

Lecture 1:

Why Parallelism? Why Efficiency?

**Parallel Computing
Stanford CS149, Fall 2021**

Hello!



Prof. Kayvon



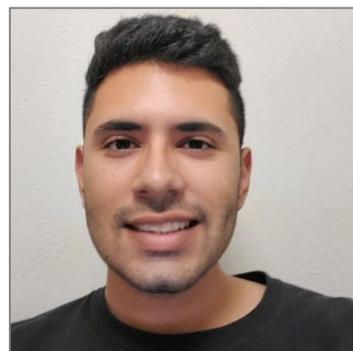
Prof. Olukotun



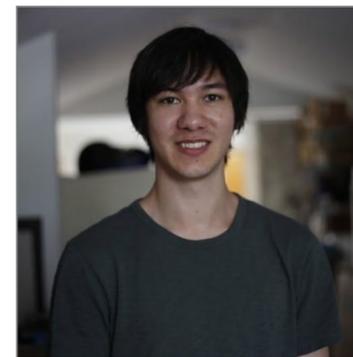
Yuhan



Olivia



Luis



Teguh



Jack

One common definition

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



We care about performance *
We care about efficiency

**We're going to use multiple
processors to get it**

* Note: different motivation from “concurrent programming” using threads like in CS110

DEMO 1

(CS149 Fall 2021'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, 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 \neq 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

- <http://cs149.stanford.edu>

■ Sign up for the course on Piazza

- <https://piazza.com/stanford/fall2021/cs149/home>

■ Fill out our partner request form

- If you want us to match you with a partner

■ Textbook

- There is no course textbook (the internet is plenty good these days), but please see course web site for suggested references

The screenshot shows the course page for Stanford CS149, Fall 2021. The page has a red header with navigation links: [Home], [Feed], [Course Info], and [Lectures/Readings]. The main title is 'PARALLEL COMPUTING'. Below the title is a paragraph describing the course's focus on parallel processing in modern computing. A 'Basic Info' section lists the course schedule (Tues/Thurs 3:15-4:45pm), virtual lectures, and instructors Kayvon Fatahalian and Kunle Olukotun. A 'Fall 2021 Schedule' section lists 12 topics with their dates and brief descriptions.

Stanford CS149, Fall 2021

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

Tues/Thurs 3:15-4:45pm
All lectures are virtual
Instructors: **Kayvon Fatahalian** and **Kunle Olukotun**

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

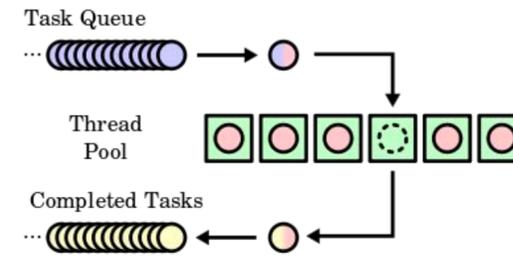
Fall 2021 Schedule

Sep 21	Why Parallelism? Why Efficiency? Challenges of parallelizing code, motivations for parallel chips, processor basics
Sep 23	A Modern Multi-Core Processor Forms of parallelism: multicore, SIMD, threading + understanding latency and bandwidth
Sep 28	Parallel Programming Abstractions Ways of thinking about parallel programs, and their corresponding hardware implementations, ISPC programming
Sep 30	Parallel Programming Basics Thought process of parallelizing a program in data parallel and shared address space models
Oct 05	Performance Optimization I: Work Distribution and Scheduling Achieving good work distribution while minimizing overhead, scheduling Cilk programs with work stealing
Oct 07	Performance Optimization II: Locality, Communication, and Contention Message passing, async vs. blocking sends/receives, pipelining, increasing arithmetic intensity, avoiding contention
Oct 12	GPU architecture and CUDA Programming CUDA programming abstractions, and how they are implemented on modern GPUs
Oct 14	Data-Parallel Thinking Data-parallel operations like map, reduce, scan, prefix sum, groupByKey
Oct 19	Distributed Computing Using Spark Producer-consumer locality, RDD abstraction, Spark implementation and scheduling
Oct 21	Cache Coherence Definition of memory coherence, invalidation-based coherence using MSI and MESI, false sharing
Oct 26	Memory Consistency + Implementation Synchronization Consistency vs. coherence, relaxed consistency models and their motivation, acquire/release semantics, implementing locks and atomic operations

Four programming assignments



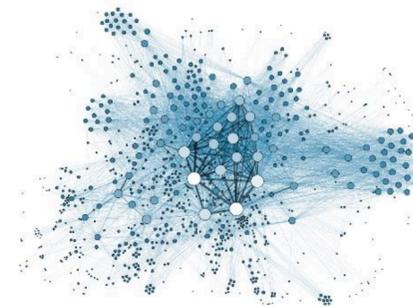
**Assignment 1: ISPC programming
on multi-core CPUs**



**Assignment 2:
scheduler for a task graph**



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



**Assignment 4: parallel
large graph algorithms
on a multi-core CPU**



**Optional assignment 5:
(will boost some prior grade)**

Plus a few optional extra credit challenges... ;-)

Written assignments

- **Approximately every two-weeks we will have a take-home written assignment**
- **Written assignments contain modified versions of previous exam questions, so consider them practice for the exam**
- **Graded on a credit/no credit basis**

Commenting and contributing to lectures

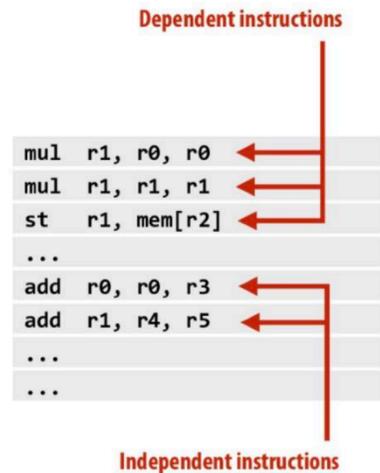
Why Parallelism? Why Efficiency?

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 appear to be executed in program order. BUT independent instructions can be executed simultaneously by a processor without impacting program correctness
- Superscalar execution: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel



Stanford CS149, Winter 2019

[Previous](#) | [Next](#) --- Slide 30 of 48

[Back to Lecture Thumbnails](#)

The website supports commenting on a per-slide basis



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 well-thought-out comment per lecture
 - Only two comments per week
 - No precise deadline, but getting them submitted “in the same week” as the lectures is the spirit of the participation
- Why do we 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, this is your participation 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**
 - “@segfault21, 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**
 - “@funkysenior21’s point is that the overhead of communication...”

Grades

48% Programming assignments (4)

15% Written assignments (5)

16% Midterm assessments(2)

- Oct 14th and Nov 11th

16% Final exam

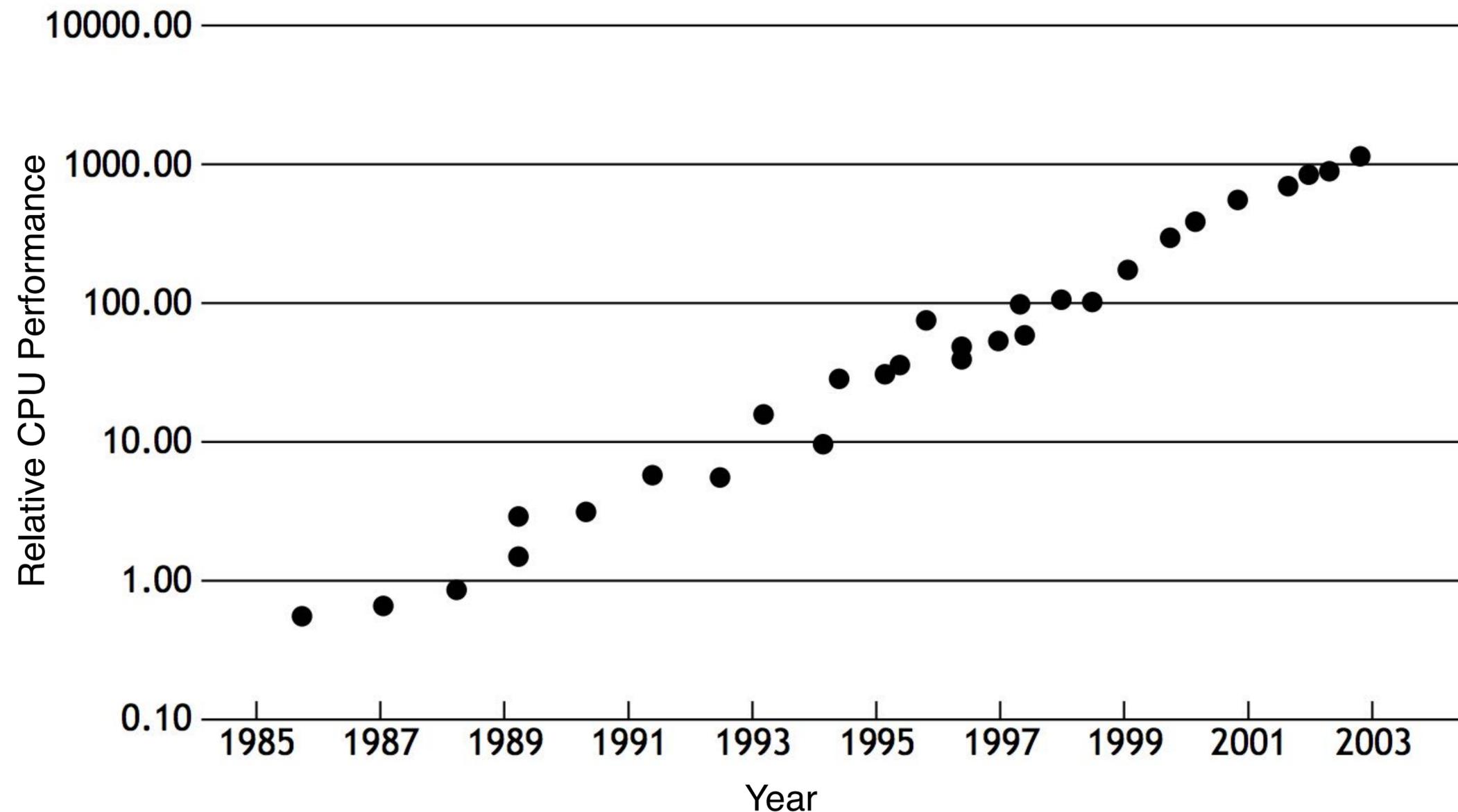
5% Asynchronous participation (comments)

Reminder: we can match you with a partner! See Piazza for our partner request form!

Why parallelism?

Some historical context: why not 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 ~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?

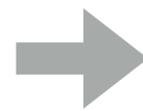
Here is a program written in C

```
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;  
}
```



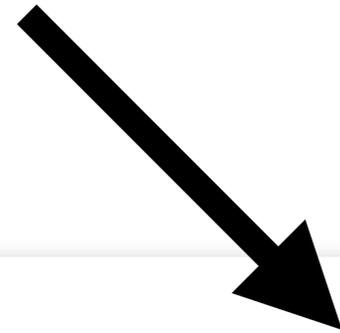
Compile
code



```
_main:  
10000f10:    pushq   %rbp  
10000f11:    movq   %rsp, %rbp  
10000f14:    subq   $32, %rsp  
10000f18:    movl   $0, -4(%rbp)  
10000f1f:    movl   %edi, -8(%rbp)  
10000f22:    movq   %rsi, -16(%rbp)  
10000f26:    movl   $1, -20(%rbp)  
10000f2d:    movl   $0, -24(%rbp)  
10000f34:    cmpl   $10, -24(%rbp)  
10000f38:    jge   23 <_main+0x45>  
10000f3e:    movl   -20(%rbp), %eax  
10000f41:    addl   -20(%rbp), %eax  
10000f44:    movl   %eax, -20(%rbp)  
10000f47:    movl   -24(%rbp), %eax  
10000f4a:    addl   $1, %eax  
10000f4d:    movl   %eax, -24(%rbp)  
10000f50:    jmp   -33 <_main+0x24>  
10000f55:    leaq   58(%rip), %rdi  
10000f5c:    movl   -20(%rbp), %esi  
10000f5f:    movb   $0, %al  
10000f61:    callq  14  
10000f66:    xorl   %esi, %esi  
10000f68:    movl   %eax, -28(%rbp)  
10000f6b:    movl   %esi, %eax  
10000f6d:    addq   $32, %rsp  
10000f71:    popq   %rbp  
10000f72:    rets
```

