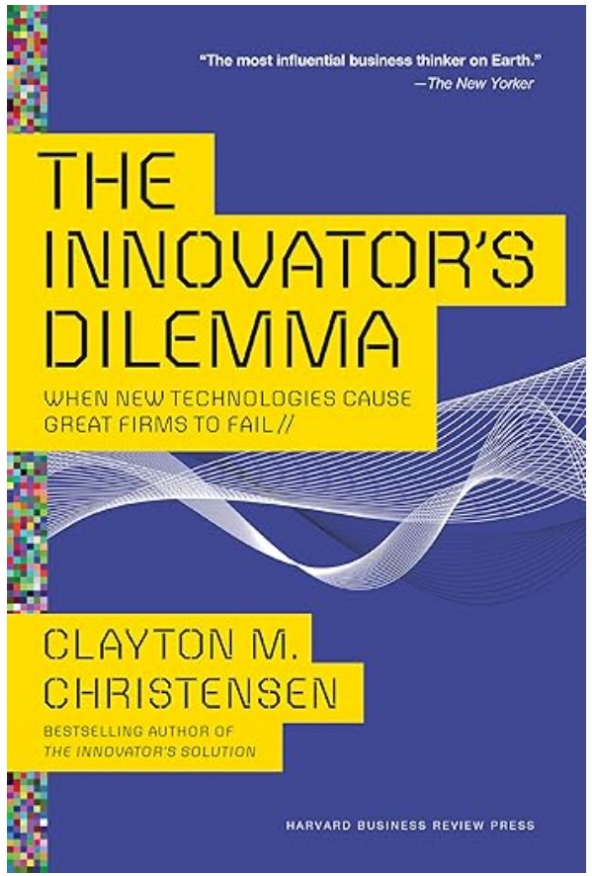
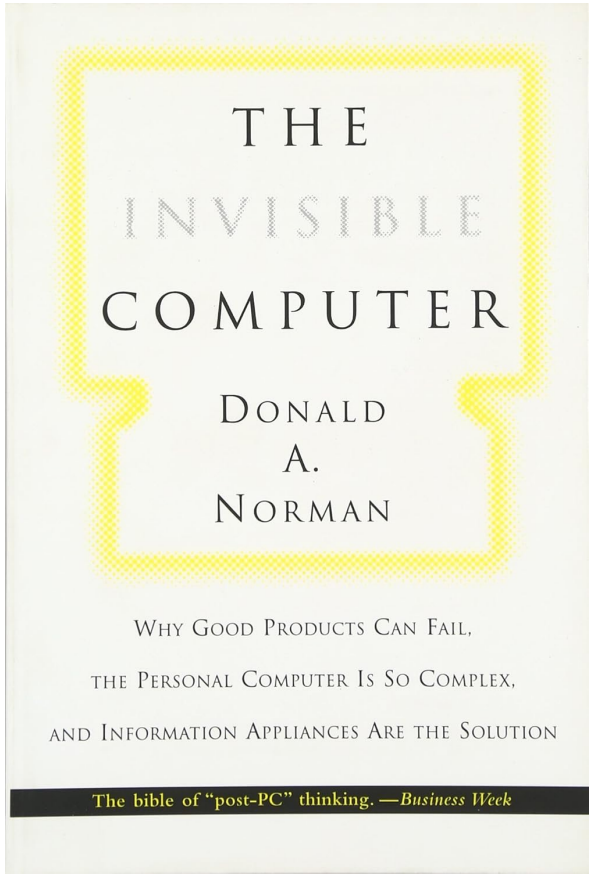
# An Introduction to computers and how software runs on them

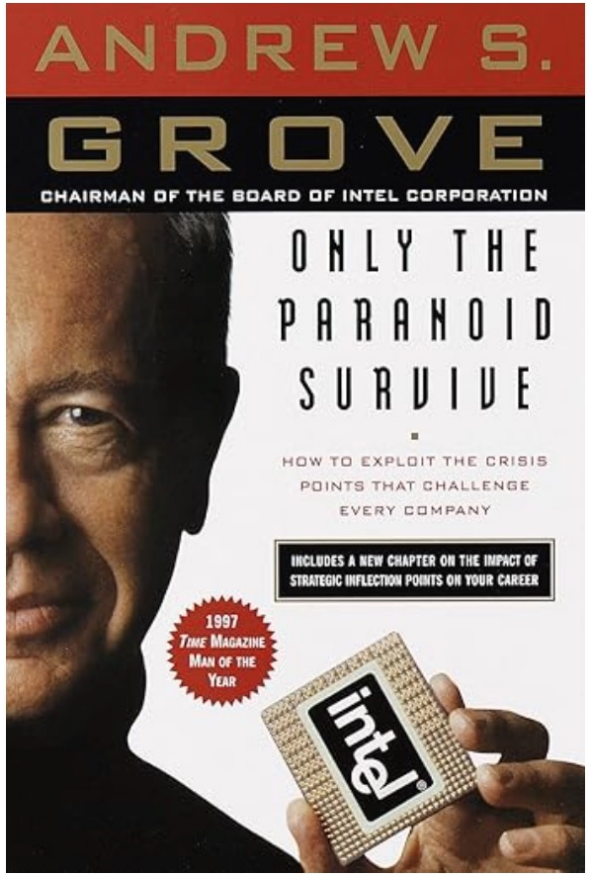
**Tim Mattson**

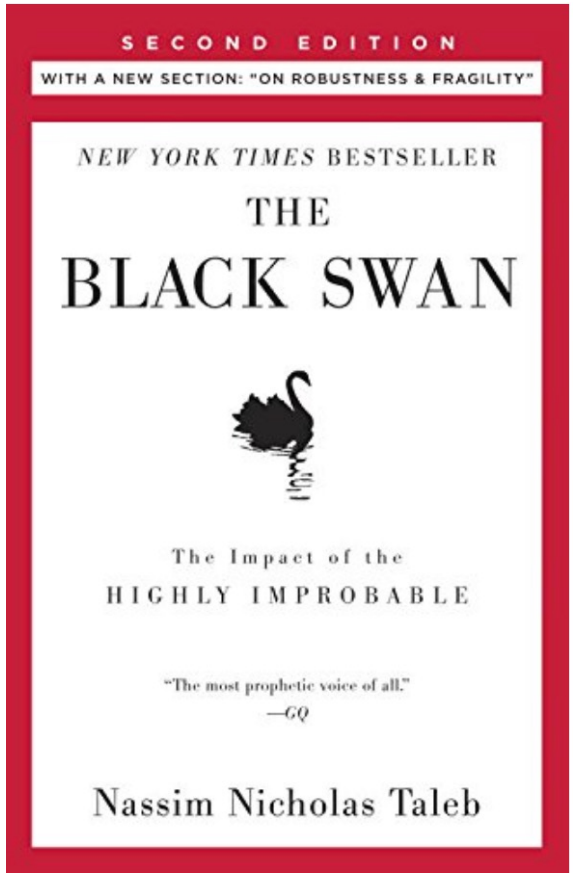

Tim demonstrates his new invention: Kayak snorkeling. Palawan Philippines, 2019

# 4 books to make sense of the Computer industry

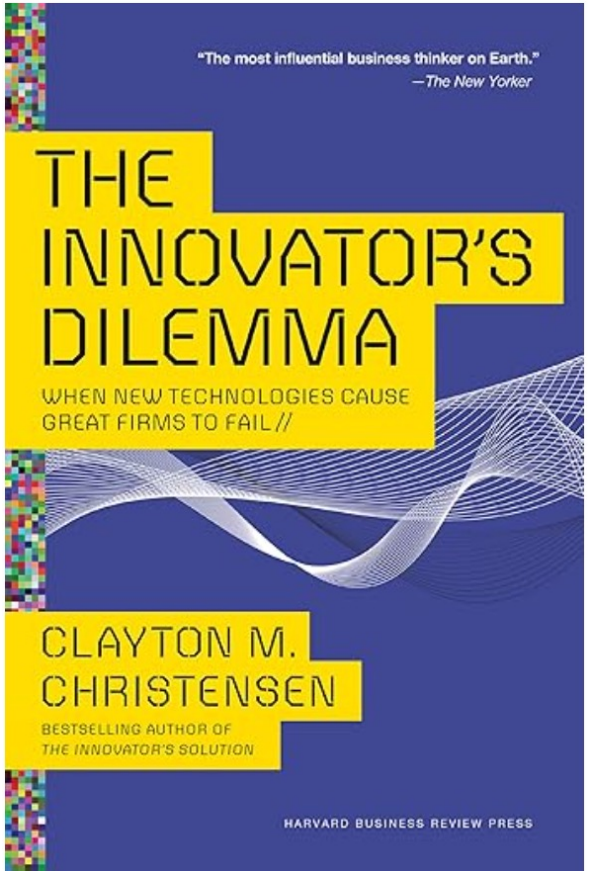


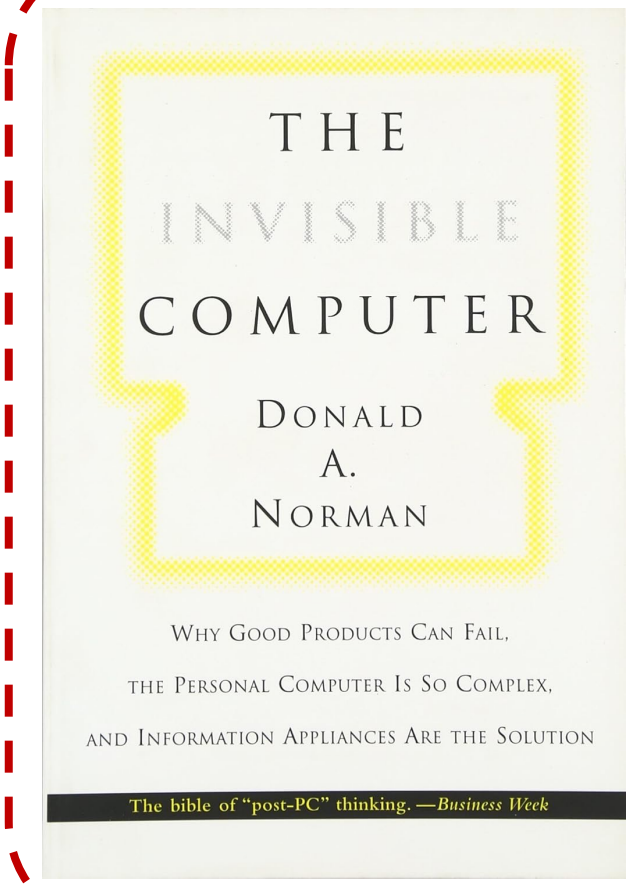



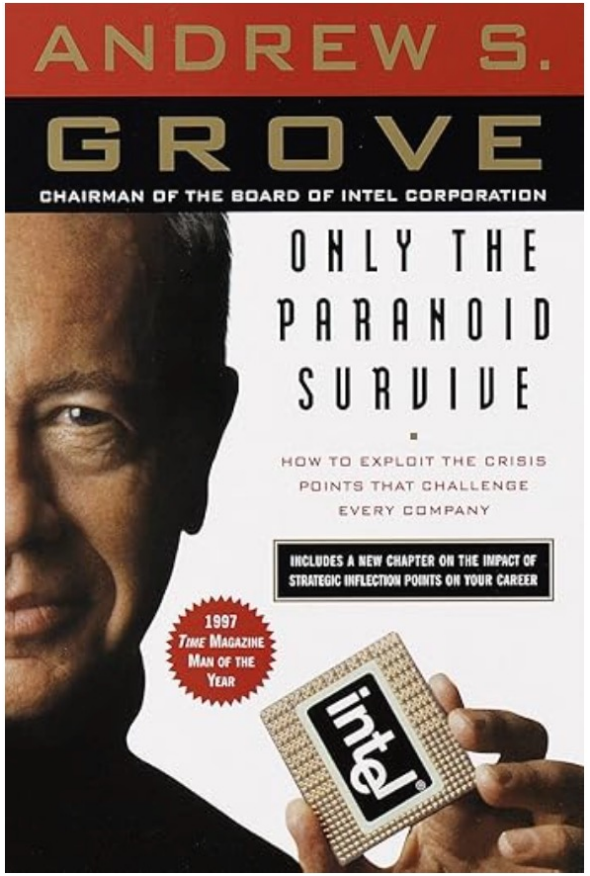



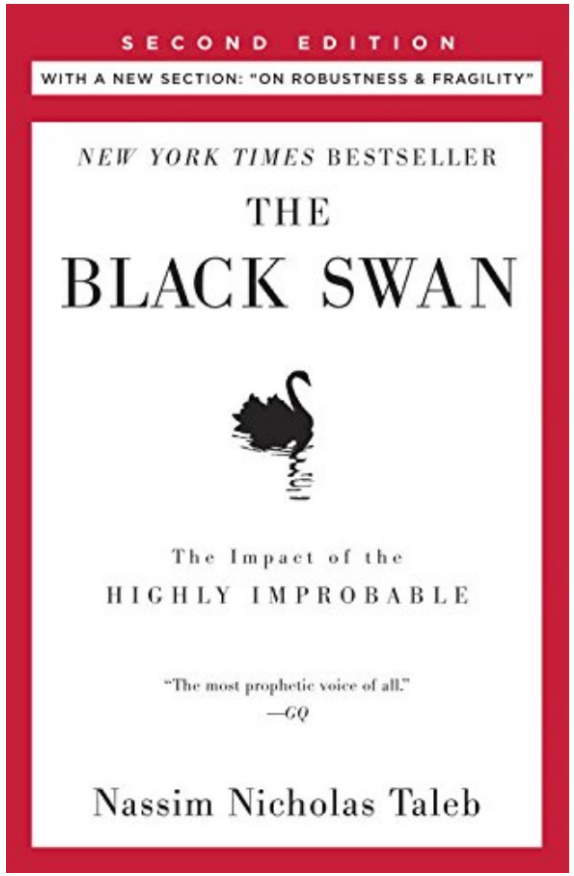
# 4 books to make sense of the Computer industry



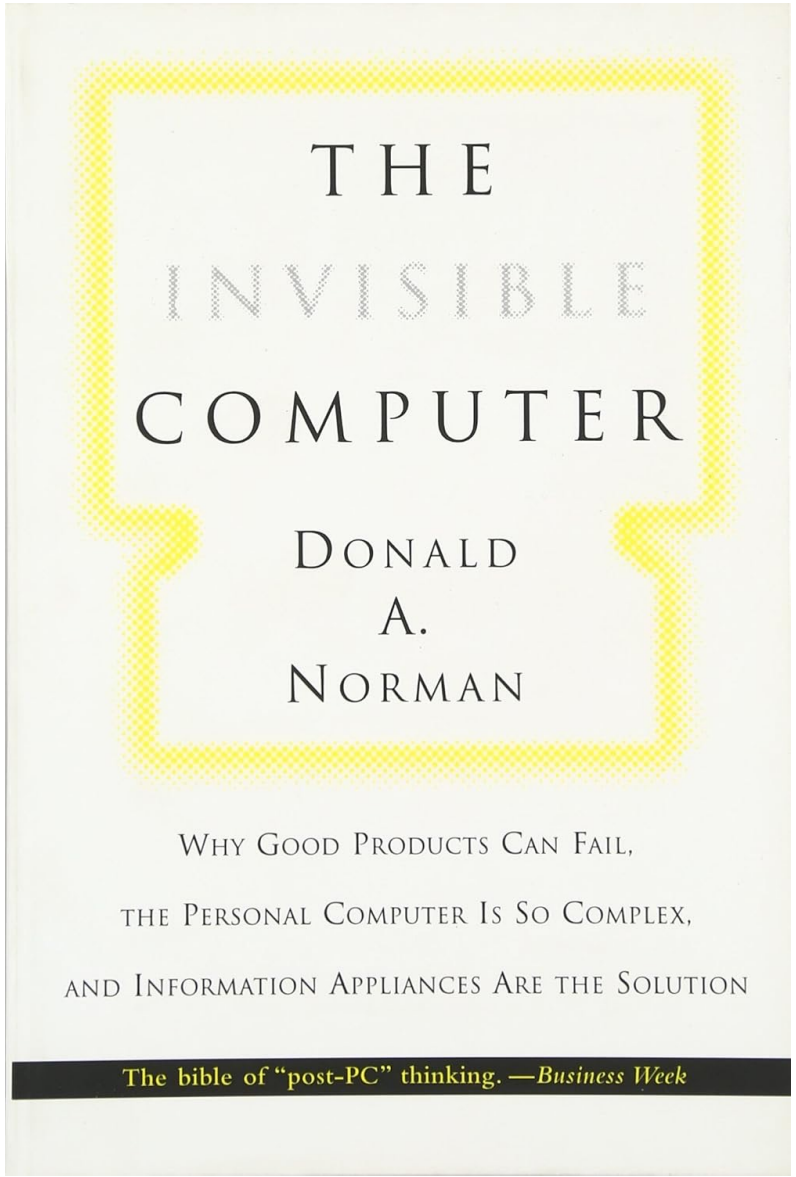




They are all good and worth reading, but we'll focus on this one for now



# Computing for Humans




Don Norman, trained in Electrical Engineering and cognitive psychology, pioneered the idea of "Human centered design" in computer system design.

Joining Apple in 1993, he helped solidify the company's approach to design … taking existing technology (e.g. Apple did not invent the smart phone) and making it better by designing it around user experience.

He summarizes his thinking in the famous book "The Invisible Computer" published in 1998.

**Using a classic example from Don Norman's book …**

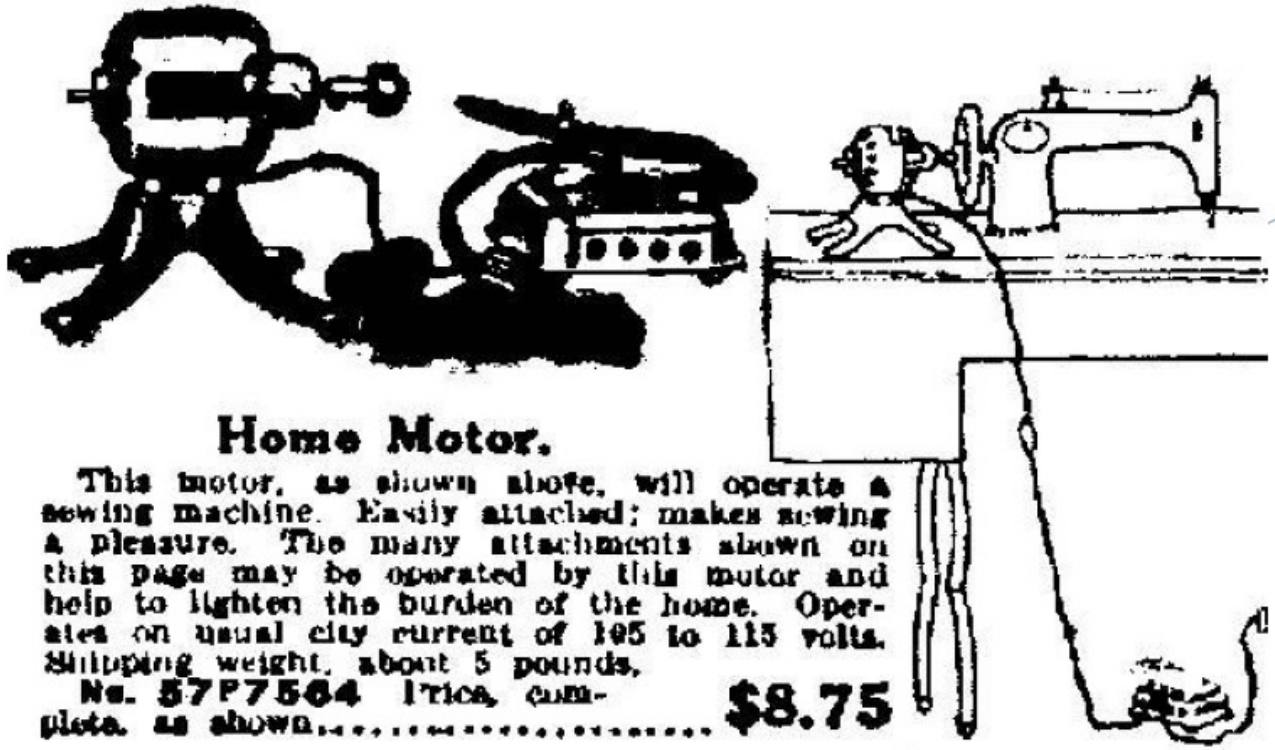
**Ask yourself …. When was the last time you bought an electric motor?**

# An example from the book “The Invisible Computer”

How many **electric motors**. Have you purchased recently?

They used to be something you thought about and bought explicitly.

