Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei

gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
October 8, 2018

Lecture 0: Preliminaries

Course Structure

- This course is made of **3** distinct modules, each one covering a specific set of topics:
    - **Python Programming (for Data Science)** (40 hours, taught by **Dr. Gabriele Tolomei**);
    - **Database Technologies** (24 hours, taught by **Dr. Nicolò Navarin**);
    - **Computer Networking** (32 hours, taught by **Dr. Armir Bujari**).

Course Info

- We use **Moodle** for sharing communications and materials for this course (lectures, exercises, etc.).
- **NOTE**: If you haven't already done it, please subscribe to the Moodle's class page at the following address: https://elearning.unipd.it/math/course/view.php?id=321
- You are encouraged to ask for a meeting with the teacher in case you need any clarification: just drop us an email, and we will do our best to satisfy your request (provided you do it with a *reasonable* notice!).

This Module's Objectives

- This module is meant to teach you the **fundamental skills of Python programming** with a special focus on data science tasks.
- Firstly, you will learn the **basics** of Python programming :)
- On top of the above, you will learn the "nuts and bolts" of *manipulating*, *processing*, *cleaning*, and *crunching* **data with Python**.
- Ultimately, you will be equipped with the toolbox you need to become a *real* **data scientist**!

Course Prerequisites

- Fundamentals of (von Neumann) **computer architecture** (*CPU*, *memory hierarchy*) and **operating systems** (*program* vs. *process*).
- Very basic **coding** skills (not necessarily in Python): *variables*, *assignment*, *function call*, etc.
- Some familiarity with **Unix-like shell** commands.
- A laptop! (If some of you don't have one, please let me know and we will find out a solution).

Plus

- Later on, we may occasionally need to refer to common data science concepts, methodologies, and techniques:
    - **Probability and Statistics**: probability distributions, random variables, expectation, mean, variance, sampling, tests of statistical significance, etc.
    - **Machine Learning**: supervised/unsupervised learning, training/test set, bias-variance tradeoff, learning algorithms, etc.

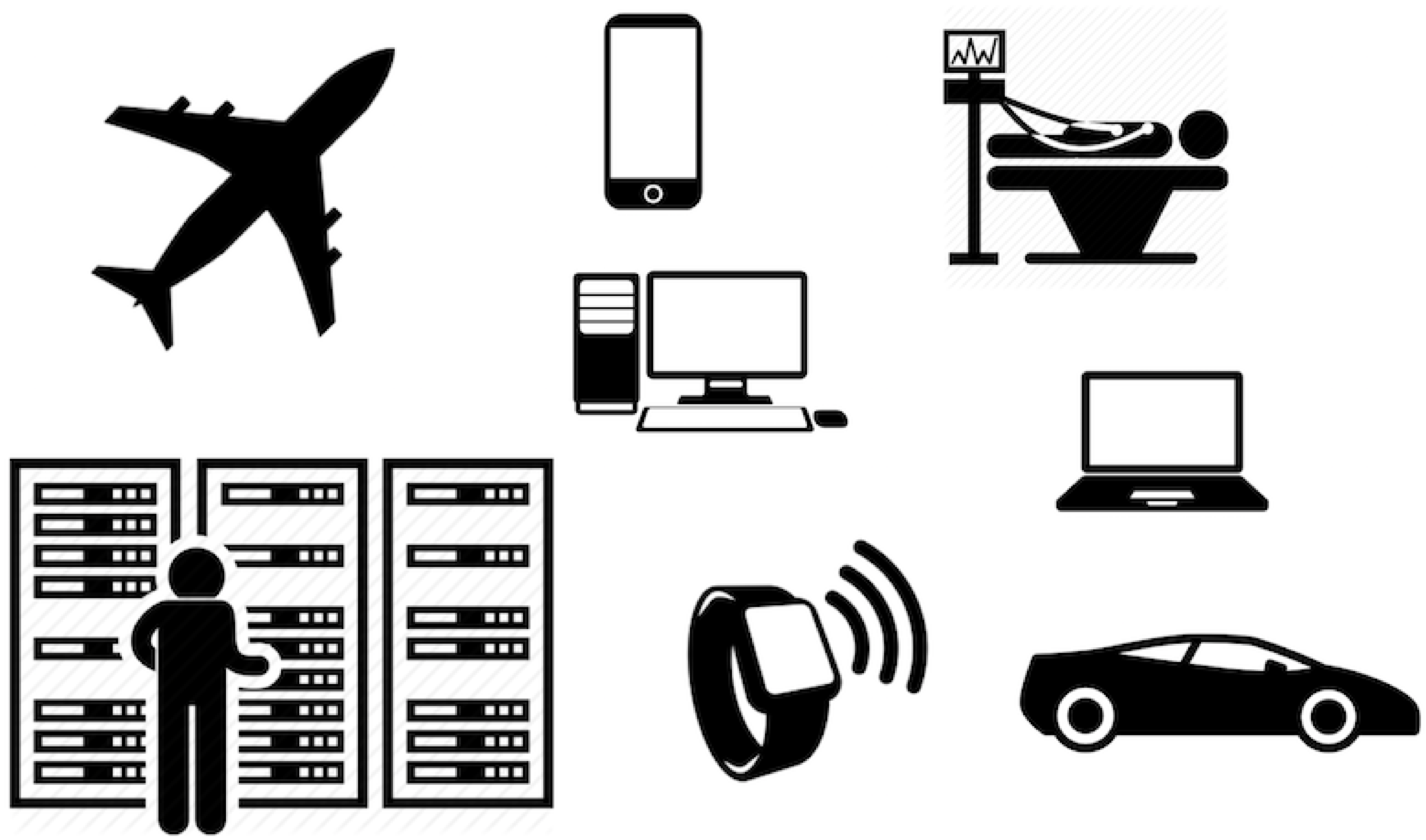
Exams

- A **single**, **unified** written test divided into **3** sections, i.e., one for each module.
- How each section is organized depends on the module it refers to;
- For instance, concerning ***this*** module you may expect to be asked to solve **coding exercises** and, possibly, answer **theoretical questions**.
- **Don't worry now**! There will be time to discuss about exams many times in the future!

(Quick) Recap

- How computer works?
  - von Neumann computing model: **CPU** + **RAM** + **I/O**
- Abstraction layers (from the "physical" machine)
  - from machine language to higher-level languages (e.g., **C/C++**, **Java**, **Python**)
- What does *actually* mean programming a computer?

