

Lecture 1:

# Why Parallelism? Why Efficiency?

---

Parallel Computing  
Stanford CS149, Fall 2025

# Hello!



**Prof. Kayvon**



**Prof. Olukotun**

# 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  
processing elements to get it

# **DEMO 1**

**(our first parallel program)**

# Speedup

**One major motivation of using parallel processing: achieve a speedup**

**For a given problem:**

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

# 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, severely limiting speedup

# Course theme 1:

## Designing and writing parallel programs ... that scale!

- **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 2025

# PARALLEL COMPUTING

From smart phones, to multi-core CPUs, to GPUs, to AI accelerators, 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.

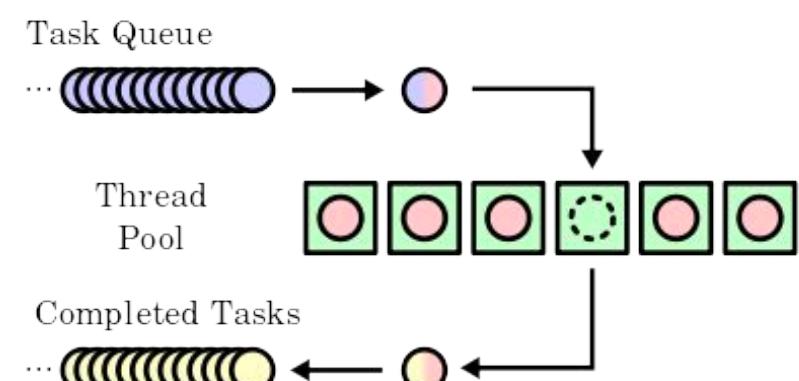
## Fall 2025 Schedule

Sep 23	<b>Why Parallelism? Why Efficiency?</b> Challenges of parallelizing code, motivations for parallel chips, processor basics
Sep 25	<b>A Modern Multi-Core Processor (Part I)</b> Forms of parallelism: multi-core, SIMD, and multi-threading
Sep 30	<b>Modern Multi-Core Architecture (Part II) + ISPC Programming Abstractions</b> Finish up multi-threaded and latency vs. bandwidth. ISPC programming, abstraction vs. implementation
Oct 02	<b>Parallelizing Code: An Example Thought Process</b> Process of parallelizing a program in data parallel and shared address space models
Oct 07	<b>Program Optimization 1: Work Distribution and Scheduling</b> Achieving good work distribution while minimizing overhead, scheduling Cilk programs with work stealing
Oct 09	<b>Program Optimization 2: Locality and Communication</b> Message passing, async vs. blocking sends/receives, pipelining, increasing arithmetic intensity, avoiding contention
Oct 14	<b>GPU Architecture and CUDA Programming</b> CUDA programming abstractions, and how they are implemented on modern GPUs
Oct 16	<b>Data-Parallel Thinking</b> Data-parallel operations like map, reduce, scan, prefix sum, groupByKey

# Five programming assignments



**Assignment 1: ISPC programming  
on multi-core CPUs**



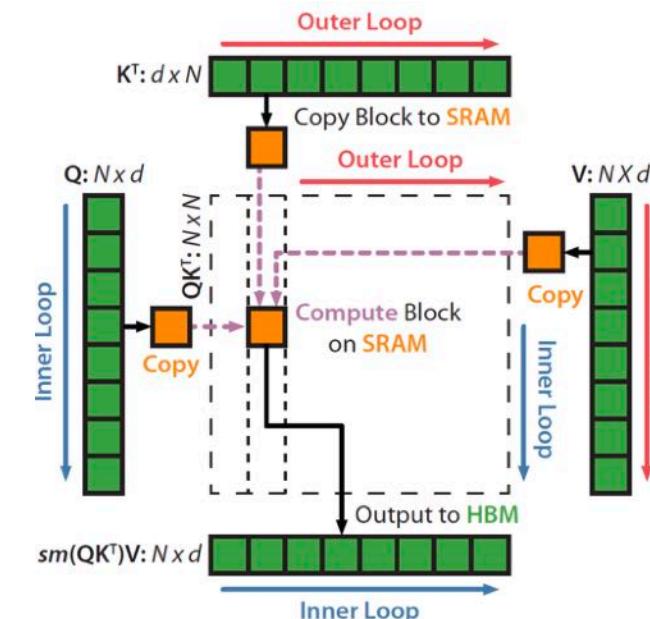
**Assignment 2:  
scheduling a task graph**



**Assignment 3: Writing a renderer  
in CUDA on NVIDIA GPUs**



**Assignment 4:  
Optimizing DNN Computations  
on an AI Accelerator**



**Assignment 5:  
Write the World's  
Fastest CUDA Kernels**

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

**We realize finding a partner can be  
stressful.** 😱 😥

**Fill out our partner request form by  
Friday noon and we will find you a  
partner!** 🎉 😎

# Late days

- You get **eight late days** for the quarter
  - For use on programming assignments 1-4 only (not programming assignment 5)
- 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/extracurricular events, academic conference travel...
  - We expect to give extra late days only under exceptional circumstances
- Requests for additional late days to accommodate foreseeable exceptional circumstances should be made 48 hours prior to the original assignment deadline.
  - We will deny requests if you could have reasonably planned ahead.

# Written assignments

- **There will be four written assignments this quarter**
  - **Some of the questions are graded on correctness, others are graded on effort**
- **Written assignments are done in teams of three**
  - **The course staff will assign you partners (different partners each assignment)**
  - **Your partners will assess your contributions to the team (and we'll pass along the feedback to you)**
- **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**

# Per-lecture participation

- **There is no attendance requirement, but there is a per-lecture attendance requirement**
  - The goal is to encourage all students to “keep up” with the class by watching and engaging with the lecture either during, or in the day following, the lecture
- **Each class, the instructor will release a “quiz question” that the students can respond to during lecture. It will be realized at a point in the lecture of the instructor’s choosing.**
  - Students in class can answer the question immediately for credit.
- **If you do not attend lecture, there will be a different (slightly longer) quiz for you to take, and it must be submitted within 36 hours of the lecture (11:50pm the following day)**

# Grades

**56% Programming assignments (5)**

**12% Written assignments (4)**

**4% Per-lecture participation**

- **(There are 17 lectures after today. You must participate in 15 for full credit)**

**12% Midterm exam**

- **Two-hour exam in the evening on Nov 18th**

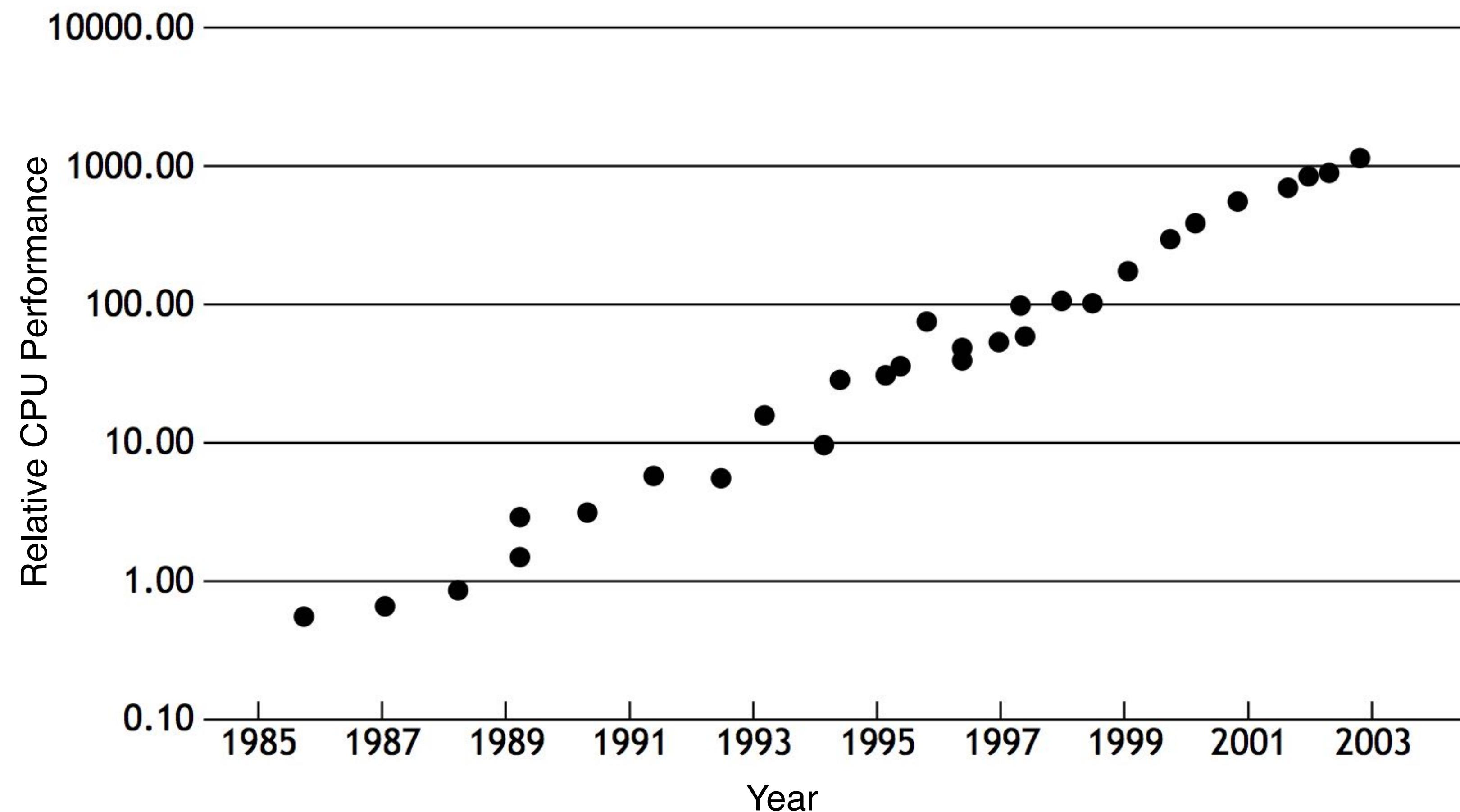
**16% Final exam**

- **During the university-assigned slot: Dec 11th, 3:30pm**

# 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!



# Until ~20 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

```
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!

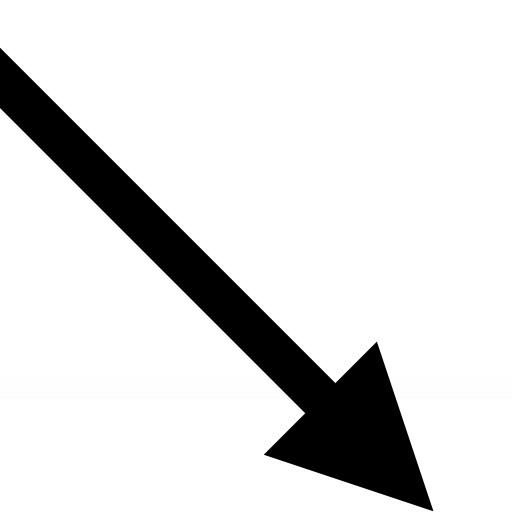
```
int main(int argc, char** argv) {  
  
    int x = 1;  
  
    for (int i=0; i<10; i++) {  
        x = x + x;  
    }  
  
    printf("%d\n", x);  
  