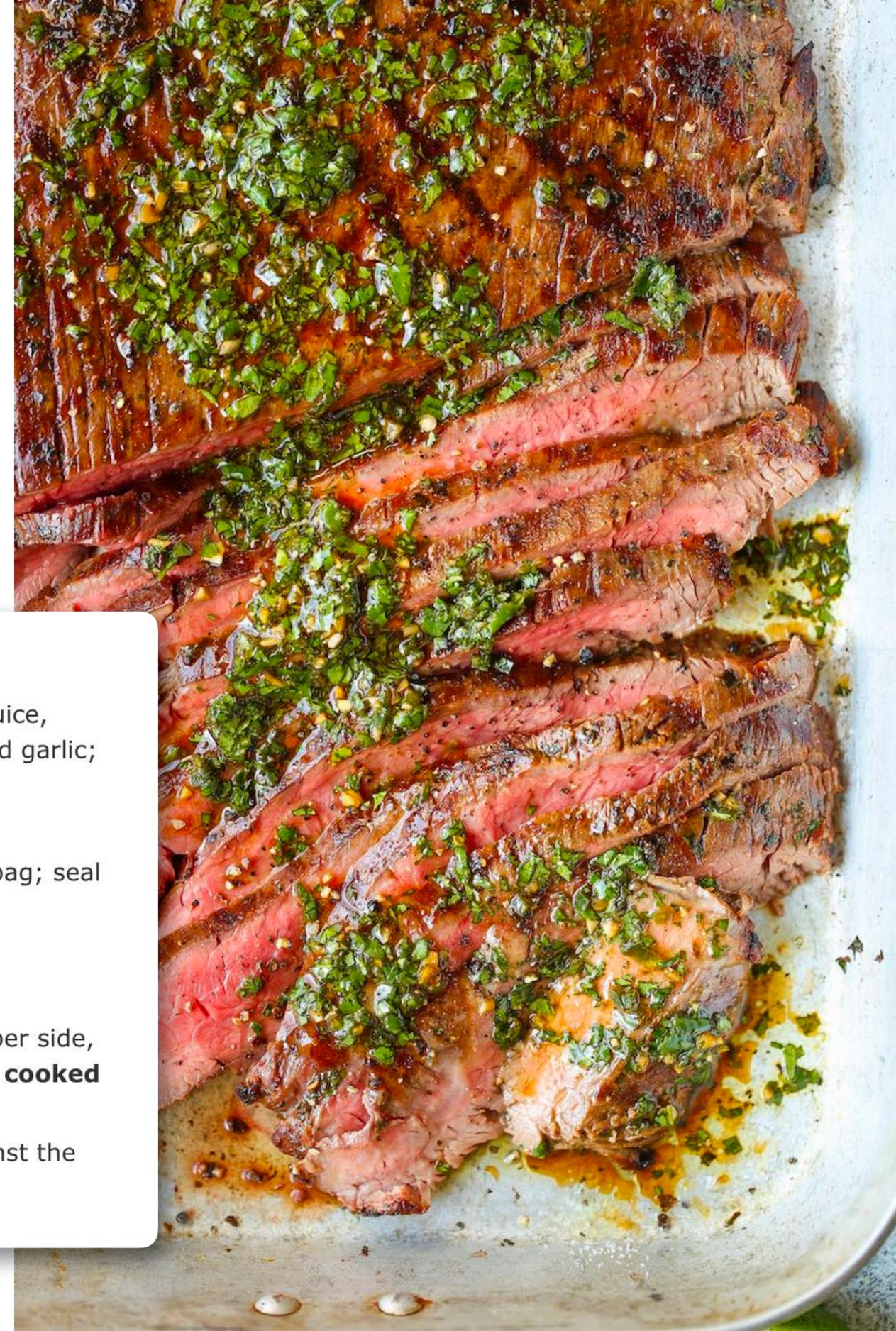
Kind of like the instructions in a recipe for your favorite meals

Mmm, carne asada



Instructions

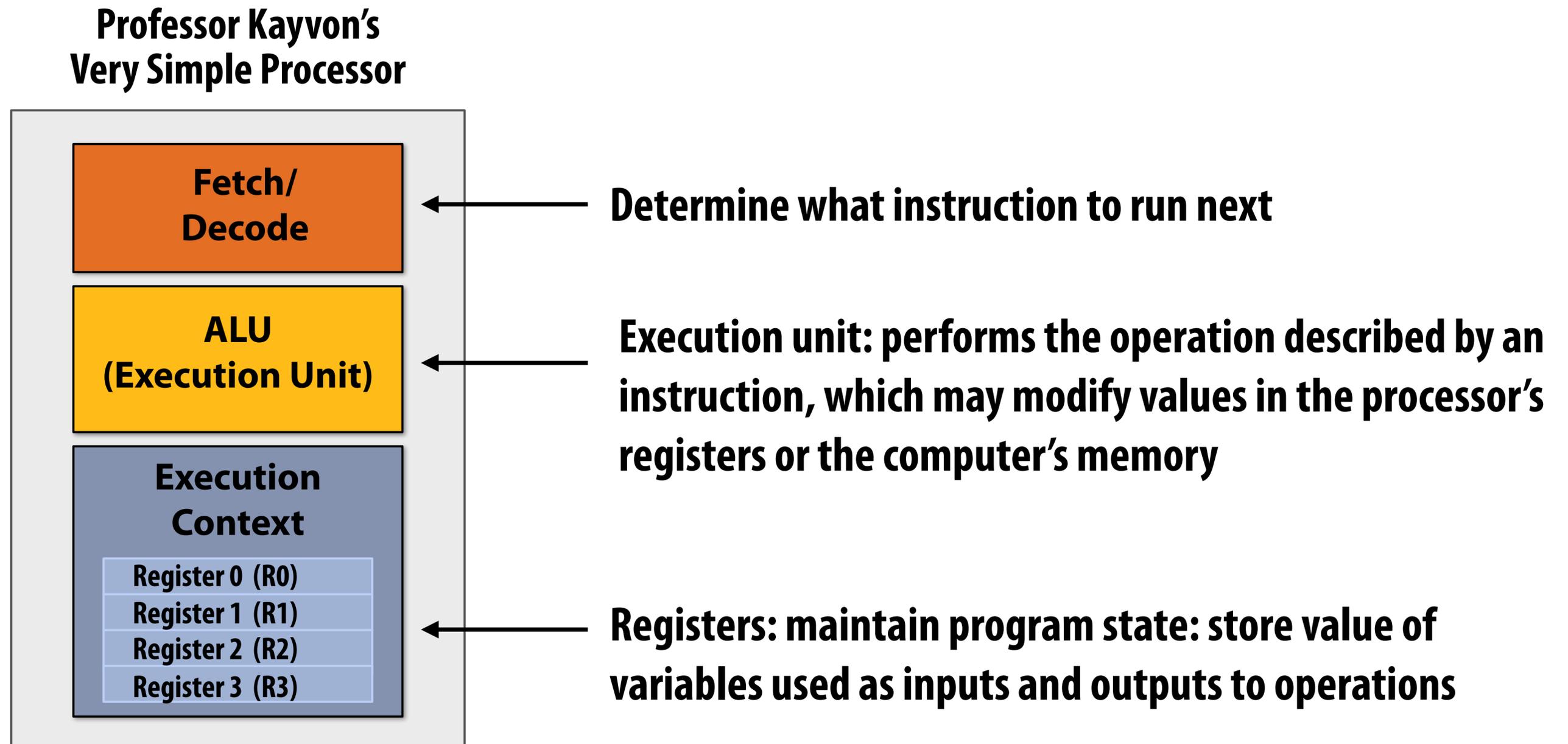
1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeno, and garlic; whisk until well combined.
2. Reserve $\frac{1}{3}$ cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and steak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

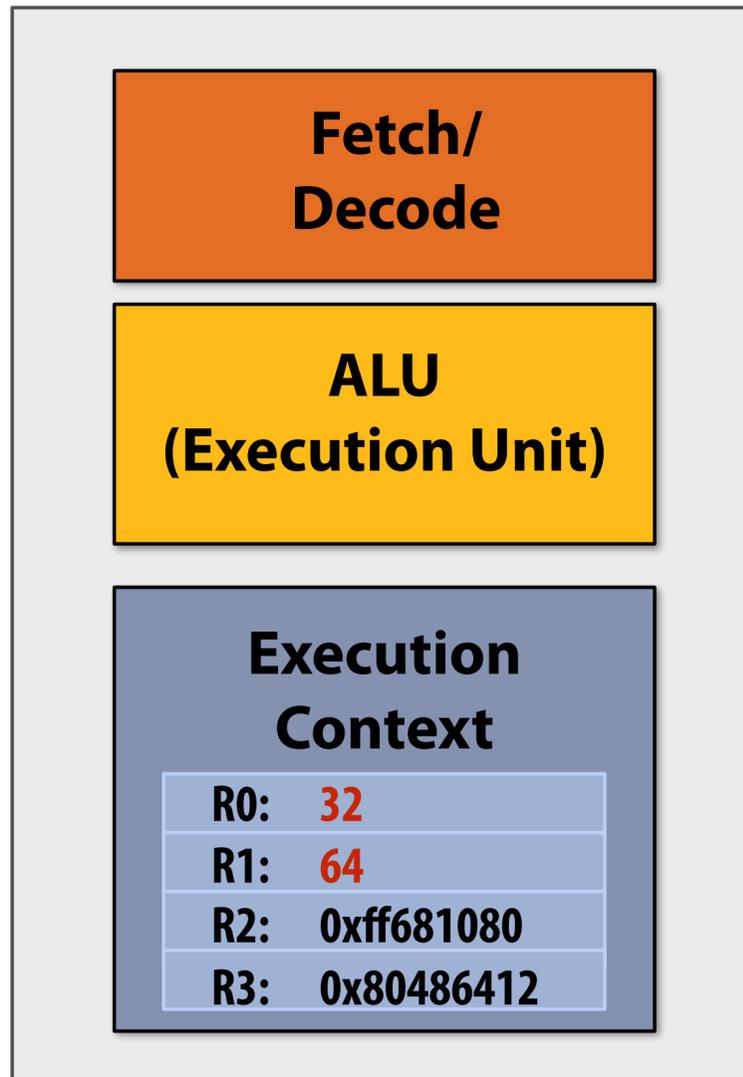


A processor executes instructions



One example instruction: add two numbers

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**

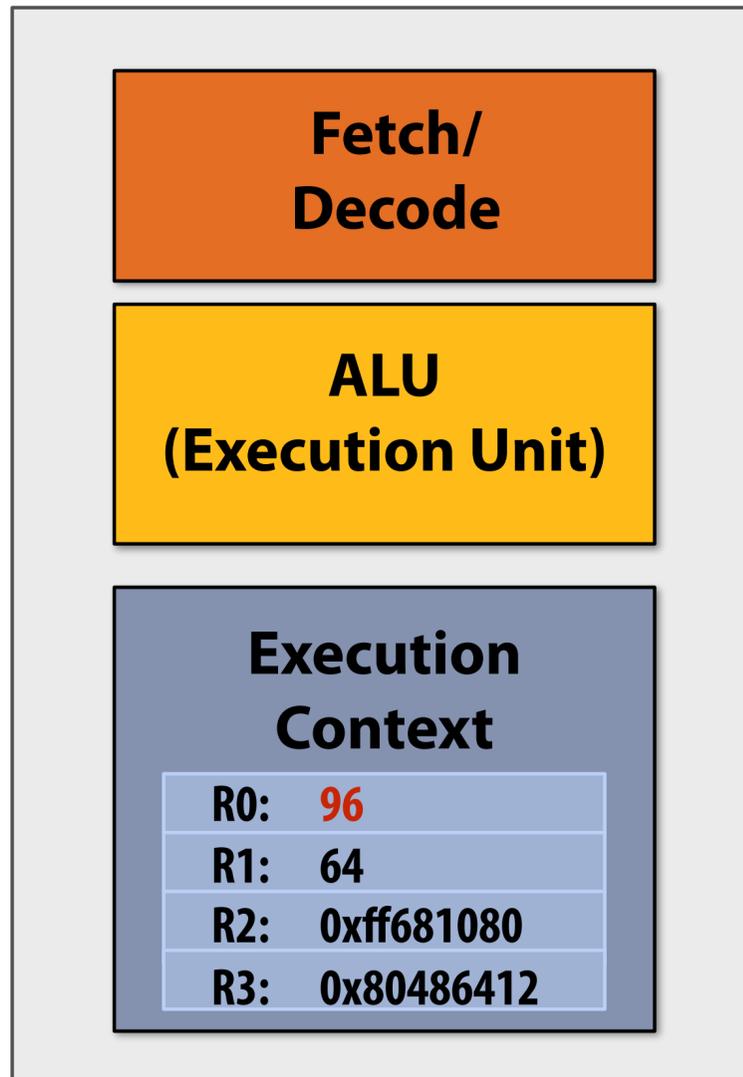
Step 3:

Perform addition operation:

Execution unit performs arithmetic, the result is: **96**

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:

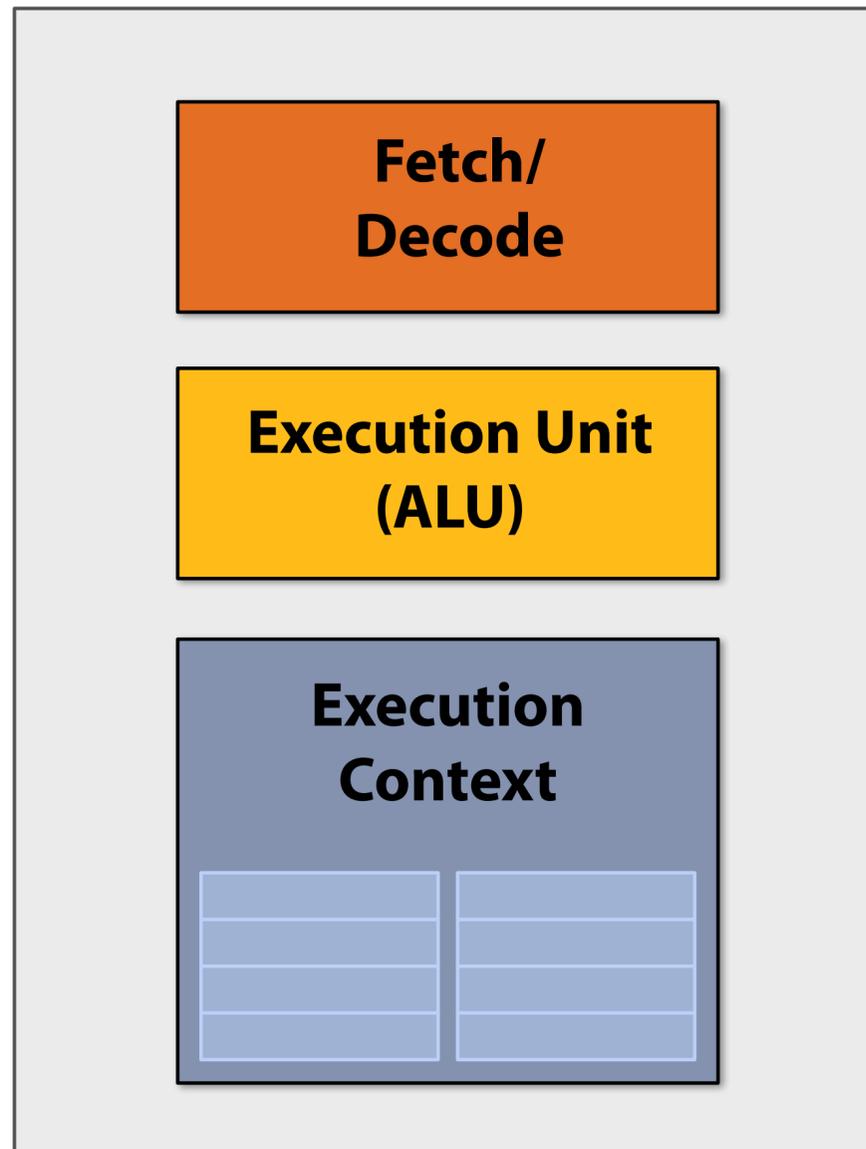
Execution unit performs arithmetic, the result is: **96**

Step 4:

Store result **96** back to register R0

Execute program

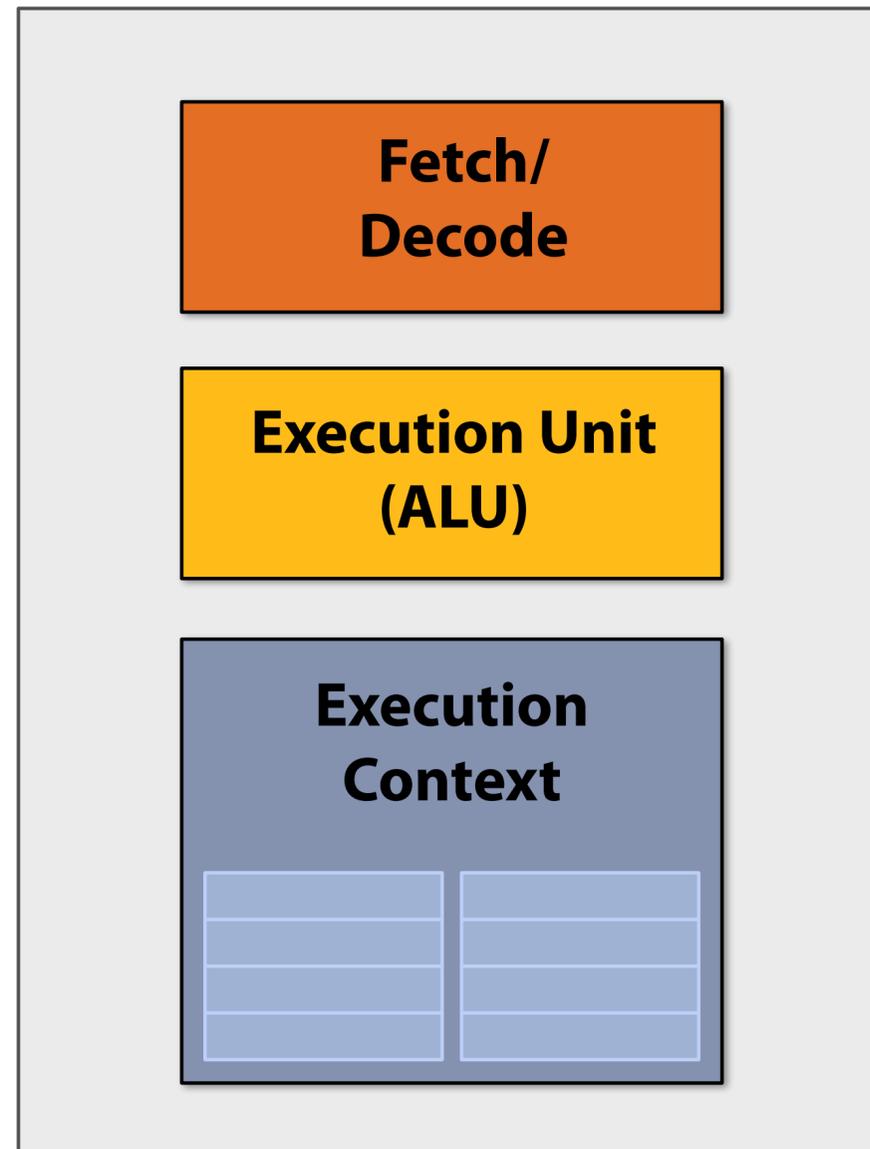
My very simple processor: executes one instruction per clock



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

Execute program

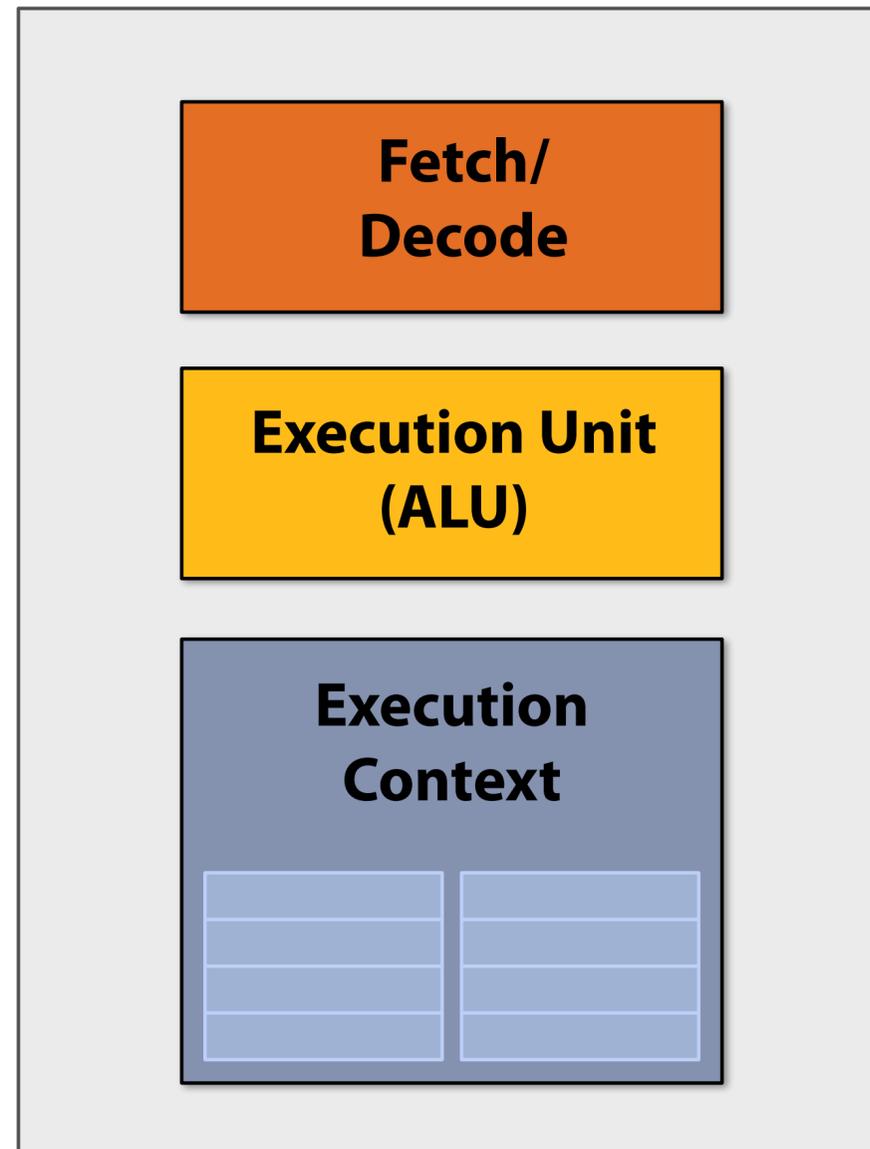
My very simple processor: executes one instruction per clock



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

Execute program

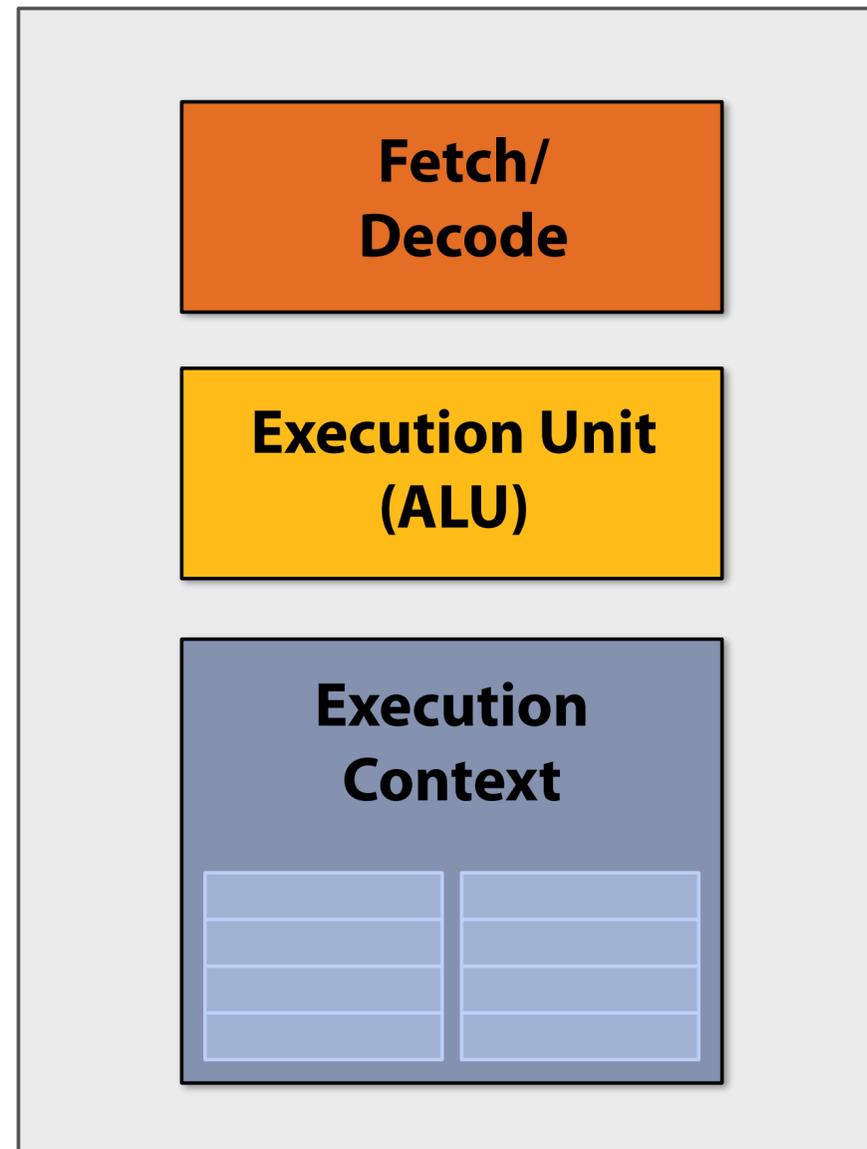
My very simple processor: executes one instruction per clock



```
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