Now they are buried inside cars, appliances and other products … **they are invisible**



**Home Motor.**

This motor, as shown above, will operate a sewing machine. Easily attached; makes sewing a pleasure. The many attachments shown on this page may be operated by this motor and help to lighten the burden of the home. Operates on usual city current of 105 to 115 volts. Shipping weight, about 5 pounds.

No. 57F7564 Price, complete, as shown..... $8.75

**Beater Attachment.**

Whips cream and beats eggs, and many other uses will be found for these attachments when used in connection with the Home Motor. Parts include the stand, handle and the beater. Shipping weight, about 14 ounces.

No. 57F7585 Price..... $1.30

**Churn and Mixer Attachment.**

Used in connection with the Home Motor, makes a small churn and mixer for which you will find many uses. The attachments include the base, supports, mixer, handle and special cover for jar. Shipping weight, about 1¾ pounds.

No. 57F7582 Price..... $1.30

**Fan Attachment.**

Includes fan and guard which can be quickly attached to Home Motor, and will be a great comfort in hot weather. Shipping weight, about 14 ounces.

No. 57F6215 Price..... $1.30

**A page from a Sears and Roebuck Catalog: 1918**

About $100 in today’s dollars.

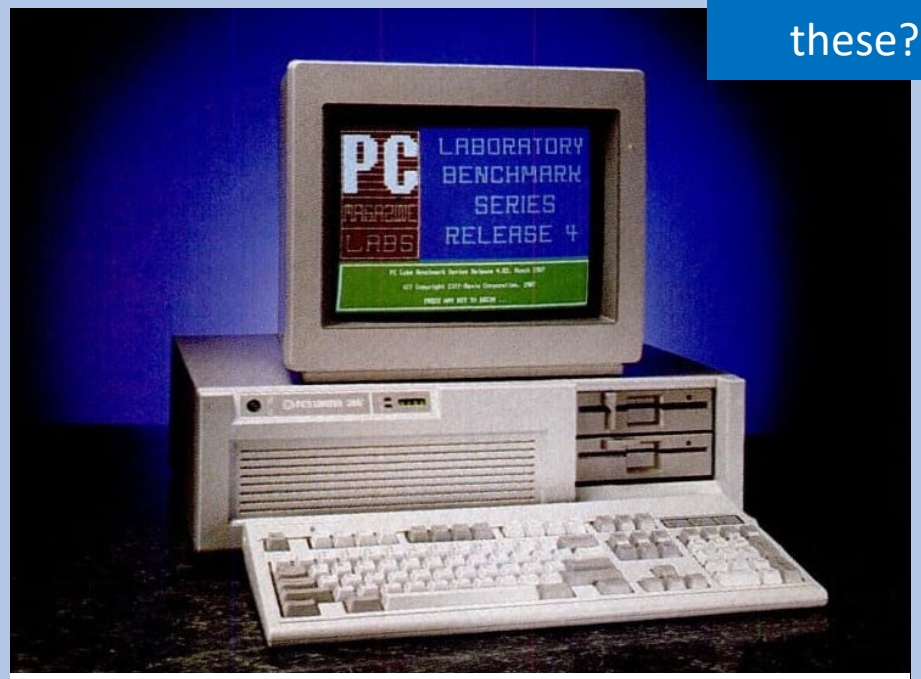
Source: S. Edwards, C.S. Columbia Univ. http://www1.cs.columbia.edu/~sedwards/presentations/eitc2002.9up.pdf From Donald Norman, *The Invisible Computer, 1998.*

# An example from the book “The Invisible Computer”

The PC is going the way electric motors.

They are becoming invisible!

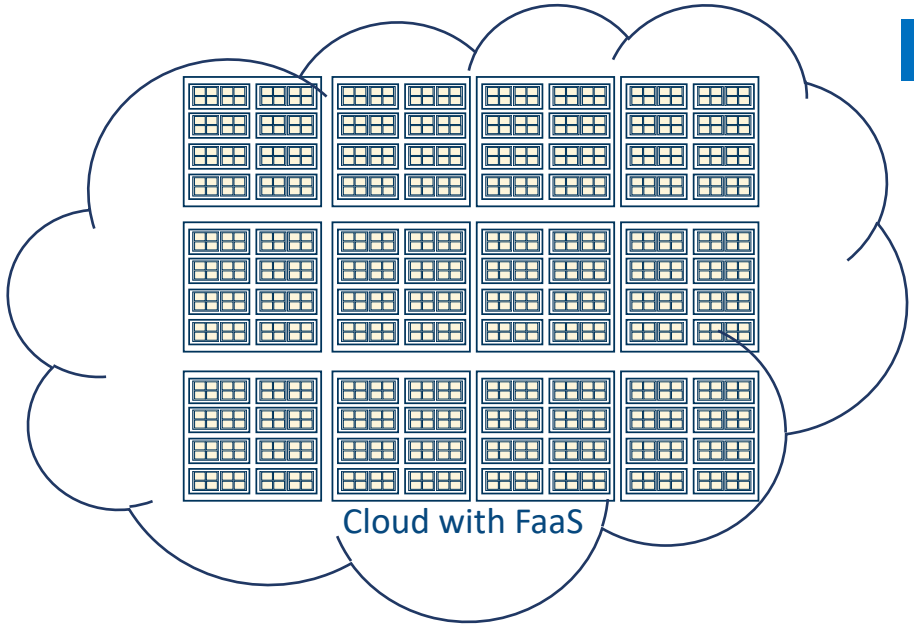
Remember these?




Dell Limited 386-16 PC (1987). 16 MHz Intel 80386 CPU, 1 MB RAM, 1.21 MB Floppy, and 40 MB hard drive. $4,799 ($13,574 in 2024 dollars).
https://www.pcmag.com/news/the-golden-age-of-dell-computers


Source: S. Edwards, C.S. Columbia Univ. http://www1.cs.columbia.edu/~sedwards/presentations/eitc2002.9up.pdf From Donald Norman, *The Invisible Computer, 1998.*

# Invisible computers




FaaS: Function as a service. You see the function, not the hardware.


Nintendo Switch™ mobile gaming console. Nvidia Tegra X1 with 4 ARM Cortex A57 cores each with a NEON vector unit, 4 GB DRAM, and an Nvidia GM20B GPU (supports CUDA and OpenCL 1.2). 64 GB Memory.


Hello Barbie™, Uses AI in the cloud to generate conversation. Launched and terminated in 2015 due to privacy concerns over stored dialog by children.
Marvell 88MW300 ARM Cortex-M4F processor plus a 24 bit Nuvoton NAU8810 audio codec. 16 Mbit Gigadevice GD25Q16 Flash memory.
https://www.microcontrollertips.com/teardown-electronics-hello-barbie/

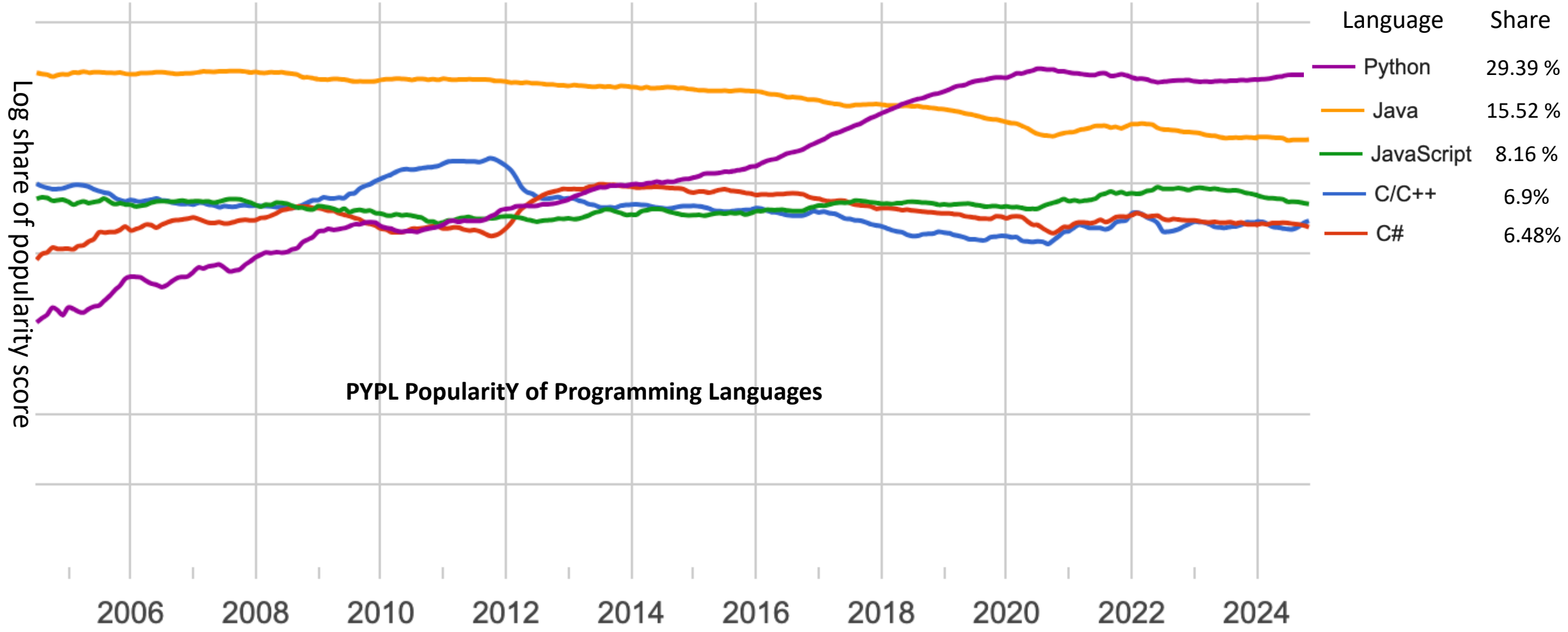
Mercedes-Benz SL roadster
http://www.extremetech.com/extreme/125621-mercedes-benz-over-the-air-car-updates

~50 computers on average in a car.
High-end cars have ~100


Roku streaming stick 4K, ARM Cortex A55, 1GB RAM
https://developer.roku.com/docs/specs/hardware.md

# Python and friends are  the languages of invisible computing

Consider the changes in most popular programming languages…

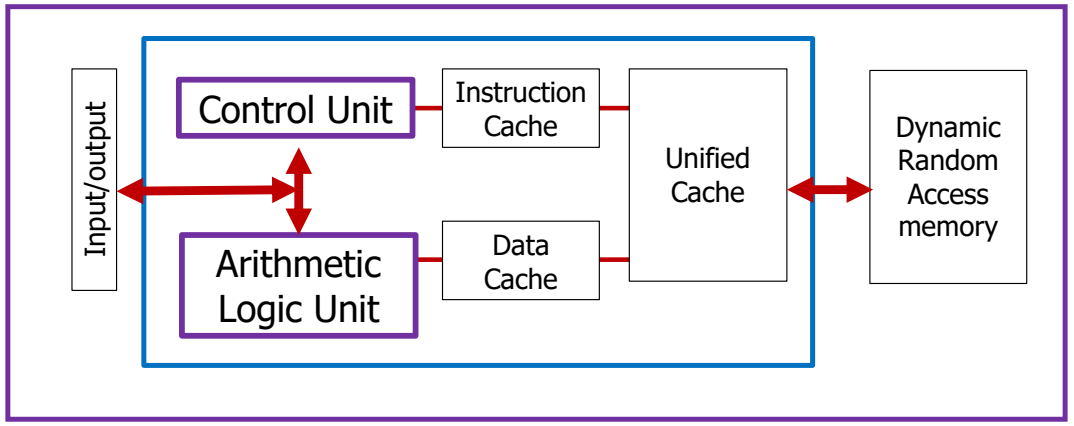



https://pypl.github.io/PYPL.html

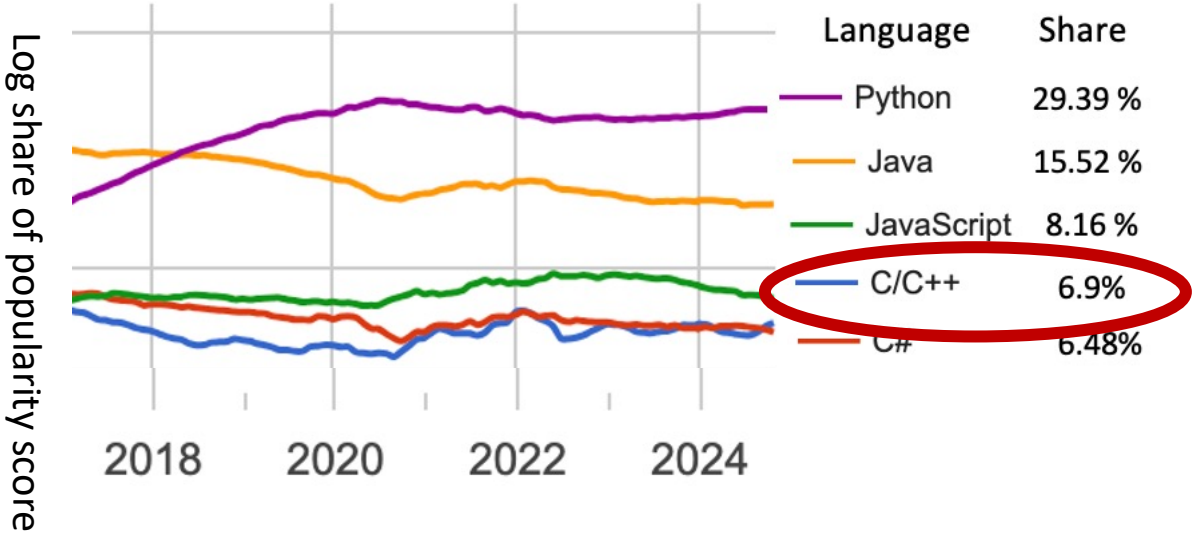
# Invisible computers and scientific computing

- In scientific computing, we are often limited by performance or problem-size.
- We must understand what is happening inside our computers so we can design software that works well with the hardware and meets our needs.

***We must make our computers visible***




- And we must use programming languages that lets us directly deal with the hardware.
  - C, C++, and Fortran are the key programming languages of Scientific Computing.



# Before proceeding, I want to pause and say something about Jargon in Computer Science

# Computer Science has a problem with Jargon …

This famous paper describes the ideas behind the design of the Java Programming Language.

## Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

guy.steele@sun.com

October 1998

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she," and when I say "his" I mean "his or her."

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an "s" or "z" sound to the end of a word that names it.)

⟨noun⟩ ::= ⟨noun that names one thing⟩ "s"
| ⟨noun that names one thing⟩ "es"

https://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

# Computer Science has a problem with Jargon …

## Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

guy.steele@sun.com

October 1998

It starts in an odd if not silly way

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she," and when I say "his" I mean "his or her."

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an "s" or "z" sound to the end of a word that names it.)

Any word with more than one syllable is defined before it is used

⟨noun⟩ ::= ⟨noun that names one thing⟩ "s"
| ⟨noun that names one thing⟩ "es"

https://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

# Computer Science has a problem with Jargon …

## Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

guy.steele@sun.com

October 1998

But later when he writes jargon filled text, the meaning is clear since everything was defined in advance

hat a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see w

"his

the e

I have said in the past, and will say now, that I think it would be a good thing for the Java programming language to add generic types and to let the user define overloaded operators. Just as a user can code methods that can be used in just the same way as methods that are built in, the user ought to have a way to define operators for user defined classes that can be used in just the same way as operators that are built in. What is more, I would add a kind of class that is of light weight, one whose objects can be cloned at will with no harm and so could be kept on a stack for speed and not just in the heap.

⟨noun⟩ ::= ⟨noun that names one thing⟩ "s"
| ⟨noun that names one thing⟩ "es"

https://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

# Computer Science has a problem with Jargon …

## Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

guy.steele@sun.com

October 1998

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she," and when I say "his" I mean "his or her."

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an "s" or "z" sound to the end of a word that names it.)

⟨noun⟩ ::= ⟨noun that names one thing⟩ "s"
| ⟨noun that names one thing⟩ "es"

https://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

Inspired by this paper … I will try to define all specialized jargon before using it (though I won't follow >1 syllable rule).

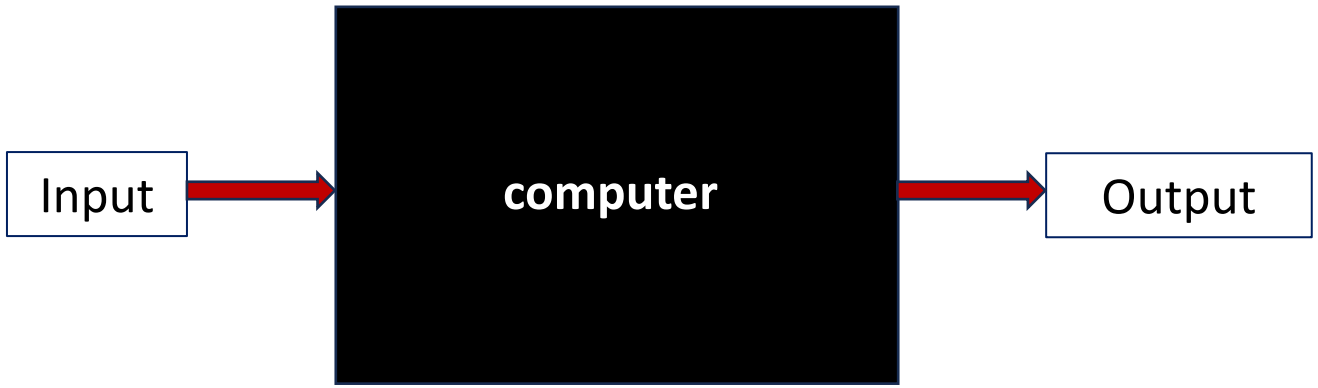
So be patient if I seem to be spending a lot of time defining jargon … in the long run, it is very important for all of us to understand fully any terms used in these lectures

## Let’s go back to basics ….

## What is a computer?

# What is a computer:

- **Computer:**
  - A machine that transforms *input values* into *output values*.

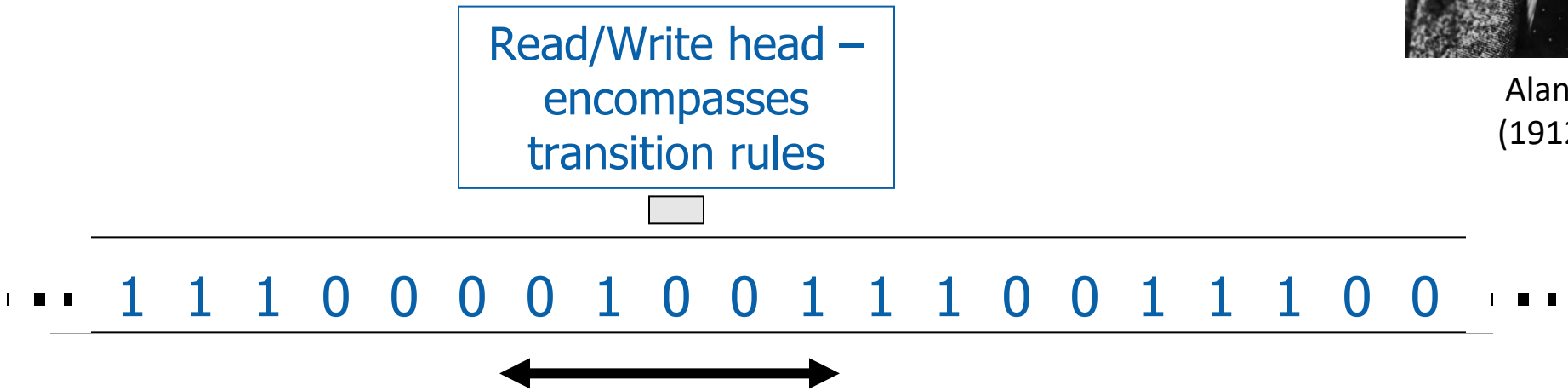



- The computer as a black-box is not very helpful. We need a bit more detail.