    return 0;  
}
```

→ **Compile code** →

```
_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
```

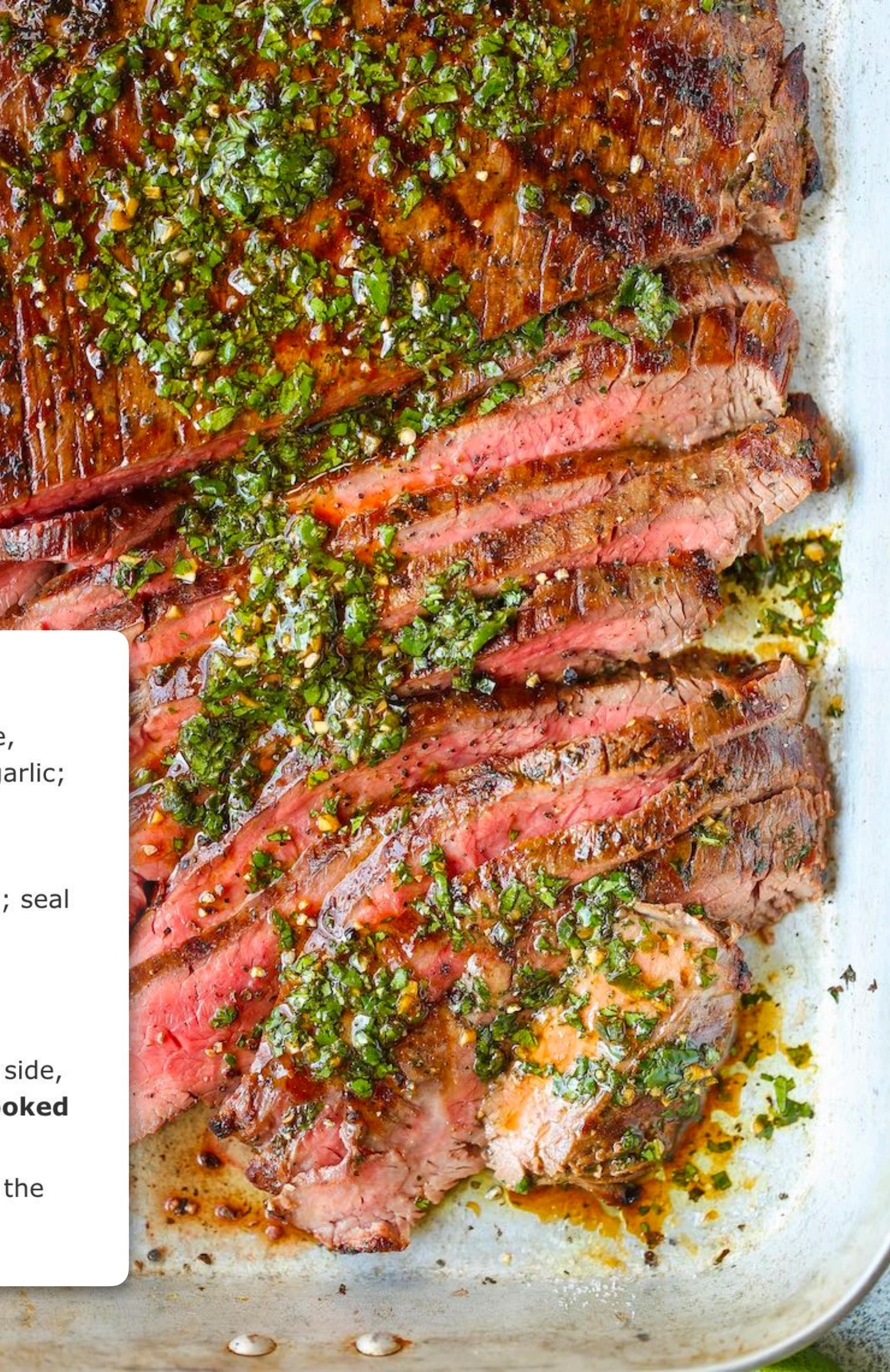
# Kind of like the instructions in a recipe for your favorite meals

Mmm, carne asada



## Instructions

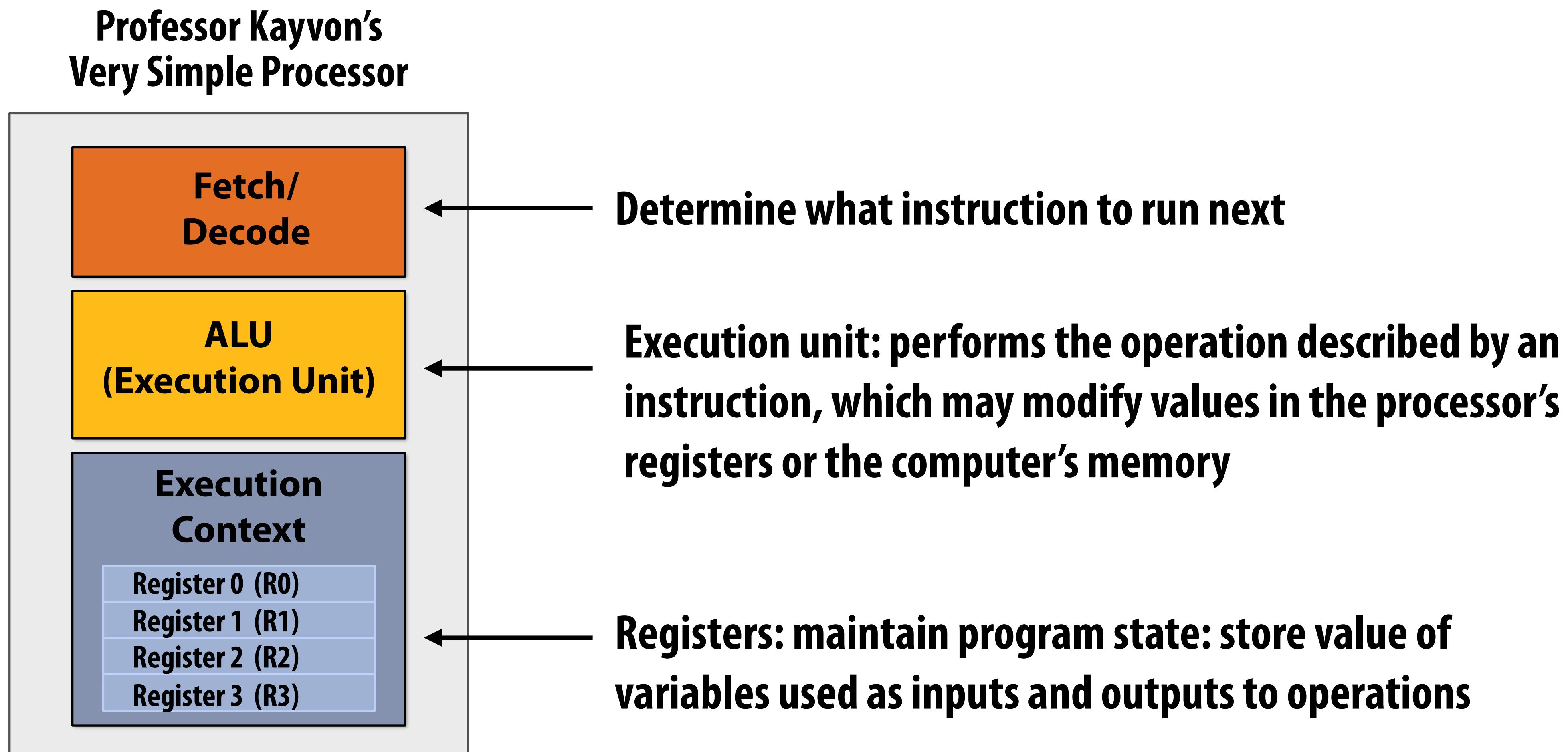
1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeno, and garlic; whisk until well combined.
2. Reserve  $\frac{1}{3}$  cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and 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.



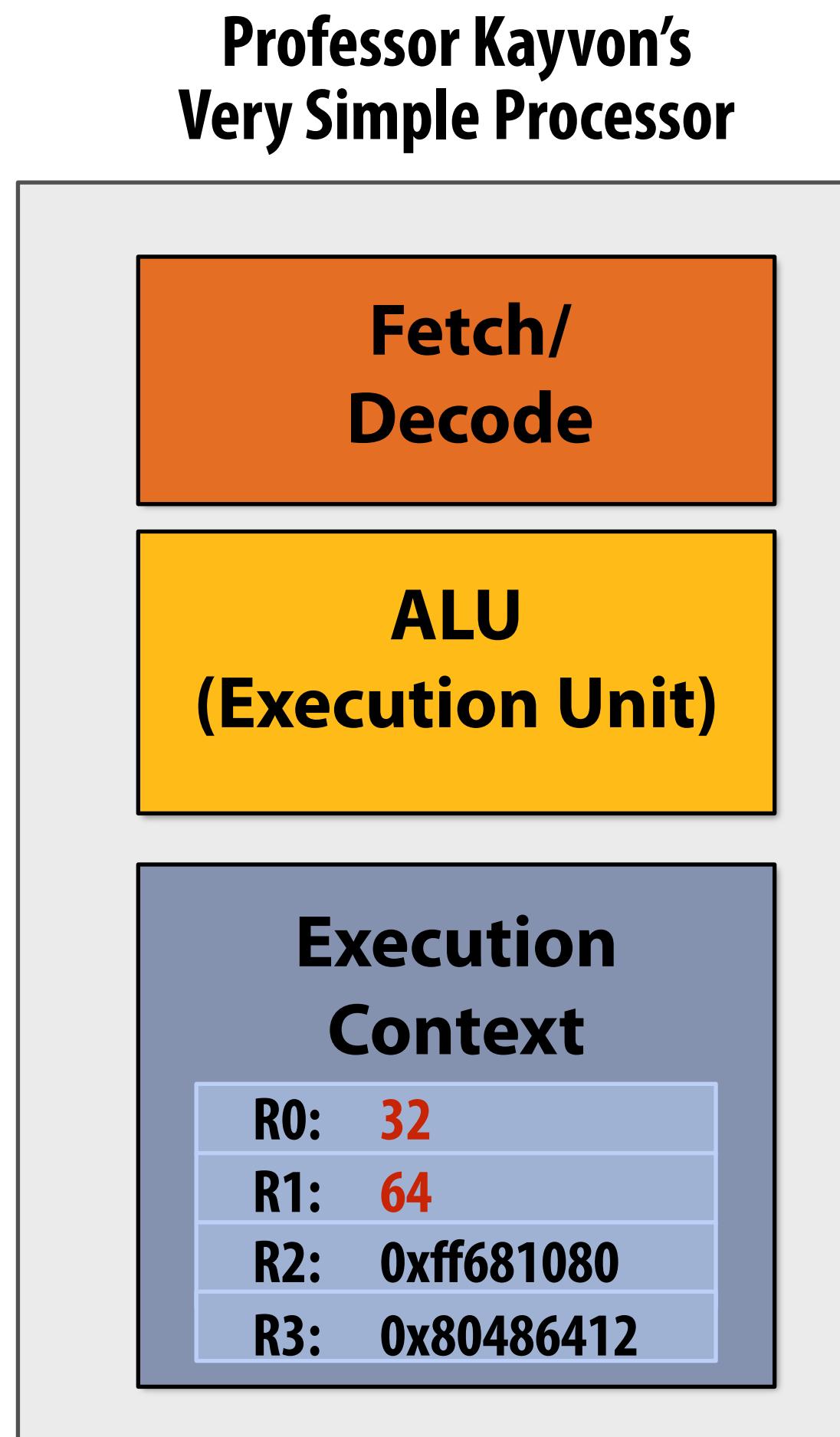
# What does a processor do?



# A processor executes instructions



# One example instruction: add two numbers



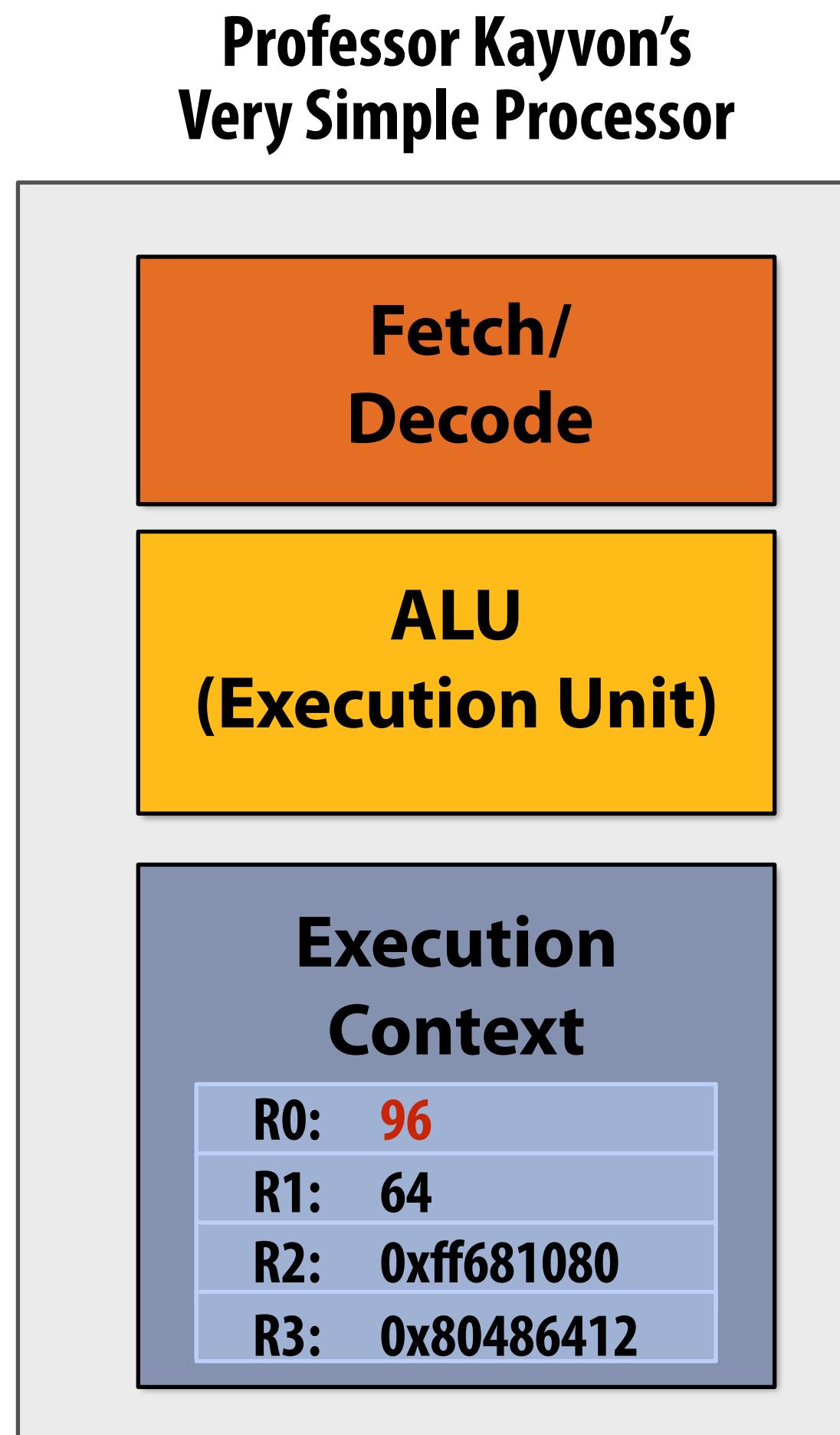
**Step 1:**  
Processor gets next program instruction from memory  
(figure out what the processor should do next)

**add R0  $\leftarrow$  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



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

**add R0  $\leftarrow$  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**

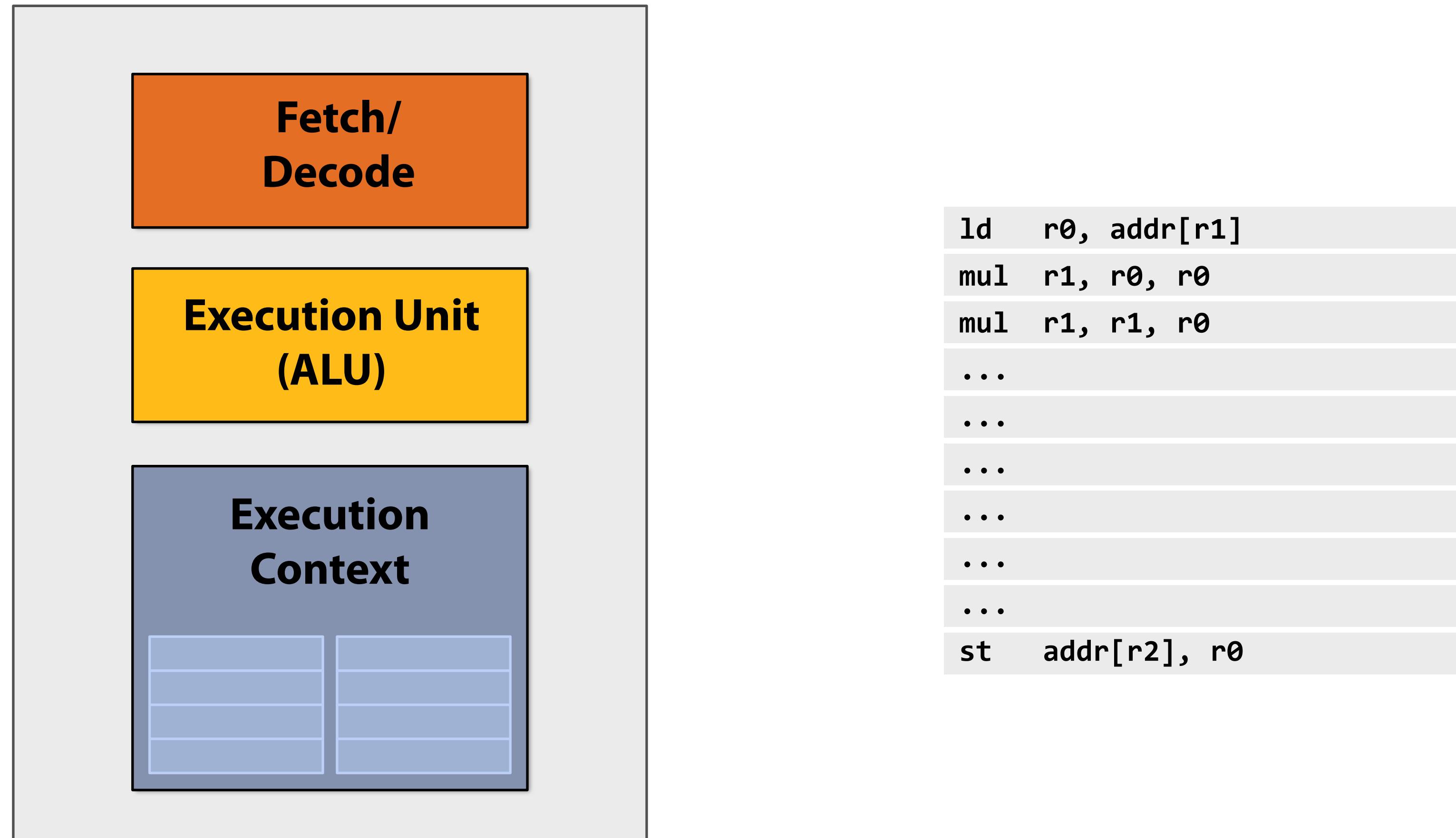
Contents of R1 input to execution unit: **64**

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

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

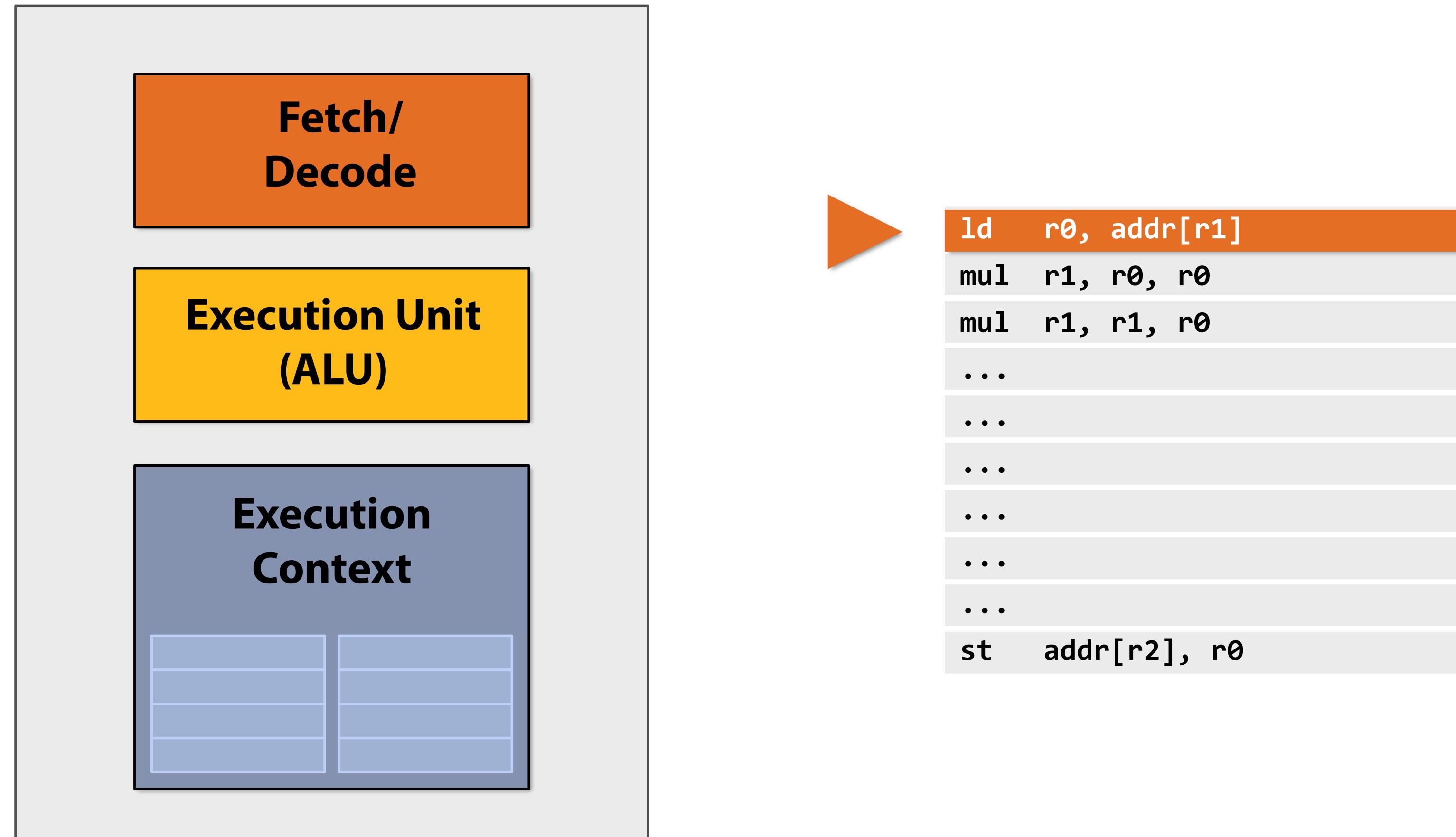
# Execute program

My very simple processor: executes one instruction per clock



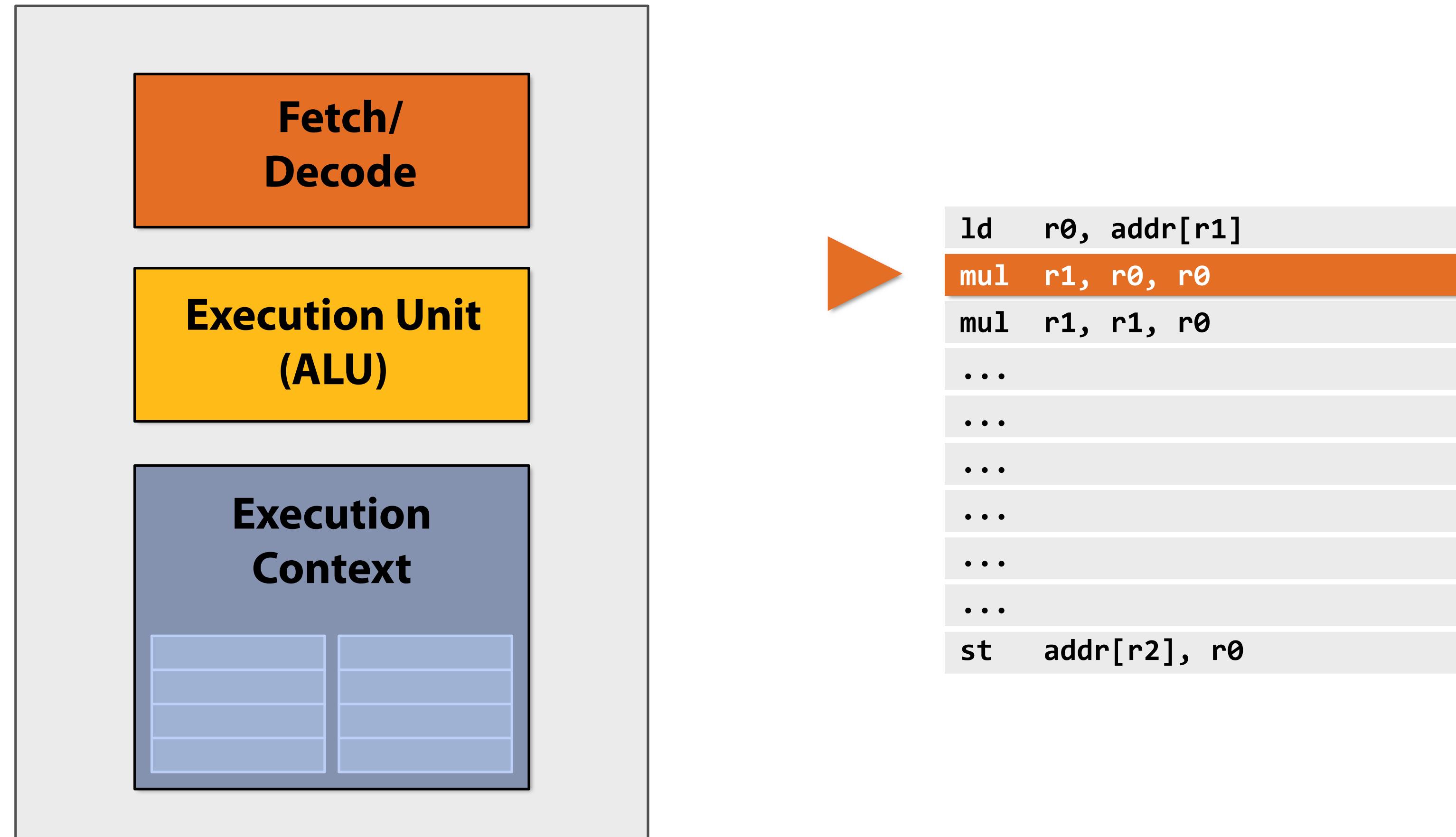
# Execute program

My very simple processor: executes one instruction per clock



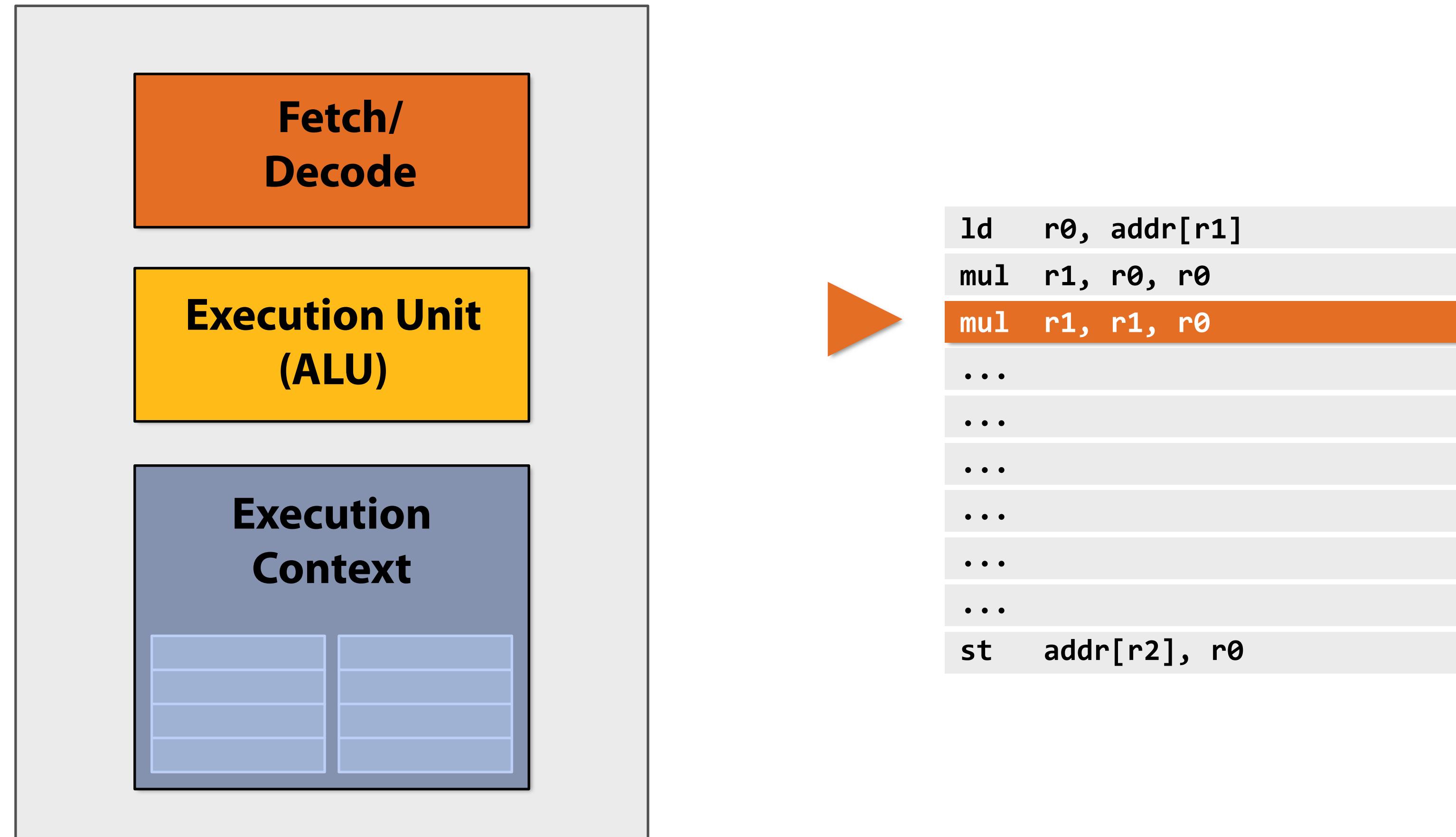
# Execute program

My very simple processor: executes one instruction per clock



# Execute program

My very simple processor: executes one instruction per clock



# 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  $R0 = x, R1 = y, R2 = z$

```
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'

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?