```
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'

time



1

2

3

4

5

Volunteer 1

Volunteer 2

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'*

time



1

2

3

4

5

Volunteer 1

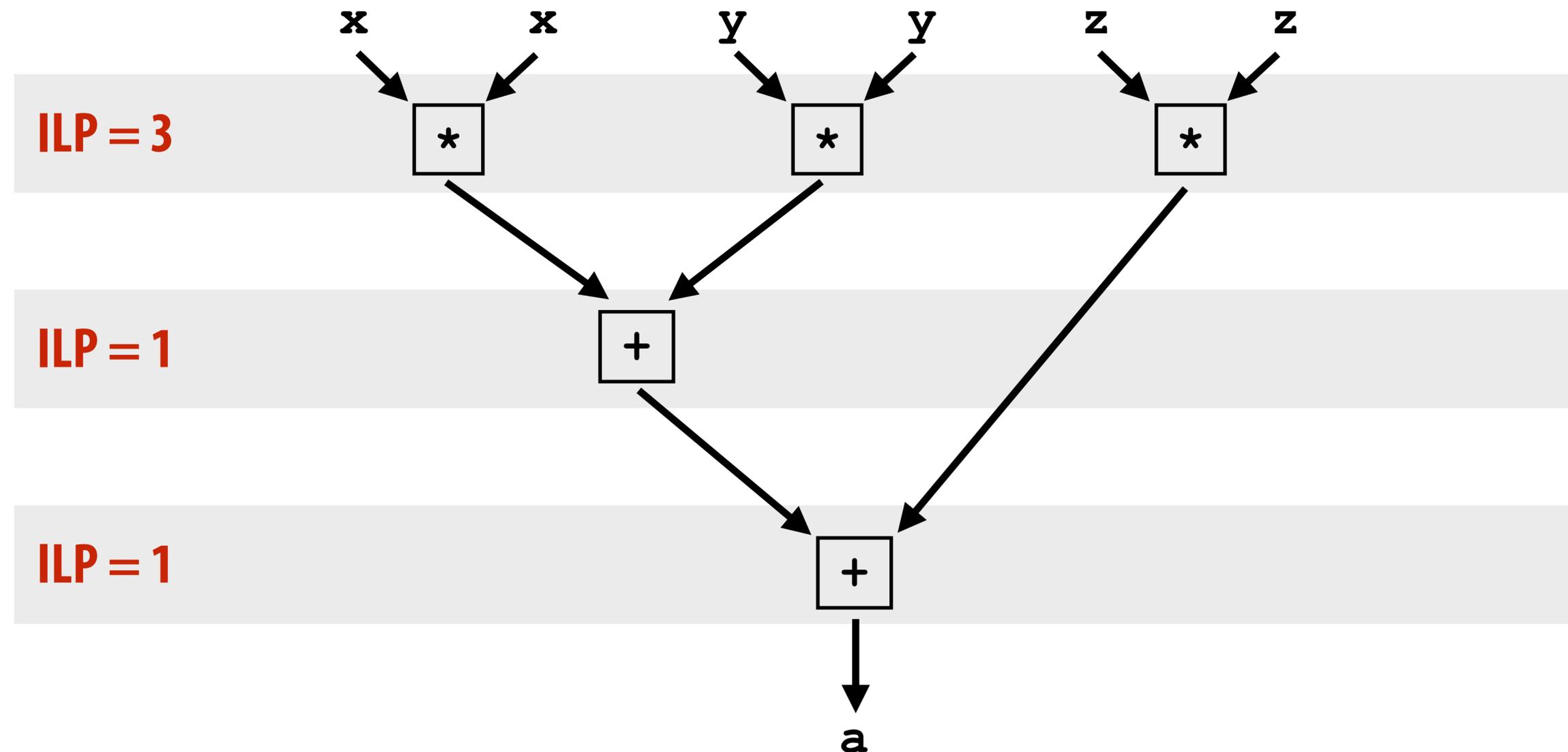
Volunteer 2

Volunteer 3

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)**

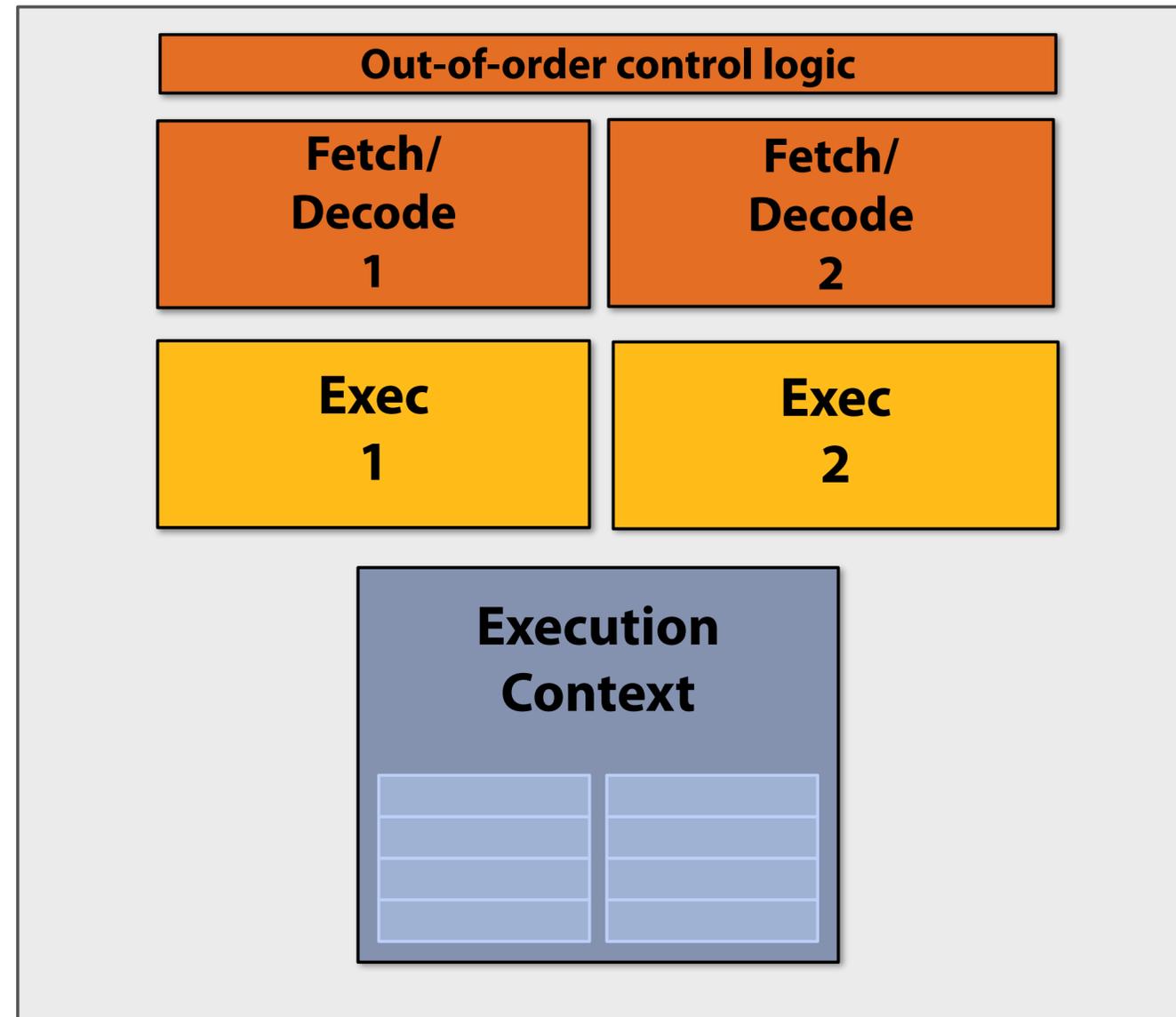
But instruction 4 must be executed after instructions 1 and 2

And instruction 5 must be executed after instruction 4

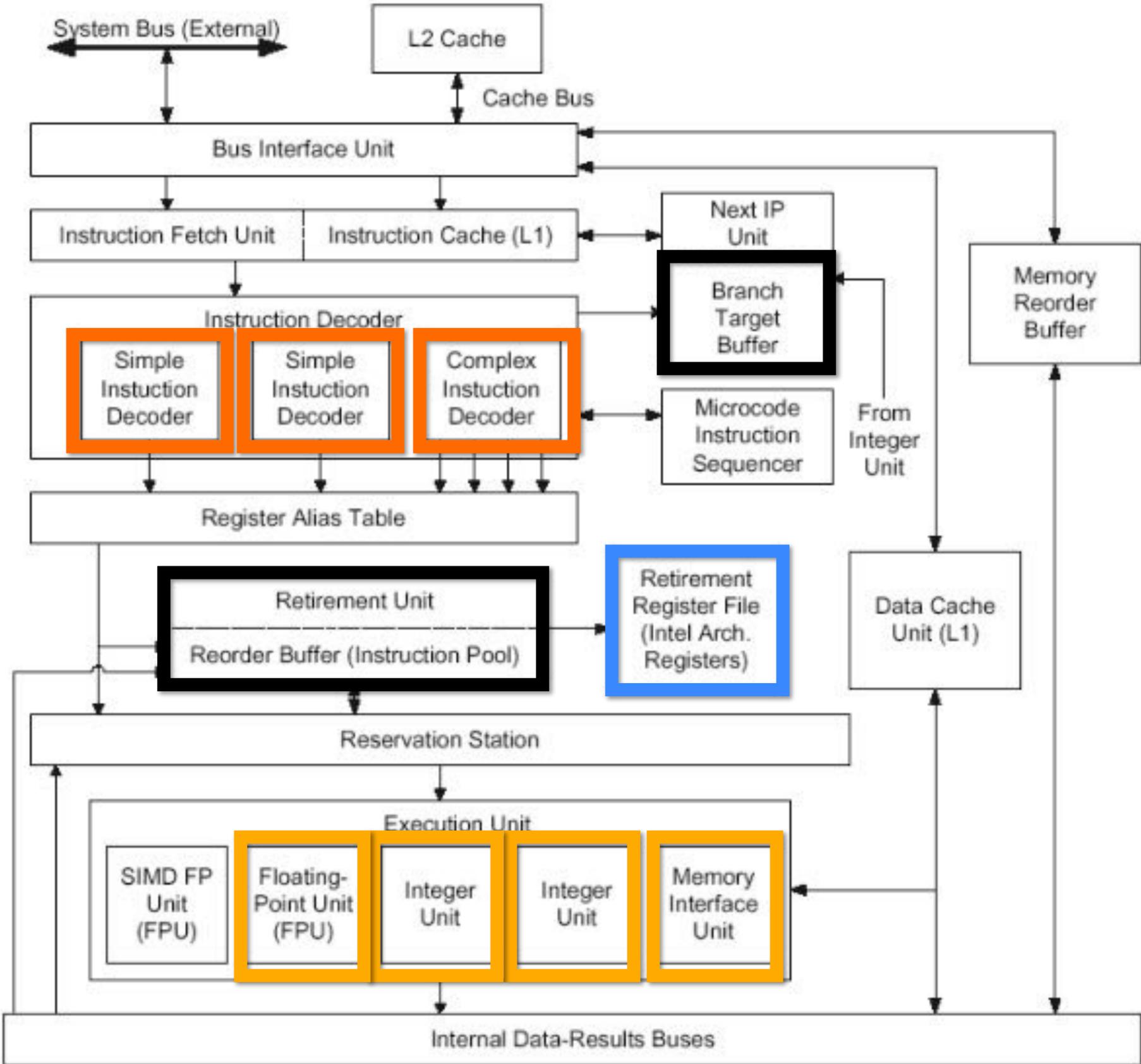
* Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.

Superscalar processor

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



Aside: Pentium 4



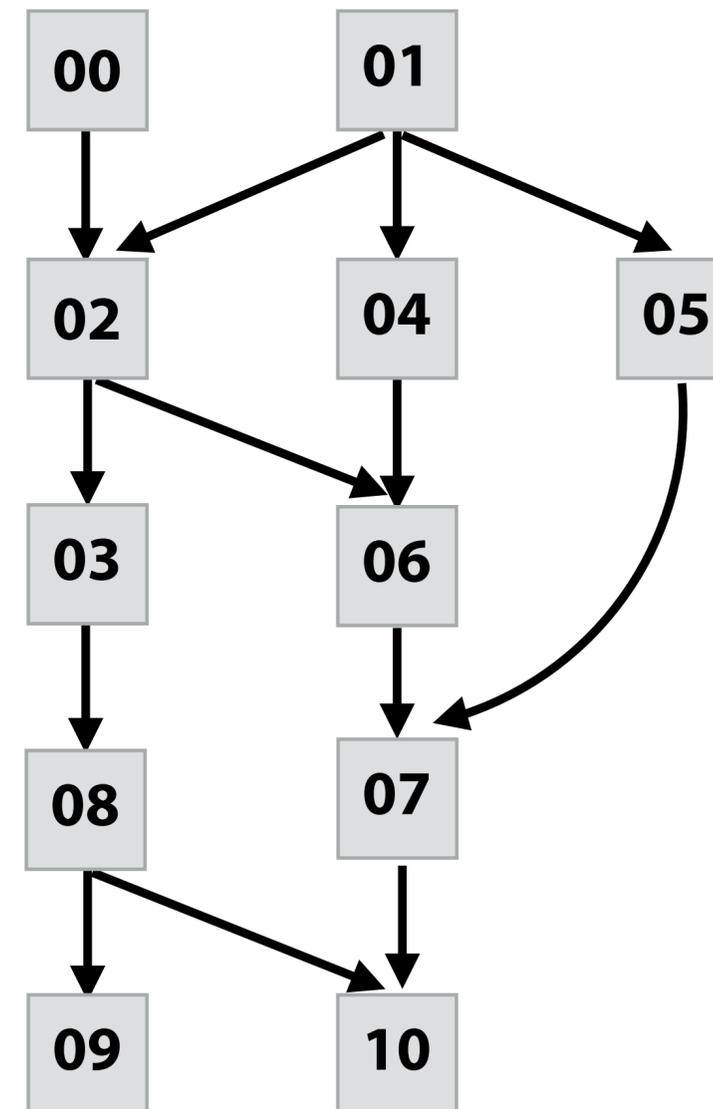
A more complex example

Program (sequence of instructions)

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	

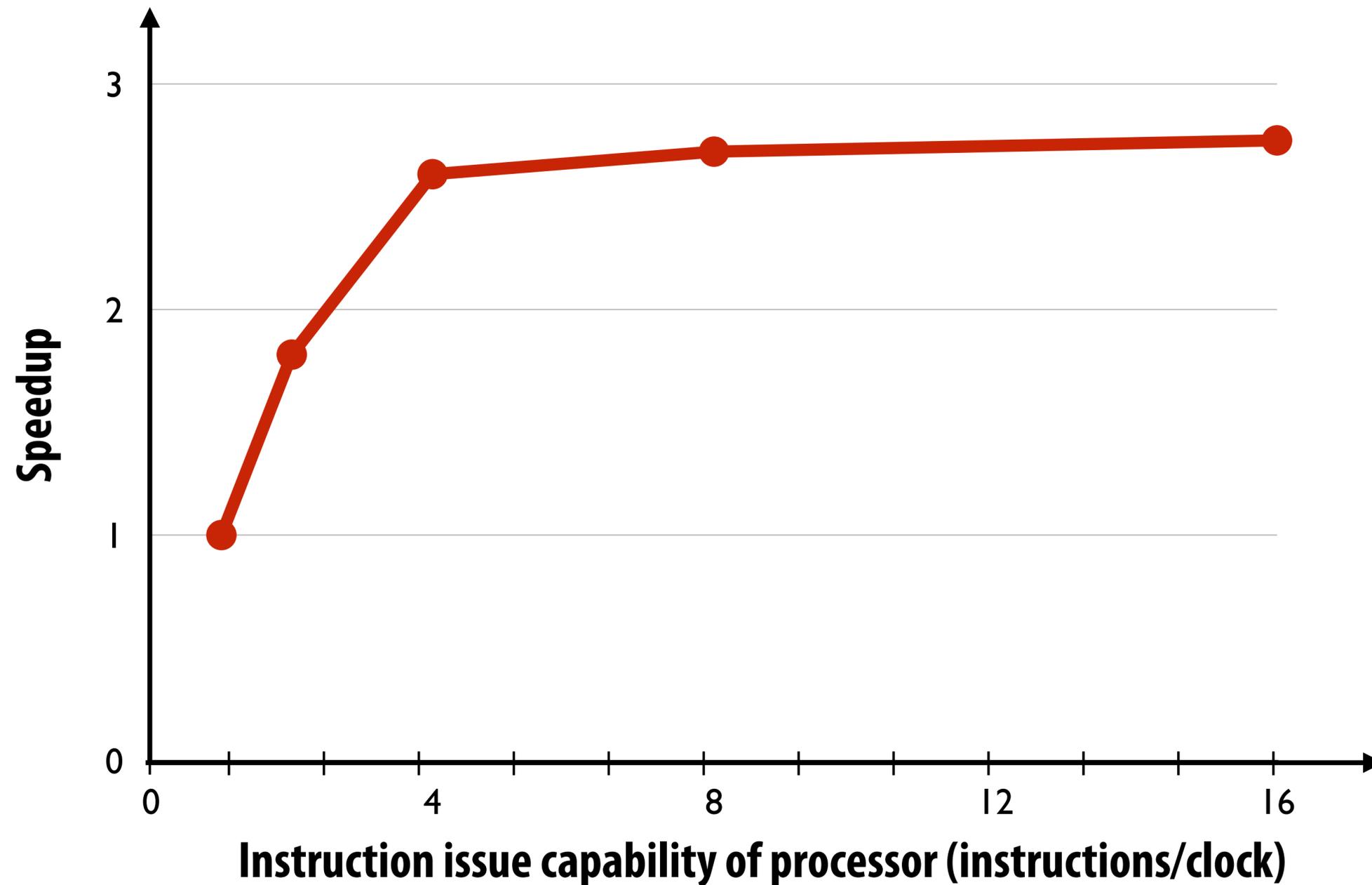
value during execution

Instruction dependency graph

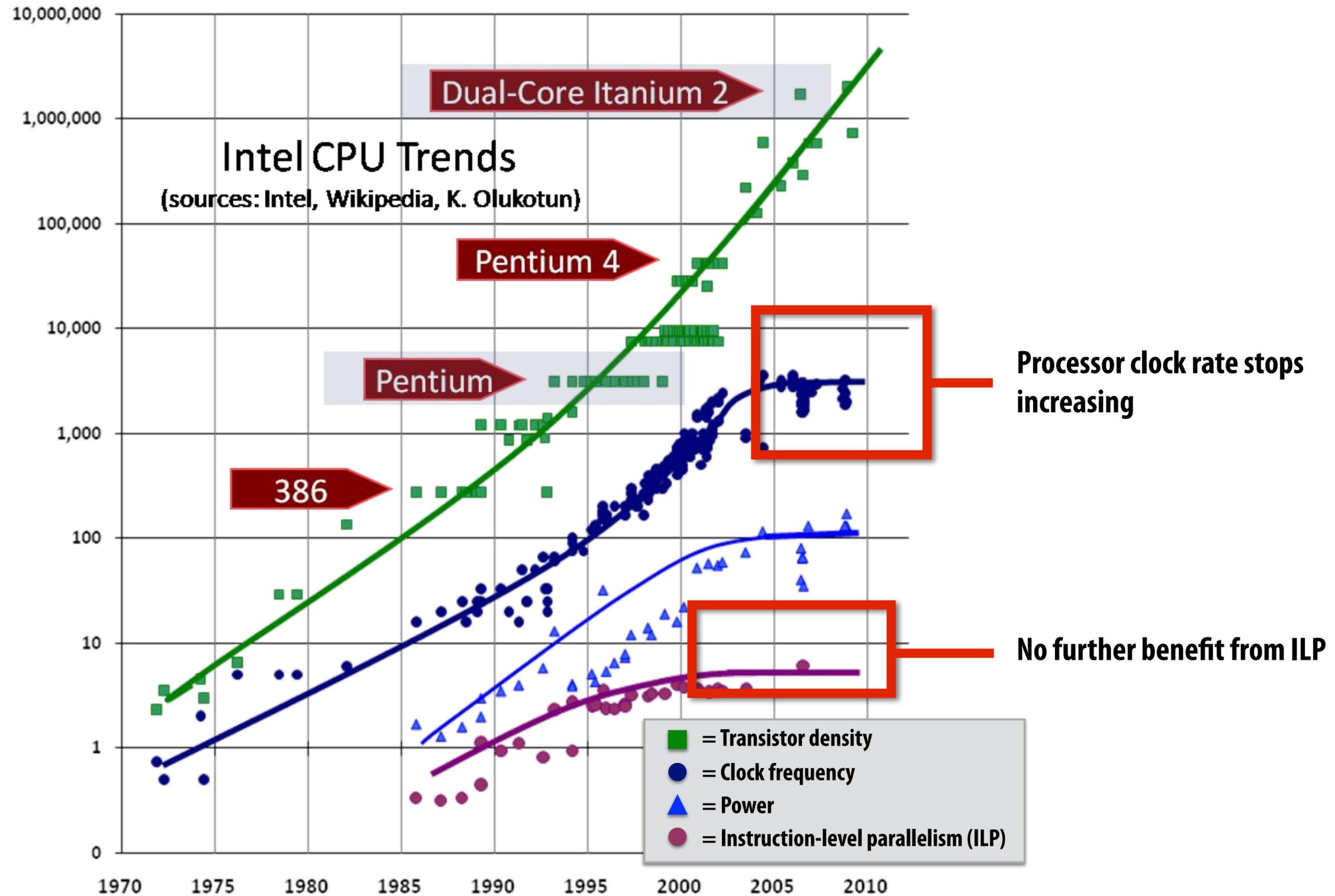


Diminishing returns of superscalar execution

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)



ILP tapped out + end of frequency scaling



The “power wall”

Power consumed by a transistor:

Dynamic power \propto capacitive load \times voltage² \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

	<u>TDP</u>
Apple M1 laptop:	13W
Intel Core i9 10900K (in desktop CPU):	95W
NVIDIA RTX 3080 GPU	320W
Mobile phone processor	1/2 - 2W
World's fastest supercomputer	megawatts
Standard microwave oven	700W