# Computer models: Turing Machine

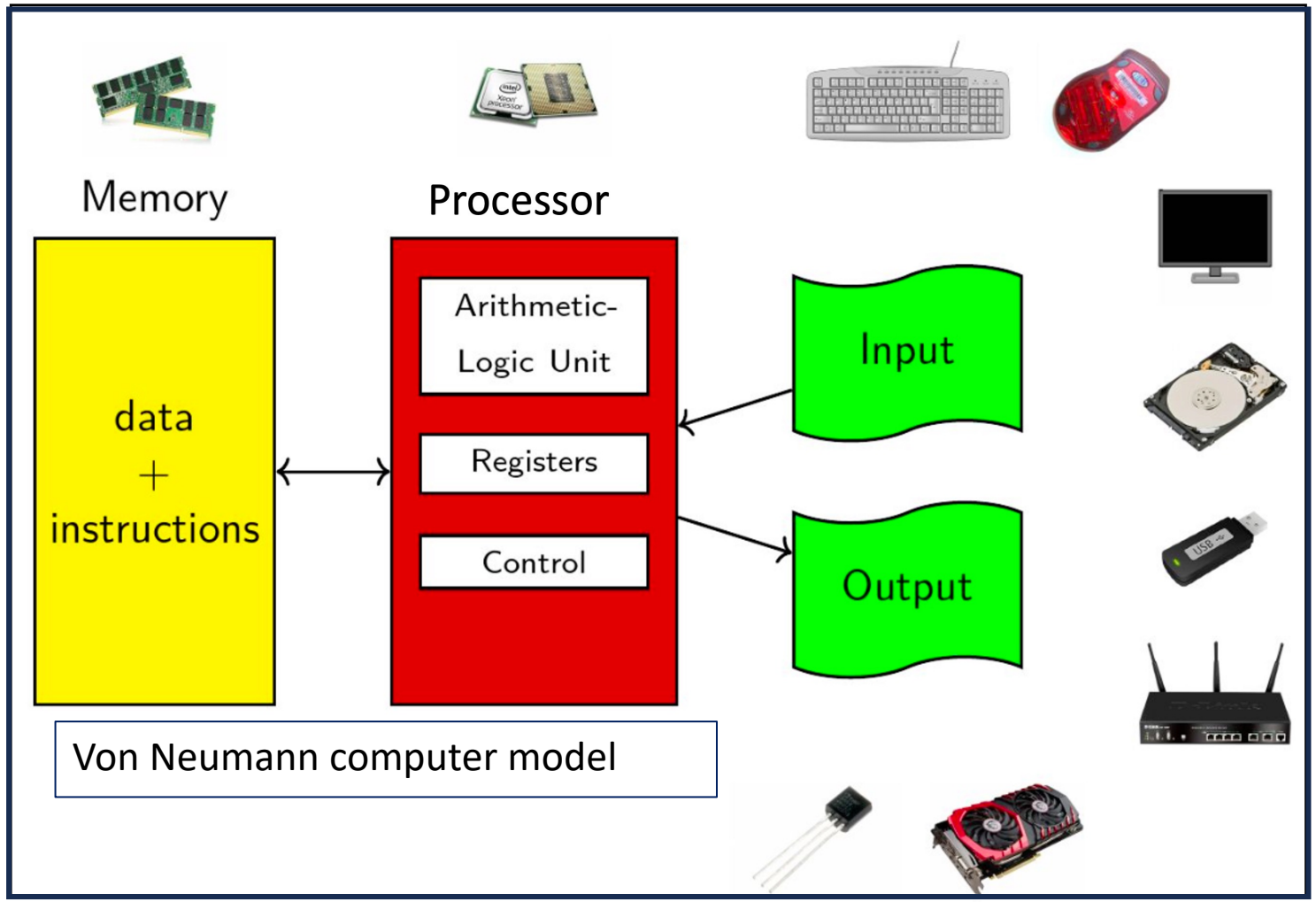
Alan Turing
(1912-1954)

- Alan Turing proposed a general model of a computer and showed that it was universal:




- Read an “infinite” tape of 1’s and 0’s. Based on the pattern of values, shift the tape, read values and write values. These are controlled by transition rules (i.e. a program)
- This was useful for proving mathematical theorems about computing, but not for actually working with computers.

# Von Neumann or a “Stored Program” Model




John von Neumann proposed a more useful model where a computer consists of:

(1) control unit,

(2) Arithmetic-Logic unit (ALU),

(3) registers that hold values close to the ALU

(4) memory that holds both the data and the sequence of instructions (the program).

Image Source: Felice Pantaleo, CERN, ESC’23

# Program Execution in the Von Neumann Model

- The program consists of a set of instructions stored in memory. An program counter (PC) points to the next instruction.
- For each instruction you decode the instruction, load registers, execute instruction, write back results, and increment the PC. Do this until the PC hits the end of the program.

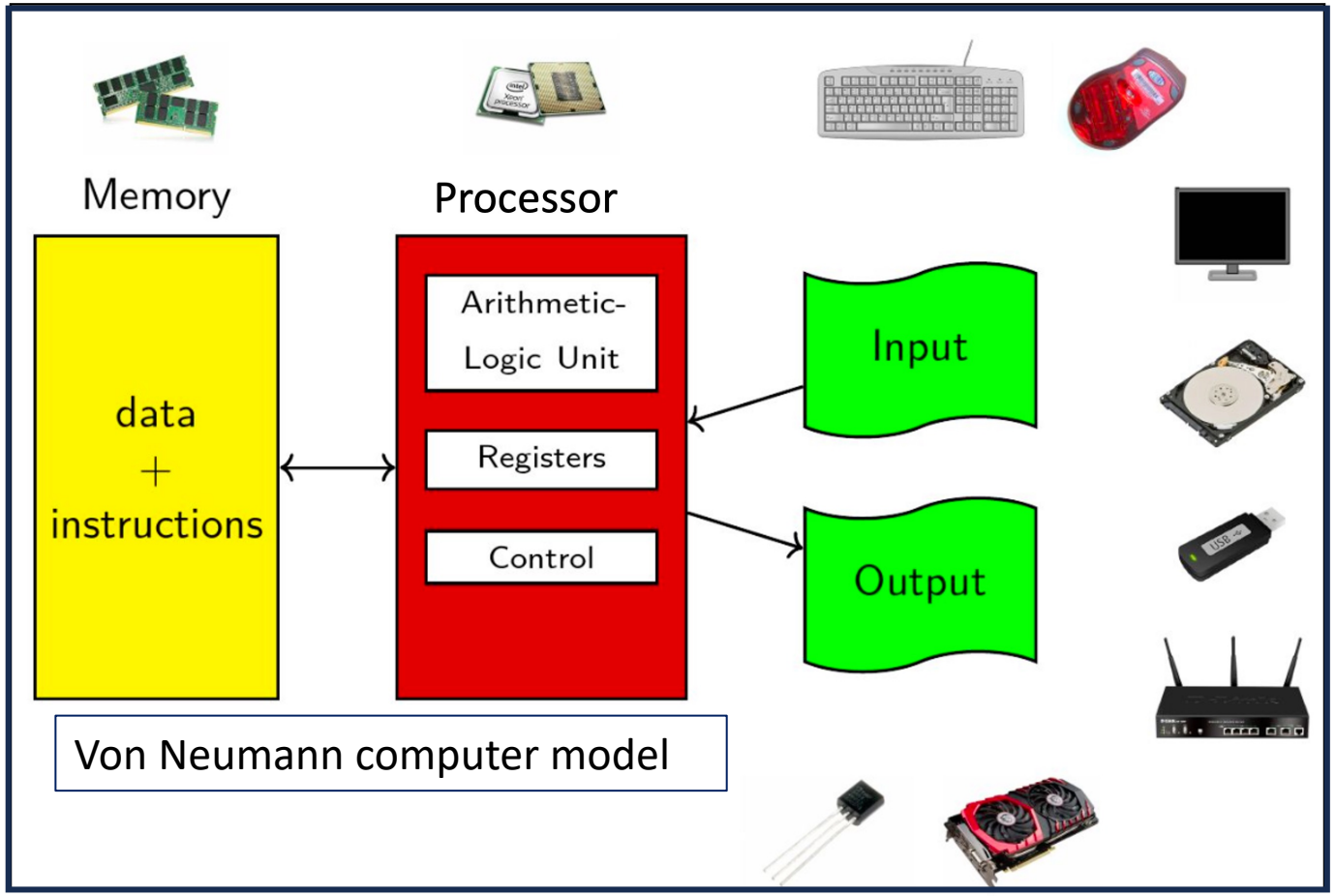
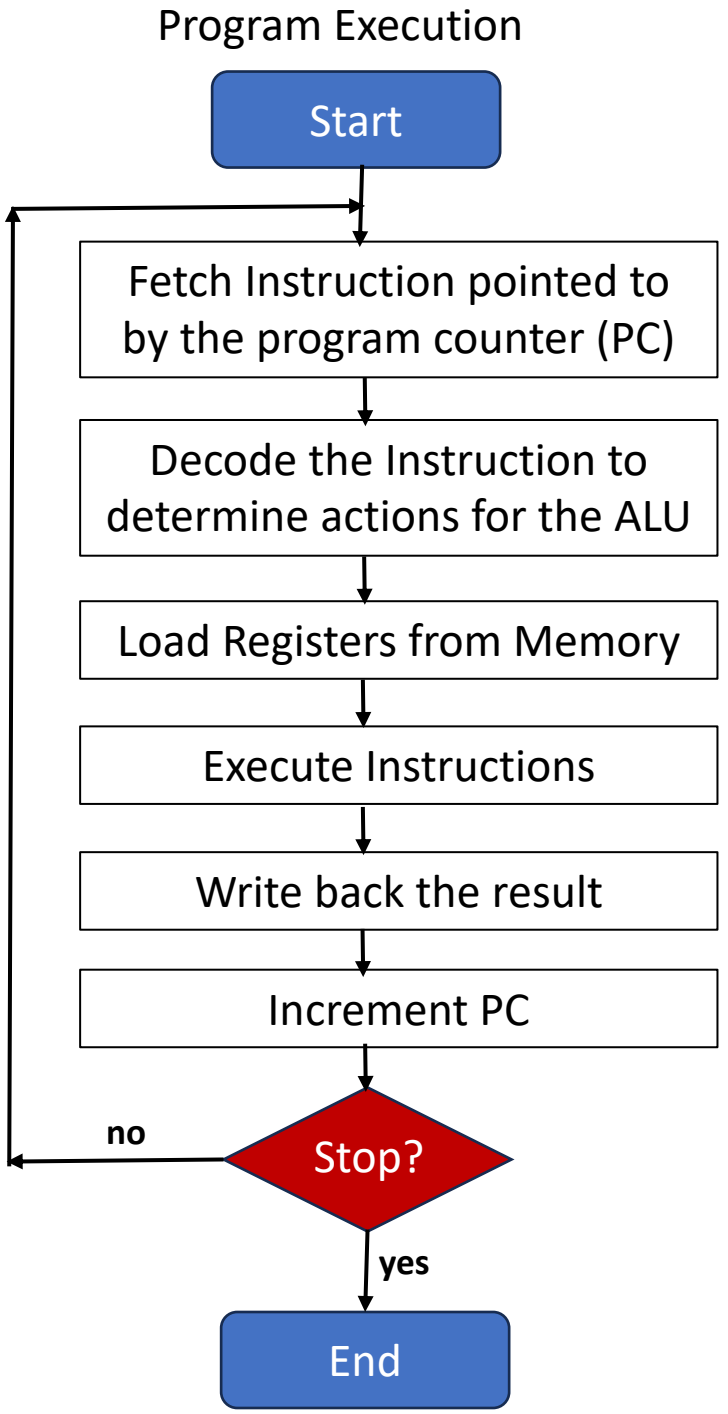
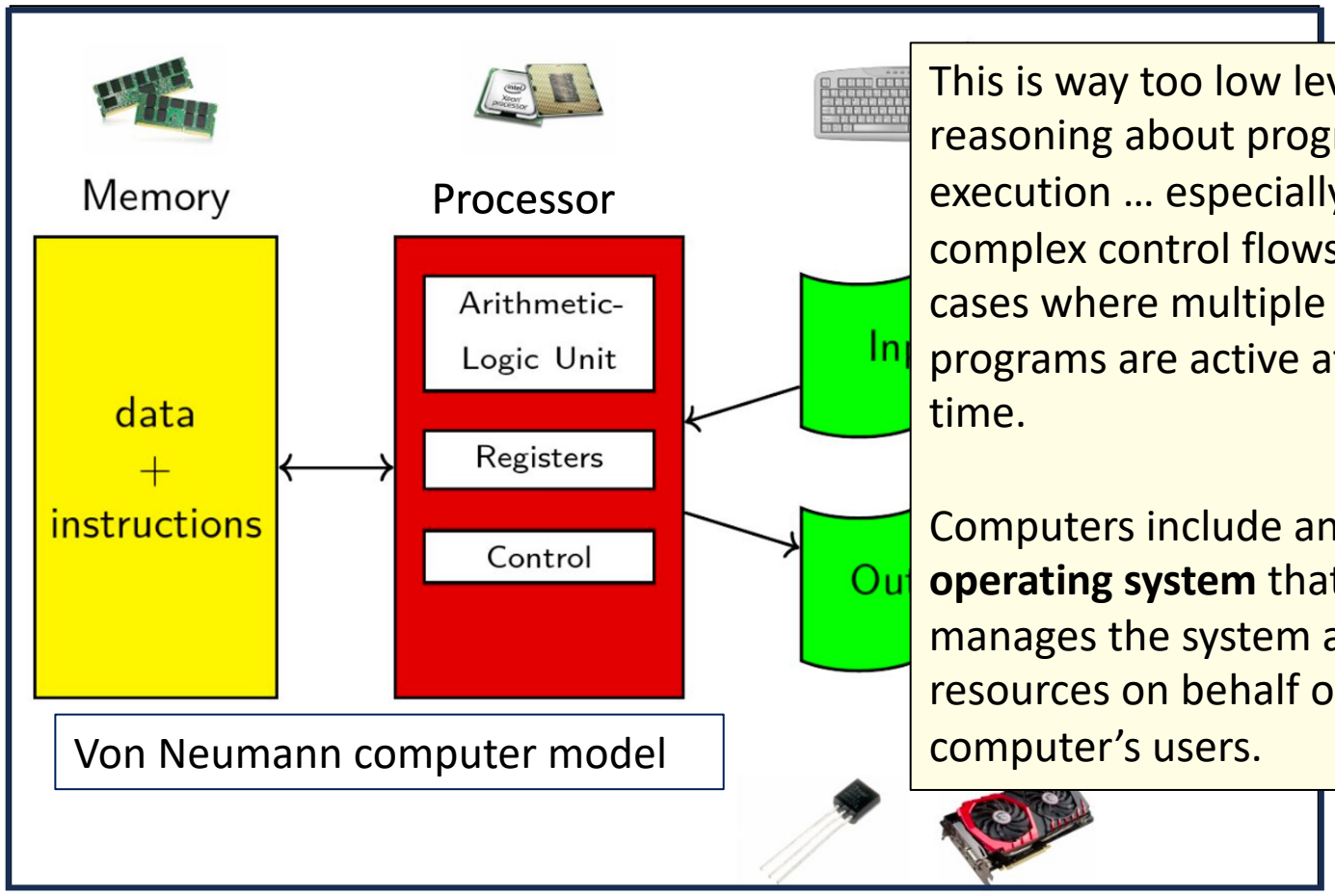



Image Source: Felice Pantaleo, CERN, ESC'23



# Program Execution in the Von Neumann Model

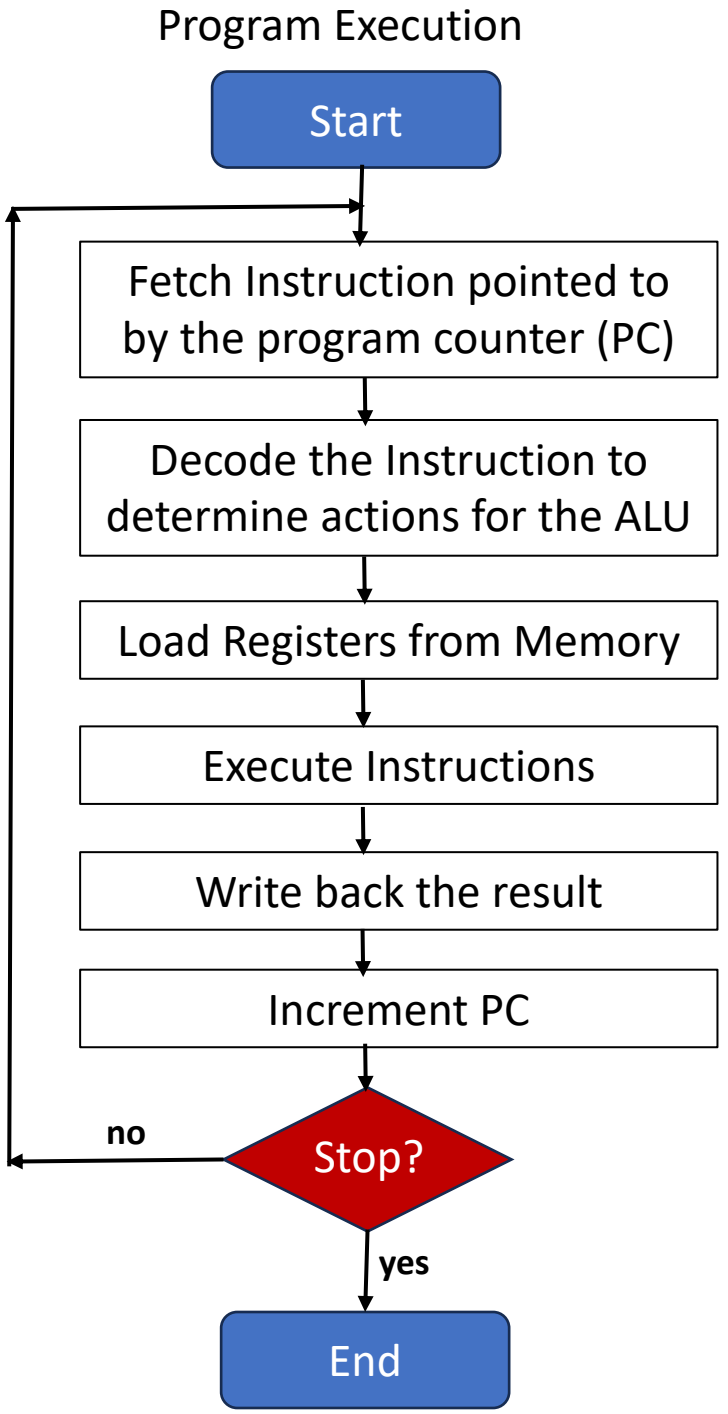
- The program consists of a set of instructions stored in memory. An program counter (PC) points to the next instruction.
- For each instruction you decode the instruction, load registers, execute instruction, write back results, and increment the PC. Do this until the PC hits the end of the program.




This is way too low level for reasoning about program execution … especially for complex control flows and cases where multiple programs are active at one time.

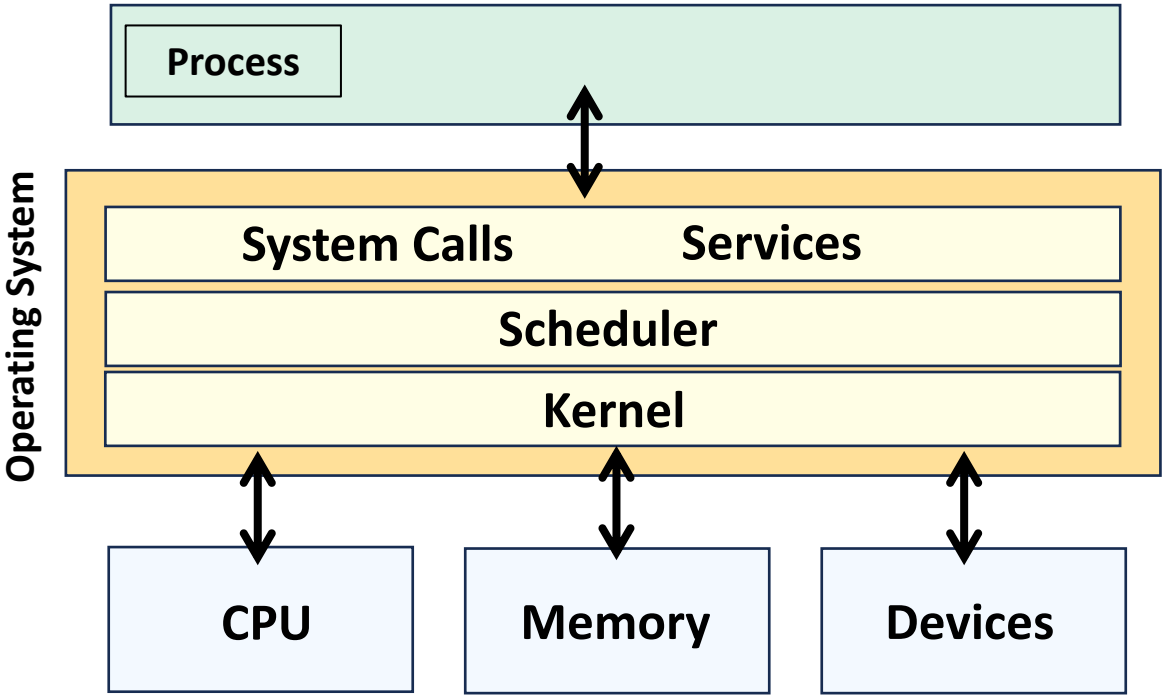
Computers include an **operating system** that manages the system and its resources on behalf of the computer's users.