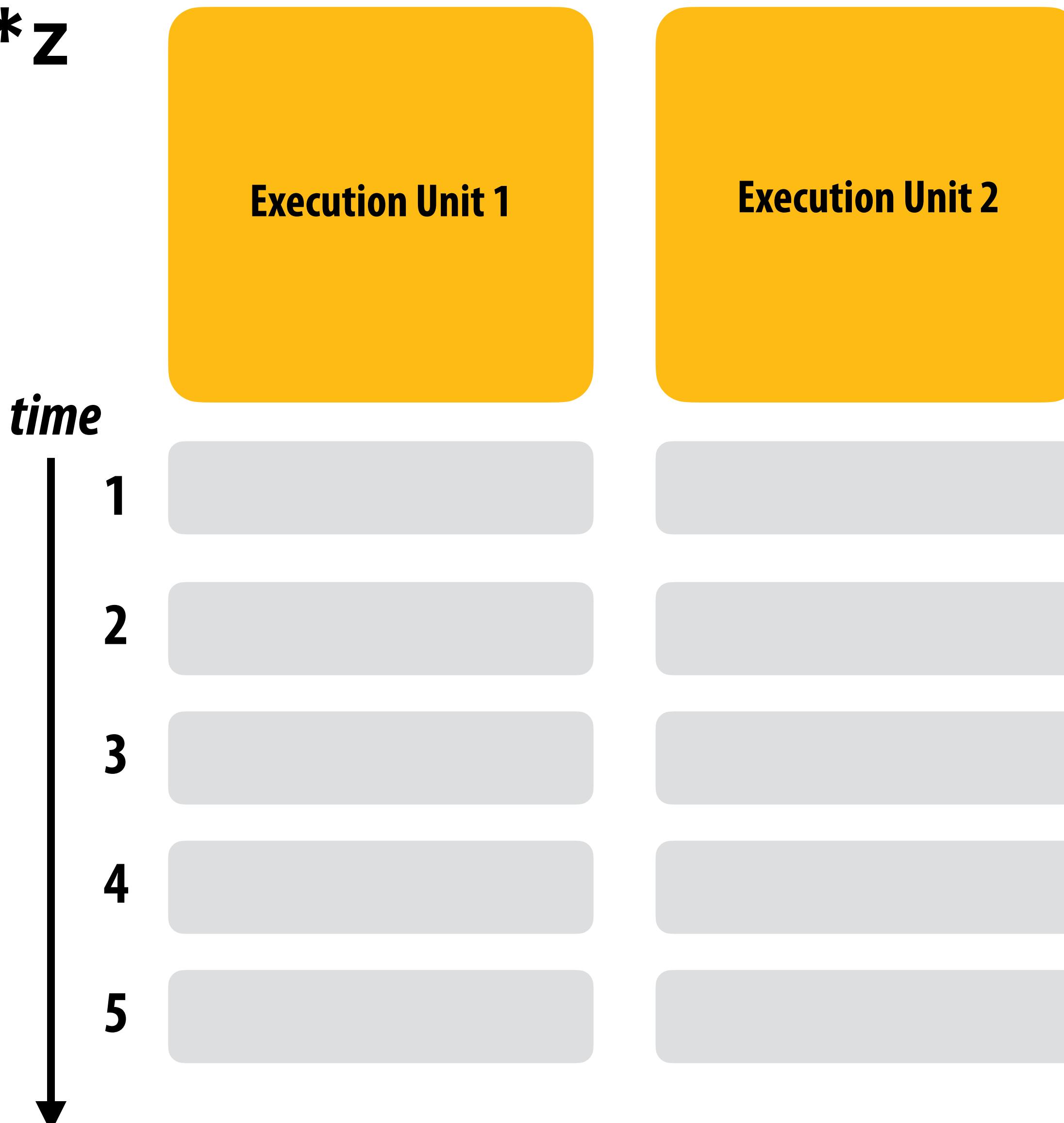
$$a = x*x + y*y + z*z$$

*Assume register*

$$R0 = x, R1 = y, R2 = z$$

```
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'*



**Question:**

**Can you assign each of the five instructions in the program to an execution unit such that:**

- If instruction X depends on the result of instruction Y, then X executes at a later time than Y.**
- The entire 5-instruction program completes as quickly as possible.**

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

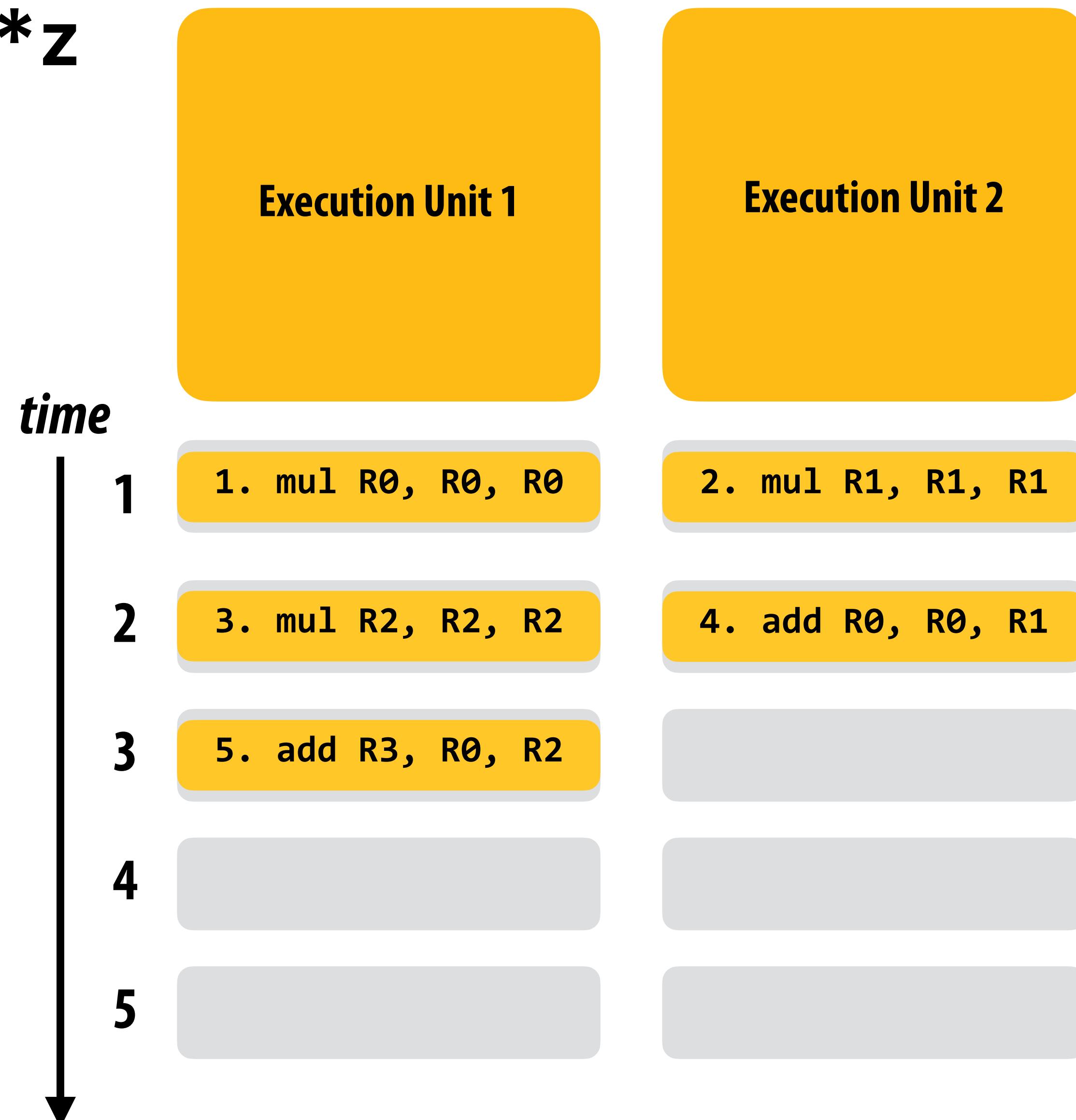
$$a = x*x + y*y + z*z$$

*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'*



## **QUESTION:**

**What does it mean for our parallel to scheduling to  
that “respects program order”?**

**Hint: What is expected of the output.**

# What about three instructions at once?

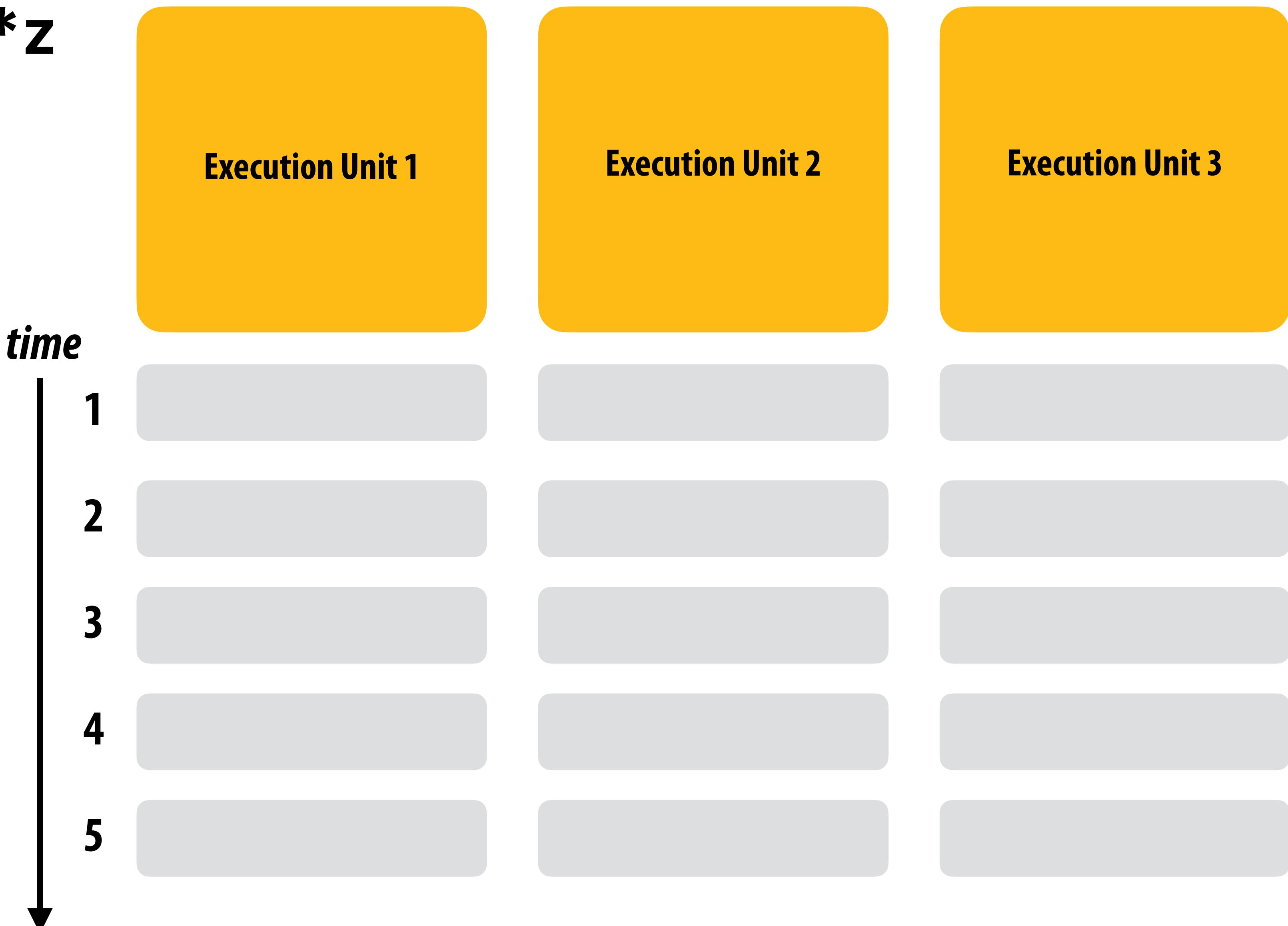
$$a = x*x + y*y + z*z$$

*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'*



# What about three instructions at once?

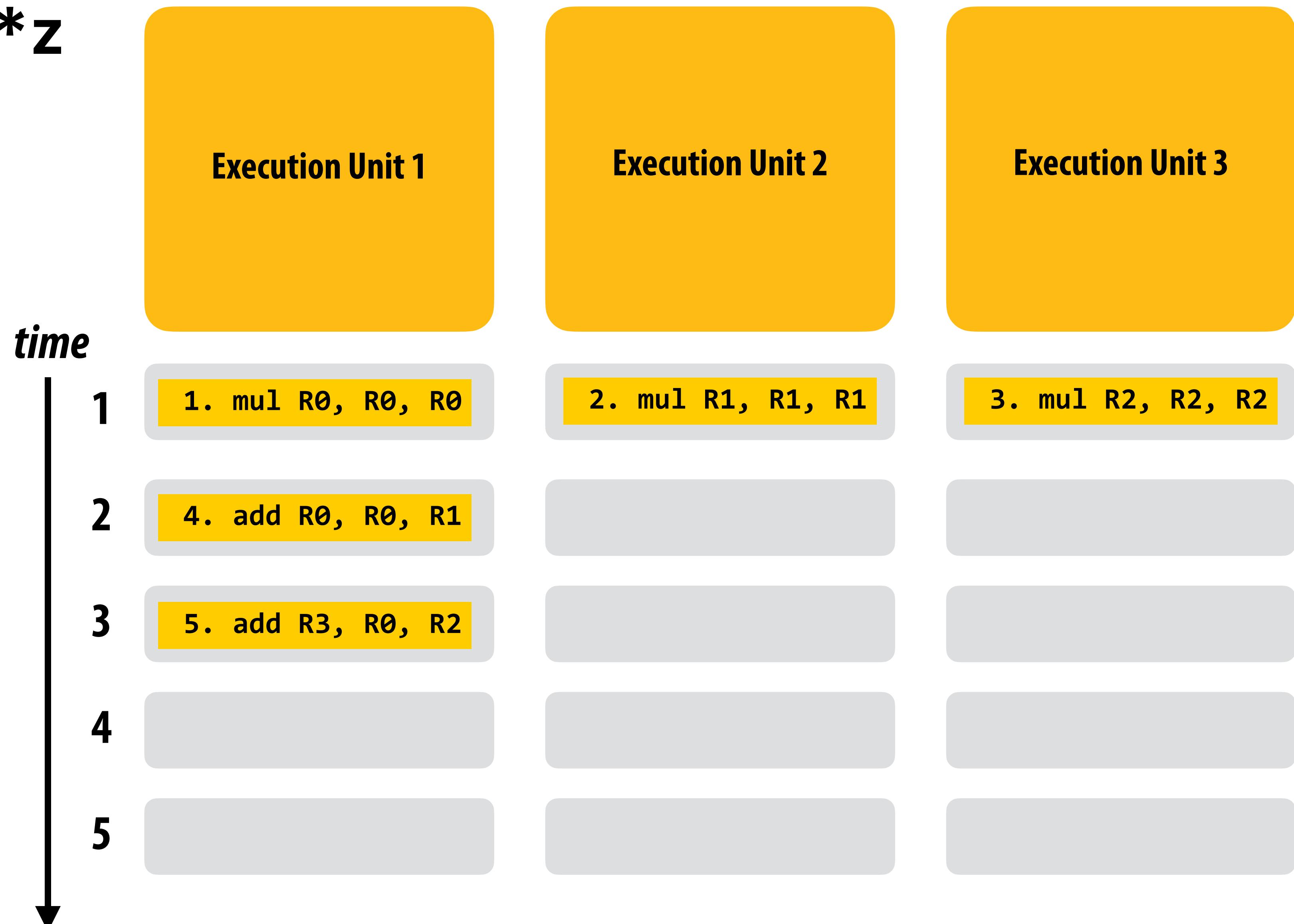
$$a = x*x + y*y + z*z$$

*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
```