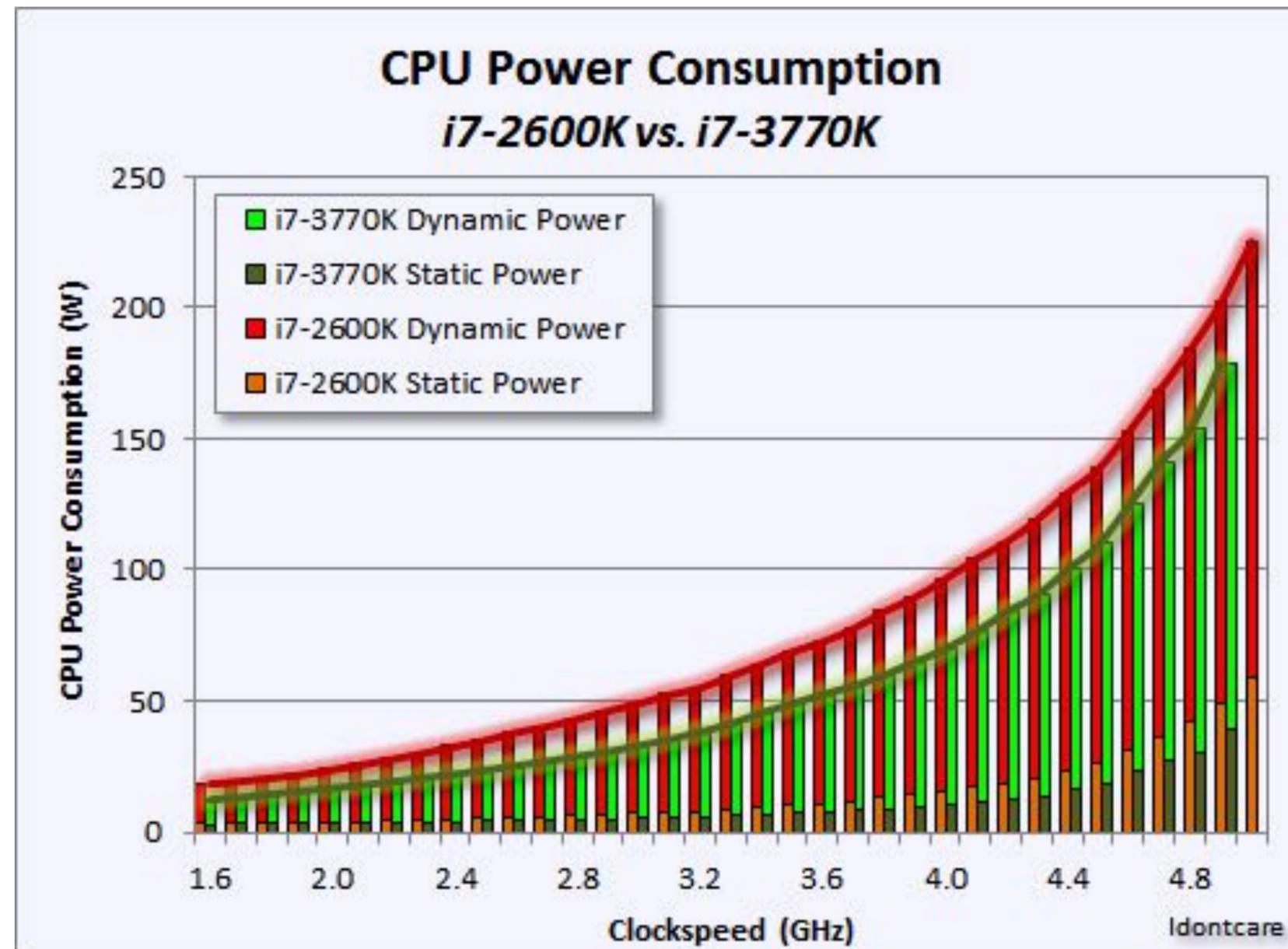


Power draw as a function of clock frequency

Dynamic power \propto capacitive load \times voltage² \times frequency

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

Maximum allowed frequency determined by processor's core voltage



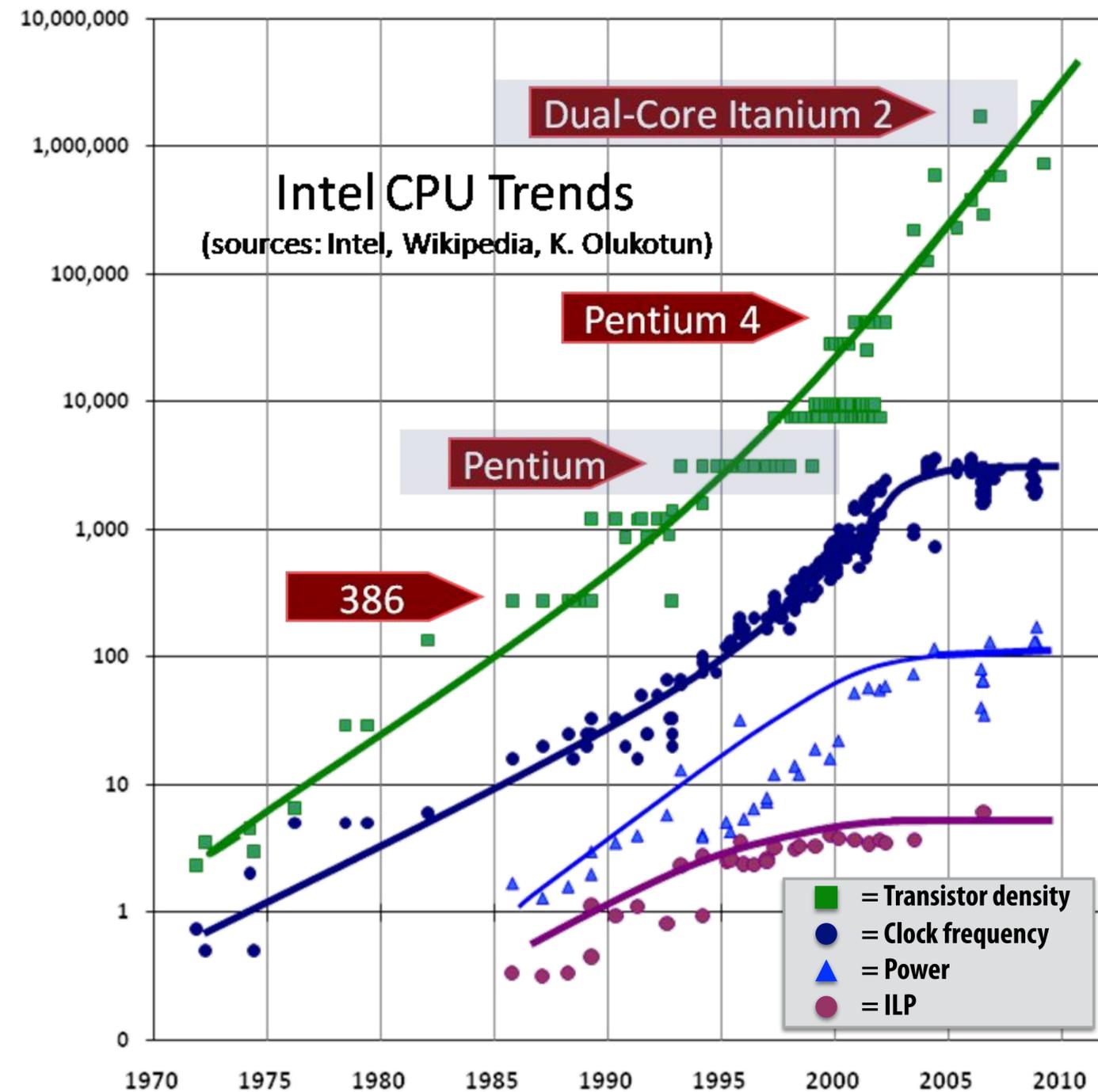
Single-core performance scaling

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

1. Frequency scaling limited by power
2. ILP scaling tapped out

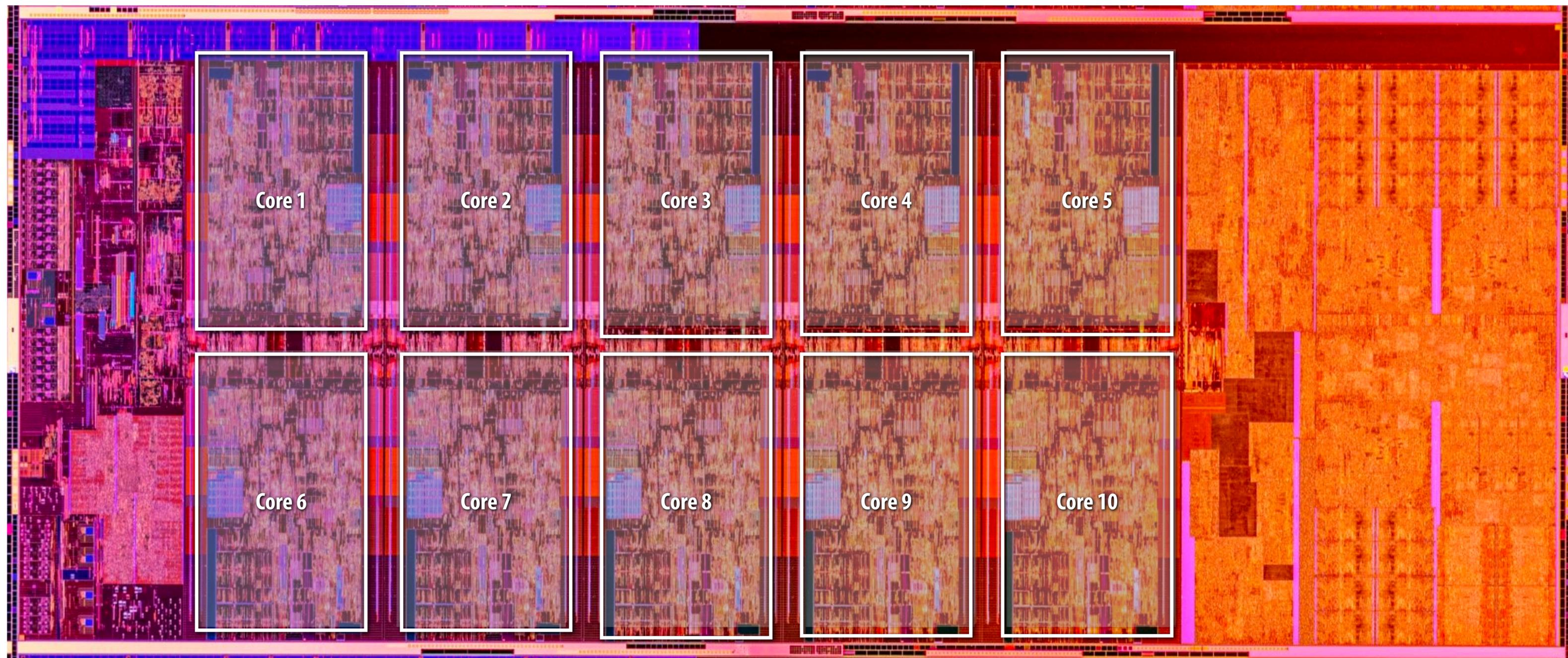
Architects are now building faster processors by adding more execution units that run in parallel (Or units that are specialized for a specific task (like graphics, or audio/video playback))

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

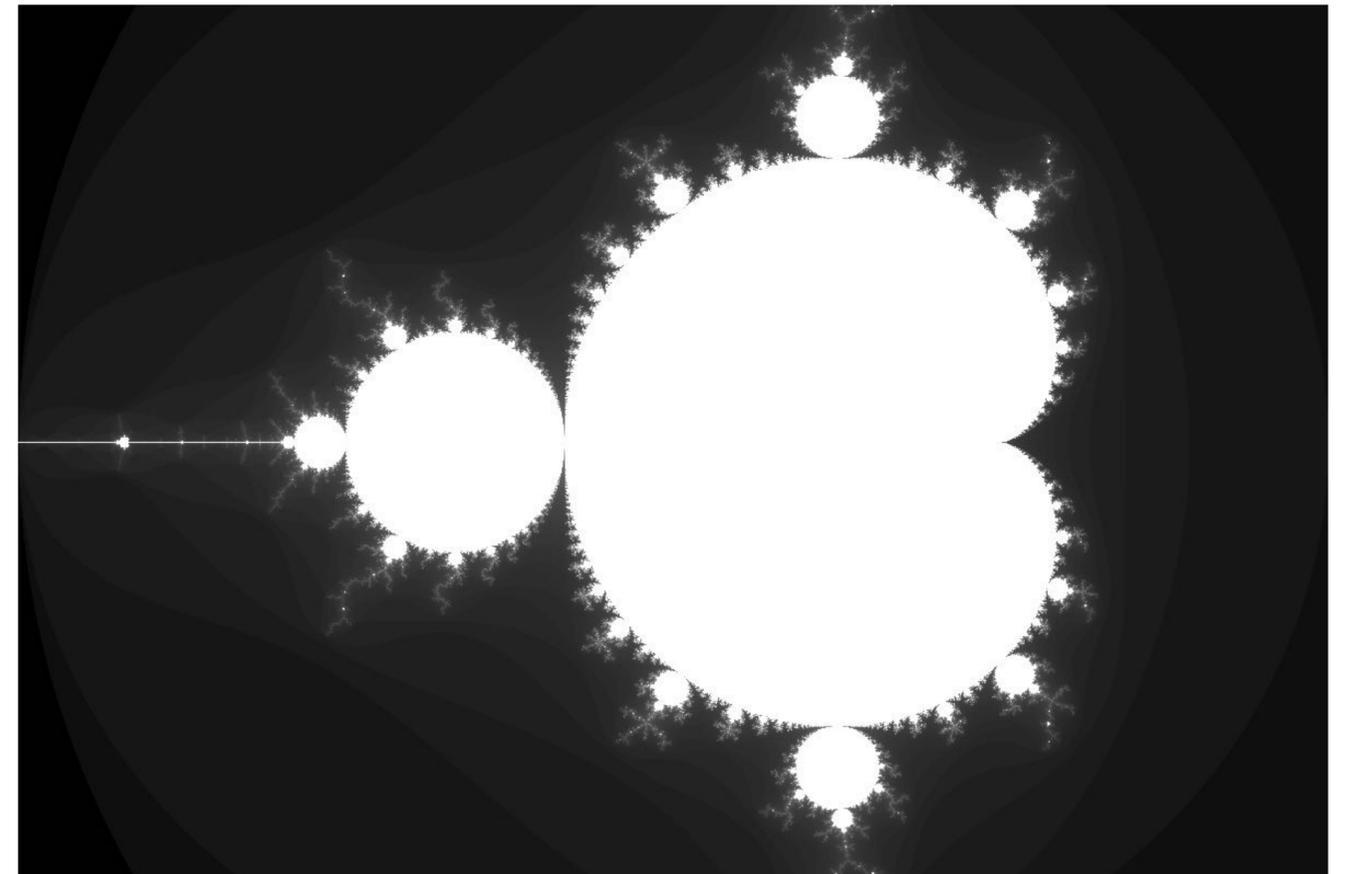
- How to write code that efficiently uses the resources in a modern multi-core CPU

- 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...

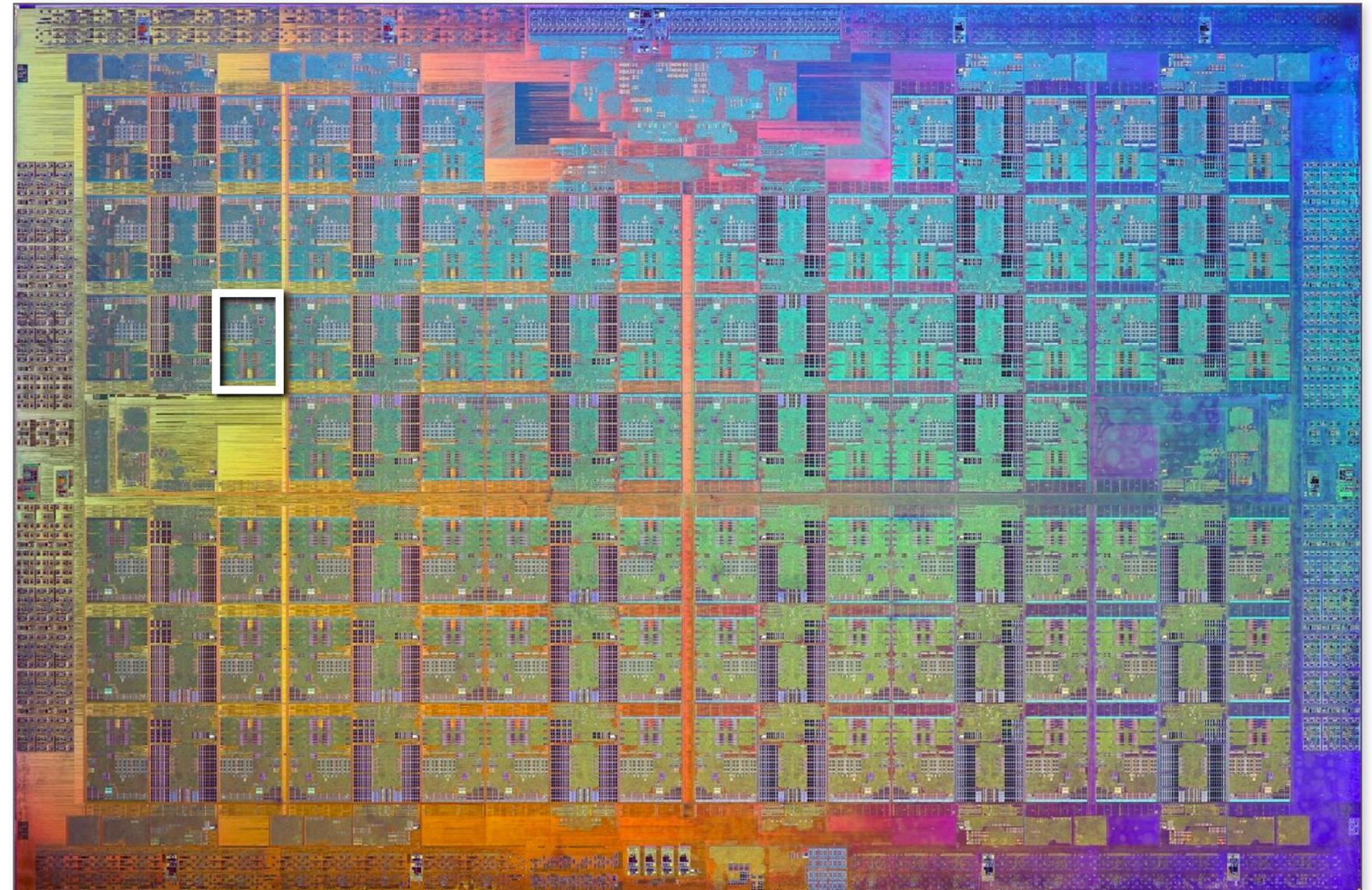
We'll talk about these terms next time!

~32-40x faster!



Intel Xeon Phi 7290 (2016)

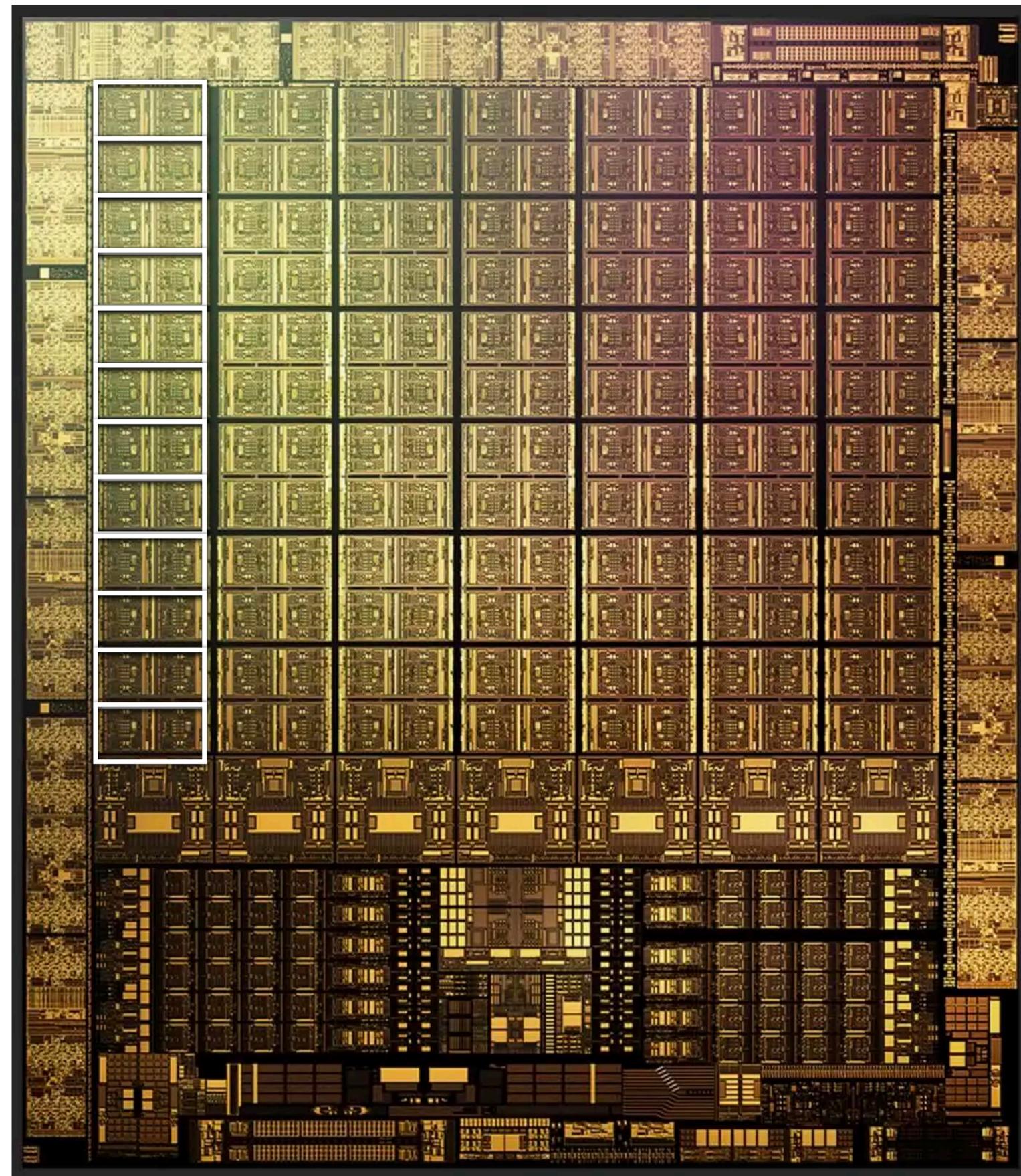
72 cores (1.5 Ghz)



NVIDIA Ampere GA102 GPU

GeForce RTX 3080 (2020)

**17,408 fp32 multipliers organized
in 68 major processing blocks.**



Supercomputing

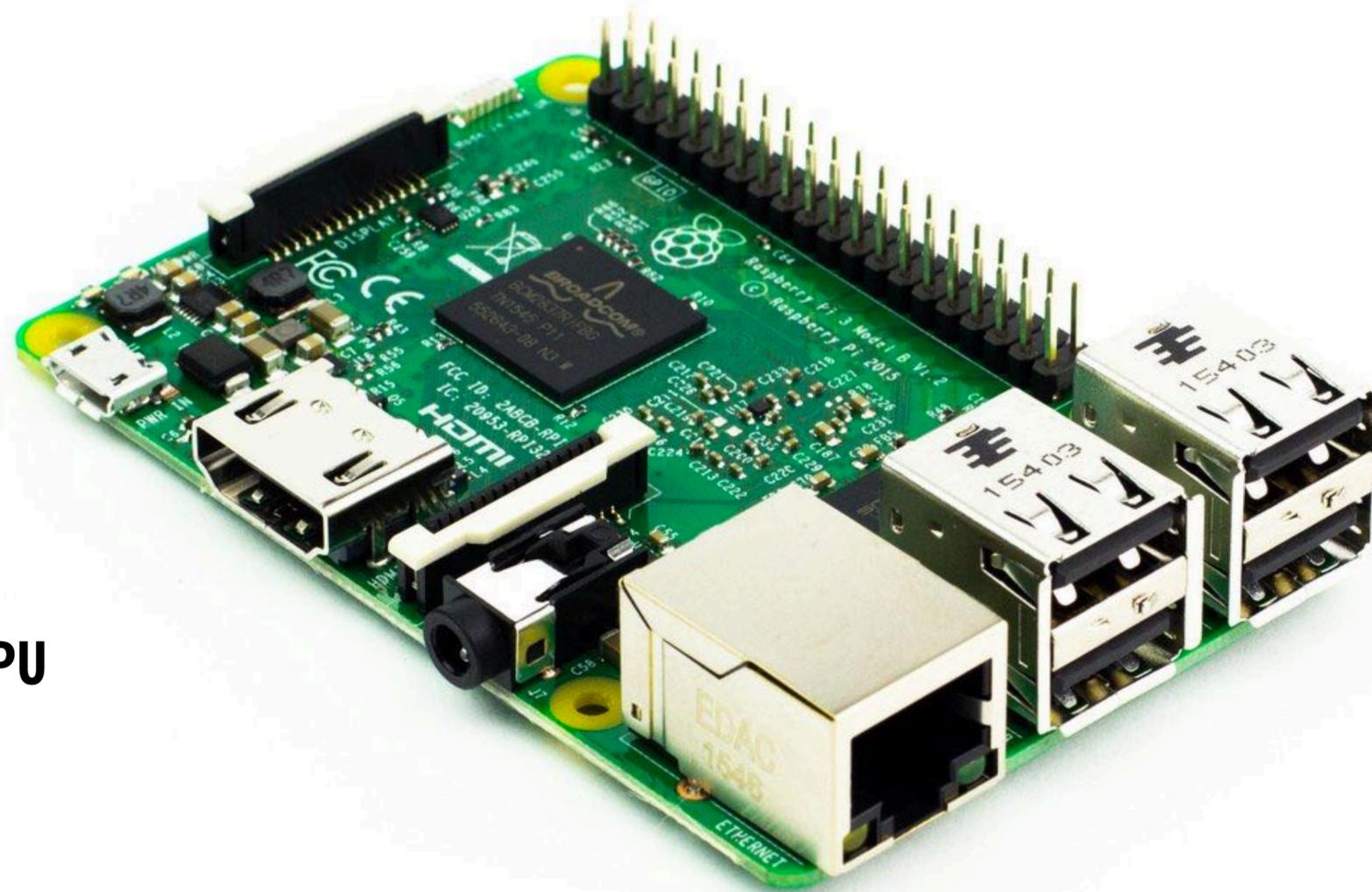
- Today: combinations of multi-core CPUs + GPUs
- Oak Ridge National Laboratory: Summit (currently #2 supercomputer in world)
 - 9,216 x 22-core IBM Power9 CPUs + 27,648 NVIDIA Volta GPUs



Mobile parallel processing

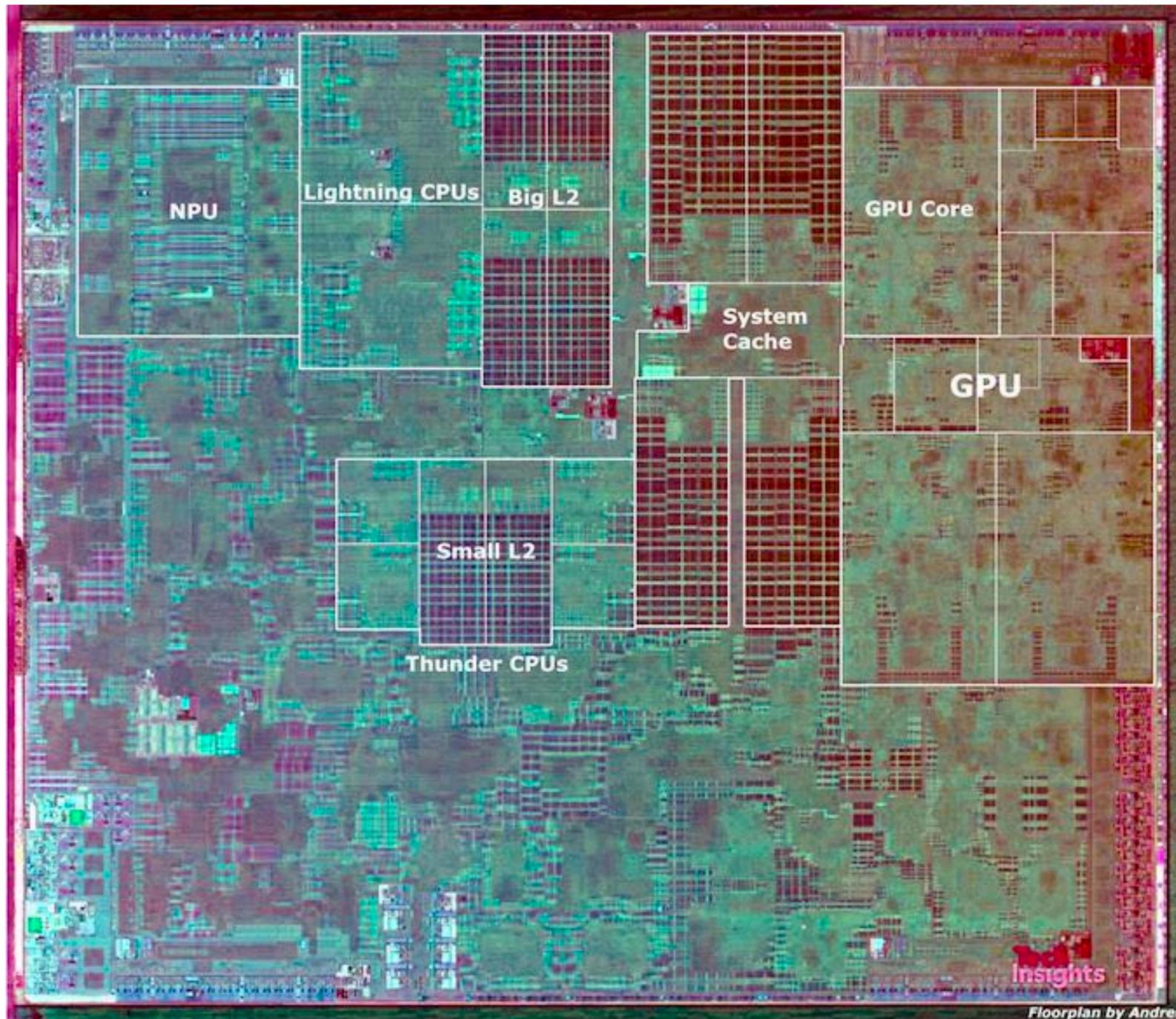
Raspberry Pi 3

Quad-core ARM A53 CPU



Mobile parallel processing

Power constraints heavily influence the design of mobile systems



Apple A13 Bionic (in iPhone 11)

2 “big” CPU cores +
4 “small” CPU cores +

Apple-designed multi-core GPU +
Image processor +
Neural Engine for DNN acceleration +
Motion processor

Parallel + specialized HW

- **Achieving high efficiency will be a key theme in this class**
- **We will discuss how modern systems are not only parallel, but also specialize processing units to achieve high levels of power efficiency**

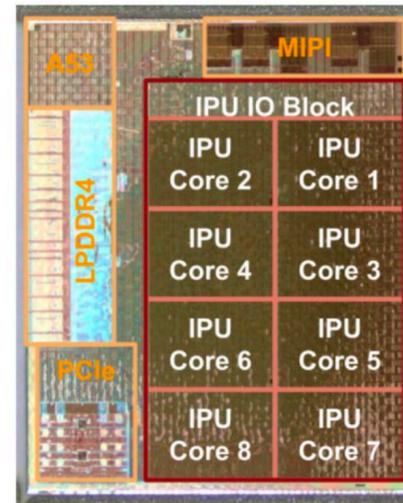
Another recent smartphone

Google Pixel 2 Phone:

Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

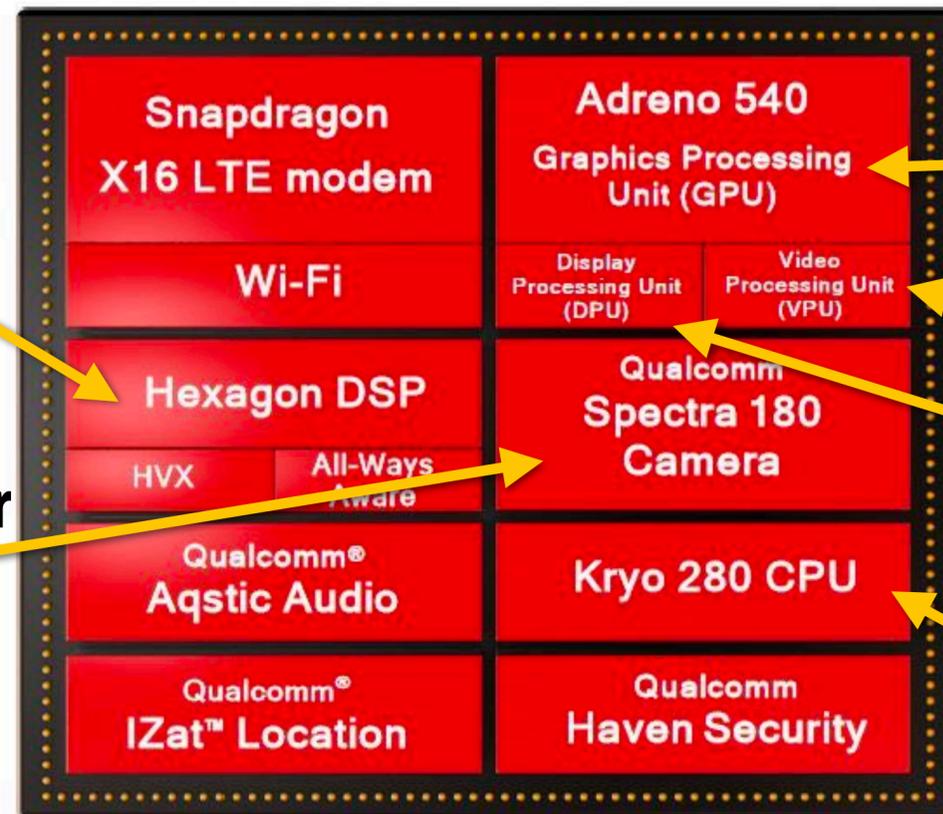


Visual Pixel Core
Programmable image processor and DNN accelerator



“Hexagon”
Programmable DSP
data-parallel multi-media processing

Image Signal Processor
ASIC for processing camera sensor pixels



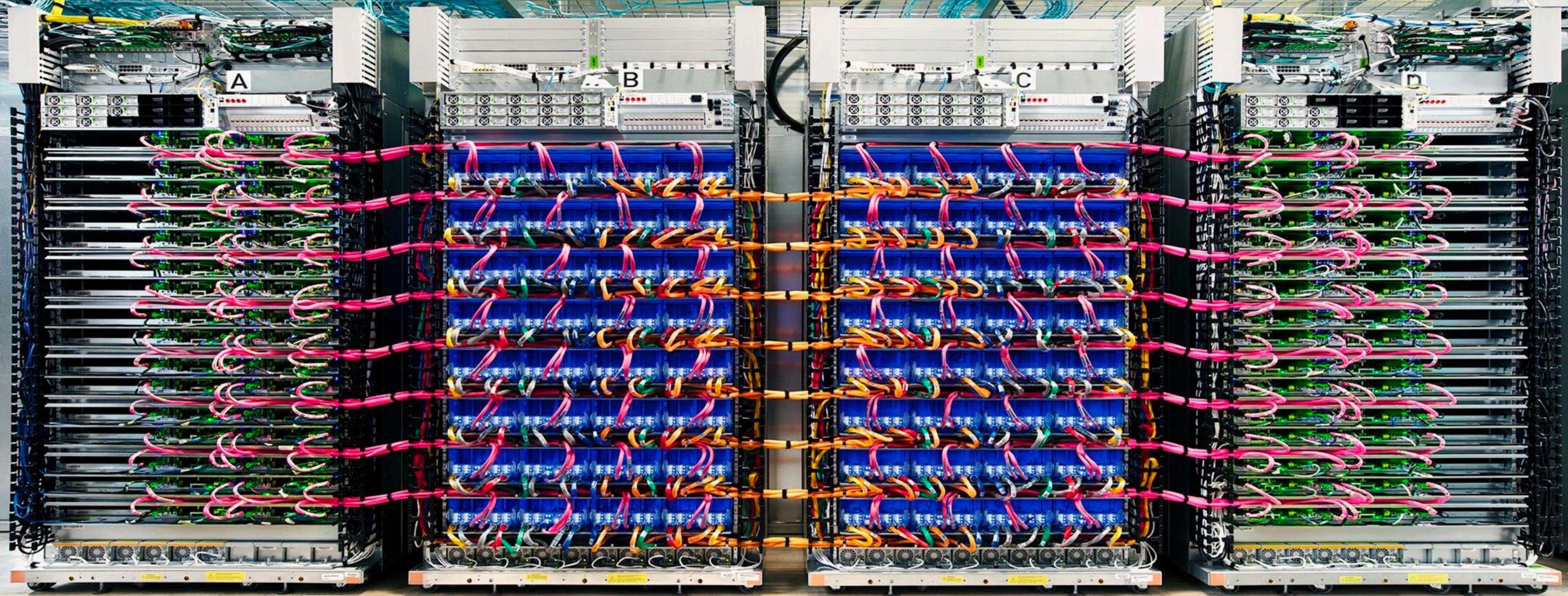
Multi-core GPU
(3D graphics,
OpenCL data-parallel compute)

Video encode/decode ASIC

Display engine
(compresses pixels for
transfer to high-res screen)

Multi-core ARM CPU
4 “big cores” + 4 “little cores”

Datacenter-scale applications



Google TPU pods

TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.

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**
 - **Which means you need to know how to write parallel code**
- **Writing parallel programs can be challenging**
 - **Requires problem partitioning, communication, synchronization**
 - **Knowledge of machine characteristics is important**
- **I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it!**

Welcome to CS149!

- Get signed up on Piazza
- Get signed up on the website
- Find yourself a partner (we will help you)
- Let me know about suggested changes to virtual lecture delivery
- Take your mask wearing seriously when around others in class activities like office hours



Prof. Kayvon



Prof. Olukotun



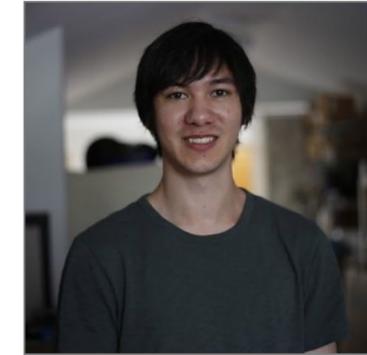
Yuhan



Olivia



Luis



Teguh



Jack