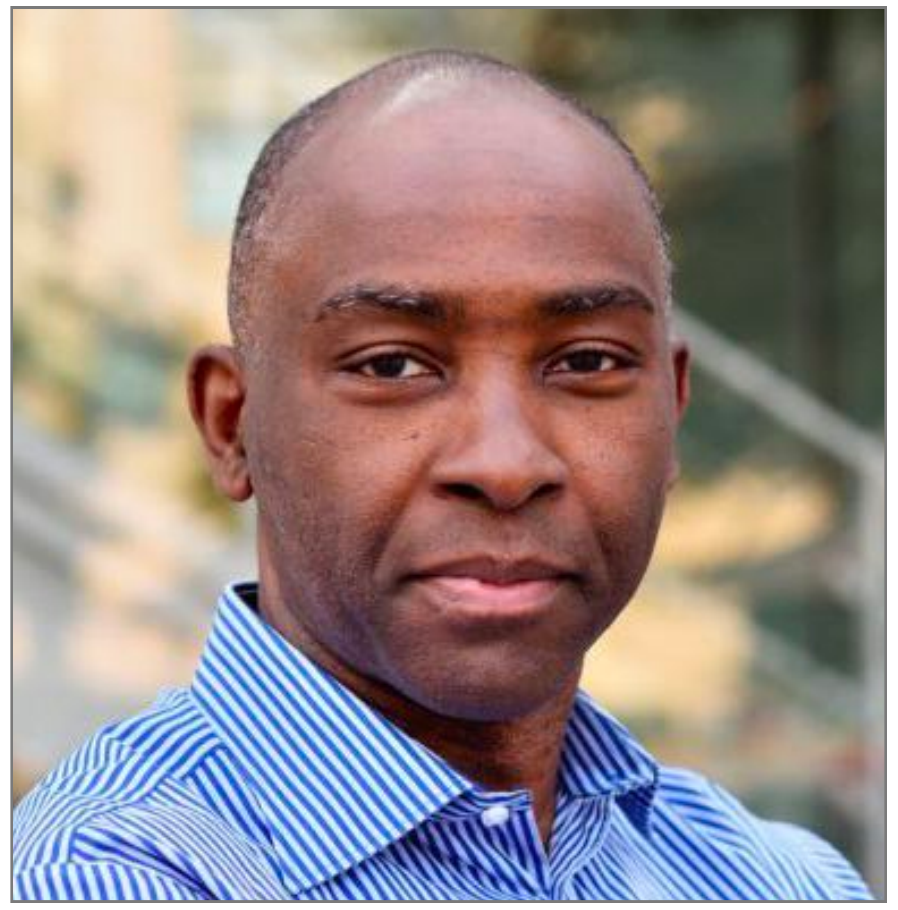
Lecture 1:

Why Parallelism?
Why Efficiency?

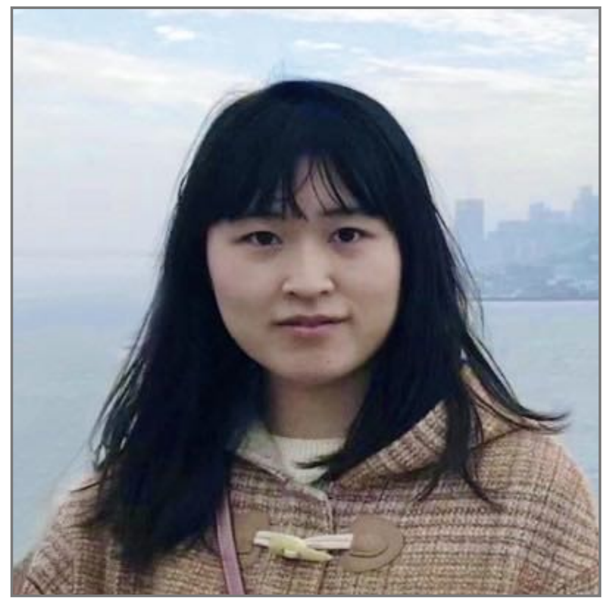
Parallel Computing
Stanford CS149, Fall 2023

Hello!

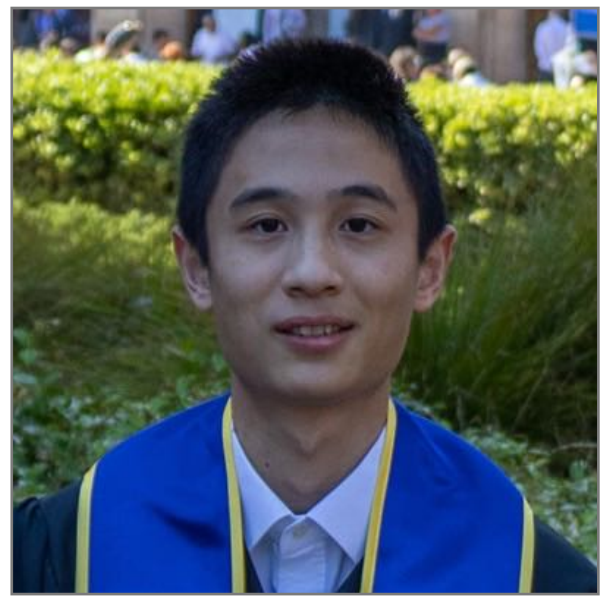
Prof. Kayvon


Prof. Olukotun

James


Minfei

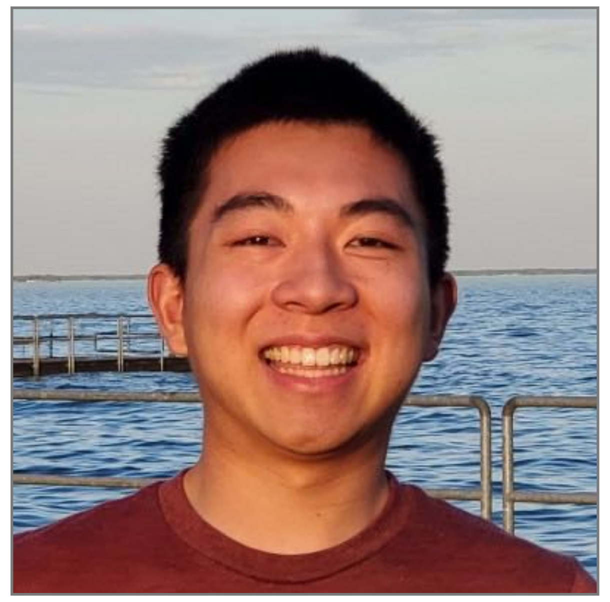
Yasmine


Senyang

Zhenbang

Neha

Michael


Jensen

Shiv

Tom

One common definition

A parallel computer is a collection of processing elements that cooperate to solve problems quickly

We care about performance and we care about efficiency

We're going to use multiple processors to get it

DEMO 1

(CS149 Fall 2023’s first parallel program)

Speedup

One major motivation of using parallel processing: achieve a speedup

For a given problem:

$$\text{speedup(using P processors)} = \frac{\text{execution time (using 1 processor)}}{\text{execution time (using P processors)}}$$

Class observations from demo 1

- Communication limited the maximum speedup achieved
  - In the demo, the communication was telling each other the partial sums

- Minimizing the cost of communication improved speedup
  - Moved students ("processors") closer together (or let them shout)

DEMO 2

(scaling up to four “processors”)

Class observations from demo 2

- **Imbalance in work assignment limited speedup**
  - **Some students ("processors") ran out work to do (went idle), while others were still working on their assigned task**

- **Improving the distribution of work improved speedup**

DEMO 3

(massively parallel execution)

Class observations from demo 3

- The problem I just gave you has a significant amount of communication compared to computation

- Communication costs can dominate a parallel computation, <u>severely limiting</u> speedup

Course theme 1:

Designing and writing parallel programs ... <u>that scale</u>!

- **Parallel thinking**
  1. Decomposing work into pieces that can safely be performed in parallel
  2. Assigning work to processors
  3. Managing communication/synchronization between the processors so that it does not limit speedup

- **Abstractions/mechanisms for performing the above tasks**
  - Writing code in popular parallel programming languages

Course theme 2:

Parallel computer hardware implementation: how parallel computers work

- Mechanisms used to implement abstractions efficiently
  - Performance characteristics of implementations
  - Design trade-offs: performance vs. convenience vs. cost

- Why do I need to know about hardware?
  - Because the characteristics of the machine really matter
    (recall speed of communication issues in earlier demos)
  - Because you care about efficiency and performance
    (you are writing parallel programs after all!)

Course theme 3:
Thinking about efficiency

- FAST != EFFICIENT

- Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently
  - Is 2x speedup on computer with 10 processors a good result?

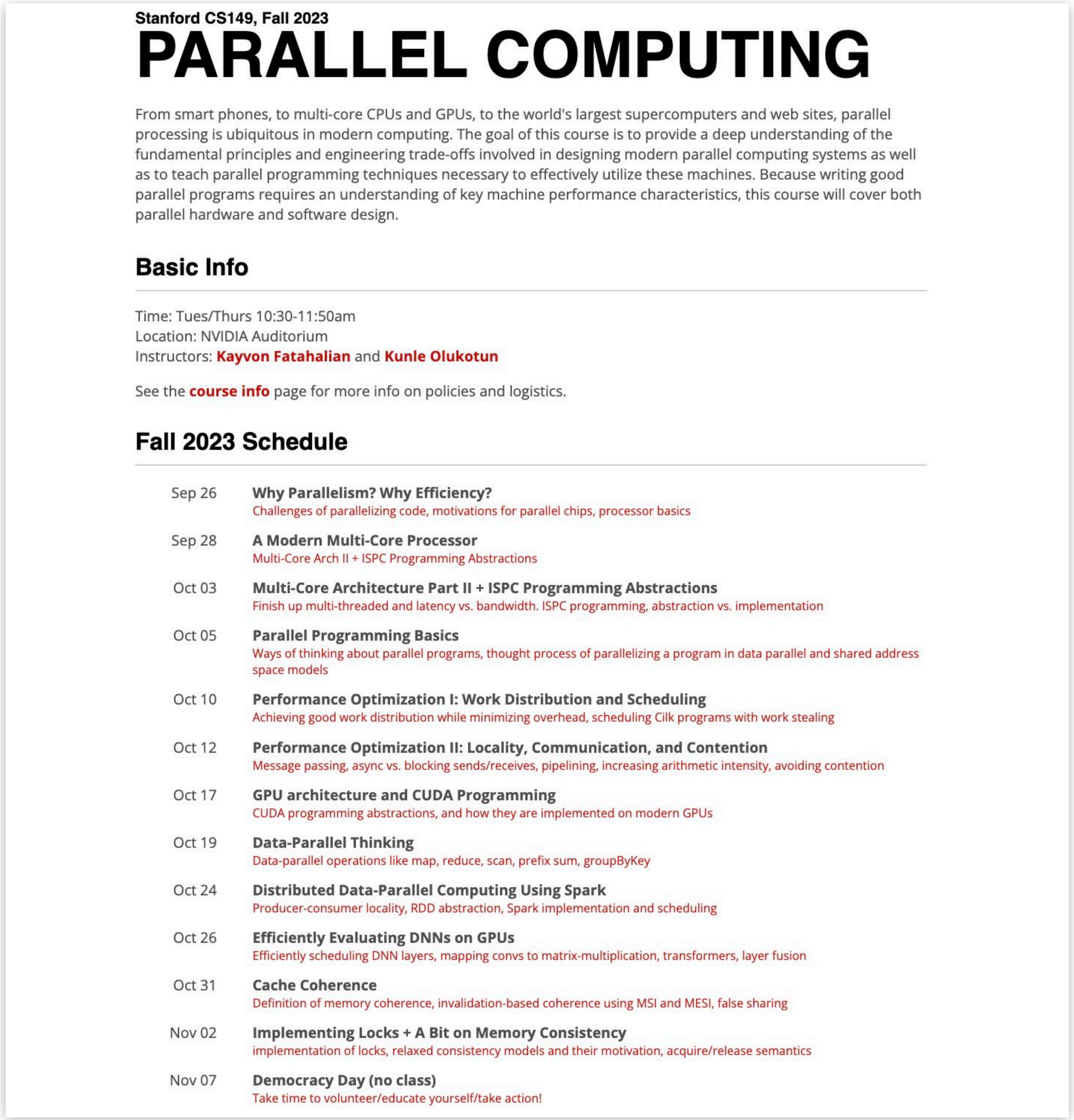
- Programmer's perspective: make use of provided machine capabilities

- HW designer's perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)

Course logistics

Getting started

- **The course web site**
  - **https://cs149.stanford.edu**
- **Textbook**
  - **There is no course textbook (the internet is plenty good these days), also see the course web site for suggested references**



Stanford CS149, Fall 2023

PARALLEL COMPUTING

From smart phones, to multi-core CPUs and GPUs, to the world's largest supercomputers and web sites, parallel processing is ubiquitous in modern computing. The goal of this course is to provide a deep understanding of the fundamental principles and engineering trade-offs involved in designing modern parallel computing systems as well as to teach parallel programming techniques necessary to effectively utilize these machines. Because writing good parallel programs requires an understanding of key machine performance characteristics, this course will cover both parallel hardware and software design.

Basic Info

Time: Tues/Thurs 10:30-11:50am
Location: NVIDIA Auditorium
Instructors: **Kayvon Fatahalian** and **Kunle Olukotun**

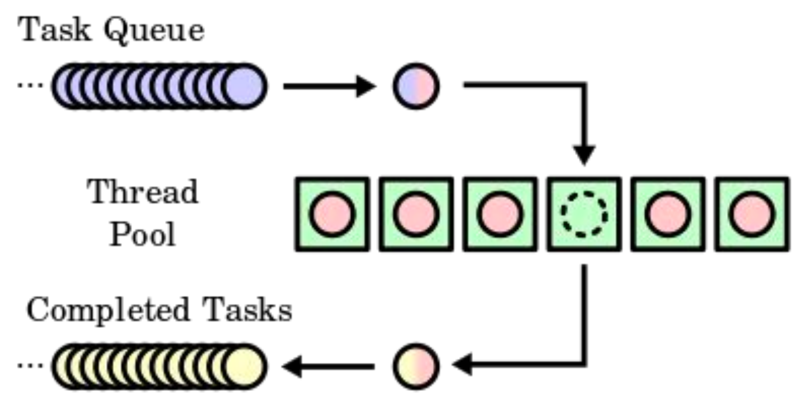
See the **course info** page for more info on policies and logistics.

Fall 2023 Schedule

Sep 26	**Why Parallelism? Why Efficiency?** Challenges of parallelizing code, motivations for parallel chips, processor basics
Sep 28	**A Modern Multi-Core Processor** Multi-Core Arch II + ISPC Programming Abstractions
Oct 03	**Multi-Core Architecture Part II + ISPC Programming Abstractions** Finish up multi-threaded and latency vs. bandwidth. ISPC programming, abstraction vs. implementation
Oct 05	**Parallel Programming Basics** Ways of thinking about parallel programs, thought process of parallelizing a program in data parallel and shared address space models
Oct 10	**Performance Optimization I: Work Distribution and Scheduling** Achieving good work distribution while minimizing overhead, scheduling Cilk programs with work stealing
Oct 12	**Performance Optimization II: Locality, Communication, and Contention** Message passing, async vs. blocking sends/receives, pipelining, increasing arithmetic intensity, avoiding contention
Oct 17	**GPU architecture and CUDA Programming** CUDA programming abstractions, and how they are implemented on modern GPUs
Oct 19	**Data-Parallel Thinking** Data-parallel operations like map, reduce, scan, prefix sum, groupByKey
Oct 24	**Distributed Data-Parallel Computing Using Spark** Producer-consumer locality, RDD abstraction, Spark implementation and scheduling
Oct 26	**Efficiently Evaluating DNNs on GPUs** Efficiently scheduling DNN layers, mapping convs to matrix-multiplication, transformers, layer fusion
Oct 31	**Cache Coherence** Definition of memory coherence, invalidation-based coherence using MSI and MESI, false sharing
Nov 02	**Implementing Locks + A Bit on Memory Consistency** implementation of locks, relaxed consistency models and their motivation, acquire/release semantics
Nov 07	**Democracy Day (no class)** Take time to volunteer/educate yourself/take action!

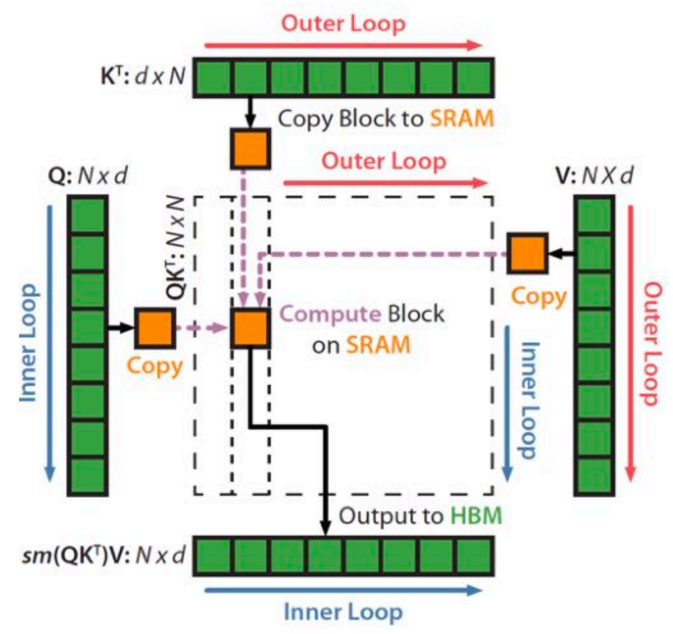
Four programming assignments

Assignment 1: ISPC programming on multi-core CPUs



Assignment 2: scheduling a task graph

Assignment 3: Writing a renderer in CUDA on NVIDIA GPUs



Assignment 4: chat149: flash-attention transformers for a mini language model

Optional assignment 5: (Can be used to boost a prior grade)

Topics TBD
programming FPGAs,
multi-core graph processing

Programming assignments can (optionally) be done with a partner.

We realize finding a partner can be stressful. 😖 😰

Fill out our partner request form by Thursday 11:59pm and we will find you a partner! 🥳 🤓

Written assignments

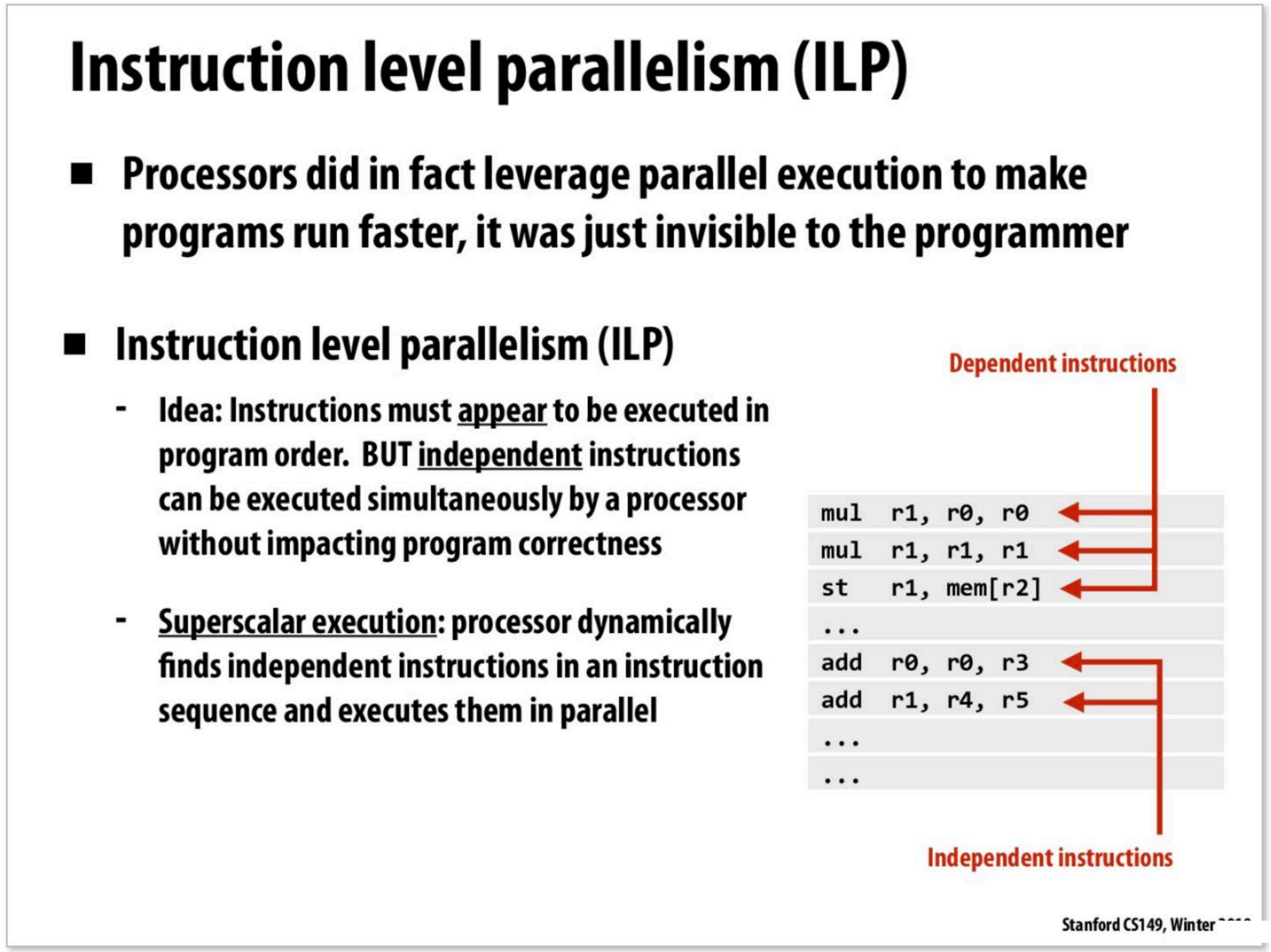
- Every two-weeks we will have a take-home written assignment graded on effort only

- Written assignments contain modified versions of previous exam questions, so they:
  - Give you practice with key course concepts
  - Provide practice for the style of questions you will see on an exam

Commenting and contributing to lectures

The website supports commenting on a per-slide basis

Why Parallelism? Why Efficiency?



Instruction level parallelism (ILP)

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer
- Instruction level parallelism (ILP)
  - Idea: Instructions must <u>appear</u> to be executed in program order. BUT <u>independent</u> instructions can be executed simultaneously by a processor without impacting program correctness
  - <u>Superscalar execution</u>: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel

```
mul  r1, r0, r0
mul  r1, r1, r1
st   r1, mem[r2]
...
add  r0, r0, r3
add  r1, r4, r5
...
...
```

Previous | **Next** --- Slide 30 of 48 Back to **Lecture Thumbnails**

rrastogi

It is computationally expensive for the processor to determine dependencies between instructions. The following PPT (slides 9/10) provides an example of how the number of checks grows with the number of instructions that are simultaneously dispatched: **http://www.cs.cmu.edu/afs/cs/academic/class/15740-f15/www/lectures/11-superscalar-pipelining.pdf**

This additional cost is likely one of the predominant reasons that ILP has plateaued at 4 simultaneous instructions. To circumvent this issue, architects have tried to force the compiler to solve the dependency issue using VLIW (very long instruction word). To summarize VLIW, if a processor contains 5 independent execution units, the compiler will have 5 operations in the "very long instruction word" that the processor will map to the 5 execution units: **https://en.wikipedia.org/wiki/Very_long_instruction_word**. This way dependency checking is the responsibility of software and not hardware.

I am not sure if VLIW has helped significantly pushed the four simultaneous instruction threshold though. If somebody knows, please share.

kayvonf

Question: The key phrase on this slide is that a processor must execute instructions in a manner "appears" as if they were executed in program order. **This is a *key idea* in this class.**

What is program order?

And what does it mean for the results of a program's execution *to appear* as if instructions were executed in program order?

And finally... Why is the program order guarantee a useful one? (What if the results of execution were inconsistent with the results that would be obtained if the instructions were executed in program order?)

void

> *And what does it mean for the results of a program's execution to appear as if instructions were executed in program order?*

A programmer might write something like the code below.

```
x = a + b
print(x)
y = c + d
print(y)
```

Participation (comments)

- You are asked to submit one <u>well-thought-out</u> comment per lecture
  - Only two comments per week
  - We expect you to submit "within the same calendar week" as the lectures (no credit for submitting all comments at the end of the quarter when you are studying for the final)

- Why do we ask you to write?
  - Because writing is a way many good architects and systems designers force themselves to think (explaining clearly and thinking clearly are highly correlated!)