Image Source: Felice Pantaleo, CERN, ESC'23

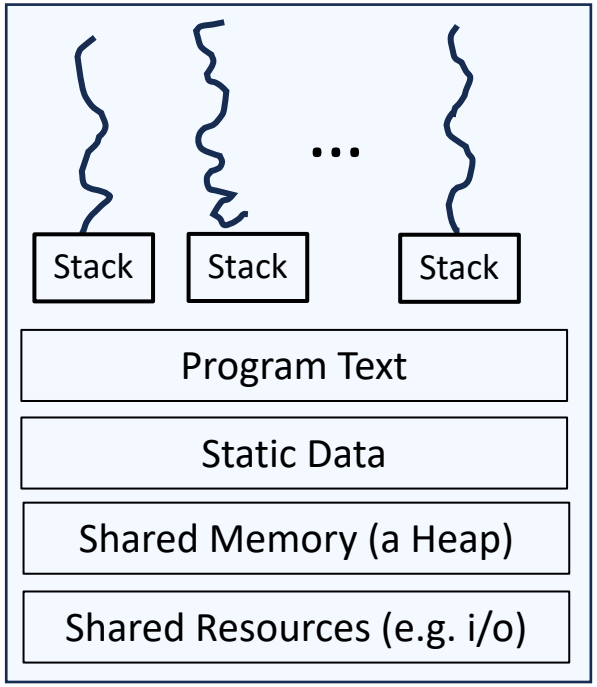


# Everything you need to know about operating systems

- The Operating System (OS) is software the manages the computer hardware.
- It consists of a low level kernel and a collection of resources that run on top of the kernel to service the needs of users of the system.
- The kernel provides security guarantees and isolation between running programs (processes).




- A process is an instance of an executing program.




One or more threads running the same program text

Each thread has private memory (a stack)

Static Memory fixed at compile time

Shared Memory and shared resources are available to all threads

- Modern operating systems support multi-tasking. This means that multiple processes are active at one time with a scheduler (part of the OS) quickly switching (a context switch) between processes.
- For the user, this creates the illusion that all of the processes are running at the same time (more on this later).

**Modern computers follow the von-Neuman model**

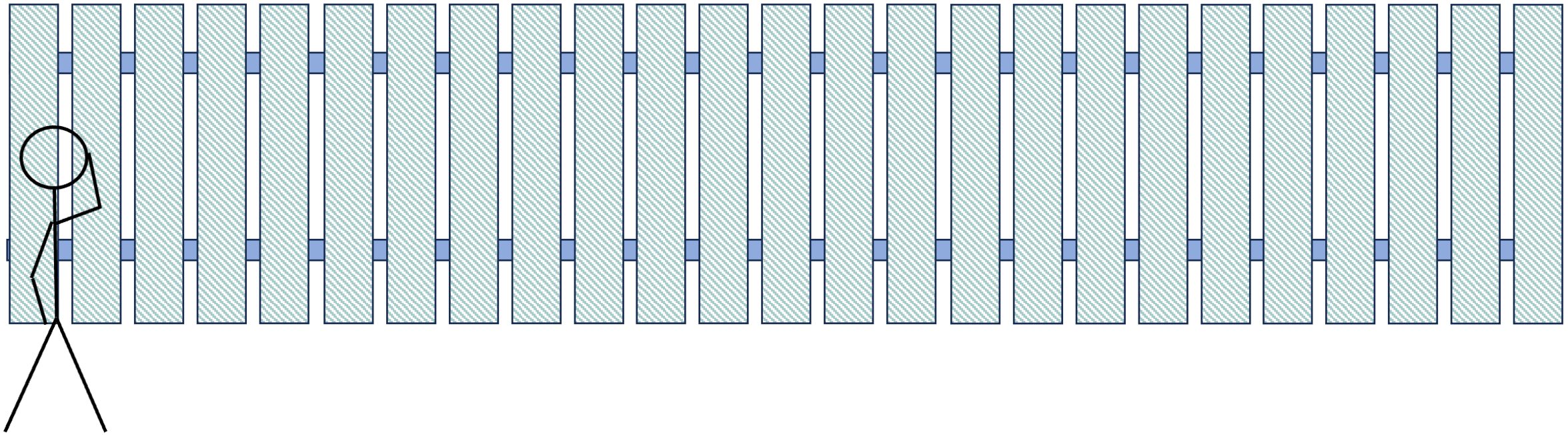
**To understand them more deeply, we need to look into computer architecture**

**But first a brief digression …**

**What do we mean by the word *parallelism***

# Painting a fence

- You must paint a fence. It has 25 planks.

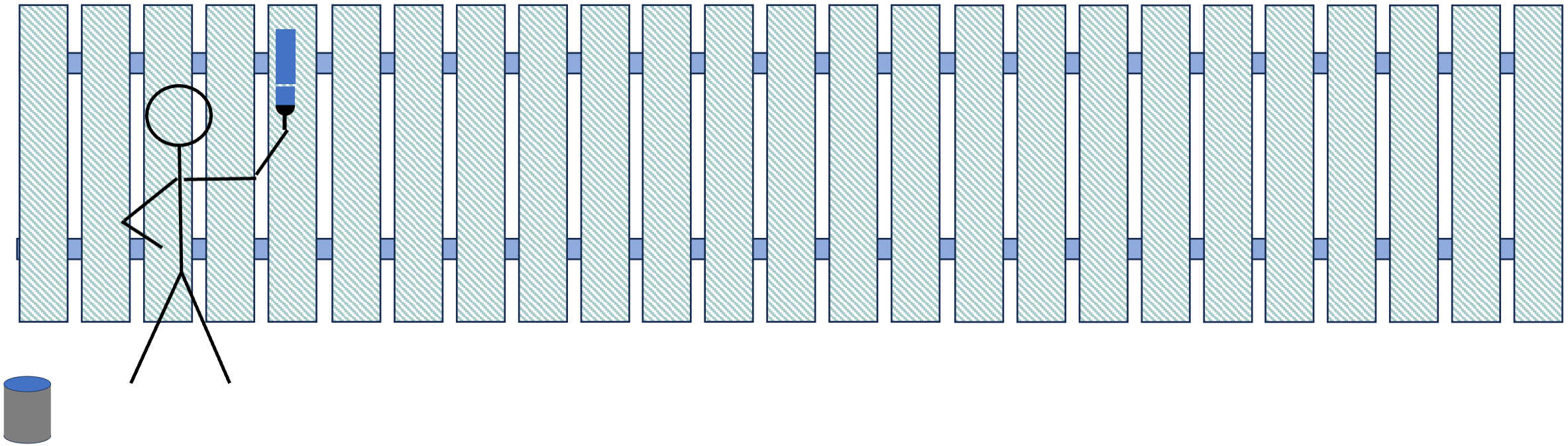


# Painting a fence:  Doing it as a serial process

- You must paint a fence. It has 25 planks. You are going to do this one plank at a time (**a serial process**)
- How long should it take?

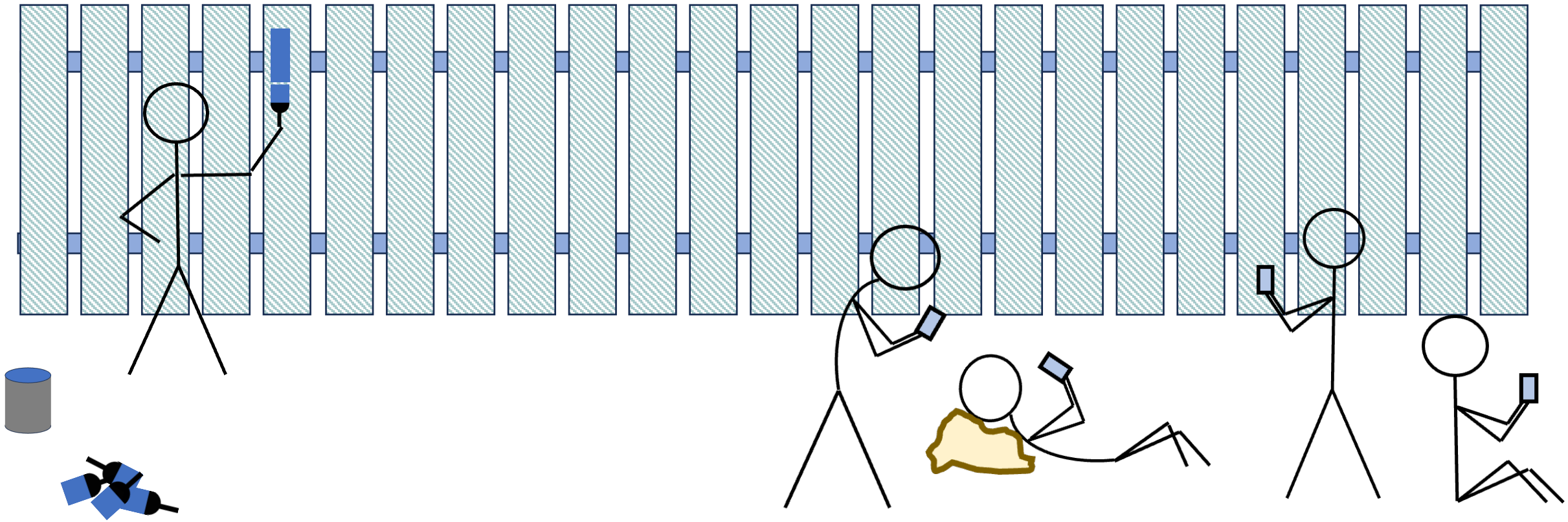
**Ideal case**

It takes $T_p$ minutes to paint a plank.  Therefore it takes $25*T_p$ minutes to paint the entire fence.



# Painting a fence:  Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take with the five of you working **in parallel**?

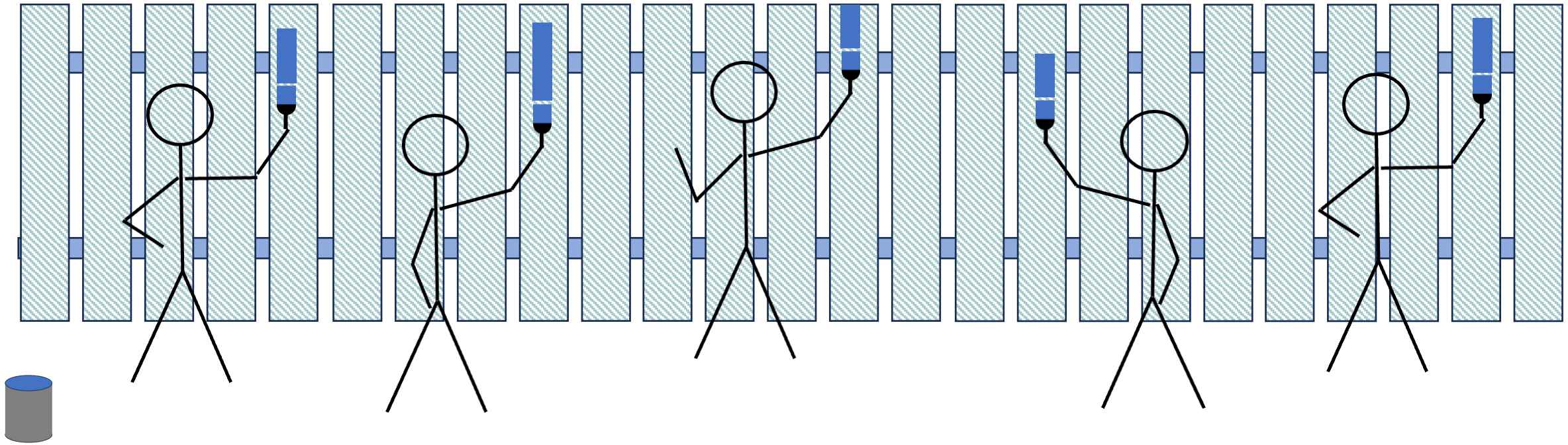


# Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

**Ideal case**

It takes $25*T_p$ minutes to paint the fence on your own ("**in serial**"). With N = 5 people each taking an equal chunk of the fence, then each person paints 25/N planks. If they do it all at the same time (that is, "**in parallel**"), the fence will be done in $(25*T_p)/N = 5*Tp$ minutes



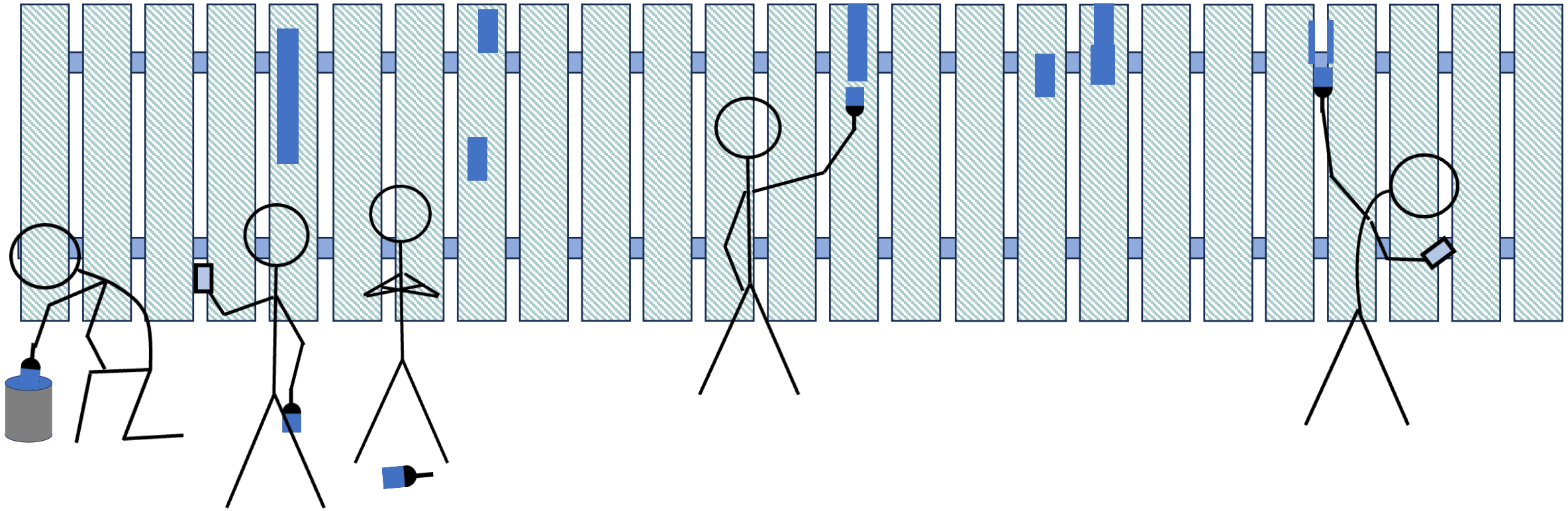
**Speedup**: How many times faster does your job complete when N people work in parallel? Ideally, Speedup = N.

# Painting a fence:  Sometimes, Reality Sucks

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

**Reality**

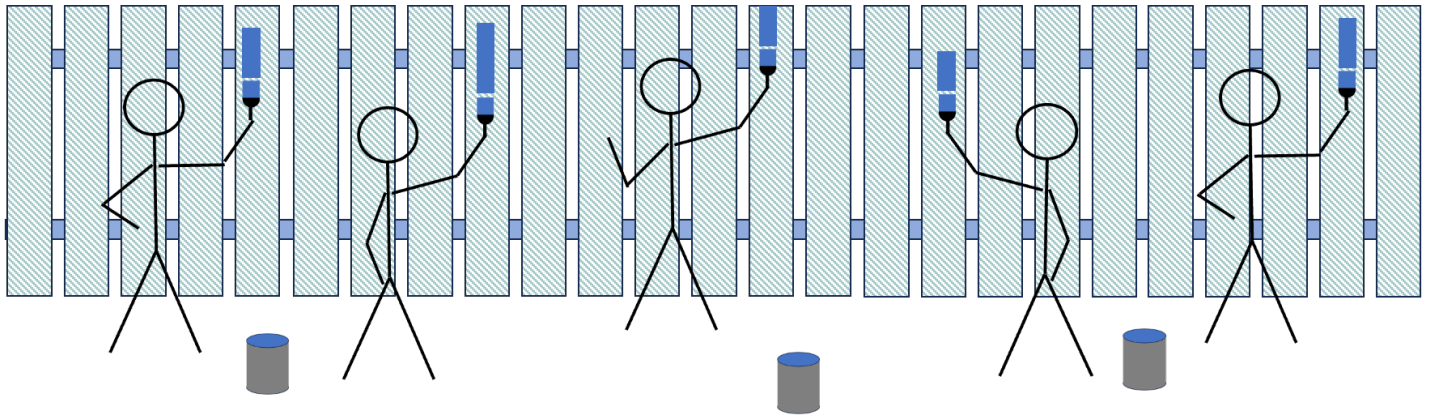
Not everyone works equally hard.  More importantly with only one can of paint (a **shared resource**) people waste time waiting for their turn to dip their brush in the paint.  These and other issues mean that you almost never achieve the ideal case.



# Summary: the concept of parallelism

- This sequence of definitions explains the term “parallelism”
  - **Agent**: A person, process, thread, or other “unit of execution” that can work on a task.
  - **Problem Decomposition**: Break the problem into a collection of distinct tasks that are mostly (if not completely) independent.
  - **Serial**: When a single agent carries out a problem’s tasks one after the other.
  - **Parallel**: When the tasks execute and make forward progress at the same time.
  - **Parallelism**: the features of problem and its solution that support parallel execution.

- Assume you have multiple agents to carry out a set of tasks. You are not done until the last agent is done. It never goes as well as you hope.
  - Making the set of tasks (the work) for each agent **balanced** so they all finish at the same time is hard (**load balancing**).
  - Agents share resources (such as paint) and often waste time waiting for their turn for the shared resource (**contention**).
  - Coordinating the work of the multiple agents is extra work you wouldn’t have if you did the job in serial. This is called **parallel overhead**.
  - Recasting the problem into a collection of distinct and largely independent subproblems (tasks) can be difficult
  - There is almost always a small fraction of the work that cannot be done in parallel (**serial fraction**).
  - The serial fraction limits the number of agents that can productively help you complete the job



# Let’s explore the key ideas in computer architecture

# Computer Architecture: Computer attributes visible to a user

- Computer architecture is composed of 3 topics

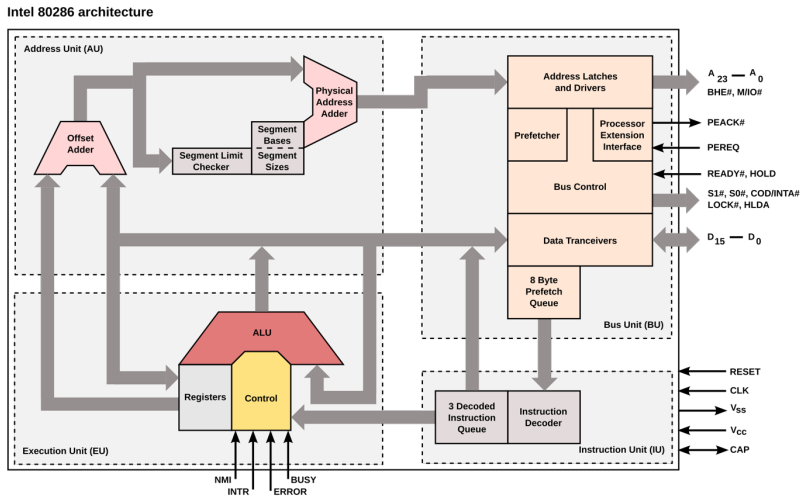
1. Instruction set architecture (ISA): the interface to the computer presented to a programmer

2. The microarchitecture: A design for how the ISA is implemented

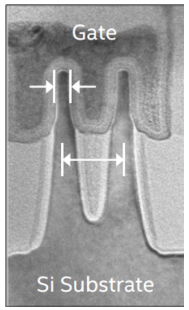
3. The hardware: The implementation of the computer itself (largely the silicon implementation)

```
12  .L3:
13          fcvt.d.w        fa5,a5
14          fcvt.d.s        fa4,fa4
15          addi    a5,a5,1
16          fadd.d  fa5,fa5,ft0
17          fmul.d  fa5,fa5,fa3
18          fcvt.s.d        fa5,fa5
19          fmul.s  fa5,fa5,fa5
20          fcvt.d.s        fa5,fa5
21          fadd.d  fa5,fa5,fa1
22          fdiv.d  fa5,fa2,fa5
23          fadd.d  fa5,fa5,fa4
24          fcvt.s.d        fa4,fa5
25          bne     a0,a5,.L3
26          fmul.s  fa0,fa0,fa4
27          ret
```

RISC-V assembly code for the “pi program loop” generated by gcc -O3




Intel 80286 microarchitecture




Electron microscope image of an Intel transistor for the 14 nm process technology

Transistor image source: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf
80286 image source: By Appaloosa - https://commons.wikimedia.org/w/index.php?curid=6902962
Assembly code from the compiler explorer at https://godbolt.org/

# Instruction Set Architecture (ISA)

- The essence of computer architecture is the instruction set architecture (ISA), which is the interface to the hardware presented to a programmer … it is how software talks to hardware.
- Two major classes of ISA
  - CISC: **Complex instruction set Computer**. Large set of instructions to cover numerous special cases. Example: Intel® x86 ISA
  - RISC: **Reduced instruction set Computer**. Smaller set of instructions, easier to work with and implement. Example: ARMv8

| ISA features | Intel x86* (CISC) | ARMv8 (RISC) |
|---|---|---|
| Class of ISA | Register memory ISA … operations can reference registers or memory. | Load-Store… can only access memory through load-store operations. |
| Memory address | Bytes addressing | Byte addressing, but objects must be aligned<br>An object of size s bytes is aligned if A mod s = 0. |
| Registers exposed in architecture definition | 16 general purpose and 16 floating point | 31 general purpose<br>32 floating point registers |
| Encoding an ISA … Instruction widths | Variable length, ranging from 1 to 18 bytes. Can result smaller executables. | Fixed length, 4 byte<br>Thumb instructions: 2-byte |
| Number of instructions | Exact count is difficult … over 3500 | Base = 354, SIMD/FP = 404, SVE = 508 … total ~1266 |

ISA details are challenging to nail down. The Intel ISA manual is over 5000 pages.
Hence numbers on this slide convey a general sense of size and miss many details and special cases.