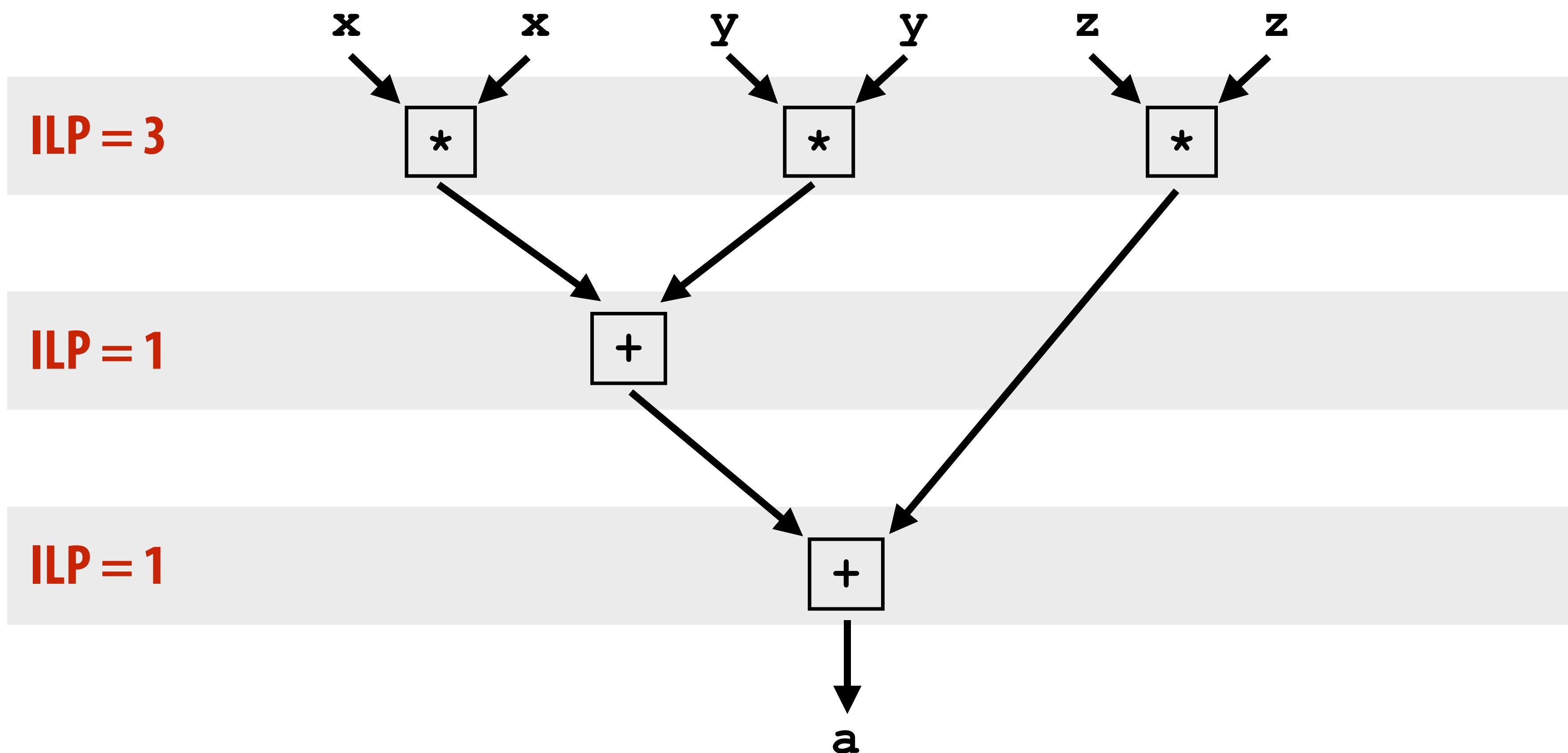
*R3 now stores value of  
program variable 'a'*



# 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)**

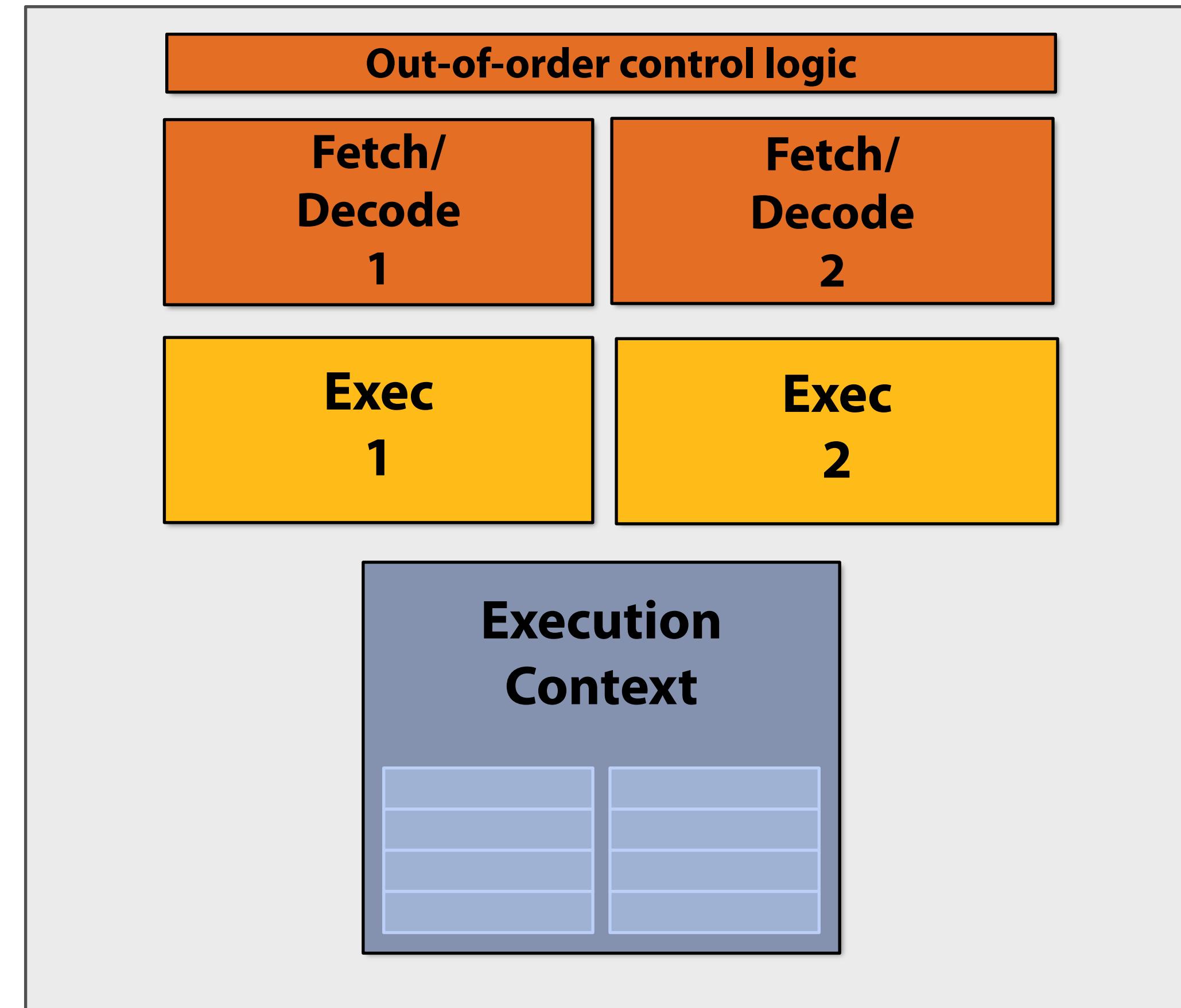
**But instruction 4 must be executed after instructions 1 and 2**

**And instruction 5 must be executed after instruction 4**

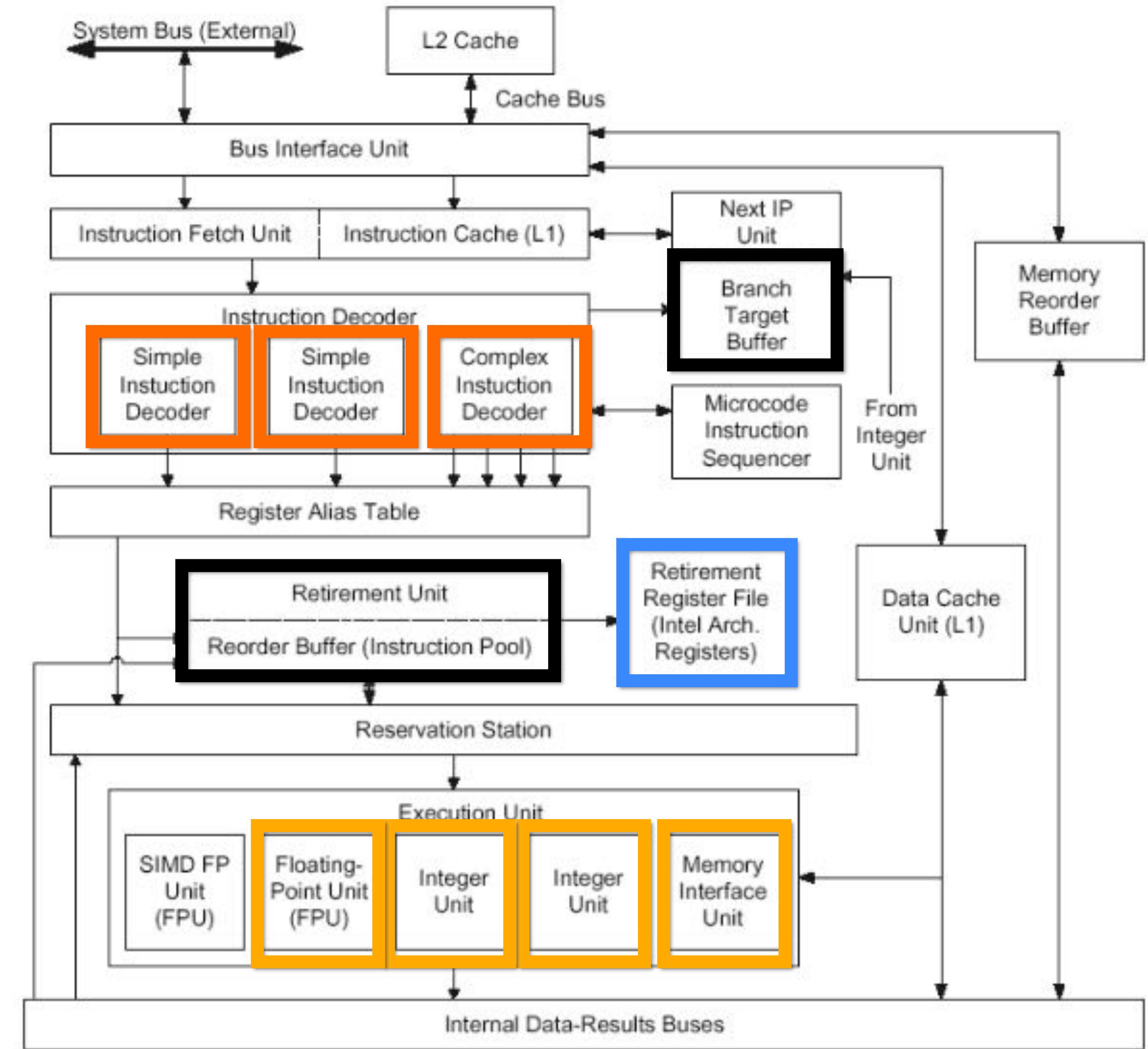
\* Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.

# Superscalar processor

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



# Aside: Old Intel Pentium 4 CPU

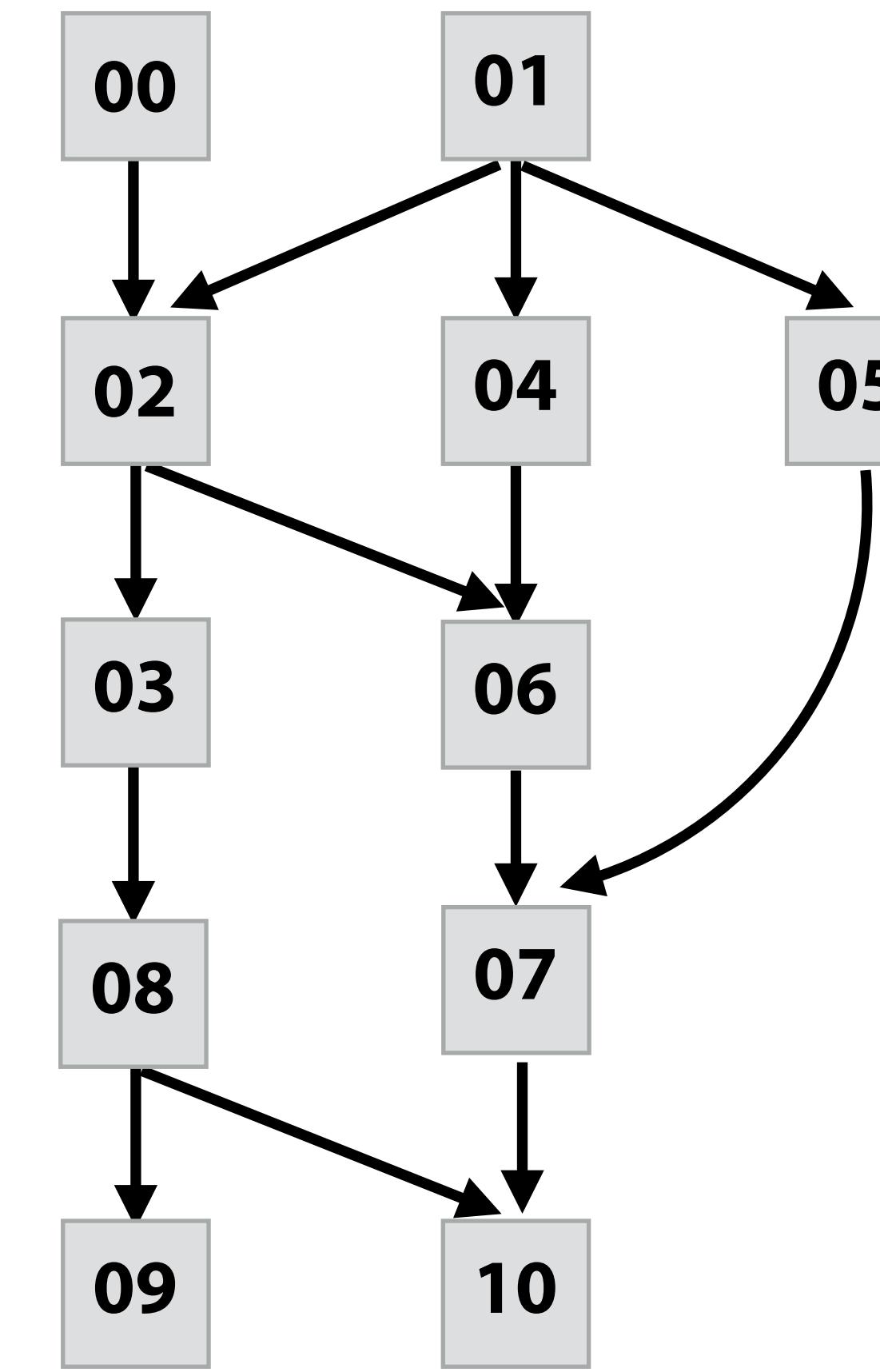


# A more complex example

Program (sequence of instructions)

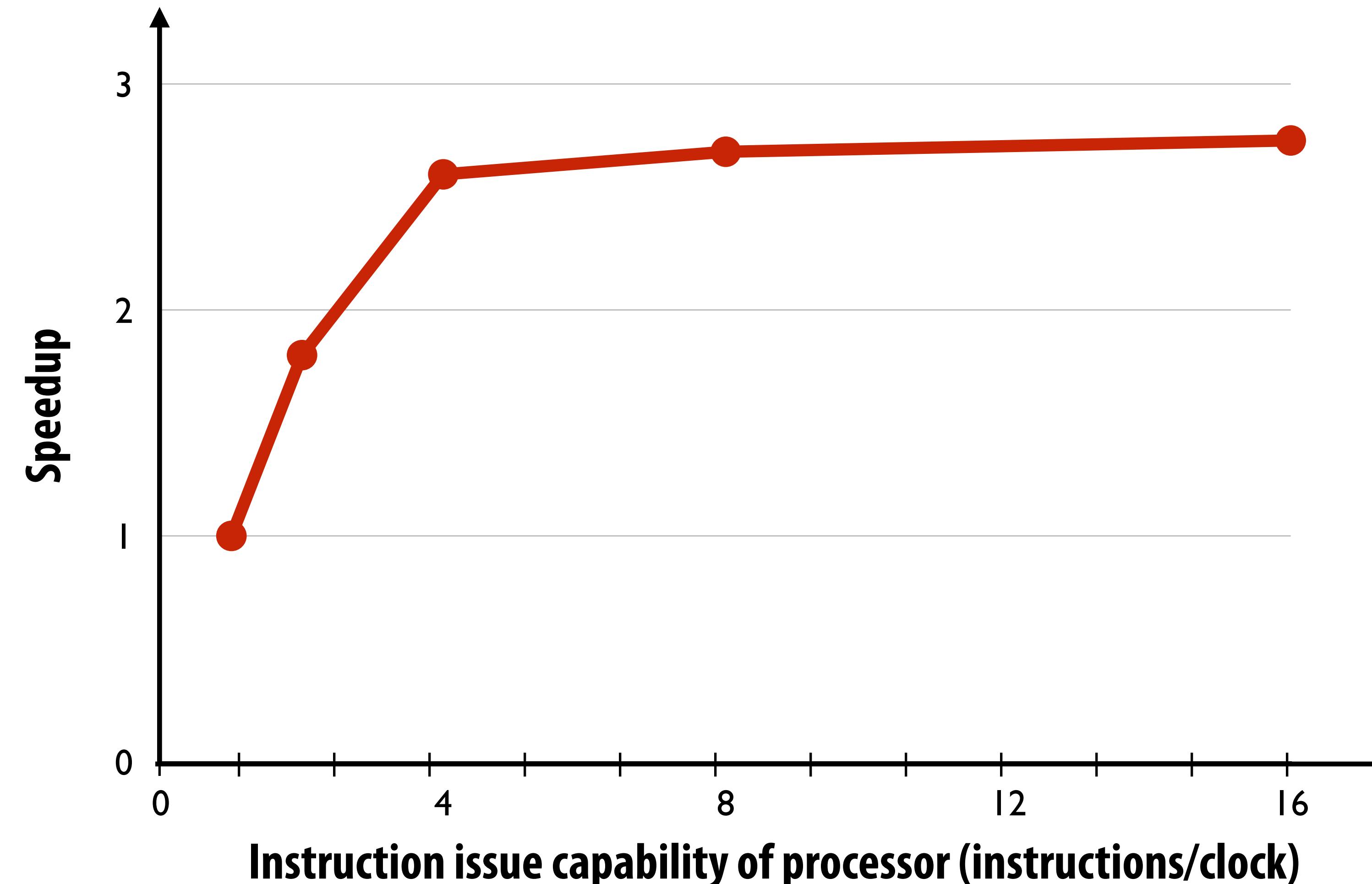
PC	Instruction	Computed value
00	$a = 2$	
01	$b = 4$	
02	$tmp2 = a + b$	// 6
03	$tmp3 = tmp2 + a$	// 8
04	$tmp4 = b + b$	// 8
05	$tmp5 = b * b$	// 16
06	$tmp6 = tmp2 + tmp4$	// 14
07	$tmp7 = tmp5 + tmp6$	// 30
08	if ( $tmp3 > 7$ )	
09	print $tmp3$	
10	else	
	print $tmp7$	

Instruction dependency graph



# Diminishing returns of superscalar execution

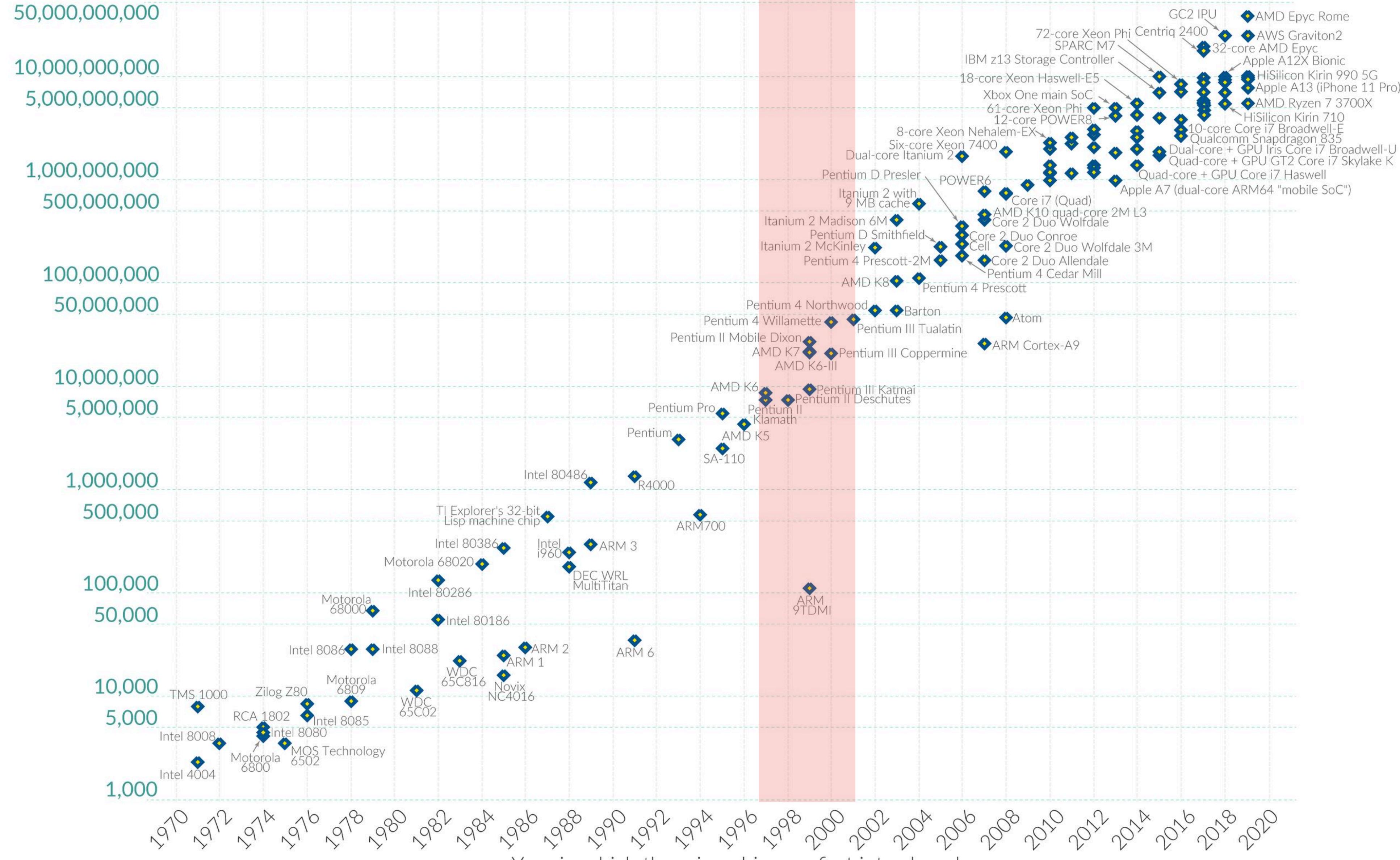
Most available ILP is exploited by a processor capable of issuing four instructions per clock  
(Little performance benefit from building a processor that can issue more)



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

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.

## Transistor count

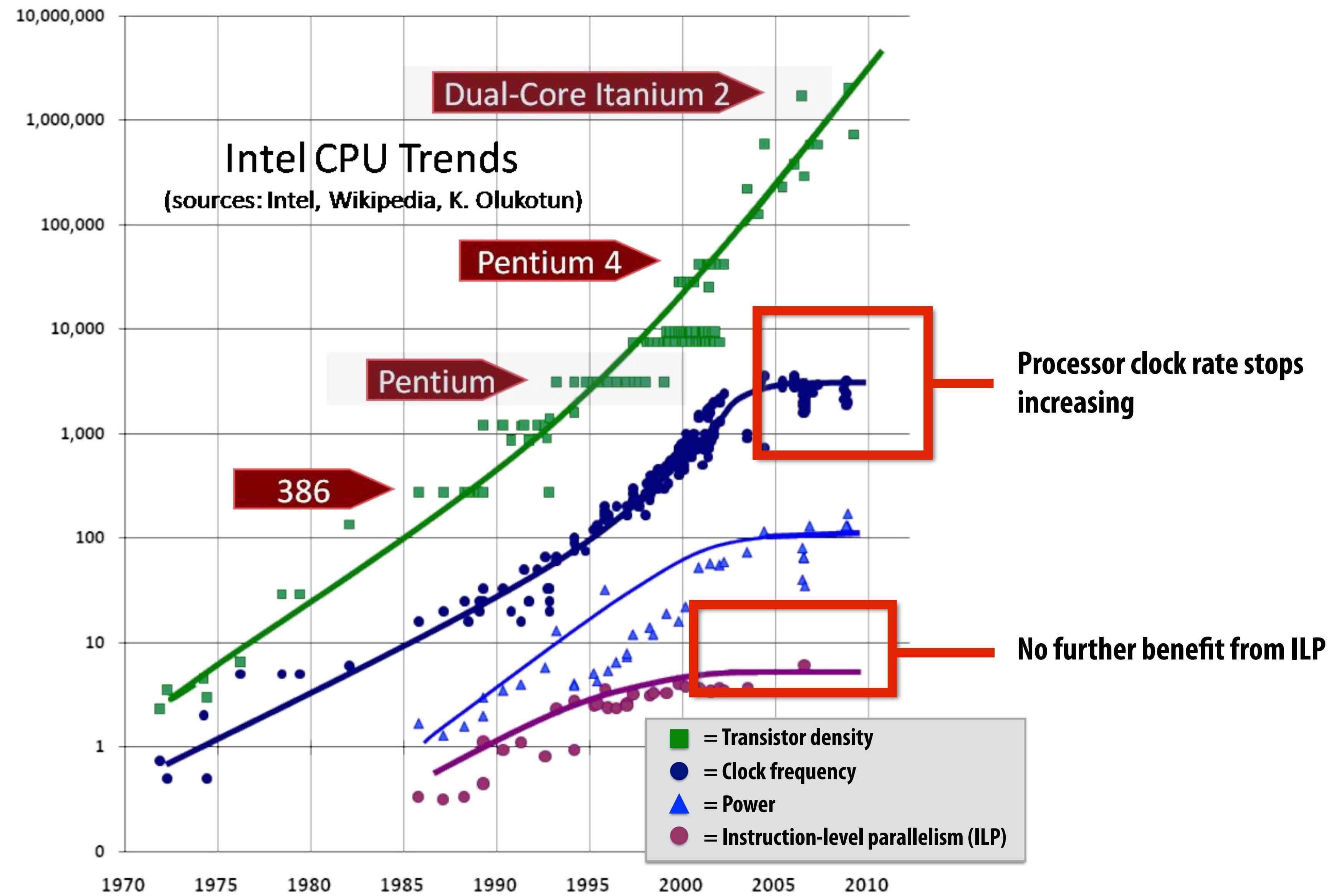


Data source: Wikipedia ([wikipedia.org/wiki/Transistor\\_court](https://en.wikipedia.org/w/index.php?title=Transistor_court&oldid=1000000000))

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

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

# ILP tapped out + end of frequency scaling



# The “power wall”

Power consumed by a transistor:

**Dynamic power  $\propto$  capacitive load  $\times$  voltage<sup>2</sup>  $\times$  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**

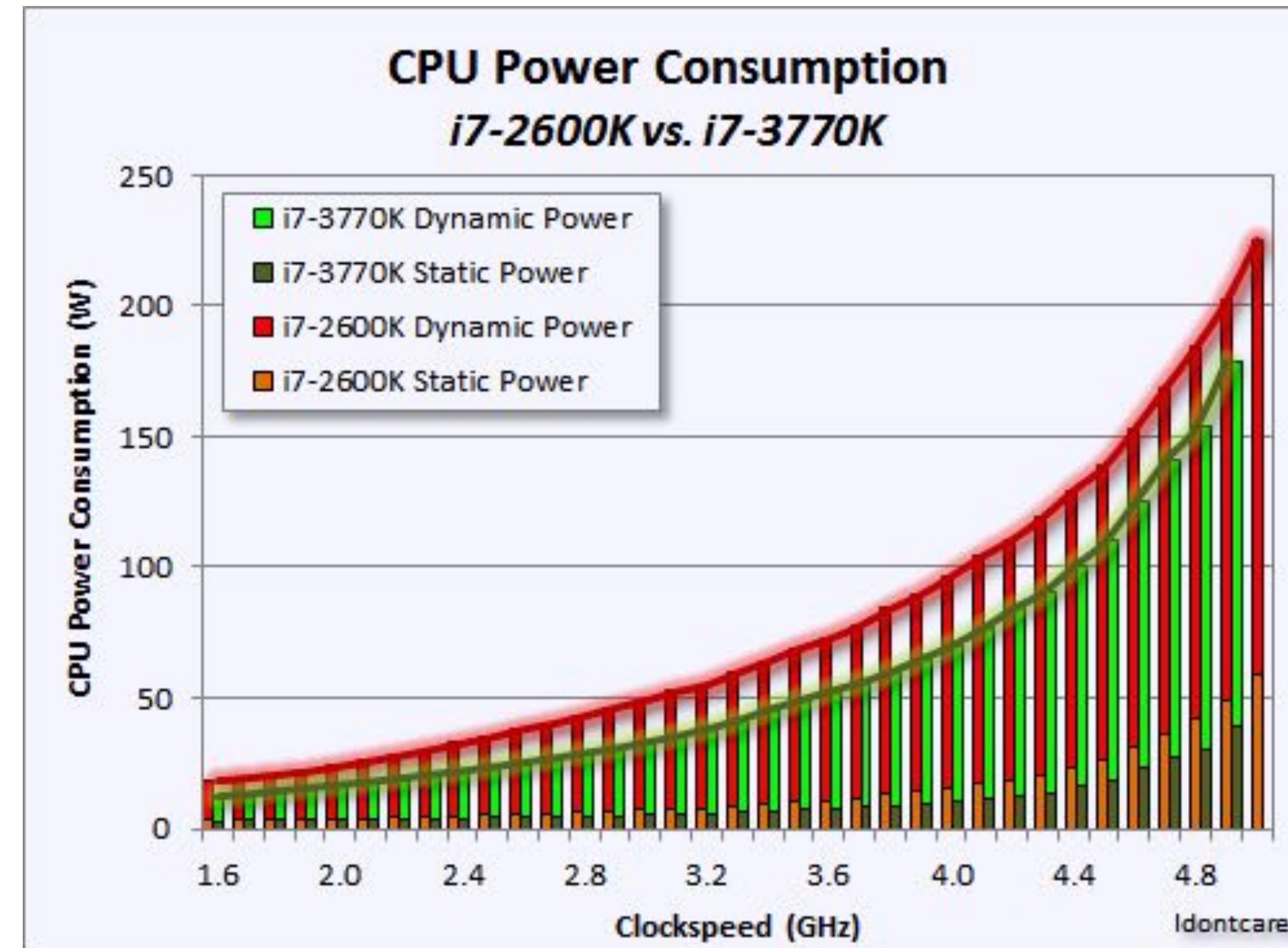


# Power draw as a function of clock frequency

Dynamic power  $\propto$  capacitive load  $\times$  voltage<sup>2</sup>  $\times$  frequency

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

Maximum allowed frequency determined by processor's core voltage



# Single-core performance scaling

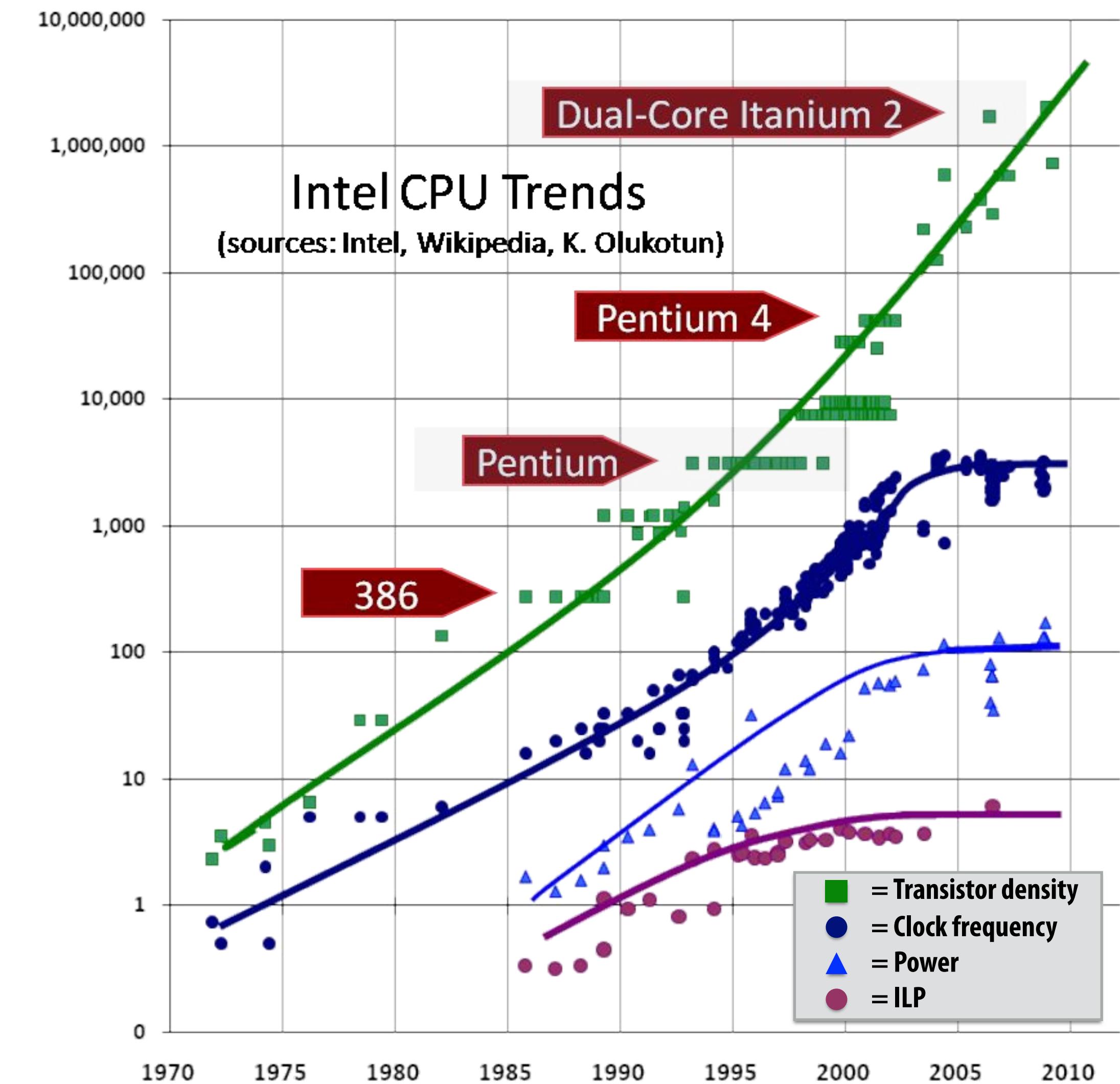
The rate of single-instruction stream performance scaling has decreased (almost to zero)

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!**



# Example: multi-core CPU

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

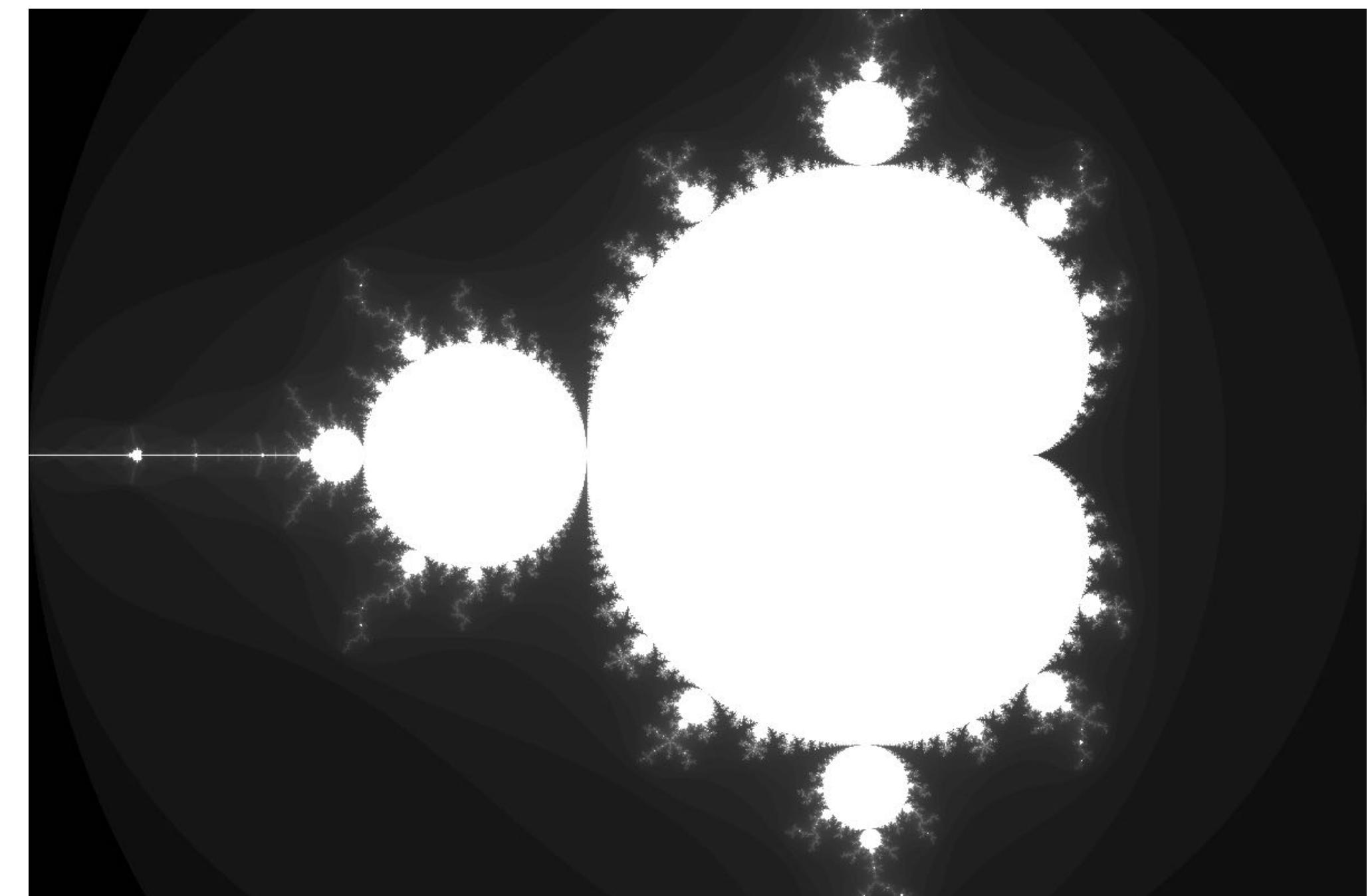


# One thing you will learn in this course

- 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...

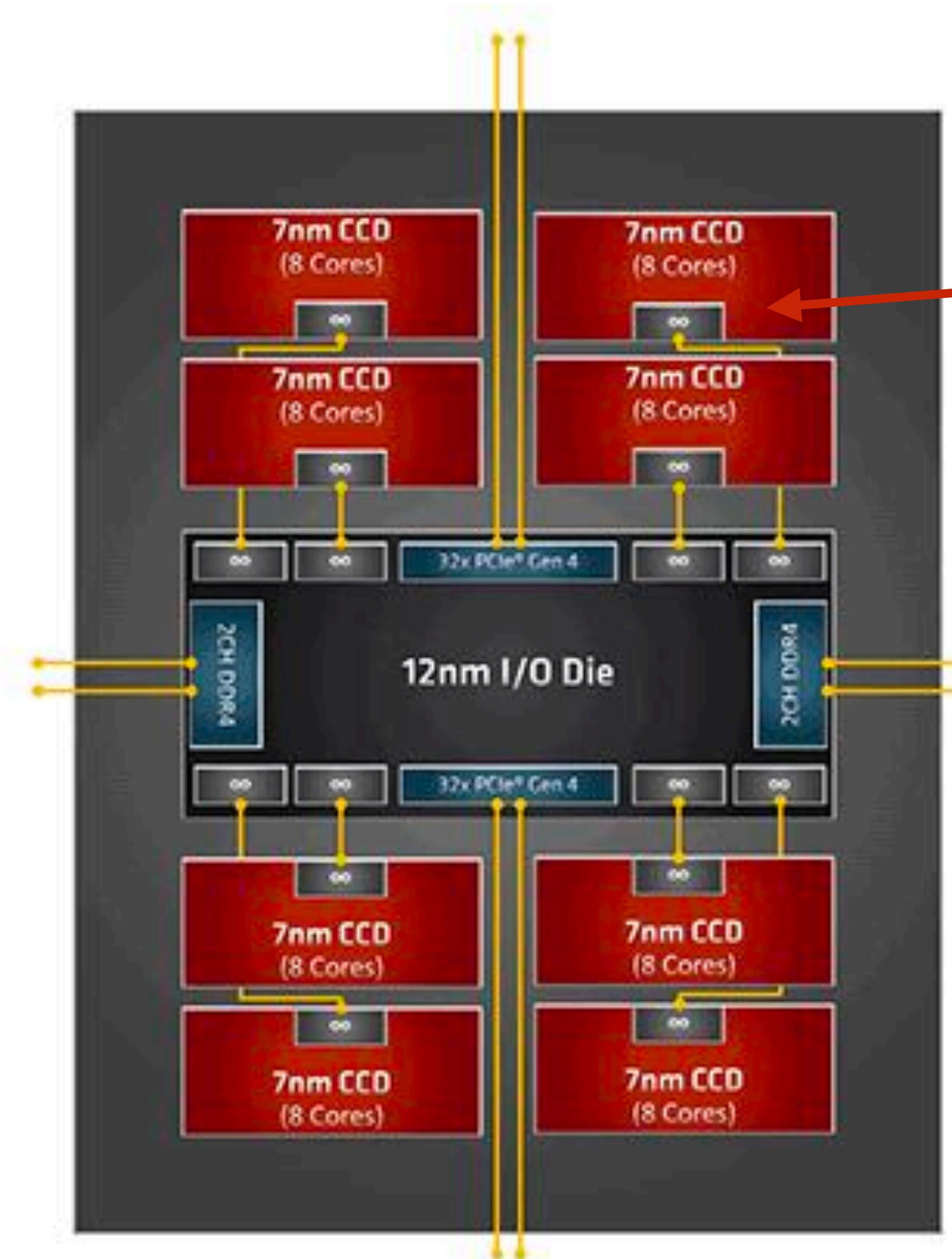
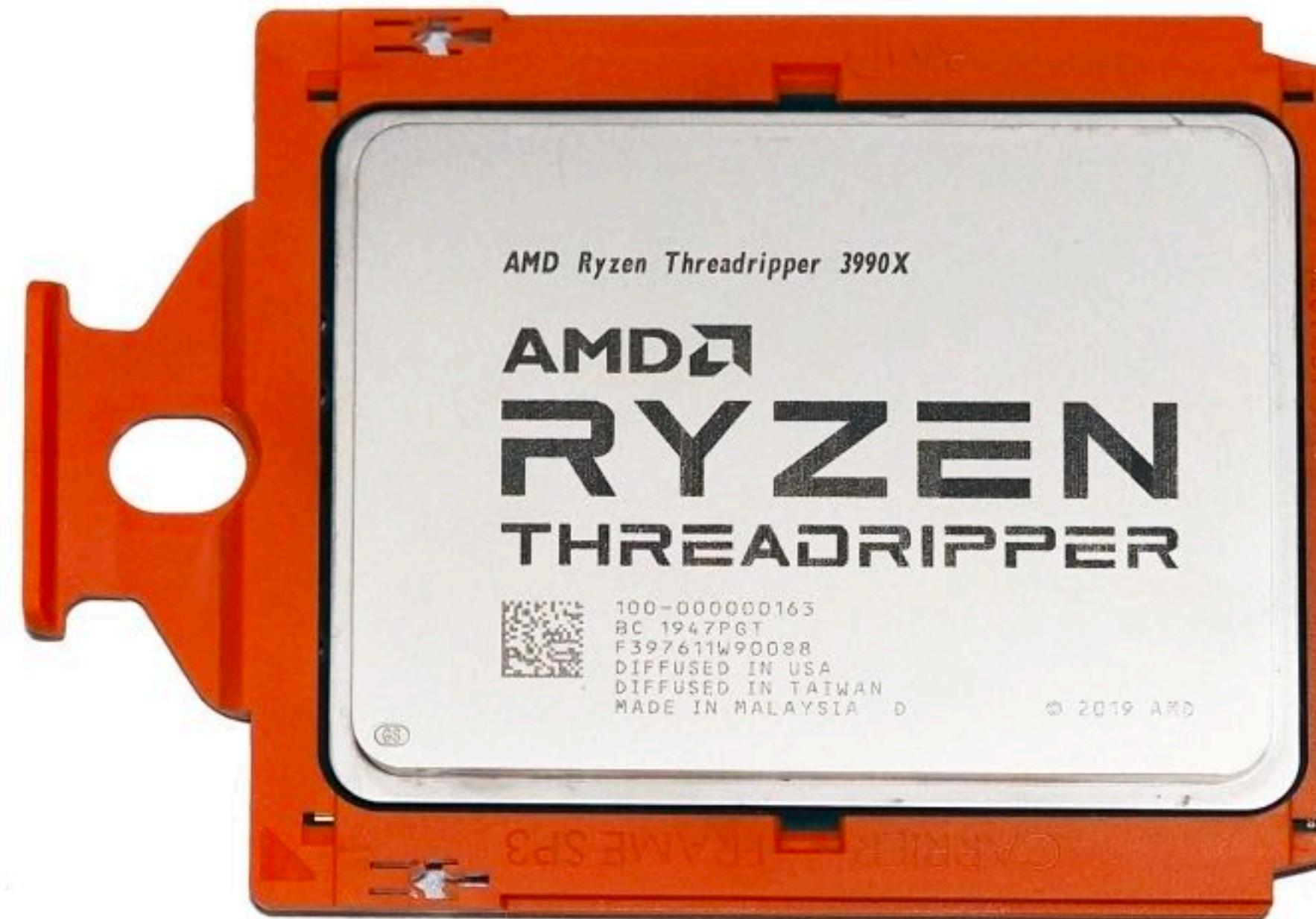
~32-40x faster!

We'll talk about these terms next time!



# AMD Ryzen Threadripper 3990X

64 cores, 4.3 GHz



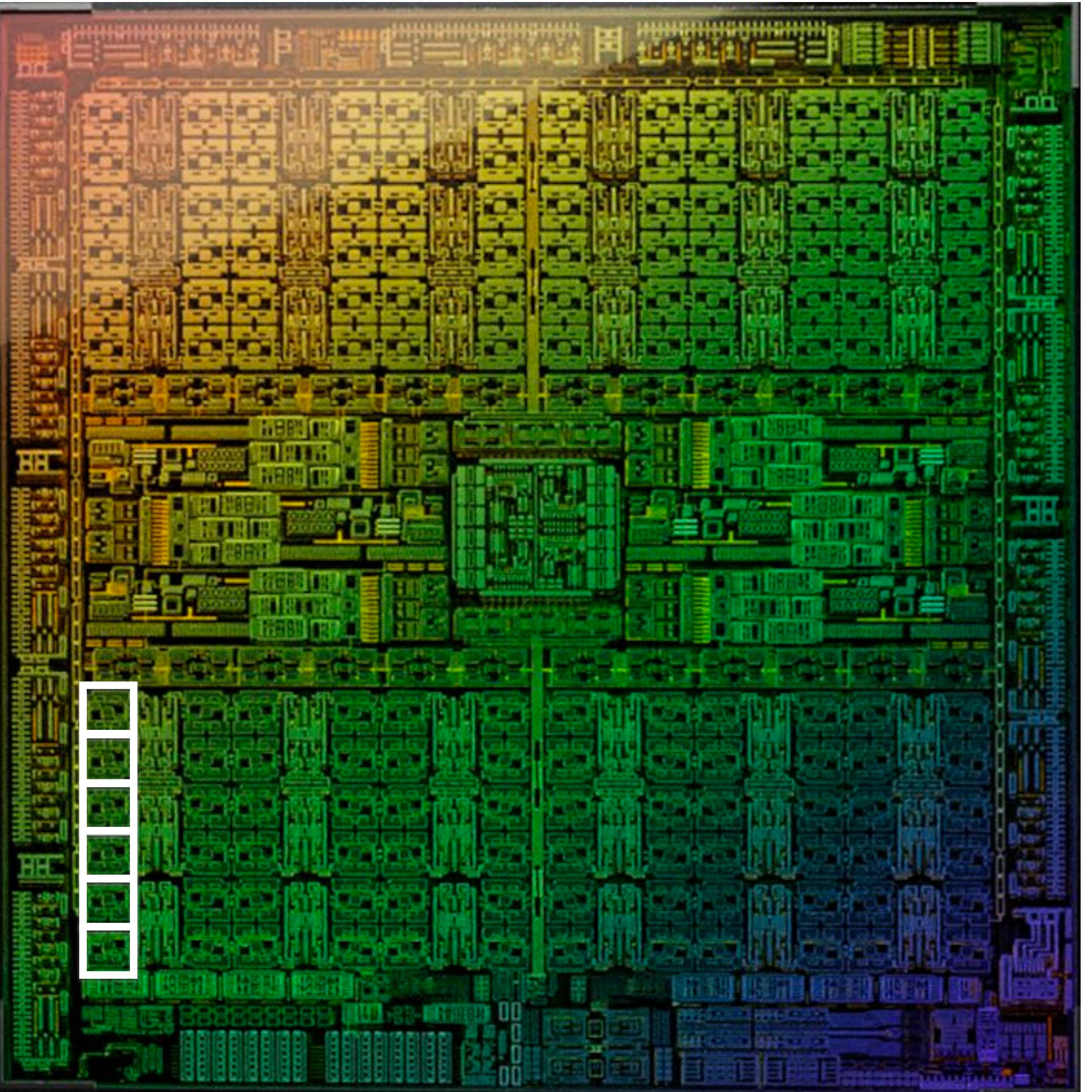
Four 8-core chiplets

# NVIDIA AD102 GPU

GeForce RTX 4090 (2022)

76 billion transistors

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



# GPU-accelerated supercomputing



Frontier (at Oak Ridge National Lab)  
(world's #1 in Fall 2022)

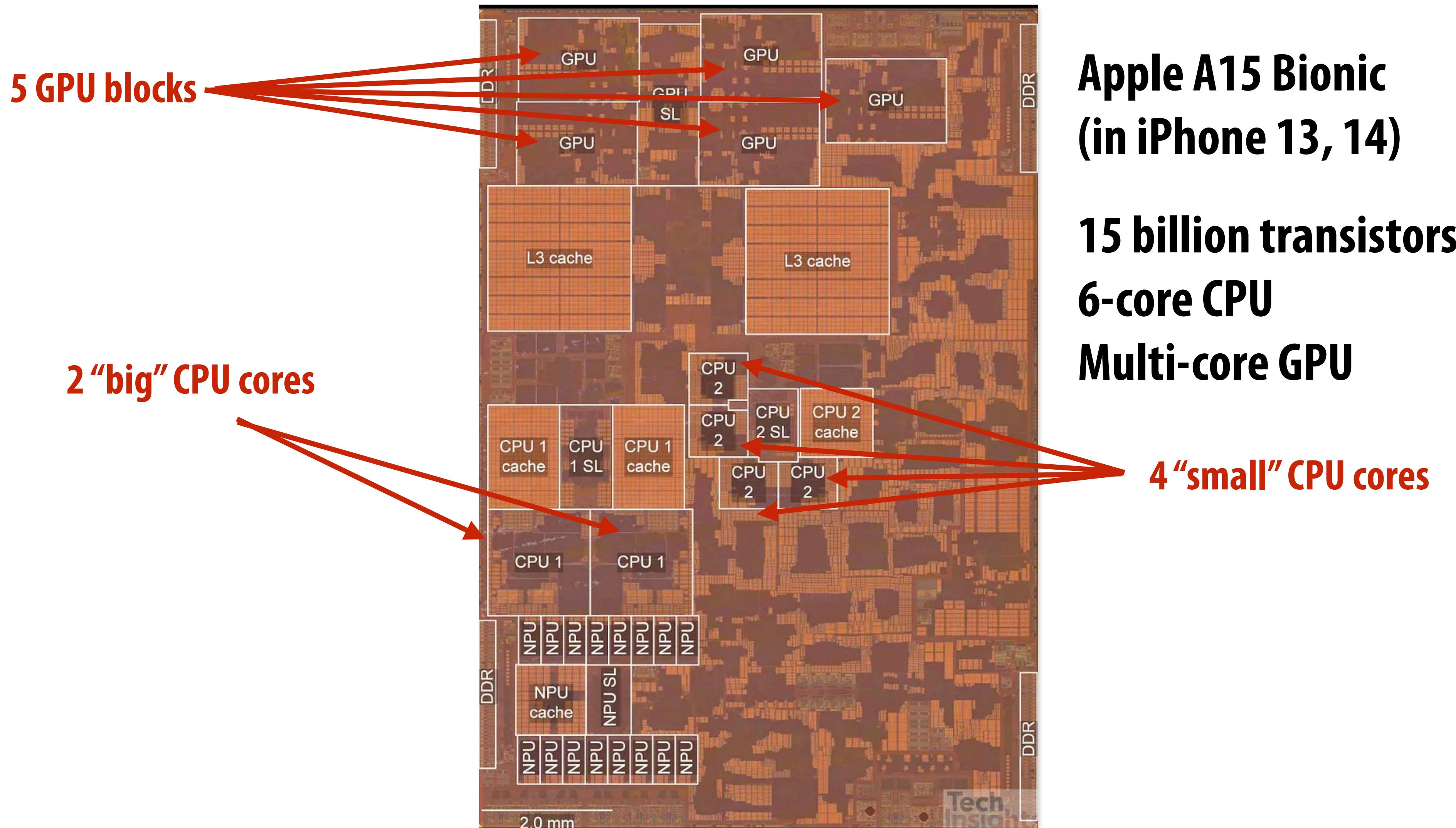
9472 x 64 core AMD CPUs (606,208 CPU cores)

37,888 Radeon GPUs

21 Megawatts

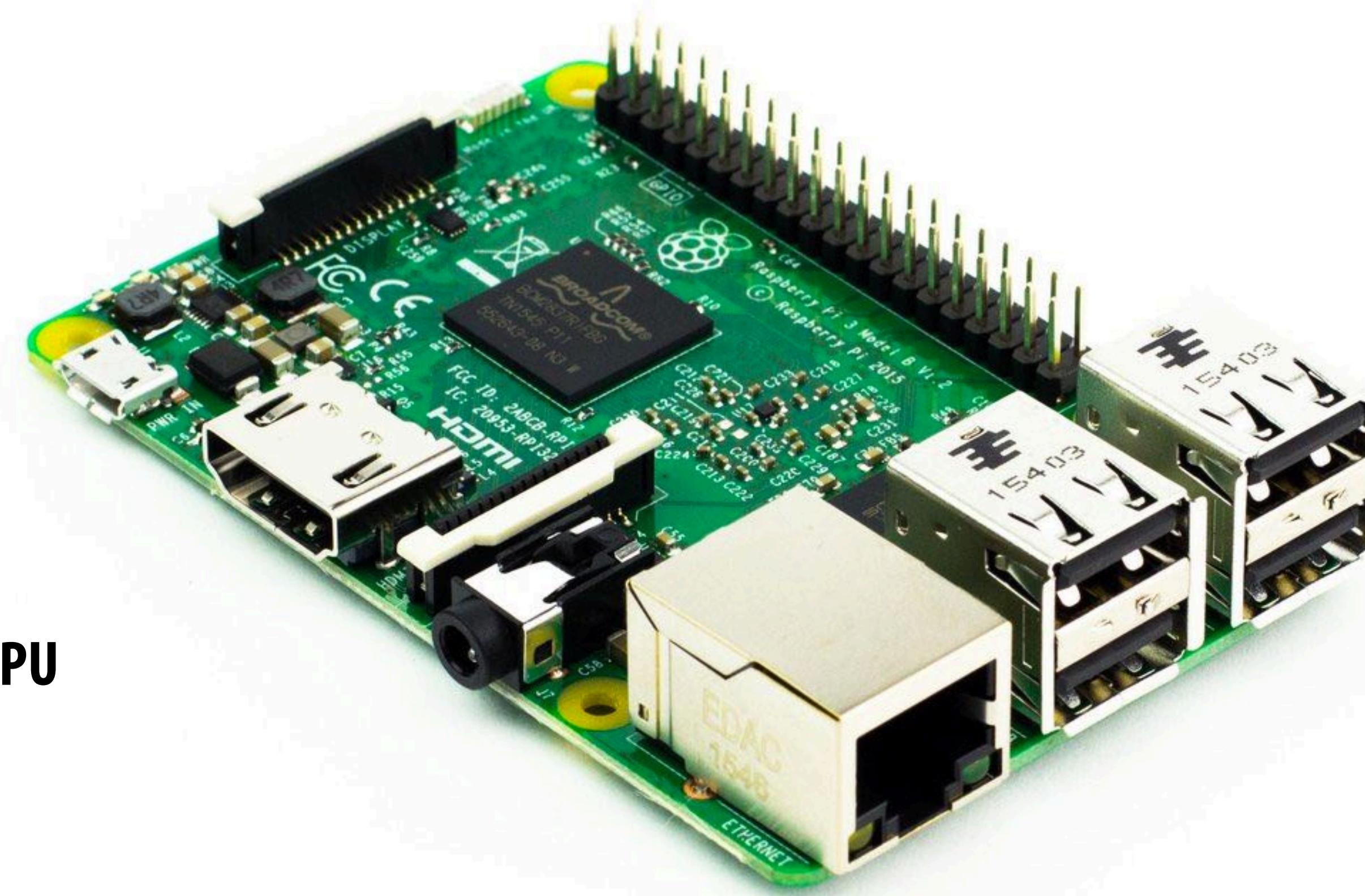
# Mobile parallel processing

Power constraints also heavily influence the design of mobile systems



# 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**

**all**  
**Q. What is a big concern in mobile computing?**

# A. Power

# Two reasons to save power

Run at *higher performance*  
for a *fixed* amount of time.



**Power = heat**  
If a chip gets too hot, it must be  
clocked down to cool off \*

Run at *sufficient performance*  
for a *longer* amount of time.



**Power = battery**  
Long battery life is a desirable  
feature in mobile devices

\* Another reason: hotter systems cost more to cool.

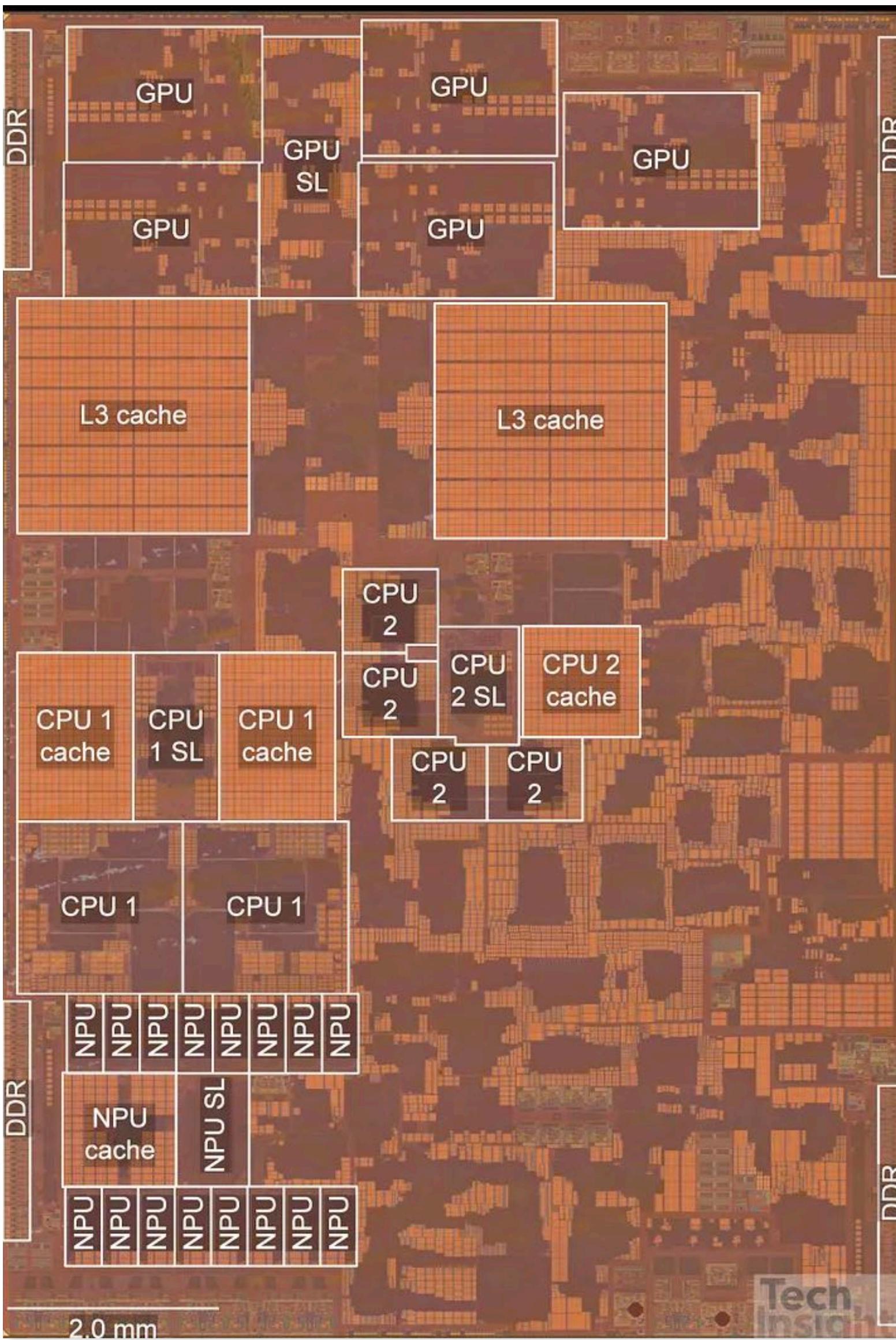
# Mobile phone example

Apple iPhone 13



**3227 mAmp hours  
(12.4 Watt hours)**

# Specialized processing is ubiquitous in mobile systems



**Apple A15 Bionic  
(in iPhone 13, 14)**

**15 billion transistors**

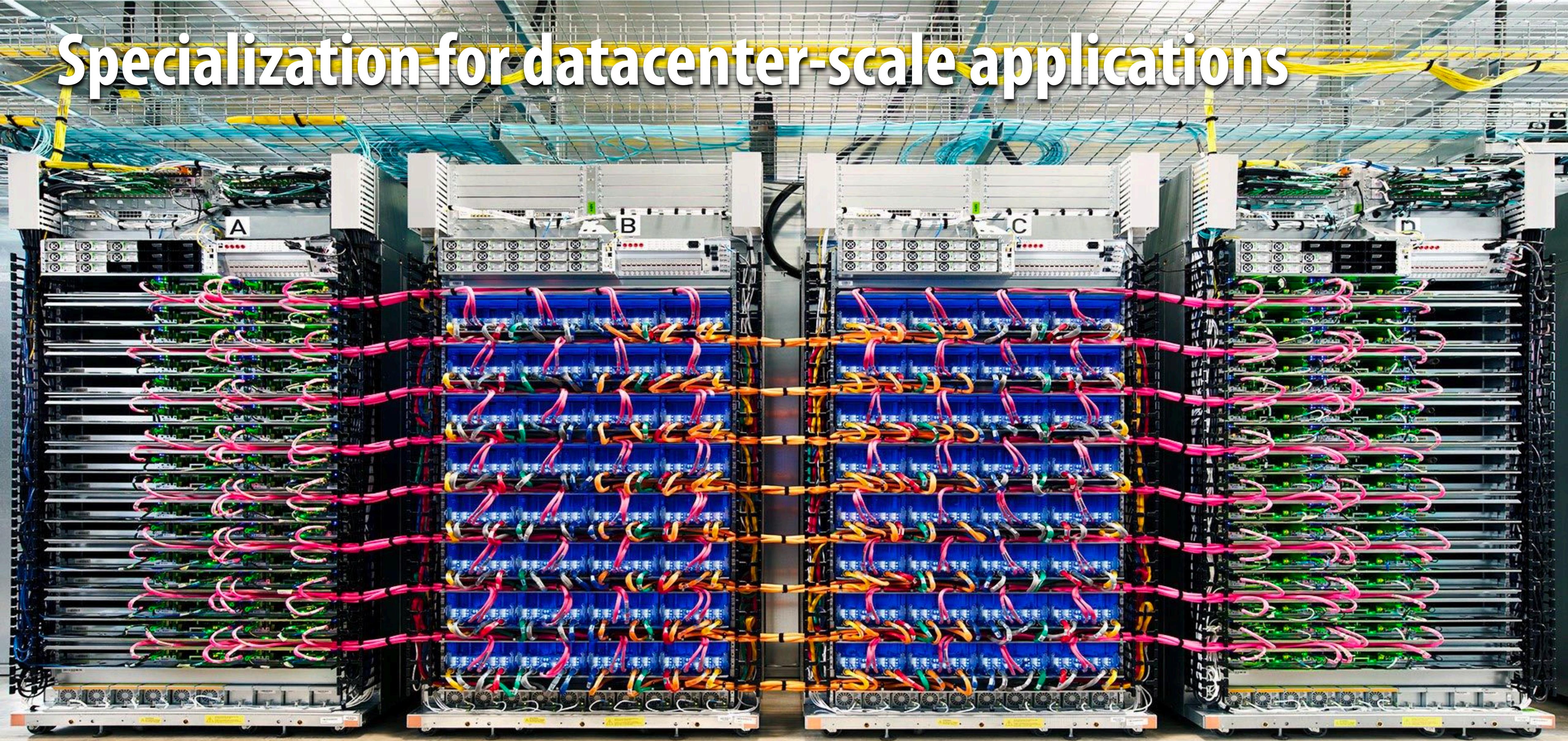
**6-core GPU**  
**2 “big” CPU cores**  
**4 “small” CPU cores**

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

# 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

# Specialization for datacenter-scale applications

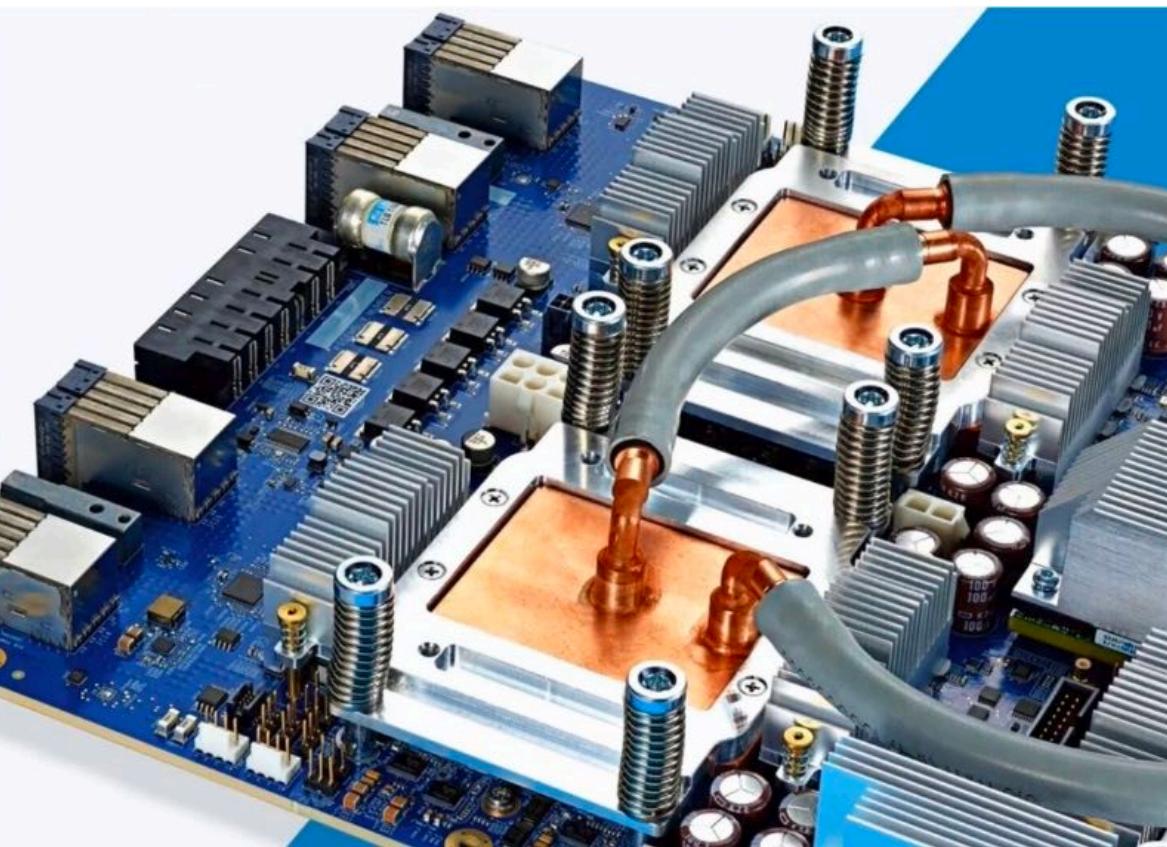


Google TPU pods

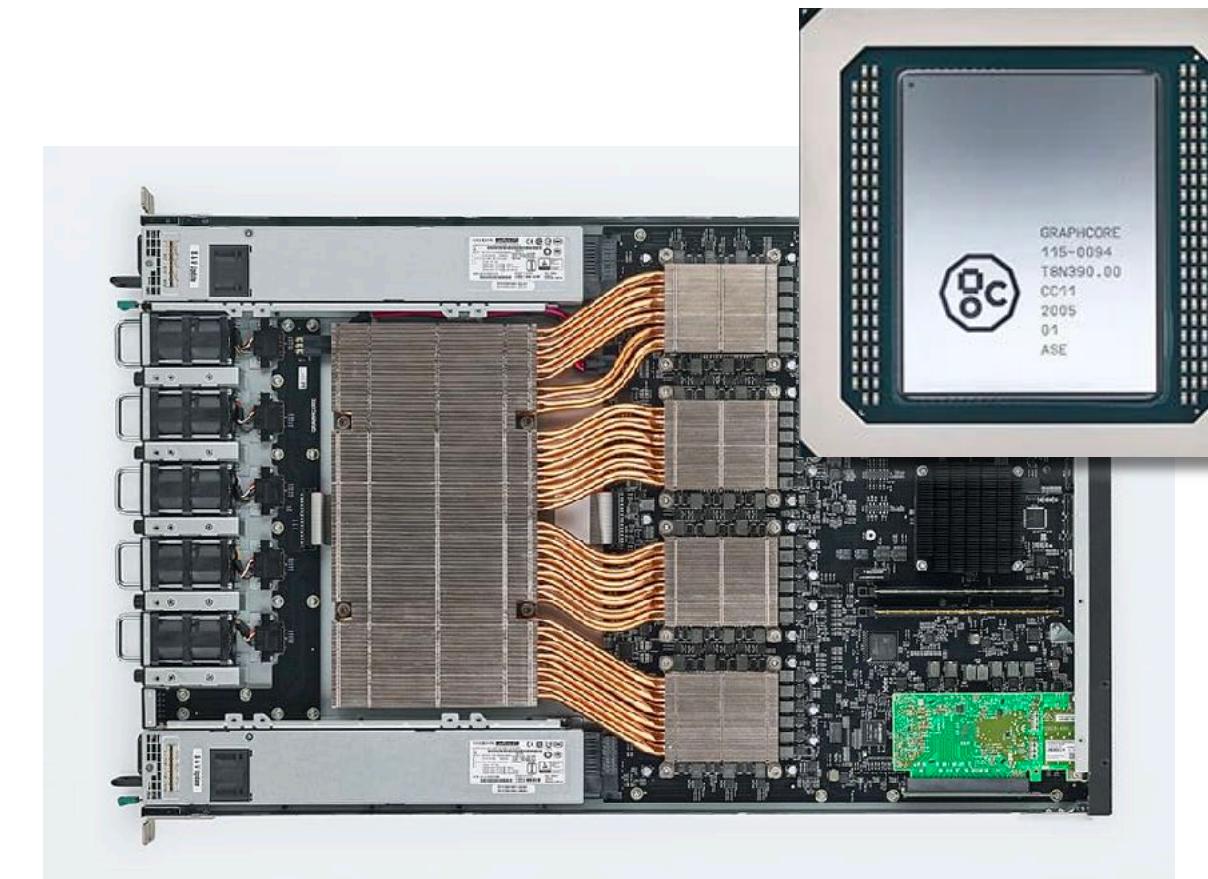
TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.