- But take it seriously, there is a participation component to the final grade

What we are looking for in comments

- **Try to explain the slide (as if you were trying to teach your classmate while studying for an exam)**
  - "The instructor said this, but if you think about it this way instead it makes much more sense... "
- **Explain what is confusing to you:**
  - "What I'm totally confused by here was..."
- **Challenge classmates with a question**
  - For example, make up a question you think might be on an exam.
- **Provide a link to an alternate explanation**
  - "This site has a really good description of how multi-threading works..."
- **Mention real-world examples**
  - For example, describe all the parallel hardware components in the PS5
- **Constructively respond to another student's comment or question**
  - "@segfault23, are you sure that is correct? I thought that Prof. Kayvon said…"
- **It is OKAY (and even encouraged) to address the same topic (or repeat someone else's summary, explanation or idea) in your own words**
  - "@funkysenior23's point is that the overhead of communication..."

Grades

58% Programming assignments (4)

8% Written assignments (5)

16% Midterm exam

- An evening in-person exam on Nov 14th

16% Final exam

- During the university-assigned slot: Dec 14th, 3:30pm

2% Asynchronous participation (website comments)

Late days

- You get eight late days for the quarter
  - For use on programming and written assignments

- The idea of late days is to give you the flexibility to handle almost all events that arise throughout the quarter
  - Work from other classes, failing behind, most illnesses, athletic/extra curricular events…
  - We expect to give extra late days only under exceptional circumstances

- Requests for additional late days for exceptional circumstances should be made days in advance if possible.

Why parallelism?

Some historical context: why avoid parallel processing?

- Single-threaded CPU performance doubling ~ every 18 months
- Implication: working to parallelize your code was often not worth the time
  - Software developer does nothing, code gets faster next year. Woot!

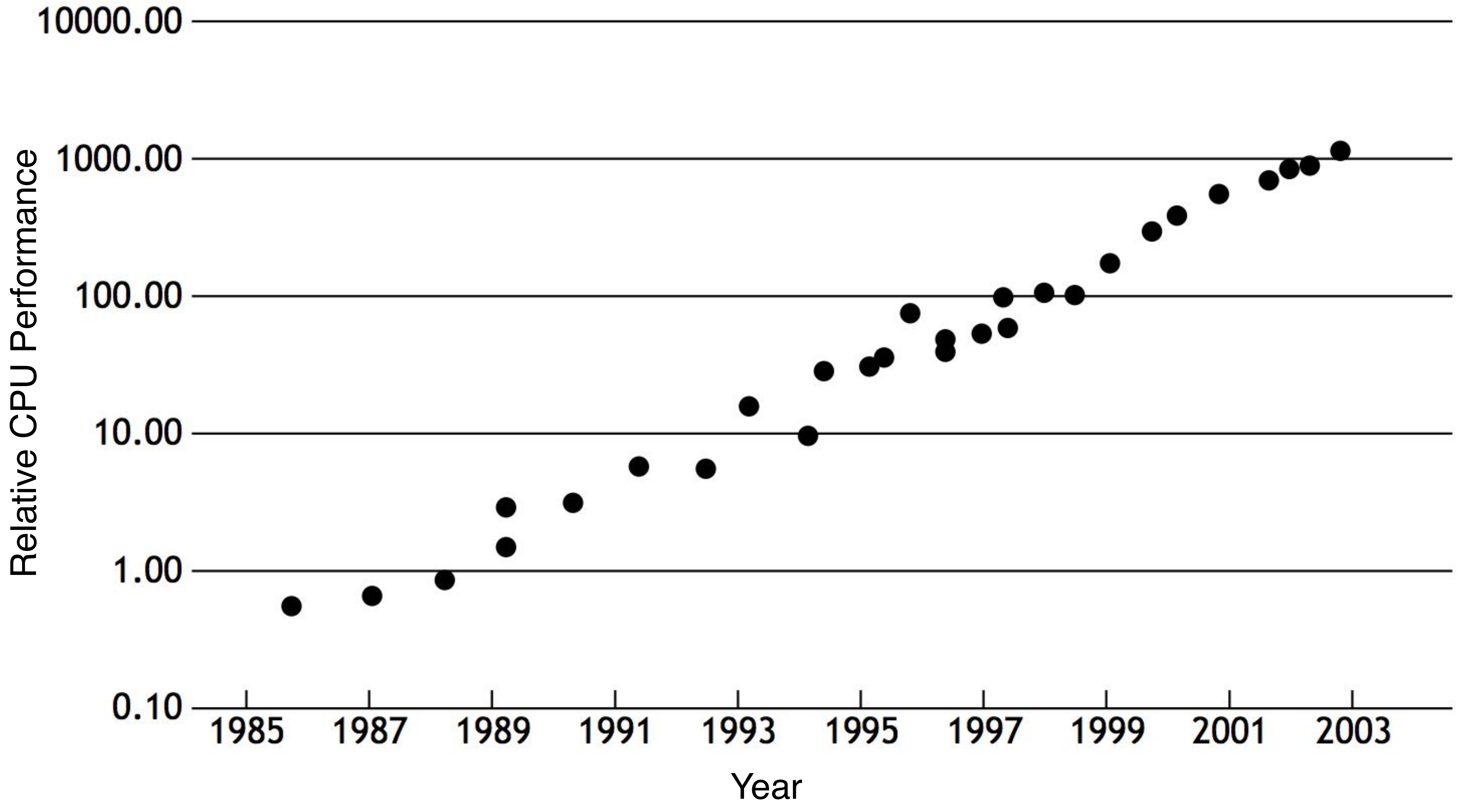


Image credit: Olukutun and Hammond, ACM Queue 2005

Until ~15 years ago: two significant reasons for processor performance improvement

1. Exploiting instruction-level parallelism (superscalar execution)

2. Increasing CPU clock frequency

What is a computer program?

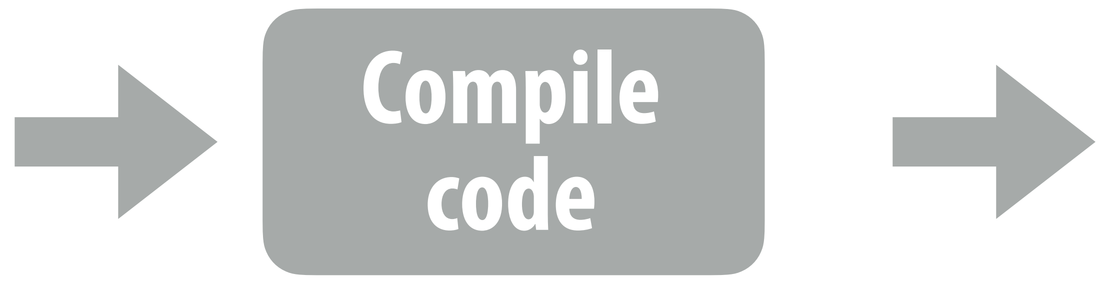
Here is a program written in C

```c
int main(int argc, char** argv) {

    int x = 1;

    for (int i=0; i<10; i++) {
       x = x + x;
    }

    printf(“%d\n”, x);

    return 0;
}
```

What is a program? (from a processor’s perspective)

A program is just a list of processor instructions!

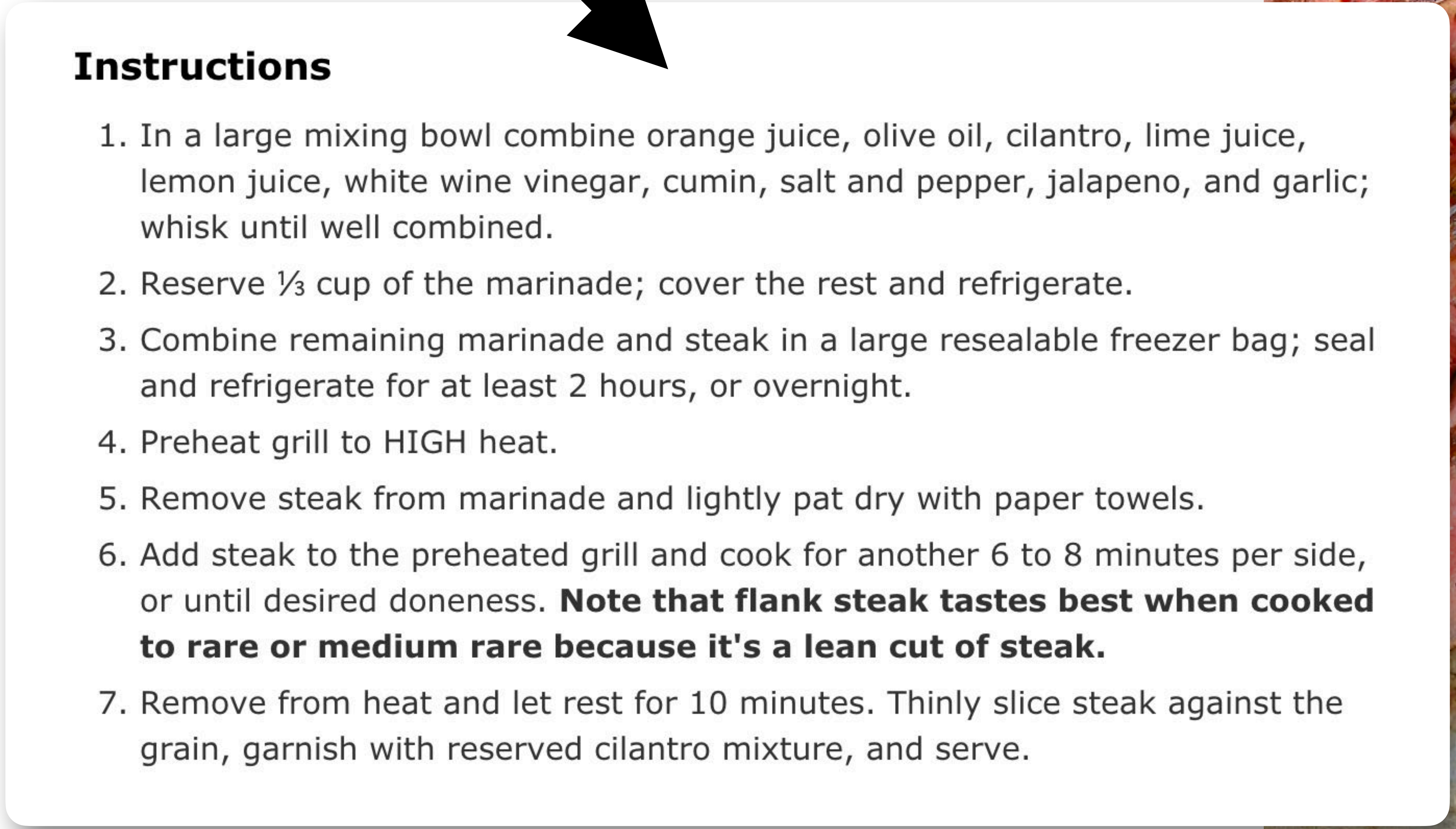
```
int main(int argc, char** argv) {

  int x = 1;

  for (int i=0; i<10; i++) {
    x = x + x;
  }

  printf(“%d\n”, x);

  return 0;
}
```



```
_main:
100000f10:    pushq    %rbp
100000f11:    movq %rsp, %rbp
100000f14:    subq $32, %rsp
100000f18:    movl $0, -4(%rbp)
100000f1f:    movl %edi, -8(%rbp)
100000f22:    movq %rsi, -16(%rbp)
100000f26:    movl $1, -20(%rbp)
100000f2d:    movl $0, -24(%rbp)
100000f34:    cmpl $10, -24(%rbp)
100000f38:    jge  23 <_main+0x45>
100000f3e:    movl -20(%rbp), %eax
100000f41:    addl -20(%rbp), %eax
100000f44:    movl %eax, -20(%rbp)
100000f47:    movl -24(%rbp), %eax
100000f4a:    addl $1, %eax
100000f4d:    movl %eax, -24(%rbp)
100000f50:    jmp  -33 <_main+0x24>
100000f55:    leaq 58(%rip), %rdi
100000f5c:    movl -20(%rbp), %esi
100000f5f:    movb $0, %al
100000f61:    callq    14
100000f66:    xorl %esi, %esi
100000f68:    movl %eax, -28(%rbp)
100000f6b:    movl %esi, %eax
100000f6d:    addq $32, %rsp
100000f71:    popq %rbp
100000f72:    rets
```

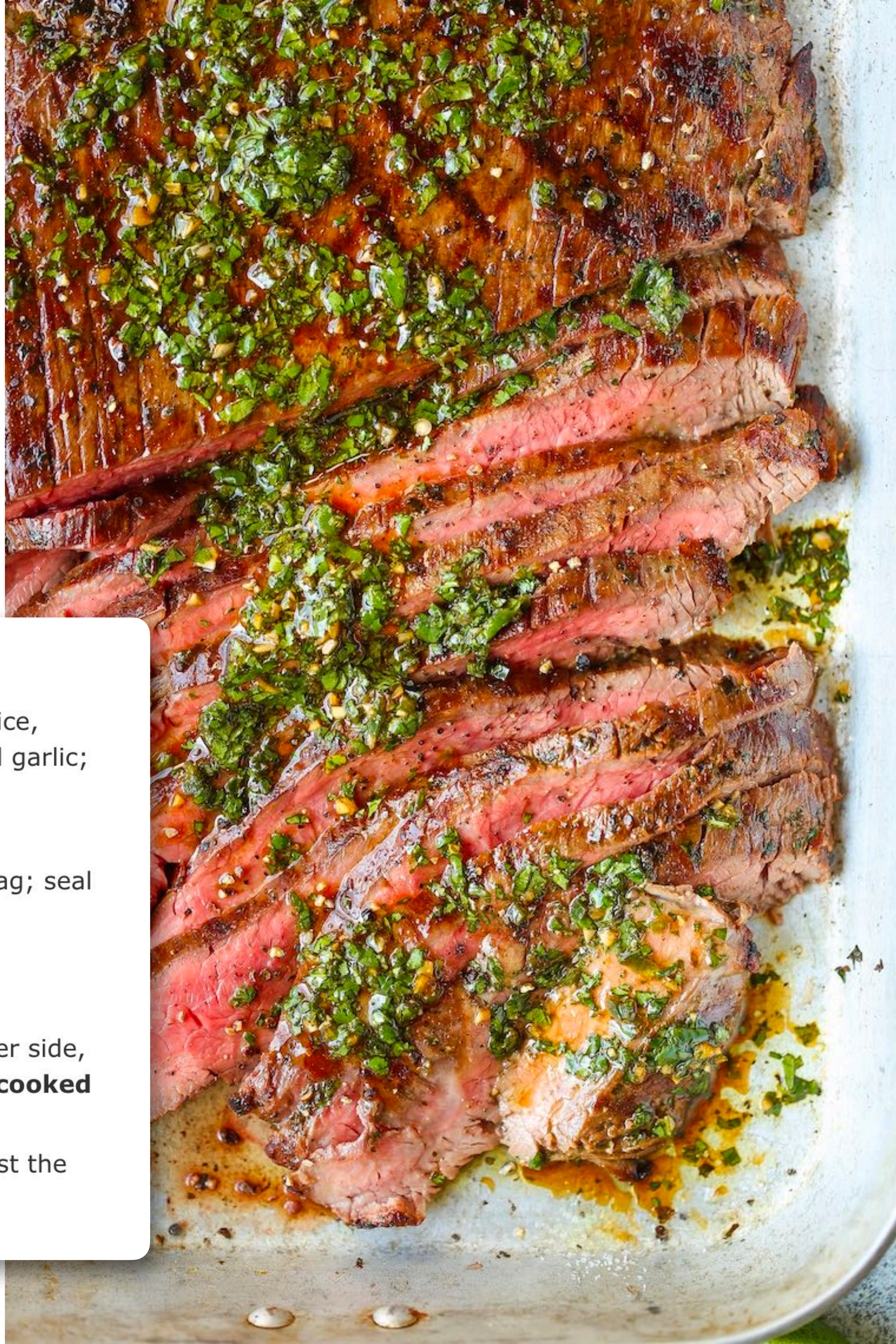
Kind of like the instructions in a recipe for your favorite meals

Mmm, carne asada



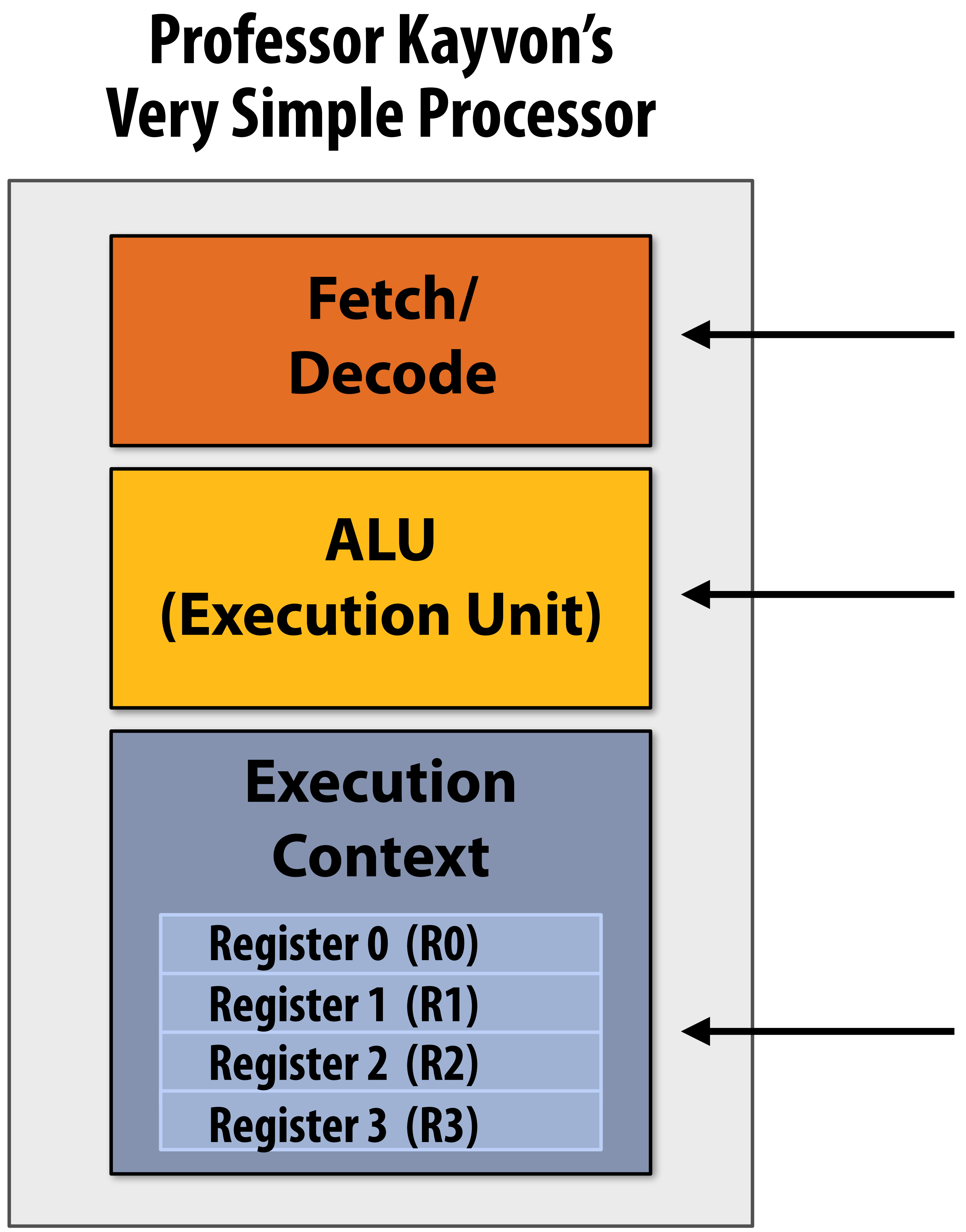
Instructions

1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeno, and garlic; whisk until well combined.
2. Reserve ⅓ cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and steak in a large resealable freezer bag; seal and refrigerate for at least 2 hours, or overnight.
4. Preheat grill to HIGH heat.
5. Remove steak from marinade and lightly pat dry with paper towels.
6. Add steak to the preheated grill and cook for another 6 to 8 minutes per side, or until desired doneness. **Note that flank steak tastes best when cooked to rare or medium rare because it's a lean cut of steak.**
7. Remove from heat and let rest for 10 minutes. Thinly slice steak against the grain, garnish with reserved cilantro mixture, and serve.



What does a processor do?

A processor executes instructions

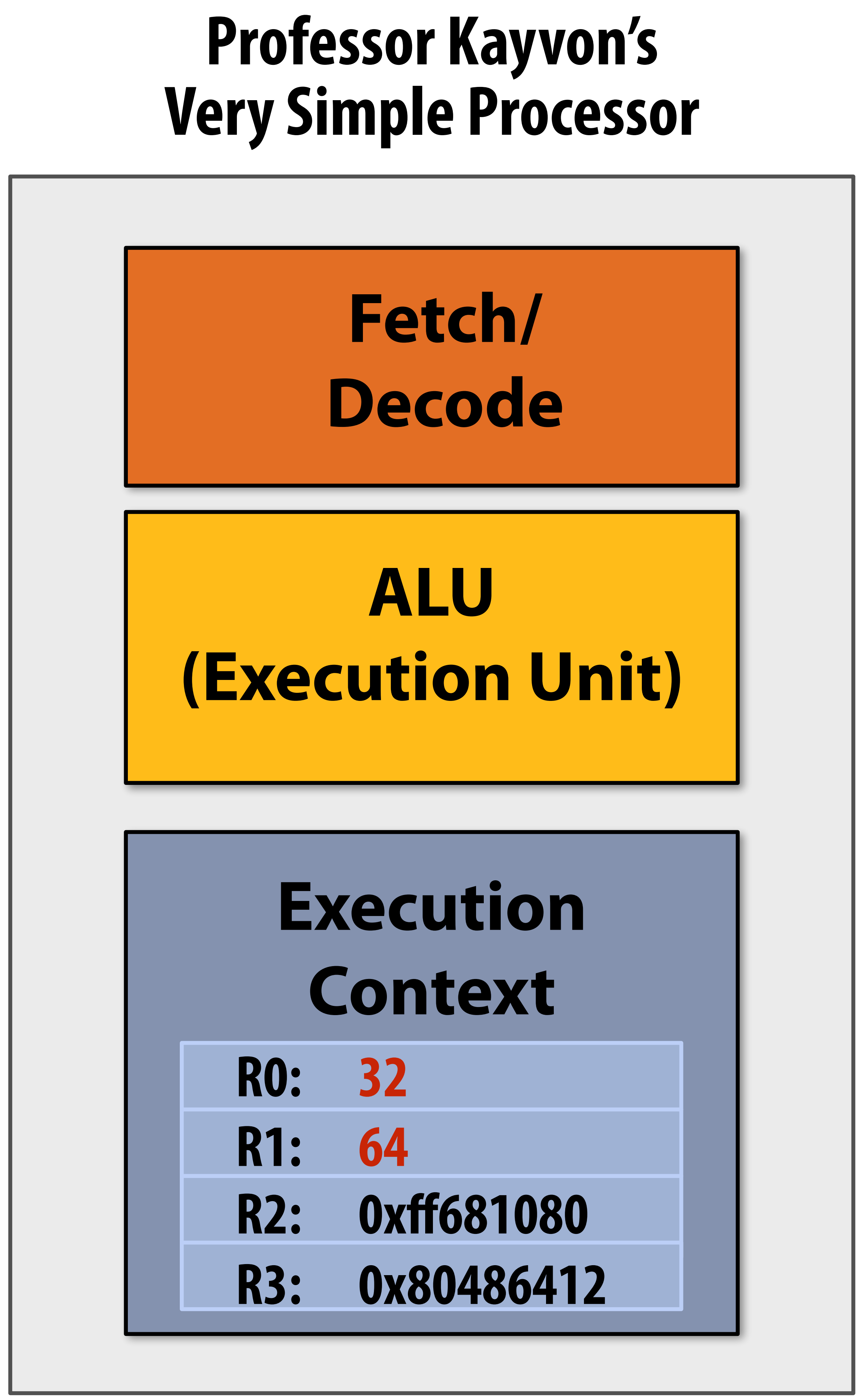


Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations

One example instruction: add two numbers



Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

```
add R0 ← R0, R1
```

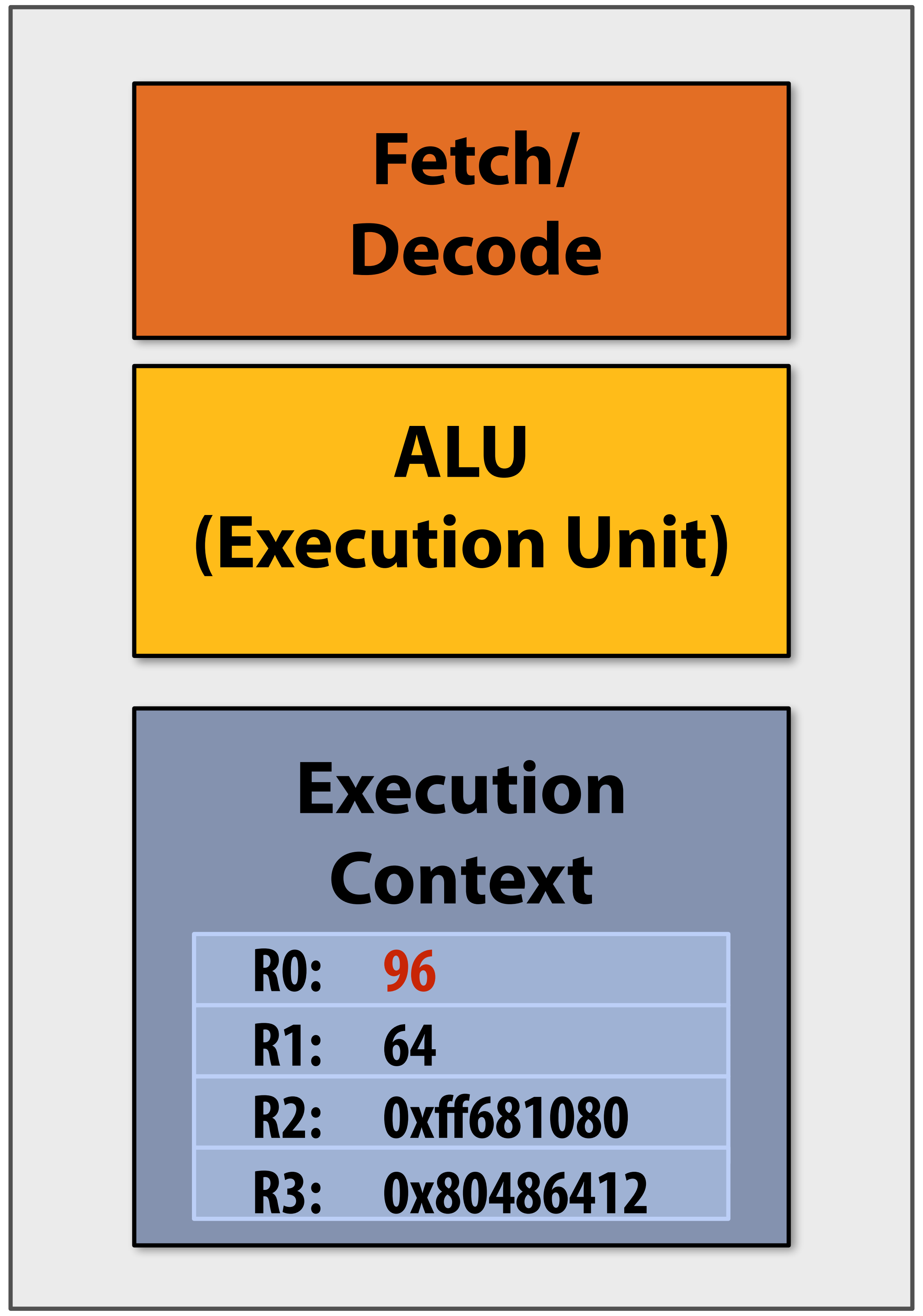
"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