* these numbers are for the Intel® 64 x86 ISA. * SIMD: single instruction multiple data or vector instructions. FP: floating point SVE: Scalable vector instructions

# Instruction sets: Complex (CISC) vs Reduced (RISC)

```
1  void add_abc(double *a, double *b, double *c)
2  {
3      *c = *a + *b;
4  }
```

Compare assembly code for a simple function for CISC (x86-64) and RISC (ARM) processors

COMPILER EXPLORER

https://godbolt.org/

**CISC**

Ops work on registers and addresses in memory.

Complex but extra options for aggressive optimization

x86-64 gcc 14.2 -O2

A Output... Filter... Libraries Overrides

```
1  add_abc(double*, double*, double*):
2          movsd   xmm0, QWORD PTR [rdi]
3          addsd   xmm0, QWORD PTR [rsi]
4          movsd   QWORD PTR [rdx], xmm0
5          ret
```

- Load double at address [rdi] into register xmm0
- Add double at address [rsi] to xmm0, put result in xmm0
- Store double in xmm0 to address [rdx]
- Branch to return address on the stack

**RISC**

All ops on registers

Consistency means smaller and simpler instruction set

ARM GCC 14.2.0 -O2

A Output... Filter... Libraries Overrides

```
1  add_abc(double*, double*, double*):
2          vldr.64 d16, [r0]
3          vldr.64 d17, [r1]
4          vadd.f64        d16, d16, d17
5          vstr.64 d16, [r2]
6          bx      lr
```

- Load double at address [r0] into register d16
- Load double at address [r1] into register d17
- Add double in d17 to double in d16, put result in d16
- Store double in d16 to address [r2]
- Branch to return address lr

# Computer Architecture: CISC vs RISC

- Intel pioneered mass-market-computing through the IBM PC.

- Intel's x86 ISA is a CISC instruction set and that played a key role in the history of computing.
  - Starting with the Intel 8086 CPU in 1978 and continuing to today as the dominant architecture for servers and laptops.

- ARM: the dominant commercial RISC vendor starting with the ARM1.
  - ARM1, 1985, 25 thousand transistors compared to Intel's 1985 CPU (i386) with 275 thousand transistors.

- **Every new CPU ISA since 1980 has been based on a RISC ISA**.
  - As we'll see later, internal to a modern CISC CPU from Intel is a RISC execution engine.

The "golden handcuffs" of legacy applications will keep CICS/x86 around for many years. But in terms of innovative designs and the future, **RISC has "won"**.

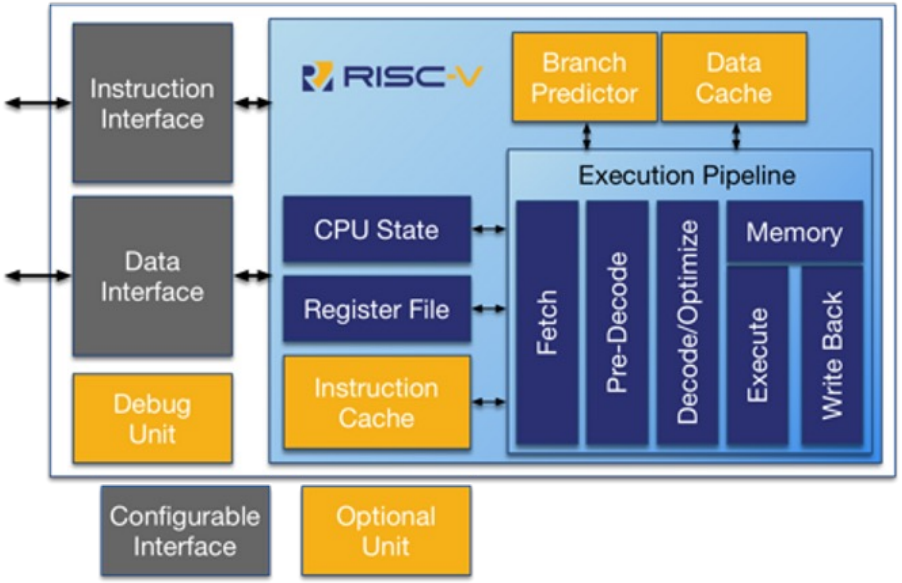
# RISC across the computer industry

- ARM licenses CPU designs for others to implement.
  - Used extensively in cell-phones, tablets, Apple laptops, embedded processors, and other devices. The number one CPU by volume.
  - ARM is moving into Servers and HPC … For example, Nvidia is shipping chips for HPC using ARM (Nvidia® Hopper™)
  - ARM charges a royalty for each unit sold and vigorously protects it's monopoly over the ISA.

- Just as Open Source Software changed the nature of the software industry, an Open Source ISA will change the hardware industry.

- RISC-V (pronounced RISK-Five) is an open source ISA.
  - 2010: research project in the Computer Science Department at the University of California, Berkeley.
  - 2011: The first RISC-V specifications were released.
  - 2015: RISC-V International was established to promote adoption and standardization of the RISC-V ISA. Now has over 200 members.

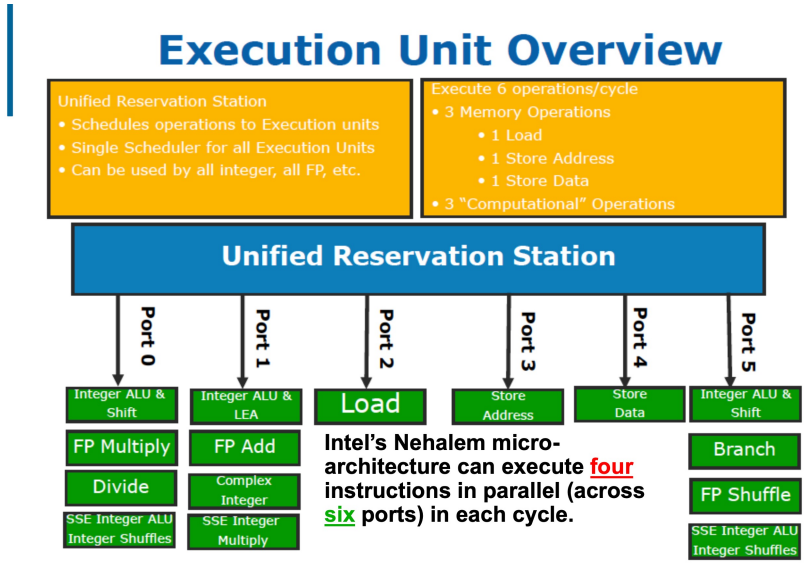
RISC-V block diagram




- Load-store ISA
- 32 bit instruction format
- RISC-V base ISA has only 50 instructions compared to 354 for ARM8 base ISA

Big companies like Apple and Google will tire of paying royalties per unit to ARM. The future is RISC-V

# Microarchitecture … the details for how an architecture is implemented



# A modern CPU

# Intel® X86 architecture

# Sandy Bridge Microarchitecture

By the time we are done, **most** of this will make sense to you.

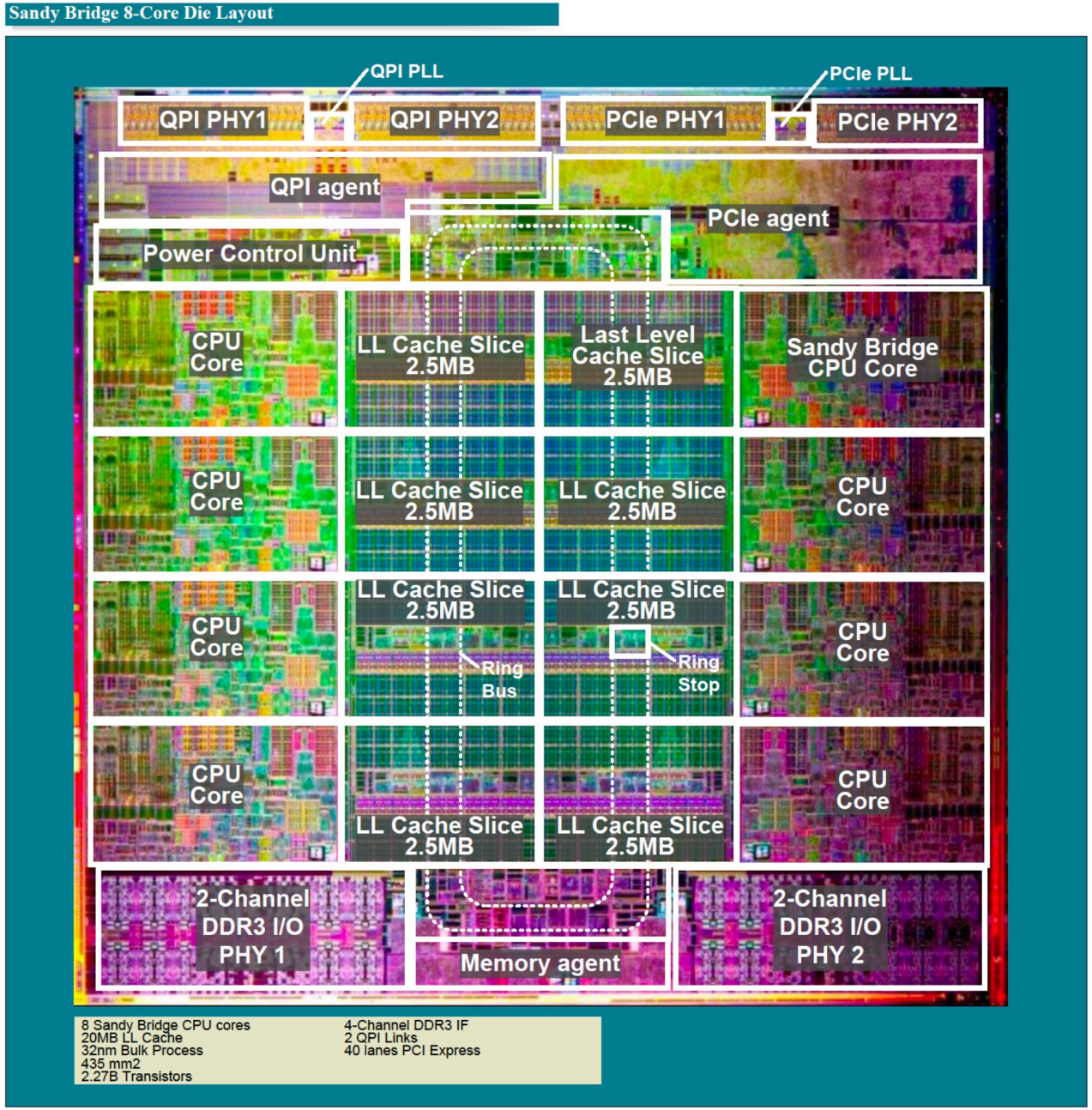


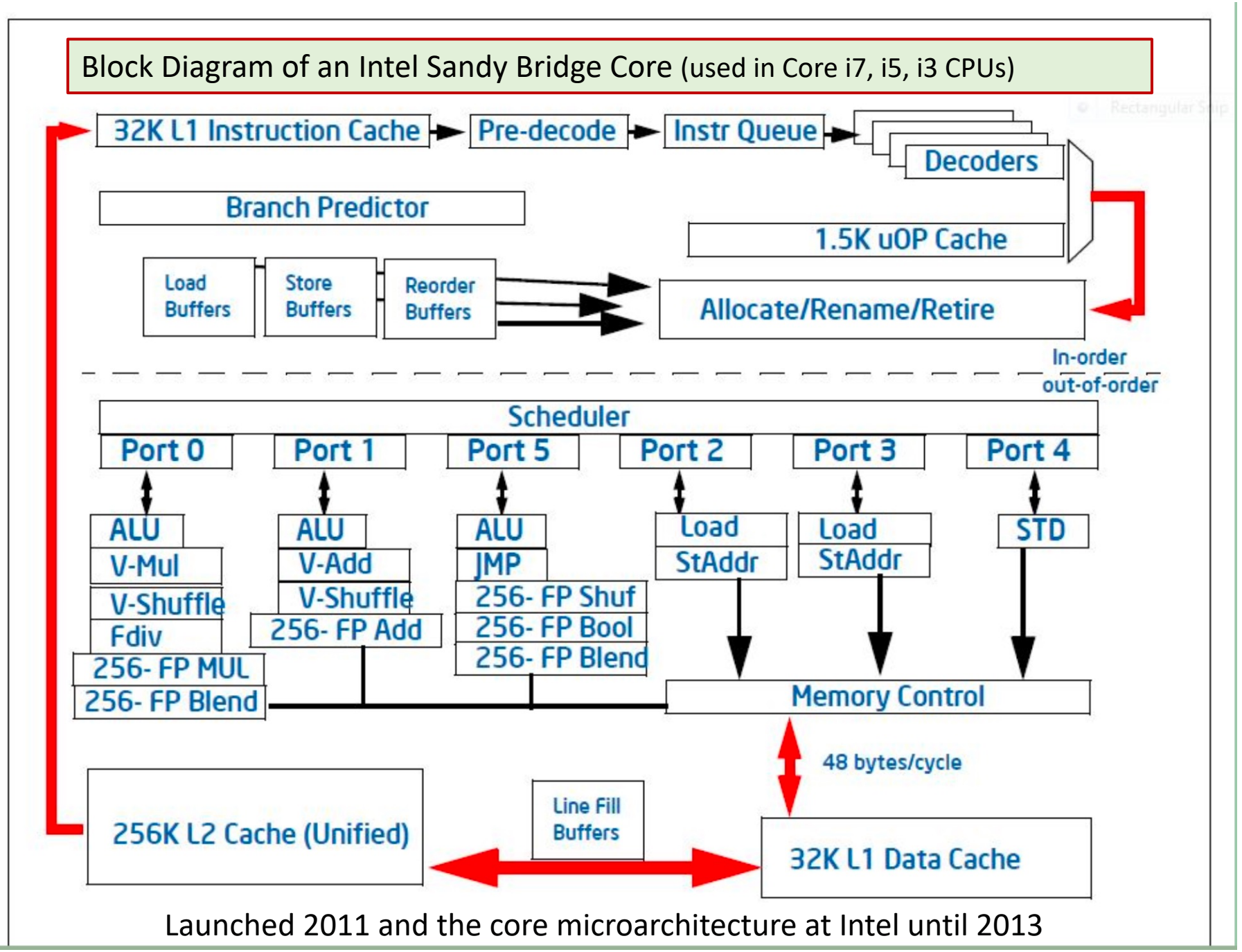



Image Source: Victor Eijkhout, processor architecture lecture, 2023

# Structure of a Sandy Bridge CPU core

- The point of a microarchitecture is to support the architecture with aggressive optimization to achieve high performance.
- A modern microarchitecture can be extremely complex … both for CISC and RISC chips

By the time we are done, **much** of this will make sense to you.




Launched 2011 and the core microarchitecture at Intel until 2013

# The key to performance inside a CPU: Instruction level parallelism (ILP)

- The fundamental equation of quantitative architecture analysis:

$$Time_{CPU} = N_{instructions} * \frac{cycles}{Instruction} * \frac{seconds}{cycle}$$

$N_{instructions}$ ≡ Number of instructions in an executable

$$\frac{cycles}{Instruction} = \frac{Total\ number\ cycles\ to\ execute\ a\ program}{N_{instructions}} = \text{CPI}$$

- An architecture that lets multiple instructions make forward progress each cycle reduces the Cycles per Instruction (CPI) … if all goes well, we can design architectures where CPI < 1.

- We do this with Instruction Level Parallelism (ILP)

- Main ways to implement ILP in a design:
  - Pipelined execution: overlap execution across multiple stages
  - Superscalar execution: multiple-issue of independent instructions
  - branch prediction, speculative execution, prefetching
  - out-of-order execution

# Let's go back to the late 80's and 90's to look at Instruction level parallelism through the lens of x86 CPUs

X86 … "an architecture that is difficult to explain and impossible to love"

Hennessy and Patterson, 2nd ed, page D-2

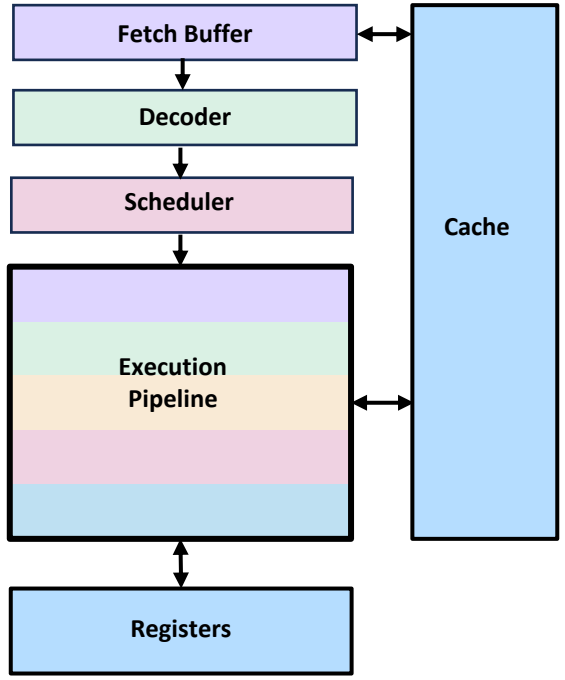
While we focus on x86 chips, all the techniques we'll discuss are used in RISC chips as well

# Pipelining

- Intel added pipelined instructions to i486 in 1989
- More than doubled the performance compared to a i386 at the same clock rate.
- The five stage i4586 pipeline, one cycle per stage

| | |
|---|---|
| F | Fetch an instruction from the instruction cache. |
| D1 | Decode the instruction. |
| D2 | Translate memory addresses and displacements for the instruction |
| EX | Execute the instruction. |
| WB | Retire the instruction, write results back to registers and/or memory. |

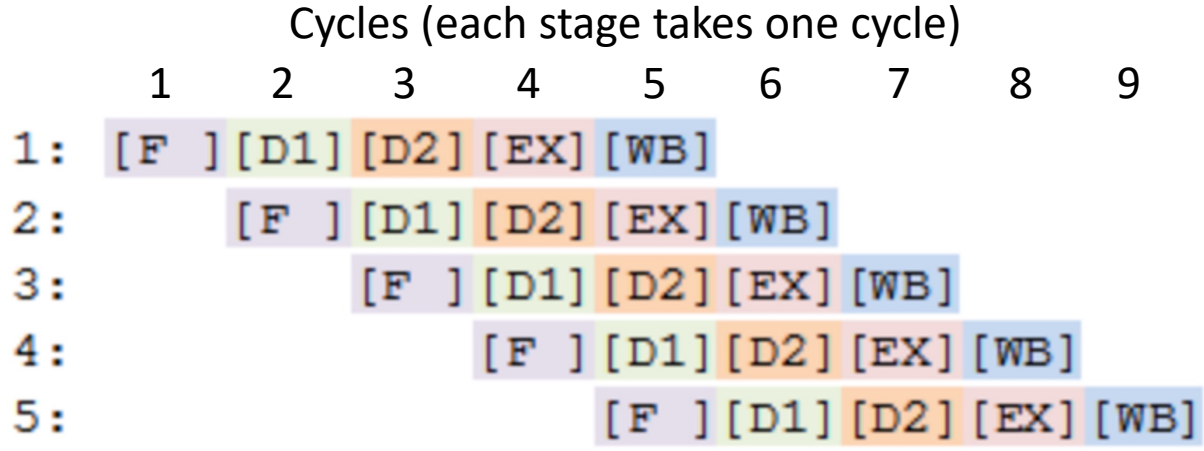
i486™ 1.2 Million transistors, 50 Mhz*




… plus an integrated floating point unit

*50 Mhz means 50 million cycles per second

Five instructions going through the pipeline

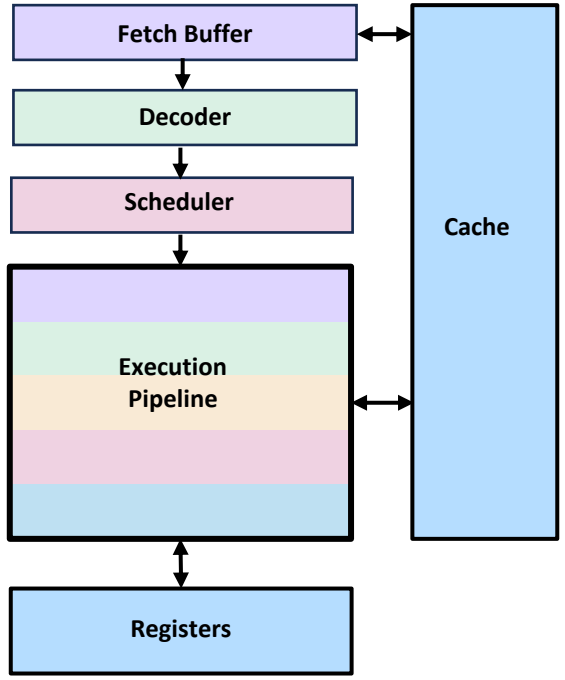



Note: after five cycles, the pipeline is full and we get a result per cycle.