# Specialized hardware to accelerate DNN inference/training



Google TPU3



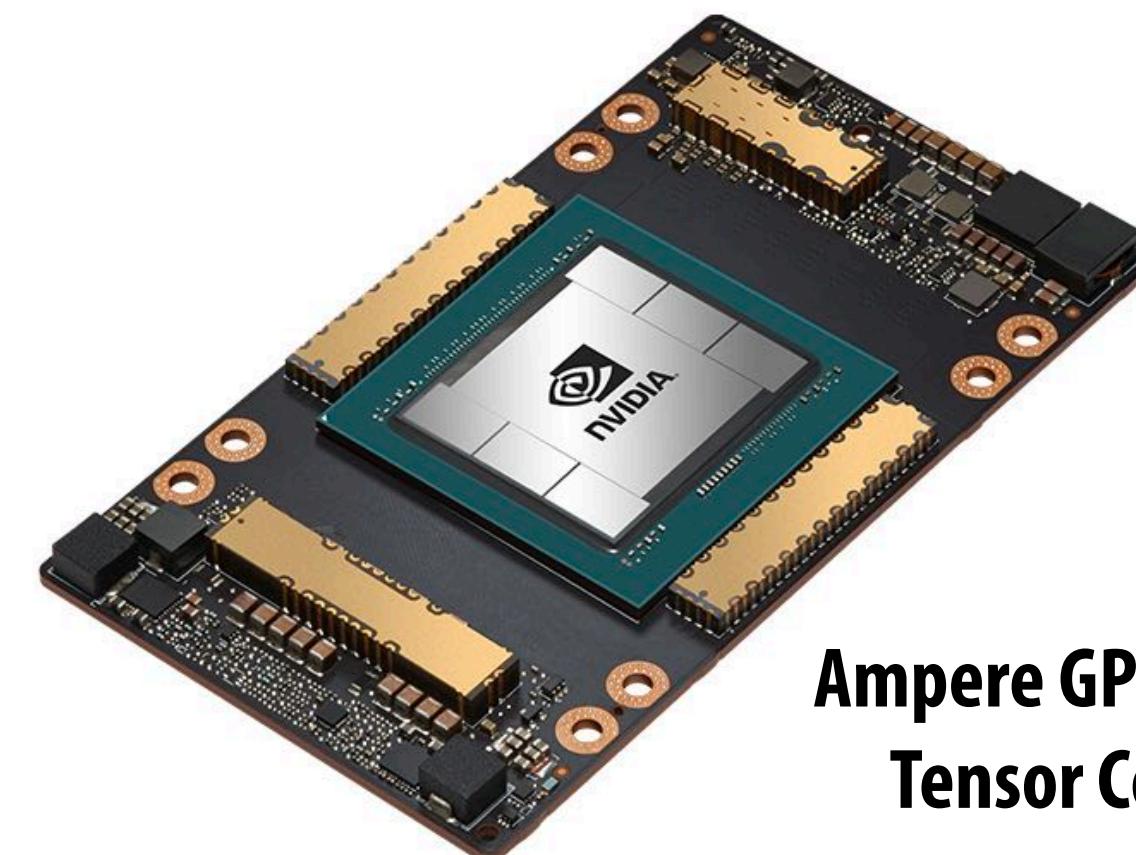
GraphCore IPU



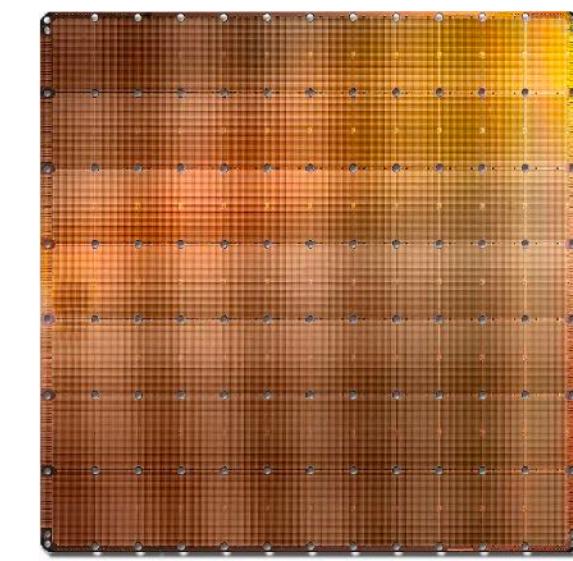
Apple Neural Engine



AWS Trainium



Ampere GPU with  
Tensor Cores



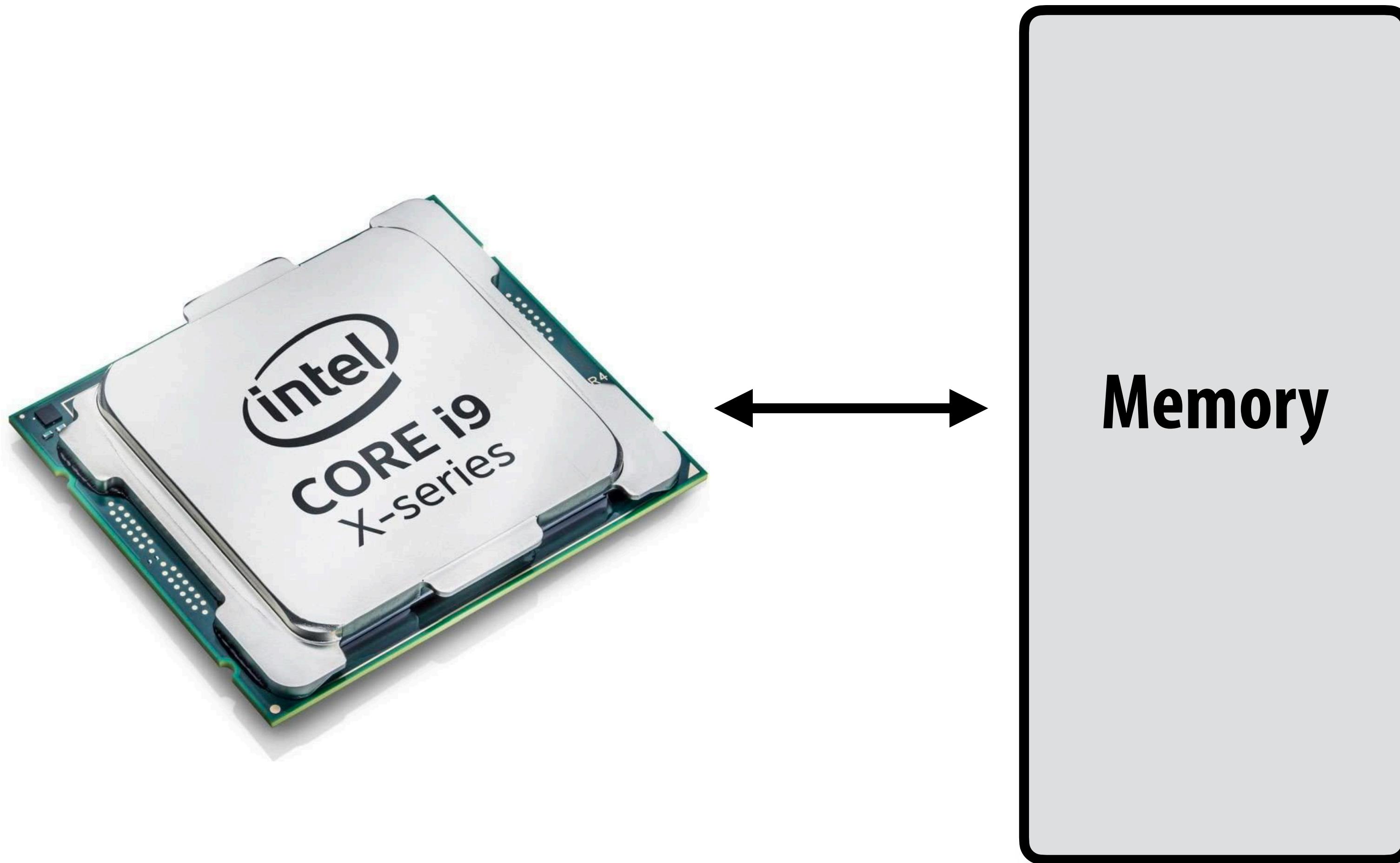
Cerebras Wafer Scale Engine



SambaNova  
Cardinal SN10

**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)

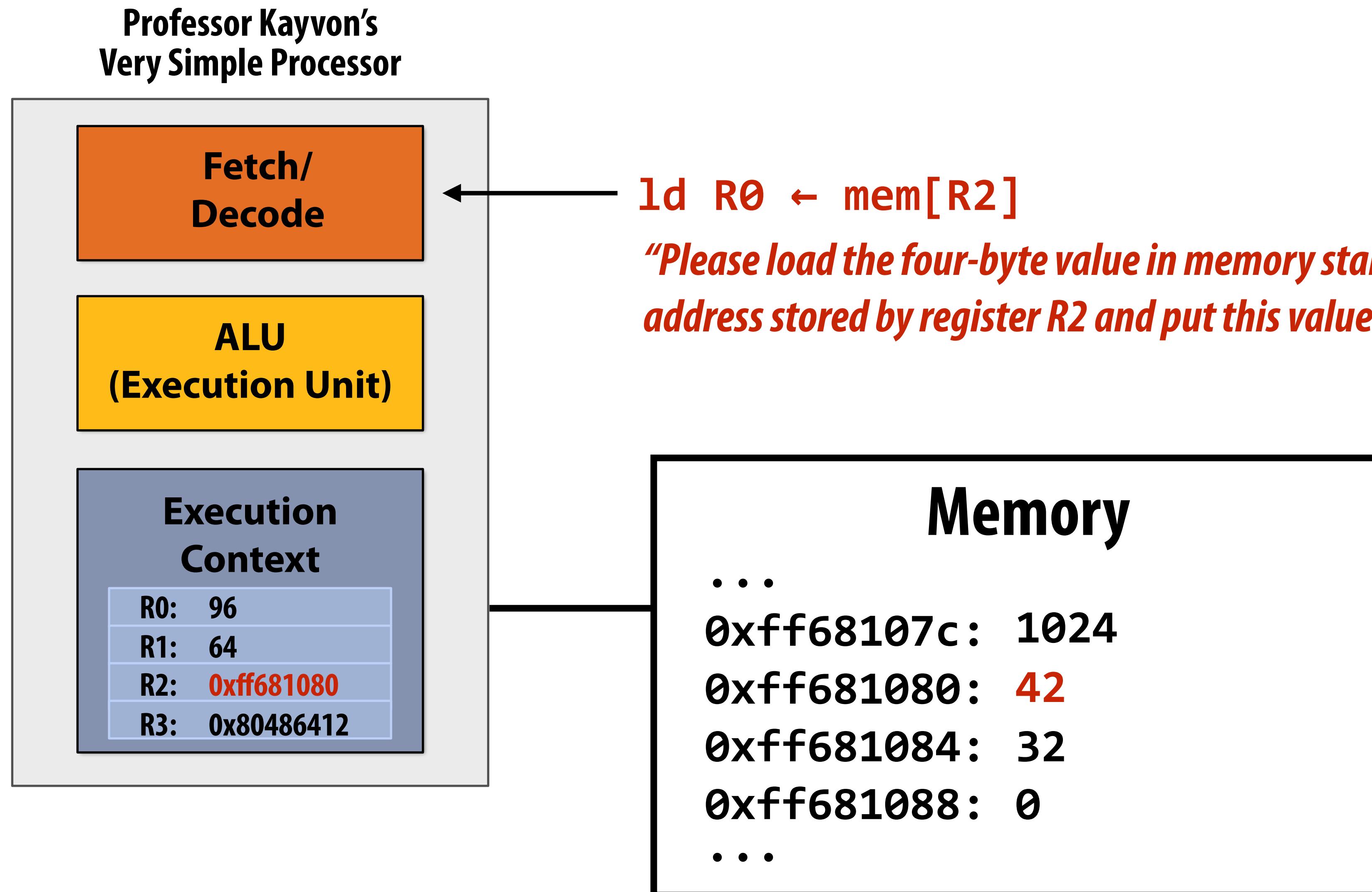
*"The byte stored at address 0x8 has the value 32."*

*"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



# 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

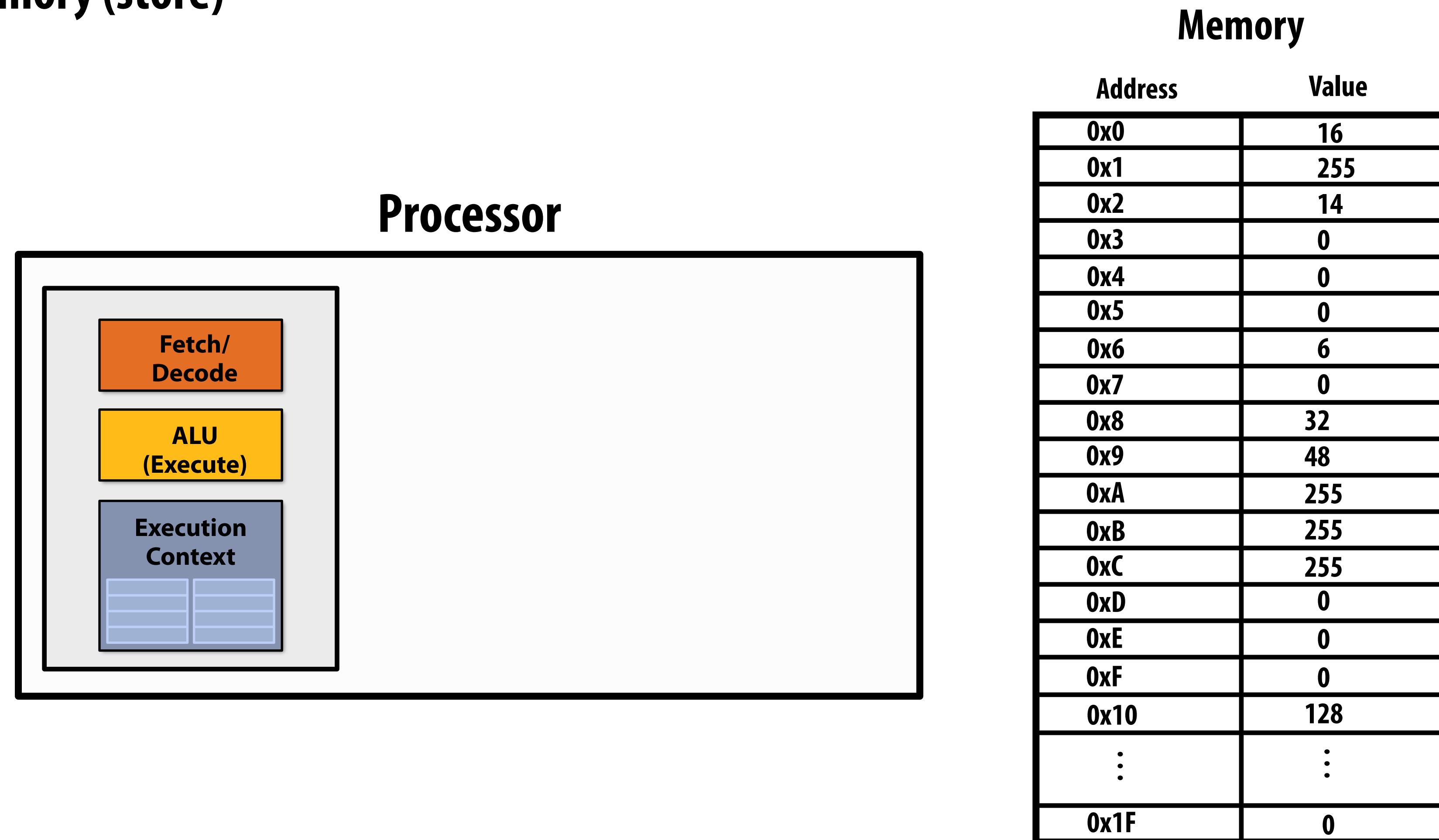
```
1d r0 mem[r2]  
1d r1 mem[r3]  
add r0, r0, r1
```

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)

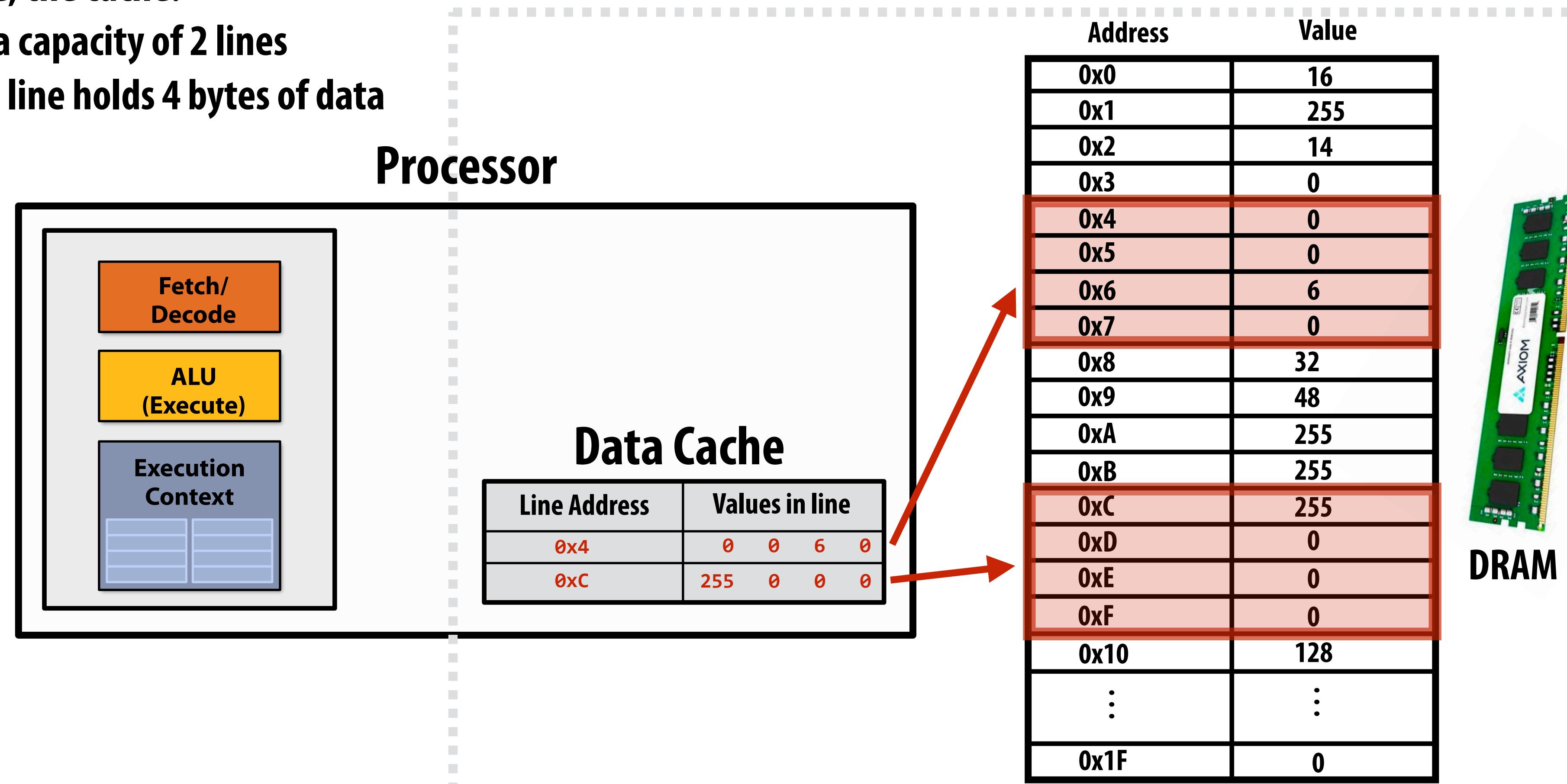


# What are caches?

- 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



# How does a processor decide what data to keep in cache?

- Outside the scope of this course, but I suggest googling these terms...
  - Direct mapped cache
  - Set-associative cache
  - Cache line
- For now, just assume that the cache of size  $N$  bytes stores values for the last  $N$  addresses accessed
  - LRU replacement policy ("least recently used") - to make room for new data, throw out the data in the cache that was accessed the longest time ago

# Cache example 1

Array of 16 bytes in memory

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
Line 0x4	0
	0
	6
	0
Line 0x8	32
	48
	255
	255
Line 0xC	255
	0
	0
	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	0x4 ●●●●
0x1	hit	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 different addresses in the same line, leading to cache hits

Temporal locality: repeated accesses to the same address result in hits.

# Cache example 2

Array of 16 bytes in memory

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
Line 0x0	0x4
	0
	0
	0x6
Line 0x4	6
	0
	0
	0x8
Line 0x8	32
	48
	255
	255
Line 0xC	255
	0
	0
	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 ●●●●

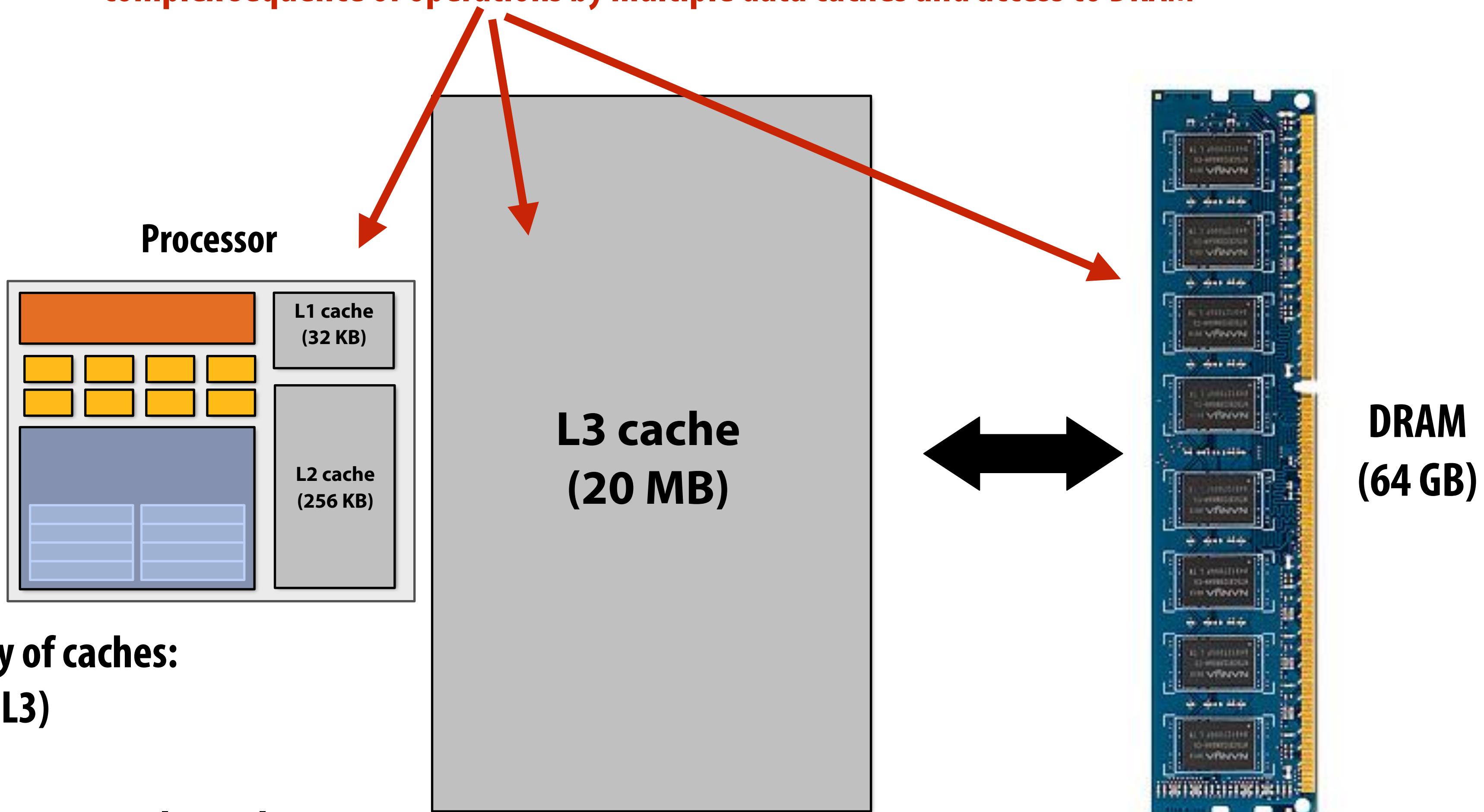
# Caches reduce length of stalls (reduce memory access latency)

- 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



Common organization: hierarchy of caches:

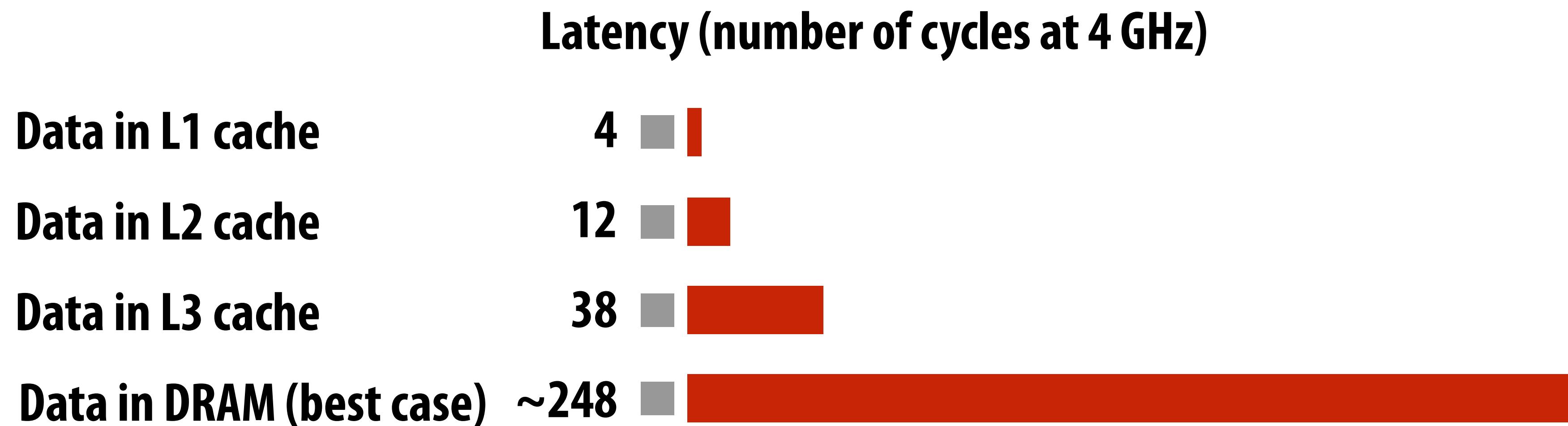
Level 1 (L1), level 2 (L2), level 3 (L3)

Smaller capacity caches near processor → lower latency

Larger capacity caches farther away → larger latency

# Data access times

(Kaby Lake CPU)



# Data movement has high energy cost

- Rule of thumb in modern system design: always seek to reduce amount of data movement in a computer
- “Ballpark” numbers
  - Integer op: ~ 1 pJ \*
  - Floating point op: ~20 pJ \*
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ
- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt  
(remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

[Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

\* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

# Summary

- **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!

- Your goal between now and Thursday: Find yourself a partner!  
(remember, we can do it for you!)



**Prof. Kayvon**



**Prof. Olukotun**