One example instruction: add two numbers

Professor Kayvon's
Very Simple Processor



Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

```
add R0 ← R0, R1
```

"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

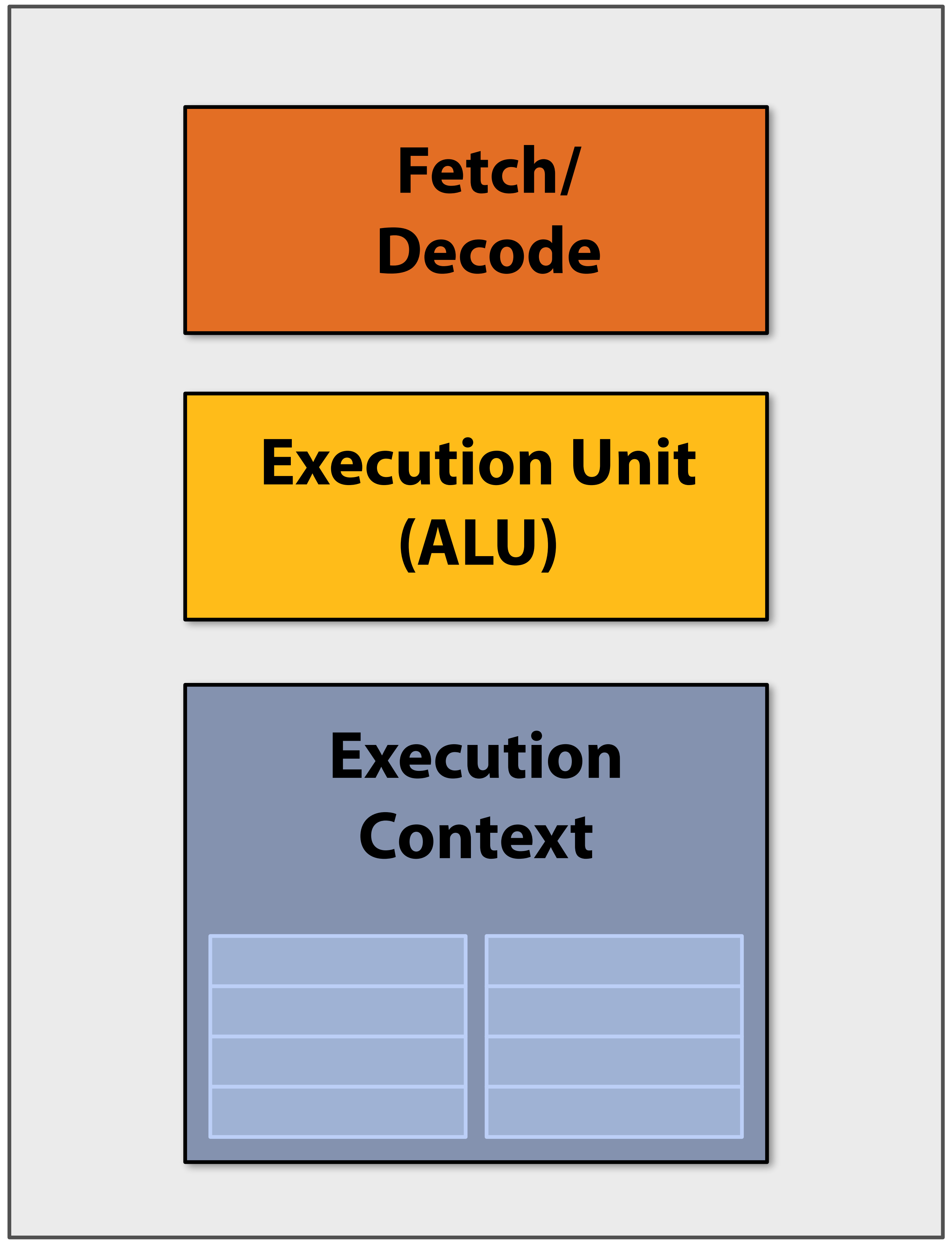
Step 2:
Get operation inputs from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

Step 4:
Store result 96 **back to register R0**

Execute program

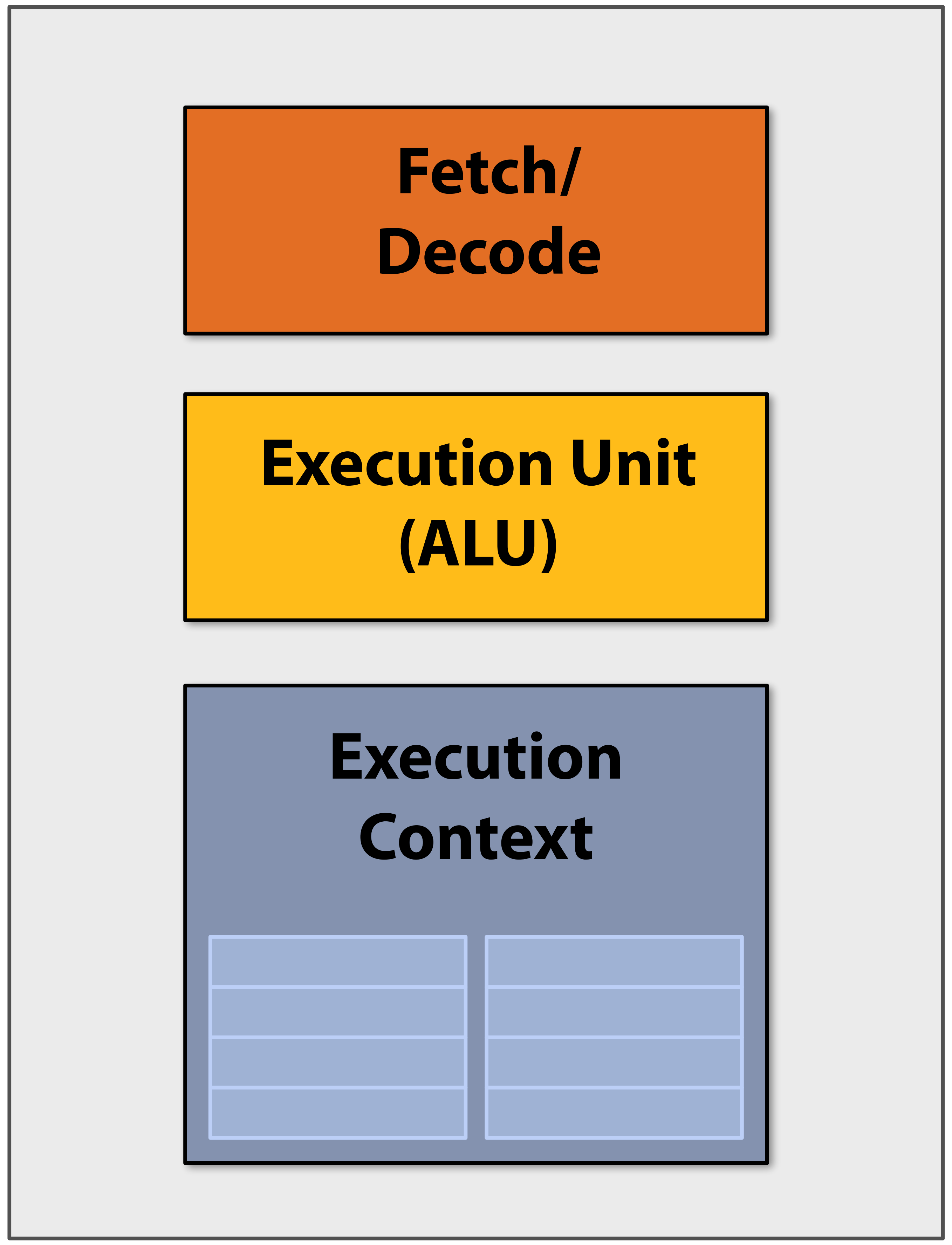
My very simple processor: executes one instruction per clock

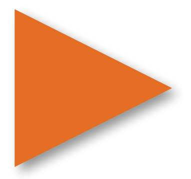


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock

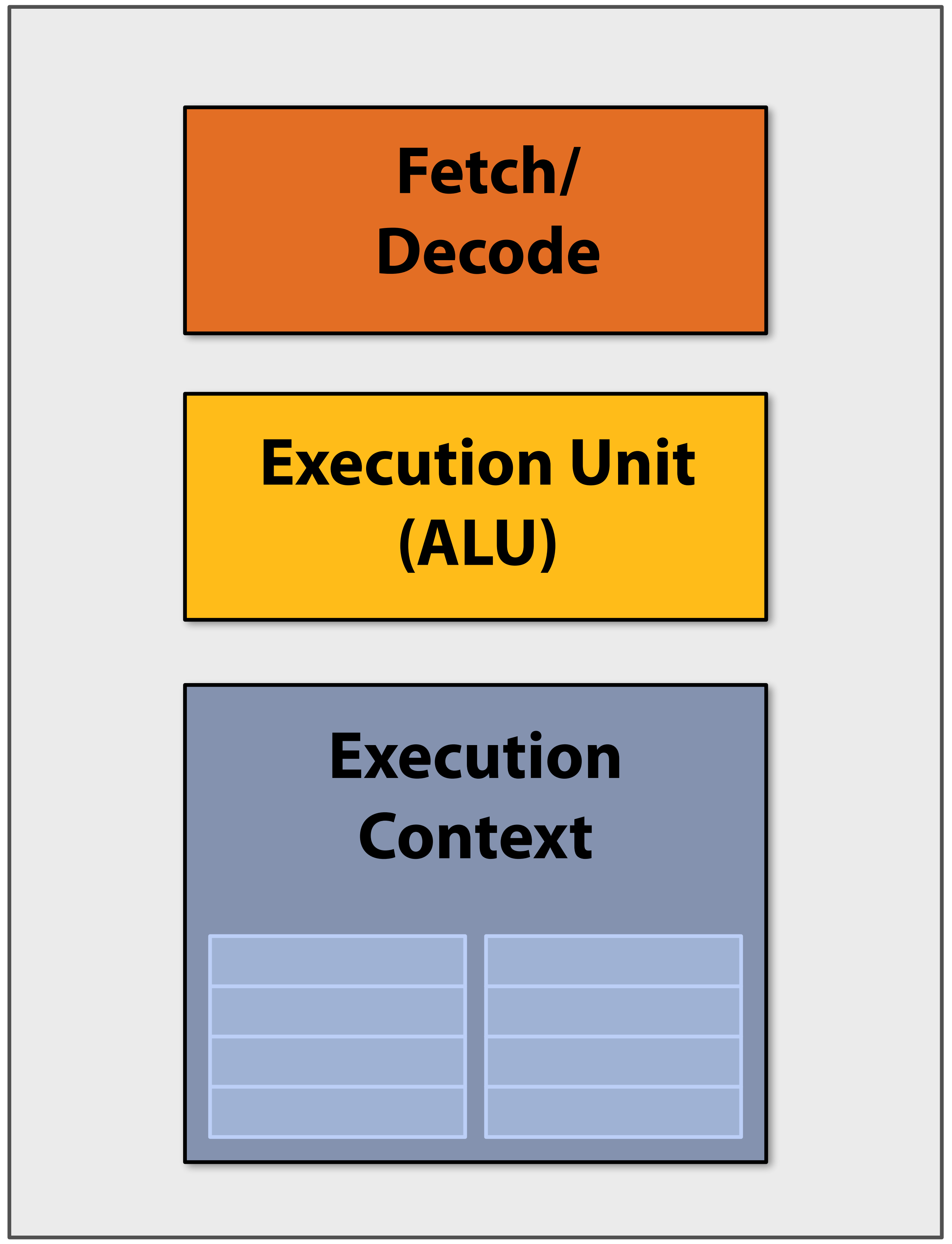




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

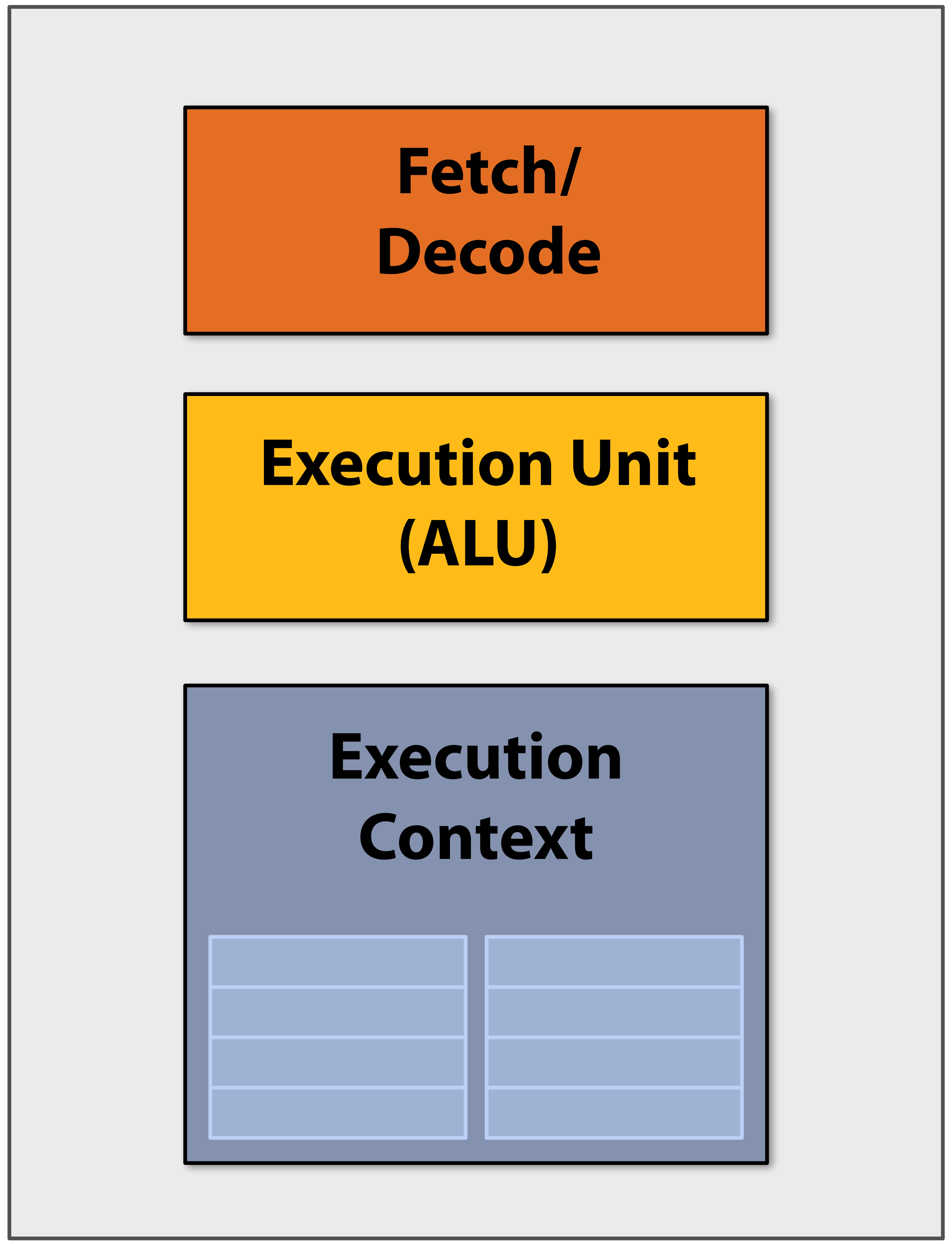
My very simple processor: executes one instruction per clock



```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Execute program

My very simple processor: executes one instruction per clock



```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Review of how computers work…

What is a computer program? (from a processor's perspective)

It is a list of instructions to execute!

What is an instruction?

It describes an operation for a processor to perform.

Executing an instruction typically modifies the computer's state.

What do I mean when I talk about a computer's "state"?

The values of program data, which are stored in a processor's registers or in memory.

Lets consider a very simple piece of code

```
a = x*x + y*y + z*z
```

Consider the following five instruction program:

Assume register R0 = x, R1 = y, R2 = z

```
mul R0, R0, R0
mul R1, R1, R1
mul R2, R2, R2
add R0, R0, R1
add R3, R0, R2
```

R3 now stores value of program variable ‘a’

This program has five instructions, so it will take five clocks to execute, correct?

Can we do better?

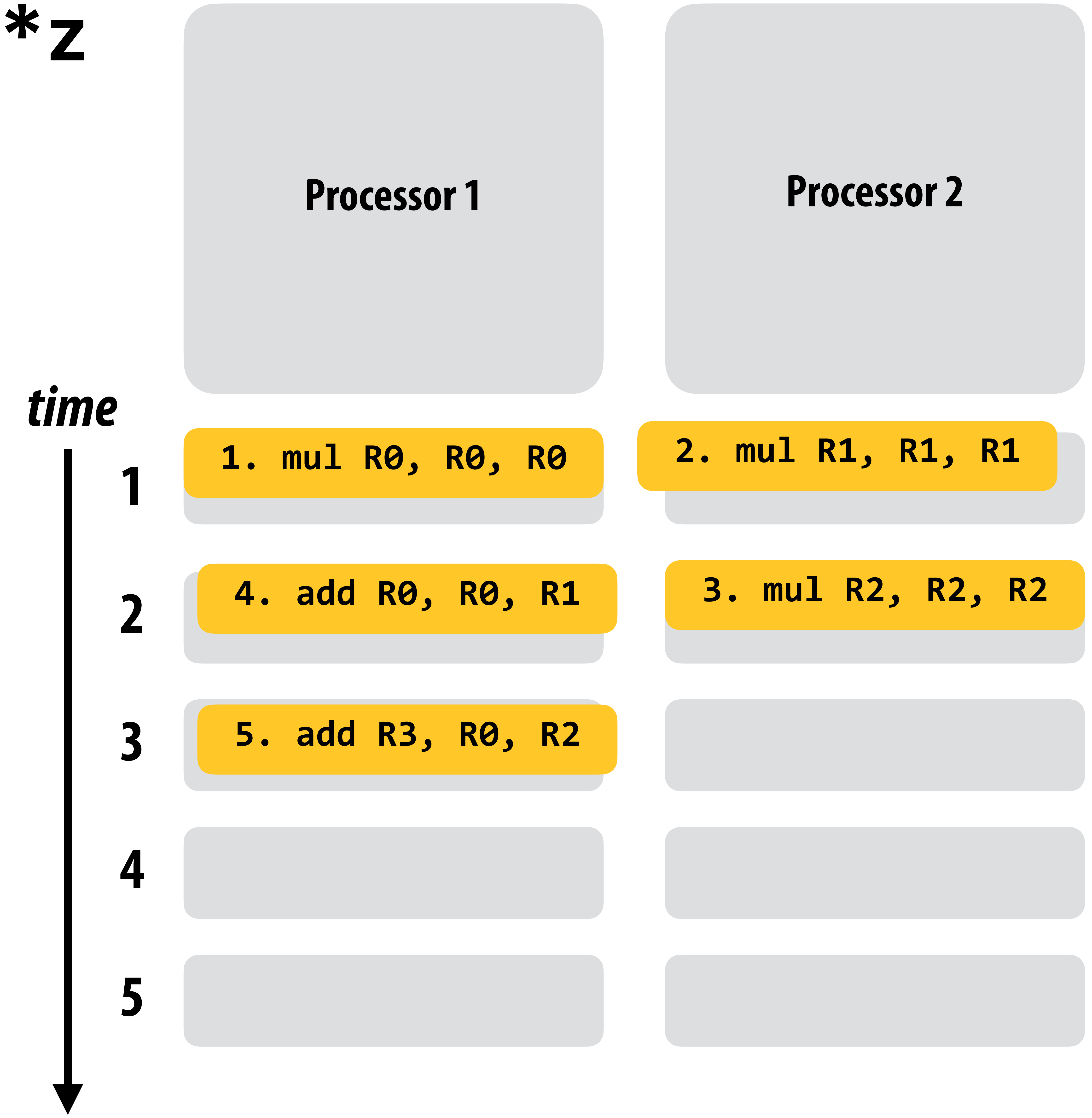
What if up to two instructions can be performed at once?

```
a = x*x + y*y + z*z
```

Assume register
R0 = x, R1 = y, R2 = z

```
mul R0, R0, R0
mul R1, R1, R1
mul R2, R2, R2
add R0, R0, R1
add R3, R0, R2
```

R3 now stores value of program variable 'a'



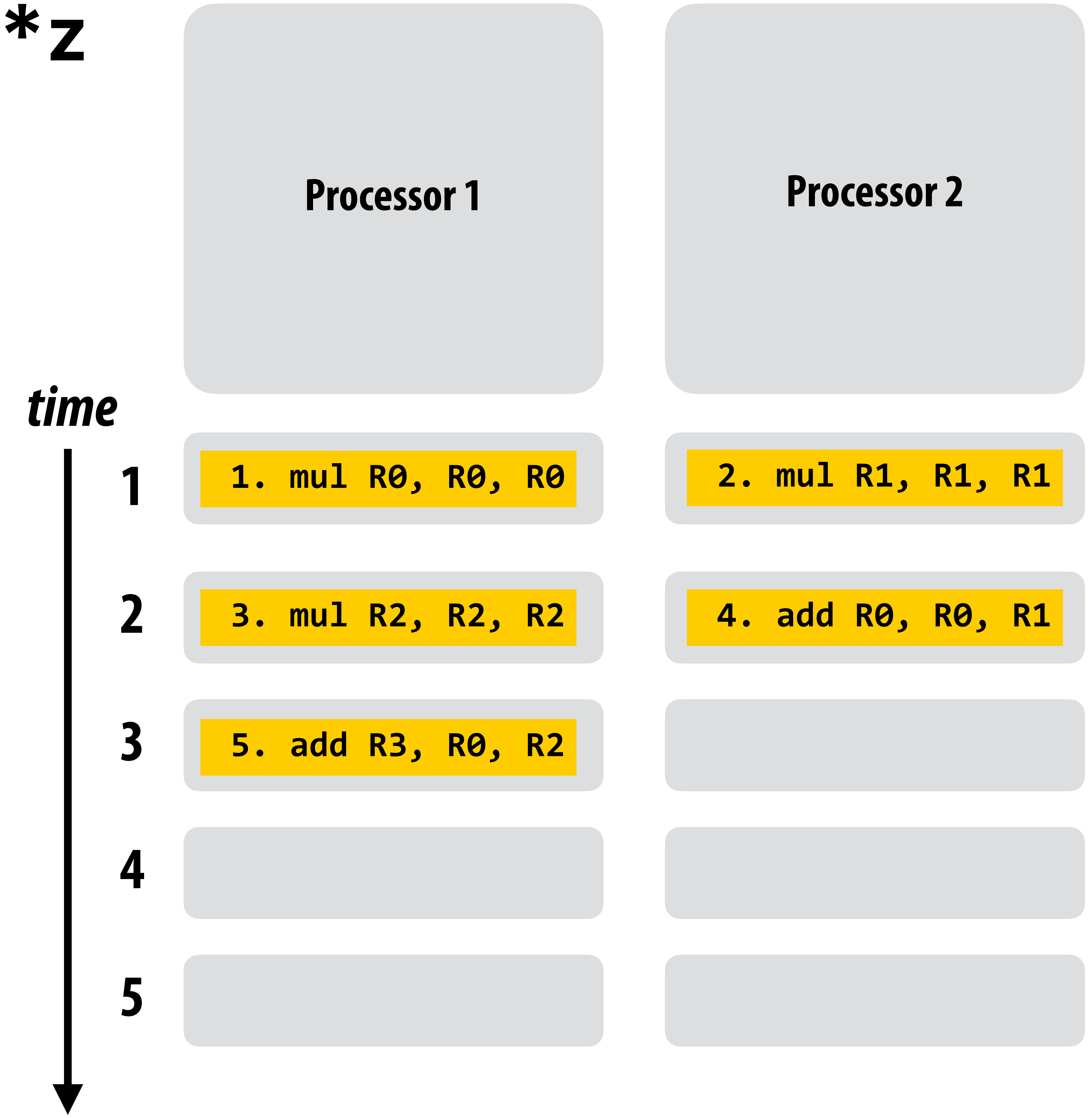
What if up to two instructions can be performed at once?

```
a = x*x + y*y + z*z
```

Assume register
R0 = x, R1 = y, R2 = z

```
1 mul R0, R0, R0
2 mul R1, R1, R1
3 mul R2, R2, R2
4 add R0, R0, R1
5 add R3, R0, R2
```

R3 now stores value of program variable 'a'



What does it mean for our parallel to scheduling to that "respects program order"?

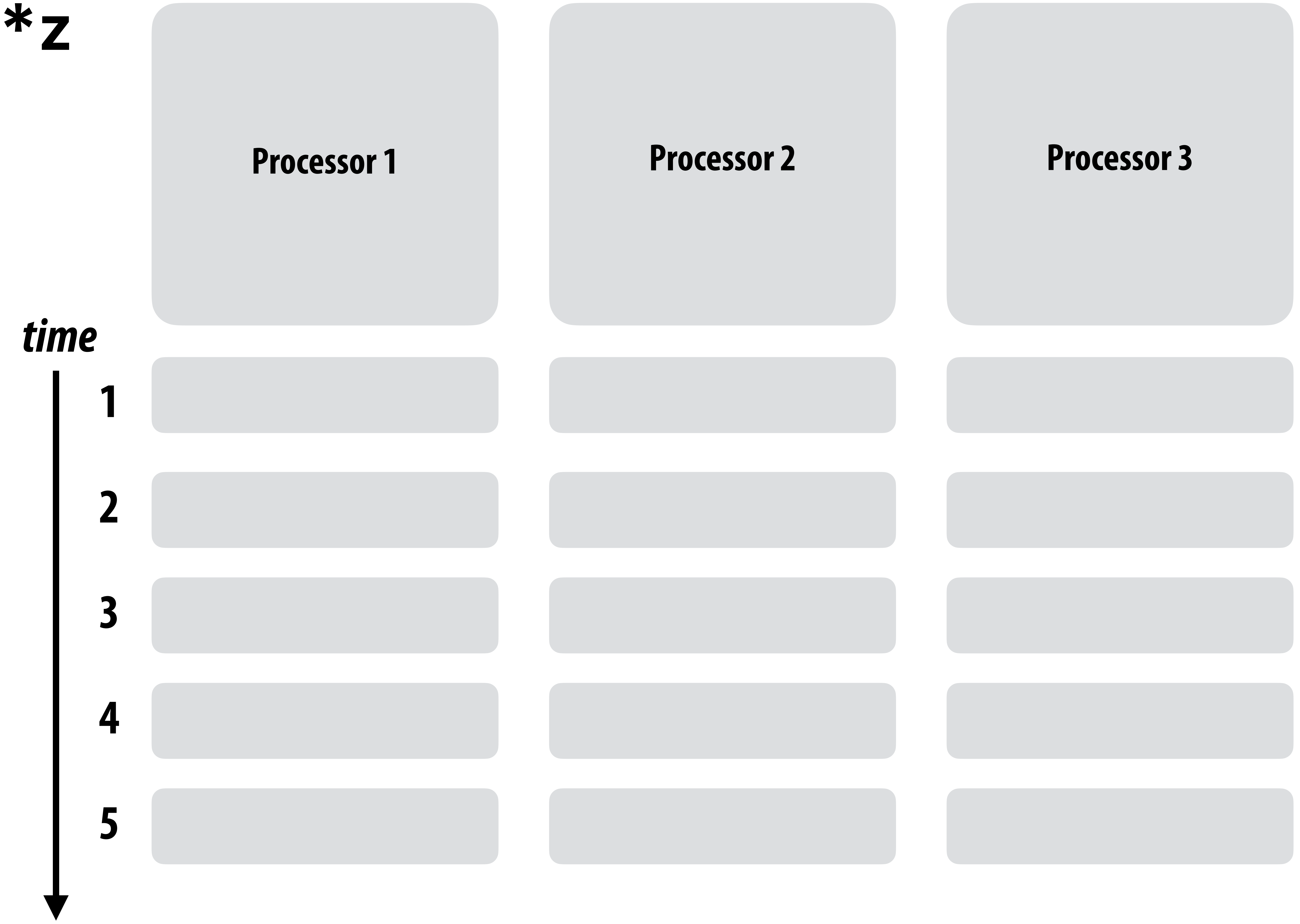
What about three instructions at once?

```
a = x*x + y*y + z*z
```

Assume register
R0 = x, R1 = y, R2 = z

```
mul R0, R0, R0
mul R1, R1, R1
mul R2, R2, R2
add R0, R0, R1
add R3, R0, R2
```

R3 now stores value of program variable 'a'



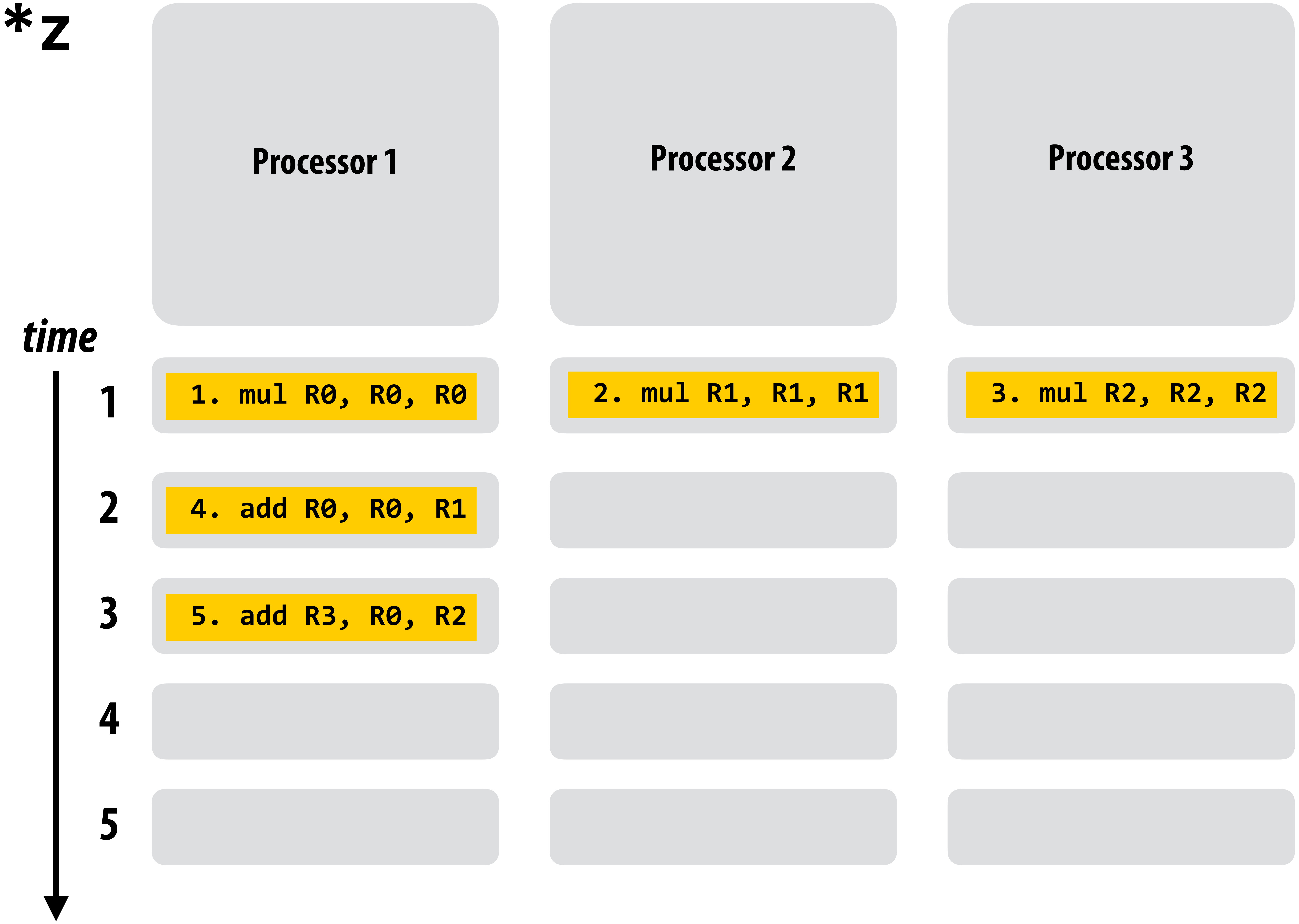
What about three instructions at once?

```
a = x*x + y*y + z*z
```

Assume register
R0 = x, R1 = y, R2 = z

```
1 mul R0, R0, R0
2 mul R1, R1, R1
3 mul R2, R2, R2
4 add R0, R0, R1
5 add R3, R0, R2
```

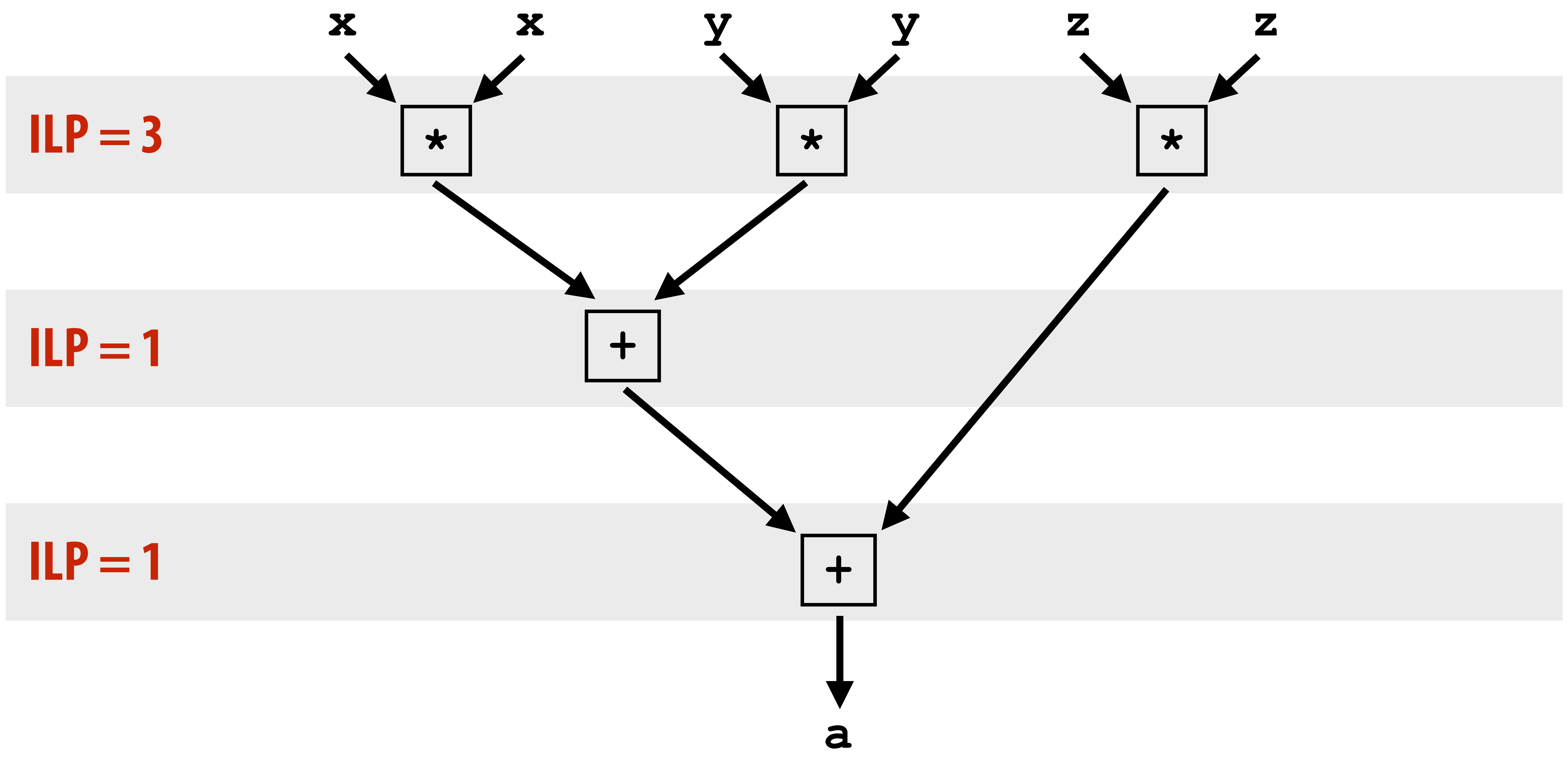
R3 now stores value of program variable ‘a’



Instruction level parallelism (ILP) example

- ILP = 3

```
a = x*x + y*y + z*z
```



Superscalar processor execution

```
a = x*x + y*y + z*z
```

```
Assume register
R0 = x, R1 = y, R2 = z
```

```
1 mul R0, R0, R0
2 mul R1, R1, R1
3 mul R2, R2, R2
4 add R0, R0, R1
5 add R3, R0, R2
```

Idea #1:

Superscalar execution: processor automatically finds* independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel without impacting program correctness (on a superscalar processor that determines that the lack of dependencies exists)

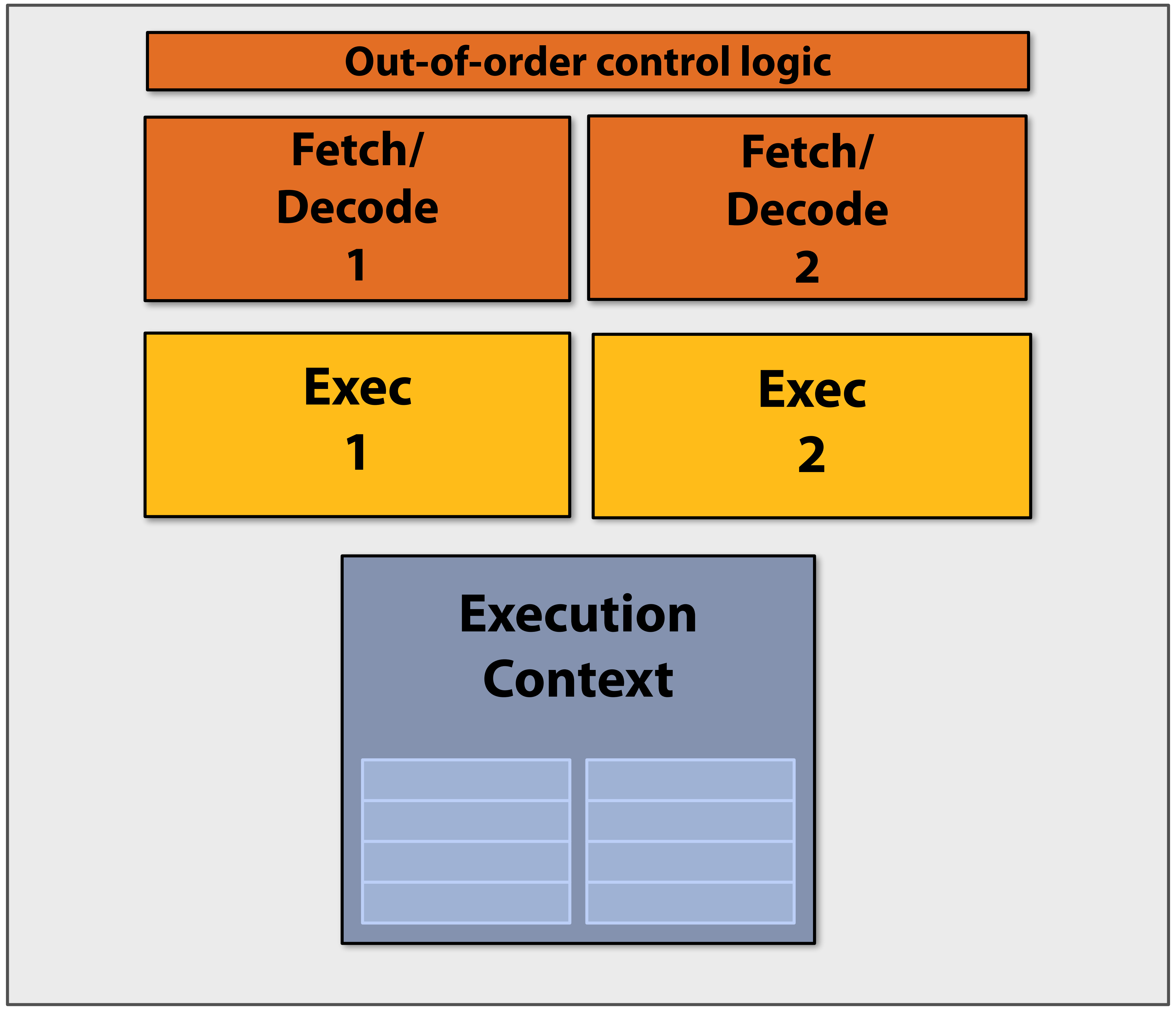
But instruction 4 must be executed after instructions 1 and 2

And instruction 5 must be executed after instruction 4

* Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.

Superscalar processor

This processor can decode and execute up to two instructions per clock



Aside:
Old Intel Pentium 4 CPU

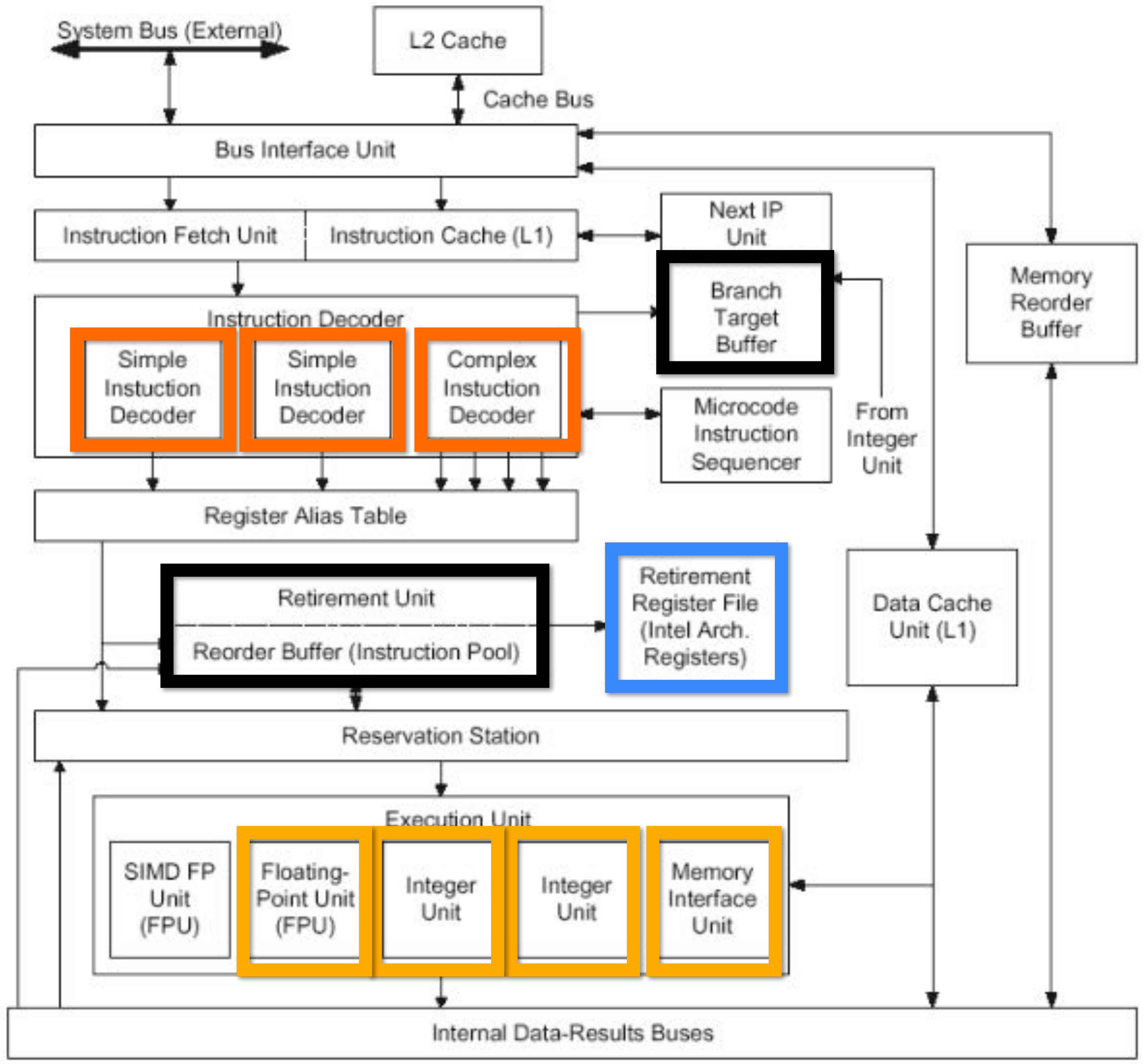


Image credit: http://ixbtlabs.com/articles/pentium4/index.html

A more complex example

Program (sequence of instructions)

value during execution

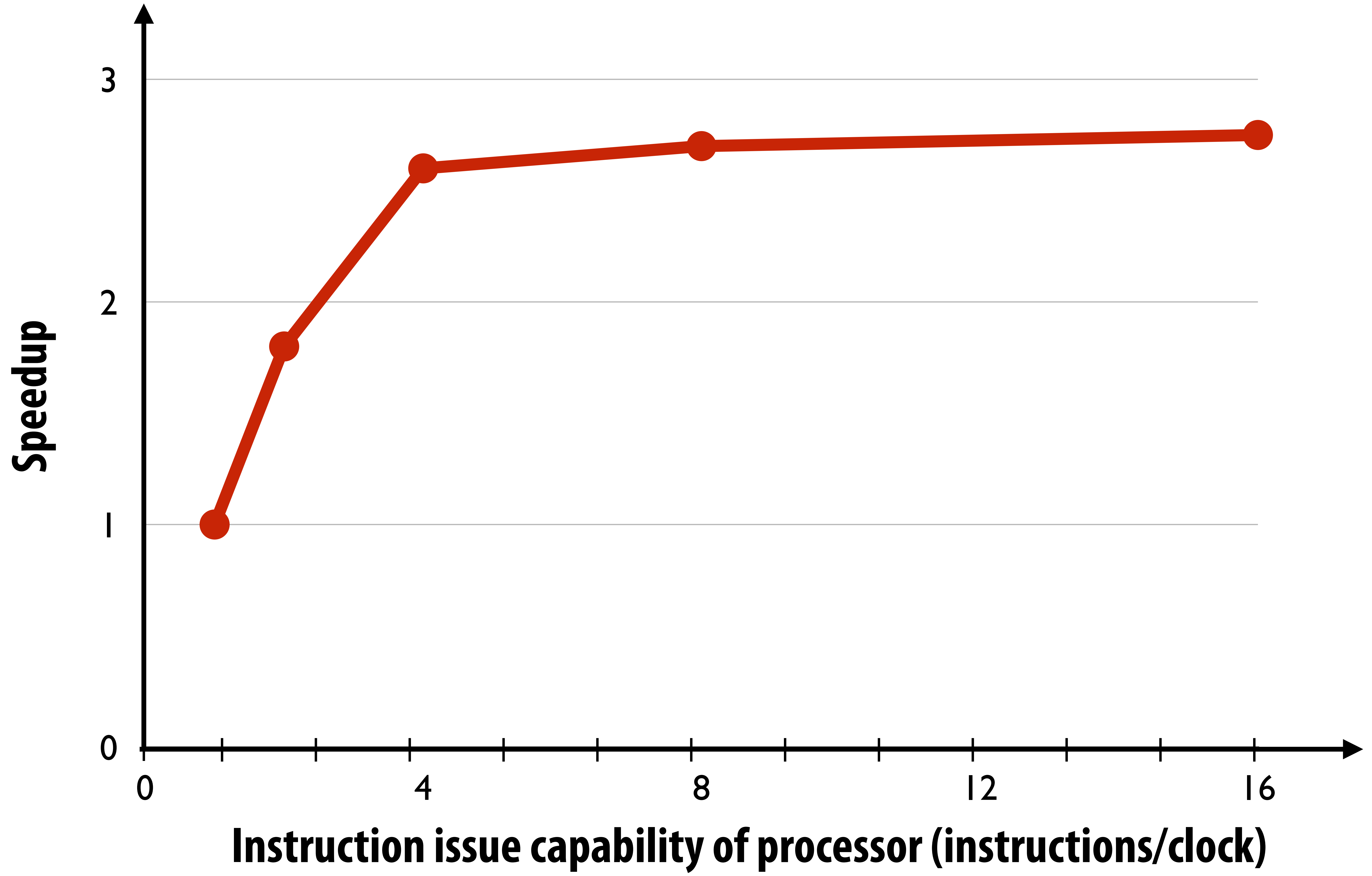
```
PC    Instruction
----------------------------------------
00    a = 2
01    b = 4

02    tmp2 = a + b          // 6
03    tmp3 = tmp2 + a       // 8
04    tmp4 = b + b          // 8
05    tmp5 = b * b          // 16
06    tmp6 = tmp2 + tmp4    // 14
07    tmp7 = tmp5 + tmp6    // 30

08    if (tmp3 > 7)
09      print tmp3
      else
10      print tmp7
```

Instruction dependency graph

$$\begin{tikzcd}
\boxed{00} \arrow[d] & \boxed{01} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{02} \arrow[d] \arrow[dr] & \boxed{04} \arrow[d] & \boxed{05} \arrow[ddl, bend left] \\
\boxed{03} \arrow[d] & \boxed{06} \arrow[d] & \\
\boxed{08} \arrow[d] \arrow[dr] & \boxed{07} \arrow[d] & \\
\boxed{09} & \boxed{10} &
\end{tikzcd}$$

Diminishing returns of superscalar execution

Most available ILP is exploited by a processor capable of issuing four instructions per clock
(Little performance benefit from building a processor that can issue more)

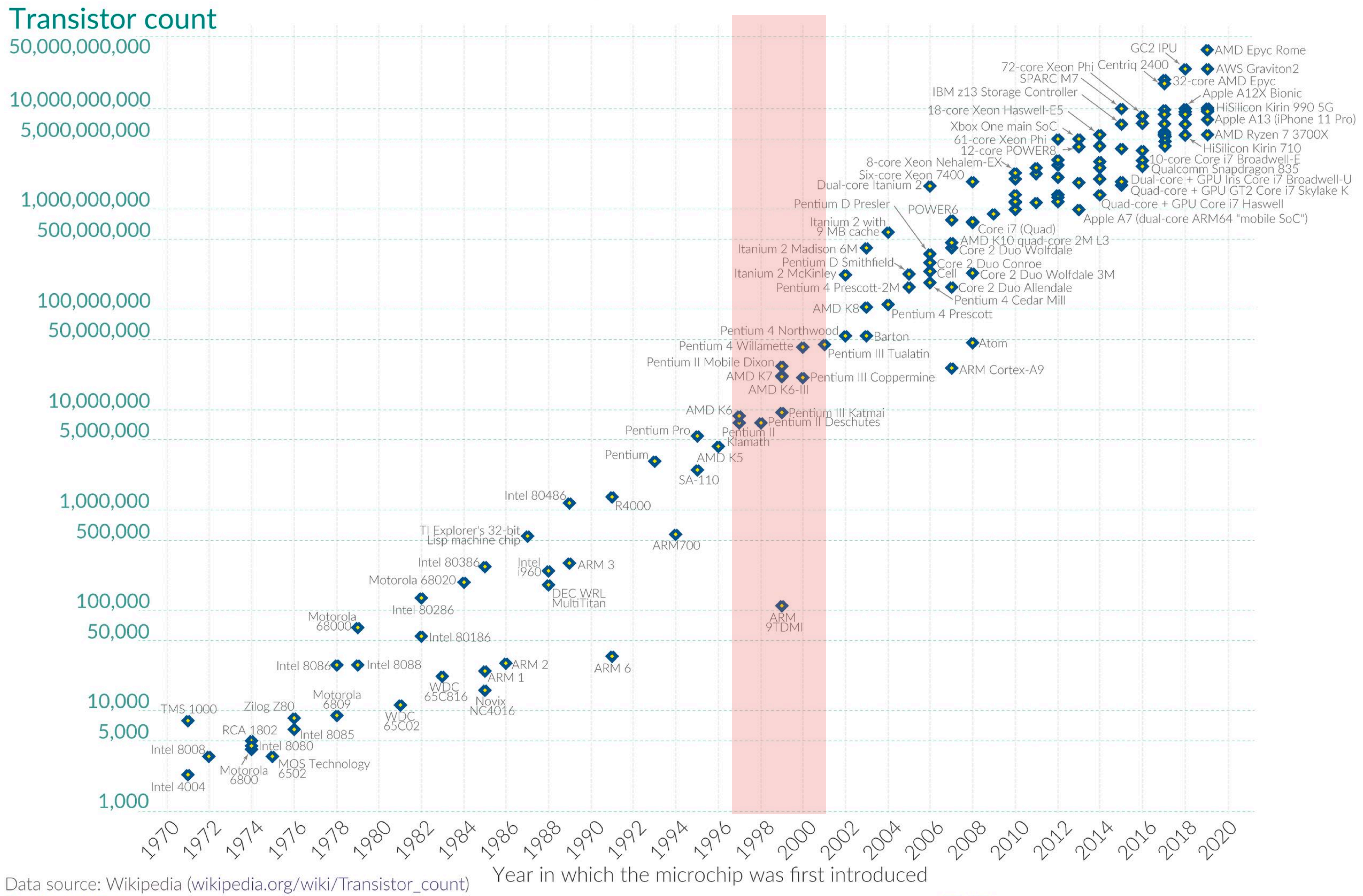


Source: Culler & Singh (data from Johnson 1991)

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

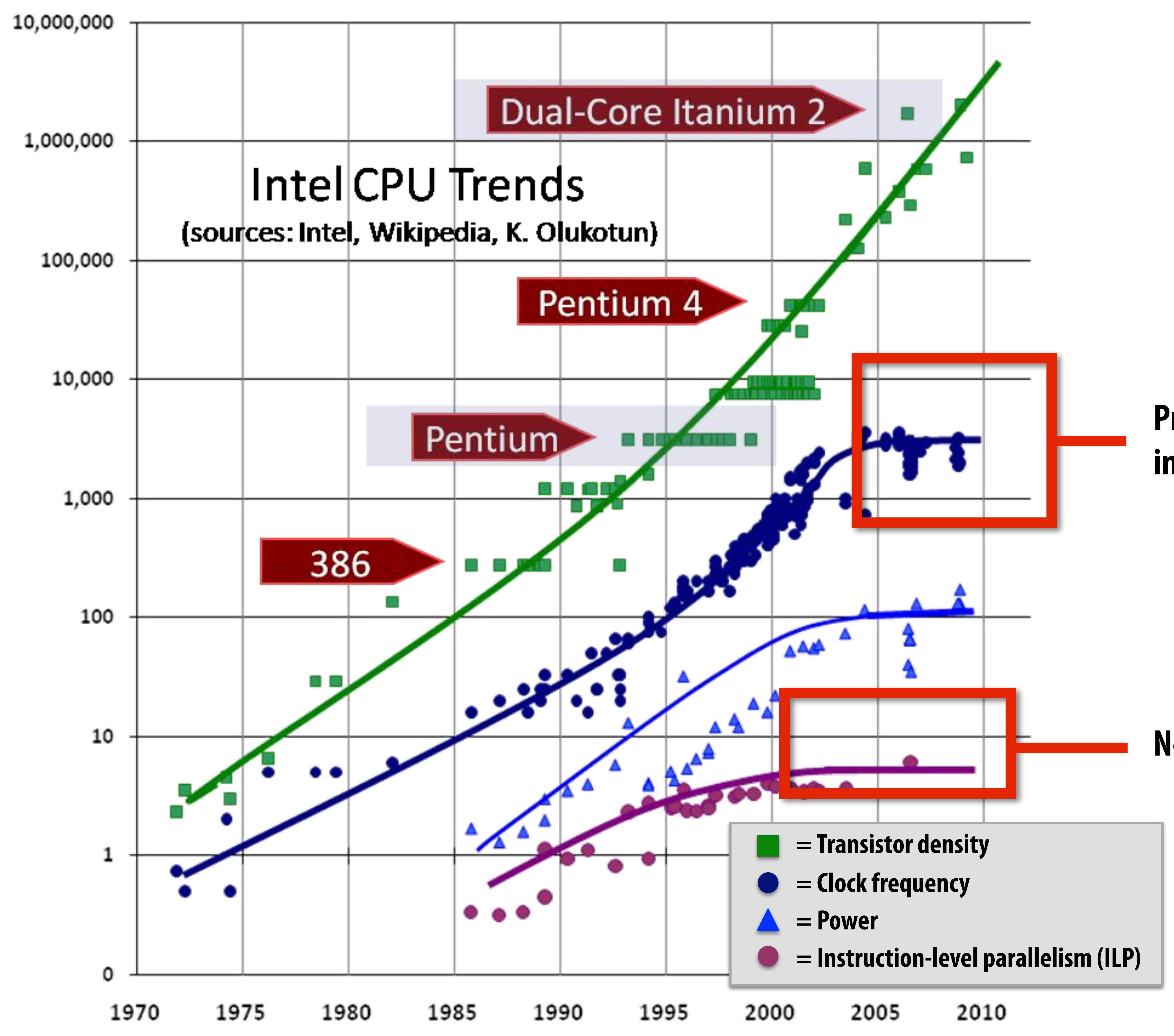
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

ILP tapped out + end of frequency scaling



Processor clock rate stops increasing

No further benefit from ILP

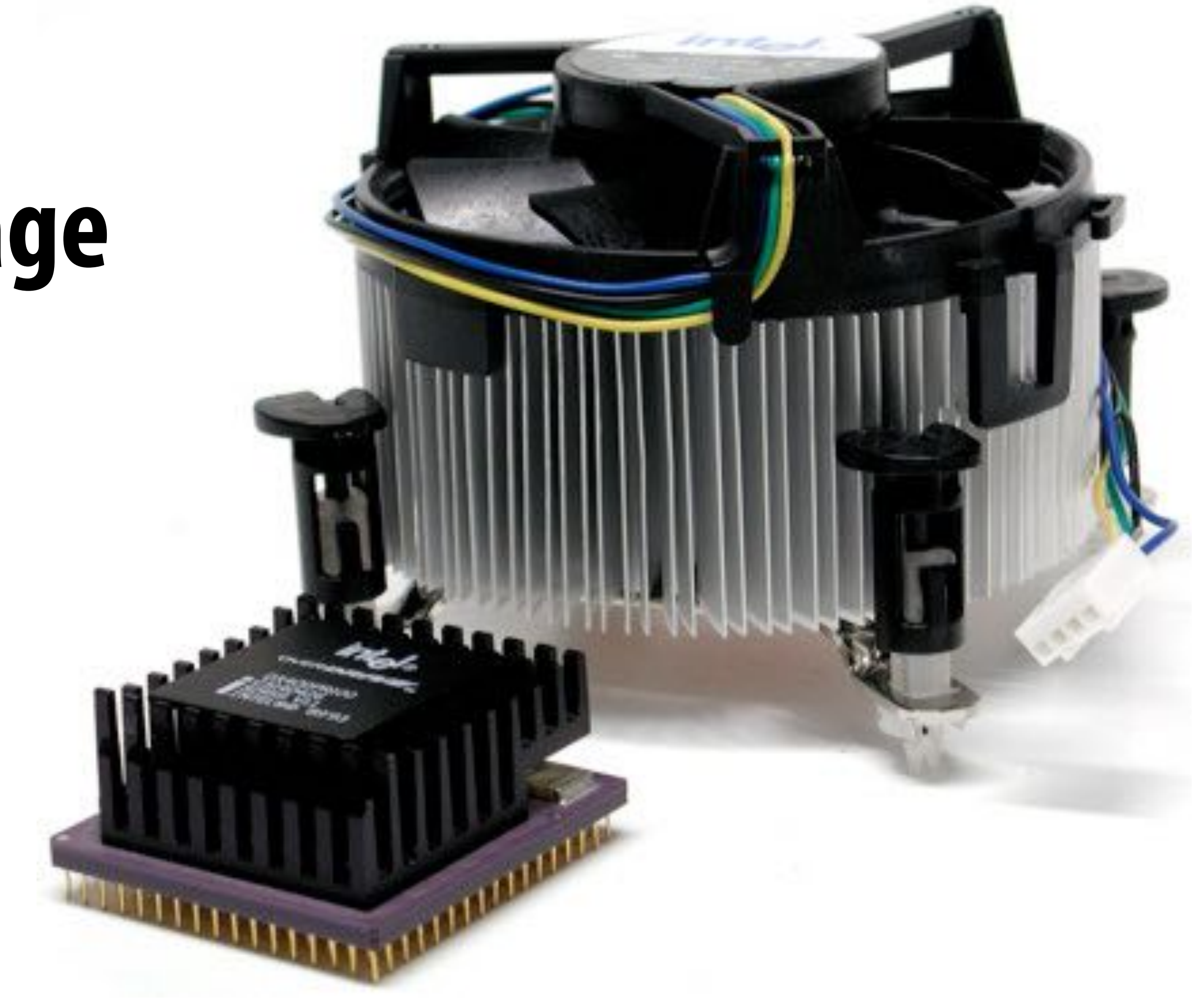
Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

The “power wall”

Power consumed by a transistor:

Dynamic power $\propto$ capacitive load $\times$ voltage^2 $\times$ frequency

Static power: transistors burn power even when inactive due to leakage



High power = high heat

Power is a critical design constraint in modern processors

	TDP
Apple M1 laptop:	13W
Intel Core i9 10900K (in desktop CPU):	95W
NVIDIA RTX 4090 GPU	450W
Mobile phone processor	$1/2$ - 2W
World’s fastest supercomputer	megawatts
Standard microwave oven	900W

Source: Intel, NVIDIA, Wikipedia, Top500.org

Power draw as a function of clock frequency

Dynamic power $\propto$ capacitive load $\times$ voltage^2 $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

Maximum allowed frequency determined by processor's core voltage

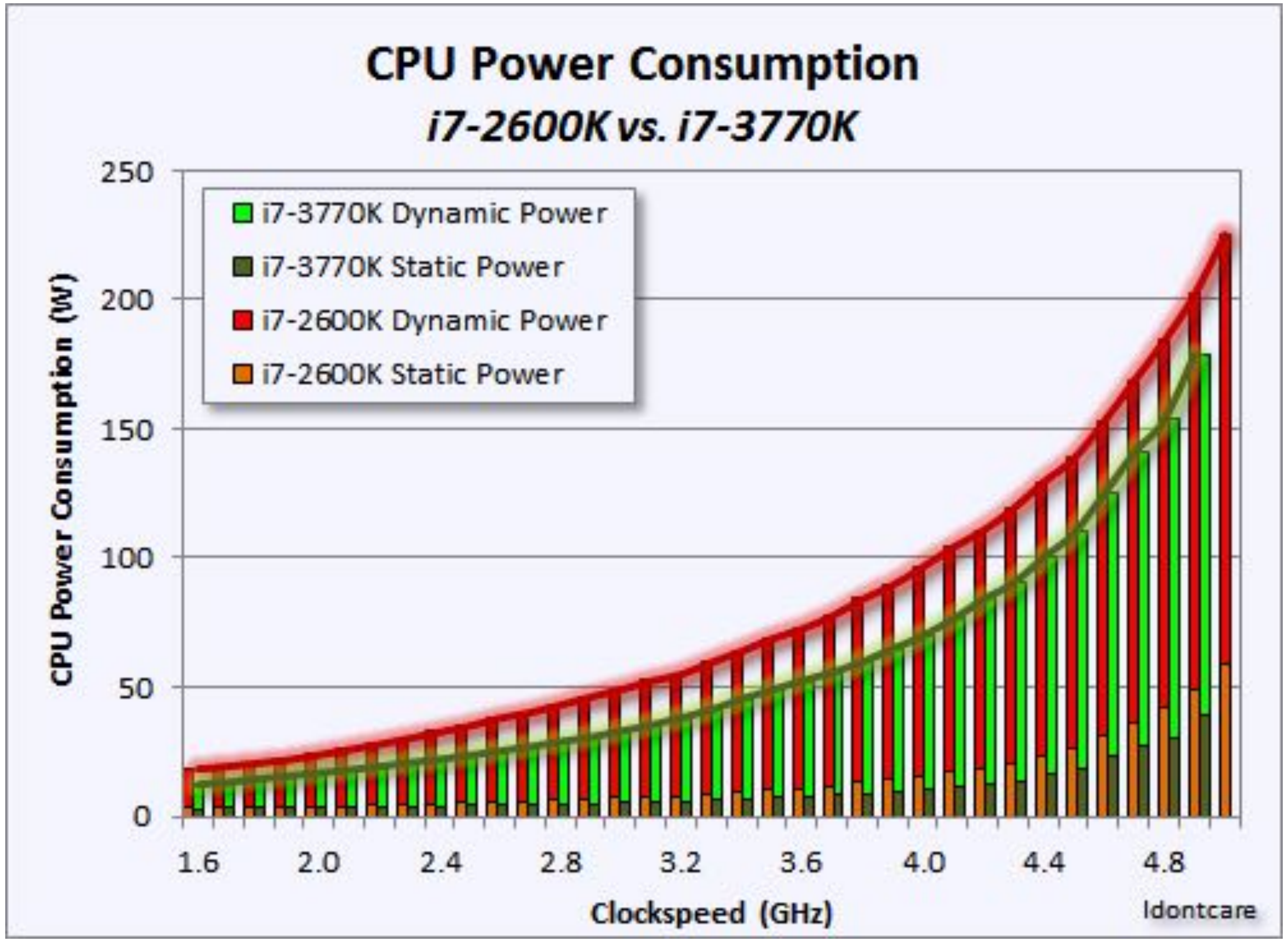


Image credit: "Idontcare": posted at: http://forums.anandtech.com/showthread.php?t=2281195

Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. Frequency scaling limited by power

2. ILP scaling tapped out

Architects are now building faster processors by adding more execution units that run in parallel

(Or units that are specialized for a specific task: like graphics, or audio/video playback)

Software must be written to be parallel to see performance gains. No more free lunch for software developers!

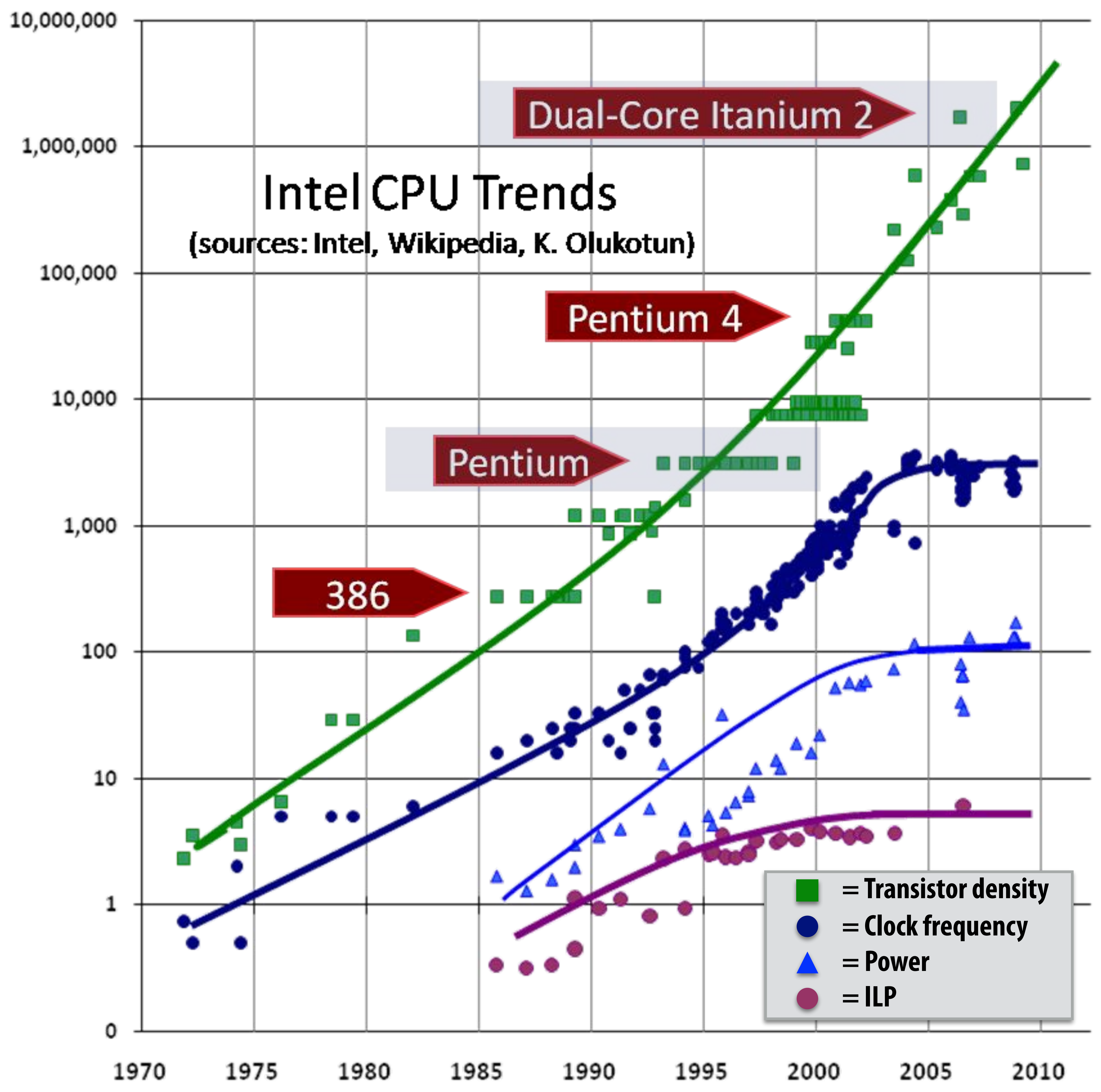
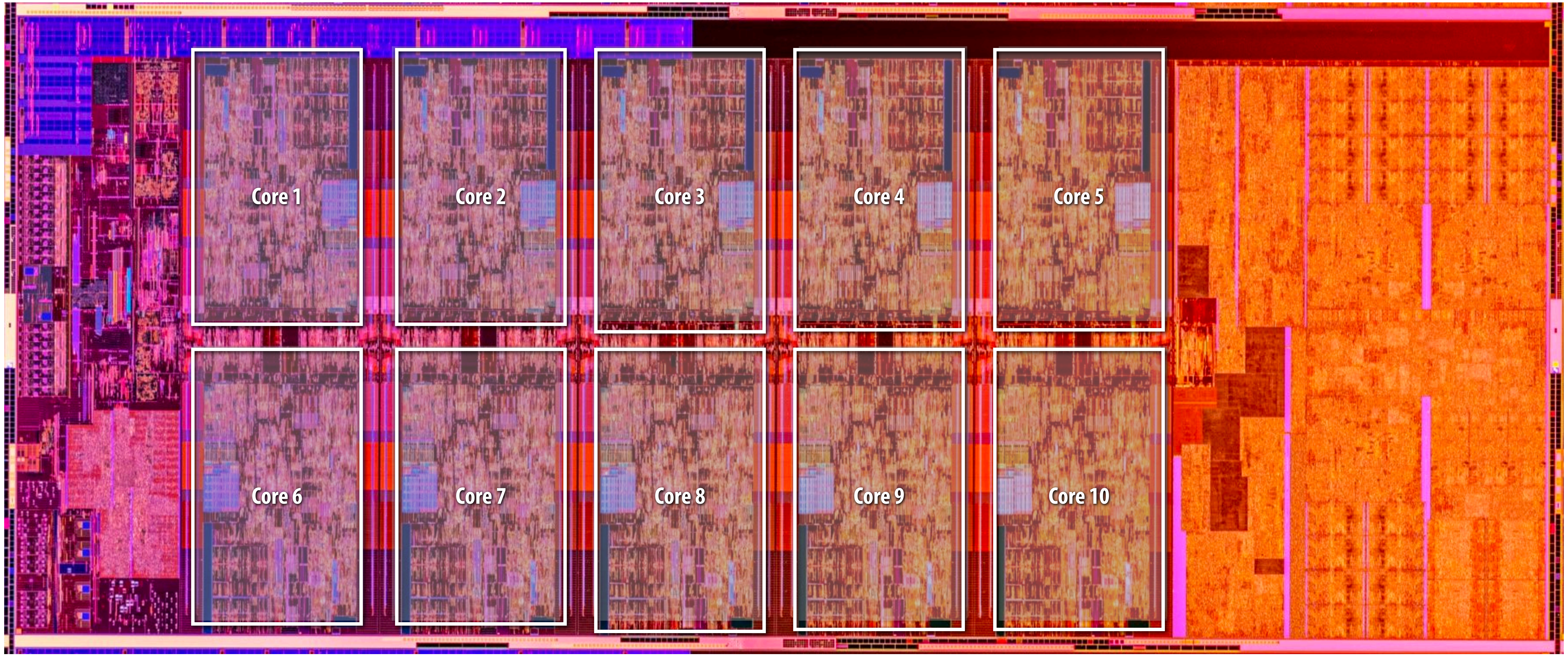


Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005

Example: multi-core CPU

Intel "Comet Lake" 10th Generation Core i9 10-core CPU (2020)



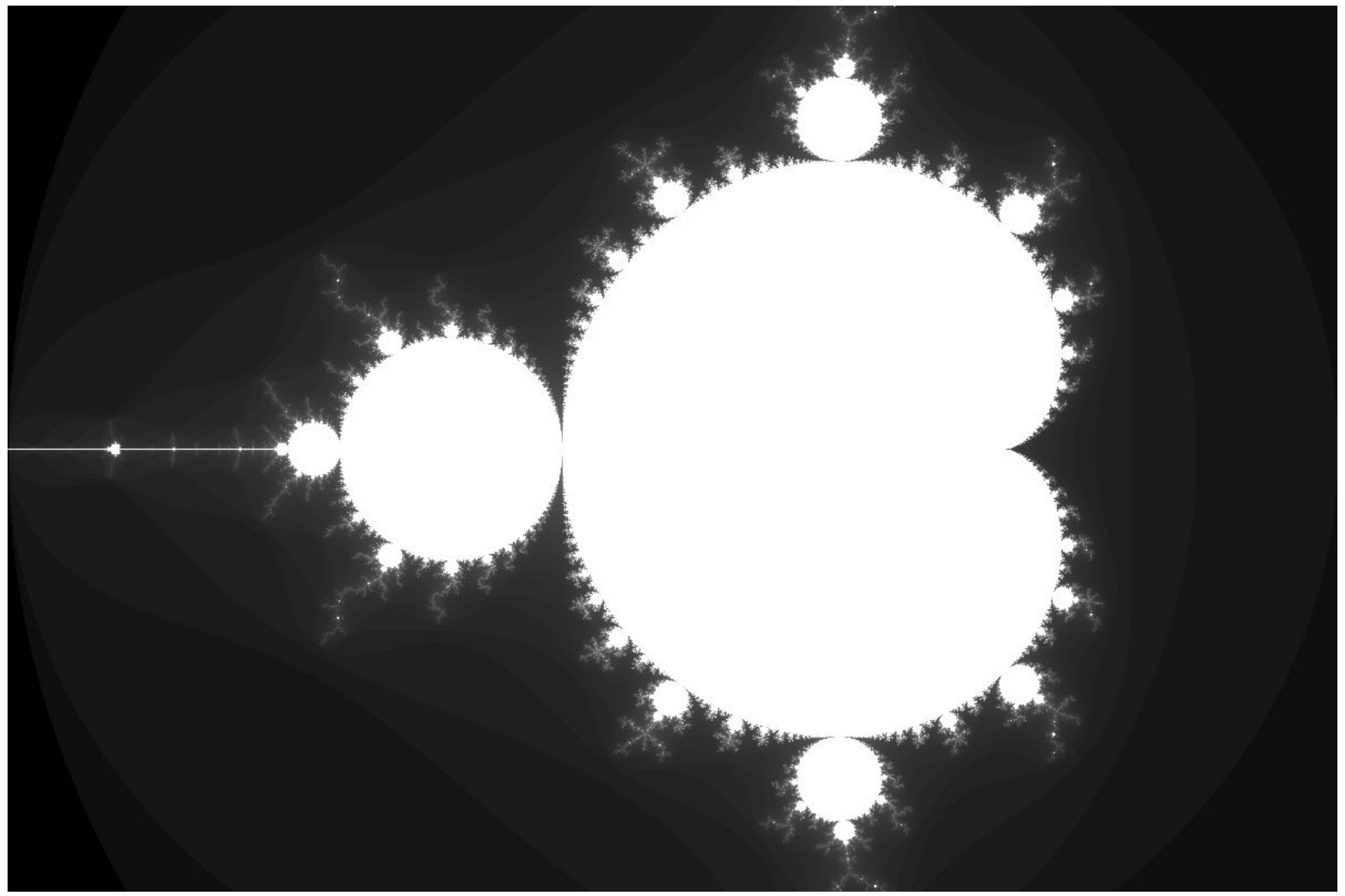
One thing you will learn in this course

- How to write code that efficiently uses the resources in a modern multi-core CPU

- Example: assignment 1 (coming up!)
  - Running on a quad-core Intel CPU
    - Four CPU cores
    - AVX SIMD vector instructions + hyper-threading
  - Baseline: single-threaded C program compiled with -O3
  - Parallelized program that uses all parallel execution resources on this CPU…

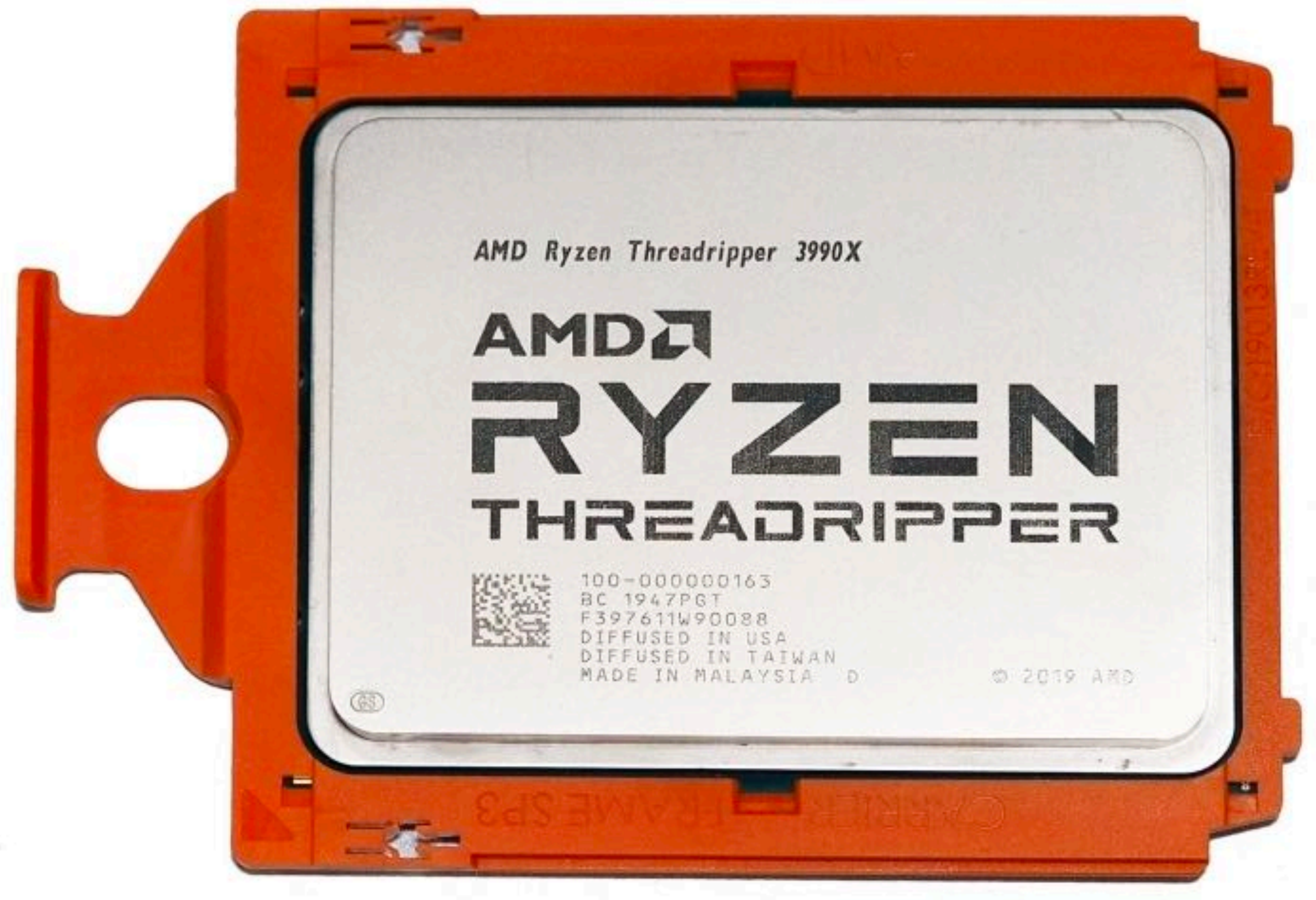
We'll talk about these terms next time!

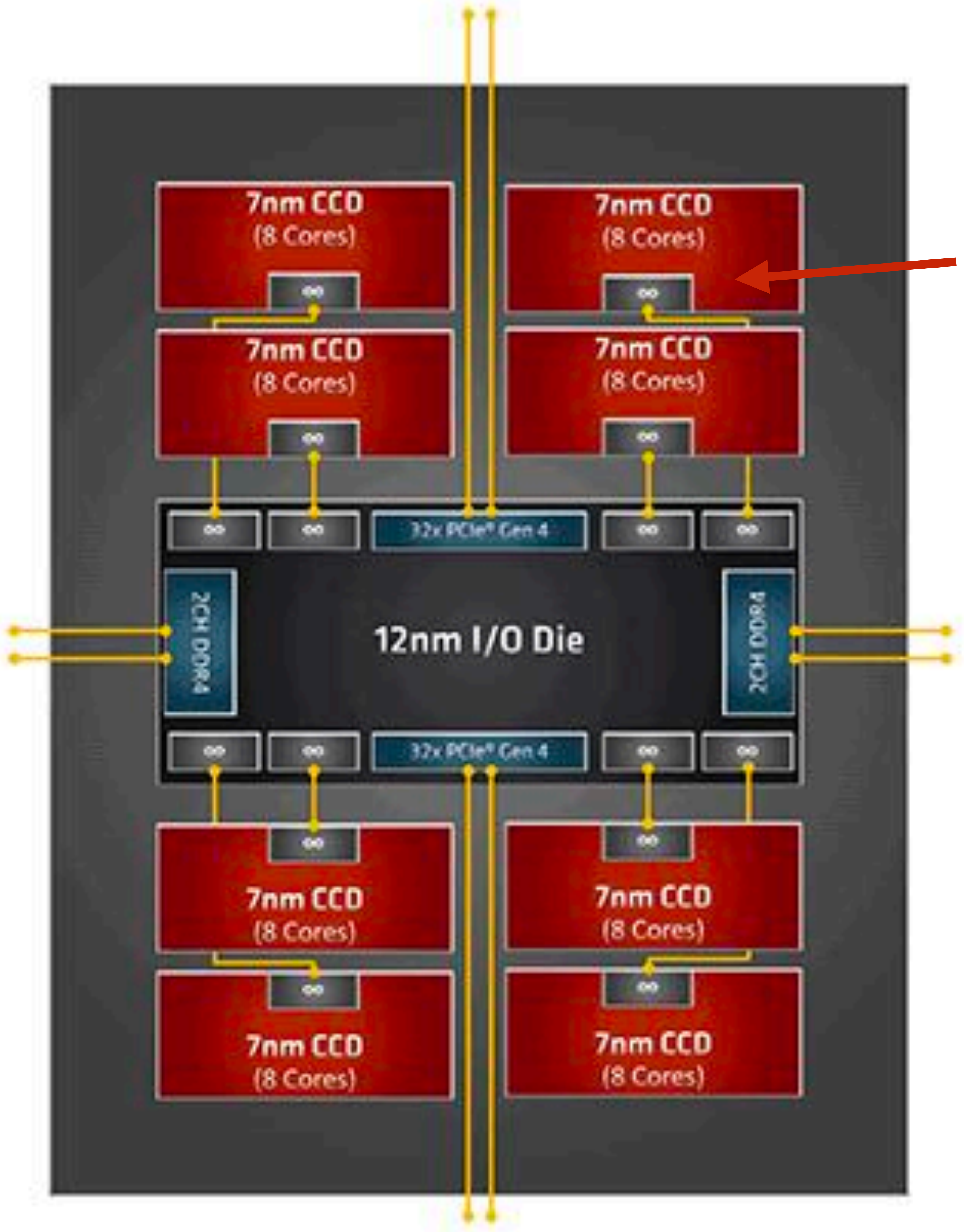
~32-40x faster!



AMD Ryzen Threadripper 3990X

64 cores, 4.3 GHz





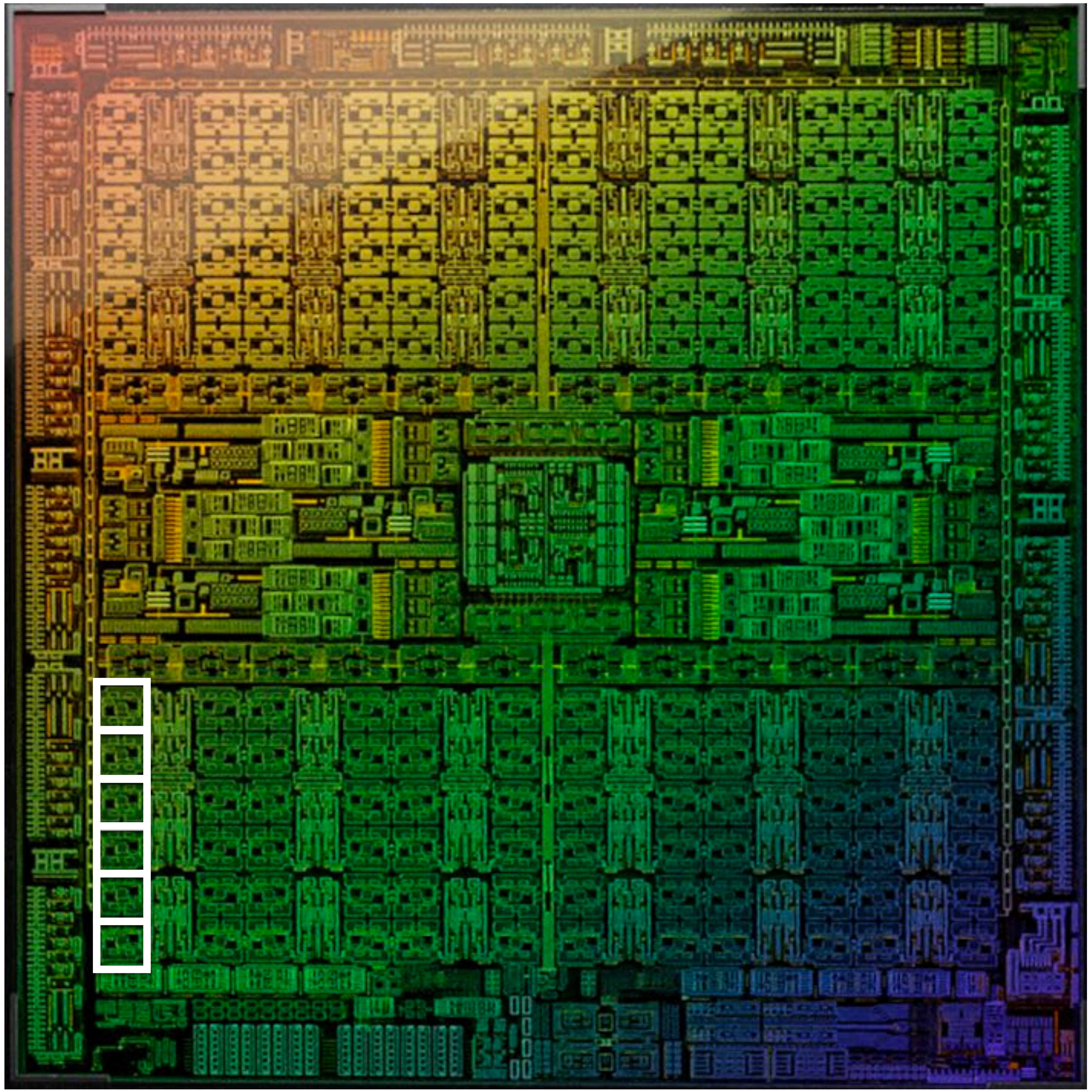
Four 8-core chiplets

NVIDIA AD102 GPU

GeForce RTX 4090 (2022)

76 billion transistors

18,432 fp32 multipliers organized in 144 processing blocks (called SMs)



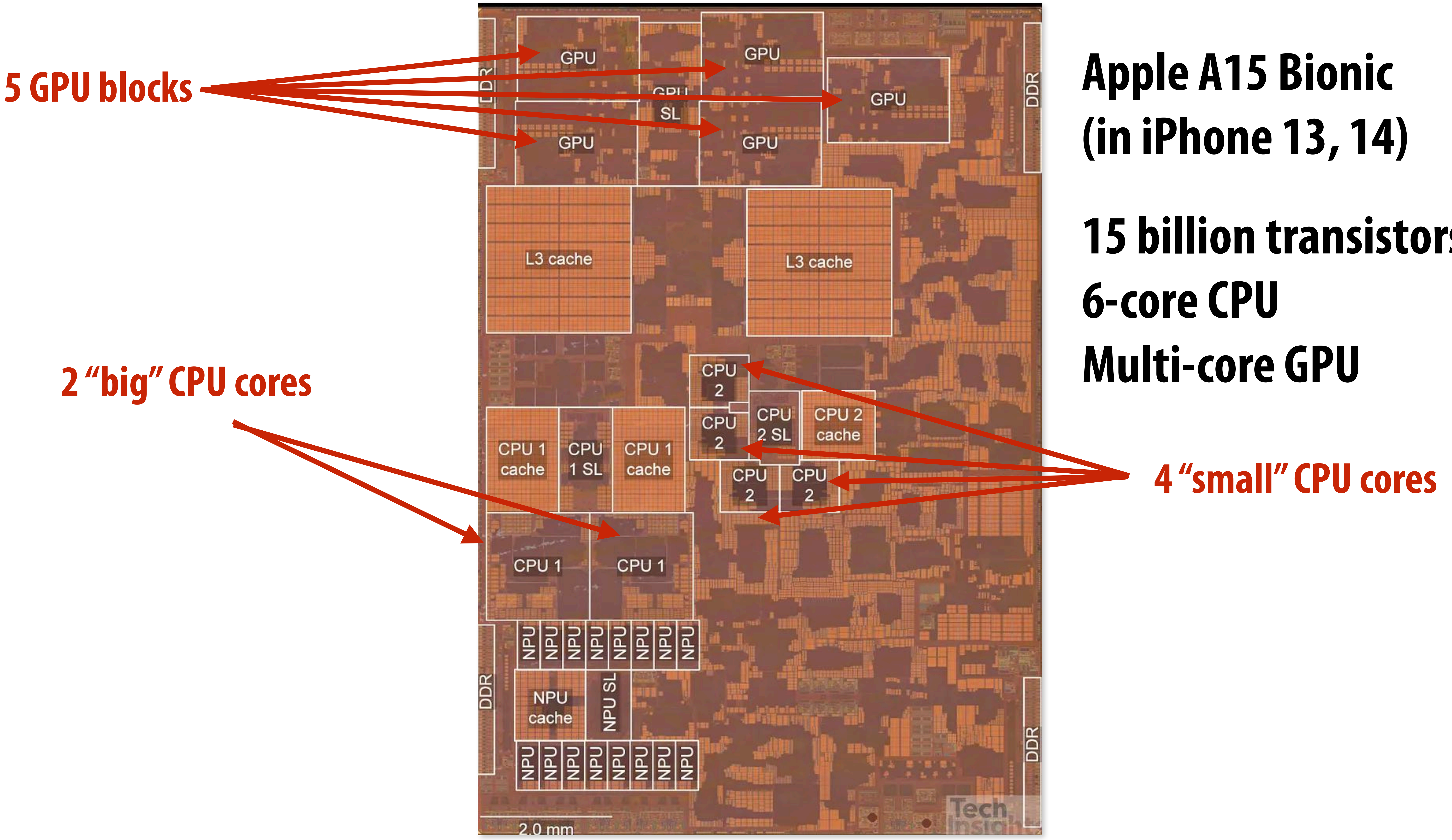
GPU-accelerated supercomputing

CRAY
a Hewlett Packard Enterprise company
HPE Cray
OAK RIDGE National Laboratory
U.S. DEPARTMENT OF ENERGY
Hewlett Packard Enterprise
AMD

Frontier (at Oak Ridge National Lab)
(world's #1 in Fall 2022)
9472 x 64 core AMD CPUs (606,208 CPU cores)
37,888 Radeon GPUs
21 Megawatts

Mobile parallel processing

Power constraints also heavily influence the design of mobile systems

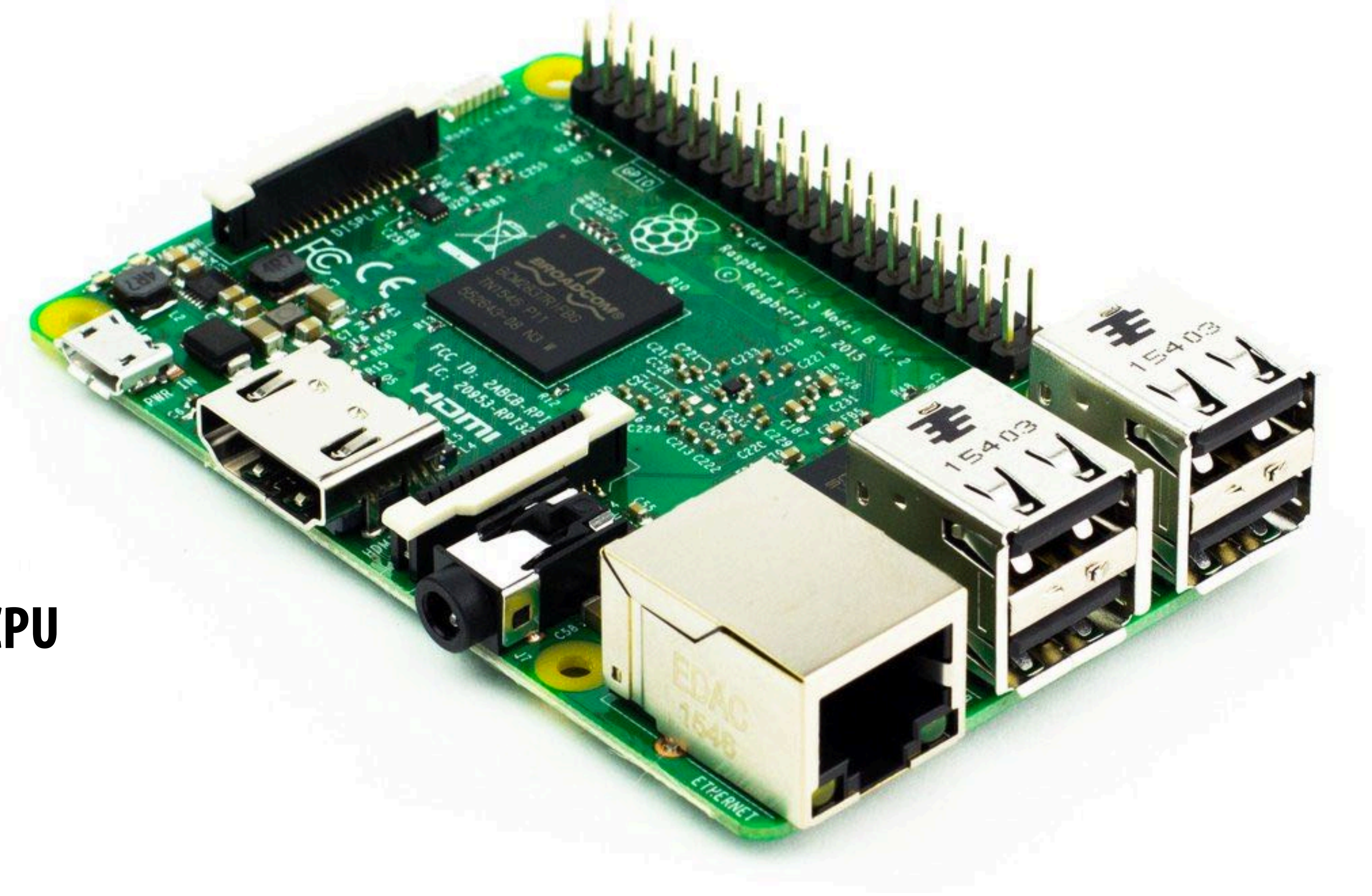


Apple A15 Bionic
(in iPhone 13, 14)

15 billion transistors
6-core CPU
Multi-core GPU

Image Credit: TechInsights Inc.

Mobile parallel processing



Raspberry Pi 3

Quad-core ARM A53 CPU

But in modern computing
software must be more than just parallel…

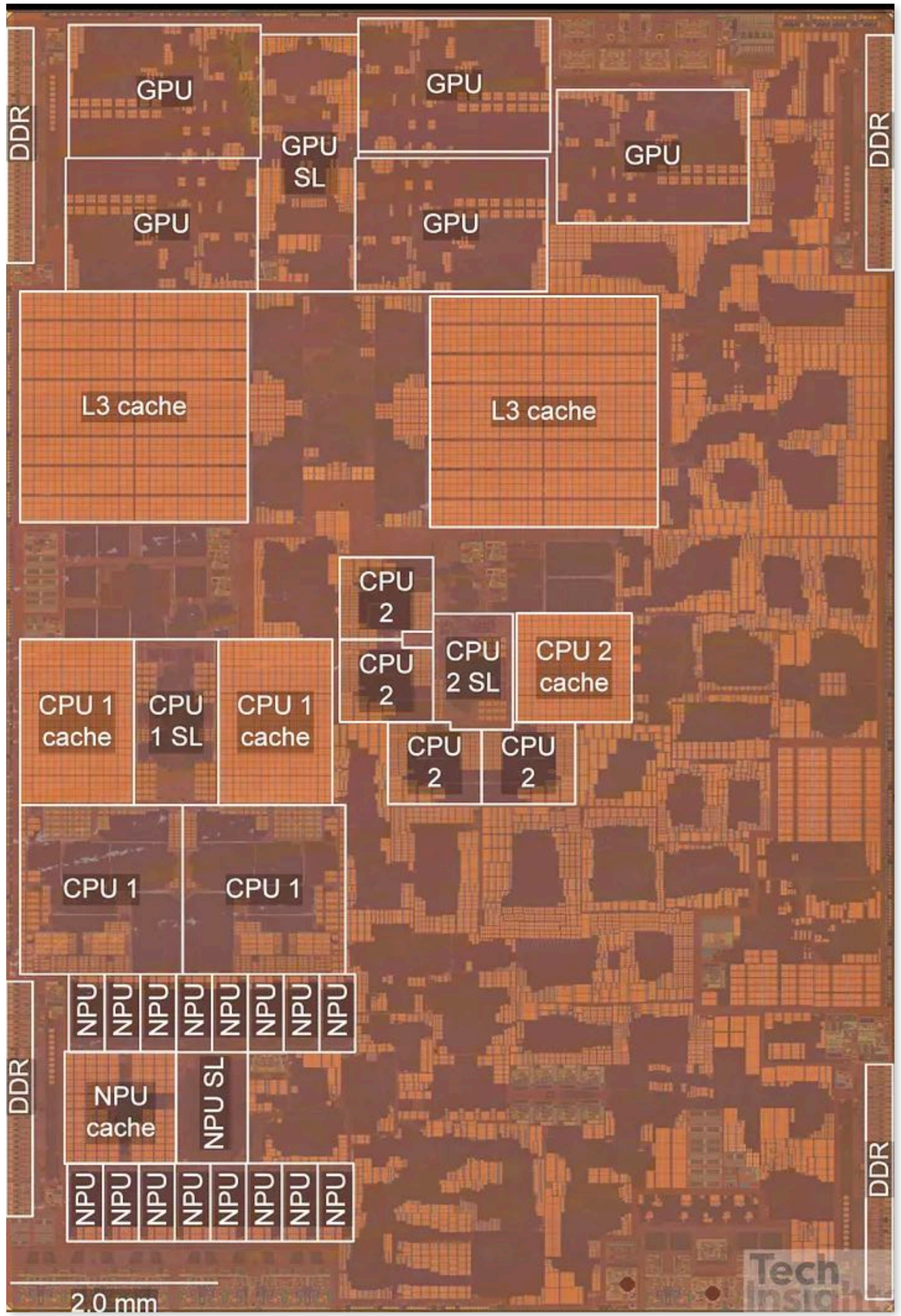
IT MUST ALSO BE EFFICIENT

Parallel + specialized HW

- **Achieving high efficiency will be a key theme in this class**

- **We will discuss how modern systems not only use many processing units, but also utilize specialized processing units to achieve high levels of power efficiency**

Specialized processing is ubiquitous in mobile systems



Apple A15 Bionic
(in iPhone 13, 14)

15 billion transistors

6-core GPU
- **2 "big" CPU cores**
- **4 "small" CPU cores**

Apple-designed multi-core GPU
Neural Engine (NPU) for DNN acceleration +
Image/video encode/decode processor +
Motion (sensor) processor

Image Credit: TechInsights Inc.

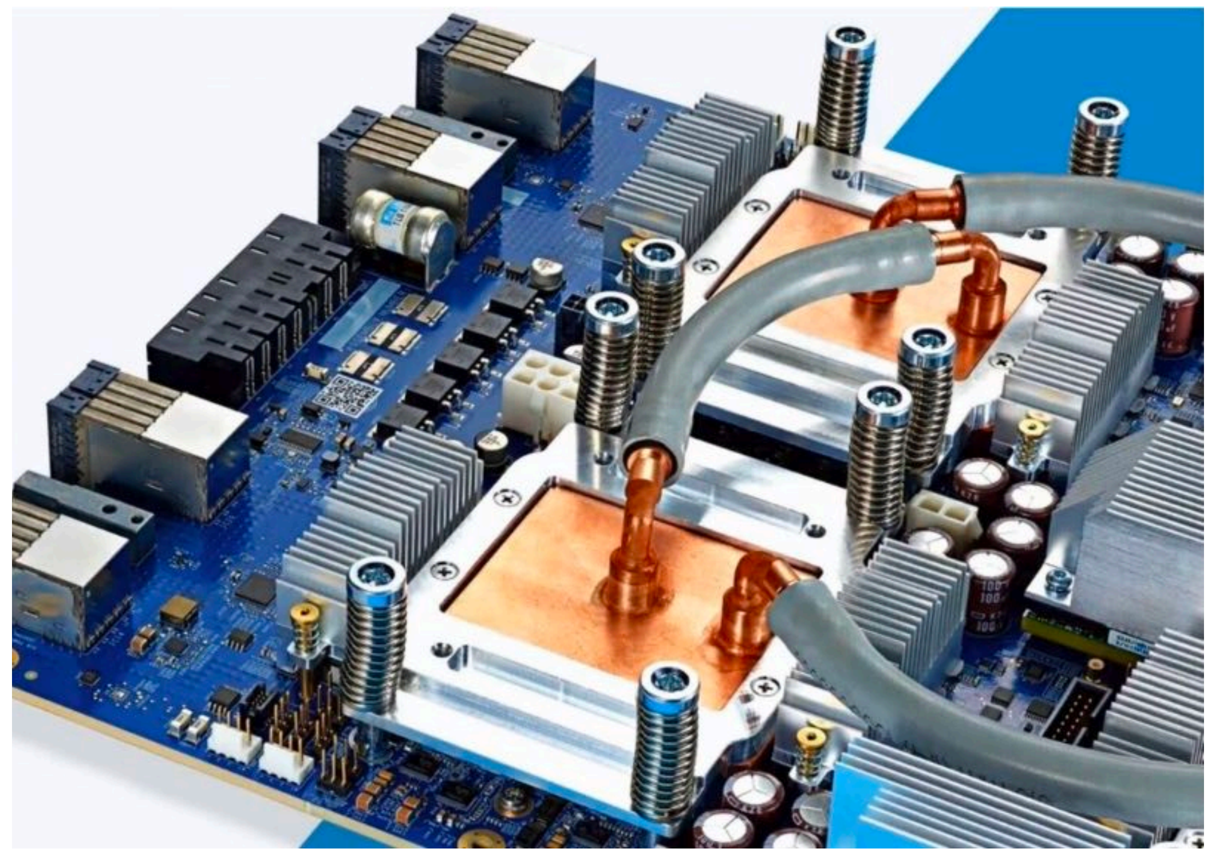
Specialization for datacenter-scale applications

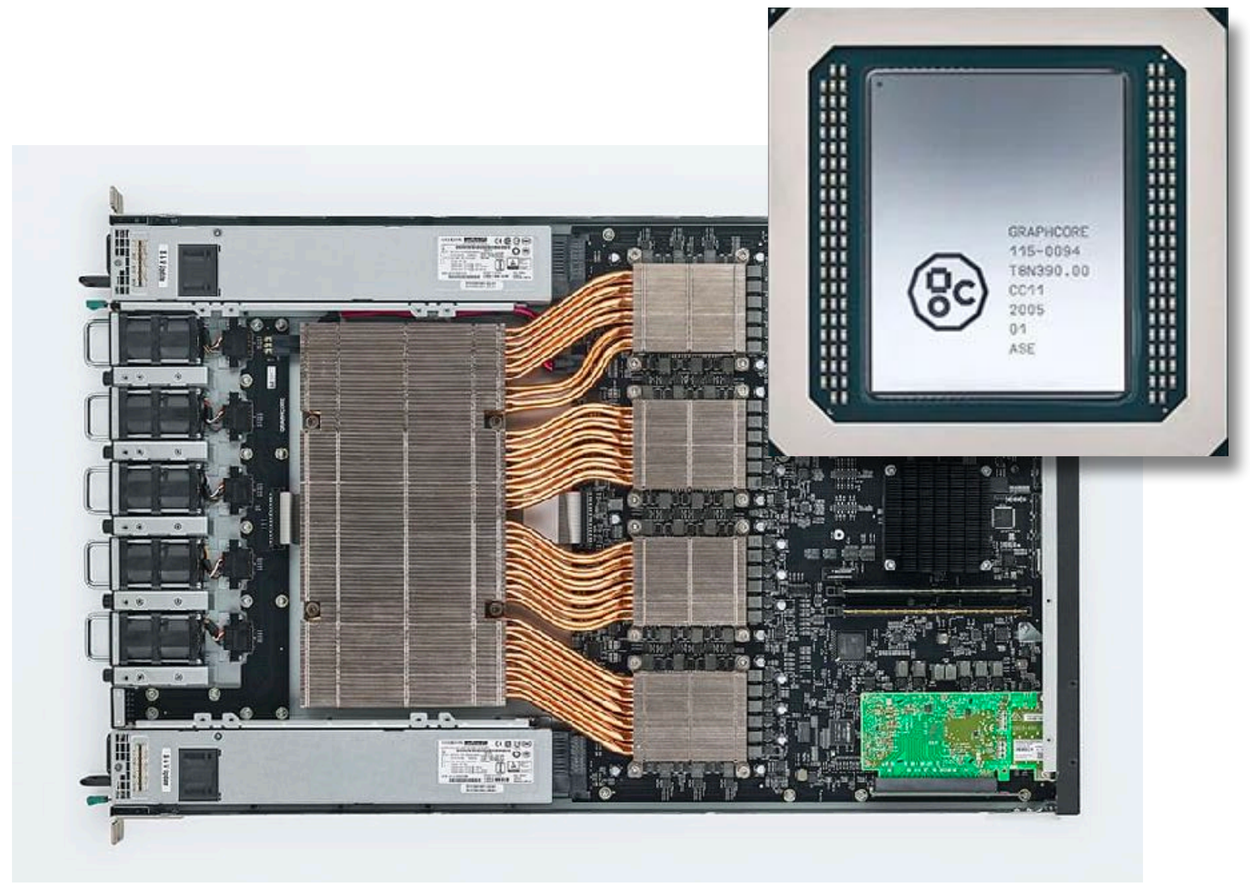
Google TPU pods

TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.

Specialized hardware to accelerate DNN inference/training

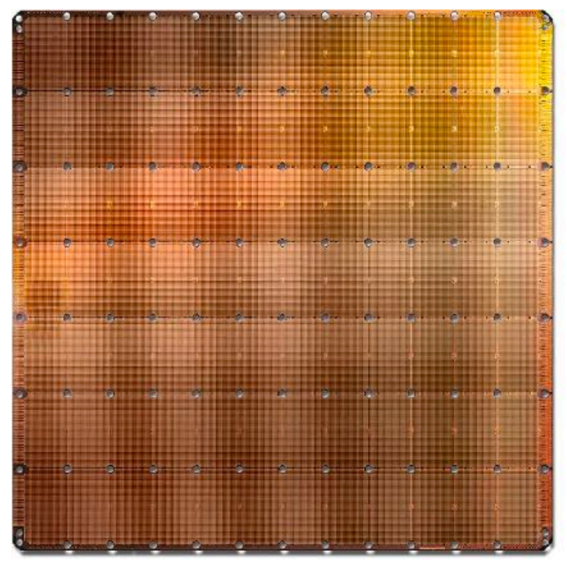

Google TPU3

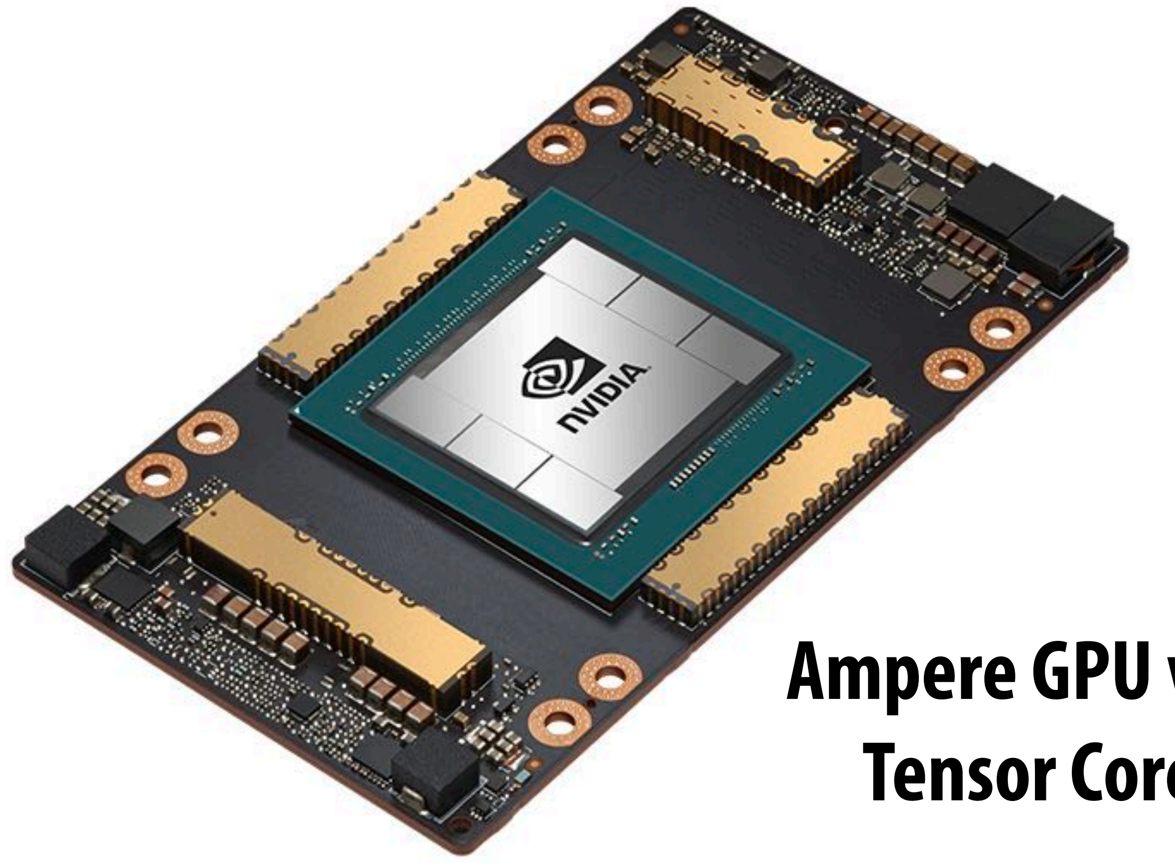

GraphCore IPU

Apple Neural Engine

Intel Deep Learning
Inference Accelerator

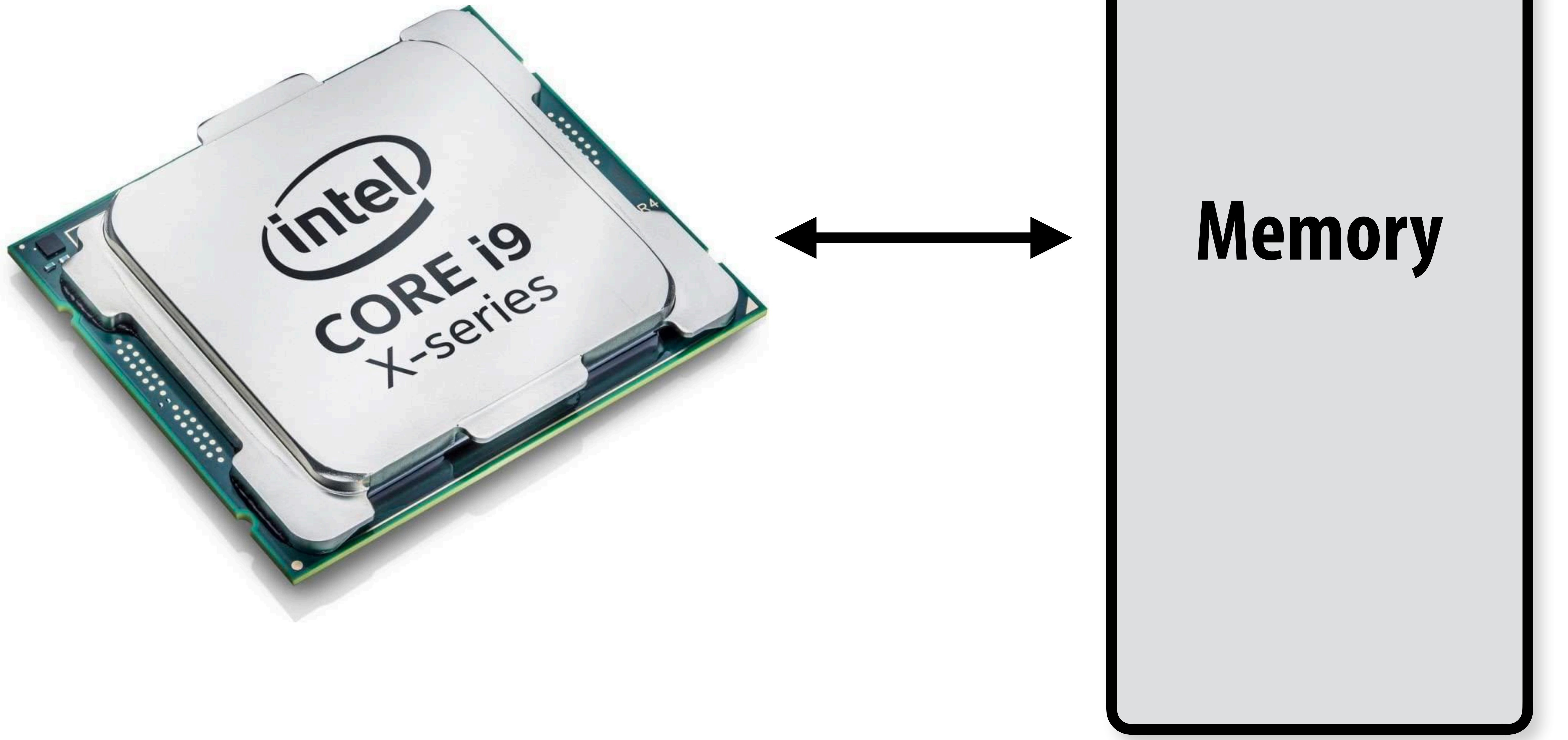
SambaNova
Cardinal SN10


Cerebras Wafer Scale Engine


Ampere GPU with
Tensor Cores

Achieving efficient processing almost always comes down to accessing data efficiently.

What is memory?



A program's memory address space

- A computer's memory is organized as an array of bytes

- Each byte is identified by its "address" in memory (its position in this array)

  (We'll assume memory is byte-addressable)

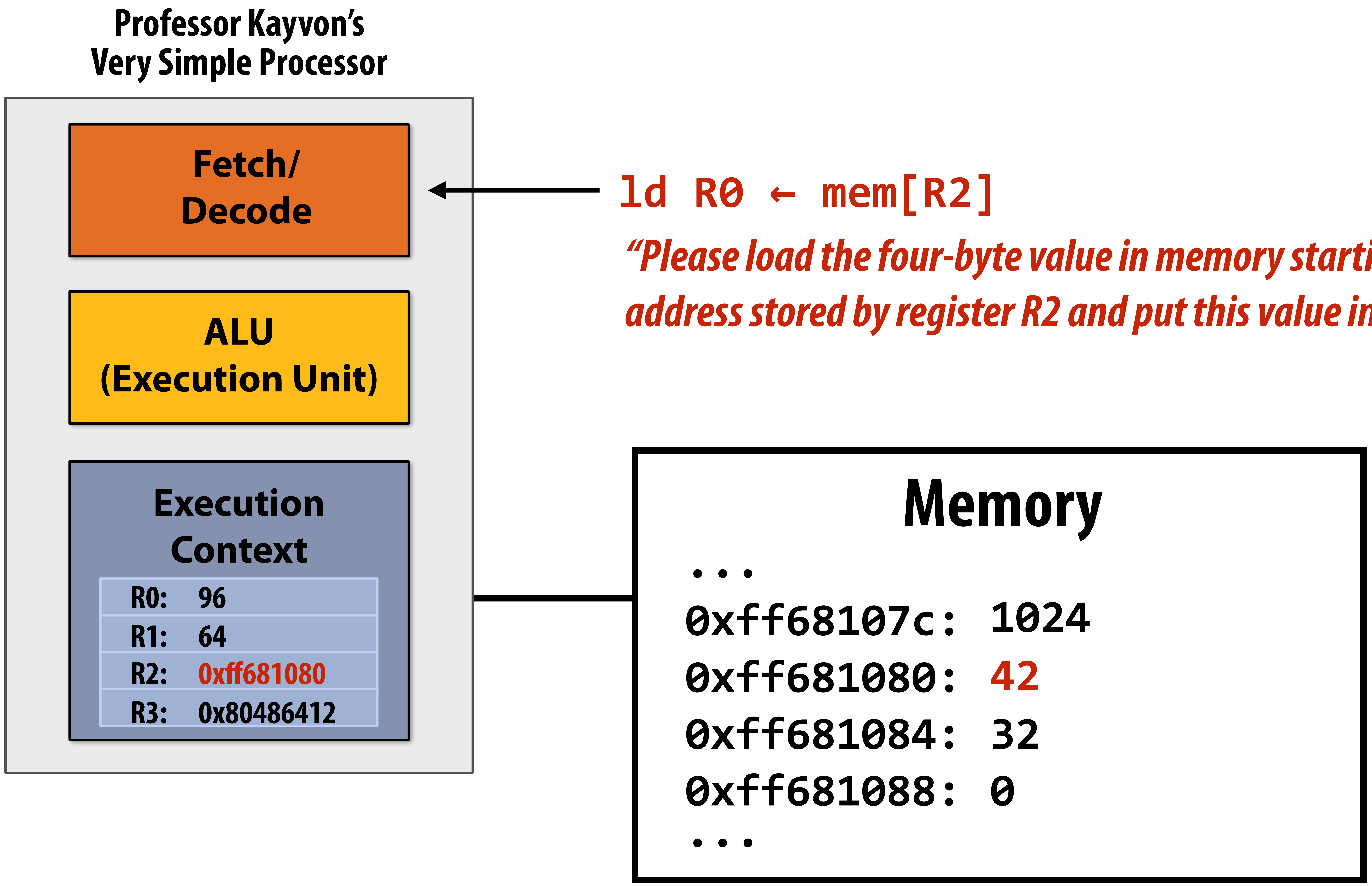
"The byte stored at address 0x8 has the value 32."

"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's memory address space is 32 bytes in size (so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Load: an instruction for accessing the contents of memory

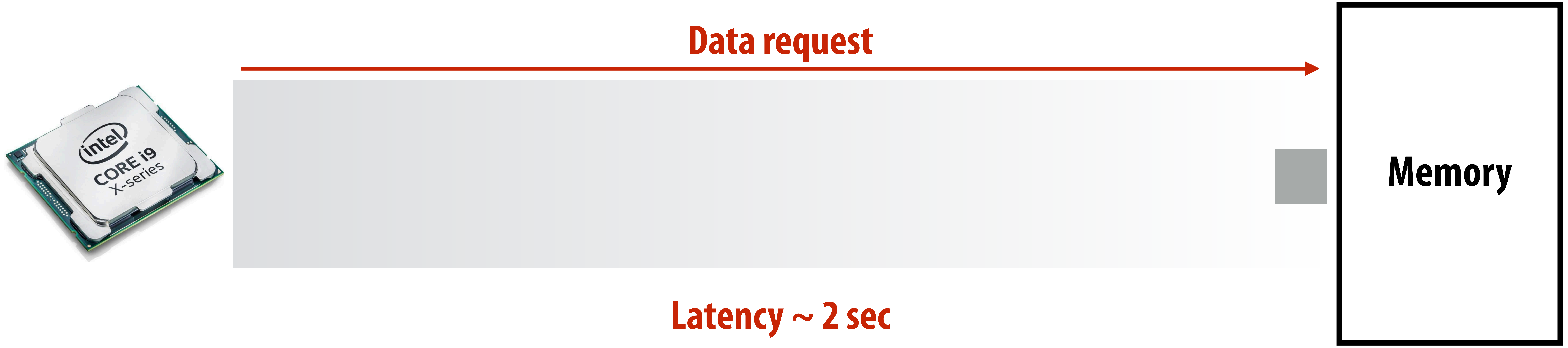


```
ld R0 ← mem[R2]
```

"Please load the four-byte value in memory starting from the address stored by register R2 and put this value into register R0."

Terminology

- **Memory access latency**
  - The amount of time it takes the memory system to provide data to the processor
  - Example: 100 clock cycles, 100 nsec



Stalls

- A processor "stalls" (can't make progress) when it cannot run the next instruction in an instruction stream because future instructions depend on a previous instruction that is not yet complete.

- Accessing memory is a major source of stalls

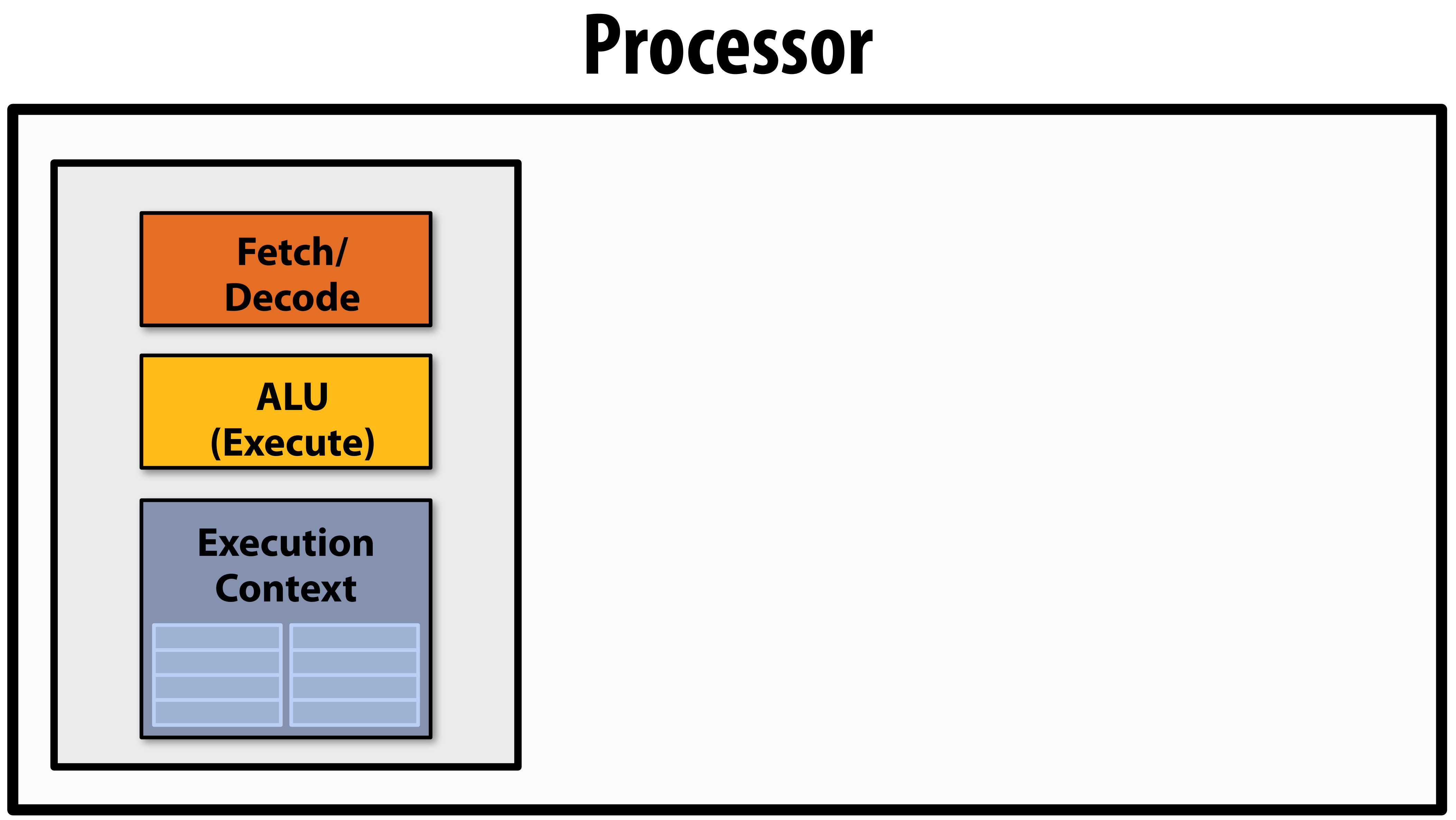
```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

Dependency: cannot execute 'add' instruction until data from mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100's of cycles
    - Memory "access time" is a measure of latency

What are caches?

- Recall memory is just an array of values
- And a processor has instructions for moving data from memory into registers (load) and storing data from registers into memory (store)



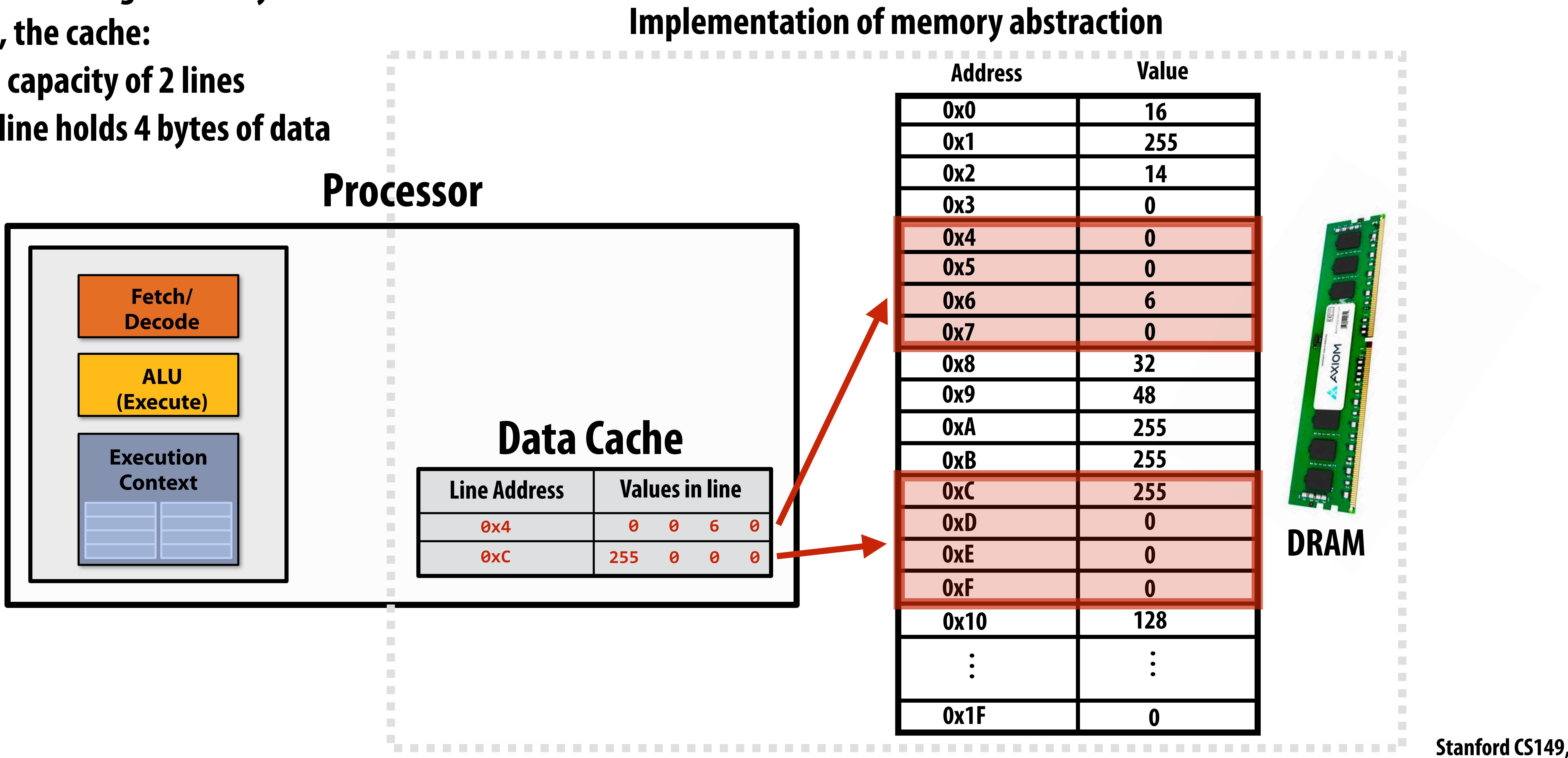
Memory

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

What are caches?

- A cache is a hardware implementation detail that does not impact the output of a program, only its performance
- Cache is on-chip storage that maintains a copy of a subset of the values in memory
- If an address is stored "in the cache" the processor can load/store to this address more quickly than if the data resides only in DRAM
- Caches operate at the granularity of "cache lines".

  In the figure, the cache:
  - Has a capacity of 2 lines
  - Each line holds 4 bytes of data



Data Cache

Line Address	Values in line
0x4	0 0 6 0
0xC	255 0 0 0

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Cache example 1

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU) replacement policy

time

Address accessed	Cache action	Cache state (after load is complete)	
0x0	"cold miss", load 0x0	0x0 ●●●●	
0x1	hit	0x0 ●●●●	
0x2	hit	0x0 ●●●●	
0x3	hit	0x0 ●●●●	
0x2	hit	0x0 ●●●●	
0x1	hit	0x0 ●●●●	
0x4	"cold miss", load 0x4	0x0 ●●●●	0x4 ●●●●
0x1	hit	0x0 ●●●●	0x4 ●●●●

There are two forms of "data locality" in this sequence:

Spatial locality: loading data in a cache line "preloads" the data needed for subsequent accesses to <u>different addresses</u> in the same line, leading to cache hits

Temporal locality: repeated accesses to the <u>same address</u> result in hits.

Cache example 2

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU) replacement policy

time

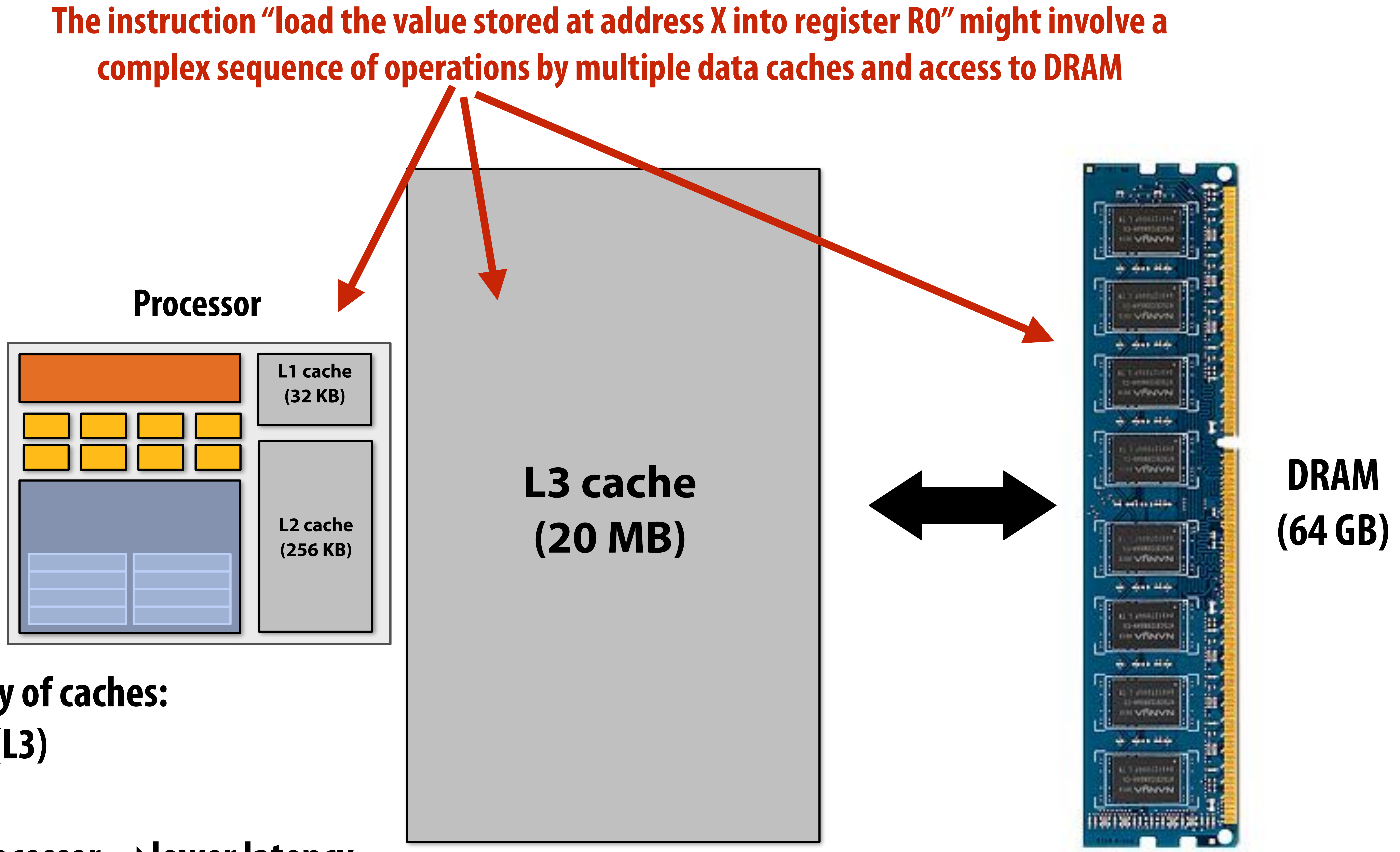
Address accessed	Cache action	Cache state (after load is complete)	
0x0	"cold miss", load 0x0	0x0 ●●●●	
0x1	hit	0x0 ●●●●	
0x2	hit	0x0 ●●●●	
0x3	hit	0x0 ●●●●	
0x4	"cold miss", load 0x4	0x0 ●●●●	0x4 ●●●●
0x5	hit	0x0 ●●●●	0x4 ●●●●
0x6	hit	0x0 ●●●●	0x4 ●●●●
0x7	hit	0x0 ●●●●	0x4 ●●●●
0x8	"cold miss", load 0x8 (evict 0x0)	0x8 ●●●●	0x4 ●●●●
0x9	hit	0x8 ●●●●	0x4 ●●●●
0xA	hit	0x8 ●●●●	0x4 ●●●●
0xB	hit	0x8 ●●●●	0x4 ●●●●
0xC	"cold miss", load 0xC (evict 0x4)	0x8 ●●●●	0xC ●●●●
0xD	hit	0x8 ●●●●	0xC ●●●●
0xE	hit	0x8 ●●●●	0xC ●●●●
0xF	hit	0x8 ●●●●	0xC ●●●●
0x0	"capacity miss", load 0x0 (evict 0x8)	0x0 ●●●●	0xC ●●●●

Caches reduce length of stalls (reduce memory access latency)

- Processors run efficiently when they access data that is resident in caches
- Caches reduce memory access latency when processors accesses data that they have recently accessed! *

* Caches also provide high bandwidth data transfer

The implementation of the linear memory address space abstraction on a modern computer is complex

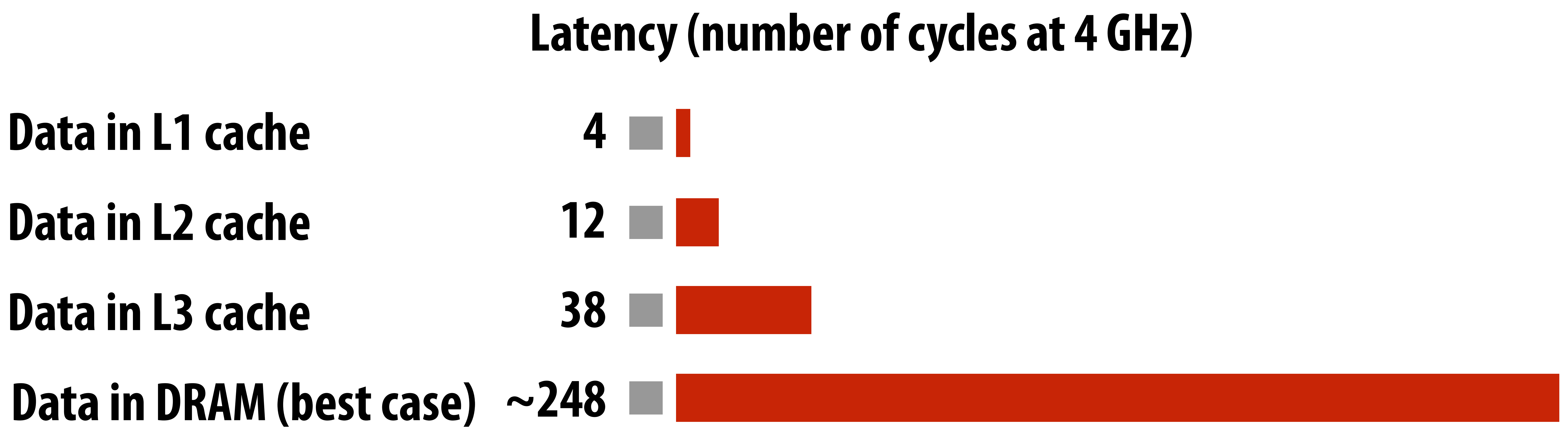

The instruction "load the value stored at address X into register R0" might involve a complex sequence of operations by multiple data caches and access to DRAM
Processor
L1 cache
(32 KB)
L2 cache
(256 KB)
L3 cache
(20 MB)
DRAM
(64 GB)

Common organization: hierarchy of caches:
Level 1 (L1), level 2 (L2), level 3 (L3)

Smaller capacity caches near processor → lower latency
Larger capacity caches farther away → larger latency

Data access times

(Kaby Lake CPU)



Summary

- Today, single-thread-of-control performance is improving very slowly
  - To run programs significantly faster, programs must utilize multiple processing elements or specialized processing hardware
  - Which means you need to know how to reason about and write parallel and efficient code

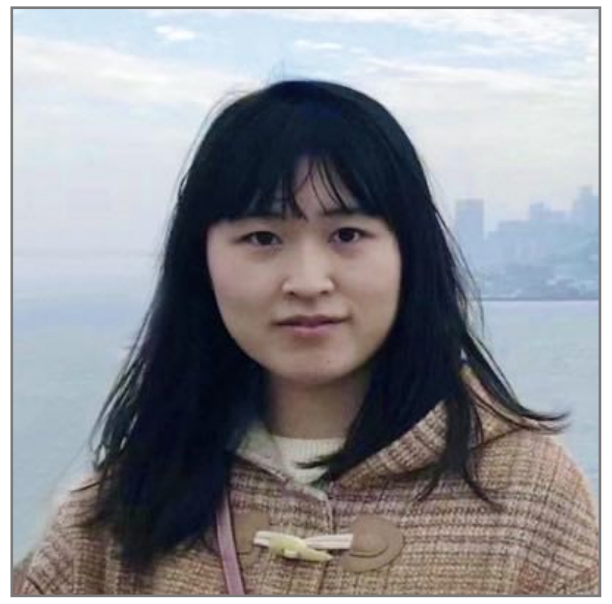
- Writing parallel programs can be challenging
  - Requires problem partitioning, communication, synchronization
  - Knowledge of machine characteristics is important
  - In particular, understanding data movement!

- I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it efficiently!

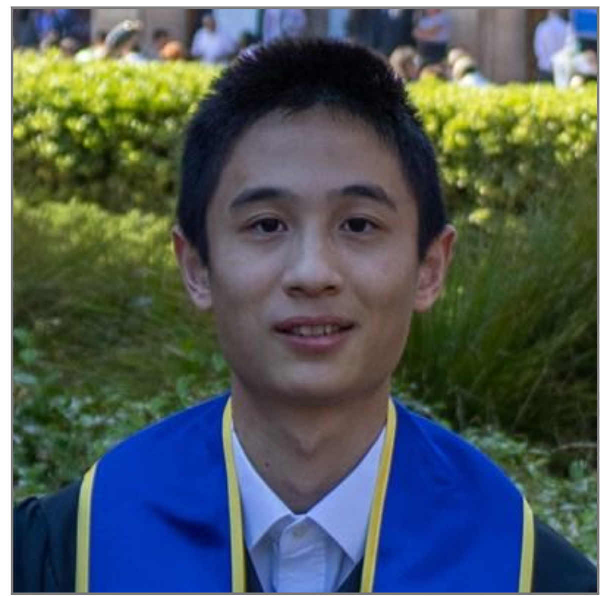
Welcome to CS149!

- Get signed up on the website
- Find yourself a partner!
  (remember, we can help you)

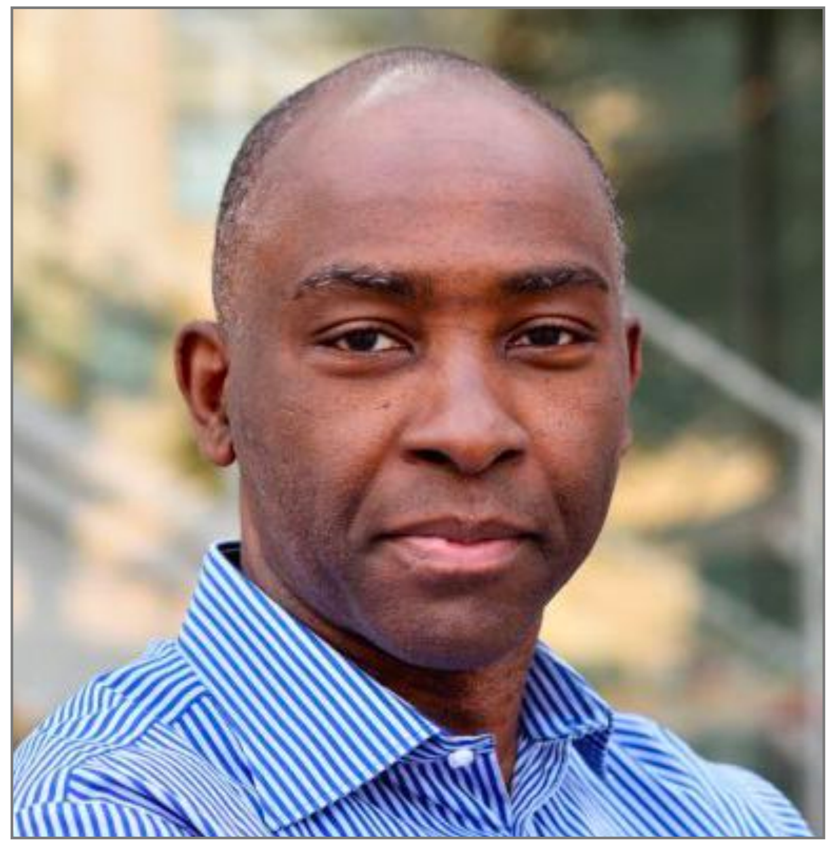
James


Minfei

Yasmine


Senyang

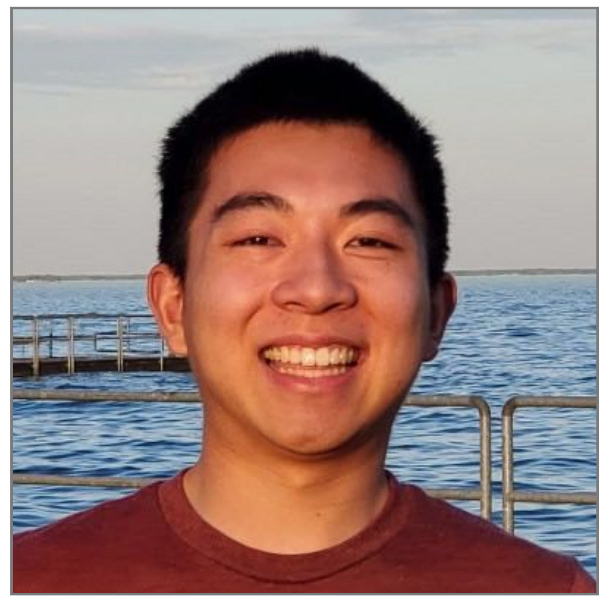
Prof. Kayvon


Prof. Olukotun

Zhenbang

Neha

Michael


Jensen

Shiv

Tom