# Pipelining Performance

1989

i486™ 1.2 Million transistors, 50 Mhz*




… plus an integrated floating point unit

*50 Mhz means 50 million cycles per second

- Given the following definitions:

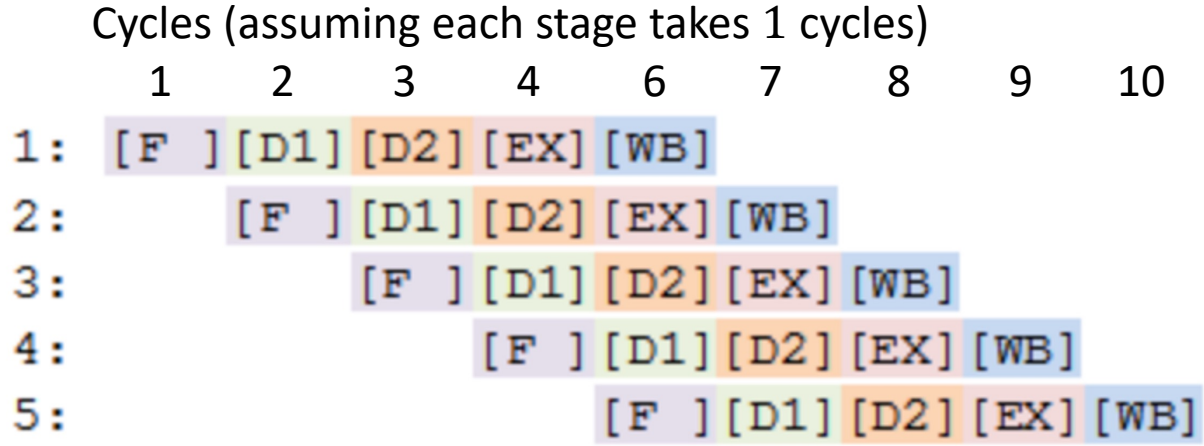
$$n = number\ of\ operations$$
$$\ell = cost\ of\ a\ stage\ in\ cycles$$
$$\tau = number\ of\ stages\ in\ the\ pipeline\ (depth)$$

- Runtime without pipelining: $t(n) = n\ell\tau$
- With pipelining you need to setup the pipeline (costs $s\ell$ cycles) and fill the pipeline (costs $\tau\ell$ cycles) at which point you have completed one instruction. Then for the next ($n$ - 1 ) $\ell$ cycles you get one result per cycle. The runtime with pipelining is:

$$t(n) = (s + \tau + n\text{-}1)\ell$$

- For n much greater than $(s + \tau - 1)$, $t(n) \approx n\ell$ , so the code runs $\tau$ times faster.

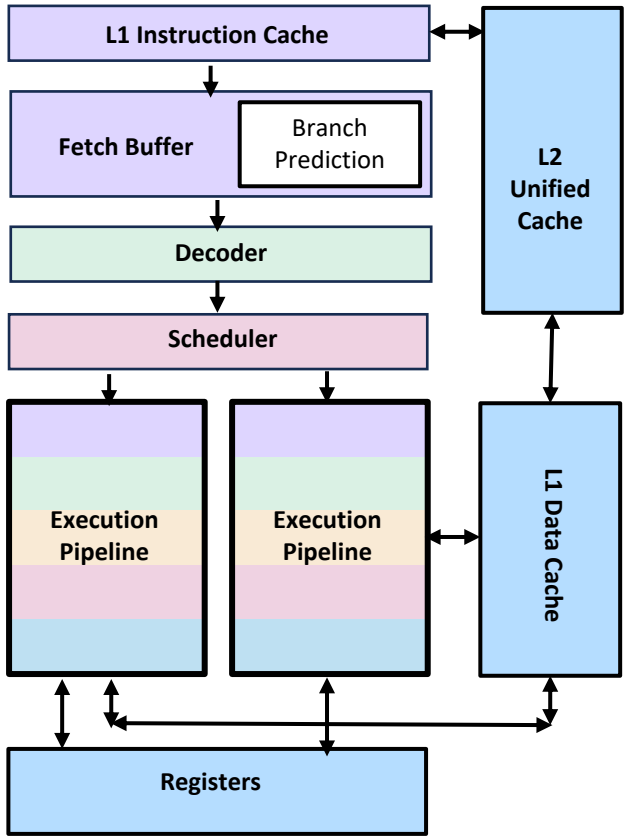
Five instructions going through the pipeline




Note: after five cycles, the pipeline is full and we get a result per cycle.

# Superscaler + branch prediction

Pentium™ 3.1 Million transistors, 66 Mhz, 5 stage pipeline



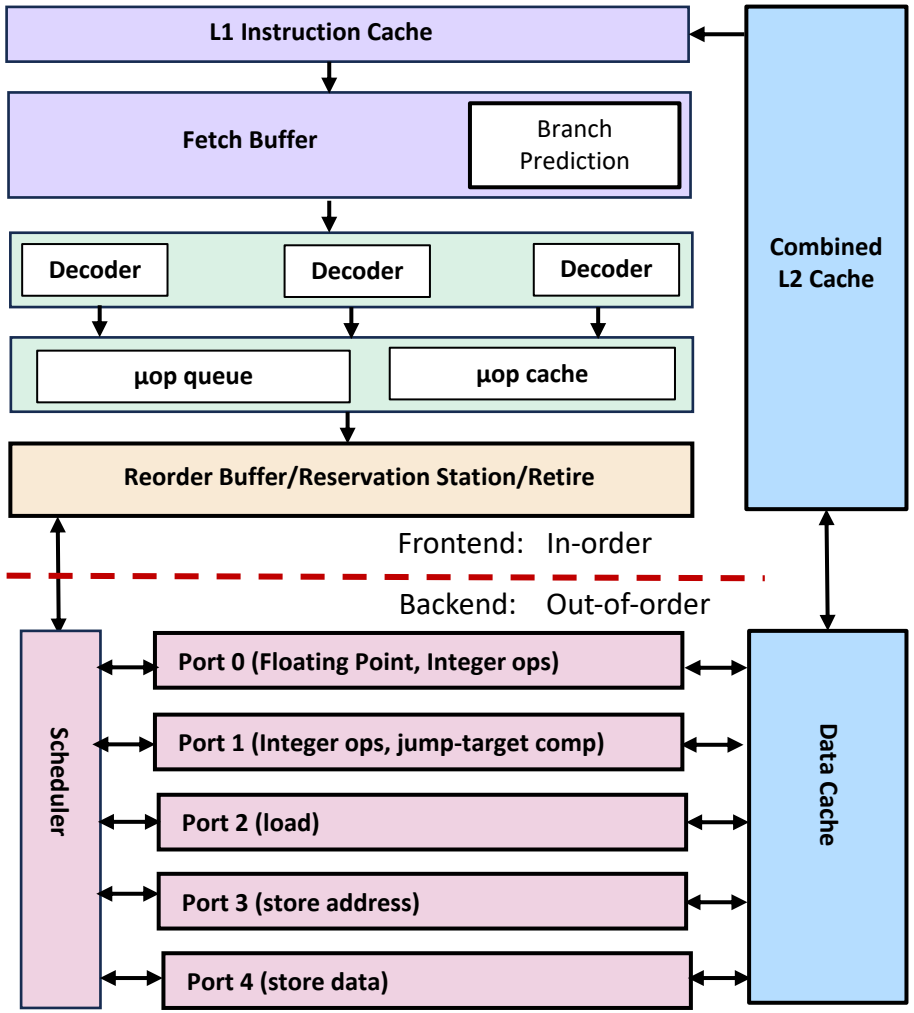

… plus an integrated floating point unit shared between pipelines

- The Pentium CPU was introduced in 1993 with major innovations
  - **Superscalar execution**: Added a second execution pipeline so it could keep two pipelined instruction streams in flight at the same time.
  - **Branch prediction**: Speculate on branches a program might take to load next instructions and get a head start should the branch be taken.
  - **Separate L1 caches for data and instructions**: A unified second level cache that holds data and instructions but separate L1 caches for data and instructions to reduce conflicts.

- Branch prediction is important. With two execution pipelines, its challenging to keep enough work in the execution units so they are fully occupied.

# Out of Order (OOO) + speculation

Pentium Pro CPU, 5.5 Million transistors, 200 MHz,
14 stage pipeline

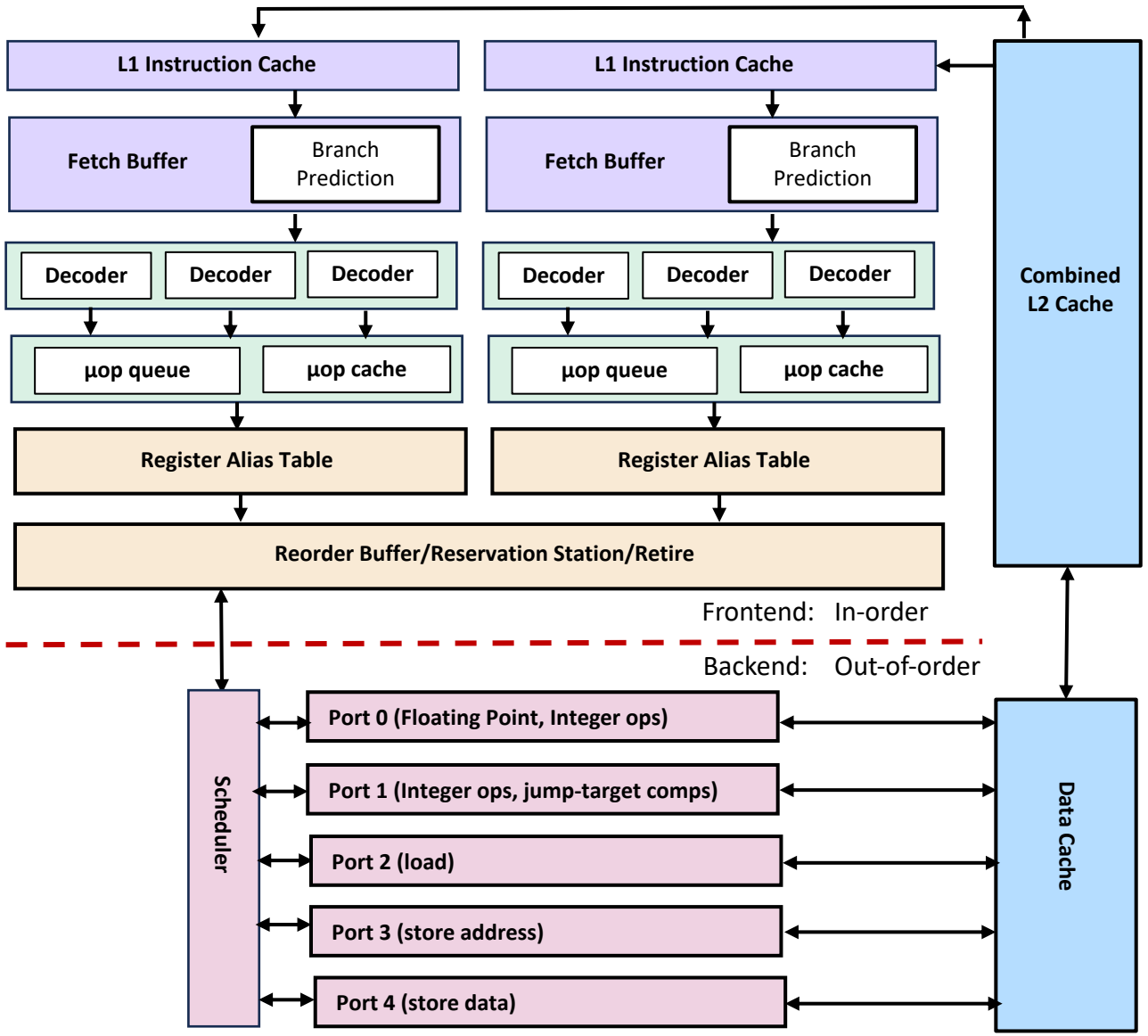



- The Pentium Pro CPU was a major performance upgrade and made Intel CPUs suitable for demanding technical computing workloads (and was used inside the computer ranked as the fastest computer in the world from 1996 to 2000).
- It added the following microarchitectural innovations:
  - Input CISC instructions were decoded into fixed-length, load/store micro-ops (µop). This made the internal execution engine inside the Intel CPU a RISC chip.
  - Micro-ops were reordered and executed based on availability of data … hence compared to input CISC instructions, they execute Out of Order (OOO)
  - Micro-ops complete in-order … i.e., they are retired in an order consistent with the input program
  - Register usage efficiency greatly enhanced by renaming them to avoid spurious conflicts due to register naming in code.
  - Dynamic speculative execution to generate enough work to fully occupy the execution units … a powerful capability but branch misprediction is very expensive as all involved pipelines must be flushed.

# Simultaneous Multithreading … or the Intel Marketing term, hyperthreading

2002

Pentium 4 CPU, 125 Million transistors, 3.06 GHz, 20 stage pipeline




- Out of order execution of pipelined execution units creates so much opportunity for instruction level parallelism that it can be challenging to keep the resources fully occupied.
- Solution … replicate the in-order front end of the processor so two front-ends feed a single out-of-order backend.
- These two in-order front ends are managed as distinct threads by the OS typically with single cycle context switching overhead between them. We call these hardware threads.
- They are usually exposed to the operating system as an additional core … so the case hyperthreading case on this slide results in the OS treating the system having two cores.

Be careful with hyperthreading. If your work load can saturate the functional units on a CPU with a single thread, hyperthreading adds overhead and can slow down your code.

HPC programmers working highly optimized, compute bound codes often turn it off by default.

# ILP is great, but we can get carried away

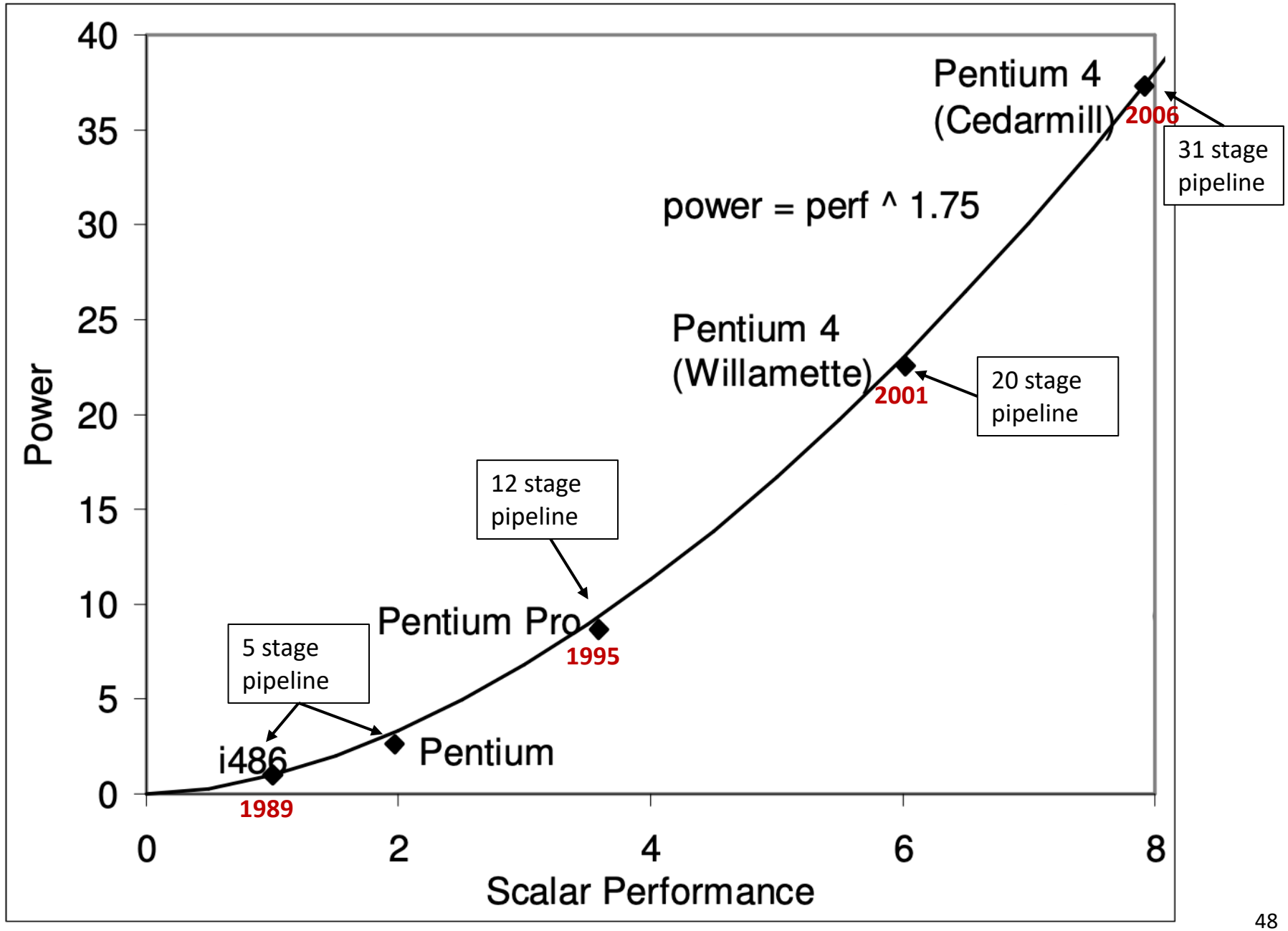
# Normalized Power vs. scalar performance for Intel CPUs

Assume multiple generations of Intel CPUs using the same process technology as for i486.

Any changes are due to microarchitectural enhancements

This shows the unsustainable power demands of every deepening pipelines.

Power and Performance scaled to the i486 … e.g., Petium 4 is 6 times faster than an i486 but uses 22 times more power.




