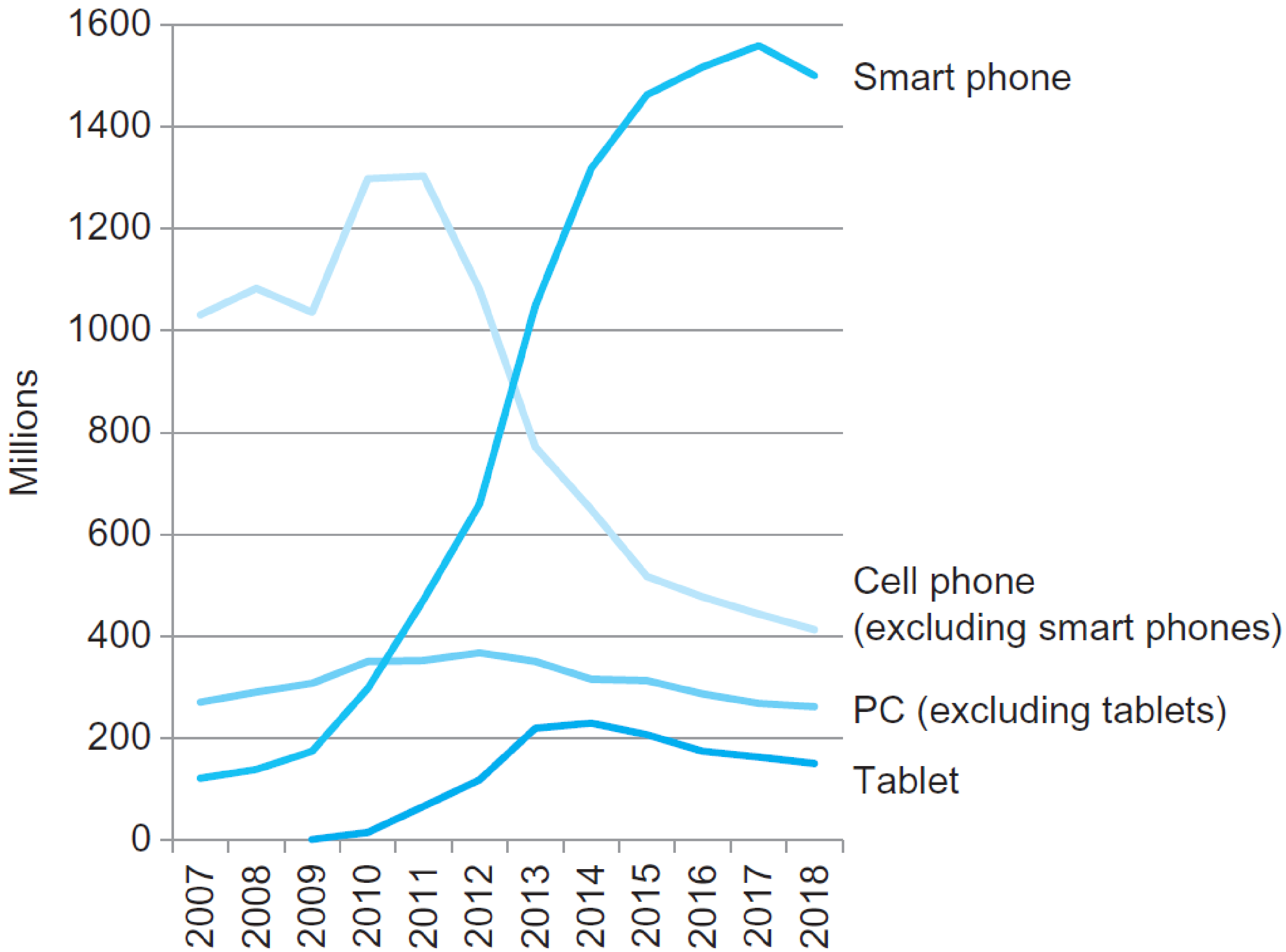
CSE261: Computer Architecture

1. Computer Abstractions and Technology

Seongil Wi

The Post-PC Era



The Post-PC Era

- **Personal Mobile Device (PMD)**
  - Battery operated
  - Connects to the Internet
  - Smart phones, tablets, electronic glasses

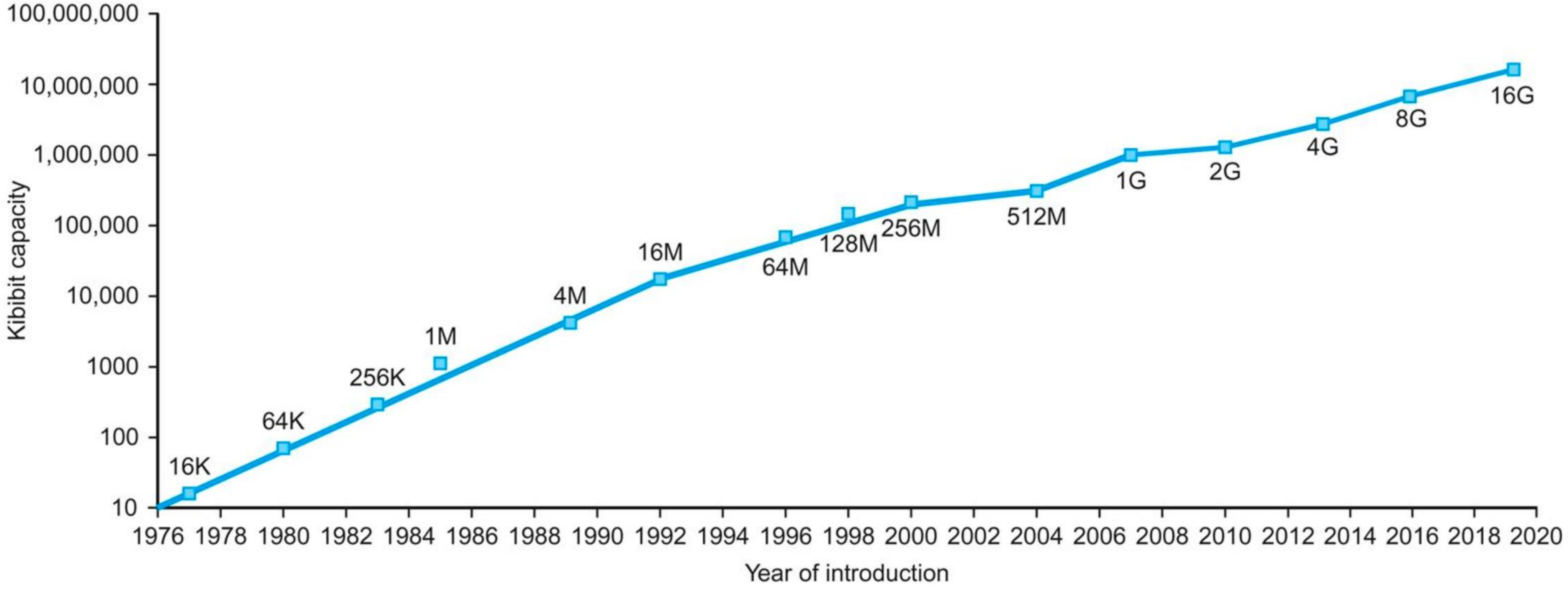
- **Embedded computers**
  - IoT (Internet of Things) and edge devices
  - Hidden as components of systems
    - Car, TV, airplane, robot, drone, surveillance camera, satellite, etc.
  - Stringent power/performance/cost constraints

- **Cloud computing**
  - Amazon, Microsoft, and Google
  - Software as a Service (SaaS)
  - Portion of software run on a PMD or embedded computer and a portion run in the Cloud

Google Drive

Technology Trends

- Electronics technology continues to evolve
  - Increased capacity



Technology Trends

- Electronics technology continues to evolve
  - Increased capacity
  - Increased performance
  - Reduced cost

Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

You will learn fundamental ***principles*** used in modern computer architectures to <u>improve the performance of computations</u>

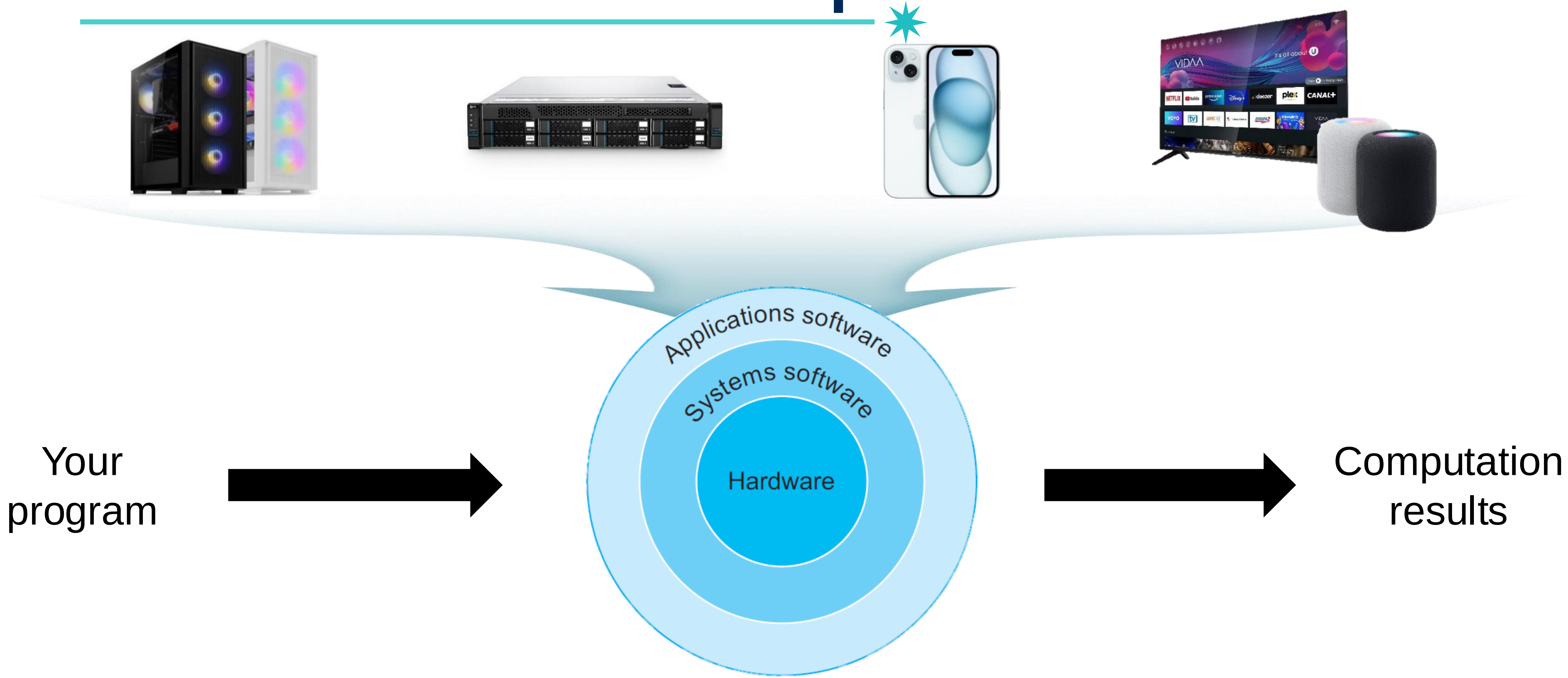
What is the structure of a computer?

Computer Abstractions

Overview of the Computer

Your program → **Computer** → Computation results

Let’s Abstract the Computer



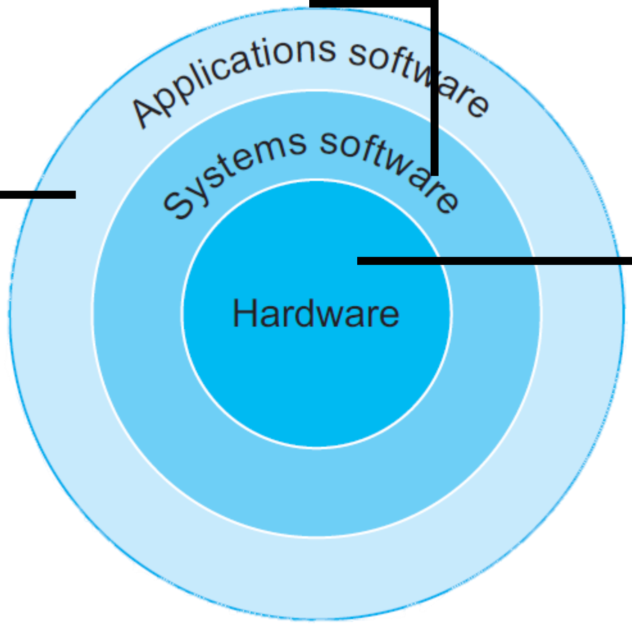
Let’s Abstract the Computer

Abstraction layer #1: Application software

Abstraction layer #2: System software

Abstraction layer #3: Hardware

Your program



Computation results

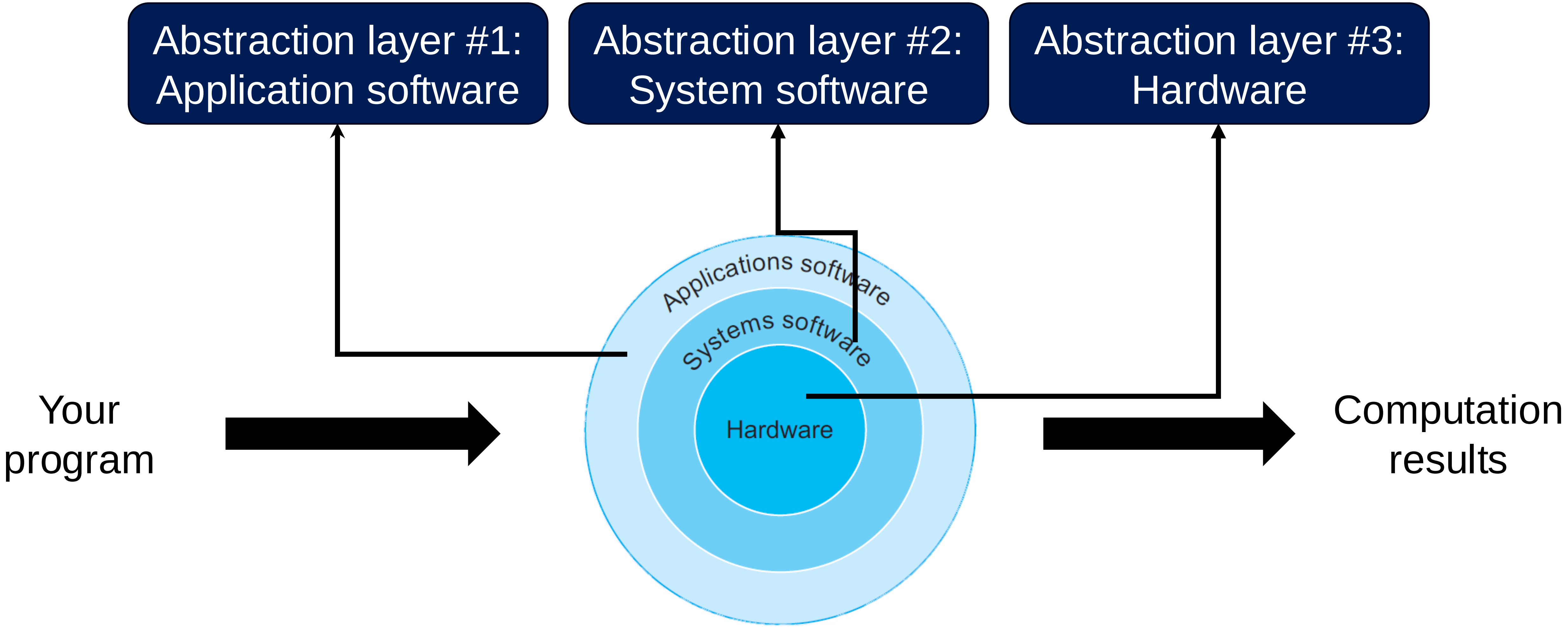
Wait, Why Abstraction?

