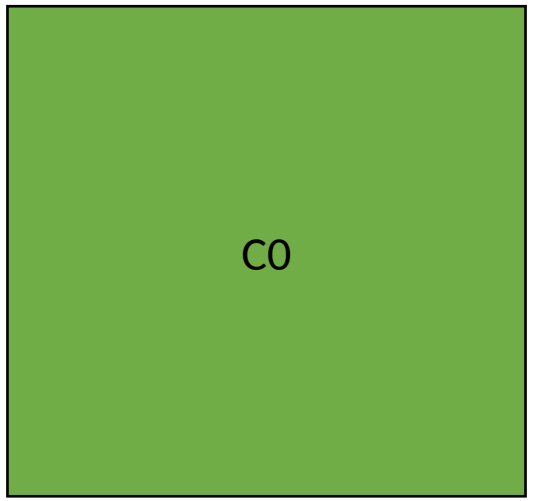
Compiled function #1

```
        movss   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
```

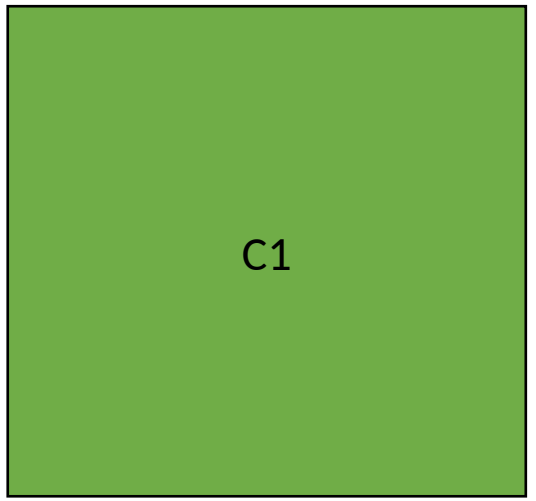
Thread 1

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



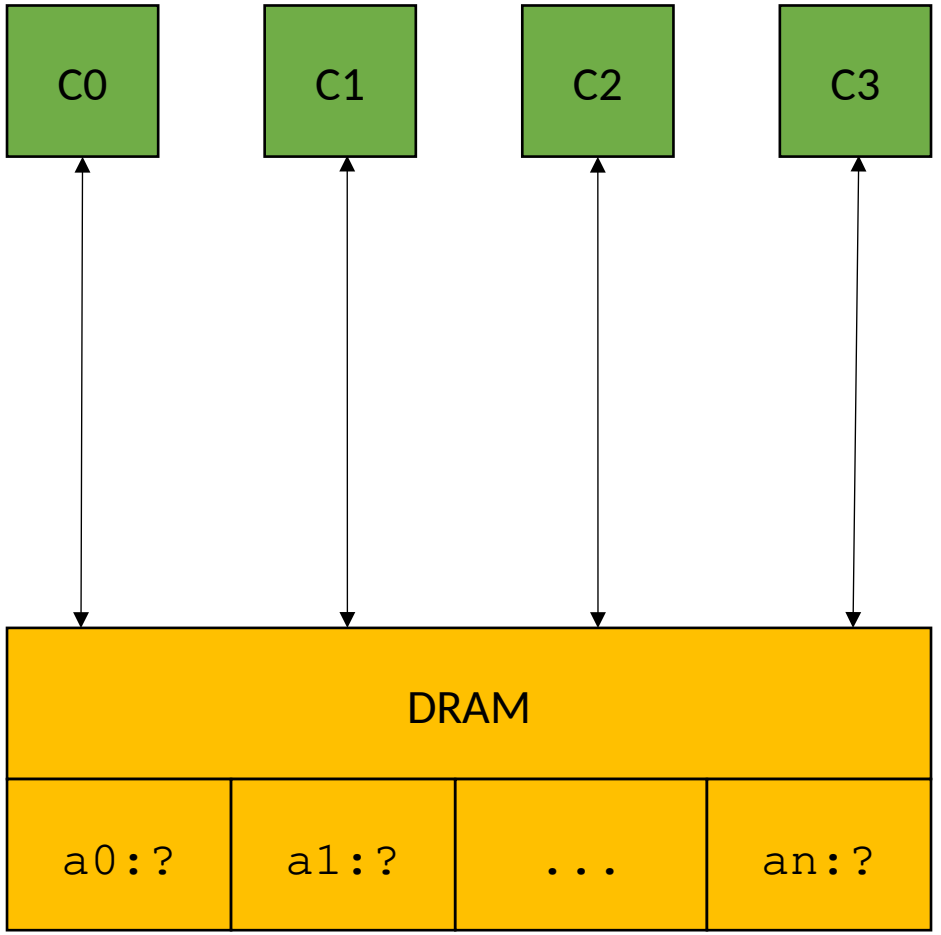
Core



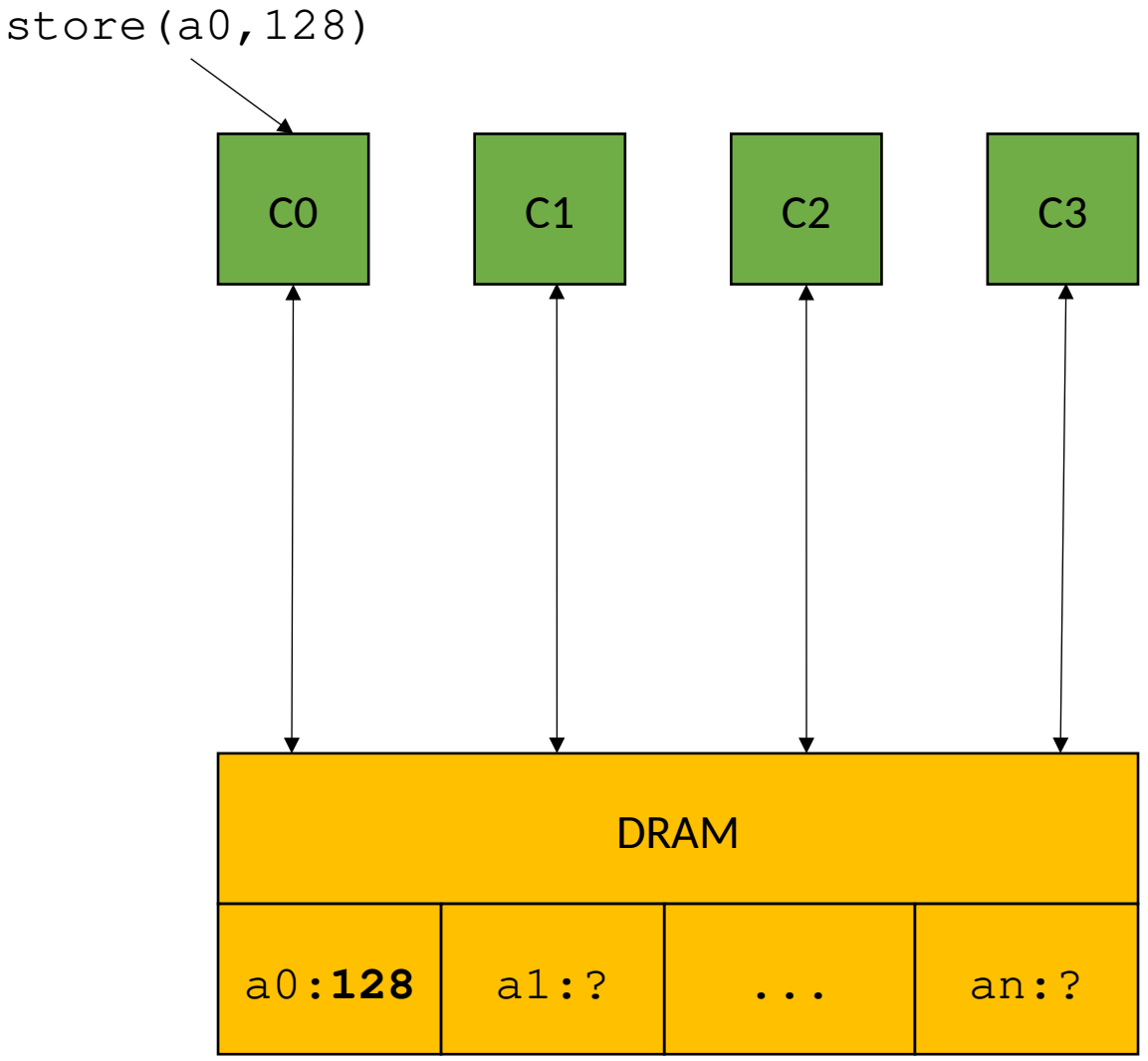
Core

Main memory

`store(a0,128)`

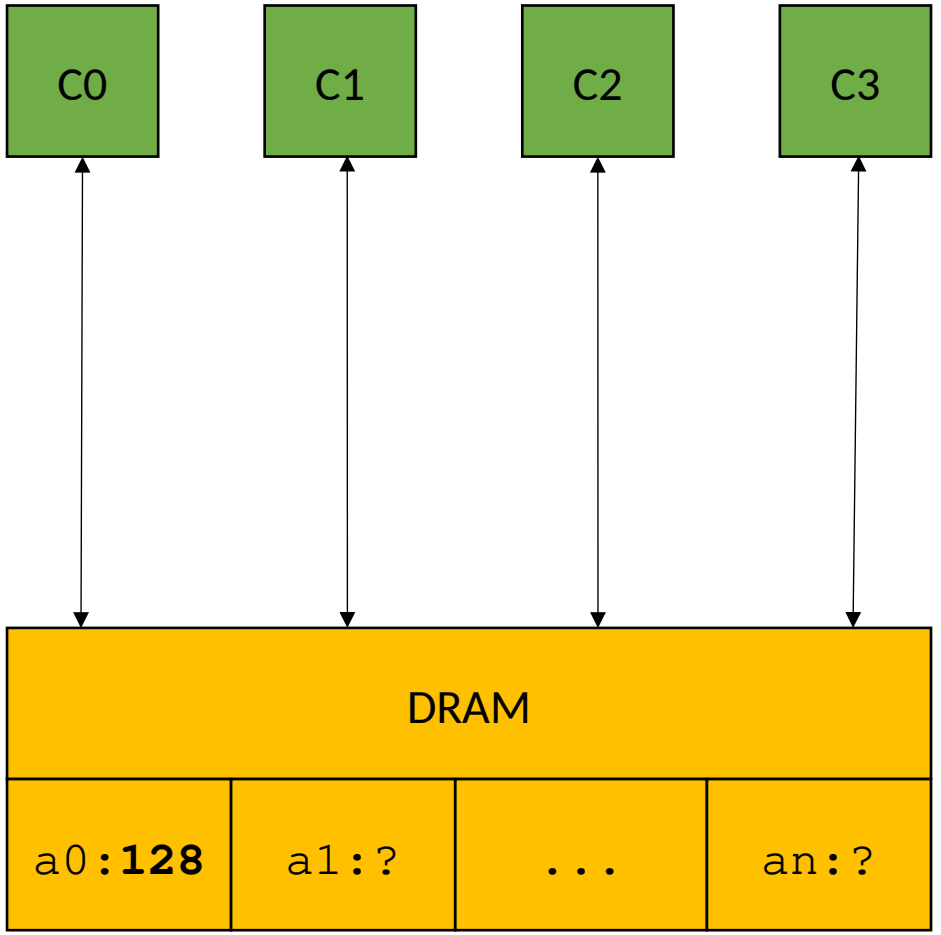


Main memory

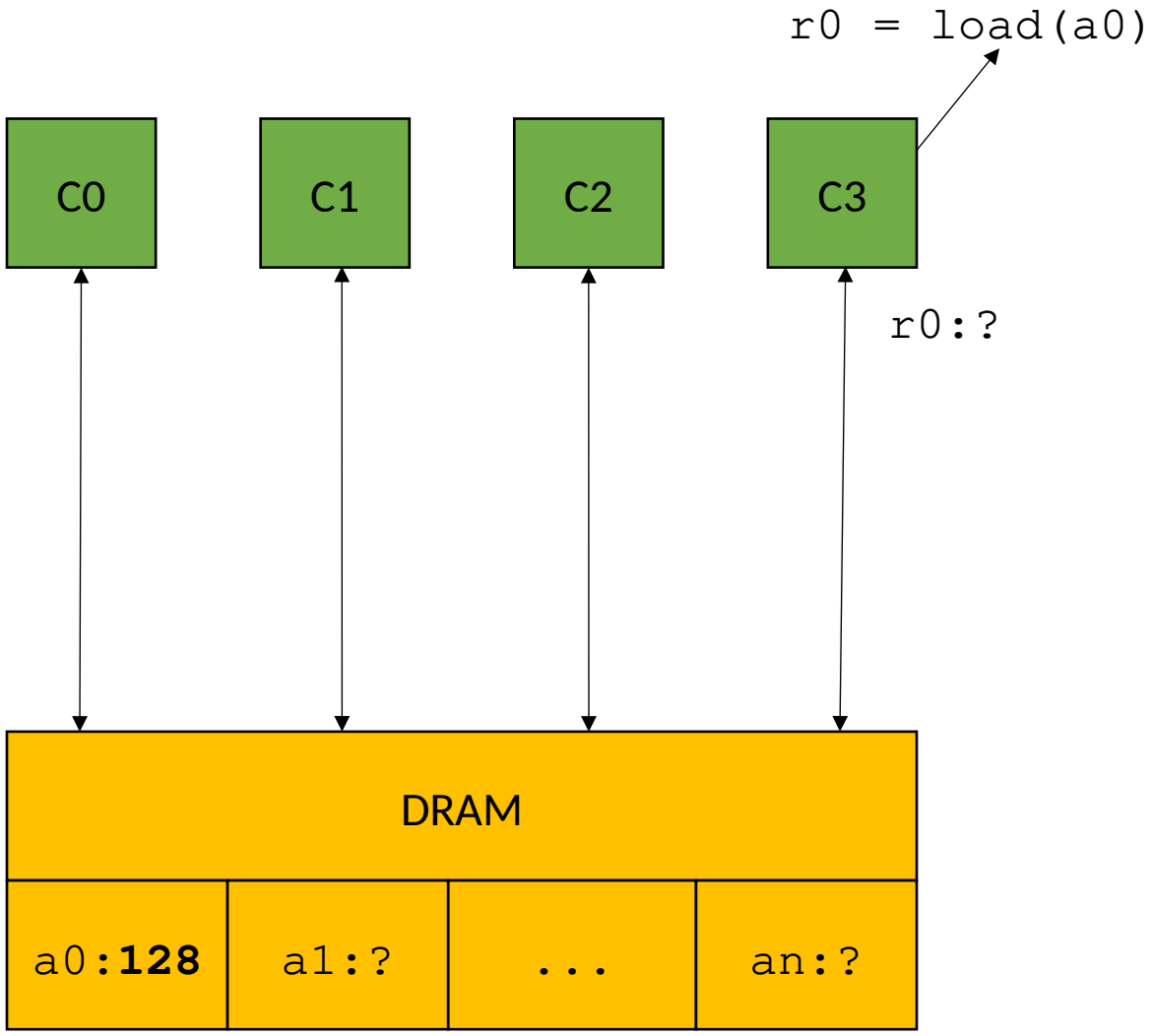


Main memory

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

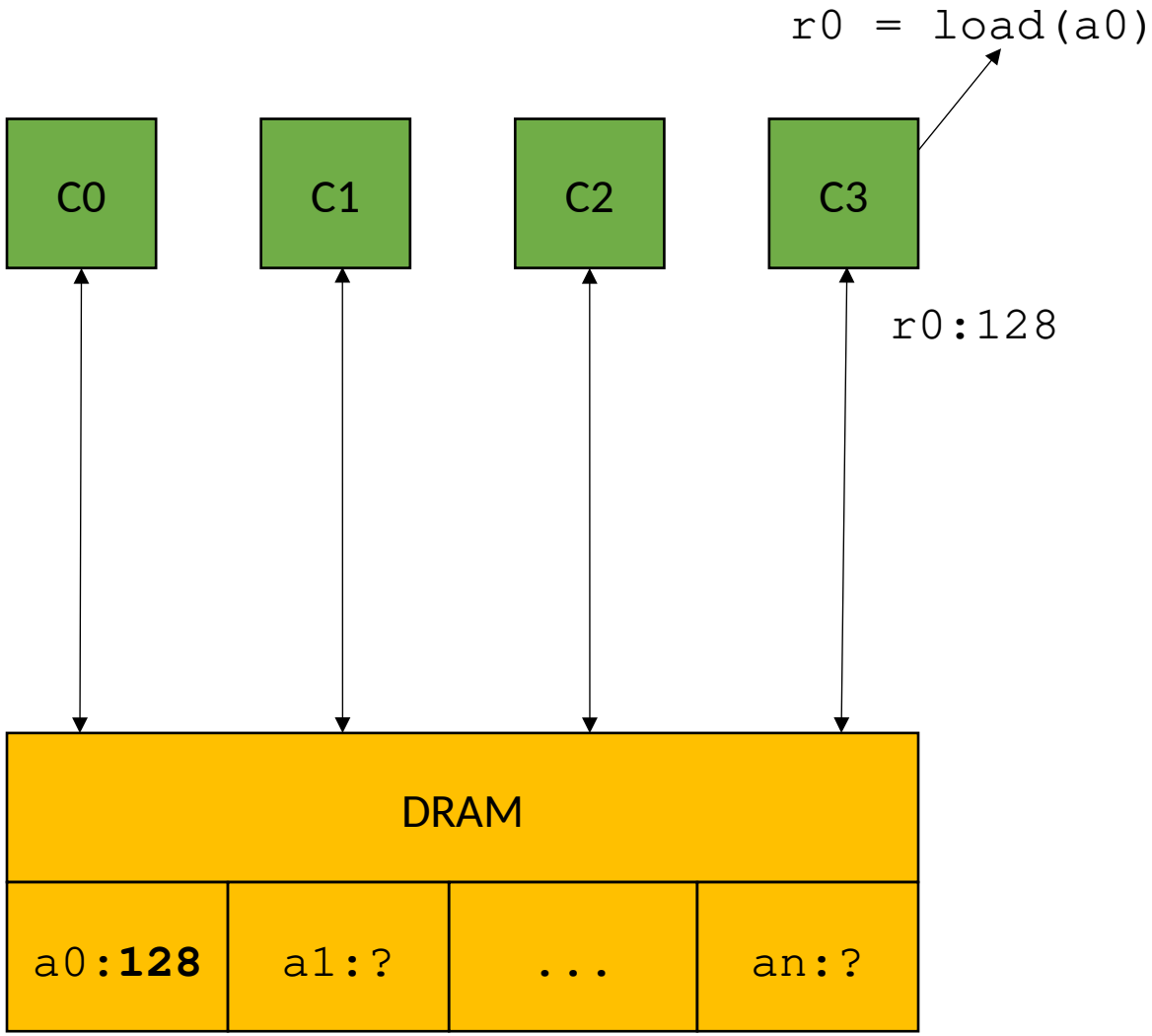


Main memory



Main memory

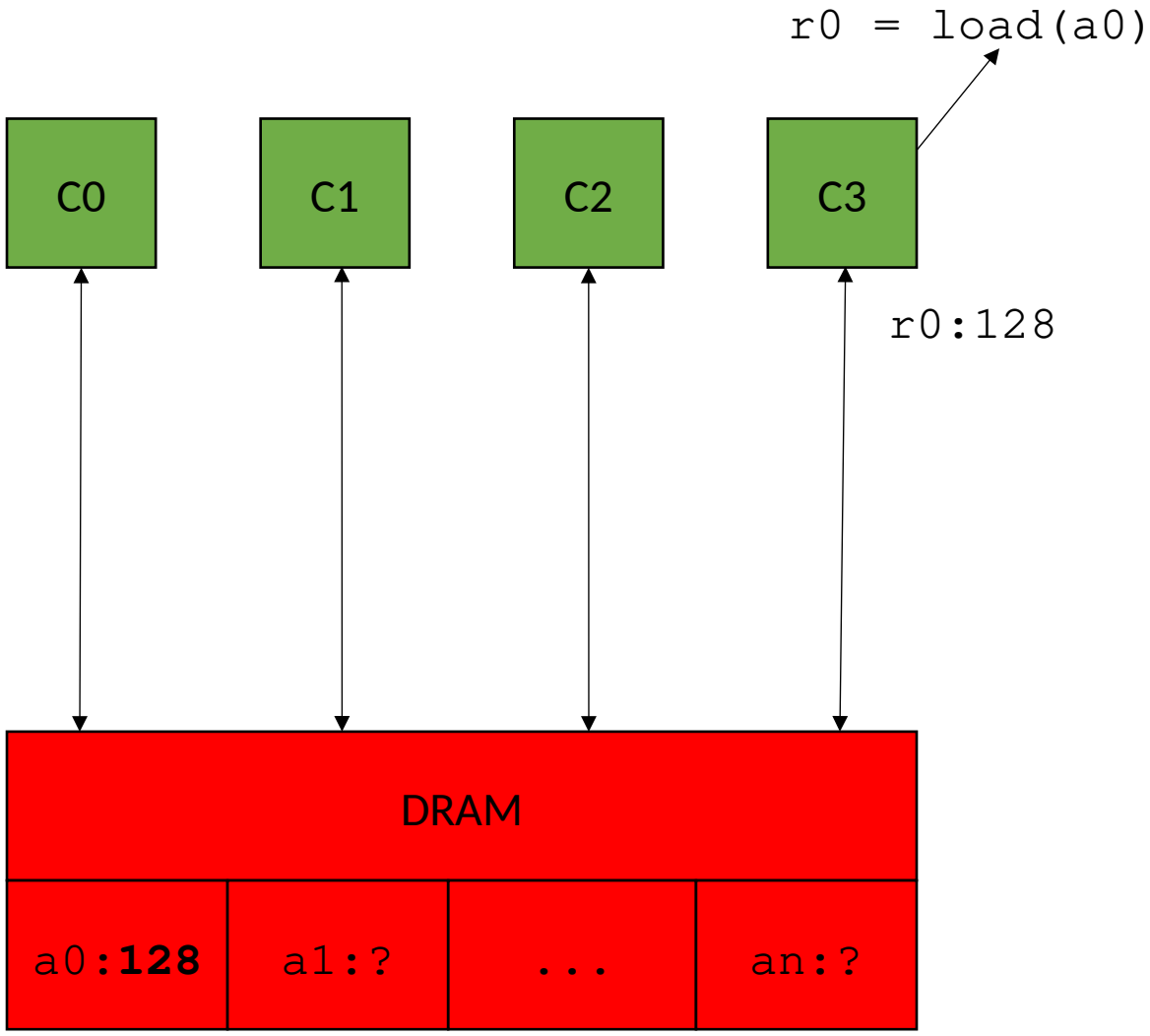
Problem solved!
Threads can communicate!



Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

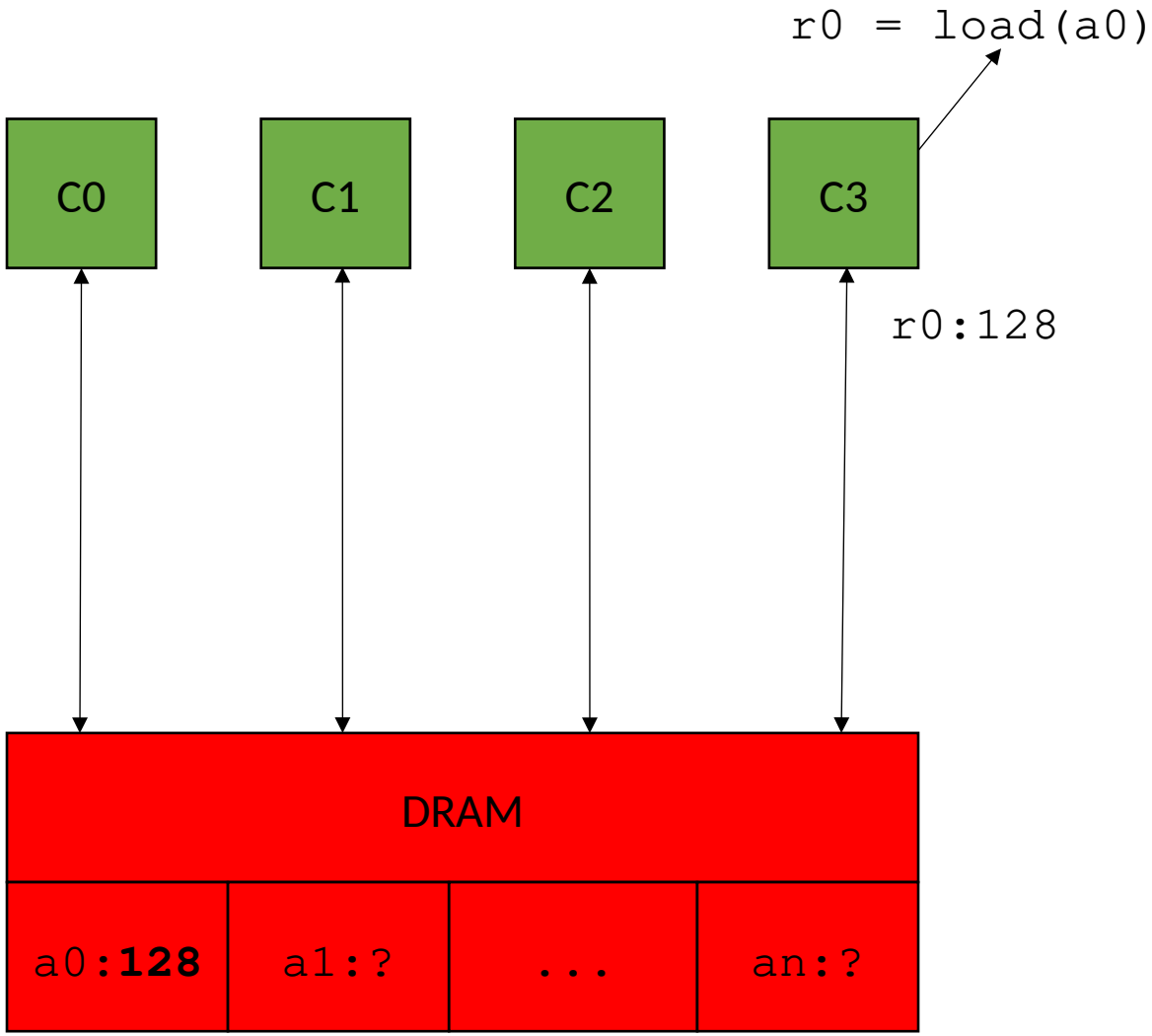


Main memory

Problem solved!
Threads can communicate!

reading a value takes ~200 cycles

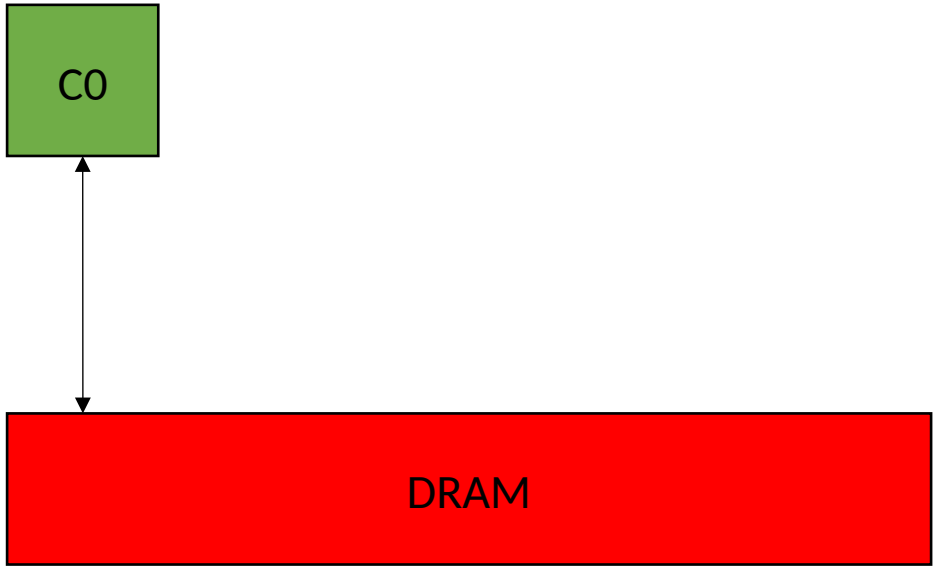
Bad for parallelism, but
also really bad for sequential
code (which we optimized for
decades!)



Main memory

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

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

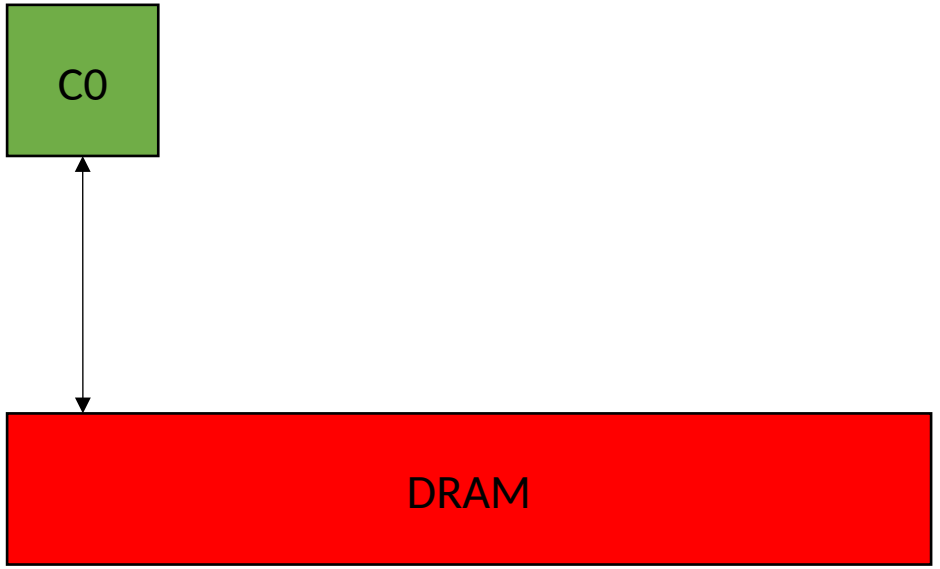


Main memory

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

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

200 cycles



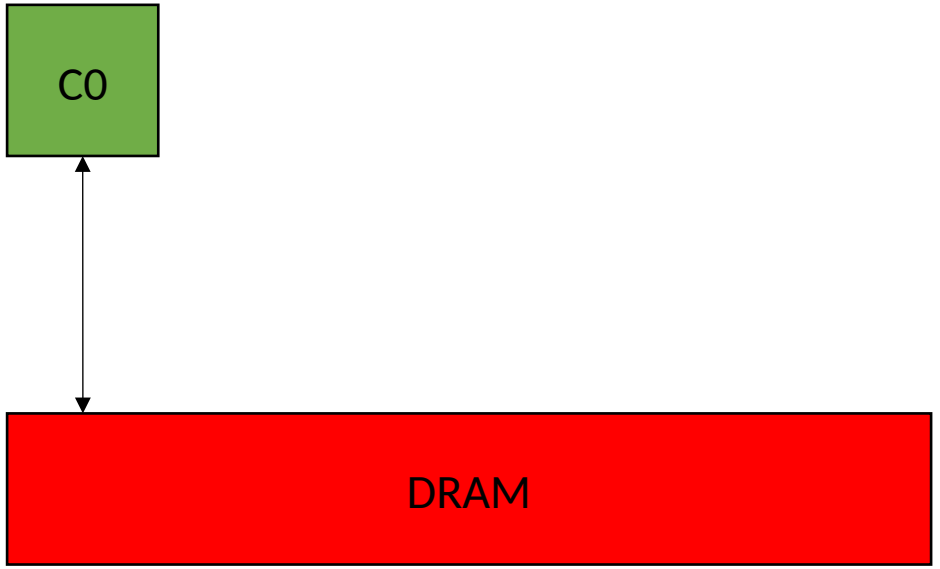
Main memory

```
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

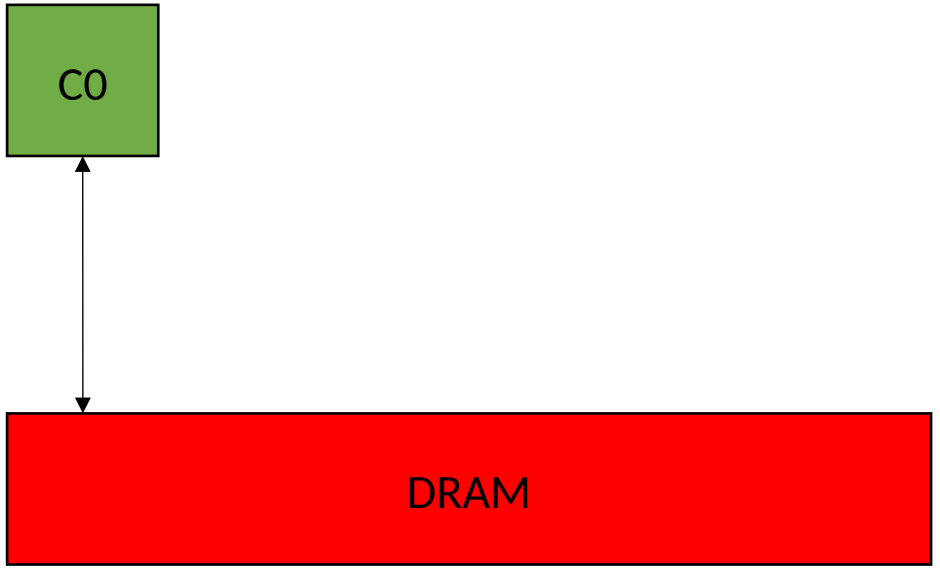
```
int increment(int *a) {
    a[0]++;
}
```

```
%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]++;
}
```

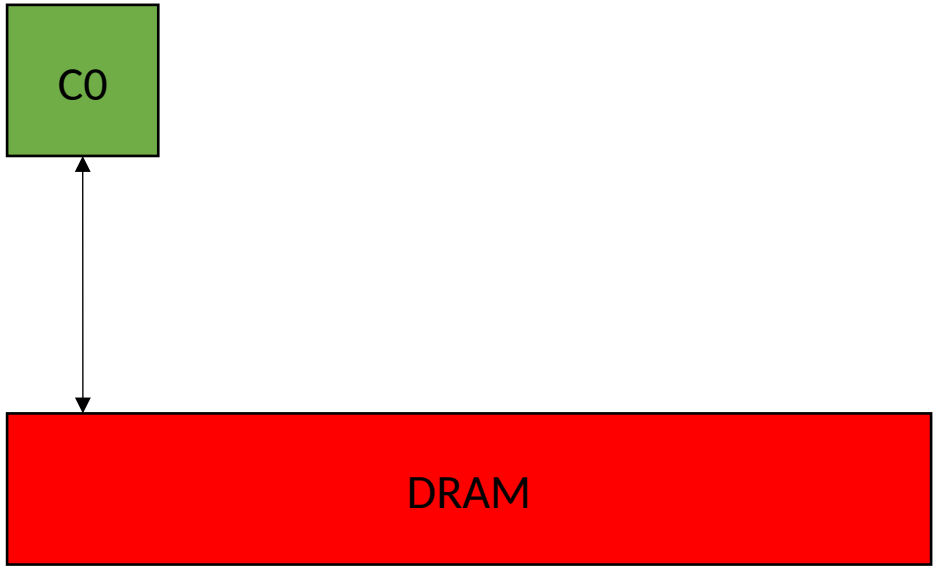
```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

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);
}
```

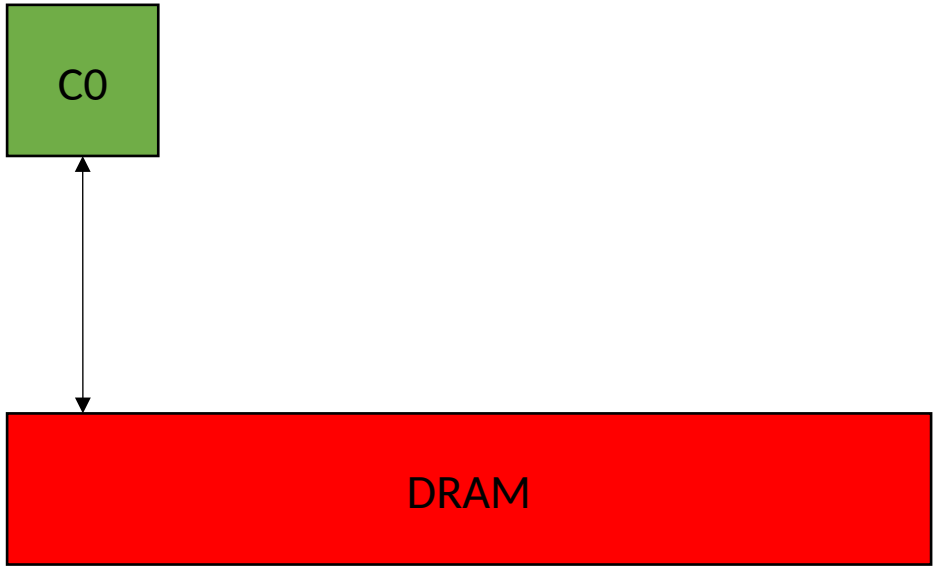
```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

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);
}
```

40100 cycles!

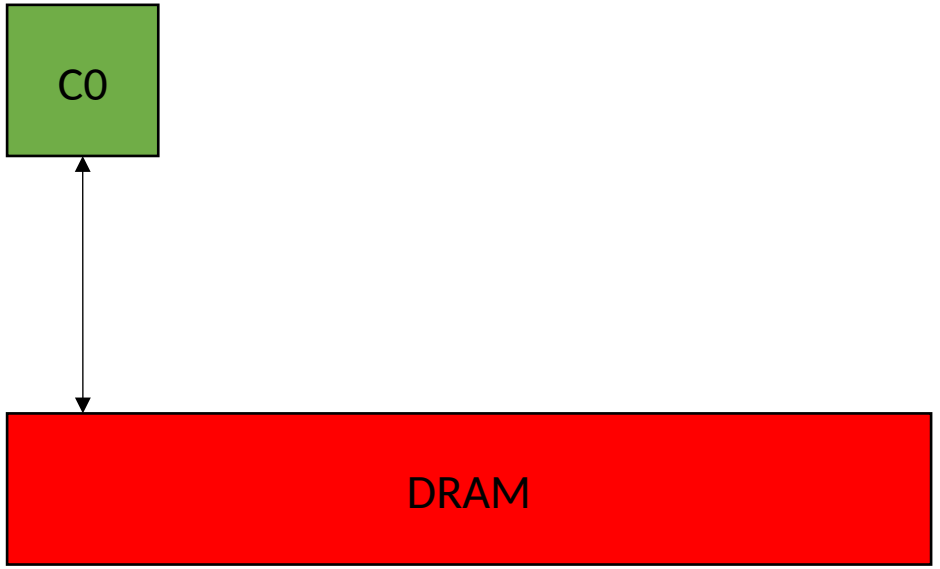
```
%5 = load i32, i32*
%4
%6 = add nsw i32 %5,
1
store i32 %6, i32* %4
```

200 cycles

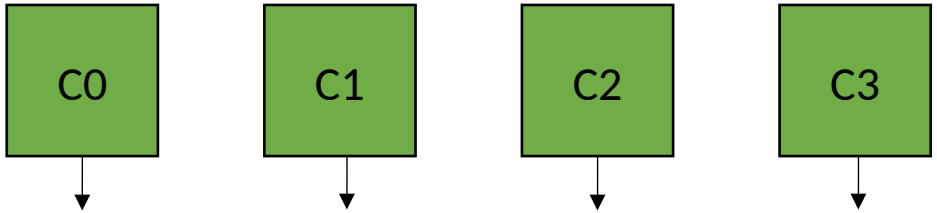
1 cycles

200 cycles

401 cycles



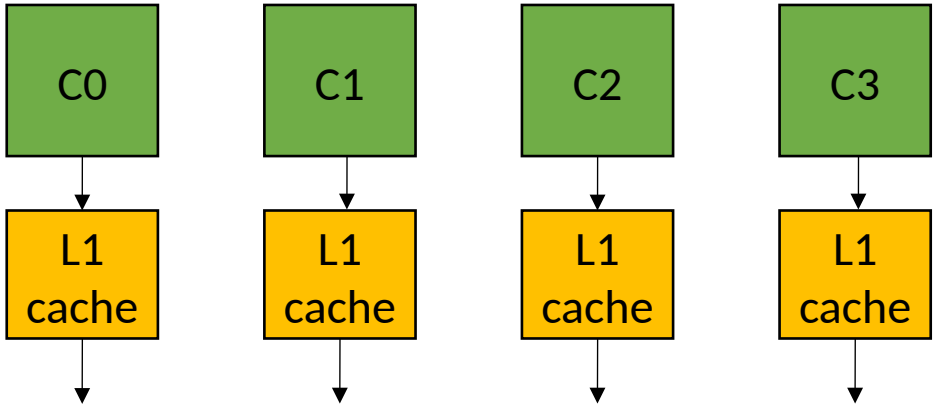
Caches



latency
~200 cycles

Many GBs
(or even TBs)

Caches



latency
~4 cycles

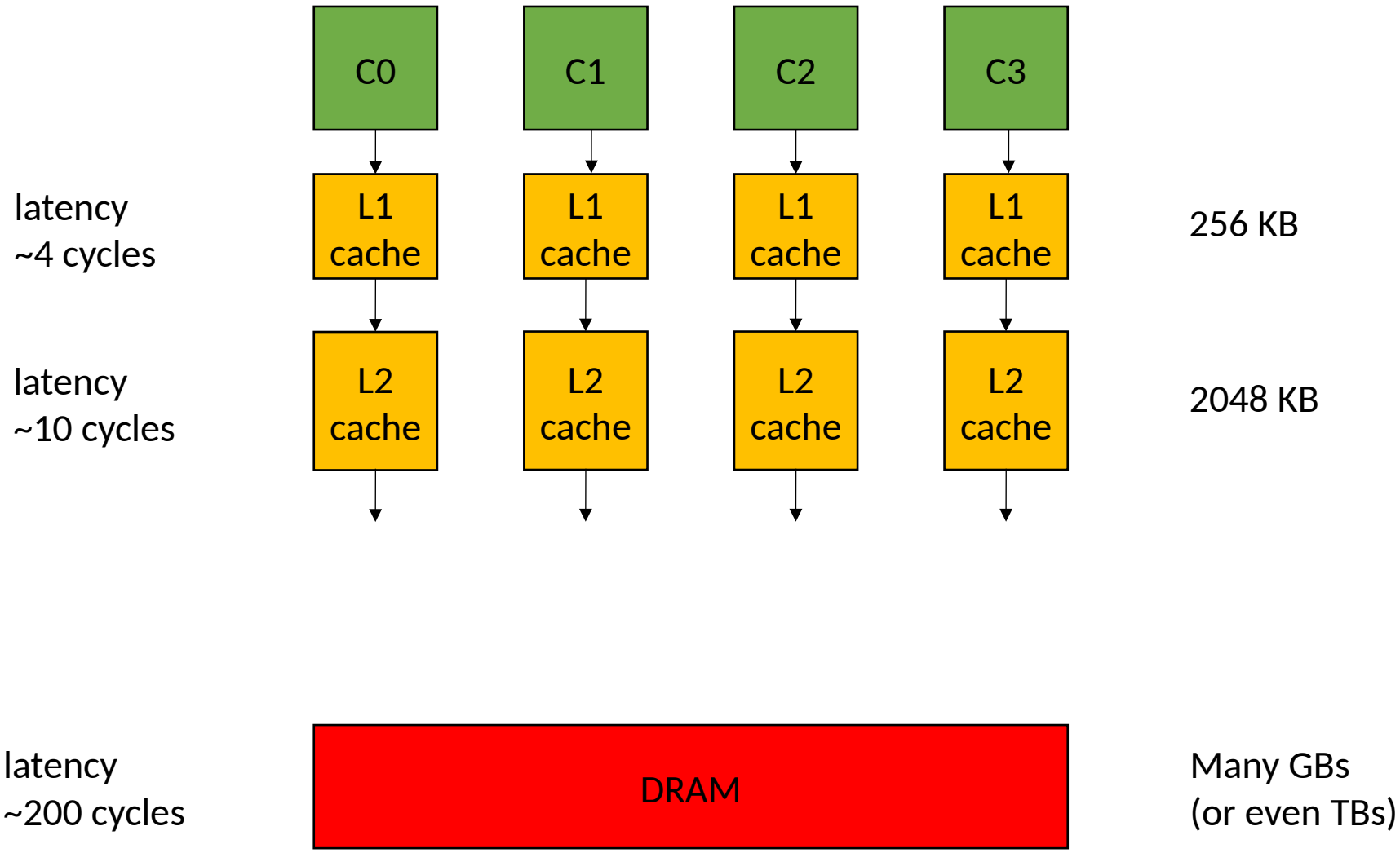
256 KB

latency
~200 cycles

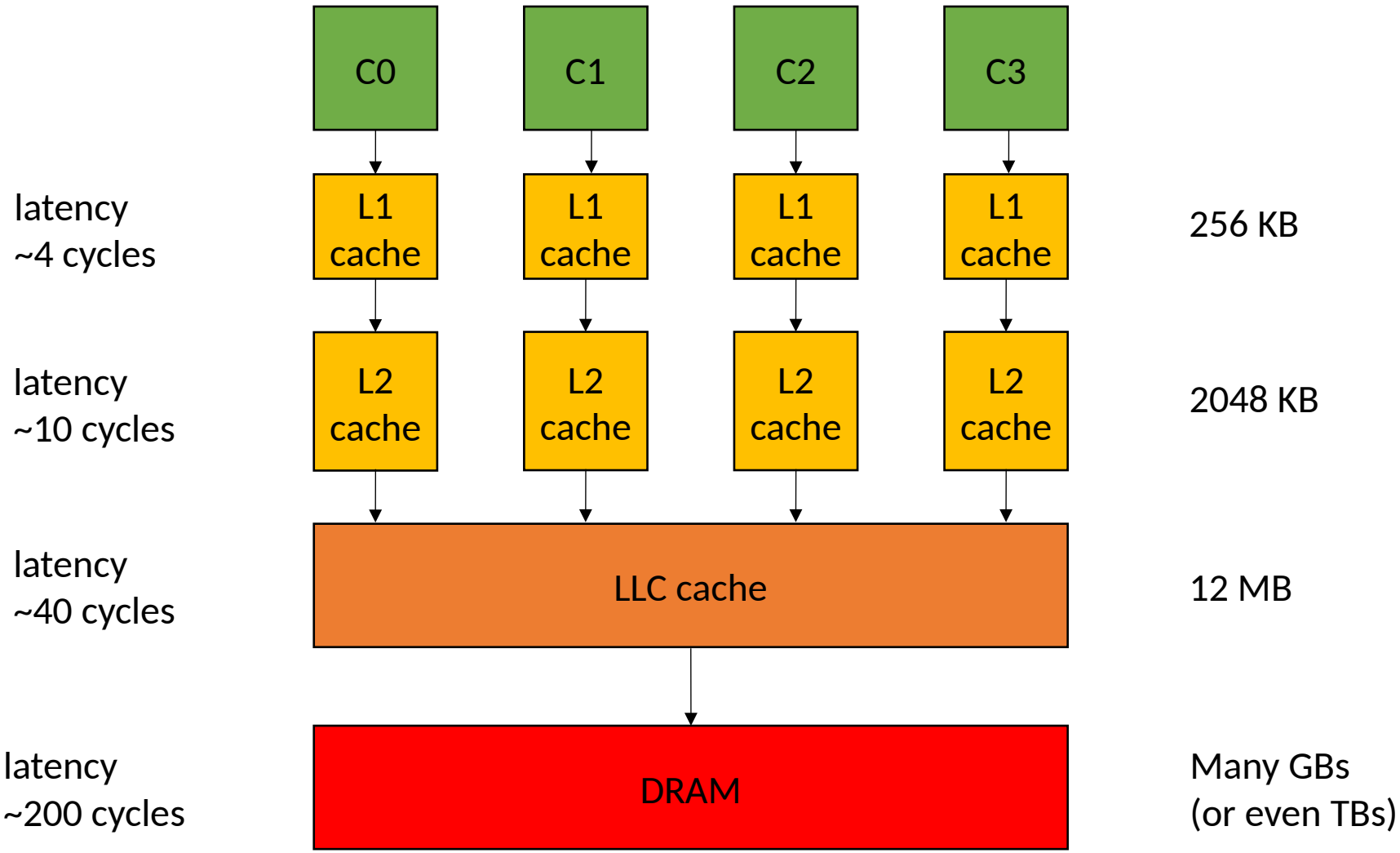
DRAM

Many GBs
(or even TBs)

Caches


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

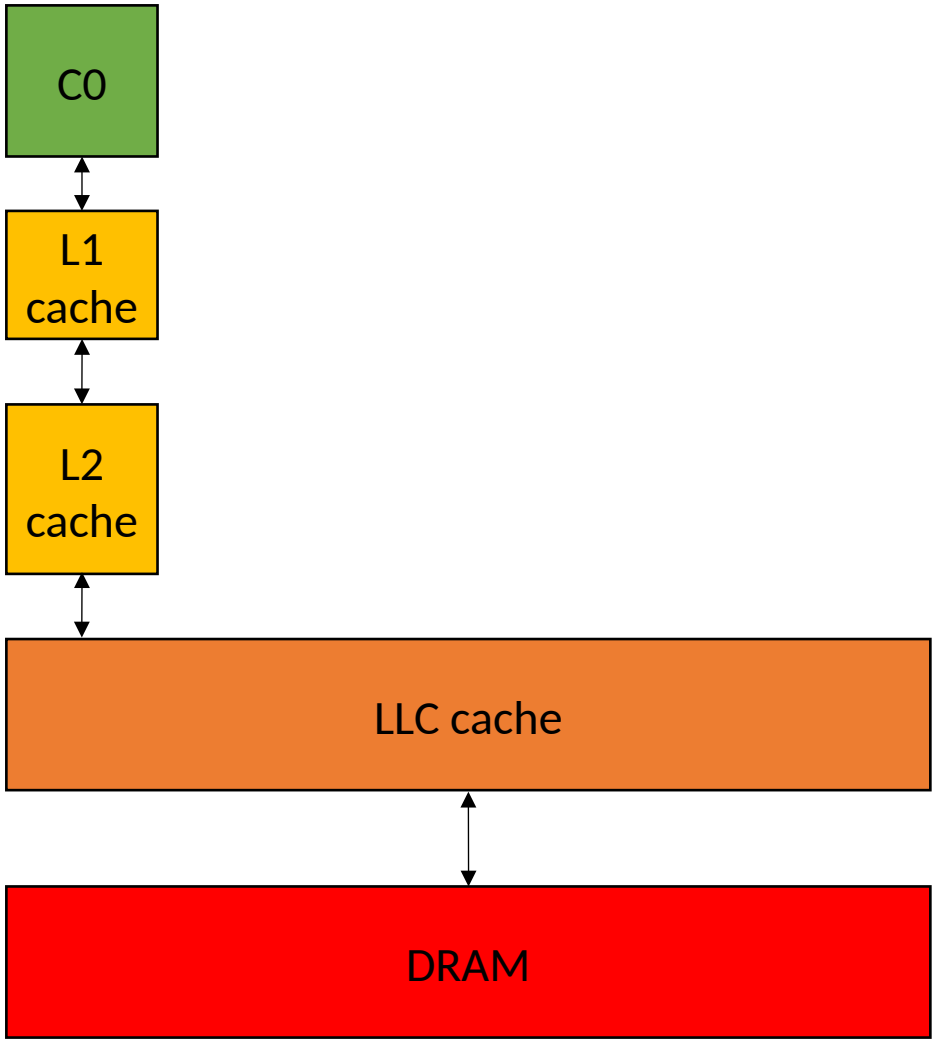
Caches


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

Caches

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

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



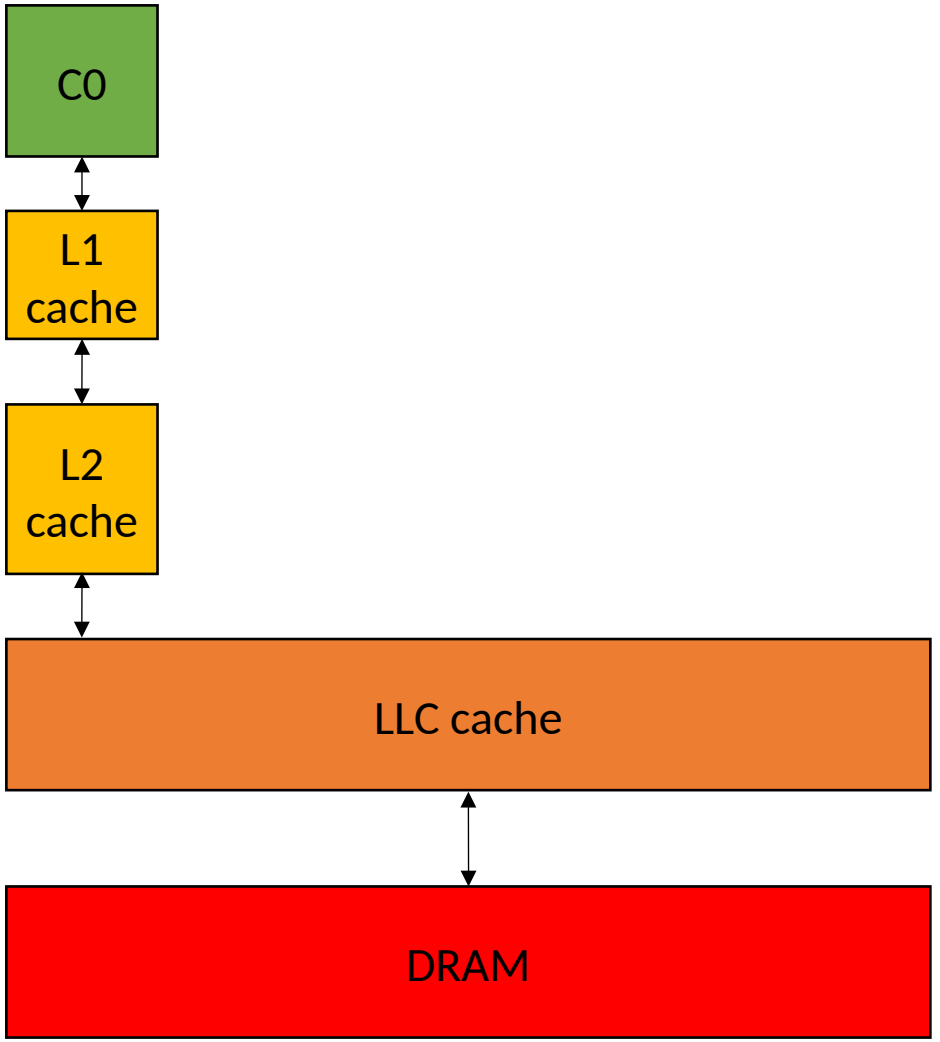
Caches

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

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

4 cycles

Assuming the value is in the cache!



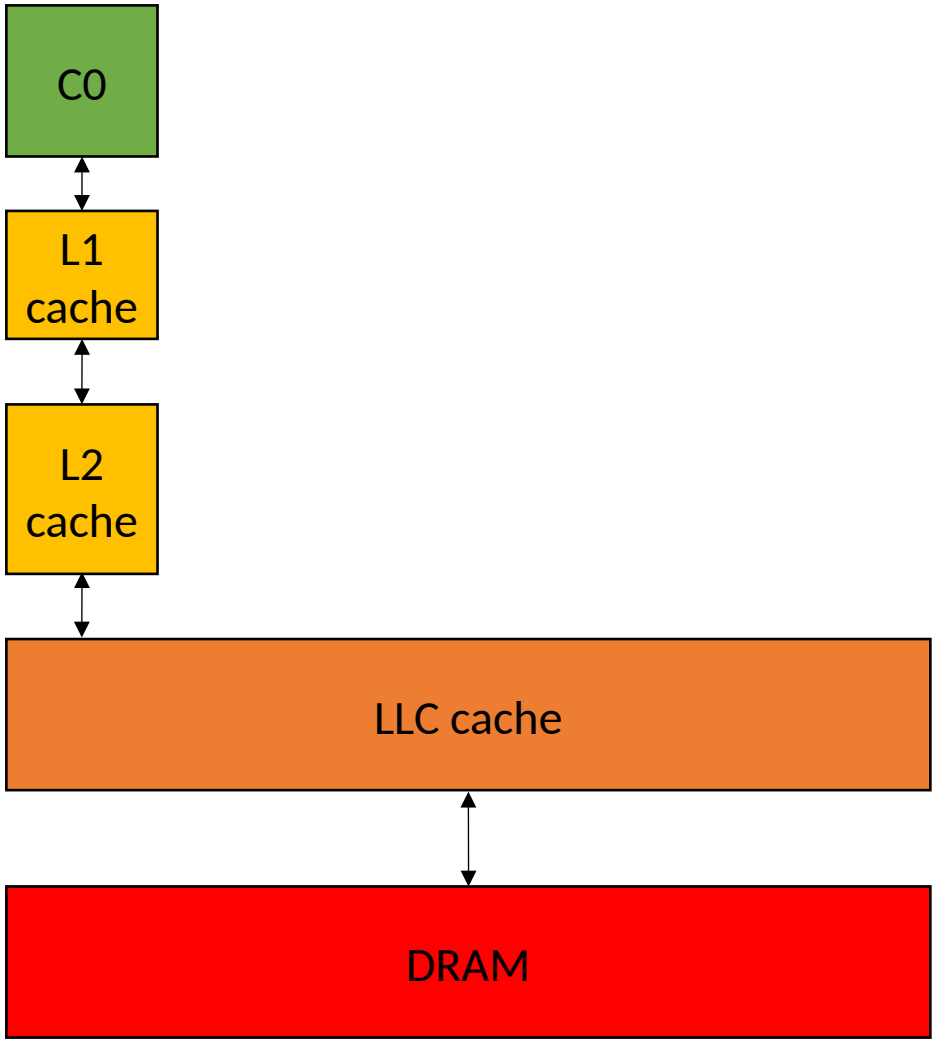
Caches

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

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

4 cycles

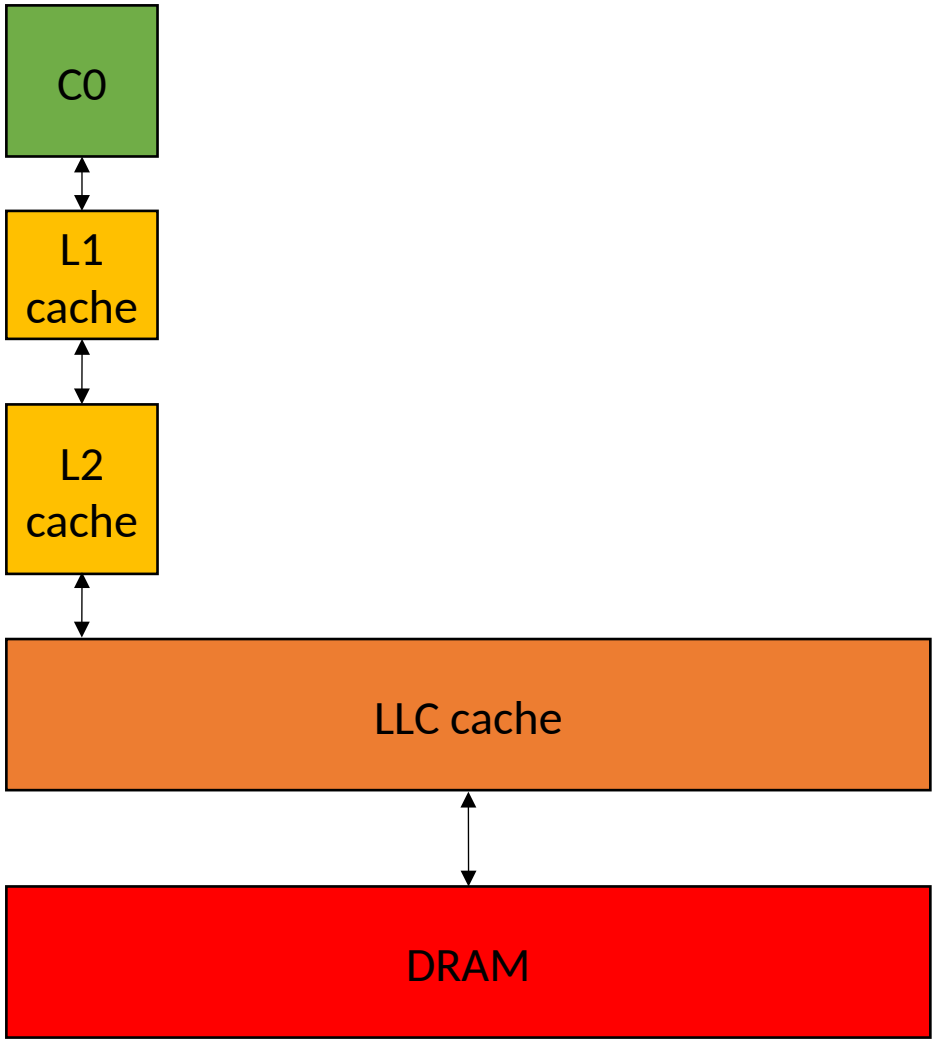
1 cycles



Caches

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

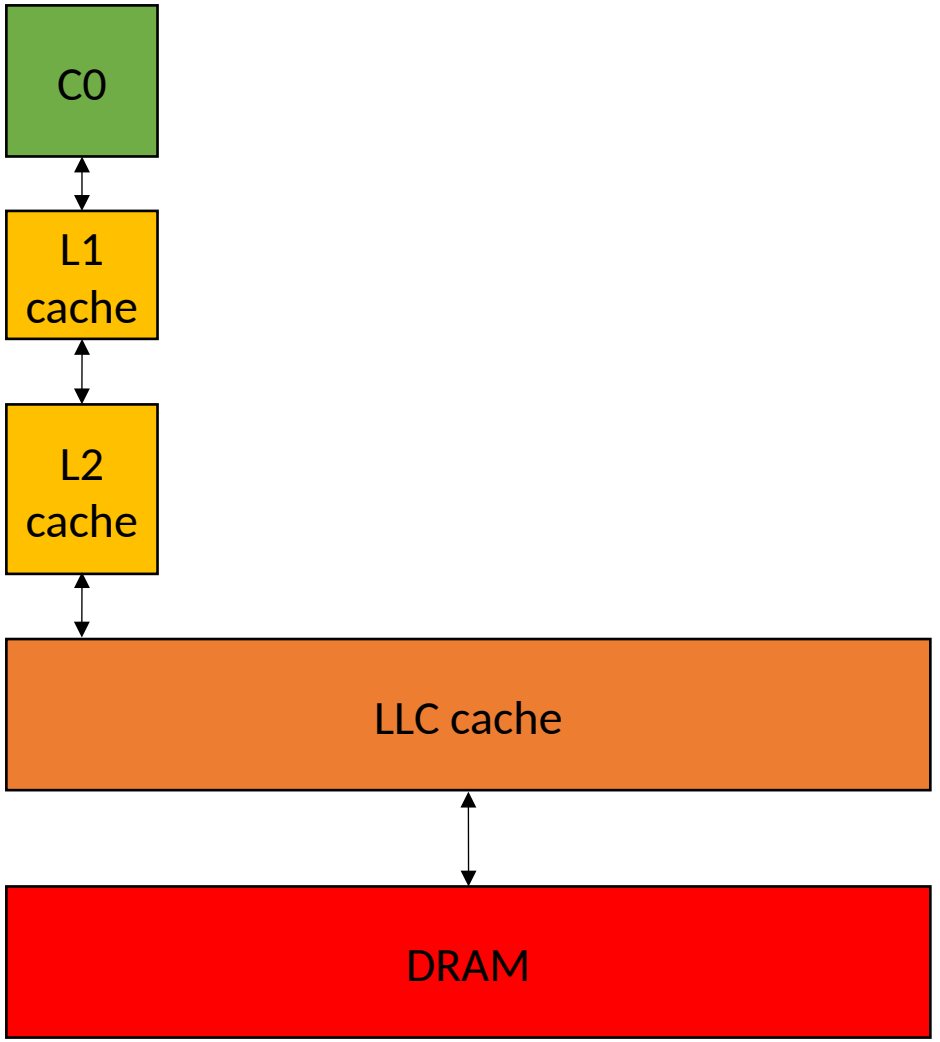
```
%5 = load i32, i32*          4 cycles
%4                           1 cycles
%6 = add nsw i32 %5,         4 cycles
1
store i32 %6, i32* %4
```



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!
```



Quick overview of C/++ pointers/memory

Passing arrays in C++

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

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

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

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

Passing pointers

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

pass pointer directly through

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

pass an offset of 8

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

another way to pass an offset of 8

Memory Allocation

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

stack allocation

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

C++ style

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

C style

Cache lines

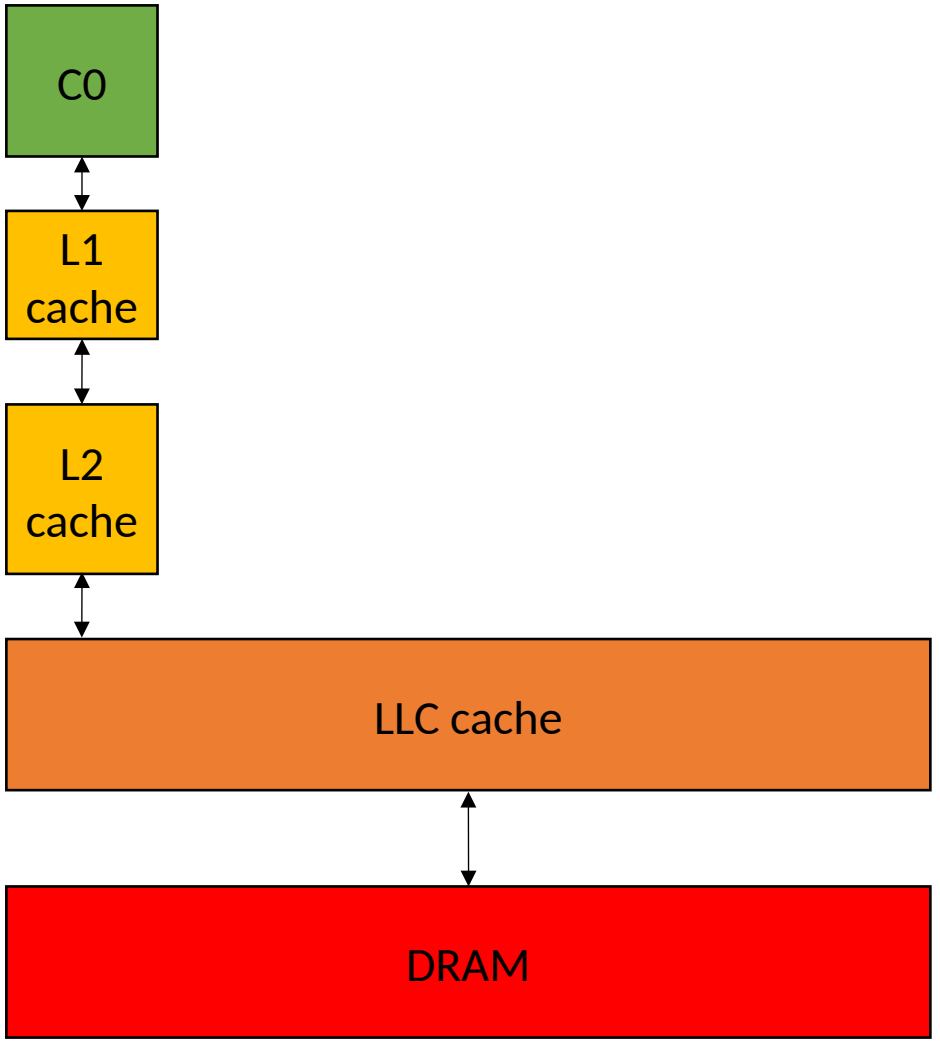
- Cache line size for x86: 64 bytes:
  - 64 chars
  - 32 shorts
  - 16 float or int
  - 8 double or long

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
```

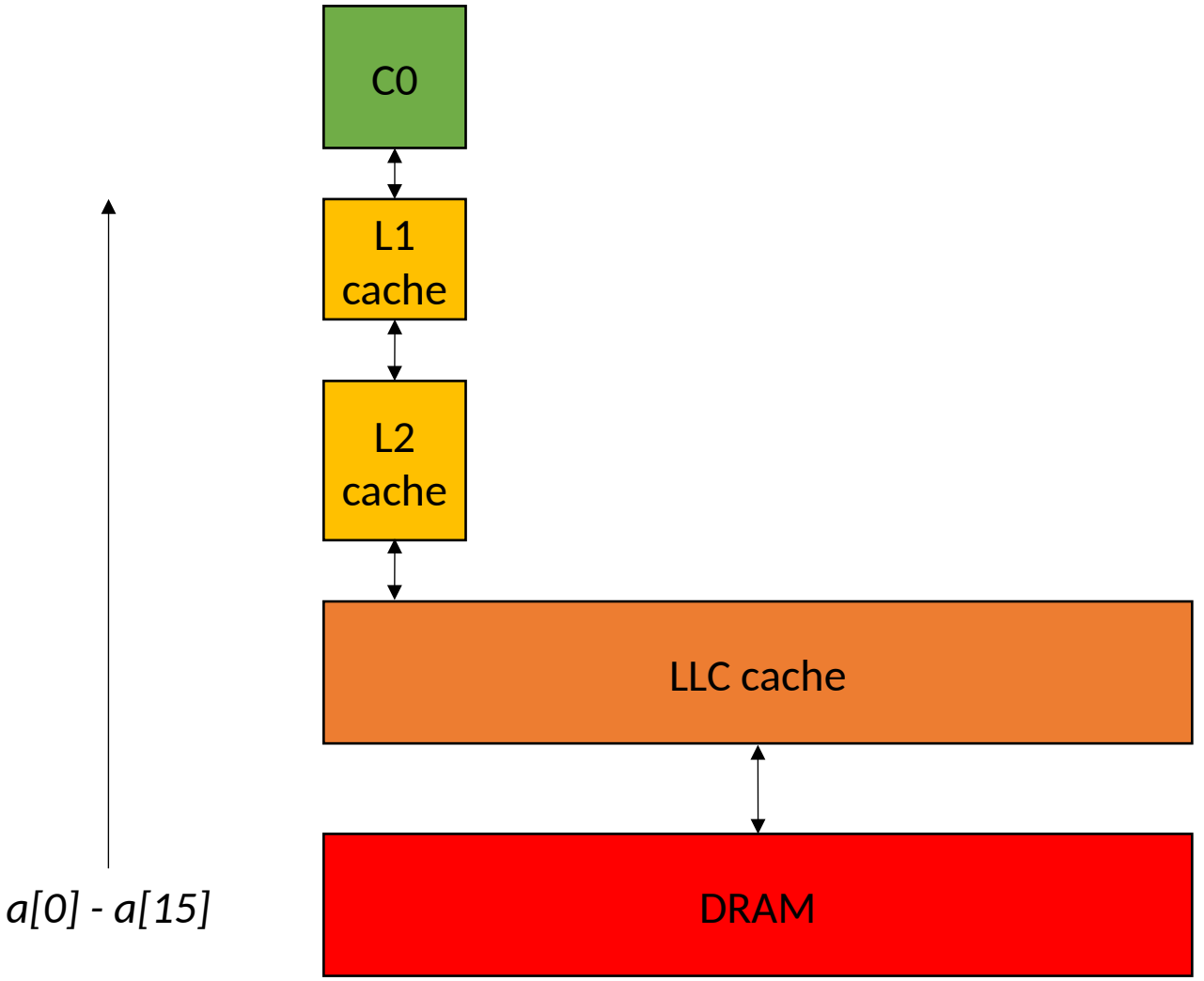


Assume a[0] is not in the cache

Caches

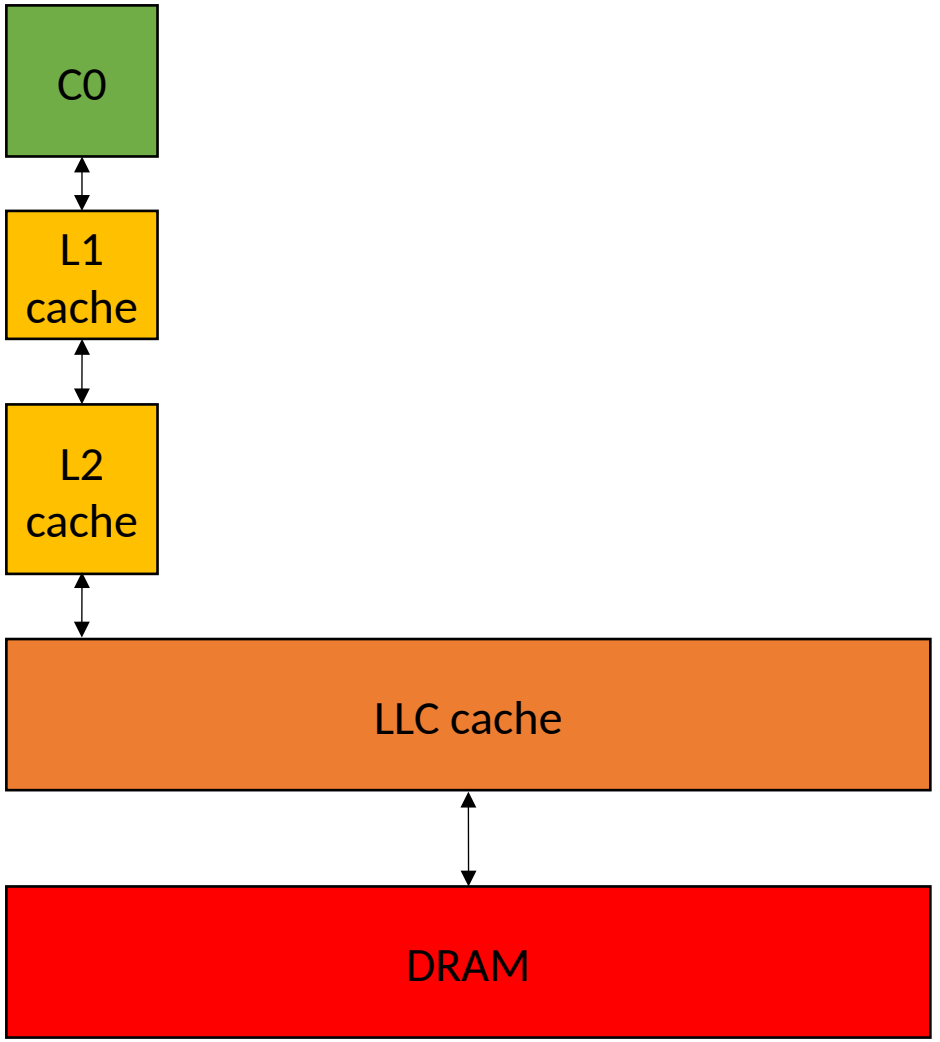
```
int increment(int *a) {
    a[0]++;
}
```

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



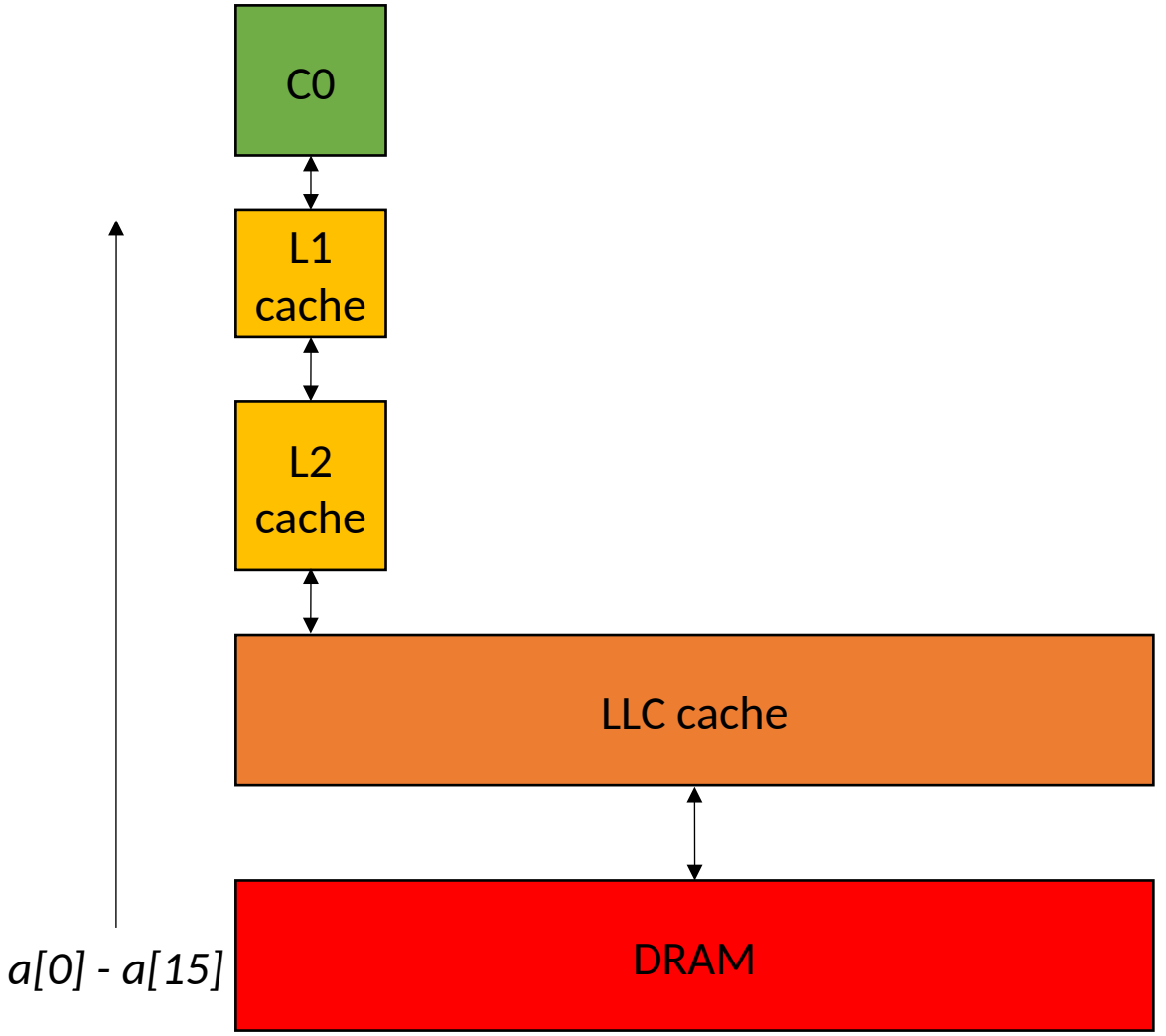
Caches

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



Caches

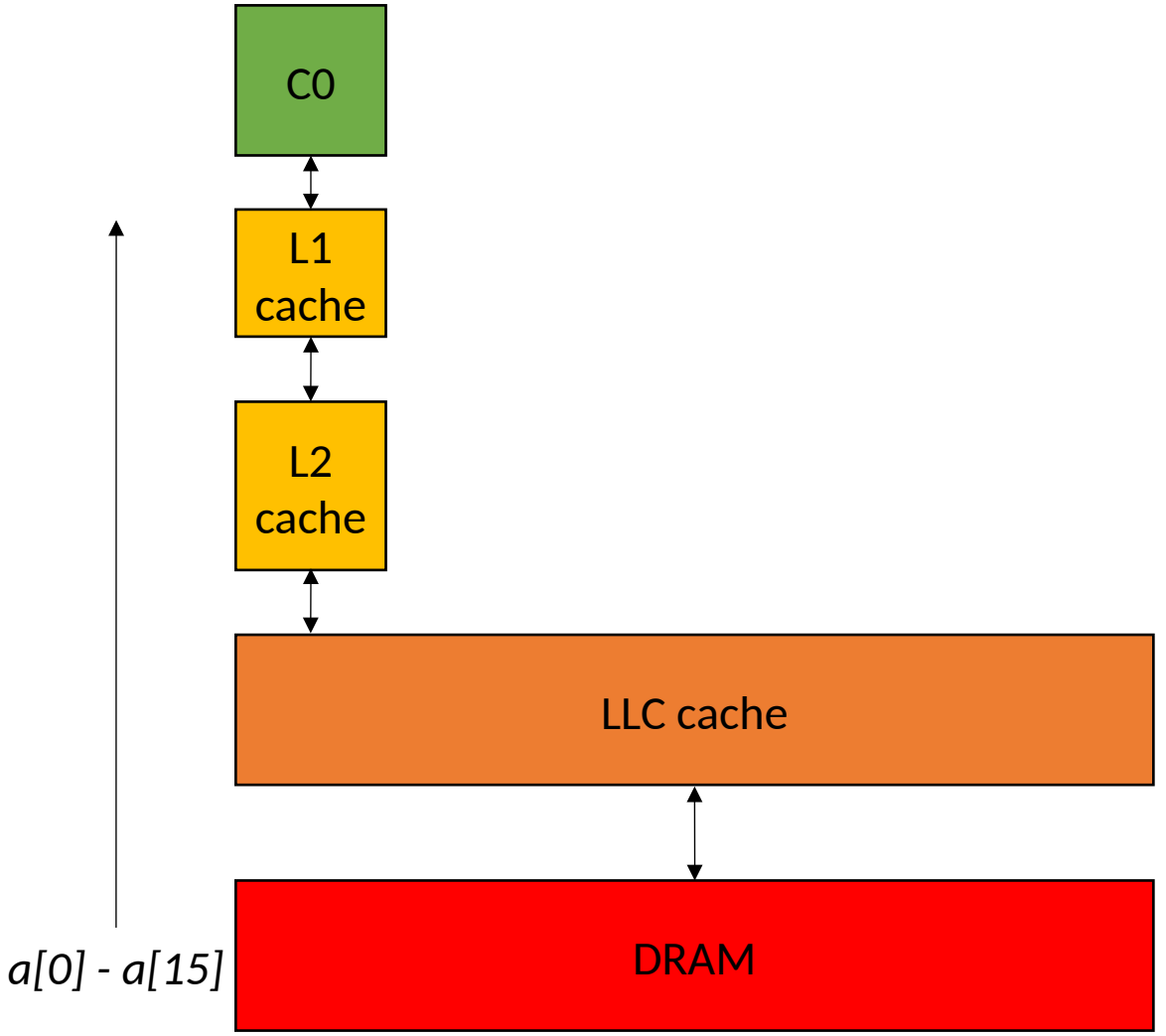
```
int increment_several(int *a) {
    a[0]++;
    a[15]++;
    a[16]++;
}
```



Caches

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

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

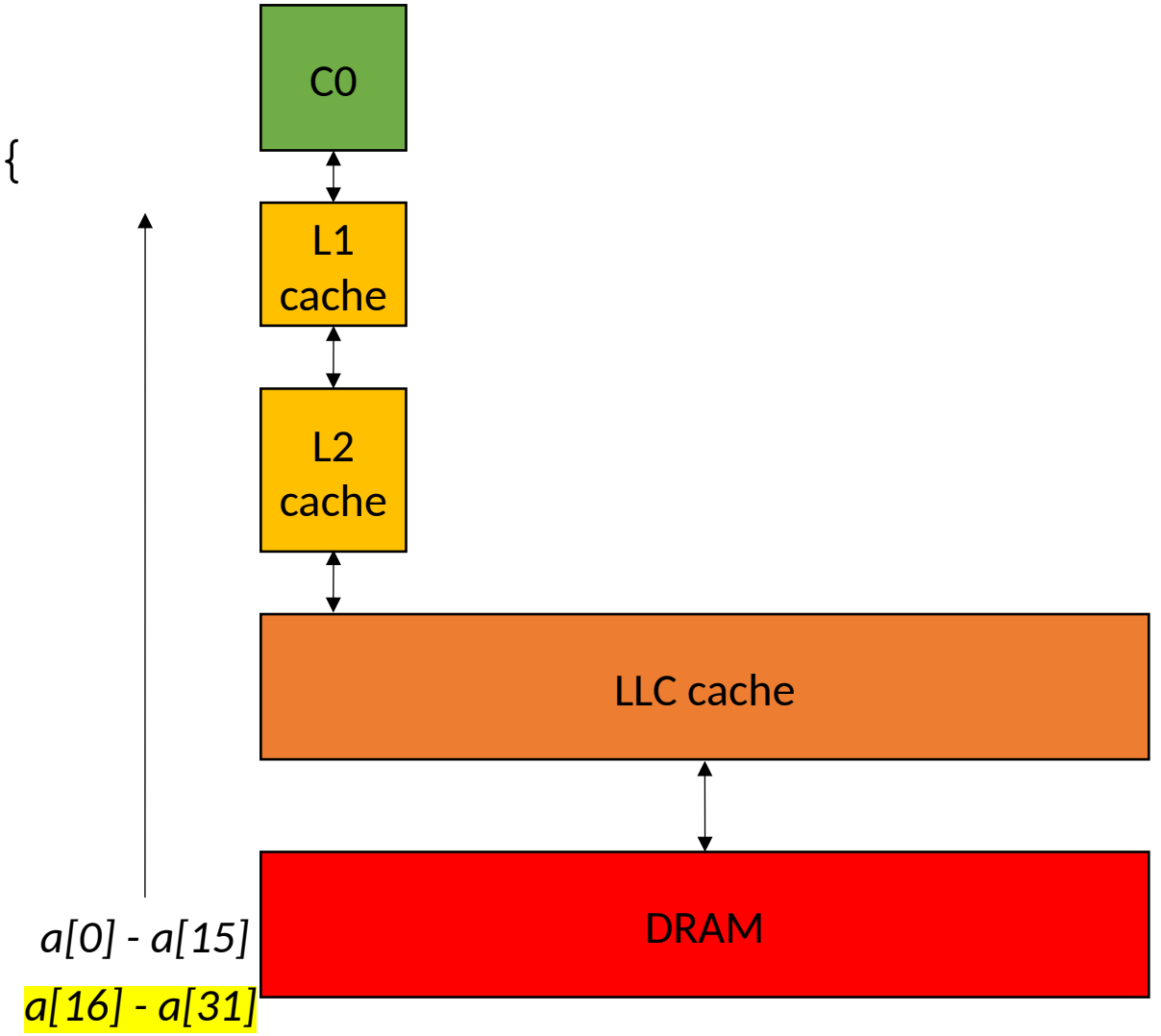


Assume a[0] is not in the cache

Caches

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

Miss

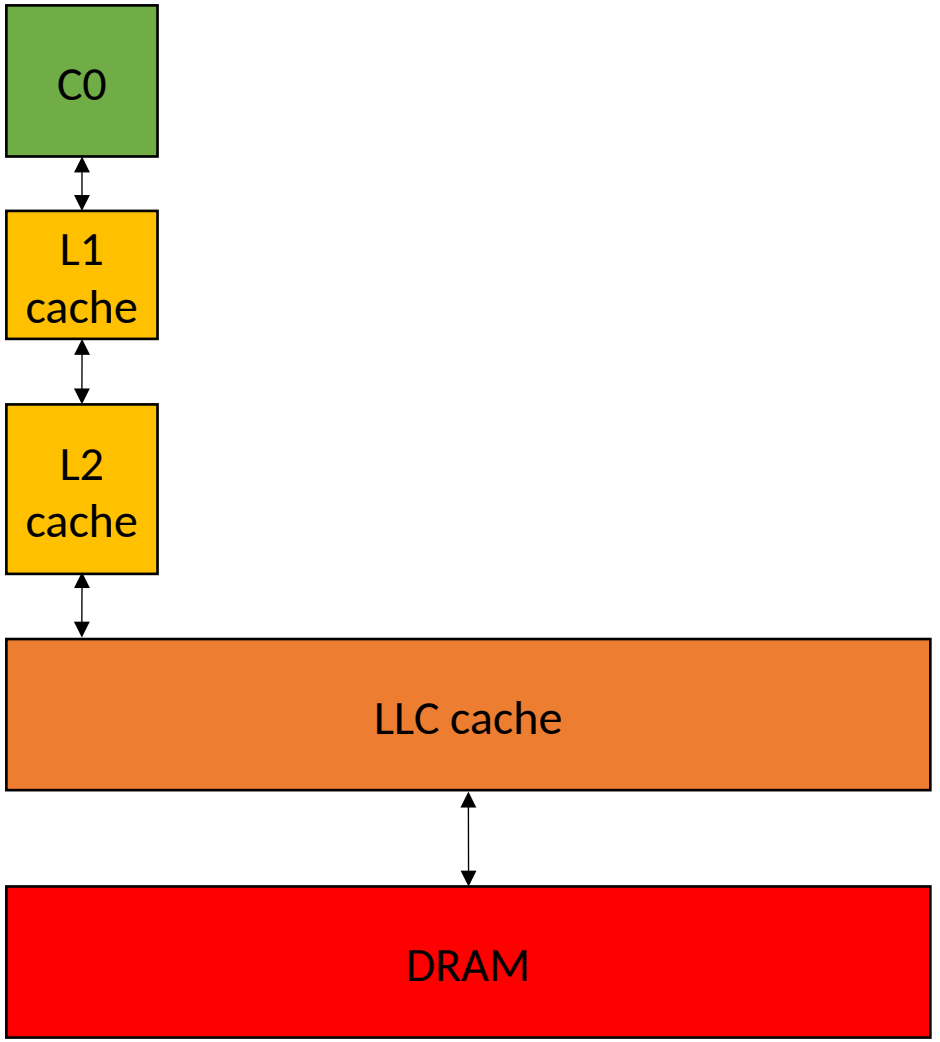


Assume a[0] is not in the cache

Cache alignment

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

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

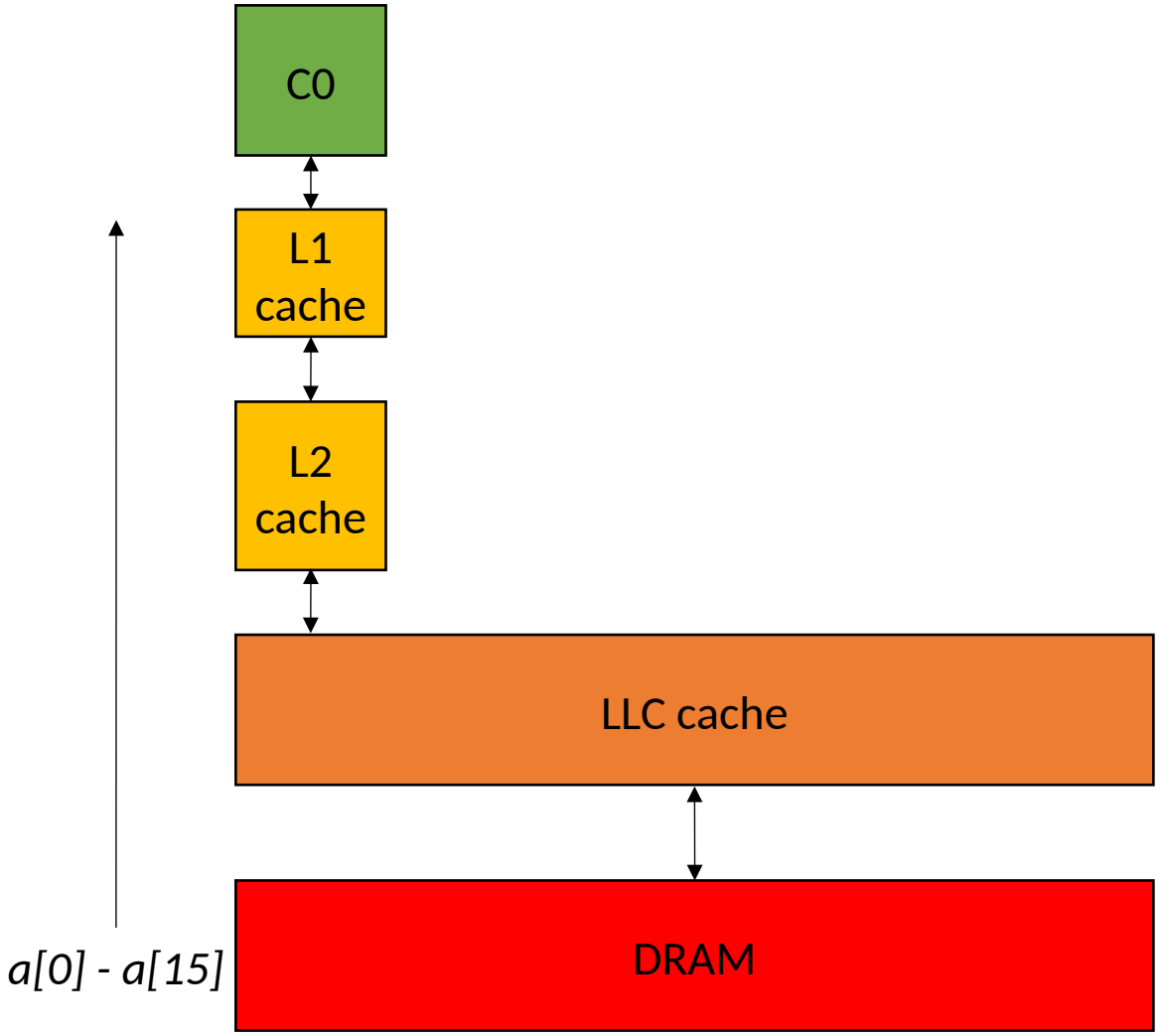


Assume a[0] is not in the cache

Cache alignment

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

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



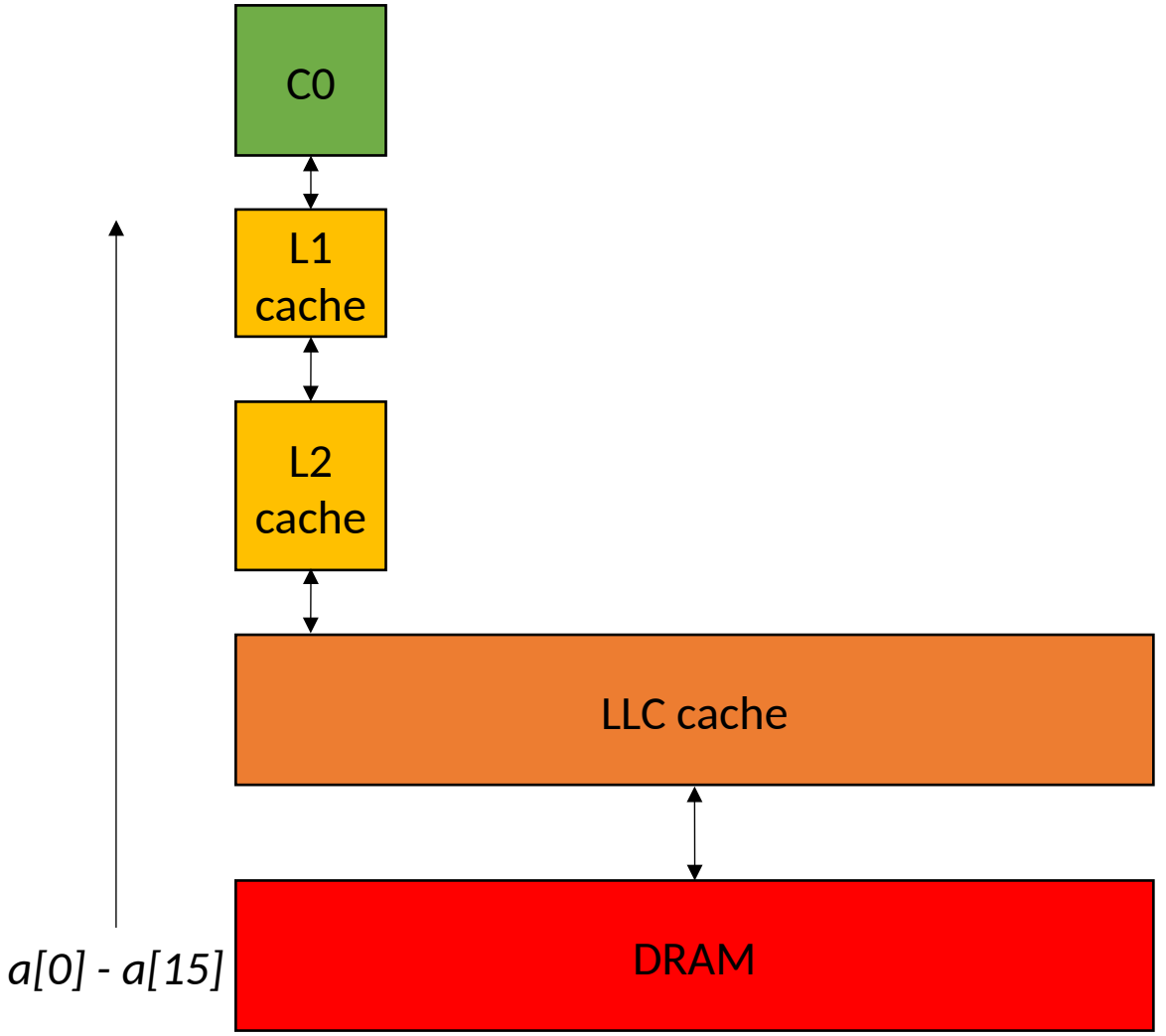
Assume a[0] is not in the cache

Cache alignment

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

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

This loads a[8]



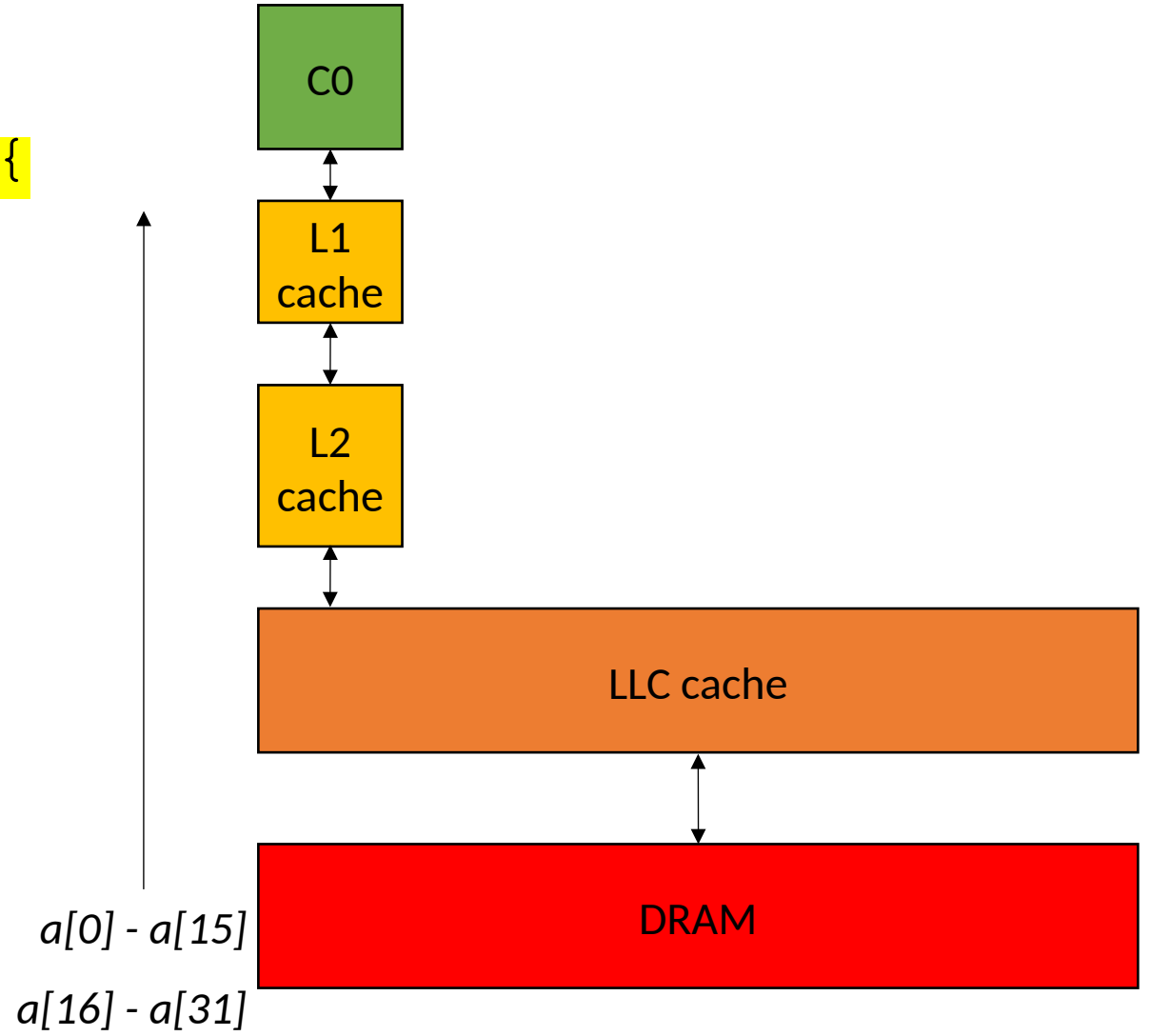
Assume a[0] is not in the cache

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.

Next time:

- Cache organization
- Cache coherence
- False Sharing