Energy per instruction Trends in Intel® Microprocessors, Ed Grochowski and Murali Annavaram, 2006

# Normalized Power vs. scalar performance for Intel CPUs

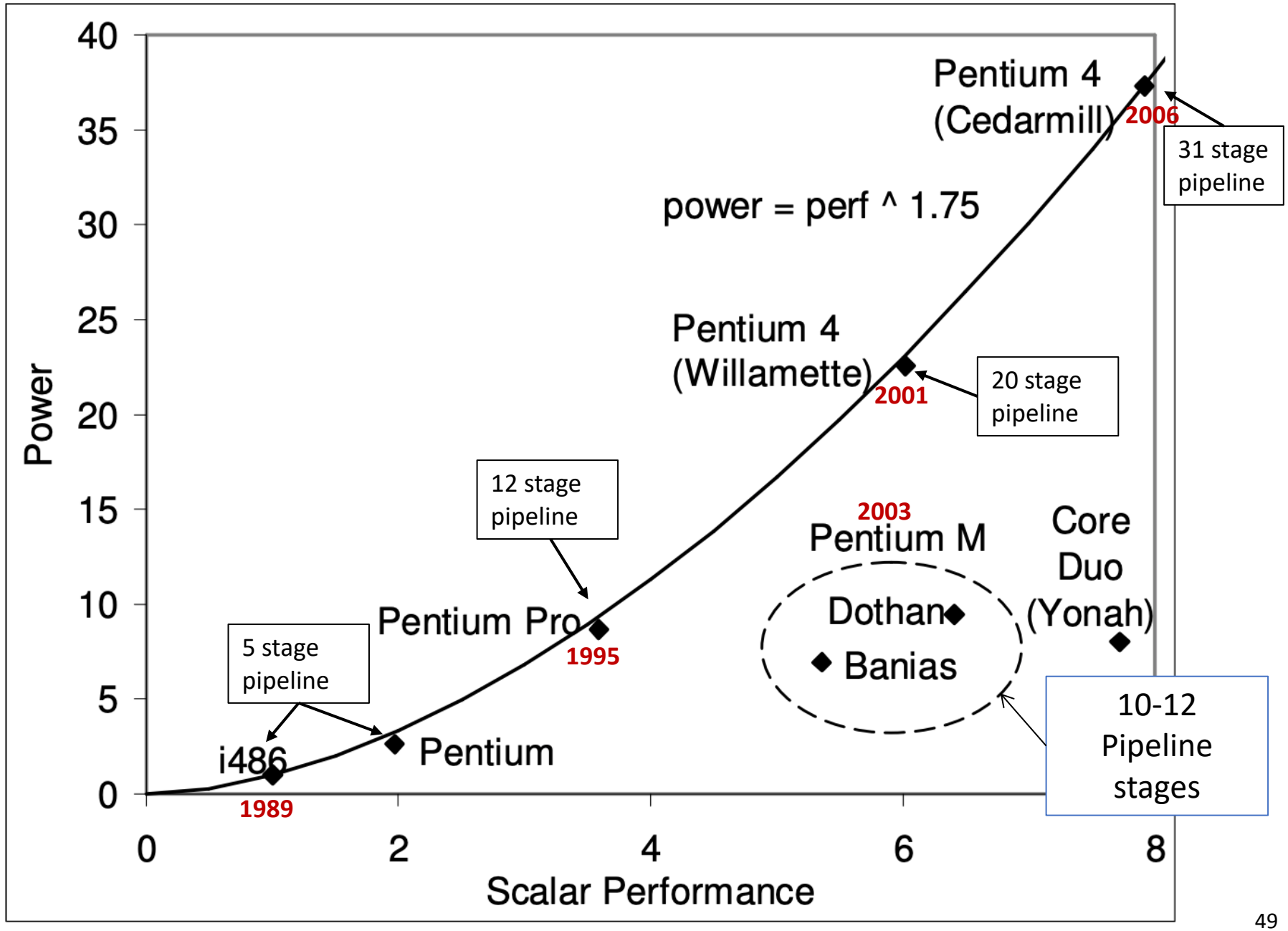
Assume multiple generations of Intel CPUs using the same process technology as for i486.

Any changes are due to microarchitectural enhancements

This shows the unsustainable power demands of every deepening pipelines.

This plot ended around 2006. More modern CPUs have pipeline depths of 14 to 20

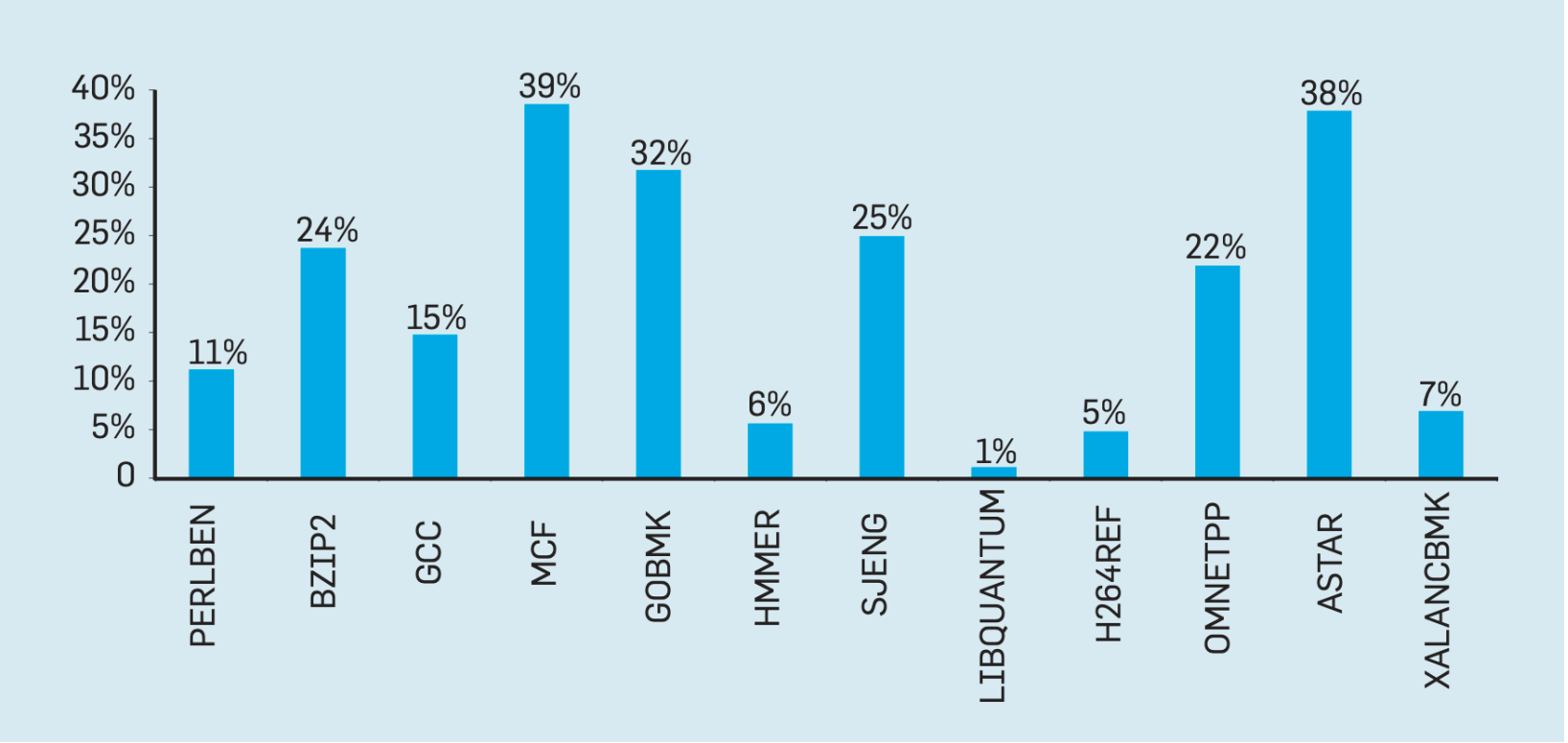
For example the Raptor Cove Intel® Xeon™ microarchitecture (2023) has 12 pipeline. stages




Energy per instruction Trends in Intel® Microprocessors, Ed Grochowski Murali Annavaram, 2006

# Branch prediction

- Percentage of instructions wasted for SPEC integer benchmarks running on an Intel core i7. These are wasted due to incorrect branch predictions.




If you cancel a mispredicted branch, you have to flush the associated pipelines

That's really bad if you have deep pipelines

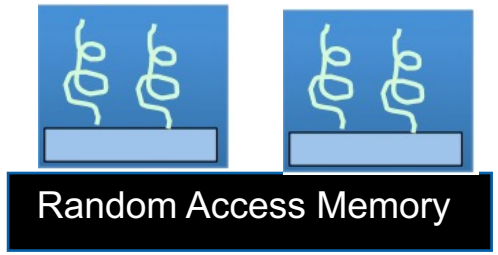
- On average, 19% of the instructions are wasted for these benchmarks on an Intel Core i7. The amount of wasted energy is greater, however, since the processor must use additional energy to restore the state when it speculates incorrectly.

Image from "A New Golden Age for Computer Architecture", John Hennessy and David Patterson, CACM, 2019

# Enough about the processors, what about the memory hierarchy

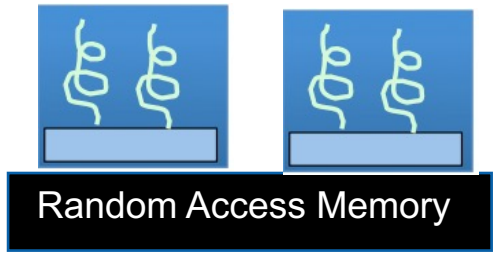
# the Memory Hierarchy

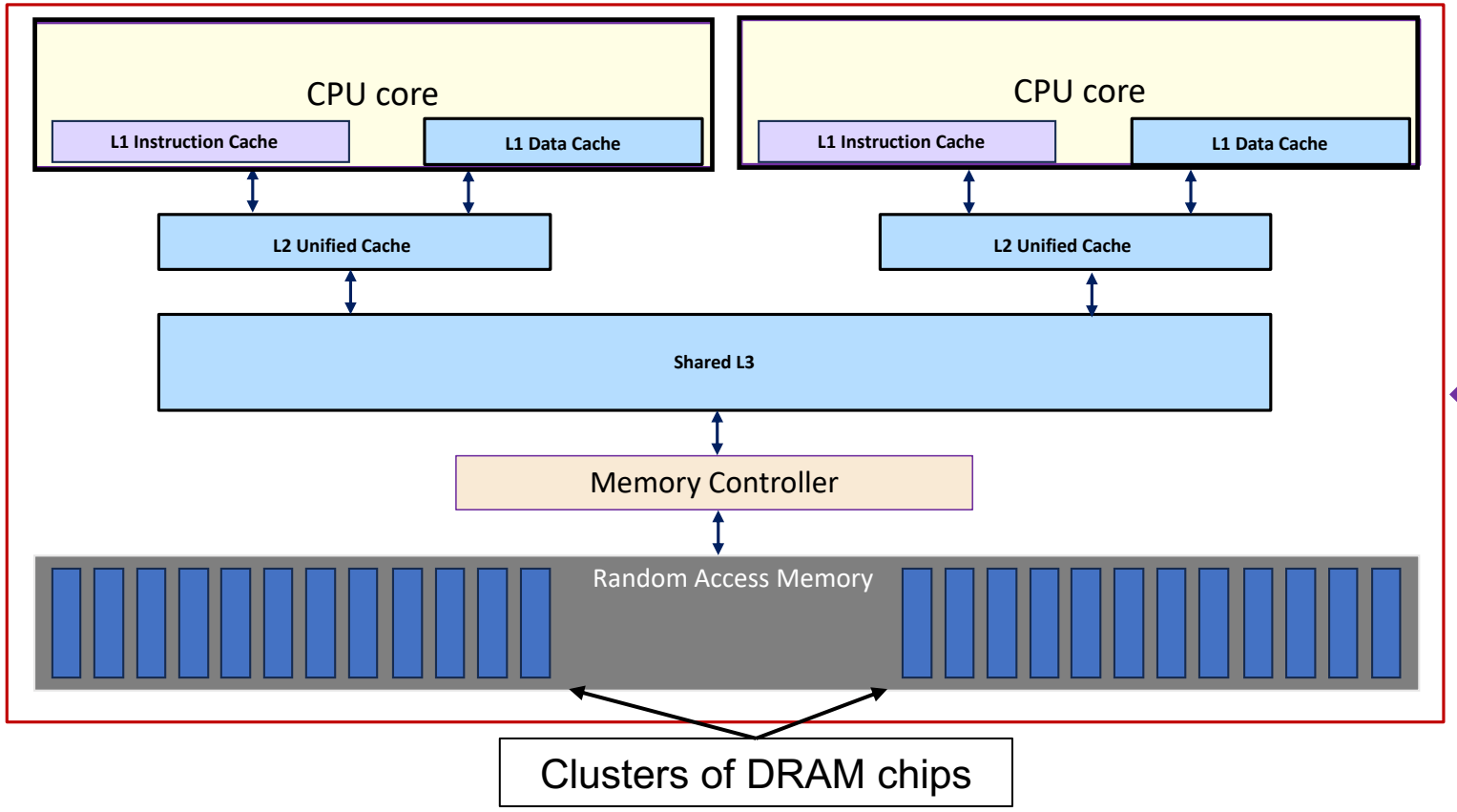
We like to draw pictures like this



# the Memory Hierarchy

We like to draw pictures like this



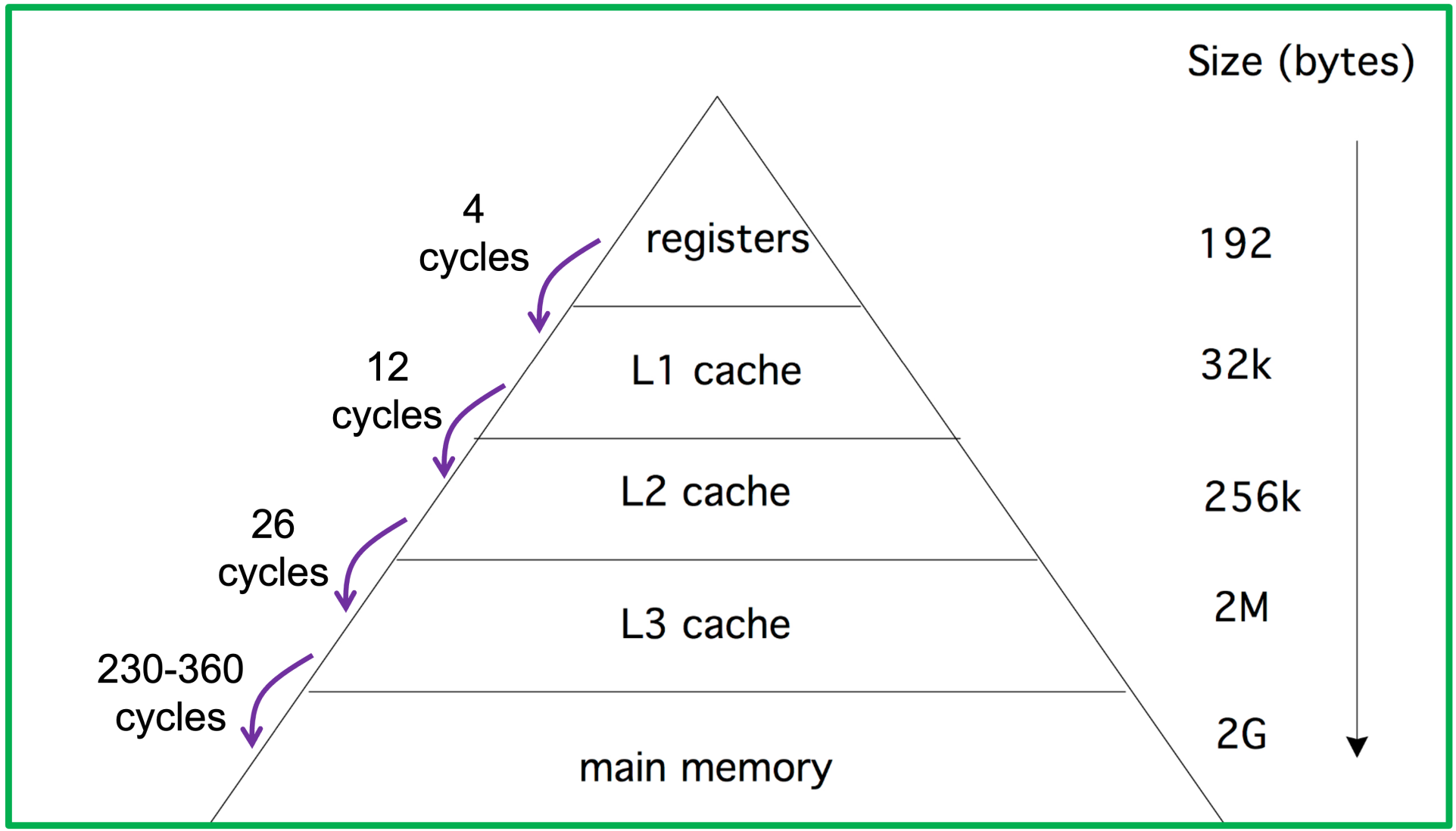




But reality is much more complex

DRAM: Dynamic Random Access Memory

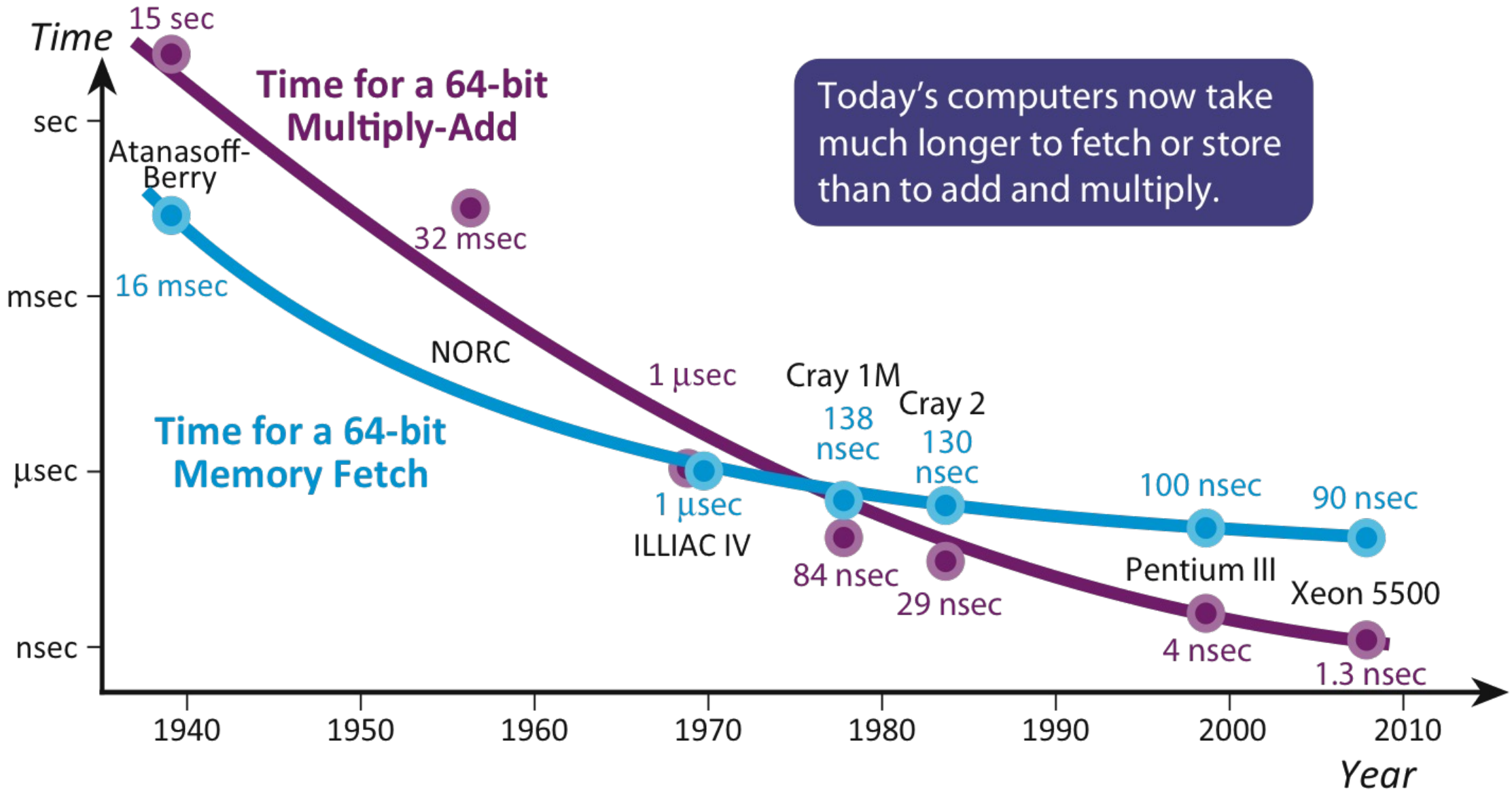
L1, L2, L3 pronounced Level 1, Level 2, Level 3. Also written sometimes as $L1, $L2, $L3

# Latencies across the Memory Hierarchy




Source: Victor Eijkhout – Processor Architecture – Fall'2023

# Latencies across the Memory Hierarchy



# Memory access occurs as Cache Lines

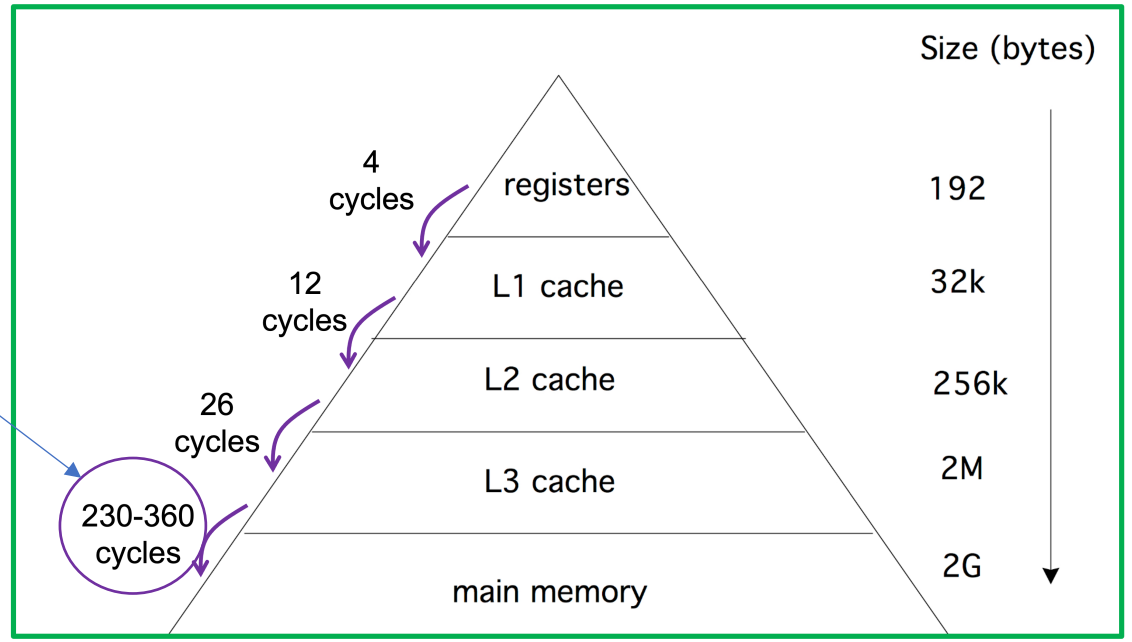
- Load/Store a data element to/from memory as a block of bytes in the level 1 cache ($L1). This is called a ***Cache line***.
- The size of an L1 cache line is a property of the architecture, but it is generally **64** bytes aligned in memory at 64 Bytes.
- Example, load element 17 of an array of doubles. You get the following cache line in the L1 data cache


A 32 KB $L1 holds 512 cache lines

Recall the Latencies across the memory hierarchy

Fetching a cache line all the way from memory costs hundreds of cycles.