1. Abstraction hides unnecessary details, **making things easier to understand**

2. Abstraction allows each developer to **focus on their parts** of a problem

3. Abstraction allows us to **build complex systems** by combining simple ideas (because abstracted ideas can be easily combined to form more complex ideas)

Let's Abstract the Computer



Let’s Abstract the Computer

Abstraction layer #1: Application software | **Abstraction layer #2: System software** | **Abstraction layer #3: Hardware**

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

High-level language

Compiler

```
swap:
   multi $2, $5, 4
   add $2, $4, $2
   …
```

Assembly language

Assembler

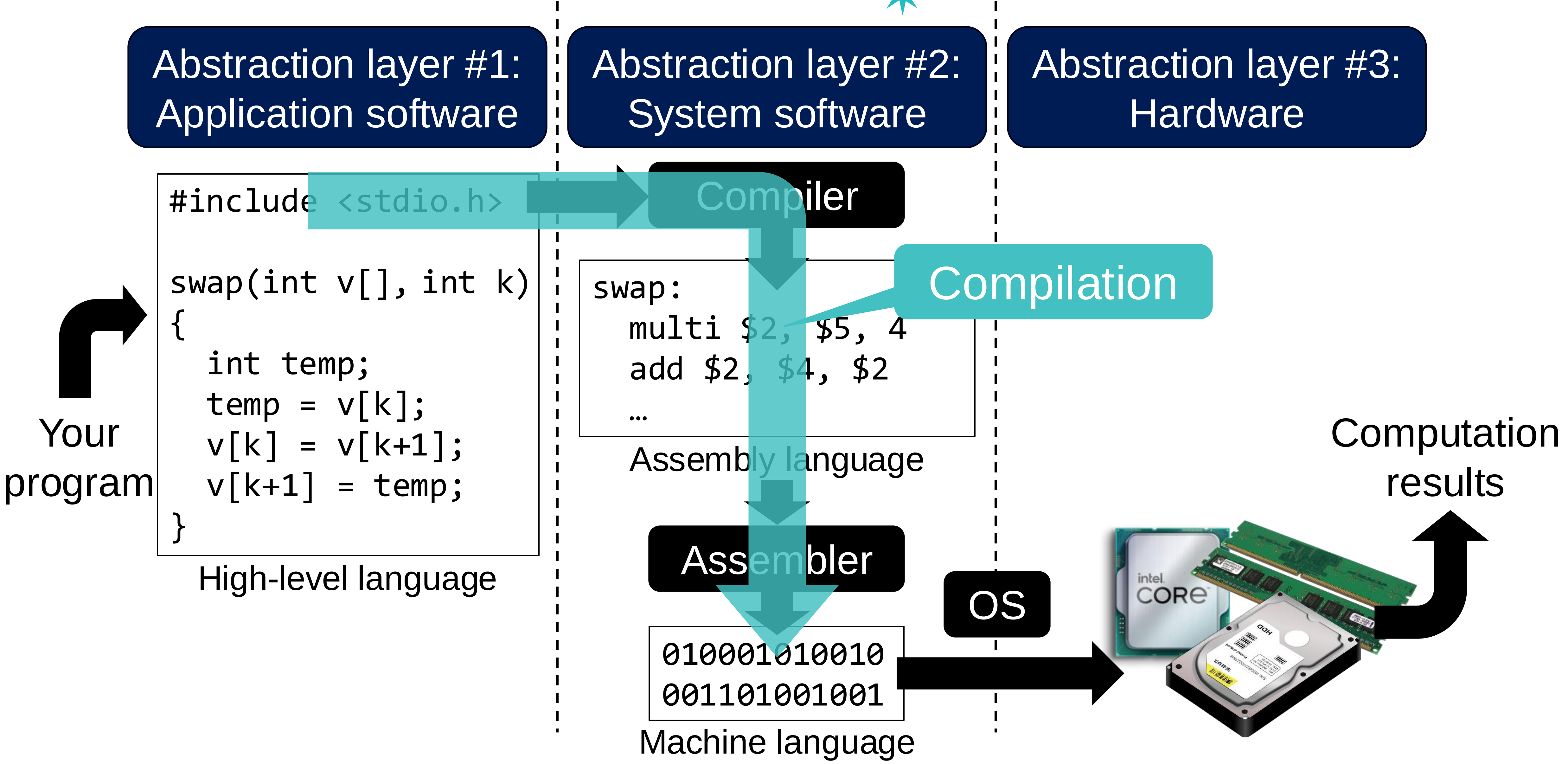
```
010001010010
001101001001
```

Machine language

OS

Computation results

Let’s Abstract the Computer



Compilation

- Converting a high-level language into a machine language that the computer can understand

```
int test (int a){
    return 32;
}
```

High-level language

Compile

```
010001010100100101
010010001000001010
111000110101010100
101010000101010010
111001010100101110
```

Machine language

Levels of Program Code

- **High-level language**
  - Closer to problem domain
  - Provides for productivity and portability

- **Assembly language**
  - The last human-readable format
  - Textual representation of instructions

```
#include <stdio.h>

swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```

Levels of Program Code

- **High-level language**
  - Closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - The last human-readable format
  - Textual representation of instructions

Instruction: a command that hardware understands

```
#include <stdio.h>

swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
```

Compiler

```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```

Levels of Program Code

- **High-level language**
  - Closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - The last human-readable format
  - Textual representation of <u>instructions</u>
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data
  - 1s and 0s (often displayed in “hex”)

Instruction: a command that hardware understands

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
```

Compiler

```
swap:
   multi $2, $5, 4
   add $2, $4, $2
   …
```

Assembler

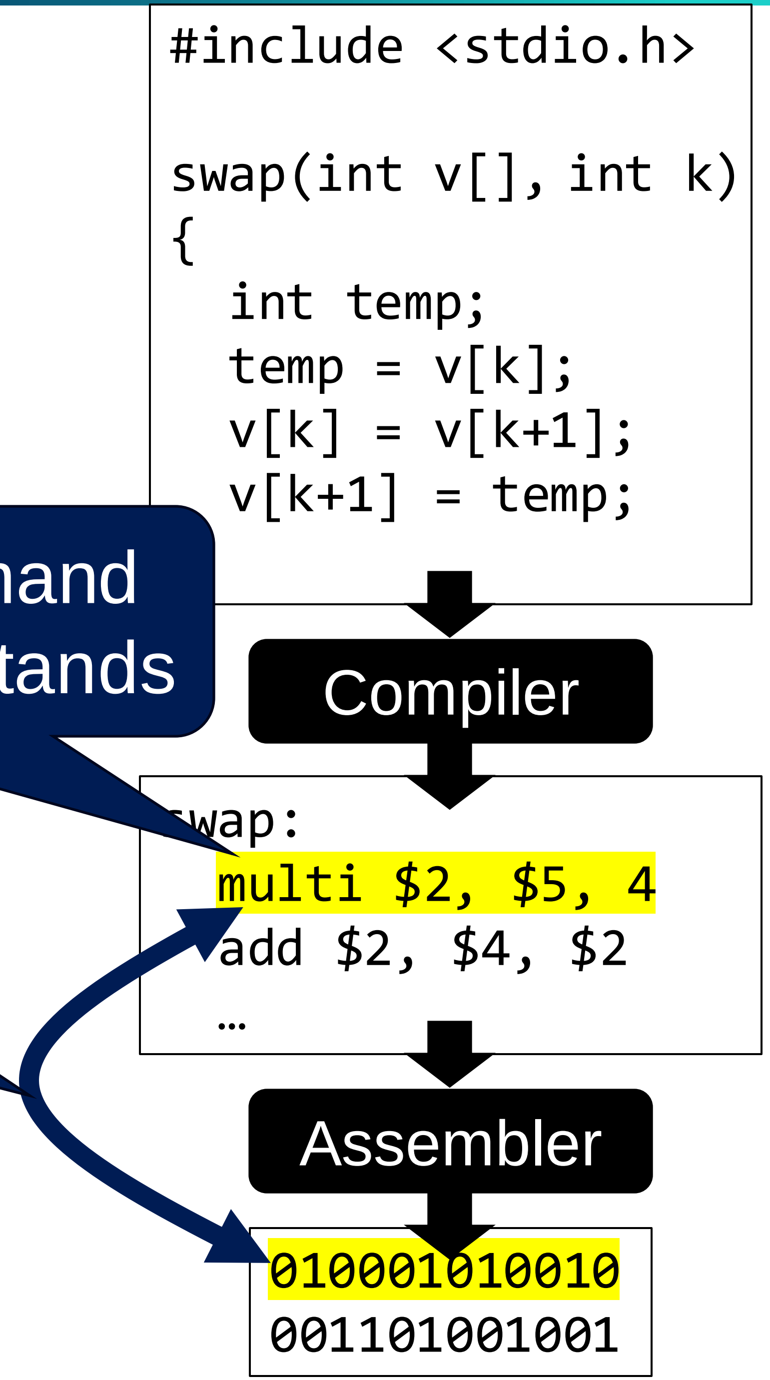
```
010001010010
001101001001
```

Levels of Program Code

- **High-level language**
  - Closer to problem domain
  - Provides for productivity and portability
- **Assembly lang**
  - The last human-readable format
  - Textual represe
- **Hardware repr**
  - Binary digits (bits)
  - Encoded instructions and data
  - 1s and 0s (often displayed in “hex”)

Instruction: a command that hardware understands

Directly mapped to each other



Let's Abstract the Computer

Abstraction layer #1: Application software	Abstraction layer #2: System software	Abstraction layer #3: Hardware

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

High-level language

Compiler

Compilation

```
swap:
   multi $2, $5, 4
   add $2, $4, $2
   …
```

Assembly language

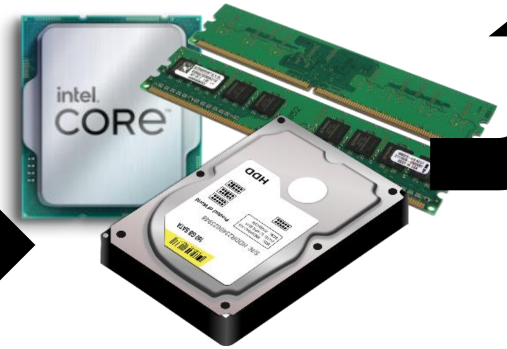
Assembler

```
010001010010
001101001001
```

Machine language

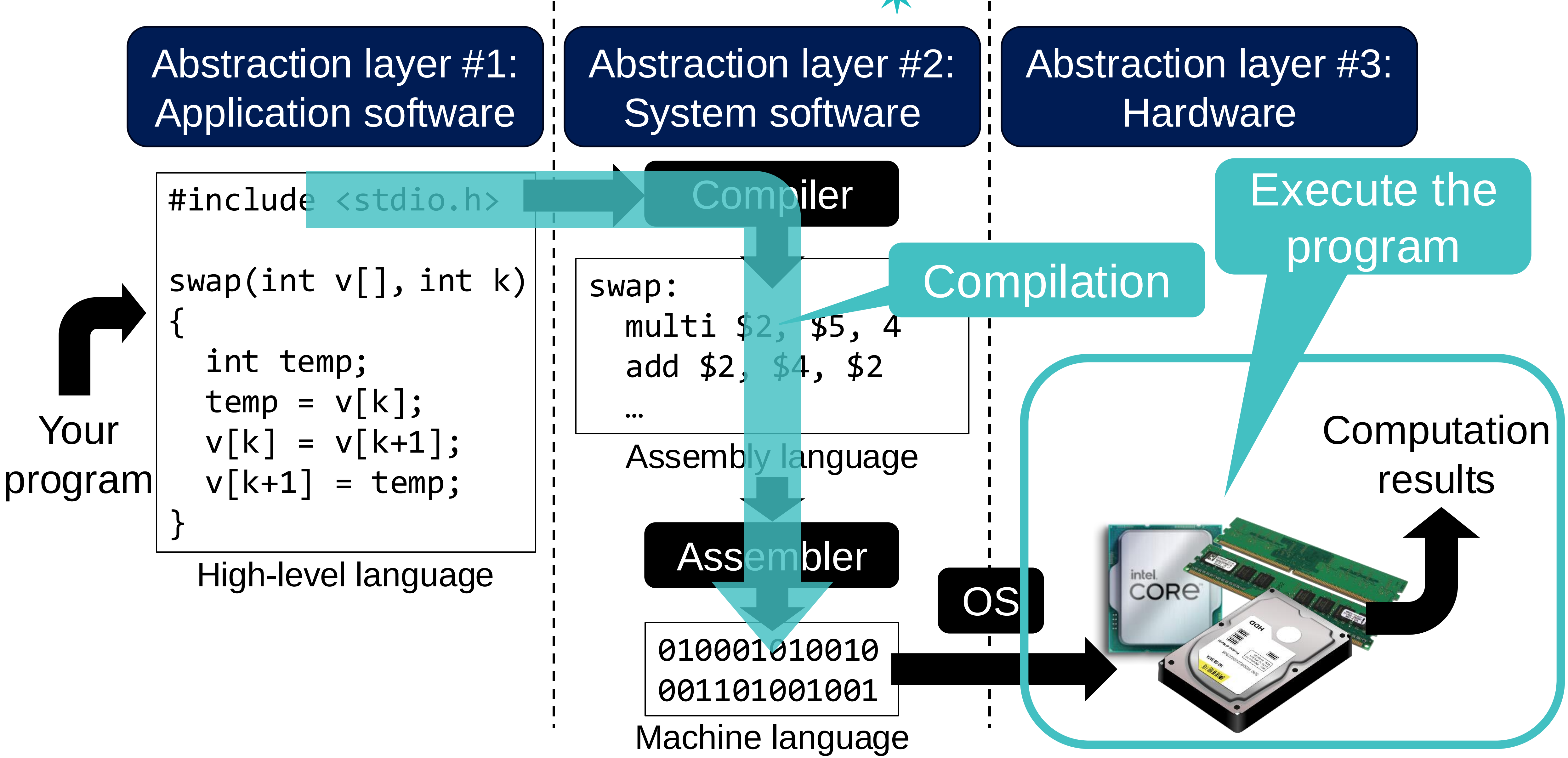
OS

Execute the program

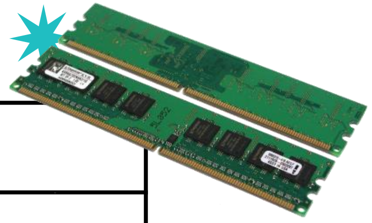


Computation results

Hardware Components of a Computer



Hardware Components of a Computer



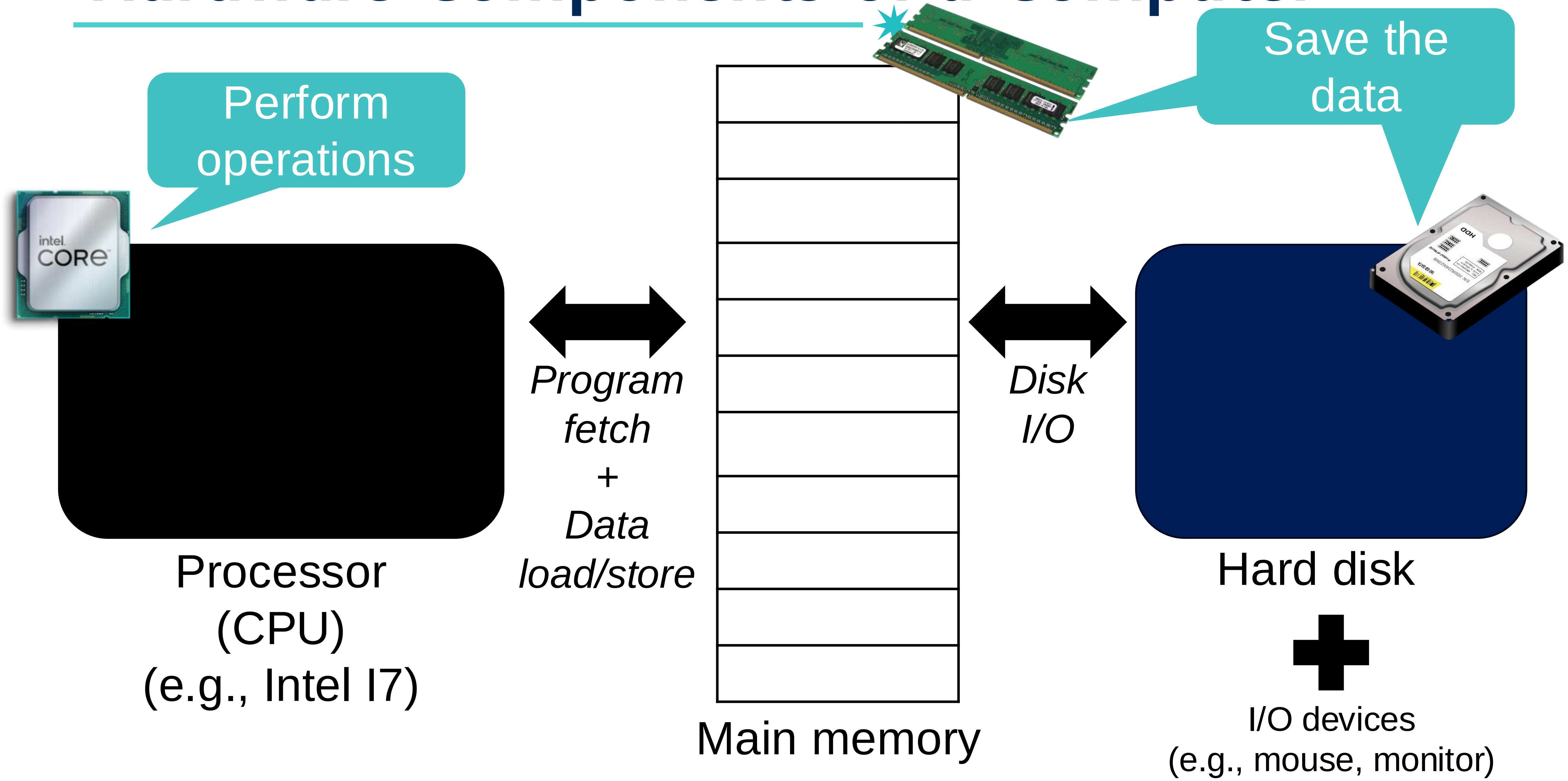
Processor
(CPU)
(e.g., Intel I7)

Main memory

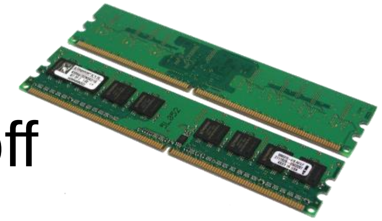
Hard disk

I/O devices
(e.g., mouse, monitor)

Hardware Components of a Computer

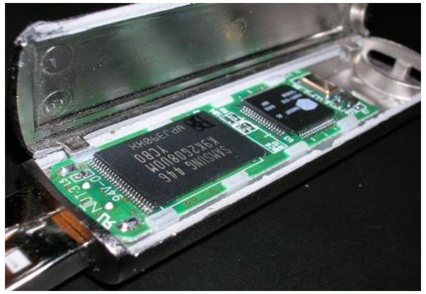


A Safe Place for Data

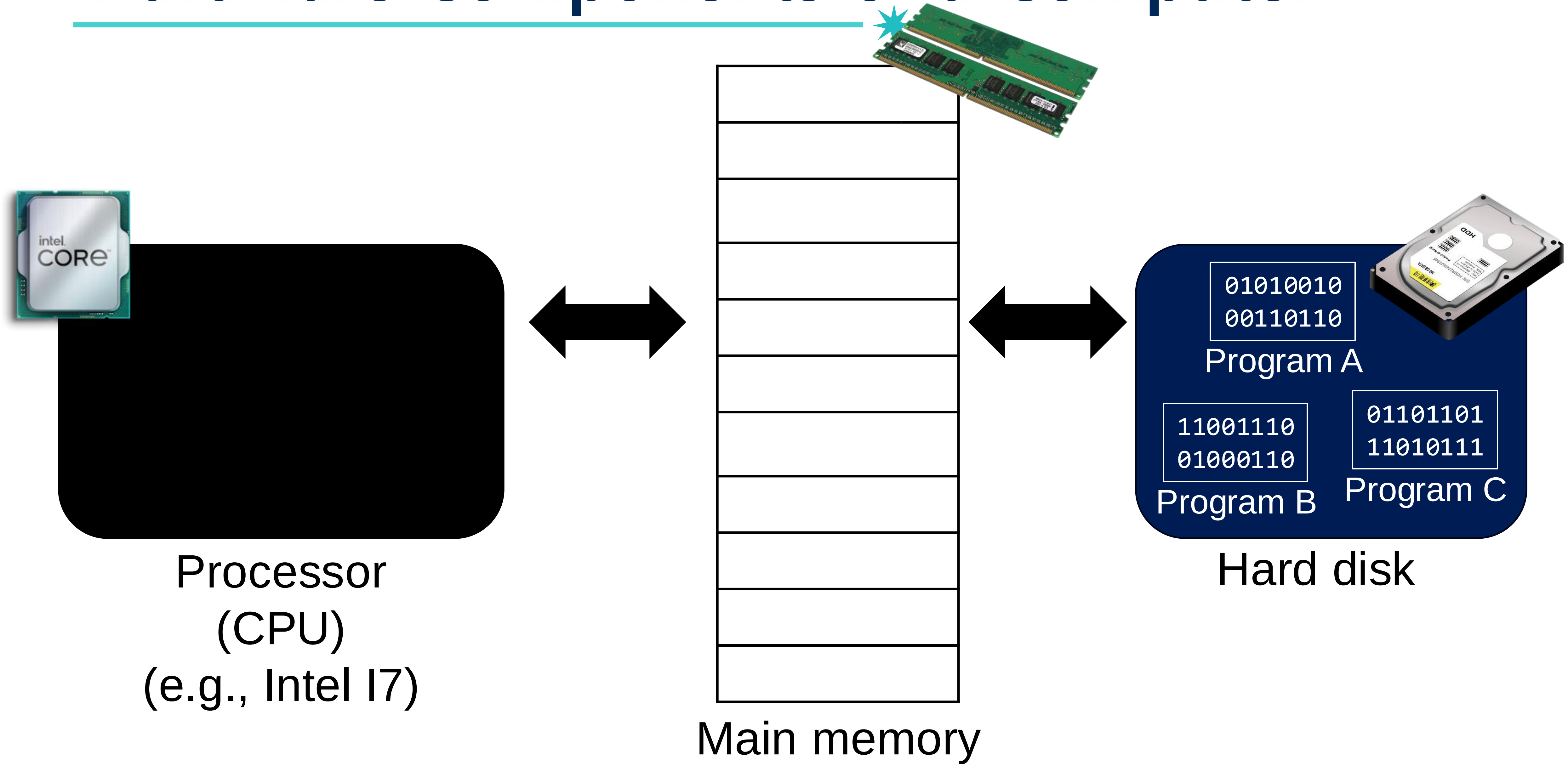


- **Volatile main memory**
  - Loses instructions and data when power off
  - E.g., DRAM

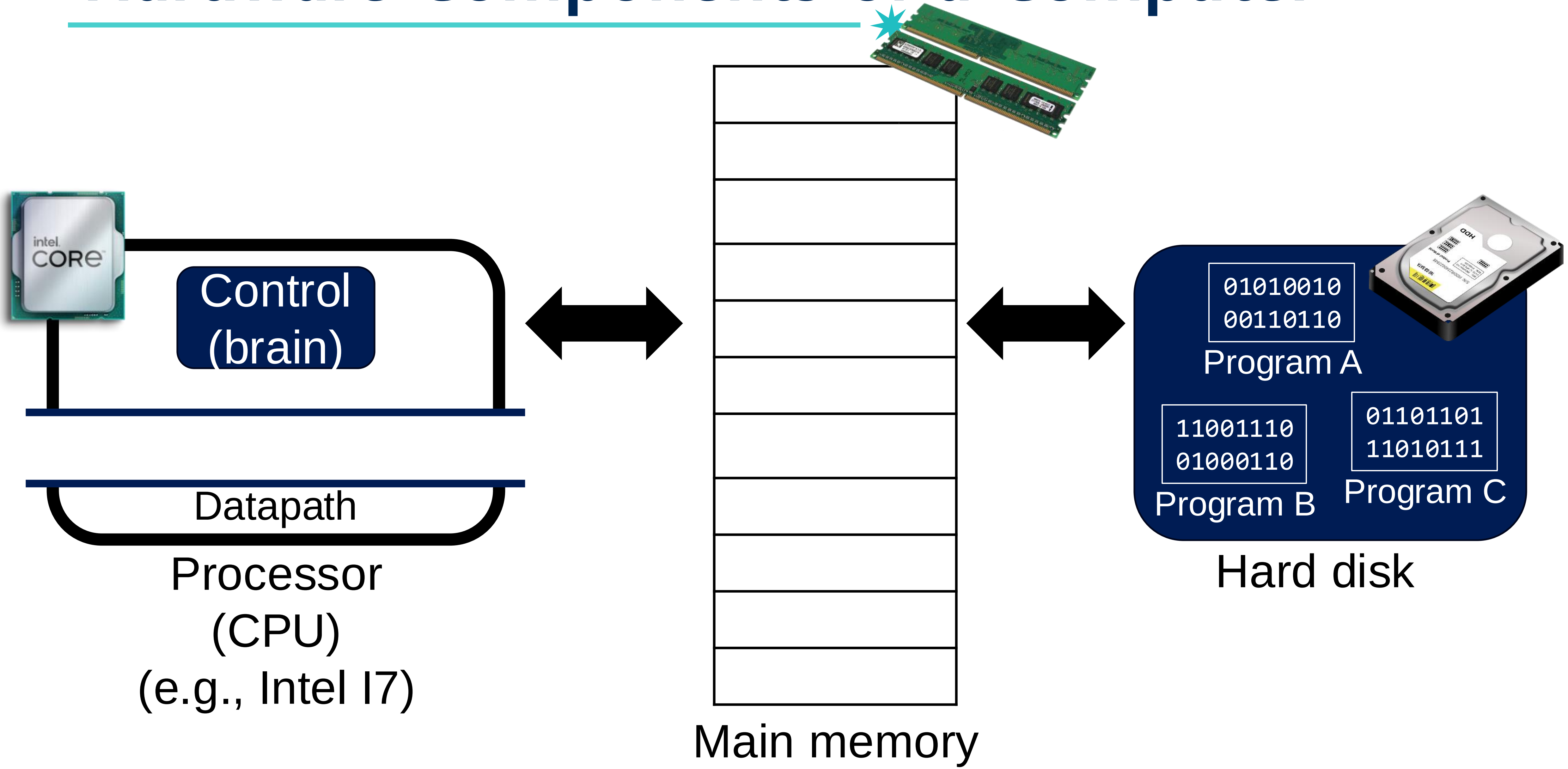
- **Non-volatile secondary memory**
  - HDD, SSD
  - Magnetic disk
  - Flash memory
  - Optical disk (CDROM, DVD)



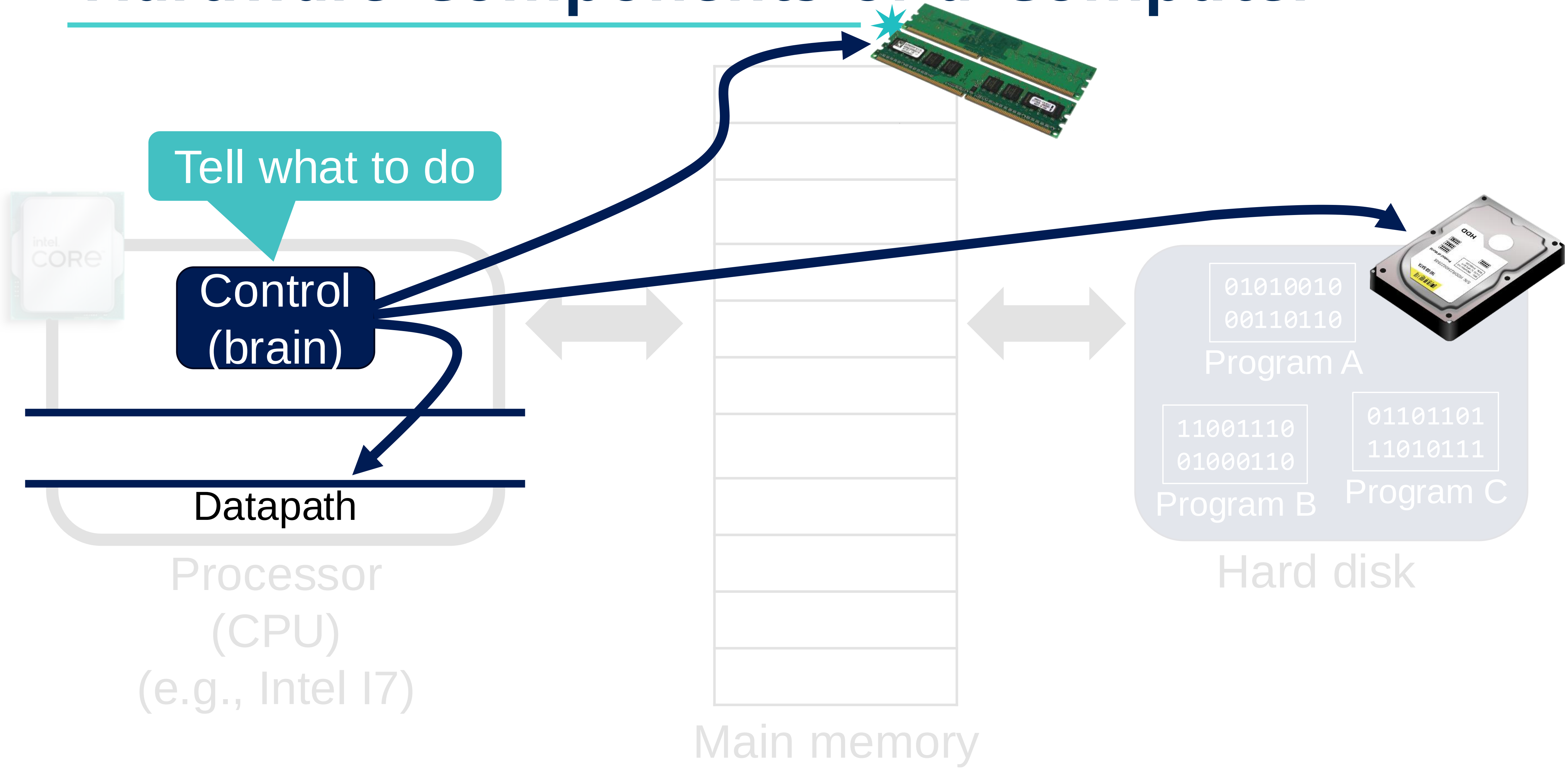
Hardware Components of a Computer



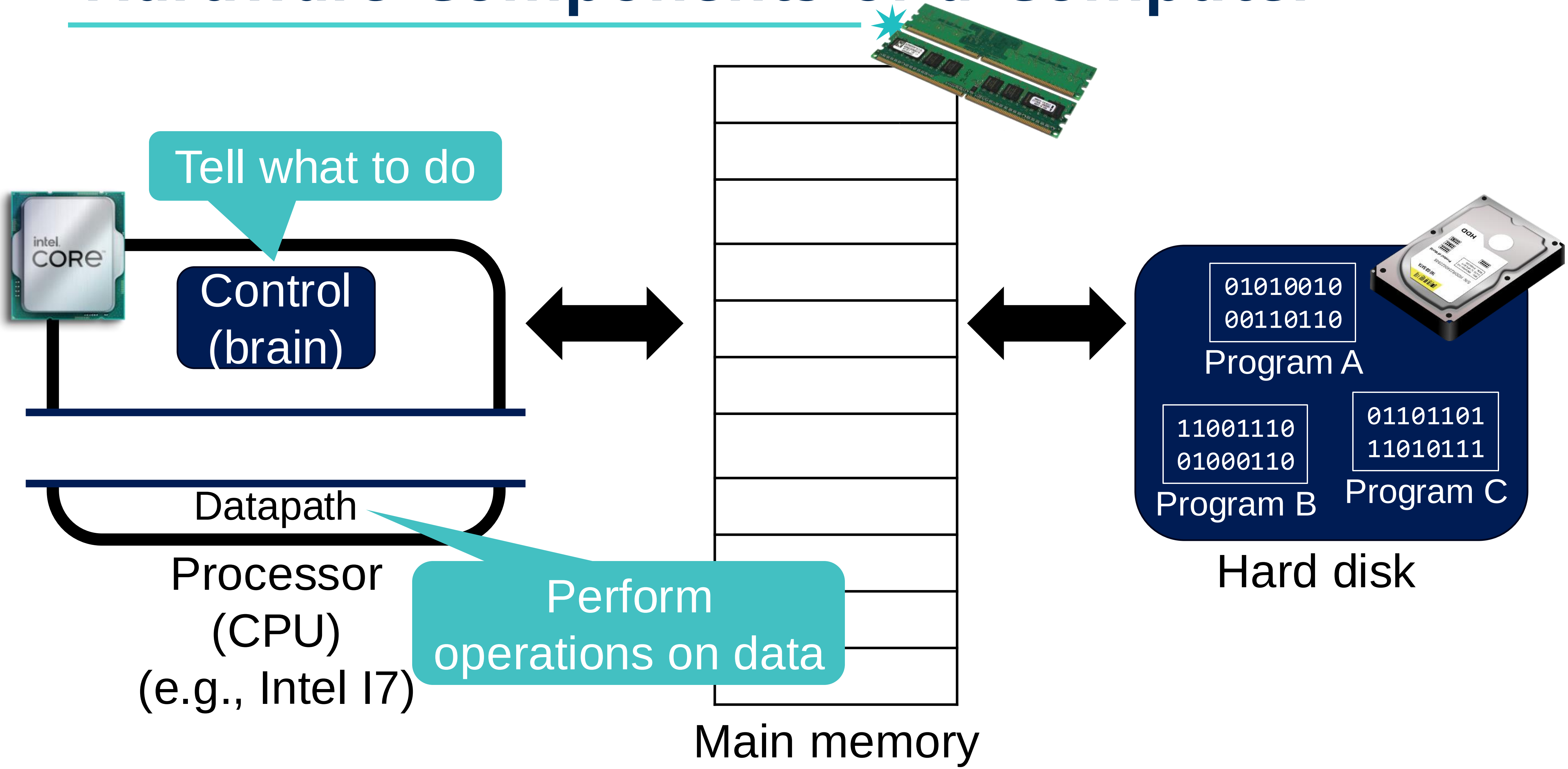
Hardware Components of a Computer



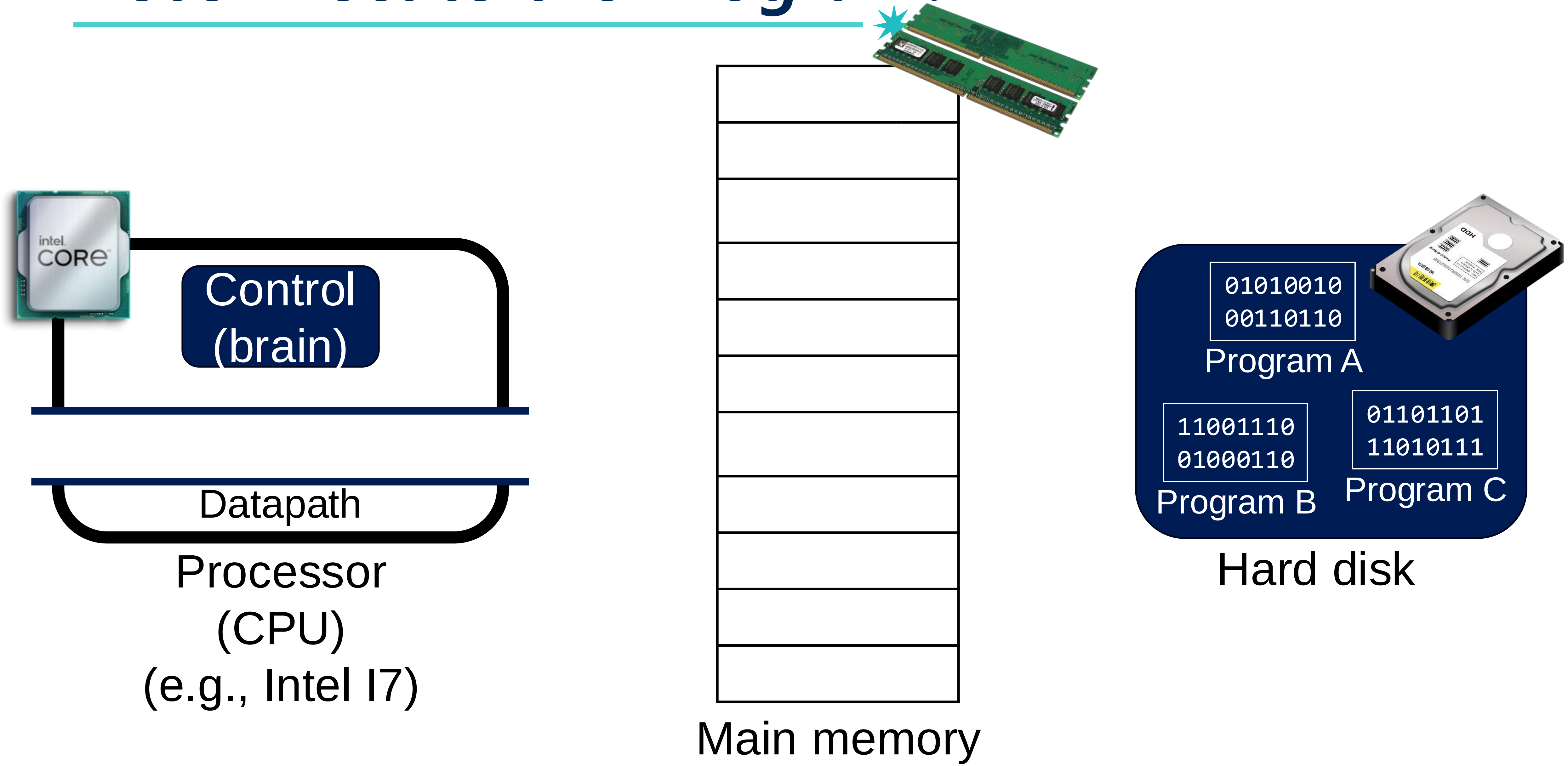
Hardware Components of a Computer



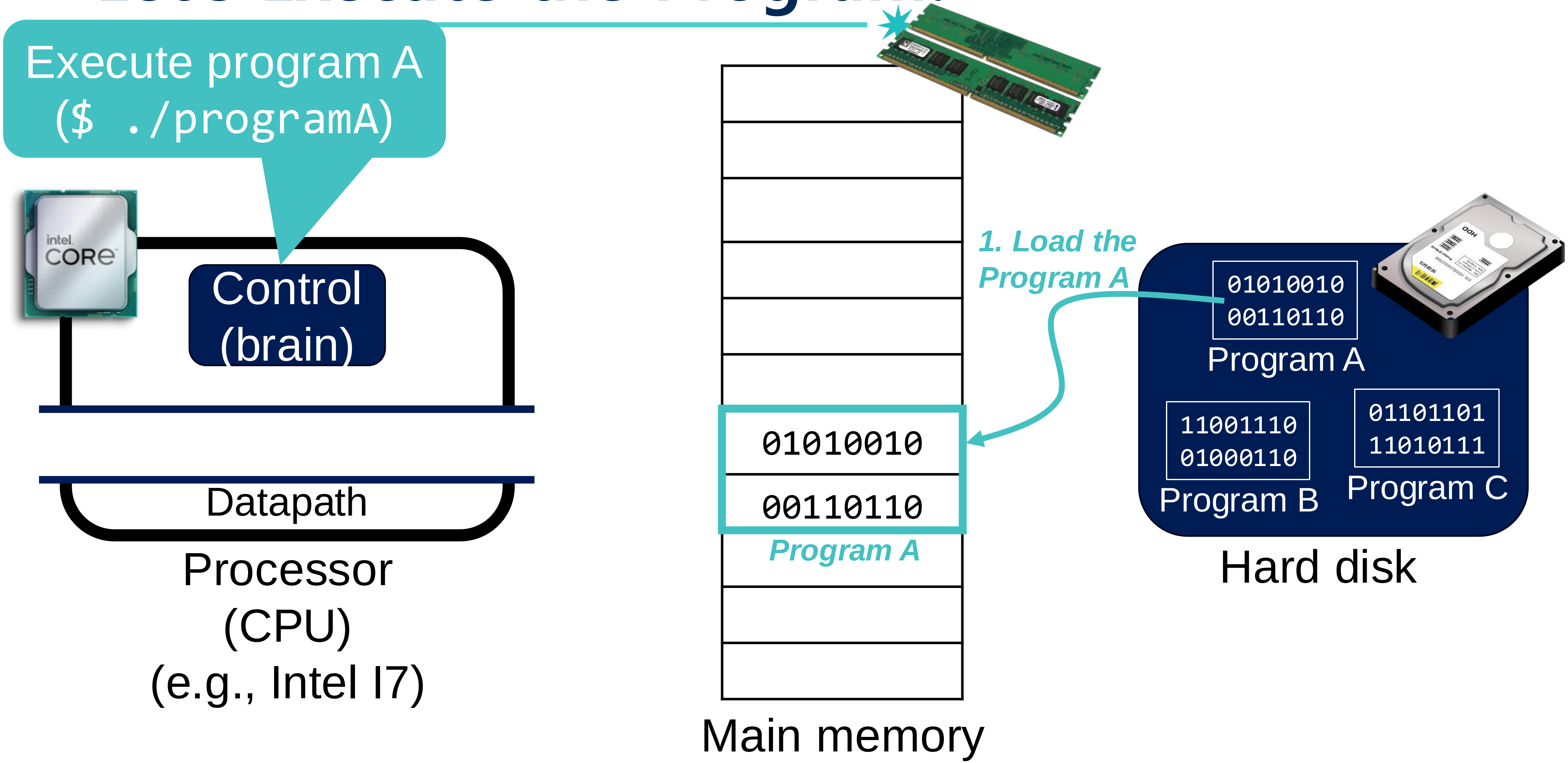
Hardware Components of a Computer



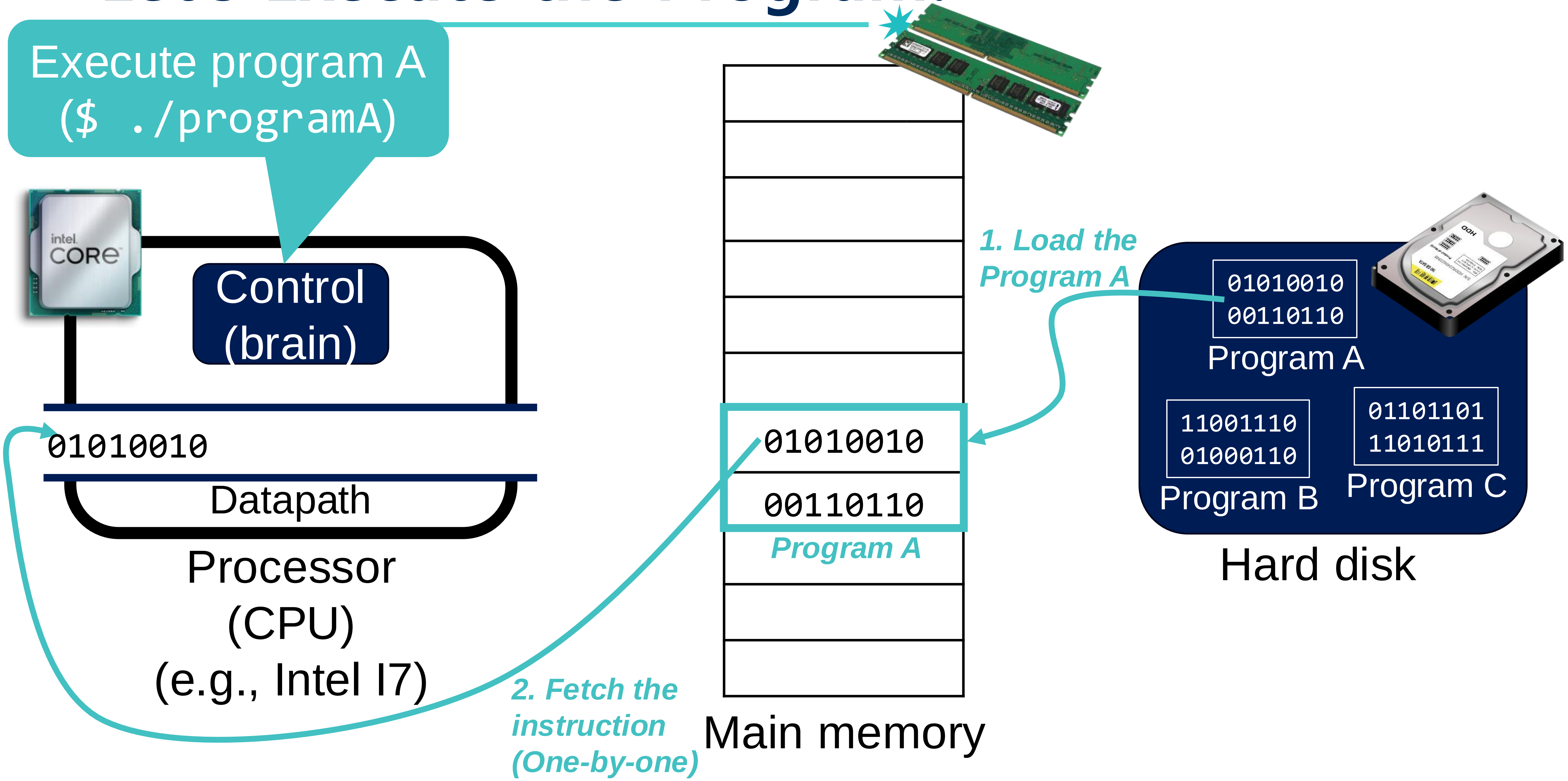
Let’s Execute the Program!



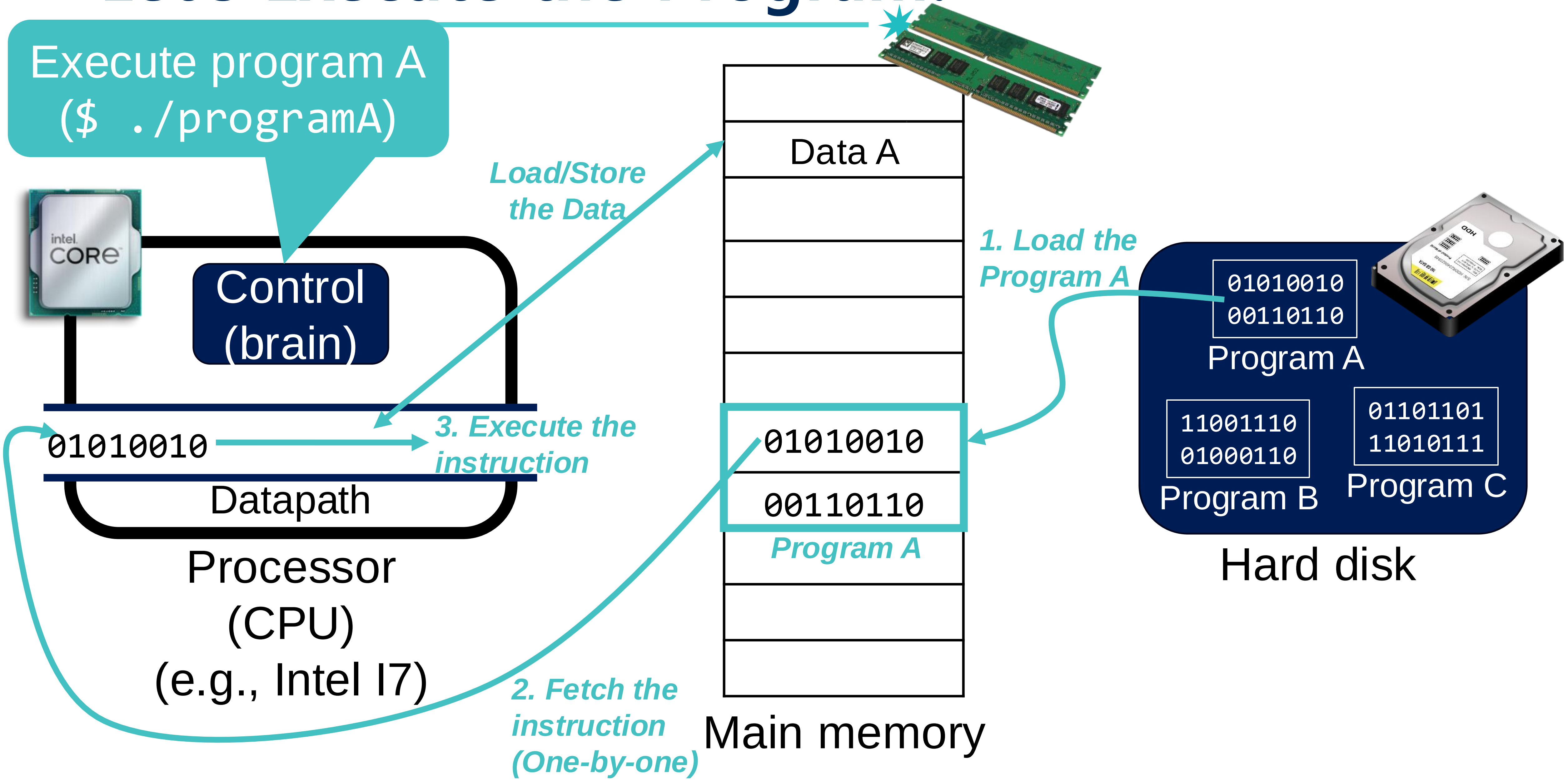
Let's Execute the Program!



Let's Execute the Program!



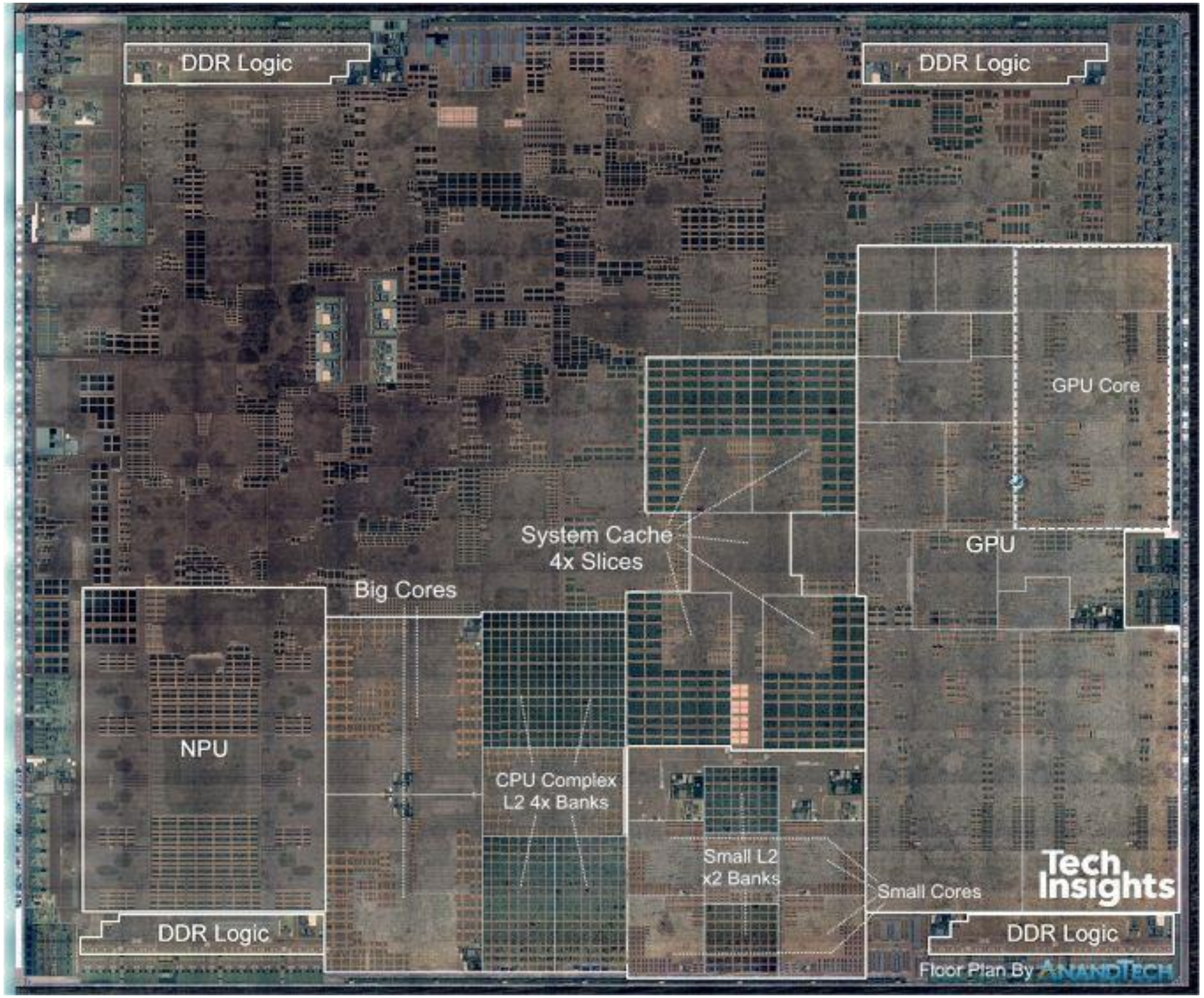
Let's Execute the Program!



Inside the Processor (CPU)

- **Datapath**
  - Perform operations on data
- **Control**
  - Tell the datapath, memory, and I/O devices what to do according to program
- **Registers/cache memory (=> next lecture)**
  - Small fast memory for immediate access to data

Apple A12 Processor



Abstraction Helps us Deal with Complexity

Abstraction layer #1: Application software

Abstraction layer #2: System software

Abstraction layer #3: Hardware

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

High-level language

Compiler

```
swap:
   multi $2, $5, 4
   add $2, $4, $2
   …
```

Assembly language

Assembler

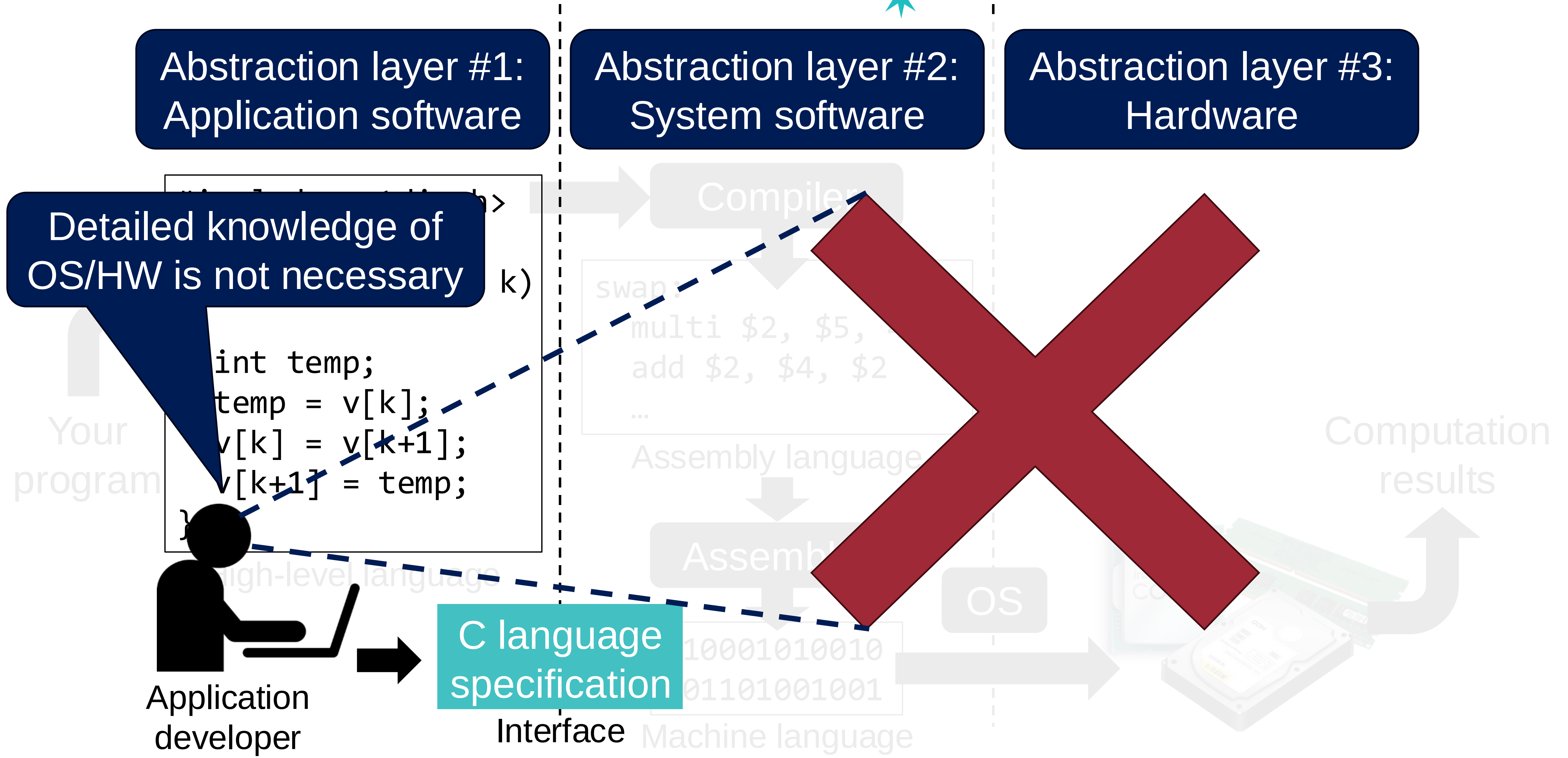
```
010001010010
001101001001
```

Machine language

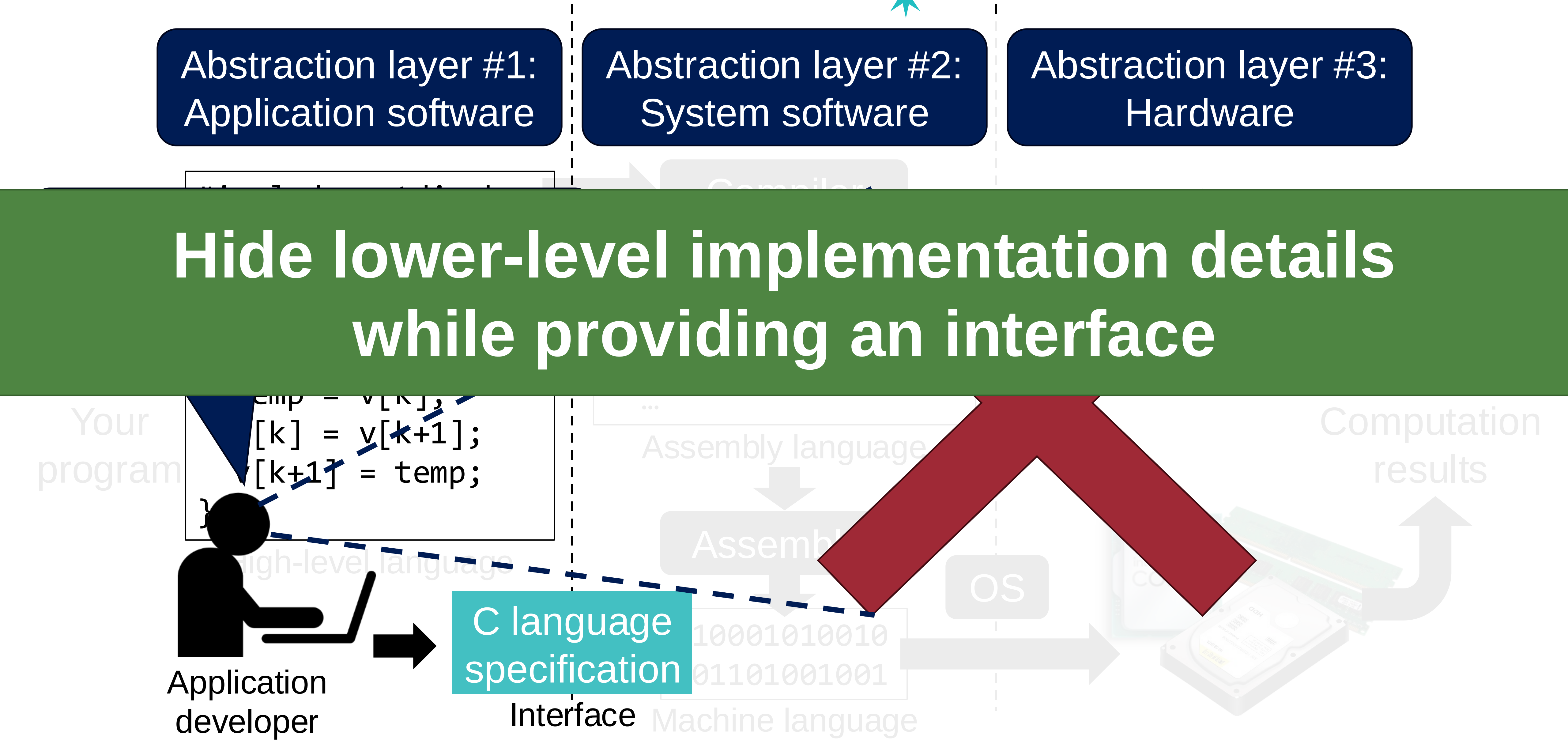
OS

Computation results

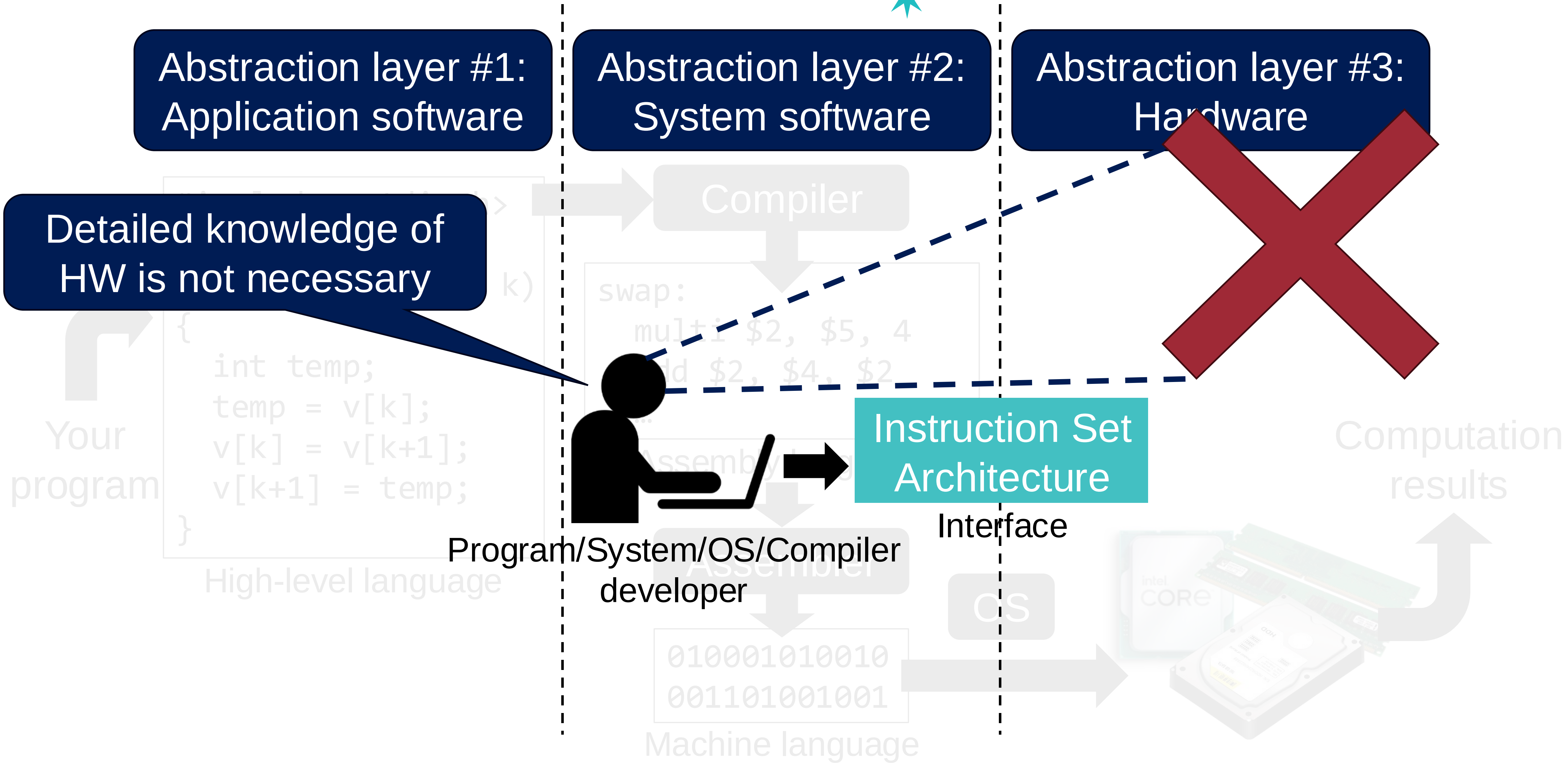
Abstraction Helps us Deal with Complexity



Abstraction Helps us Deal with Complexity



Similarly, We Can Consider the Hardware/Software Interface



Instruction Set Architecture (ISA)

- The interface between hardware and low-level software

- **Hardware abstraction** visible to software (compiler or programmer)
  - Instruction set
  - Operand types
  - Data types (integers, FPs, ...)
  - Memory addressing modes
  - ...

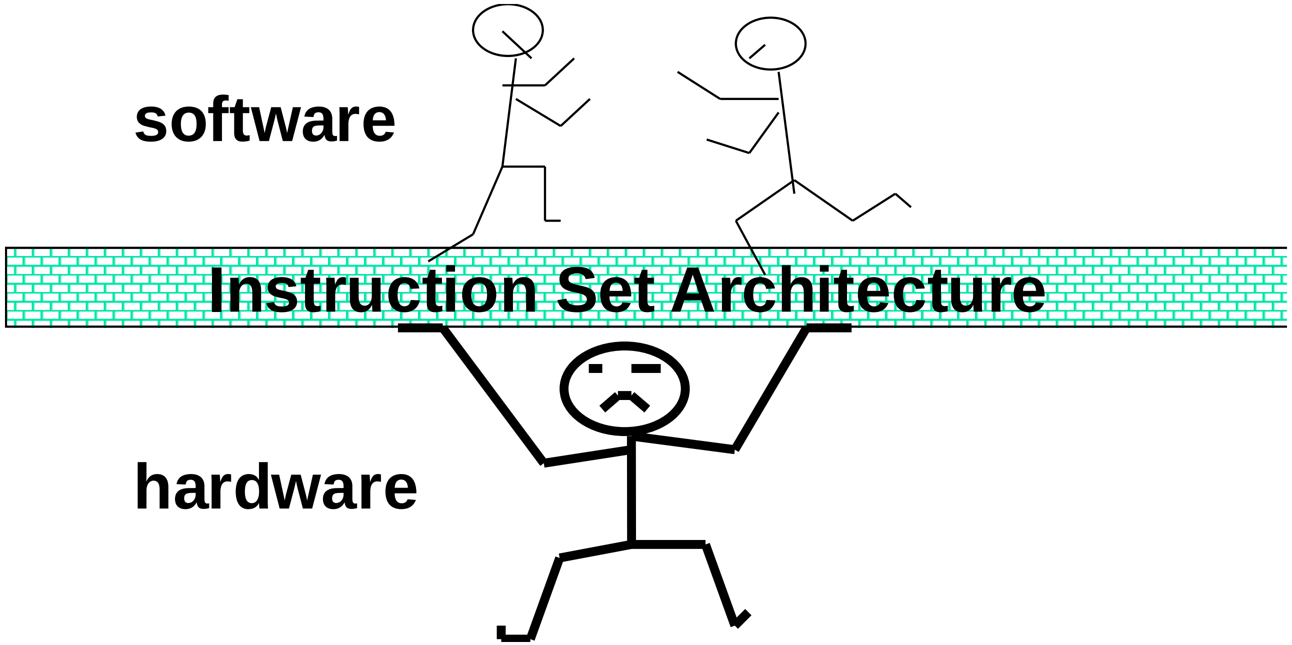
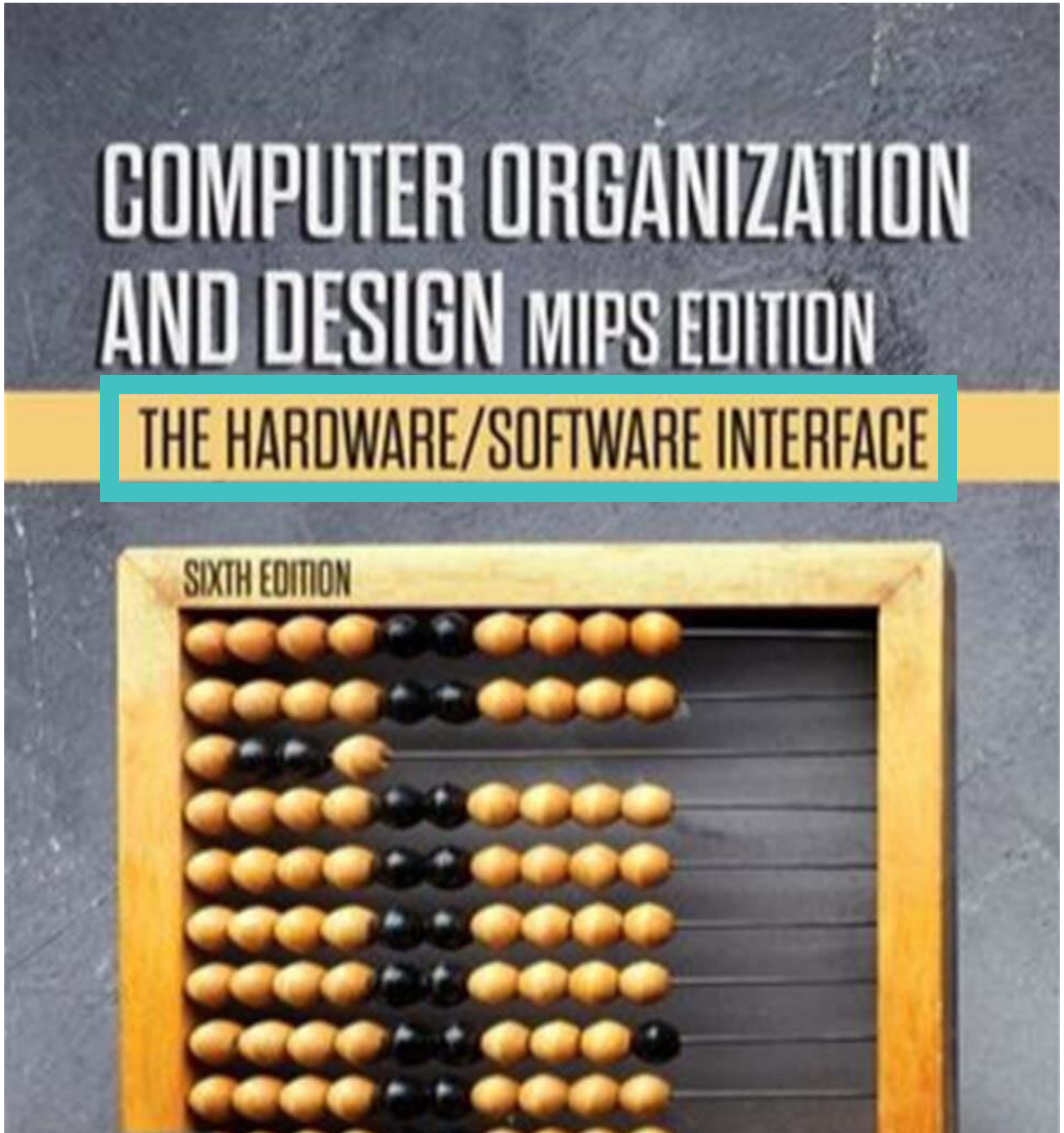


Image from: D. Culler @ UC Berkeley



Now, you can understand the meaning of the subtitle in the textbook ☺

Instruction Set Architecture (ISA)

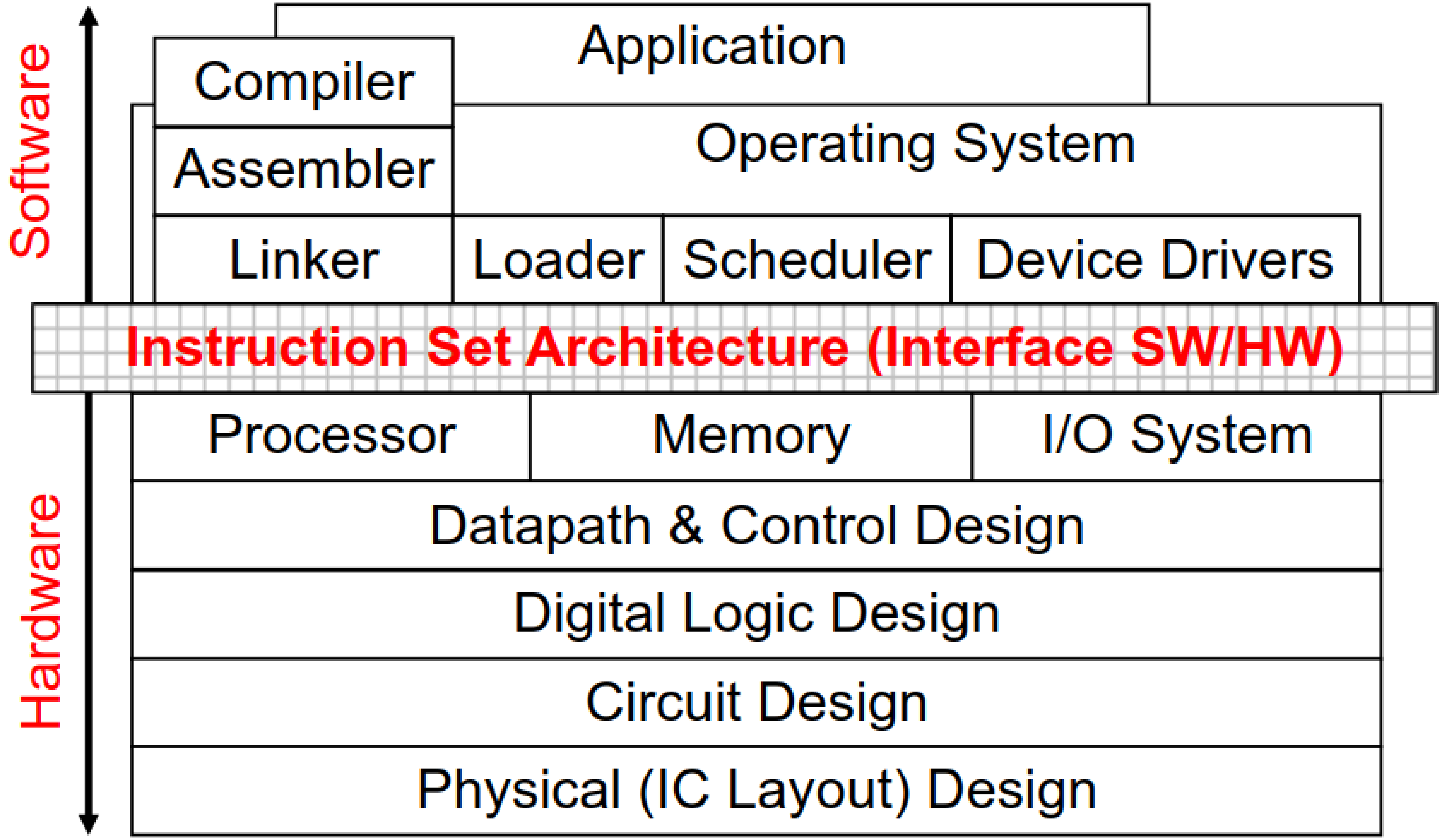


Image from: https://medium.com/@shb8086/tutorial-series-isa-fa9648c5e9e5

Seven Great Ideas in Computer Architecture

- Use **abstraction** to simplify design
- Make the **common case fast**
- Performance via **parallelism**
- Performance via **pipelining**
- Performance via **prediction**
- **Hierarchy** of memories
- **Dependability** via redundancy

You do not need to memorize.
We will cover each topic!

You will learn fundamental ***principles*** used in modern computer architectures to <u>improve the performance of computations</u>

break

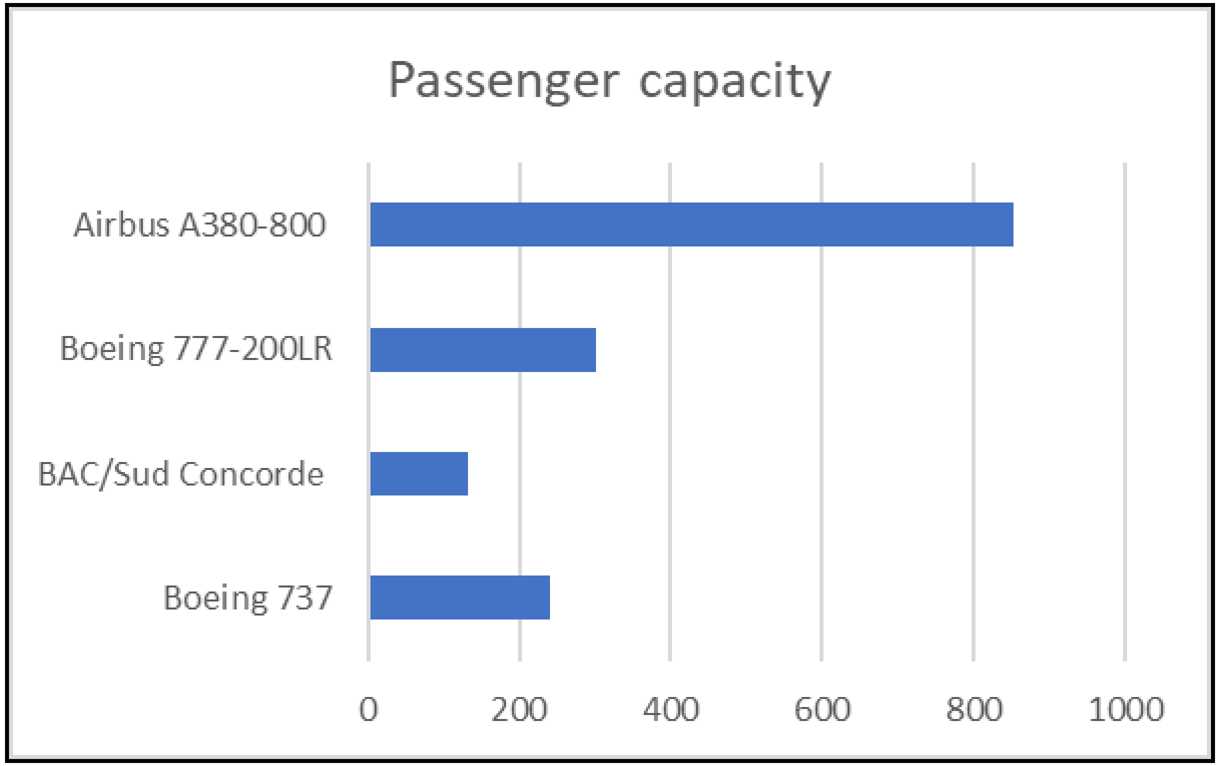
How we calculate the performance?

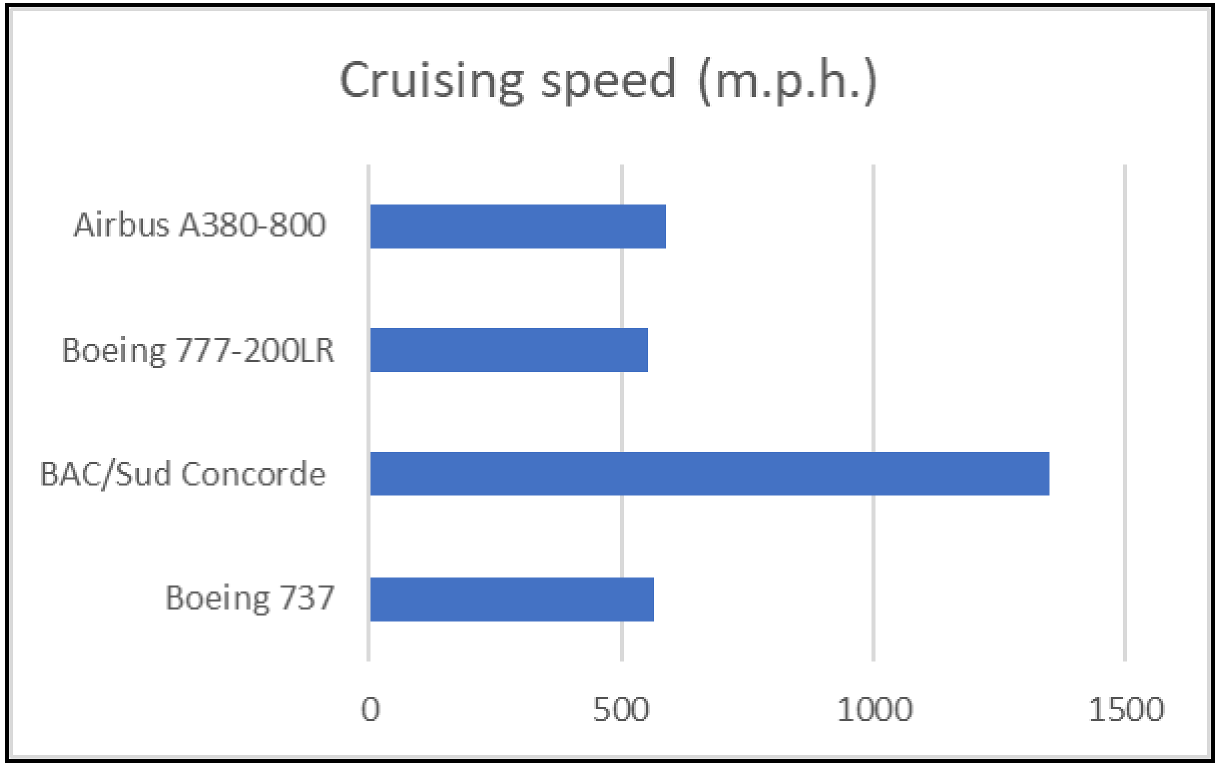
What is the structure of a computer?

Performance

Defining Performance

- Which airplane has the best performance?





It depends on the metrics!

We Focus on the Time

- Most important thing: time, time, and time

Metrics: CPU Time

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Metrics: CPU Time

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Our goal!

of Instructions per Program (Instruction Count)

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```

# of instructions per program

Affected by:

- *Compiler*
- *Algorithm*
- *Programming language*
- *ISA*

Clock Cycles per Instruction (CPI)

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```

Clock Cycles per Instruction (CPI)

- Most important thing: time, time, and time

CPU clocking:
Operation of digital hardware governed by a constant-rate clock

$$\frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

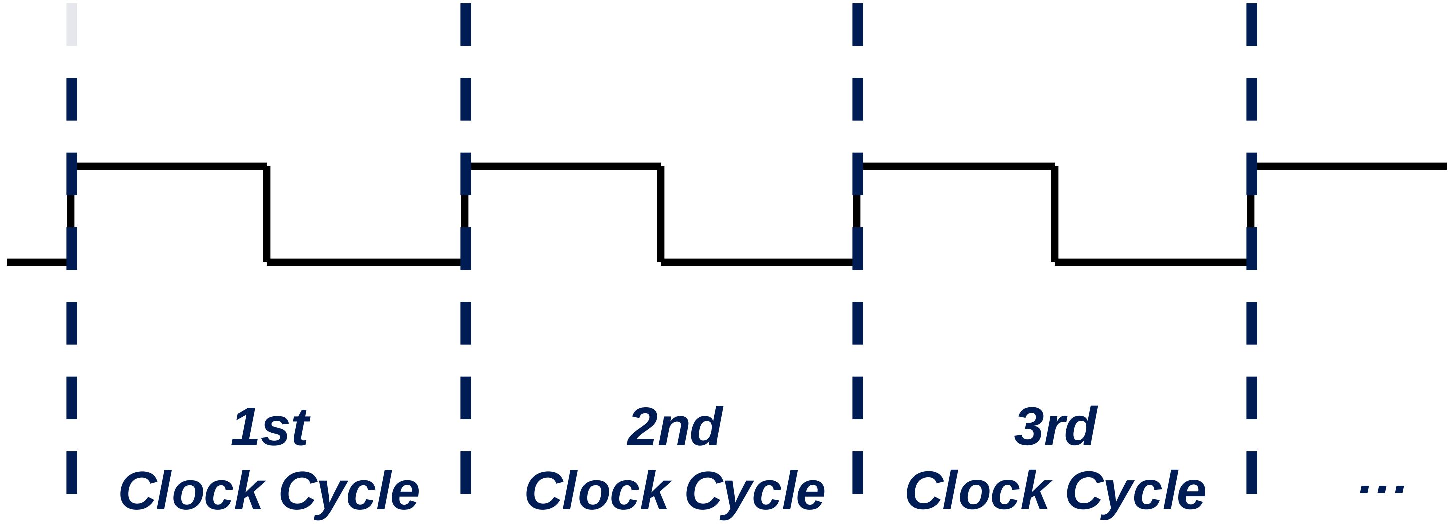
```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```

Clock Cycles per Instruction (CPI)

- Most important thing: time, time, and time

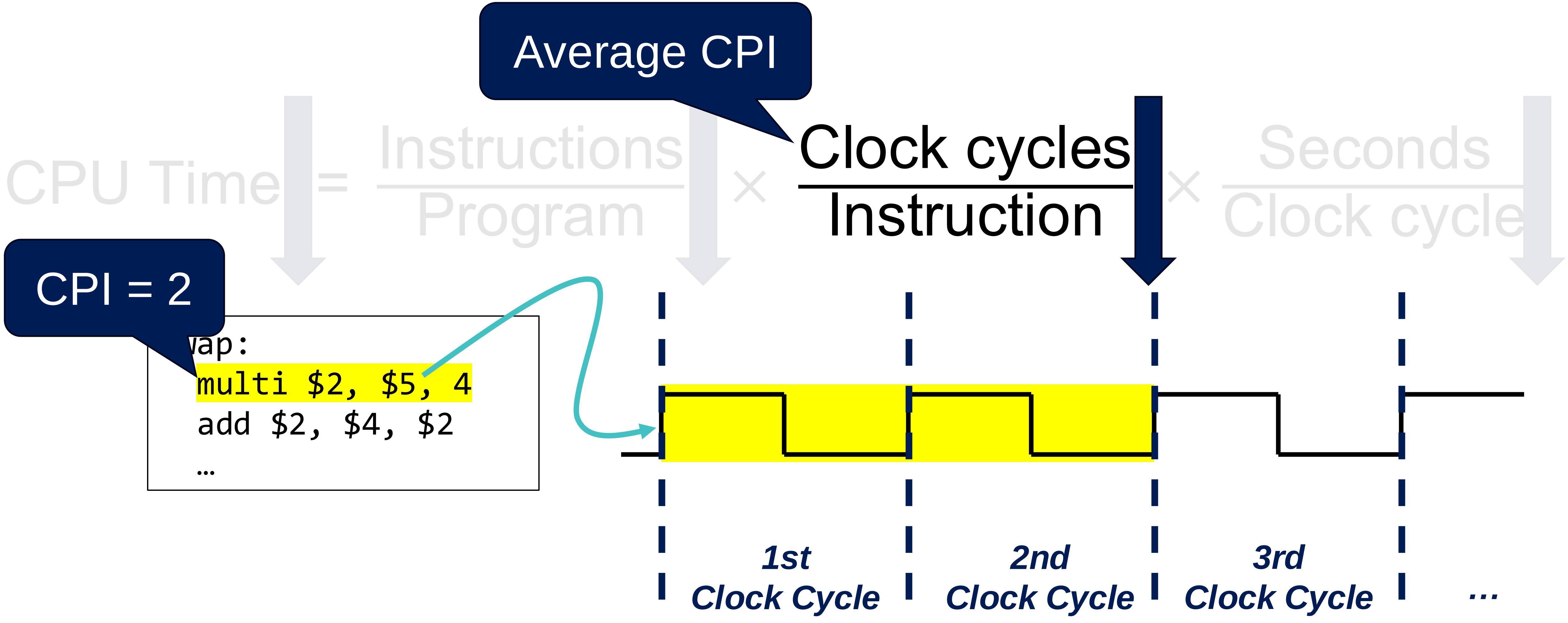
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

```
swap:
  multi $2, $5, 4
  add $2, $4, $2
  …
```



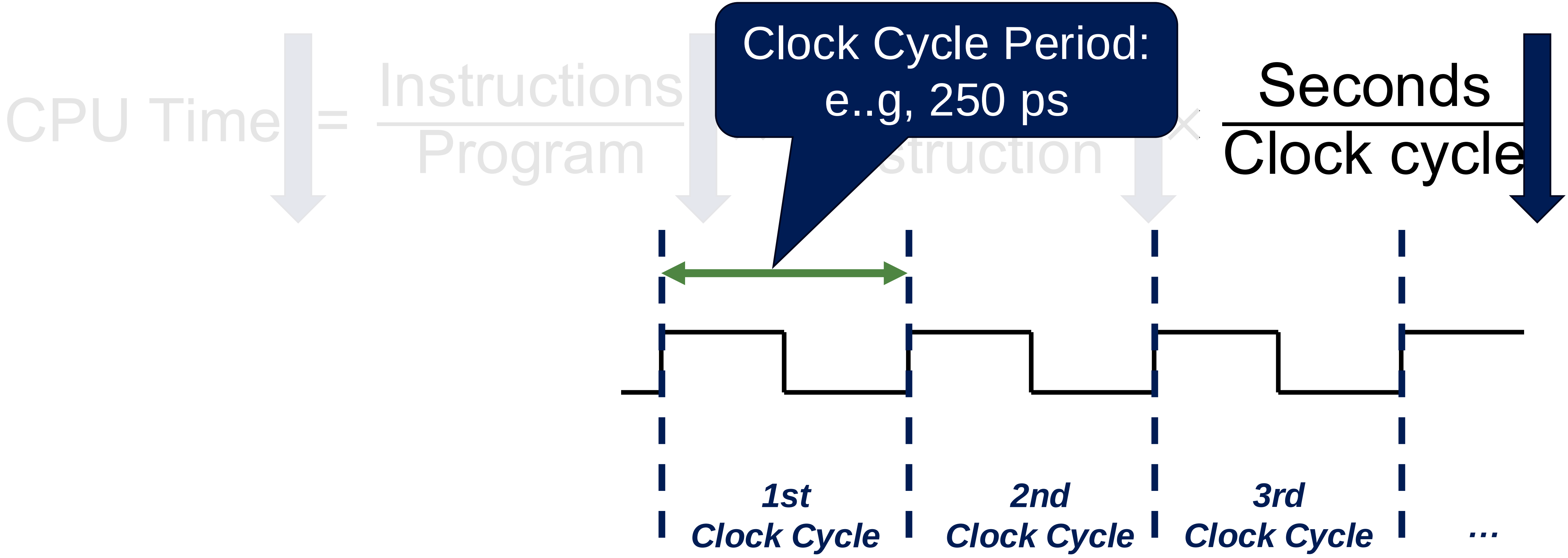
Clock Cycles per Instruction (CPI)

- Most important thing: time, time, and time



Clock Cycle Period (Clock Cycle Time))

- Most important thing: time, time, and time



FYI: CPU Frequency (Clock Rate)

$$\text{CPU Frequency (Hz)} = \frac{1}{\text{Clock Cycle Period}}$$

Number of cycles per one second

[INTEL] 코어i7-14세대 14700K 랩터레이크 리프레시 (3.40GHz/33MB) 정품박스

인텔(소켓1700) / 8+12코어 / 16+12쓰레드 / 기본 클럭: 3.4GHz / 최대 클럭: 5.6GHz / L3 캐시: 33MB / PBP : 125W / PCIe5.0 , 4.0 / 메모리 규격: DD
픽: 탑재 / 인텔 UHD 770 / 기술 지원: 하이퍼스레딩 / 쿨러: 미포함

Metrics: CPU Time

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Affected by:

- *ISA*
- *Hardware implementation*

Metrics: CPU Time

- Most important thing: time, time, and time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Let's skip the details for now.
We'll cover them properly later.

Power Trends

Power Trends

$$\textbf{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{CPU Frequency}$$

Power Trends

Frequency (Hz)↑
⇒ CPU Time↓

Power = Capacitive load × Voltage^2 × CPU Frequency

With the advancement of technology, the frequency can continue to increase

1GHz

1000GHz

However, ...

Frequency (Hz)↑
⇒ CPU Time↓

Power = Capacitive load × Voltage^2 × CPU Frequency

Power↑ ⇒
Heat↑

1GHz

1000GHz

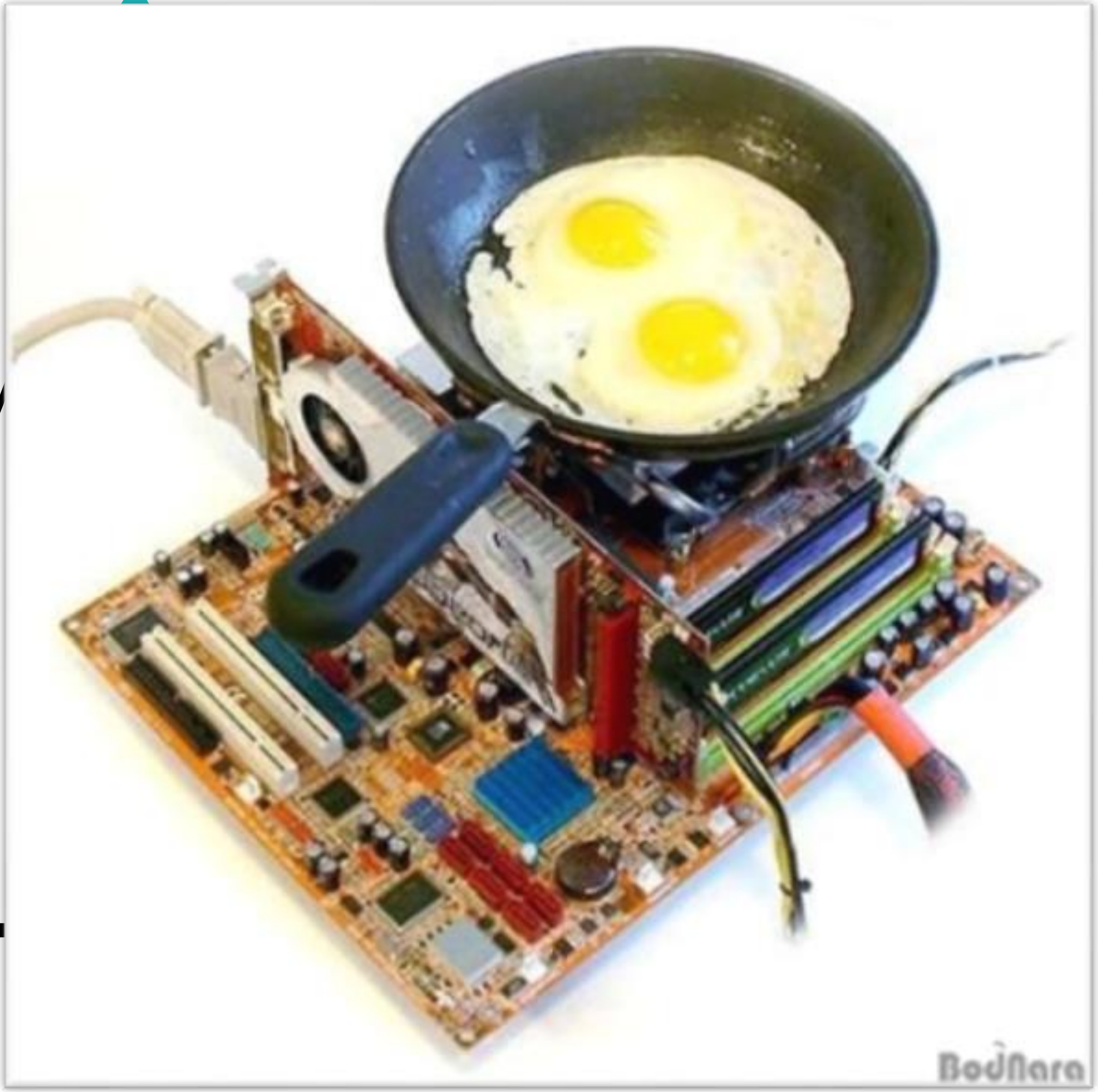
X1000 Power

However, There is the Power Wall

Power = Capacitive load × V

Power↑ ⇒ Heat↑

X1000 Power



How about Reducing Voltage?

Frequency (Hz)↑
⇒ CPU Time↓

Power = Capacitive load × Voltage^2 × CPU Frequency

5V ↓ 1V

1GHz ↓ 1000GHz

X40 Power ← 1V

How about Reducing Voltage?

Frequency (Hz)↑
⇒ CPU Time↓

However, we can't reduce voltage further due to leakage power

X40 Power ← 1V 1000GHz

Example

- Suppose a new simpler CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

- Q. What is the impact on power?

$$\frac{P_{new}}{P_{old}} =$$

Example

- Suppose a new simpler CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

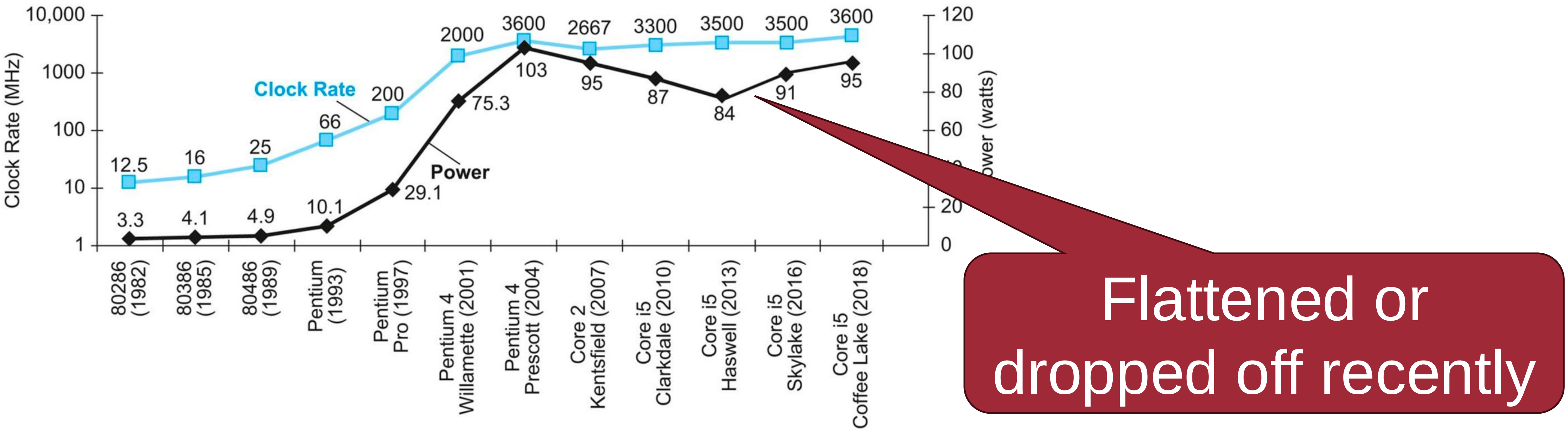
- Q. What is the impact on power?

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

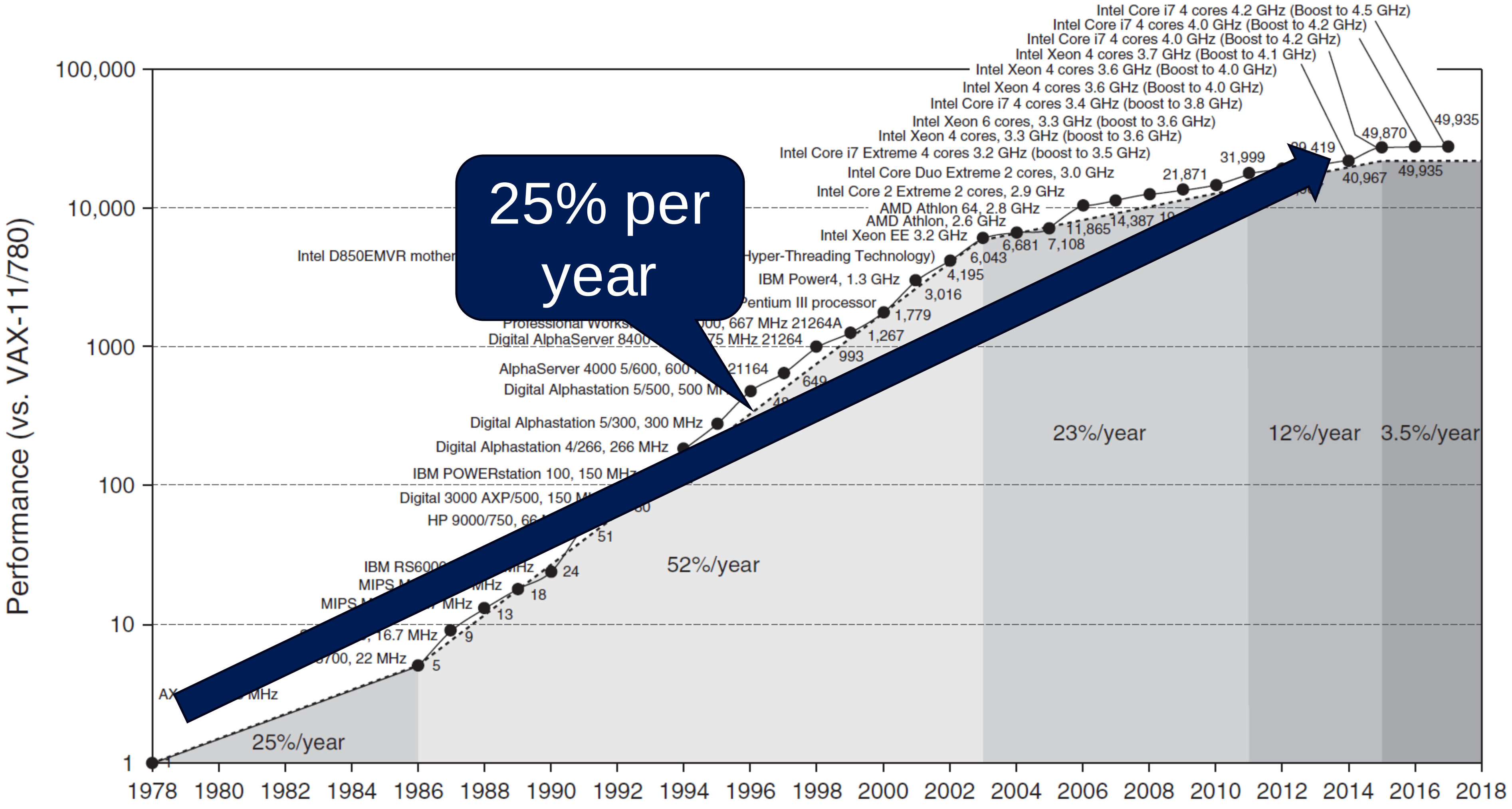
The new processor uses about
half the power of the old processor

The Power Wall

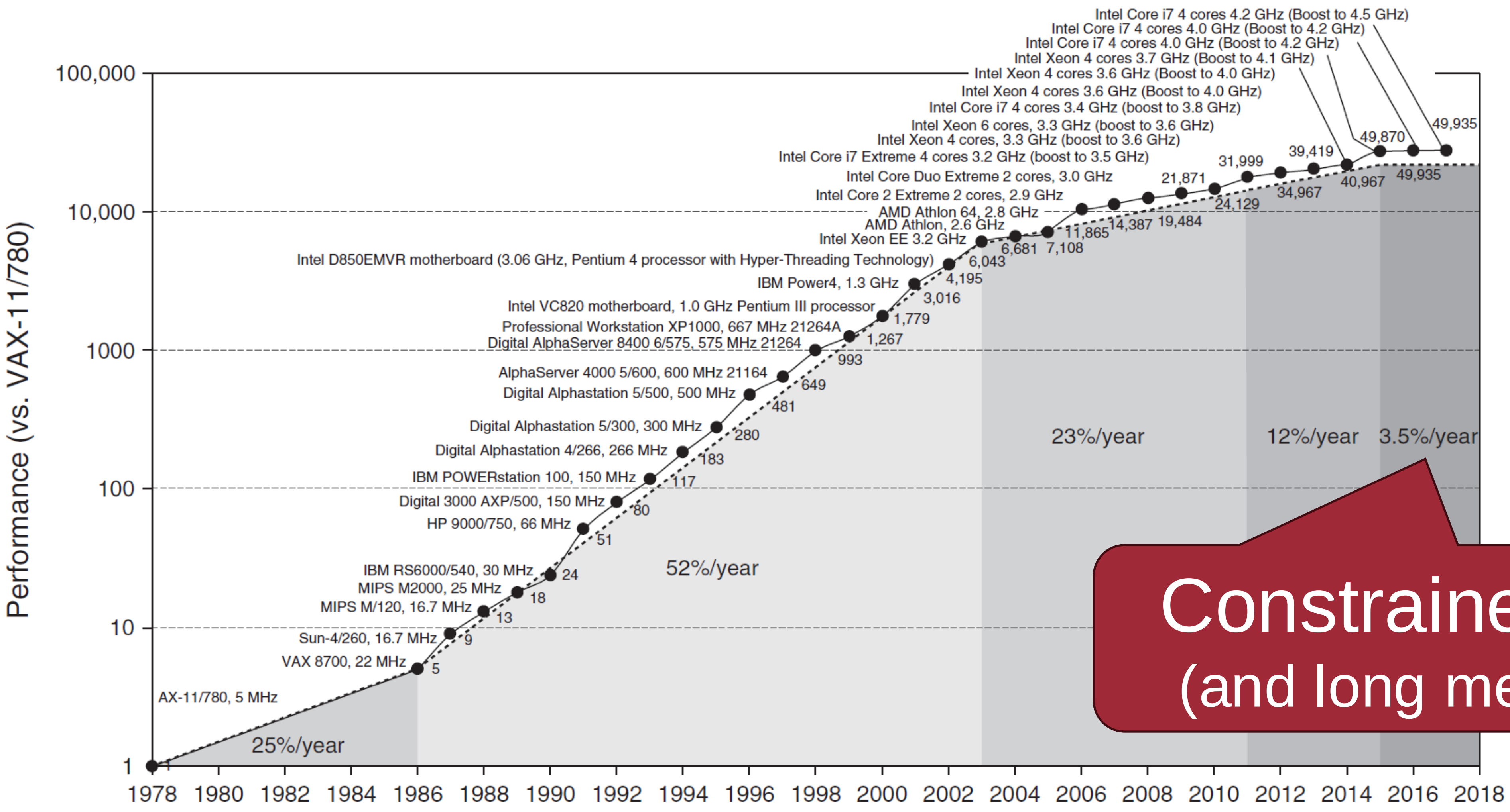
- We can't reduce voltage further due to leakage power
- We can't remove more heat due to costs and complexities



Uniprocessor Performance



Uniprocessor Performance



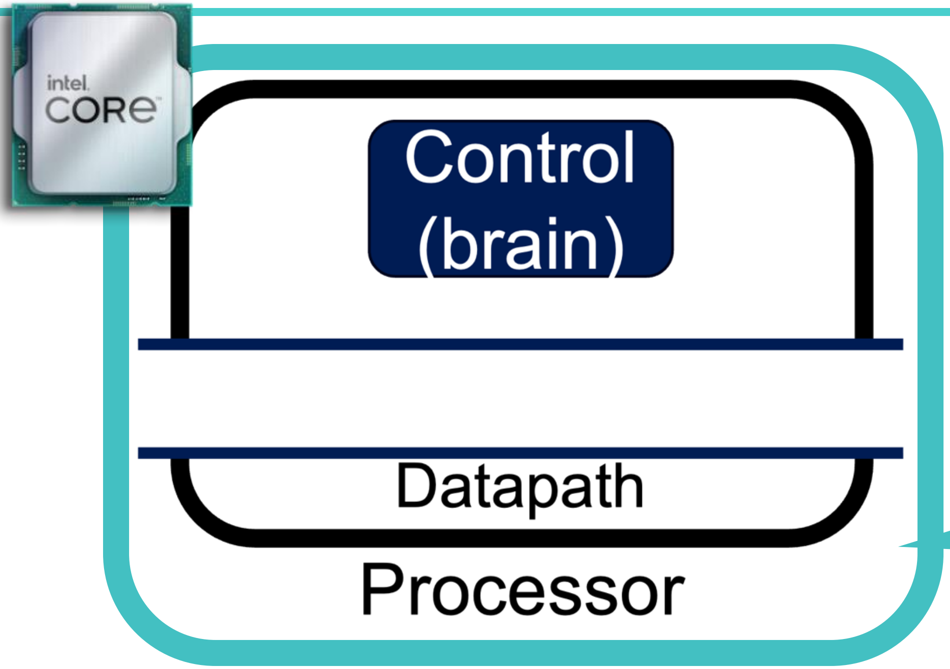
Constrained by power

(and long memory latency)

How we address the power wall?

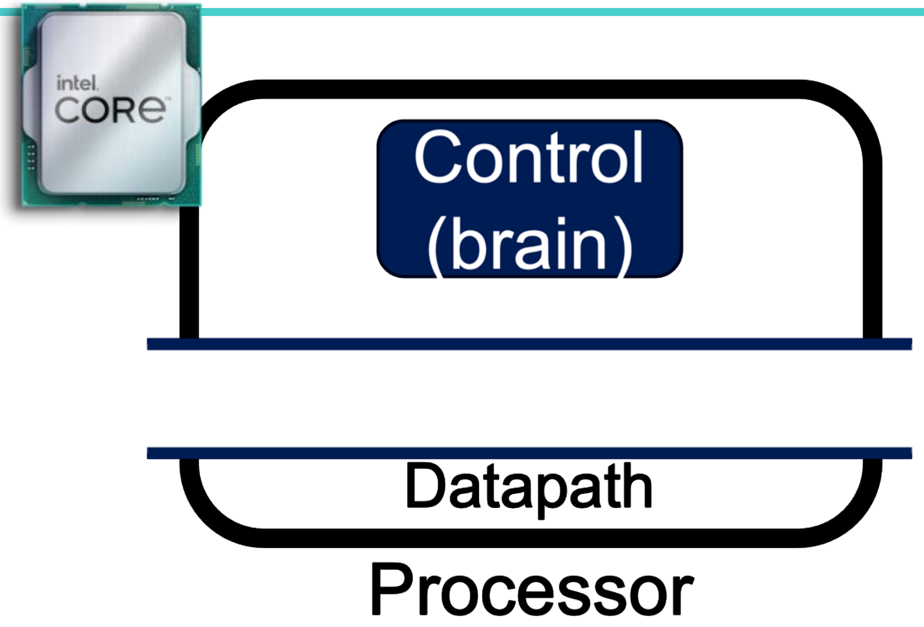
⇒ Multiprocessors!

Uniprocessor to Multiprocessors

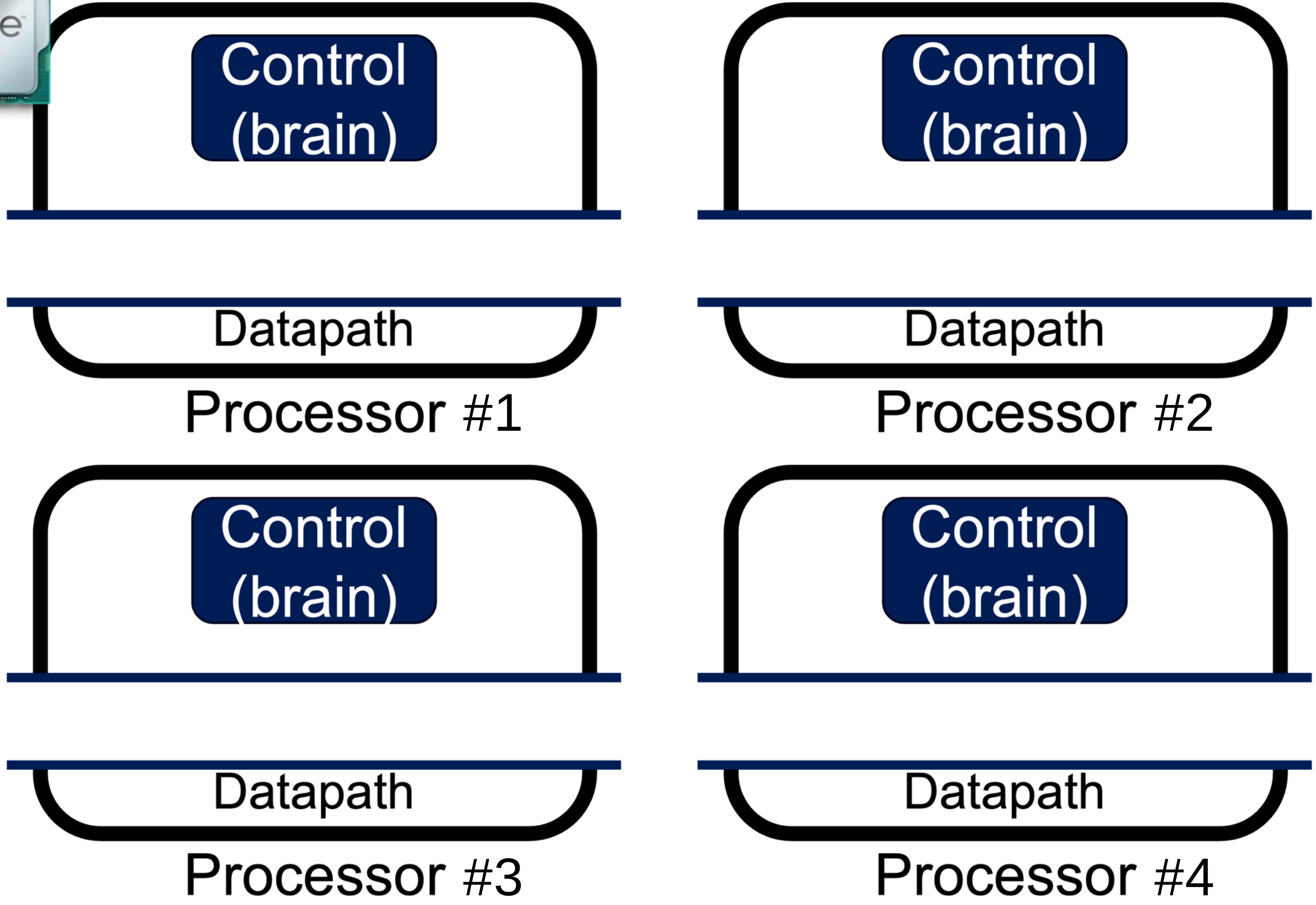


Uniprocessor: one processor (core) per chip

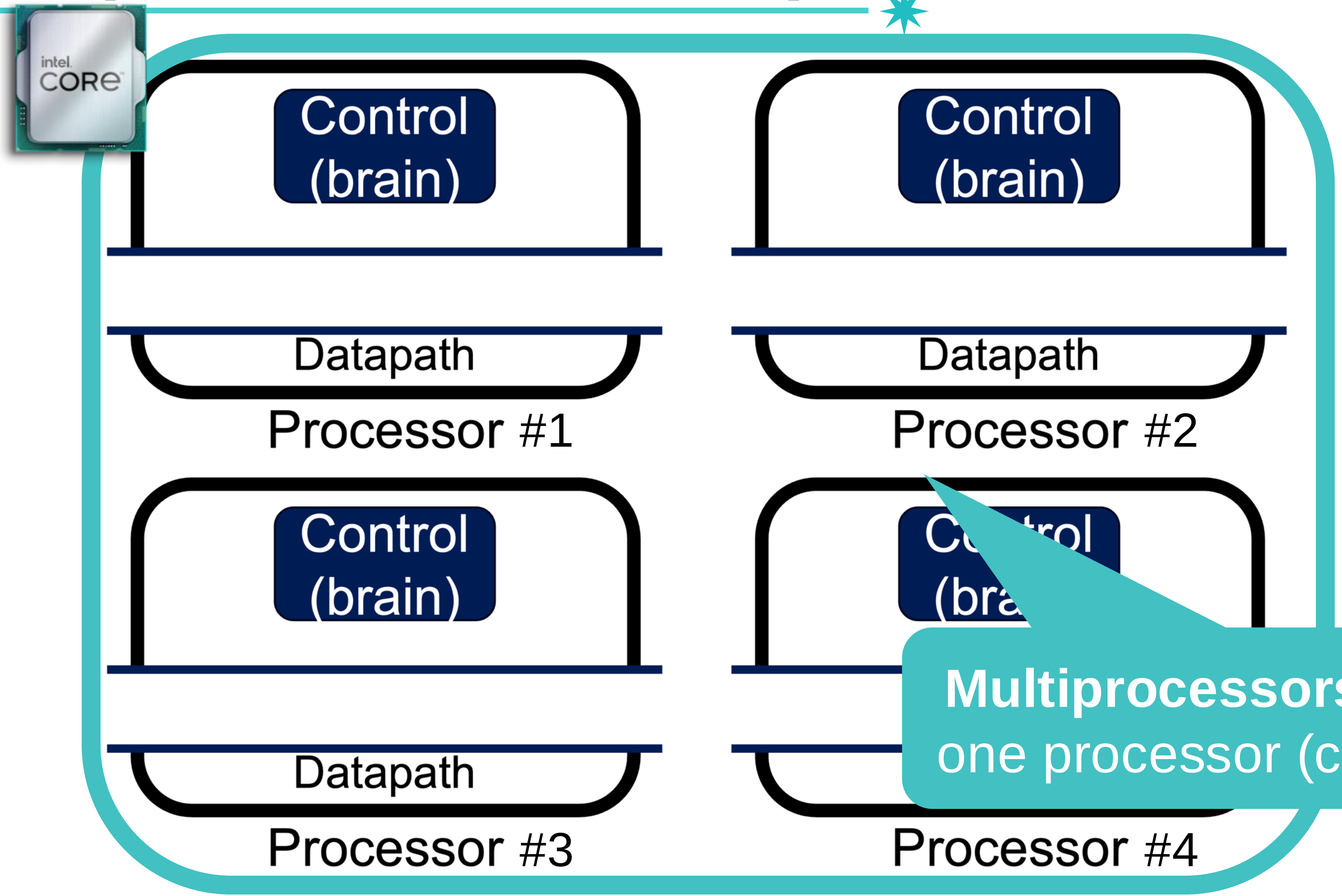
Uniprocessor to Multiprocessors



Uniprocessor to Multiprocessors



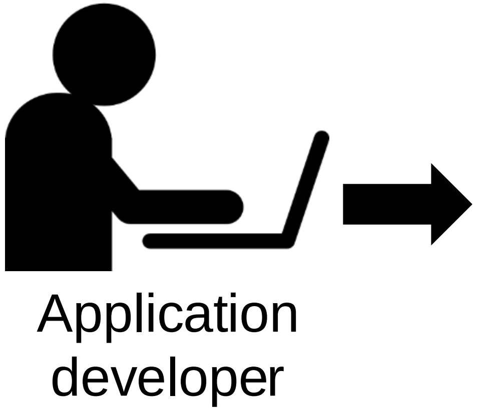
Uniprocessor to Multiprocessors



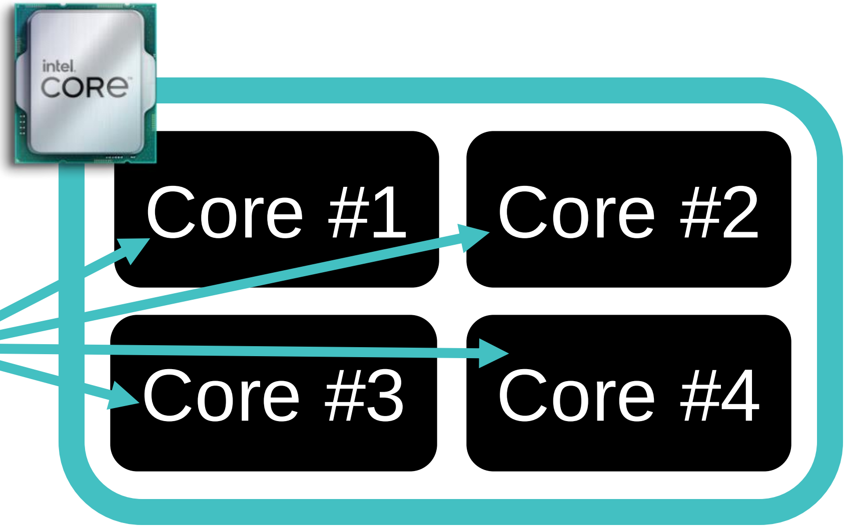
Multiprocessors: more than one processor (core) per chip

Multiprocessors

- Multicore microprocessors
  - More than one processor (core) per chip
- Requires explicitly parallel programming
  - Programming for performance
  - Load balancing
  - Optimizing communication and synchronization



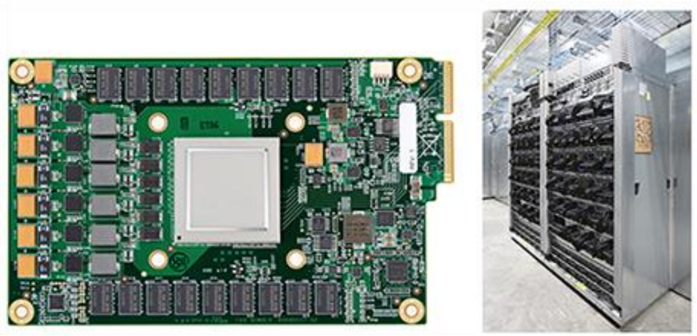

```
import multiprocessing

multiprocessing.pool(processes=4)
...
```



Recent Evolution of Computer Architecture

- GPGPU (General Purpose GPU)
  - Suited to embarrassingly parallel problems
  - Matrix and vector computation

- Special purpose HW
  - Google's TPU (Tensor Processing Unit)
  - Microsoft's Brainwave



Google's TPU and its datacenter

MS Project Brainwave

Conclusion

What will You Learn in This Course?

- Understand general ***principles (NOT*** about learning coding skills***)***

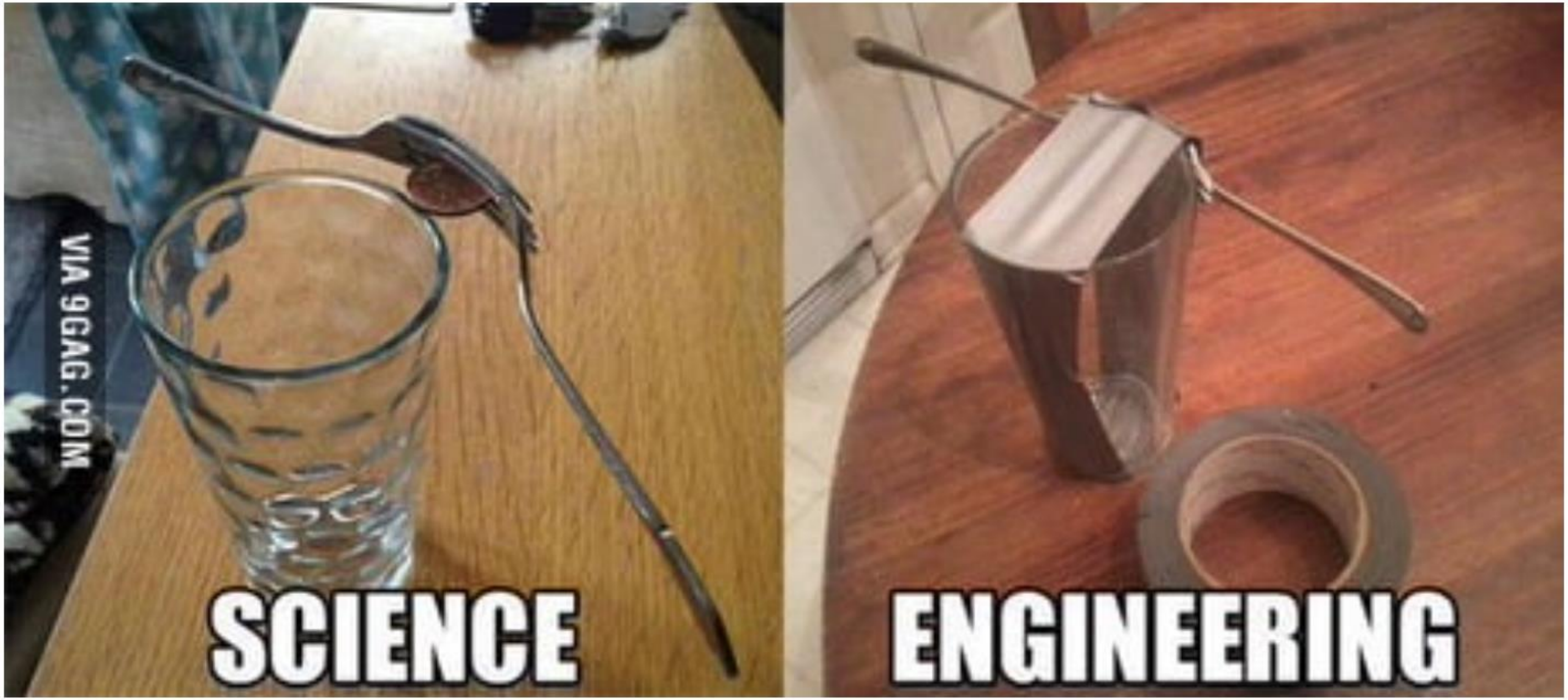
- How programs are translated into the machine language
  - And how the hardware executes them!

- Instruction Set Architecture
- Below of the Instruction Set Architecture

- What determines program performance
  - And how it can be improved

Why Learn this Stuff?

- You want to call yourself a “computer scientist”



Why Learn this Stuff?

- You want to call yourself a “computer scientist”
- You want to build software that people use (requires performance)
- You need to make a purchasing decision or offer “expert” advice

Summary

- Abstraction is fundamental to understanding computer systems
  - In both hardware and software
- ISA is an interface between SW and HW
- Performance metric: CPU Time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Power is a limiting factor
  - Use parallelism to improve performance

Question?