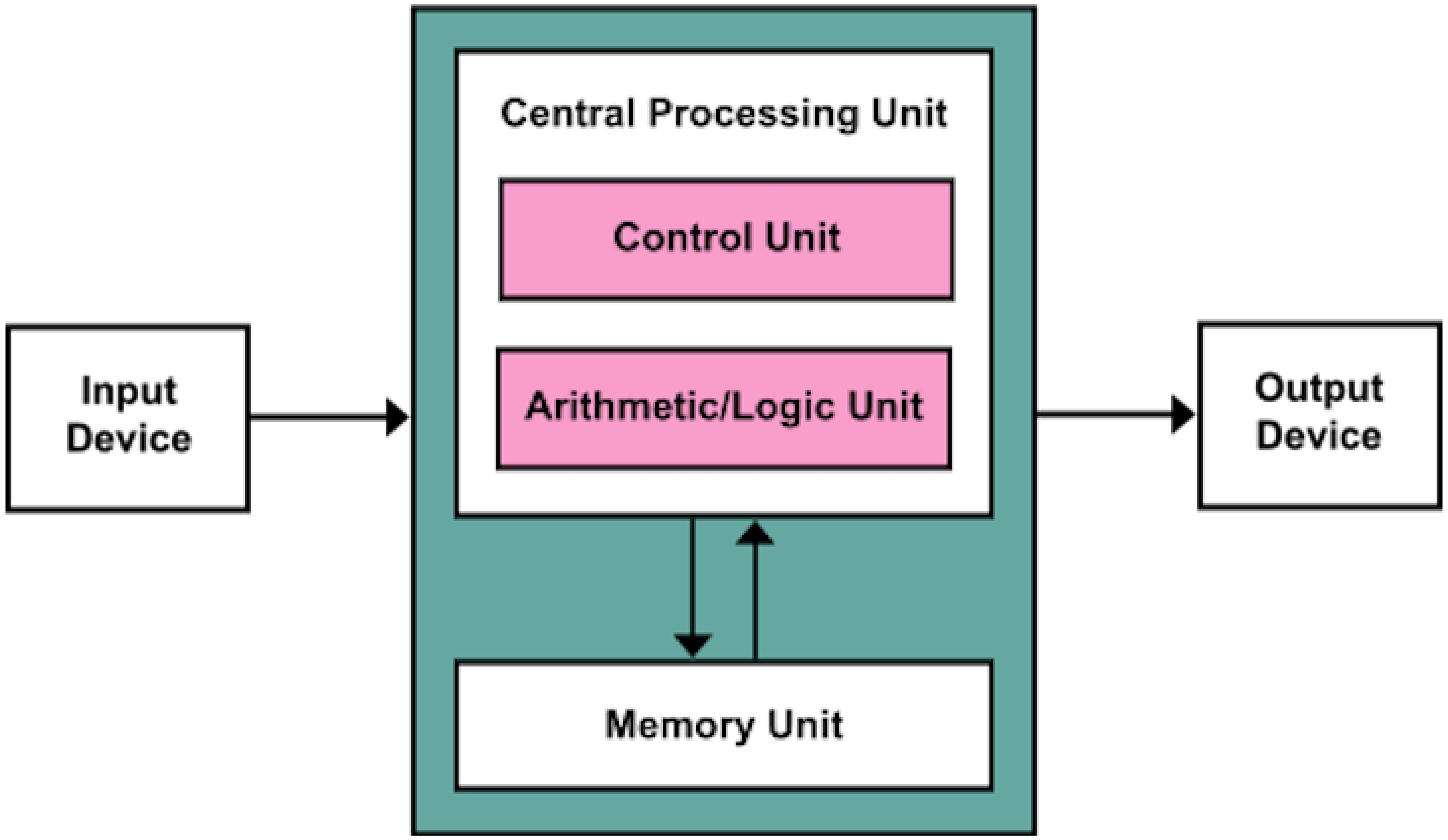
Computers are *everywhere*, seriously!



Conceptually, they are all the same!

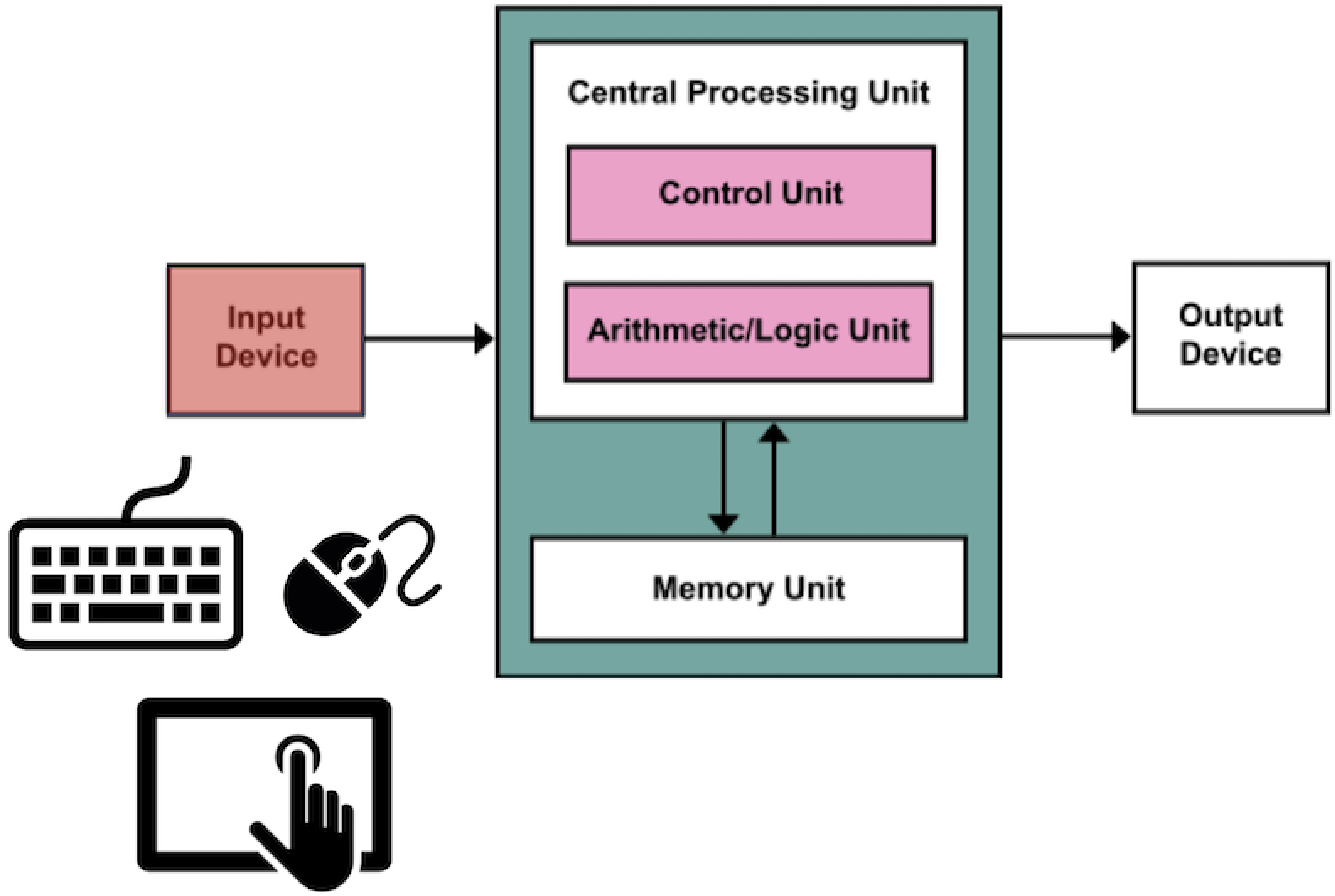
They all follow the same architectural model introduced by **John von Neumann** back in 1945

von Neumann's Computing Architecture

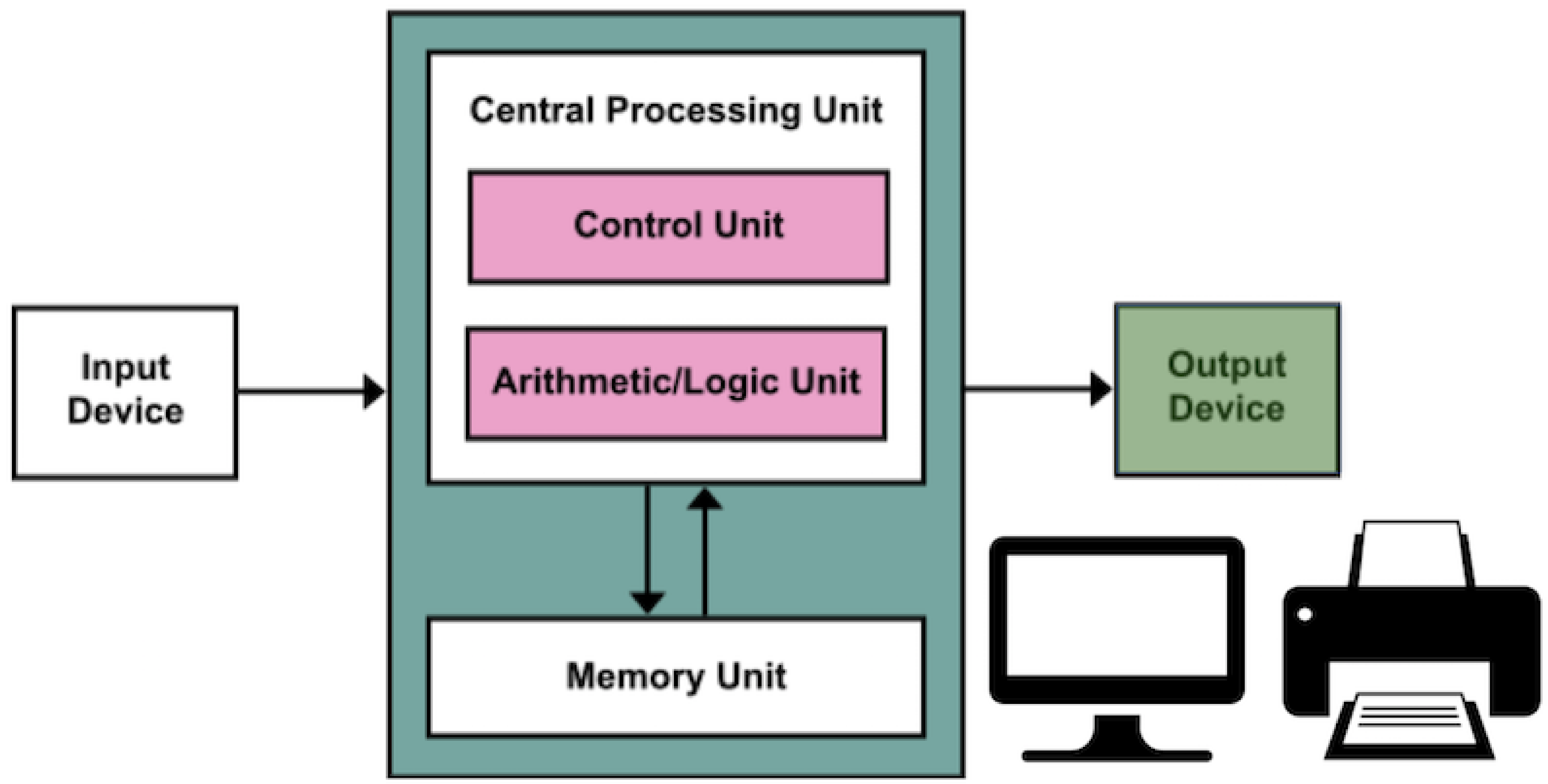


Source: Wikipedia

von Neumann's Computing Architecture: Input

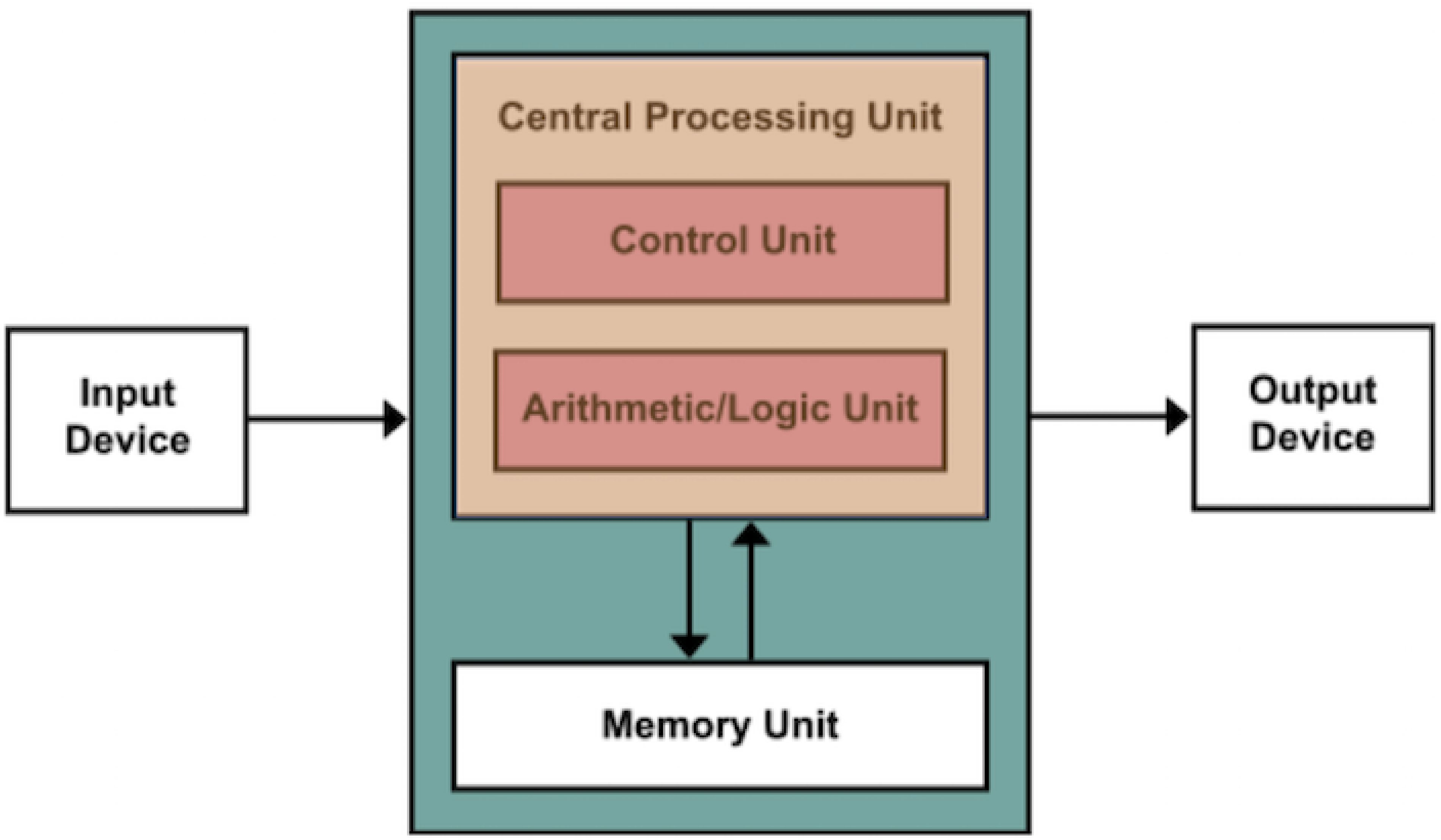


von Neumann's Computing Architecture: Output



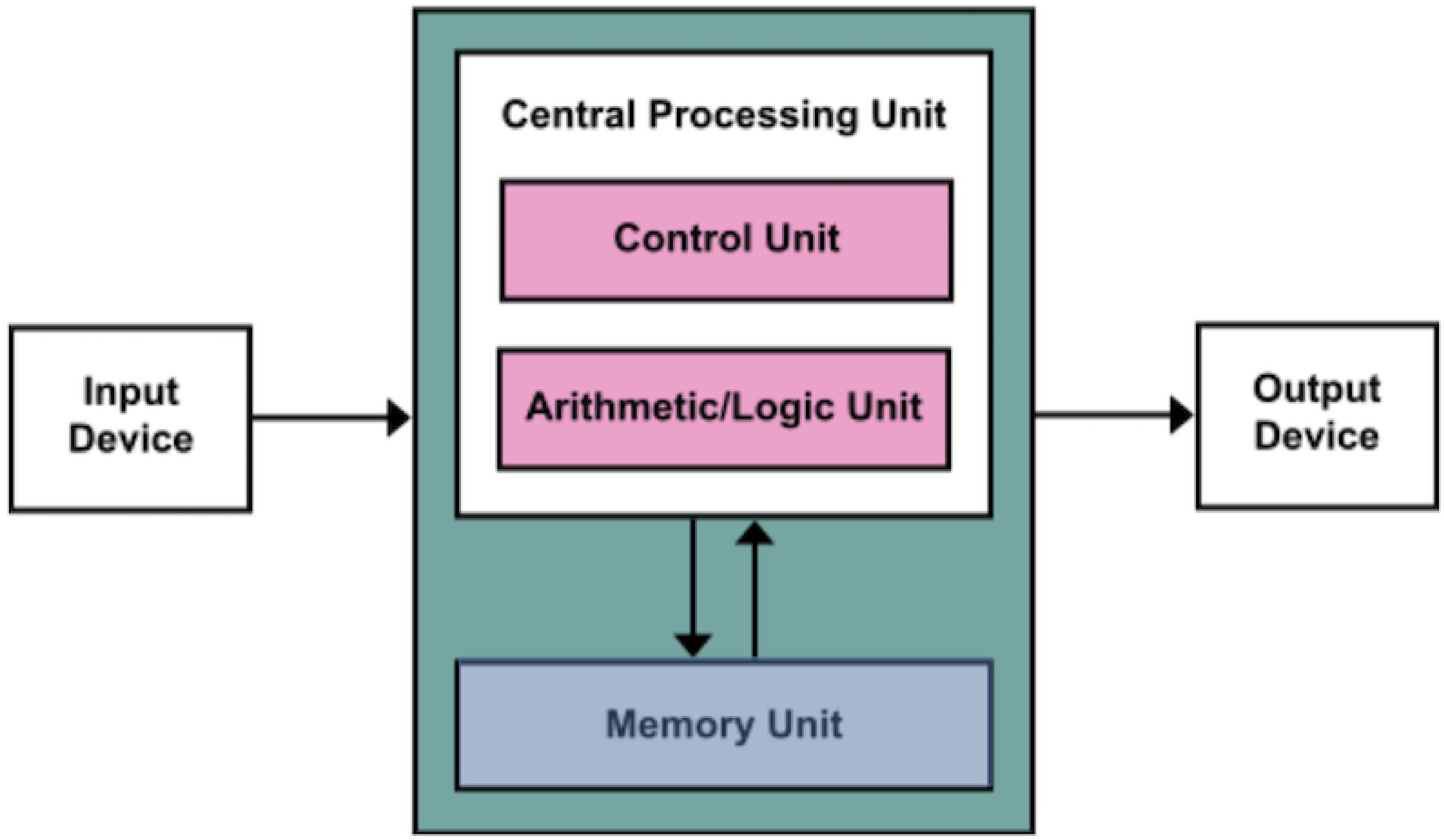
von Neumann's Computing Architecture: CPU

- The **C**entral **P**rocessing **U**nit (**CPU**) is meant of executing *sequences of instructions*, one instruction by the other.
- Each instruction encodes basic arithmetic and logic computations, using CPU's internal registers.



von Neumann's Computing Architecture: RAM

- **R**andom **A**ccess **M**emory (**RAM**) contains *instructions* and *data*, which instructions operate on

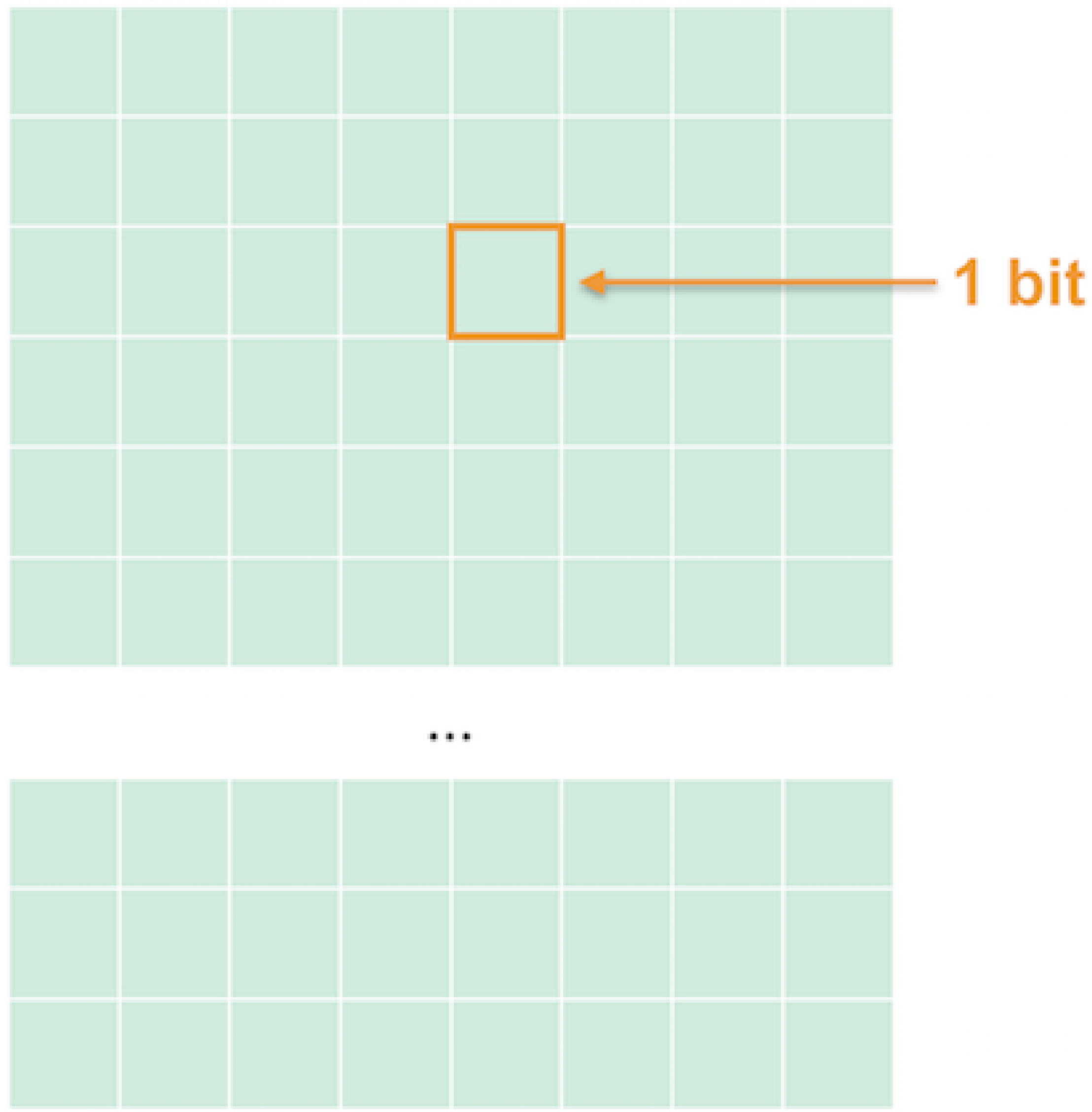


A Closer Look into Main Memory (RAM)

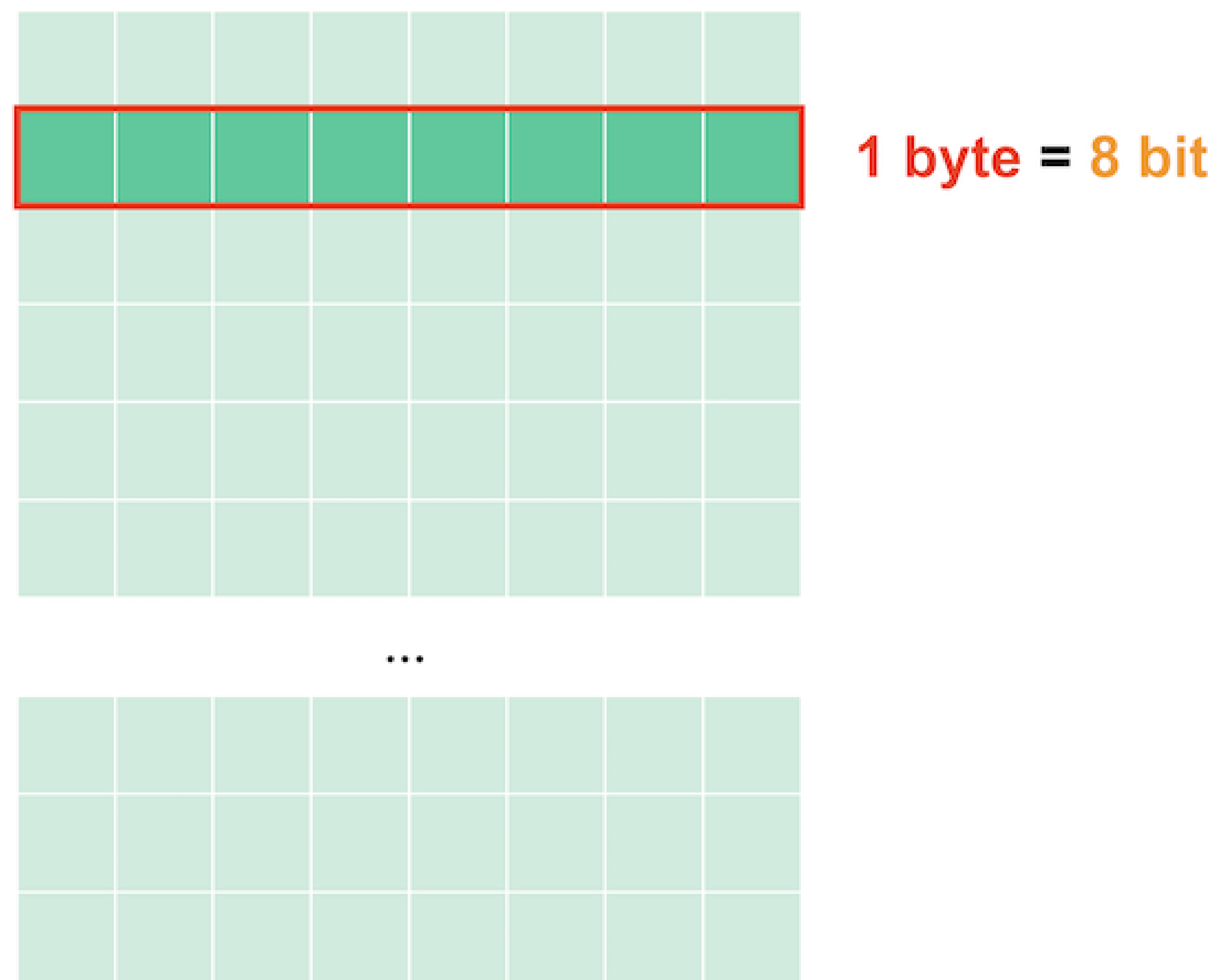
- Represented as a sequence (i.e., array) of contiguous **cells**, a.k.a. **locations**.
- Each memory cell is logically organized into groups of **8 bits** (**1 byte**) each, or multiple of it (e.g., 32 bits = 4 bytes).
- Each cell is uniquely identified by its own **memory address**.
- CPU and I/O units may read from/write to main memory by specifying memory address.
- Addressing is usually performed at the single byte level.

Binary Digit (Bit)

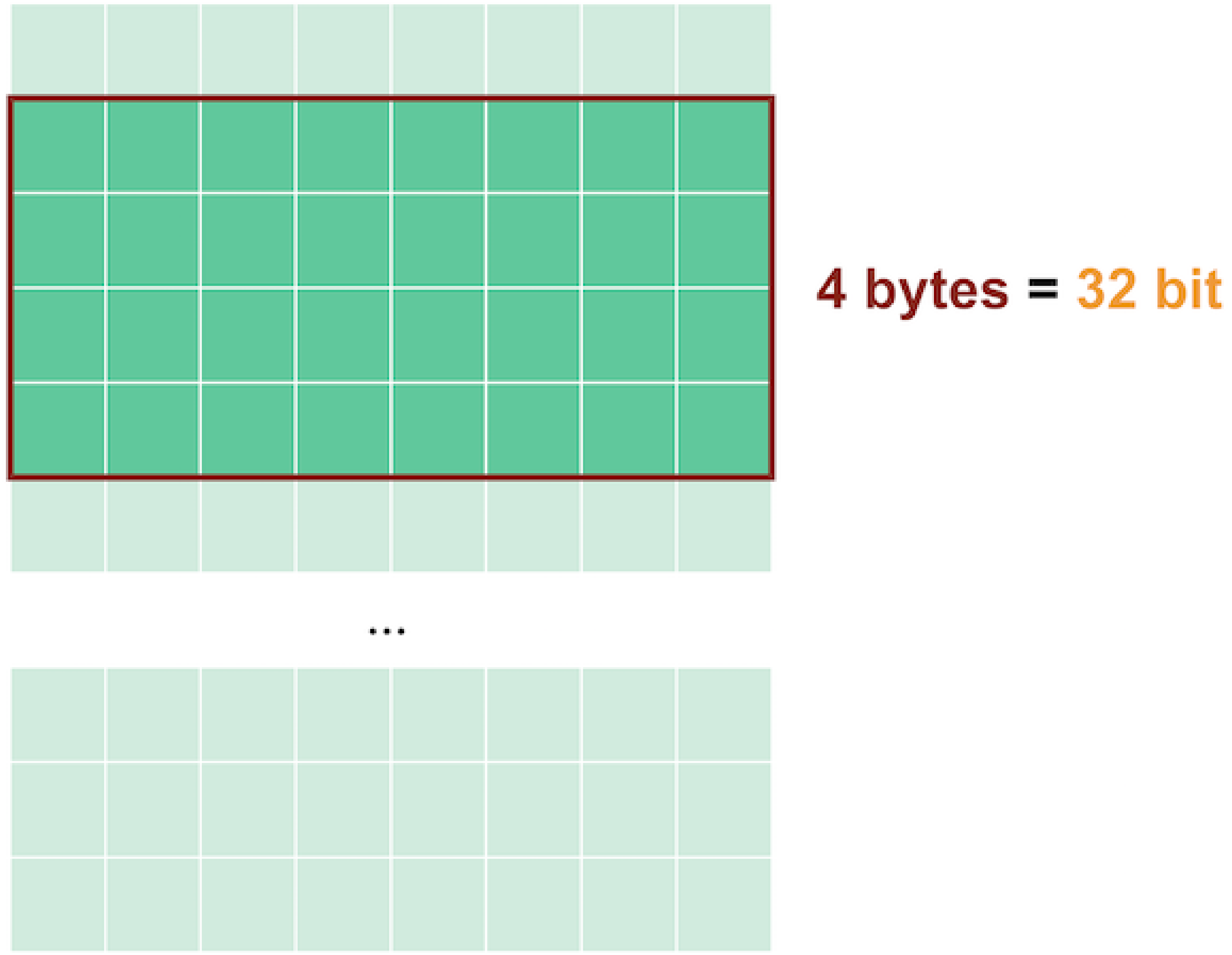
- Can take on **2** possible values: `0` or `1`.
- Suitable encoding to represent the smallest amount of information (e.g., voltage of digital circuits).



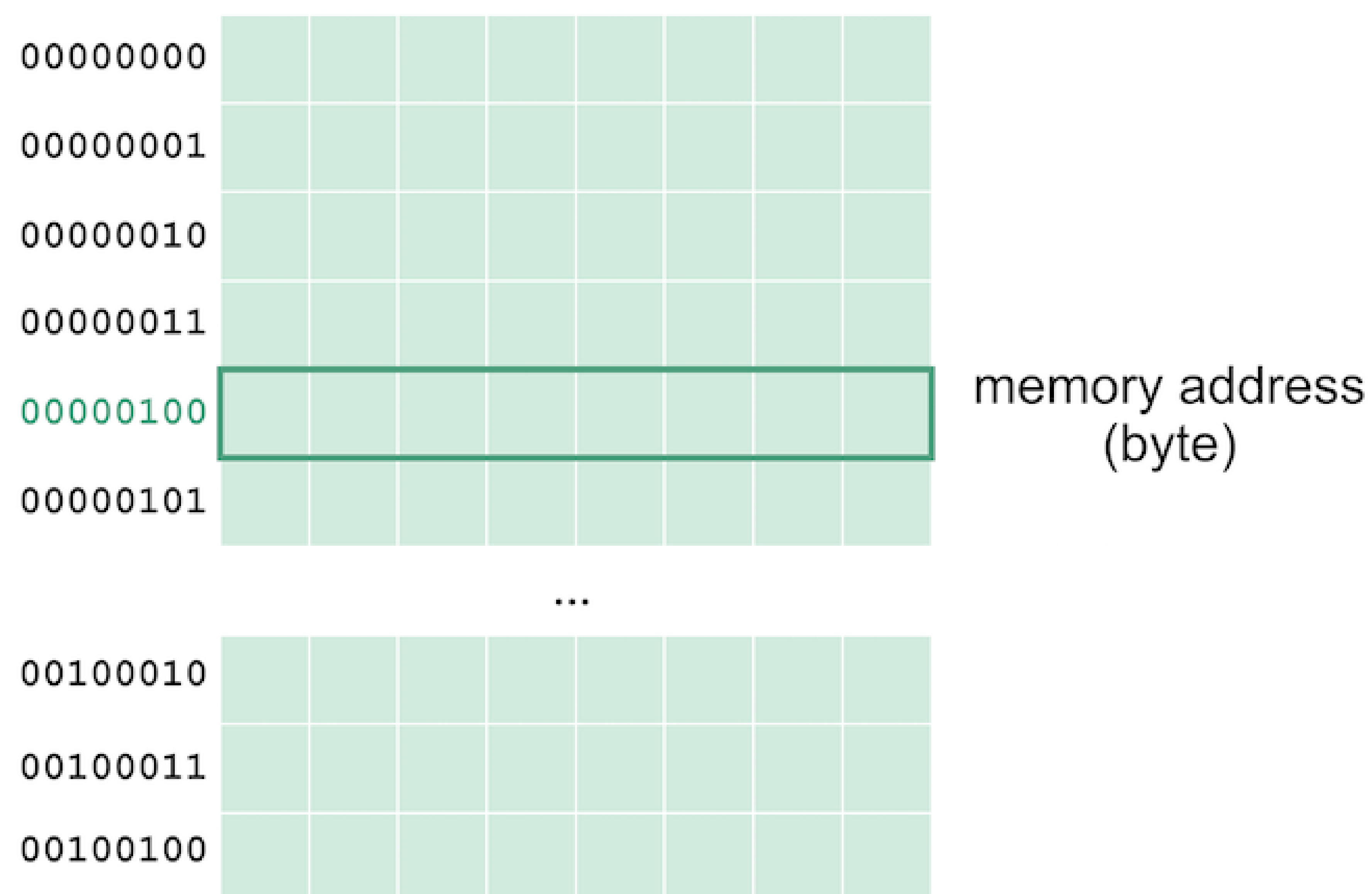
Memory Cell/Location



Memory Cell/Location



Memory Address



CPU Interpreter

- The CPU realizes an **interpreter** which cyclically does the following **3** operations:
  - **Fetch**: retrieve from main memory an instruction which is stored at a specific address whose value is contained in a dedicated CPU register, called **Program Counter**;
  - **Decode**: decode the retrieved instruction;
  - **Execute**: execute the decoded instruction.

Machine Language

- Defines a set of (elementary) instructions which the CPU interpreter is able to execute directly.
- Such a language is expressed using **binary numerical system**.
- In other words, each instruction of a specific machine language must be encoded as a **sequence of bits**.

Binary vs. Decimal Numerical System

- In the **decimal numerical system** (base 10), each digit can take only one out of **10** possible values: `0`, `1`, ..., `9`.

1	0	1
10^2	10^1	10^0

$$1*10^0 + 0*10^1 + 1*10^2 = 101$$

- In the **binary numerical system** (base 2), each digit is a bit:

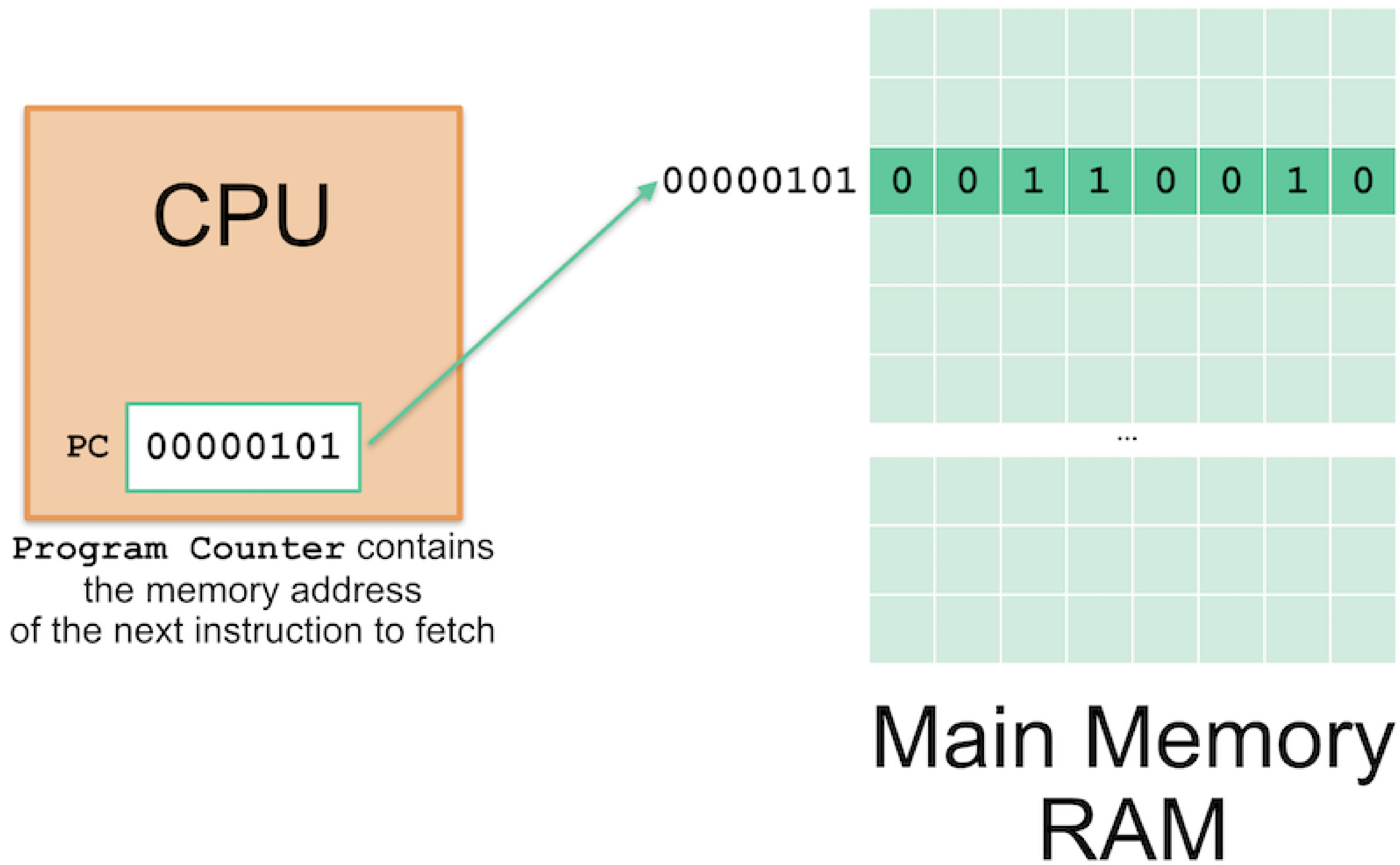
1	0	1
2^2	2^1	2^0

$$1*2^0 + 0*2^1 + 1*2^2 = 5$$

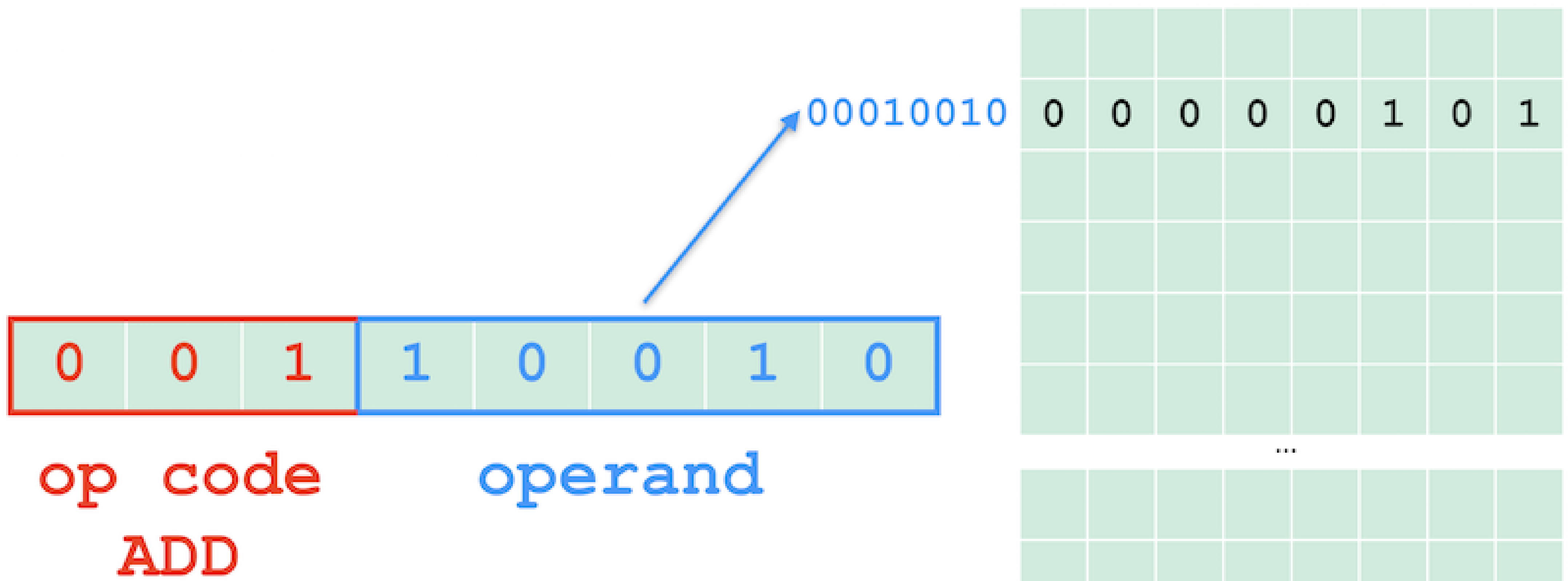
Machine Language: Specifications

- Instructions defined by a machine language are composed of **2** parts:
  - An **operator** (`op code`);
  - One or more **operands** representing either CPU's internal registers or main memory addresses.
- The collection of instructions defined by a certain machine language (i.e., **instruction set**) is specific to a hardware implementation: e.g., *Intel x86*, *ARM*, *Sparc*, *MIPS*.
- In a nutshell, machine language indicates the number of bits each instruction dedicates to the operator and operands.

CPU Cycle: 1. Fetch



CPU Cycle: 2. Decode



Add the value stored at the memory address contained in the operand with that of a CPU's internal register (**accumulator**)

CPU Cycle: 3. Execute

73	0	1	0	0	1	0	0	1	Accumulator
					+				
5	0	0	0	0	0	1	0	1	Operand
					=				
78	0	1	0	0	1	1	1	0	Accumulator

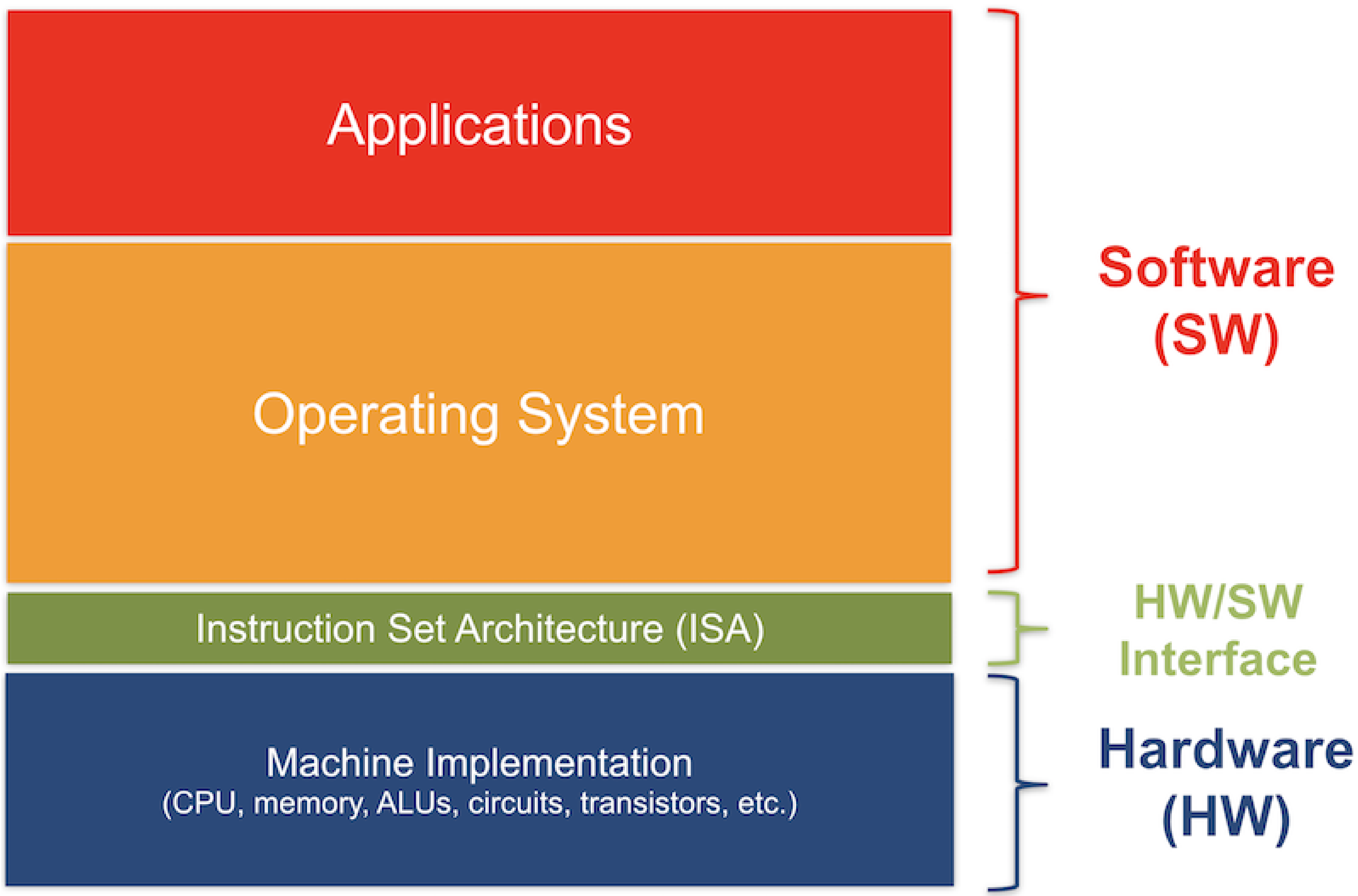
Instructions vs. Data

- According to the von Neumann's architecture, memory cells contains **both** instructions and data.
- CPU wouldn't be able to distinguish between those by simply "reading" a bit sequence stored at a given memory address.
- The **Program Counter** register allows separating instructions from data as it *always* contains the memory address of an instruction.

Back to Machine Language

- (Binary) Machine language indicates how to write a (representation of a) **program** which is the closest to the bare metal machine (hardware).
- Theoretically, we could write programs using machine language instructions, directly.
- In practice, though, this would be totally insane! (Think about how complex are programs running on our computers, smartphones, etc.).
- **SOLUTION**: abstracting low-level machine language using higher-level languages that are closer to our natural language.

Abstraction Layers



Abstraction Layers

- Each layer is associated with a **language** (which is adopted by *that* layer).
- Every functionality (of the language) of a layer is implemented by a **program** which is written using language(s) of the layer(s) below.
- **PRO**:
    - Separation between ***what*** has to be done (*specifics*) and ***how*** this has to be done (*implementation*)
- **CON**:
    - The more we abstract from the physical machine the less will be the control we will have over it (delegated to lower layers)

What Does Computer Programming Mean?

- Generally speaking, it means writing a program using a high-level language to solve a given problem/task (e.g., find the minimum element of a set of integers).
- In this module, we will be using **Python** as the high-level language to write down our programs.
- Code written using a high-level language is usually referred to as **source code**, and it ***cannot*** be directly executed by the computer.
- **REMEMBER:** The CPU (interpreter) can only directly execute instructions that are defined by a specific (binary) machine language.

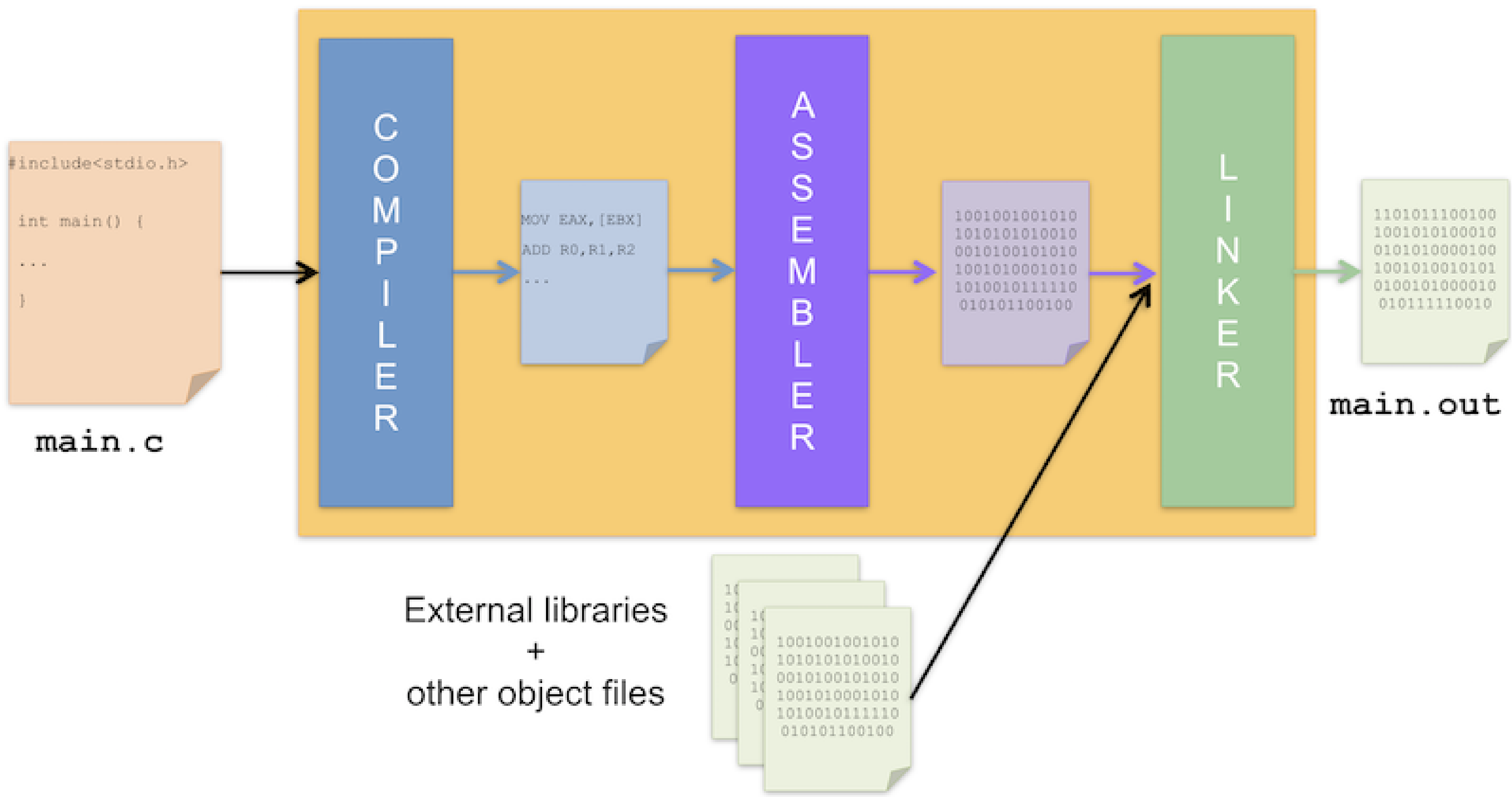
From High-Level Source Code to Low-Level Machine Code

- Different ways of achieving this, which depends on the *implementation* of the high-level language:
    - **Compilation**
    - **Interpretation**
    - **Hybrid**: Compilation + Interpretation

Compilation

- Uses a special computer program, called **compiler**.
- Takes as input a program written in some language (**source code**) and translates it into another language (**target code**): e.g., from C/C++ to assembly.
- Results of compilation is **not** directly executable by the CPU interpreter (although the compiler is tied to a specific CPU): e.g., assembly needs to be further transformed to binary (object) code via **assembler**.
- Eventually, another program, called **linker**, combines multiple object codes and external libraries into a **single** machine executable code.

Example: C/C++



#include<stdio.h>
int main() {
...
}
main.c
COMPILER
MOV EAX,[EBX]
ADD R0,R1,R2
...
ASSEMBLER
LINKER
main.out
External libraries
+
other object files

Interpretation

- Uses a special computer program, called **interpreter**.
- Also interpreters "translates" a high-level language into a low-level one, but it does so at the moment the program is run, one instruction at a time.
- The easiest example of an interpreter is the CPU, which realizes an interpreter of machine language via the **3**-phase cylce: *fetch*, *decode*, *execute*.
- Purely interpreted implementations of high-level languages are now rare (**Smalltalk**, 1980), due to performance reasons.

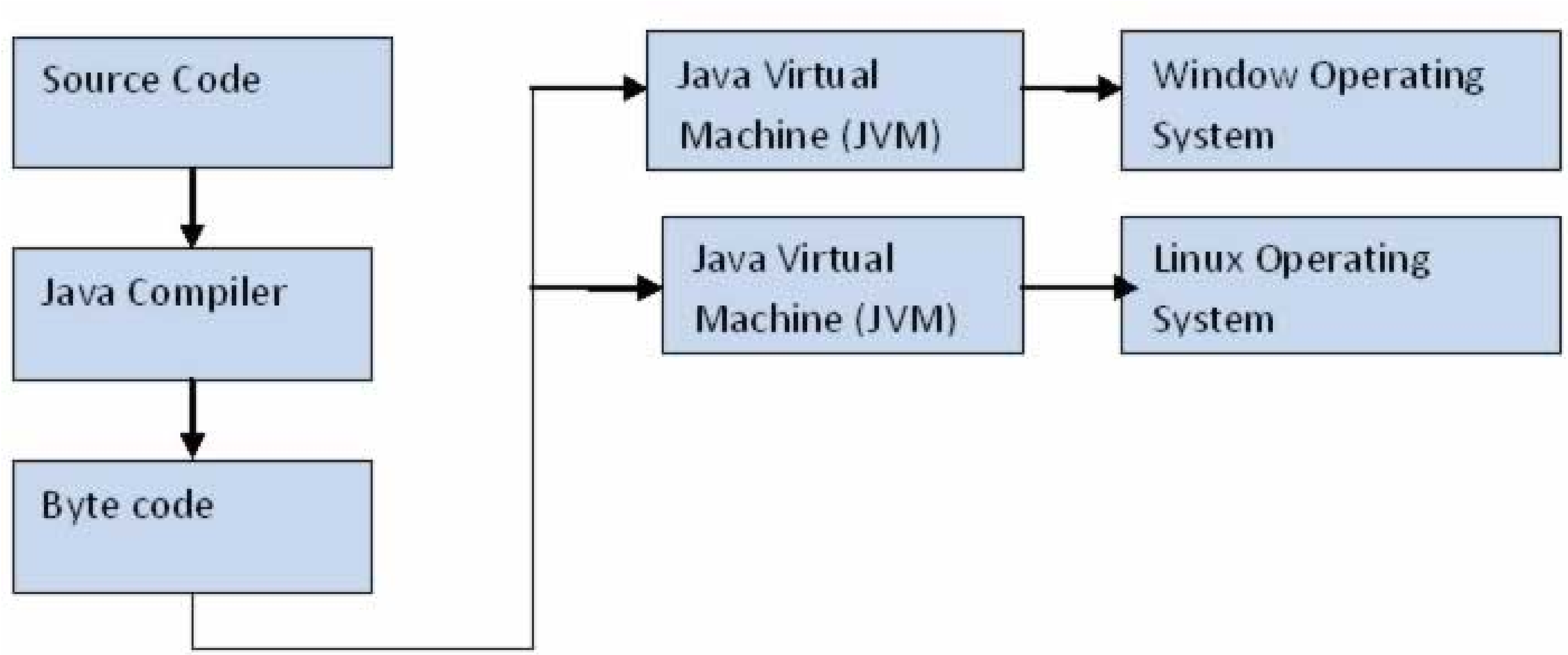
Hybrid: Compilation + Interpretation

- Tries to take advantage of both worlds.
- High-level language is firstly compiled into an **intermediate** language (usually, referred to as **bytecode**)
- Bytecode is not tied to a specific hardware/CPU and can be interpreted (i.e., executed directly) by a so-called **virtual machine** hosted on top of the physical machine

bytecode : virtual machine = machine code : CPU

- Notable examples of high-level languages whose major implementations mix compilation and interpretation: **Java**, **Python**, **Lisp**, etc.

Example: Java



Considerations on Hybrid Approach

- The bytecode interpreter of a virtual machine is itself a program.
    - Oracle's HotSpot JVM interpreter for Java compiled bytecode (implemented in C++ and assembly)
    - CPython interpreter for Python compiled bytecode (implemented in C)
- Hybrid implementations allow **portability**.
- The same Java code can be compiled on a machine (e.g., Windows/Intel x86) and run everywhere (e.g., Linux/SPARC) as long as there is an implementation of the JVM bytecode interpreter.

Take Away Message

- Computational model based on CPU + Main Memory (+ I/O).
- Abstraction layers eases writing computer programs.
- Low-level binary machine language vs. high-level programming languages.
- Different language implementations (even for the same language specifications).