Therefore, make sure you do as much as you can with any cache line you access



# Use as much of a cache line as you can per access

- If you only use the one item you requested, you waste a huge amount of bandwidth to memory.


- **Spatial locality**: use all the elements in a line before fetching the next line.
  - Good Spatial locality … C/C++ use a row-major layout for multidimensional arrays, make the last index change the fastest.
  - Its easier to see what's happening if we represent the array through pointers (which is the most common case in C anyway).

```
float matrix[Nrows][Ncols];
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        matrix[i][j] += increment;
```

```
float *matrix;
matrix = (float*) malloc( Nrows*Ncols*sizeof(float);
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        *matrix(i*Nrows+j) += increment;
```

In most modern CPUs, the hardware will try to hide latency by prefetching cache lines before you need them.

  - Terrible Spatial locality … traverse a linked list (pointer chasing). Generate lots of "cache thrashing".

```
while (p != NULL) {
        DoWork(p);
        p = p->next;
}
```

- **Temporal locality**: Try to complete "all" work with data while you know its in the cache.

# Use as much of a cache line as you can per access

- If you only use the one item you requested, you waste a huge amount of bandwidth to memory.

This is our first encounter with the C programming language

The variable **matrix** as an array (**matrix[Nrows][Ncols]**). Elements of the array are **matrix[i][j]**.

This code has pair of nested loops. The loops have loop-control indices (**i** and **j**) which run over a range of index values. For example, for the first loop we start at **i=0**, increment **i** by one at each iteration (**i++)** and keep going as long **i** < **Nrows**. So **i** goes from **0** to **(Nrows-1)**

```
float matrix[Nrows][Ncols];
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        matrix[i][j] += increment;
```

```
float *matrix;
matrix = (float*) malloc( Nrows*Ncols*sizeof(float);
for (int i = 0; i<Nrows; i++)
    for (int j= 0; j<Ncols; j++)
        *matrix(i*Nrows+j) += increment;
```

Experienced C programmers work with arrays through pointers to locations in memory and offsets. We allocate a block of space with a memory allocator (**malloc()**).

The array syntax **matrix[i][j]** is equivalent to the pointer syntax ***matrix(i*Nrows+j)**.

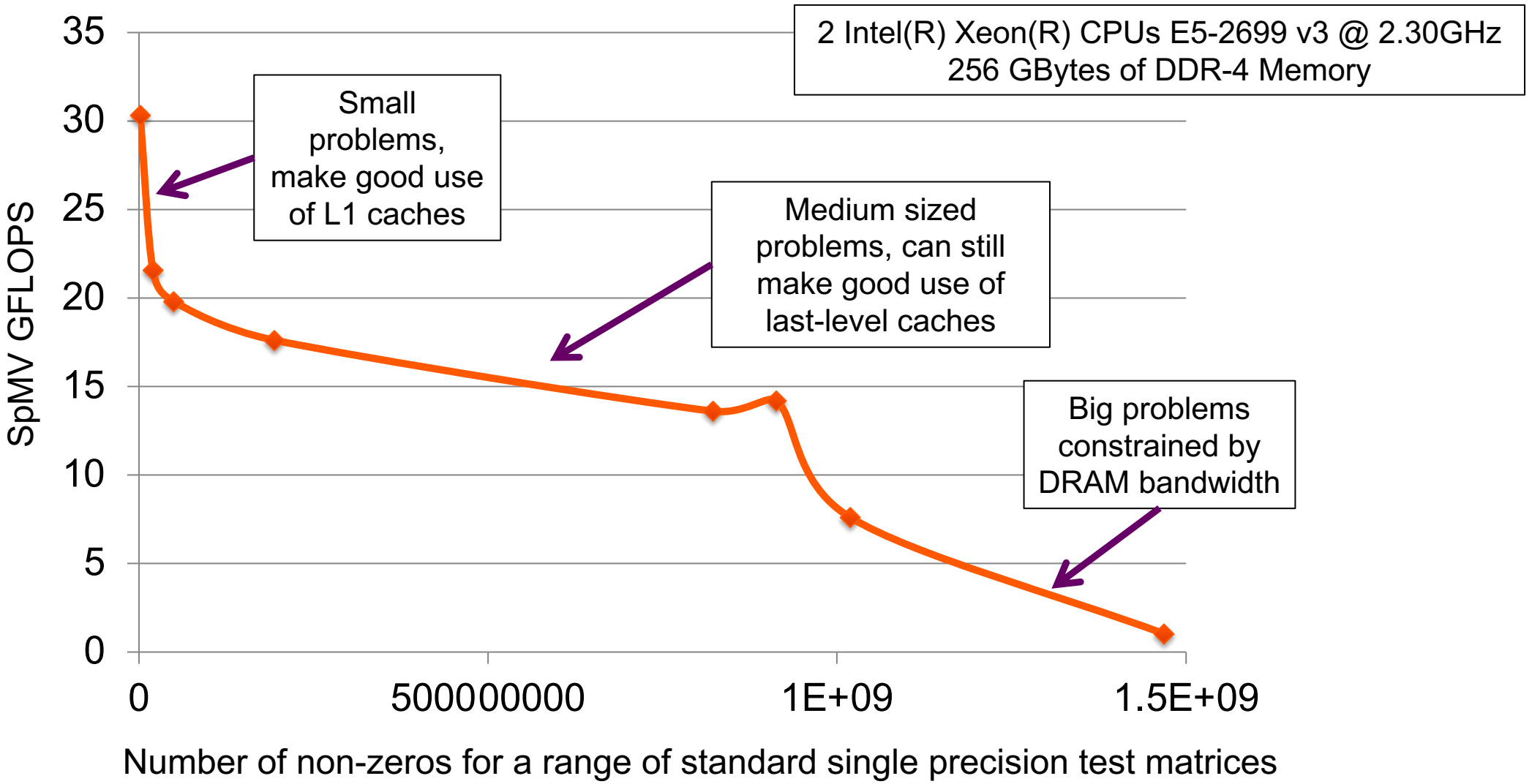
- Terrible Spatial locality … traverse a linked list (pointer chasing). Generate lots of "cache thrashing

This is just another style of loop. The loop continues as long as the loop control condition (**p != NULL**) is true.

```
while (p != NULL) {
        DoWork(p);
        p = p->next;
}
```

- **Temporal locality**: Try to complete "all" work with data while you know its in the

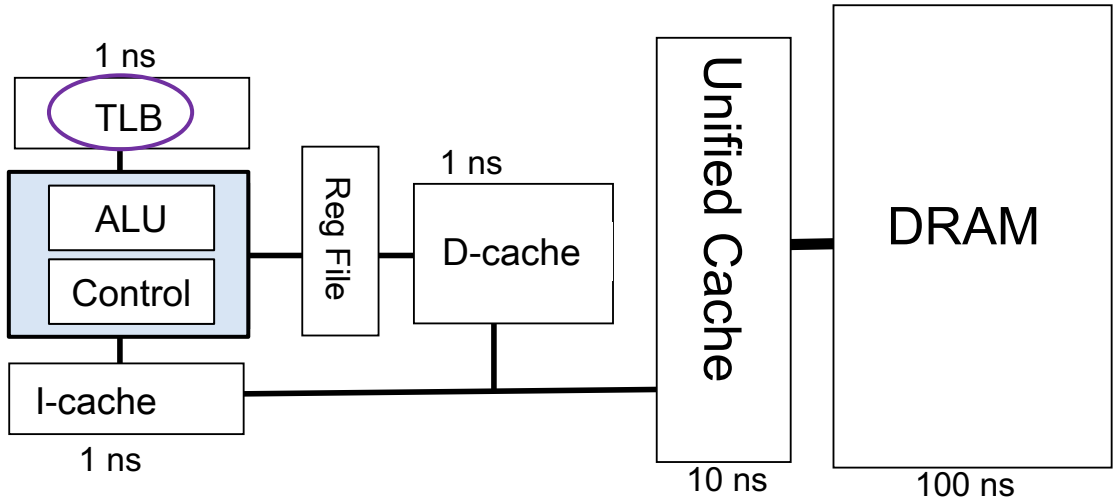
# Sparse matrix vector multiplication (SpMV) is a good way to see the impact of caches across the memory hierarchy

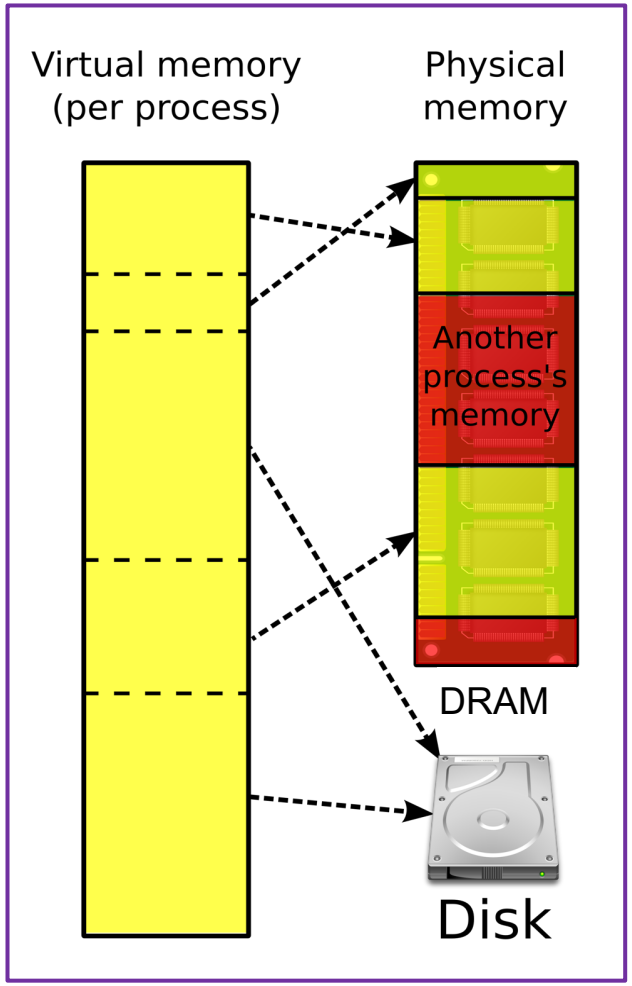



Source: Intel® MKL™ 11.3, “mkl_cspblas_scsrgemv” with KMP_AFFINITY='verbose,granularity=fine,compact,1,0’

# Memory Hierarchies

- A typical microprocessor memory hierarchy (with a simplified cache hierarchy)




- Consider applications with memory demands larger than the physical memory allocated to a process.
- We use virtual memory backed-up by the file system … bringing in pages of physical memory … to handle such problems
- The **TLB (Translation Lookaside Buffer)** implements virtual memory and caches the mapping from virtual addresses to physical addresses
  - If the entry is in the TLB page table, a small overhead of reading the entry and computed the physical address is incurred.
  - If the entry is not in the page table, a page fault exception will be raised. Requires expensive OS operations and stalls the CPU.

# Consider the “simple” Matrix Transpose

- Copy the transpose of A into a second matrix B.

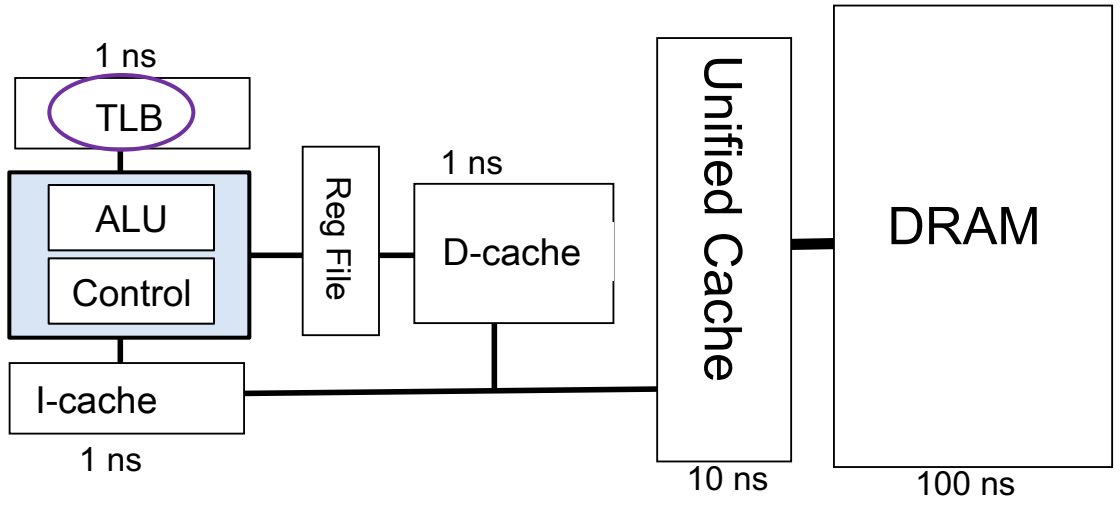
$A \in R^{NxN}$ $B \in R^{NxN}$

```
for (i=0;i<N; i++) {
    for (j=0;j<N;j++) {
        B[i+N*j] = A[j+N*i];
    }
}
```

column j of B

Row i of A

- Consider this operation and how it interacts with the TLB (Translation Lookaside Buffer).




For large N, as you march across addresses of A and B, you span multiple pages in memory … causes multiple page faults

# Optimizing Matrix Transpose for the TLB

- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
    for (it=i; it<MIN(N,i+tile_size); it++){
  for (j=0; j<N; j+=tile_size) {
    for (jt=j; jt<MIN(N,j+tile_size);jt++){
            B[it+N*jt] = A[jt+N*it];
        }
      }
   }
}
```

Step 1: split the “i” and “j” loops into two … loop over tiles of a fixed size

# Optimizing Matrix Transpose for the TLB

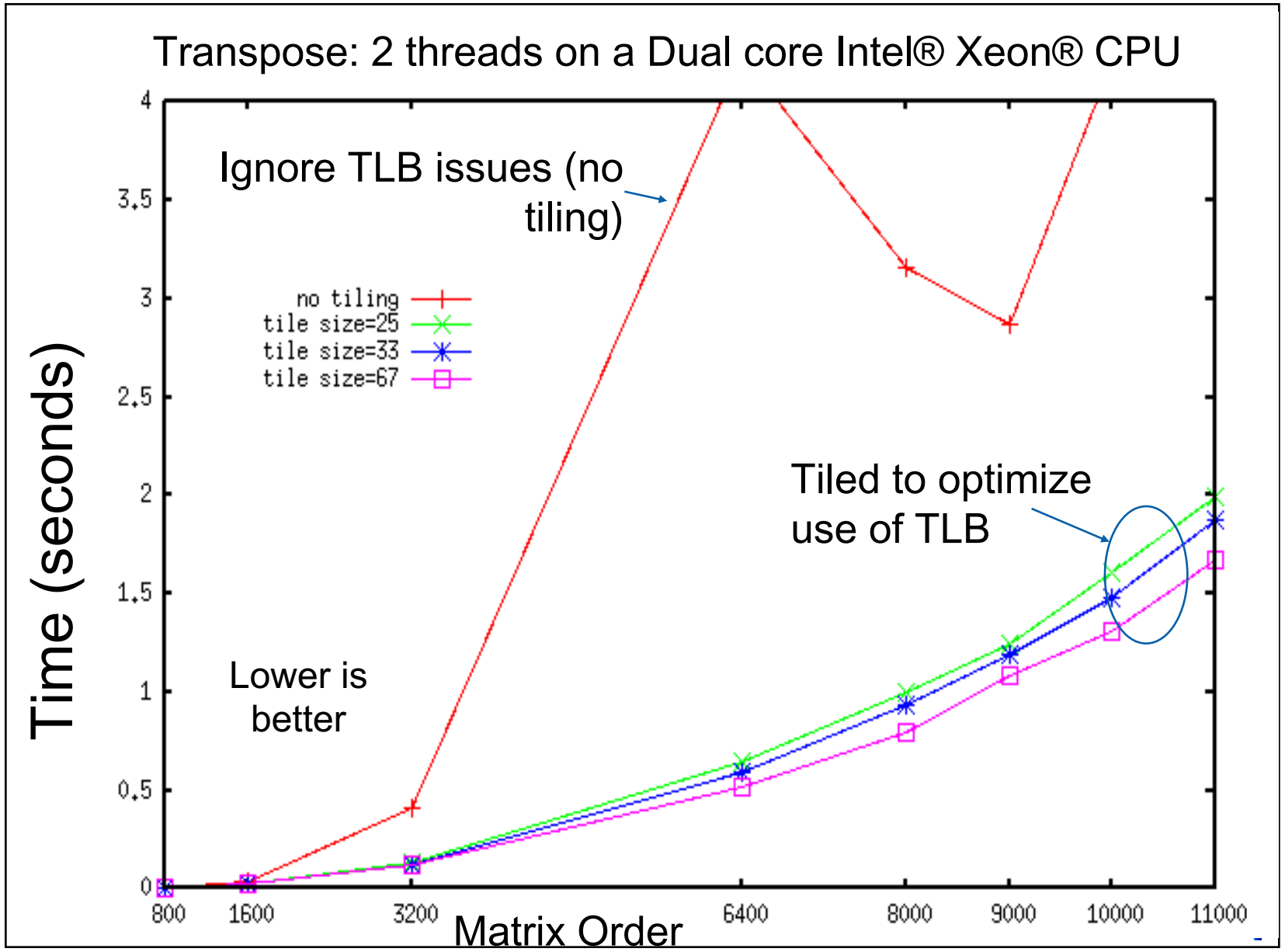
- Solution … break the loops into blocks so we reduce the number of page faults

```
for (i=0; i<N; i+=tile_size) {
  for (j=0; j<N; j+=tile_size) {
      for (it=i; it<MIN(N,i+tile_size); it++){
          for (jt=j; jt<MIN(N,j+tile_size);jt++){
              B[it+N*jt] = A[jt+N*it];
          }
      }
  }
}
```

Step 2: rearrange the loops. Move the loops over the tiles to the innermost loop-nest

The result … you grab a tile, transpose that tile, then go to the next tile

# Do you need to worry about the TLB?




Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel