Programming Languages

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se

Outline

How to use these slides

- **Build vocabulary**: As a reference whenever you don’t understand what a word mean
- **Review concepts**: whenever a concept is explained during tutorials, bring these along and search for it
- **Future reference**: if anything about computers rings a bell but you don’t remember what it is, check here.

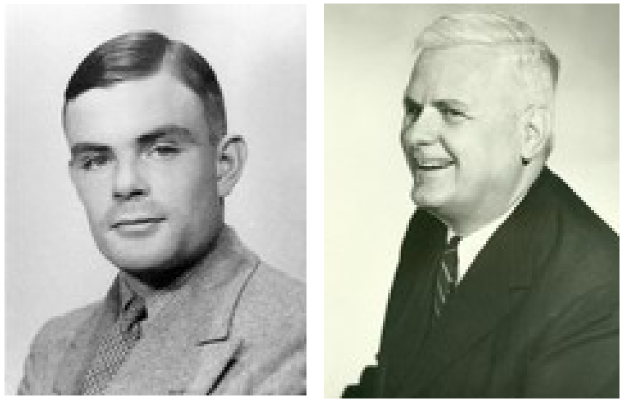
General concepts in programming

- **Programming** is the process of writing a **computer program,** that is, *translating an idea* into something that can be **executed** by a computer.
- This *translation* happens in several steps and, like a recipe for cooking a meal, one needs to understand the *ingredients* and how to *mix/cook* them.
- The *idea* usually takes the form on an **algorithm**.

Ingredients of programming: What is an **algorithm**?

- A **finite sequence of instructions** to carry out a task or solve a problem.
- An algorithm can be written in natural language or in mathematical terms.
- The term is derived from the name of the Islamic scholar Al-Khwarizmi.

Ada Lovelace, First programmer in history


Alan Turing, Alonso Church Hypothesis on computability

Ingredients of programming: Code

- Code or *source code*
  - Is a **structured description** of an algorithm, it determines what a program will do
  - It is usually stored in digital format on one or more **files**
  - The description is usually done via a **programming language**
    - It is called **language** because one must respect several *grammar rules*, like in spoken or written natural human languages.

From algorithm to code

- The **translation of an algorithm into code**, using a programming language, is called **implementation**
- The transition between an algorithm and and its implementation can have an intermediate representation that is still human readable, which mixes natural language and programming language. This is often called **pseudo-code**.
    - Writing pseudo-code is one of the best techniques to implement an algorithm, although can be time consuming.

What is source code like?

- It is a list, a **sequence of statements**, also called **lines of code**.
- These statements usually come in a defined **structure**, that is, **an order** in which one should write them
- It can be stored digitally in one or more text **files**
- It can refer to other programs or program components, often called **libraries**

Ingredients of programming: Code example

Code might look weird at first. But there is a strive to make it human-readable. Consider the following example of **C** code, what do you think it does?

```
printf ("%s \n", "Hello World!");
```

Ingredients of programming: Code example

Yes, it `prints` on screen the text *string*

```
Hello World!
```

Let's analyze the components of the language **statement**:

```
printf ( "%s \n", "Hello World!" );
```

Issue a command:
function or procedure `printf();`

Command argument:
two function arguments

1. Formatting information:
   - “`%s \n`” means “I want you to print a string (`%s`) and then go to next line (`\n`)
2. Content information:
   “`Hello World!`” is the actual thing to print.

Grammar syntax:
<function name>**(**<argument or parameter>**);**

WARNING:
NOT A MATHEMATICAL FUNCTION!!!!

How humans count: the decimal system as a language

- Our way of counting numbers is based on the decimal notation. It is called **decimal** because is is based on **10** basic symbols:
  0 1 2 3 4 5 6 7 8 9
- The decimal **notation** is **positional**. The position represents the powers of the base (that is, the number of basic simbols)
  - Each position starting from the rightmost represents how many times a base elevated at a given power is multiplied by itself. The powers belong to the set of Natural numbers, starting from 0.
- Example:
  $2048 = 2*10^3 + 0*10^2 + 4*10^1 + 8*10^0 =$
  $2*1000 + 0*100 + 4*10 + 8*1 =$
  $2000 + 0 + 40 + 8 = 2048$

How computers count: the binary system as a language

- In a computer everything is based on the binary system. That means, the number of symbols of the binary notation is just **2**:
  0,1
- The binary **notation** is **positional**. The position represents the powers of the base exactly like the decimal one. The difference is that we can only multiply by 0 or 1.
- Example:

```
1101 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =
       1*8   + 1*4   + 0*2   + 1*1   =
         8   +   4   +   0   +   1   = 12 (decimal!)
```

Why binary?

- Digital circuits are based on mapping **voltage** to **information**
- Measuring voltage can be error-prone, so one must minimize the error
- Years of engineering studies showed that the safest choice is either to have three voltage states or two
- Two proved to be safest and easiest to handle as the number of circuits on a circuit board grows: they interfere with each other! (magnetic fields etc)
- Modern computing sets the voltage difference to be $\mp$5V
- Mapping: $\mp$5V = 0, 0V = 1 (yeah, I know, misleading. But there are practical reasons for it. We don't have to care.)

Mapping things to binary

- In a computer, a sequence of bits contained in a memory chunk can be one of:
  - **Boolean expression** (True/False) or binary string
  - Number or **Value**
  - Operation or **Instruction**

Binary Logic: Boolean Algebra

- Here I describe the so called “first order logic”, where a **preposition** is assigned a **truth value**.
  - We will denote prepositions with capital letters P, Q.
- Only **two** possible **values**:
  - 1 or T as **True**
  - 0 or F as **False**
  - Unfortunately the above is sometimes the opposite for some languages.
- Few **operators**:
  - ¬, **negation**. Unary. Ex: ¬P.
    - Code: usually **!** or **!=**
  - ∧, **AND**, conjunction. Binary. Ex: P AND Q
    - Code: usually **&** or **&&**
  - ∨, **OR**, disjunction. Binary, P OR Q
    - Code: usually **|** or **||**
  - ⇒, **implication**, P ⇒ Q not used in the programming languages we will see in this course.
  - **Parentheses** can be used to compose expressions.

Binary logic: Truth tables

P	NOT P
F	T
T	F

NOT **flips** the binary value

P	Q	P AND Q
F	F	F
F	T	F
T	F	F
T	T	T

AND = 1 ONLY when BOTH P,Q = 1

P	Q	P OR Q
F	F	F
F	T	T
T	F	T
T	T	T

OR = 1 if ANY of P,Q = 1
OR = 0 when BOTH P,Q = 0

P	Q	P ⇒ Q
F	F	T
F	V	T
T	F	F
T	T	T

⇒ = 0 if and only if P = 0
ex falso (sequitur) quodlibet
"from falsehood, anything (follows)"

Binary logic: Truth tables

P	**!P**
0	1
1	0

! *flips* the binary value

P	**Q**	**P && Q**
0	0	0
0	1	0
1	0	0
1	1	1

&& = 1 ONLY when BOTH P,Q = 1

P	**Q**	**P || Q**
0	0	0
0	1	1
1	0	1
1	1	1

|| = 1 if ANY of P,Q = 1
|| = 0 when BOTH P,Q = 0

P	**Q**	**P ⇒ Q**
0	0	1
0	1	1
1	0	0
1	1	1

⇒ = 0 if and only if P = 0

Boolean algebra

- **NOT** *flips* values and operators:
  !(P && Q) == !P || !Q
  !(P || Q) == !P && !Q
  !(!P) == P
- && and || have the following properties: *commutative, associative, distributive* unless otherwise specified.
  - In some programming languages the evaluation might stop depending on associativity.
- Notable other operators are defined on top of those, like
  XOR (eXclusive OR):
  P⊕Q = (P && !Q) || (!P && Q) == applying distributive ==
  (P || !P) && (P || Q) && (!Q || !P) && (!Q || Q) ==
  T && (P || Q) && (!Q || !P) && T ==
  (P || Q) && (!Q || !P) == (P || Q) && (!P || !Q) ==
  (P || Q) && !(P && Q)

Binary logic: XOR an XNOR

P	**Q**	**P XOR Q**	**XNOR !(P XOR Q)**
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

XOR = 1 if P,Q differ
! XOR = 1 if P,Q have exactly the same value

Most programming languages provide a specific operator for XOR...

Note: Usually **==** or **!=** are different from XOR **because they're not binary – they can compare much more than just binary strings.** But XOR might be used behind the scenes in the circuits.

Information as a binary mapping: Memory, Bits and Bytes

- The fundamental unit of measure of information is the **bit** (**bi**nary digi**t**): either **0** or **1**.
- Assume a fundamental memory component of a circuit can store exactly one bit. That means, that component can be used to represent two decimal integer values: 0 or 1, depending on its voltage status.
- Two memory components can represent **two bits**. If we consider them ordered as in the binary notation, we can represent up to **four integer values** as in the table. That is, with **2 bits** we can represent **2^2** different values. This can be generalized, **n bits** represent **2^n** values.

Binary	Decimal
00	0
01	1
10	2
11	3

- The **value** is called ***magnitude*** of a sequence of bits.
- For historical reasons, an **ordered group of 8 bit** is used as the fundamental unit of measure of computer memory. This is called a **byte**.
  - How many different integer values can a **byte (8 bit)** represent?
    - The range is 00000000 – 11111111, We can represent numbers from 0 to 255 (256 numbers in total)
    - In other words, $2^8 = 256$

Information as a binary mapping: Memory, Bits and Bytes

- If I want to represent at least 1000 values,
  I need an integer **i** such that $2^i \sim 1000$.
  For example for i=10, $2^{10}=1024$ values,
  that is, **10 bits** can represent **1024 values**.
- In modern computer architectures, the 32bit and 64bit buzzword that you frequently hear refers to the size of the **CPU registers**, that is, where the processor copies information from the memory to be processed. I will present it later.
  - A 32bit machine can contain in its registers up to 2^{32} different values.
    - Note: $2^8 * 2^8 * 2^8 * 2^8 = 2^{4*8} = 2^{32}$: A CPU register is made out of **4 bytes**!
  - A 64bit machine can contain in its registers up to 2^{64} different values.
    - In a 64 bit machine a register is made out of **8** bytes.

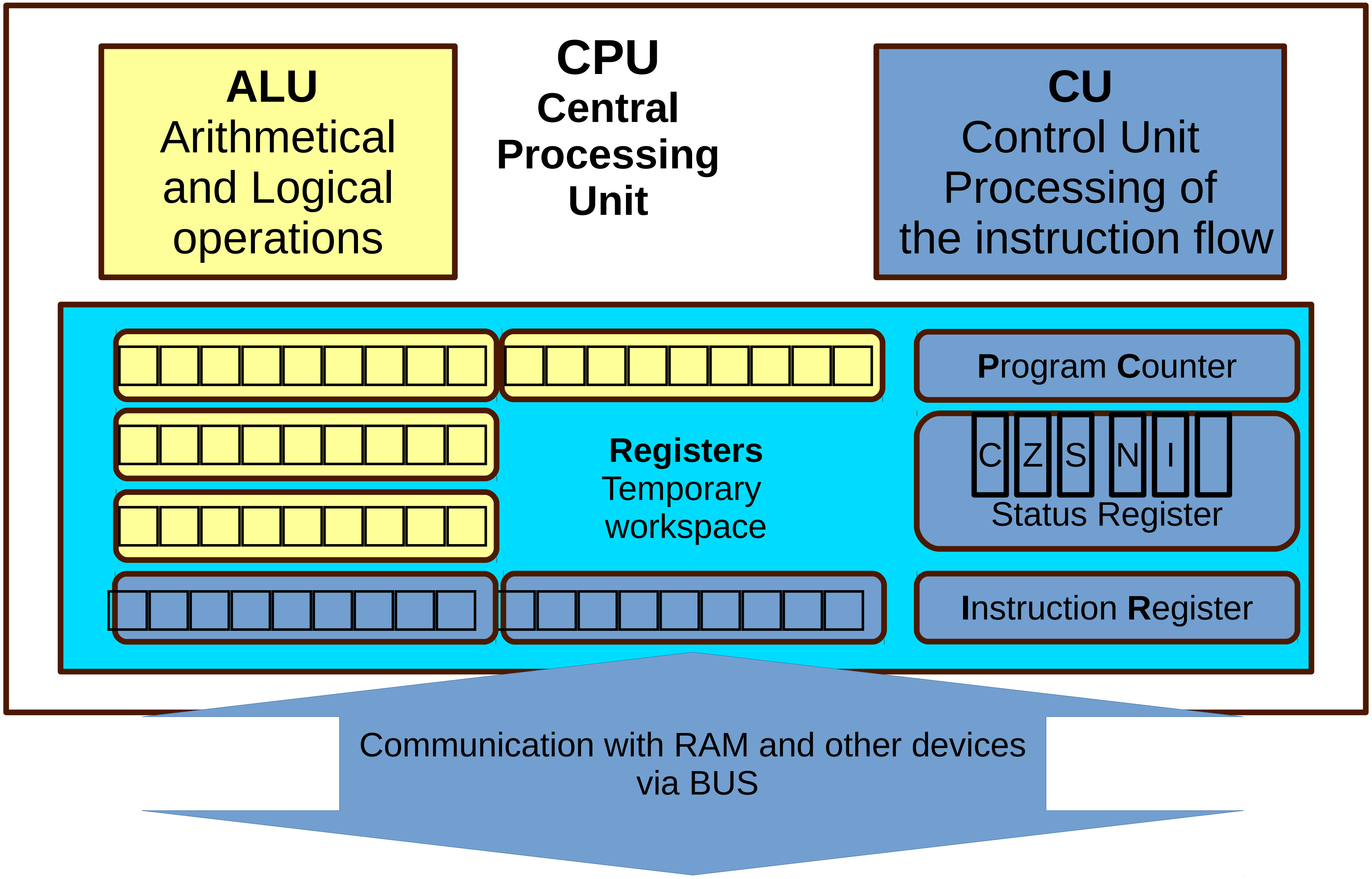
Digital circuits are discrete (countable)

- **Digitalization** is the process of transforming what is continuous into something **discrete** with electronic devices.
- A dreadful consequence of having a finite set of countable memory components representing information is that **there is a finite set of numbers we can represent.**

Problems:

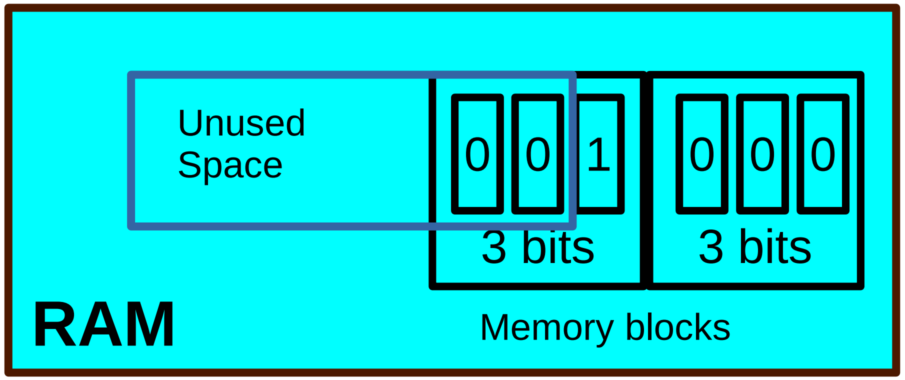
- What happens when the result of an operation **exceeds the finite representation** space?
- How do one represents **negative** numbers?
- How do we represent **fractions/irrational** numbers/**periodic** numbers/**complex** numbers?
- How do we represent the concept of **infinity**?

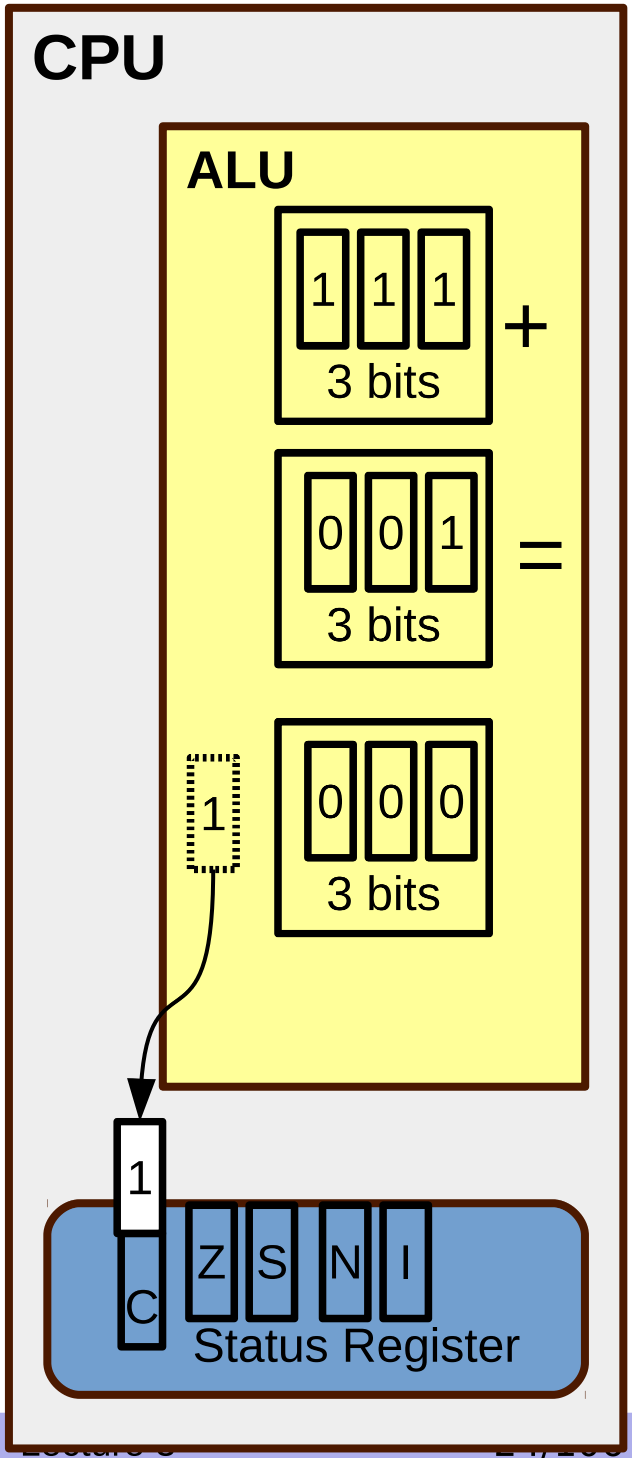
CPU components


CPU
Central Processing Unit
ALU
Arithmetical and Logical operations
CU
Control Unit
Processing of the instruction flow
Program **C**ounter
C Z S N I
Status Register
Registers
Temporary workspace
Instruction **R**egister
Communication with RAM and other devices via BUS

Limitations of finite representation: addition

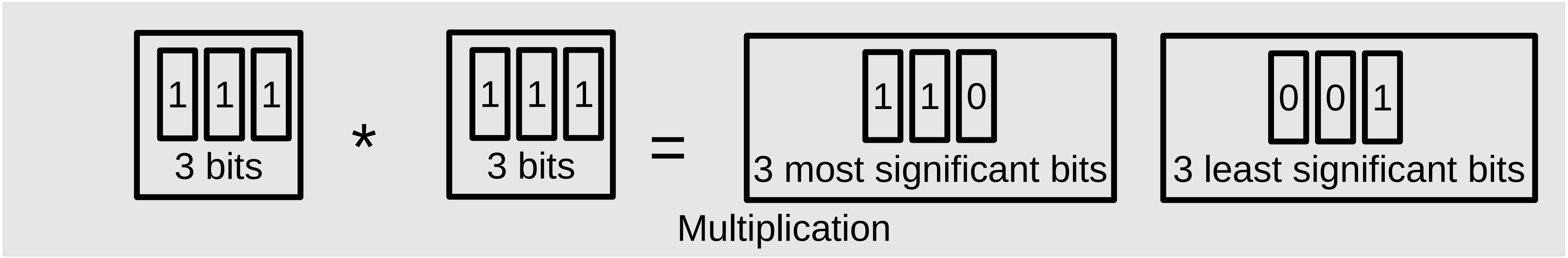
- **Carry overflow** and **register reset**: imagine we have only 3 bit registers (numbers from 000 to 111):
  - 111 + 001 = 1000 = 1 carry and 000 but:
    our registers can only contain 000.
    - Need to keep info about carry somewhere (usually special **carry bit** in the arithmetic circuitry).
    - Need to store the result in **two** memory blocks.



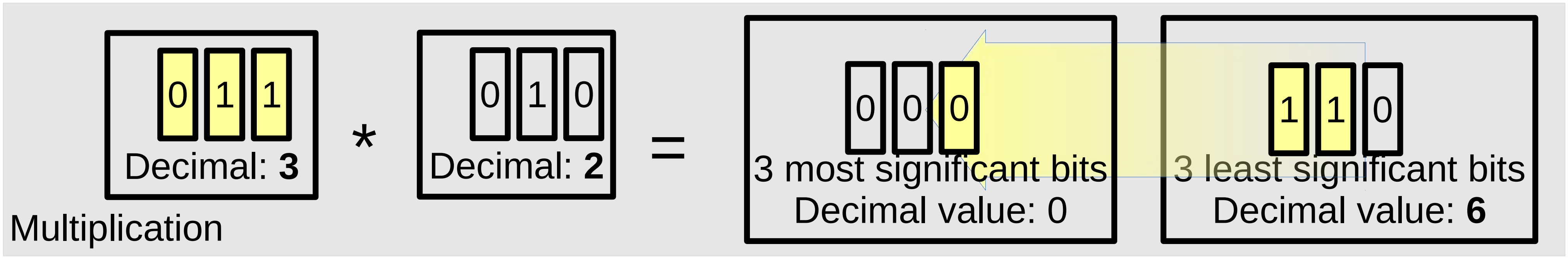


Limitations of finite representation

- Multiplication requires double the size:
  - 111 * 111 = 110001 : it's 6 bits!
    - Need to manage multiplications in a special way.



- Property: multiplication/division by 2 is a *shift left* (multiplication) or *shift right* (division with no remainder)



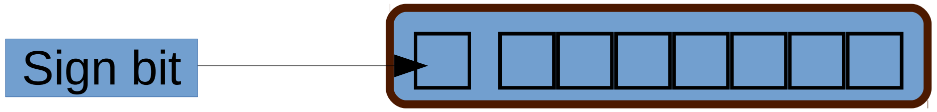
Limitations of finite representation

- Circuits for computation are different from our way of counting:
combinations of adders and shifters to achieve all operations
- **In general, one must be very careful when doing calculations at the edge of the possible representations:**

  Digitalization of continuous data leads to loss of information if at the edge of the representable values.

Integers

- 2- complement representation



8 bit registers example

+127	0 1 1 1 1 1 1 1
0	0 0 0 0 0 0 0 0
-128	1 0 0 0 0 0 0 0

- **Complement** is the operation of flipping a binary value (a sequence of **NOT**): complement_of(1) = 0   complement_of(0) = 1
- The most significant bit (msb) is the **sign bit**, with value of 0 representing positive integers and 1 representing negative integers.
- The remaining n-1 bits represent the **magnitude** of the integer, as follows:
  - for positive integers, the absolute value of the integer is equal to "the magnitude of the (n-1)-bit binary pattern".
- for negative integers, the absolute value of the integer is equal to "the **magnitude** of the complement of the (n-1)-bit binary pattern plus one" (hence called 2's complement).
  - Example: **-**128 = 10 00 00 00 = **-** [magnitude_of (complement_of(0 00 00 00)) + 1] = **-** [magnitude_of (1 11 11 11) + 1] = **-** [127 + 1] = **-**128

From: https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Integers

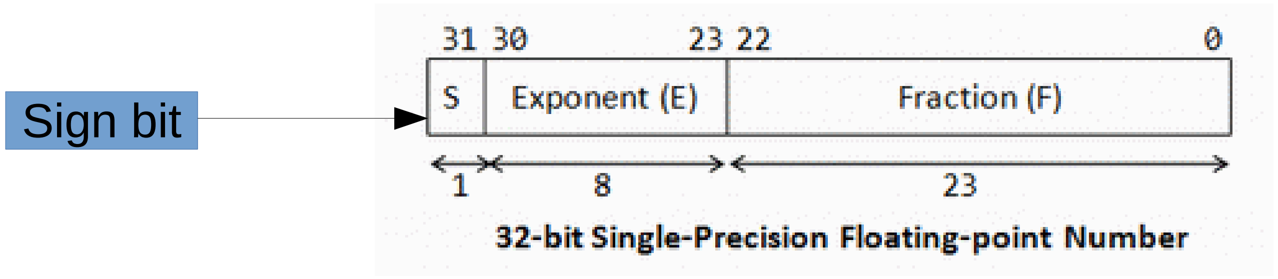
- 2- complement representation limits

Size of registers	minimum	maximum
8	-(2^7) (=-128)	+(2^7)-1 (=+127)
16	-(2^15) (=-32 768)	+(2^15)-1 (=+32 767)
32	-(2^31) (=-2,147,483,648)	+(2^31)-1 (=+2,147,483,647) (9+ digits)
64	-(2^63) (=-9,223,372,036,854,775,808)	+(2^63)-1 (=+9,223,372,036,854,775,807) (18+ digits)

From: https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Real numbers

- IEEE-754 32-bit Single-Precision Floating-Point Numbers

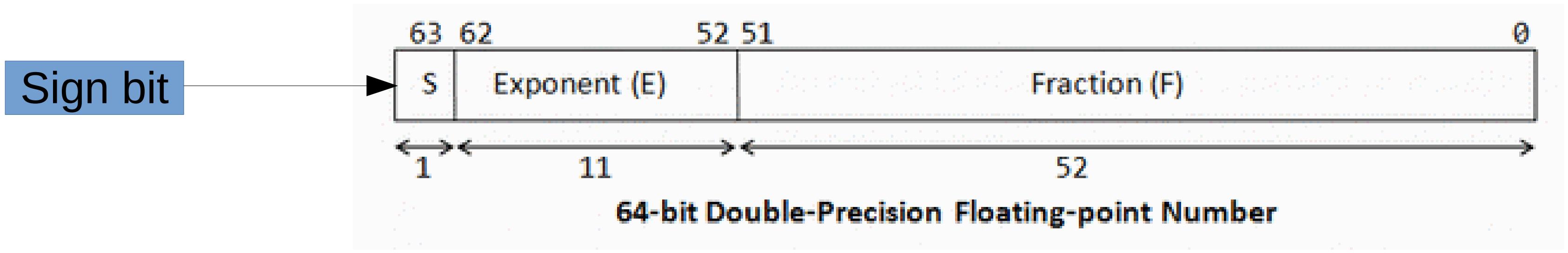


- For $\mathbf{1 \le E \le 254}$, $N = (-1)^S \times 1.F \times 2^{(E-127)}$. These numbers are in the so-called **normalized** form.
  - The sign-bit represents the sign of the number.
  - Fractional part (1.F) are normalized with an implicit **leading 1**.
  - The exponent is bias (or in excess) of 127, so as to represent both positive and negative exponent. The range of exponent is -126 to +127.
- For $\mathbf{E = 0}$, $N = (-1)^S \times 0.F \times 2^{(-126)}$. These numbers are in the so-called **denormalized** form.
  - The exponent of $2^{(-126)}$ evaluates to a very small number.
  - Denormalized form **is needed to represent zero** (with F=0 and E=0). It can also represents very small positive and negative number close to zero.
- For $\mathbf{E = 255}$, it represents special values, such as **±INF** (positive and negative infinity) and **NaN** (not a number).

From: https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Real numbers

- IEEE-754 64-bit Double-Precision Floating-Point Numbers



- Normalized form:
  For **1 ≤ E ≤ 2046**, N = $(-1)^S \times 1.F \times 2^{(E-1023)}$.
- Denormalized form:
  For **E = 0**, N = $(-1)^S \times 0.F \times 2^{(-1022)}$. These are in the denormalized form.
- For **E = 2047**, N represents special values, such as **±INF** (infinity), **NaN** (not a number).

From: https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Other notations

- Octal: 0 1 2 3 4 5 6 7
  - Decimal 8 = Octal 10
- Hexadecimal:
  - Useful for memory maps (hex-editors)

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

  - Decimal 16 = Hexadecimal 10

Memory map

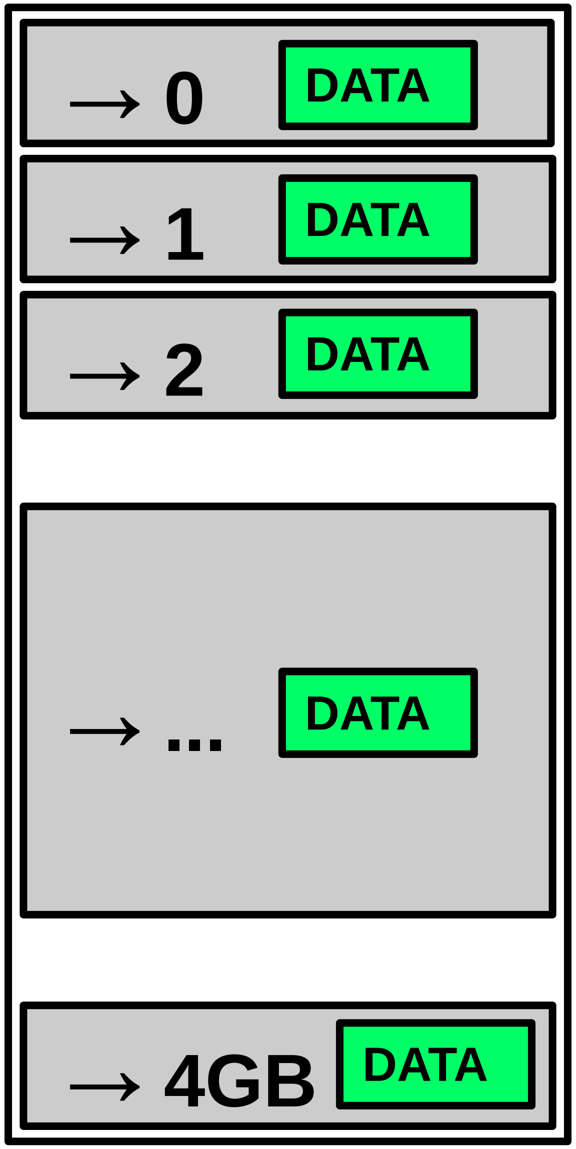
Memory map	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0								7 hex 7 dec								0F hex 15 dec
1																1F hex 31 dec
2								27 hex 39 dec								
3																
4																
5																

Accessing Memory

- Memory size
- Addressing memory: pointers
- Stack
- Heap
- Relative relocation

Addressing memory (RAM)

- Computer memory is divided in a certain number of **locations**.
- A physical memory element at a specific location is like a register, and has a size in bit. Usually is 8 bits, a byte.
- A location is a memory space identified by a **memory address**
- A memory address is a in integer **number**.
- This number is usually called **pointer** (→), as it points to a memory location.



Addressing memory and size: bits and bytes

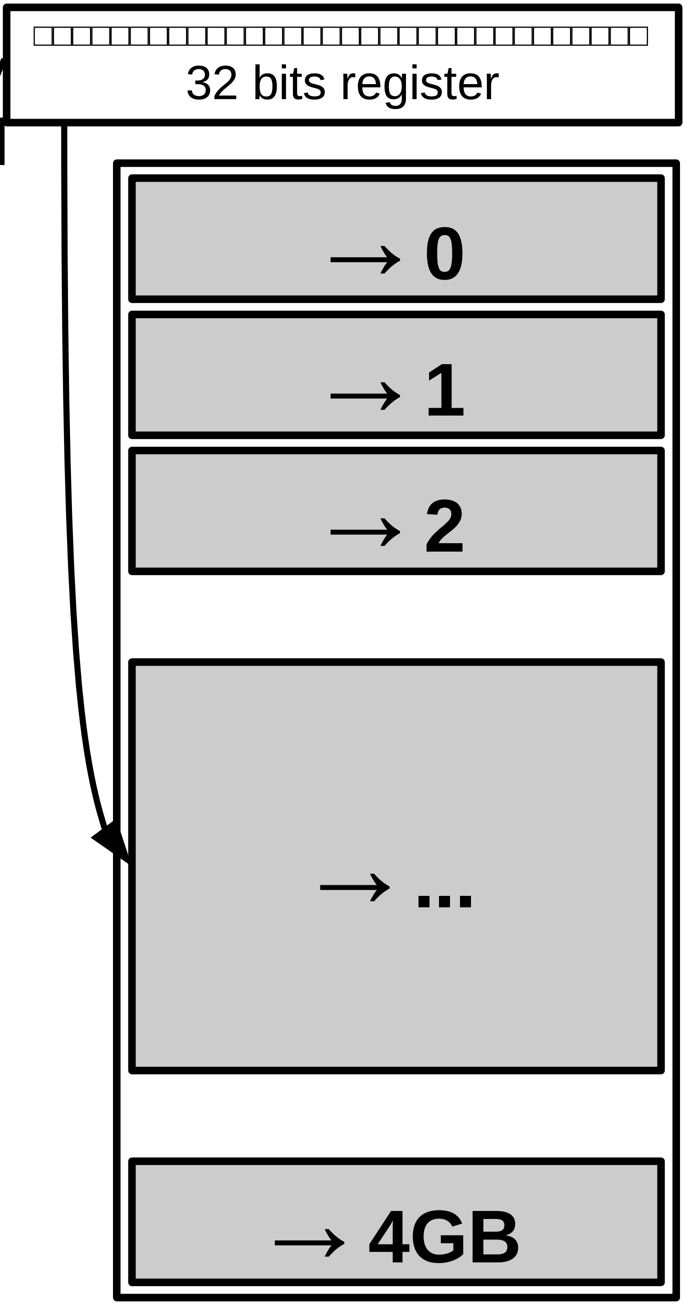
- The size of a RAM memory bank tells how many memory locations can be **pointed** or **referenced** within that bank of memory.
- This size is measured in **bytes**.
  - Remember: 1 byte is made out of 8 bits
- 1024 bytes are called a **Kilobyte**. Often noted as Kb or kb or KB (unfortunately producers never agreed on the notation). We will use **KB**.

Memory size

- Conversion to the different orders is done by dividing/multiplying for **1024** in decimal notation. Examples:
  - 1 KiloByte = 1 KB = 1024 Bytes
  - 1 MegaByte = 1 MB =
    1024 KB =
    1 048 576 Bytes
  - 1 GigaByte = 1 GB =
    1024 MB =
    1 048 576 KB =
    1 073 741 824 Bytes = about 1 Million bytes.
- A 4 GB memory bank contains
  4*1 GB =
  4*1024 MB = 4096 MB =
  4*1048576 KB = 4194304 KB =
  4*1073741824 Bytes = 4 294 967 296 Bytes

Addressing memory: pointers

- If one wants to address each and every byte in a memory of 4GB, she will need at least 32bits register:
  - **4GB** = 4 millions memory locations = 4096MB = 4 294 967 296 = $\mathbf{2^{32}}$
  - The number contained in the register is usually called a **pointer,** as it **points** to a memory location
- However, things are not that easy. Not all the represented numbers can be used for referencing memory, see: http://en.wikipedia.org/wiki/3_GB_barrier
- We can anyway assume that the accessible memory space depends on the computer architecture, i.e. a 64bit machine can access $\mathbf{2^{64}}$ memory locations.



Addressing memory: the compromise

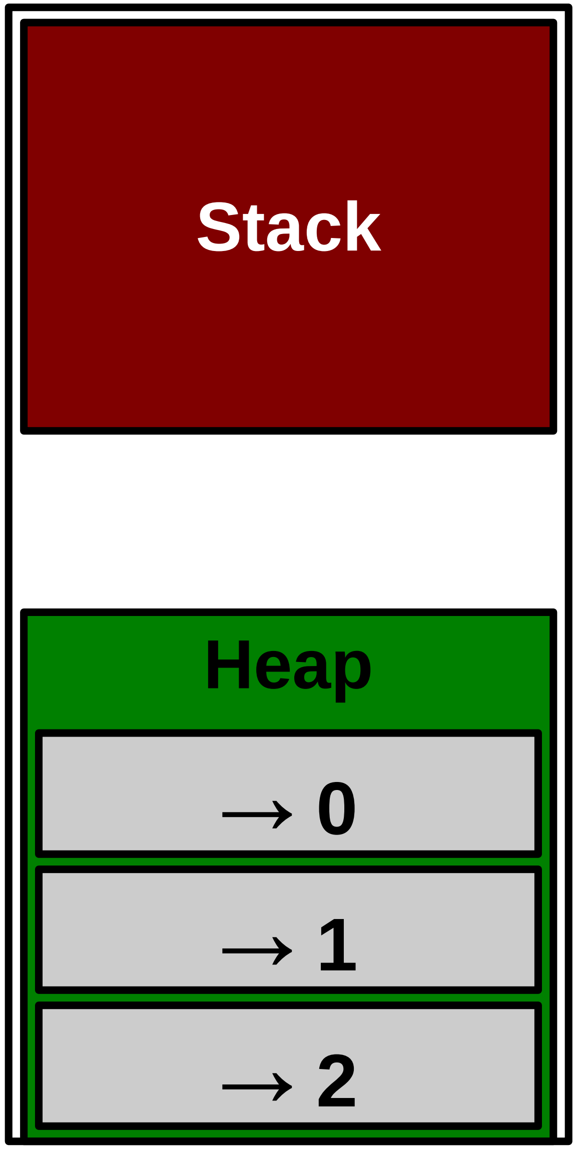
- Observe the following:
  - If I have a big memory, I want a big pointer (64 bit)
  - I also want to store memory pointers in memory
  - Each pointer uses 64bit
  - Negative consequences:
    - The same application compiled for using 32bit and 64bit memory will be **bigger,** or have **higher memory requirements**, when using 64bit pointer.
    - **Modern 64bit computers just need double the memory of the old 32bit :(**
- What is the only benefit?
  - **Bigger memory space**
    - We can actually memorize double the things, provided that we are careful in specifying that we can pack them in a 64 bit space (compilers can do this, but at a cost)
  - **Precision:**
    - We can represent more integer numbers
    - non integer numbers can be more close to the theoretical representation (reducing the approximation error)

Stack and Heap

Modern programming saves you from specifying the exact pointer location. The memory is represented as a **logical memory** available to a programmer.
It is modelled like partitioned in two sets:

- Stack: Managed by a tool called **compiler**.
  - Memory is allocated and deallocated (freed) automatically by the **compiler**.
  - It usually only survives for a short term.
  - Function recursion uses that heavily.
- Heap: Managed by developer directly using system libraries functions.
  - developer allocates and deallocates memory by writing explicit programming language statements.
  - **It can survive a whole program if the developer forgets to deallocate it!!**

The use of these will be clearer during the tutorials.



Questions?

From binary to programming languages

Binary as machine language

- A machine only has the binary alphabet to describe things. All that moves between the CPU and the Memory is chunks of memory of the maximum size as the number of bits given by the architecture (i.e. 64 bits)
- These memory chunks can be either data or **instructions,** that is, *words* of the machine language.
- When an instruction is copied from RAM to a special register inside the **CPU**, the **Instruction Register**, this will be **executed**, the operation that it represents will be carried on.

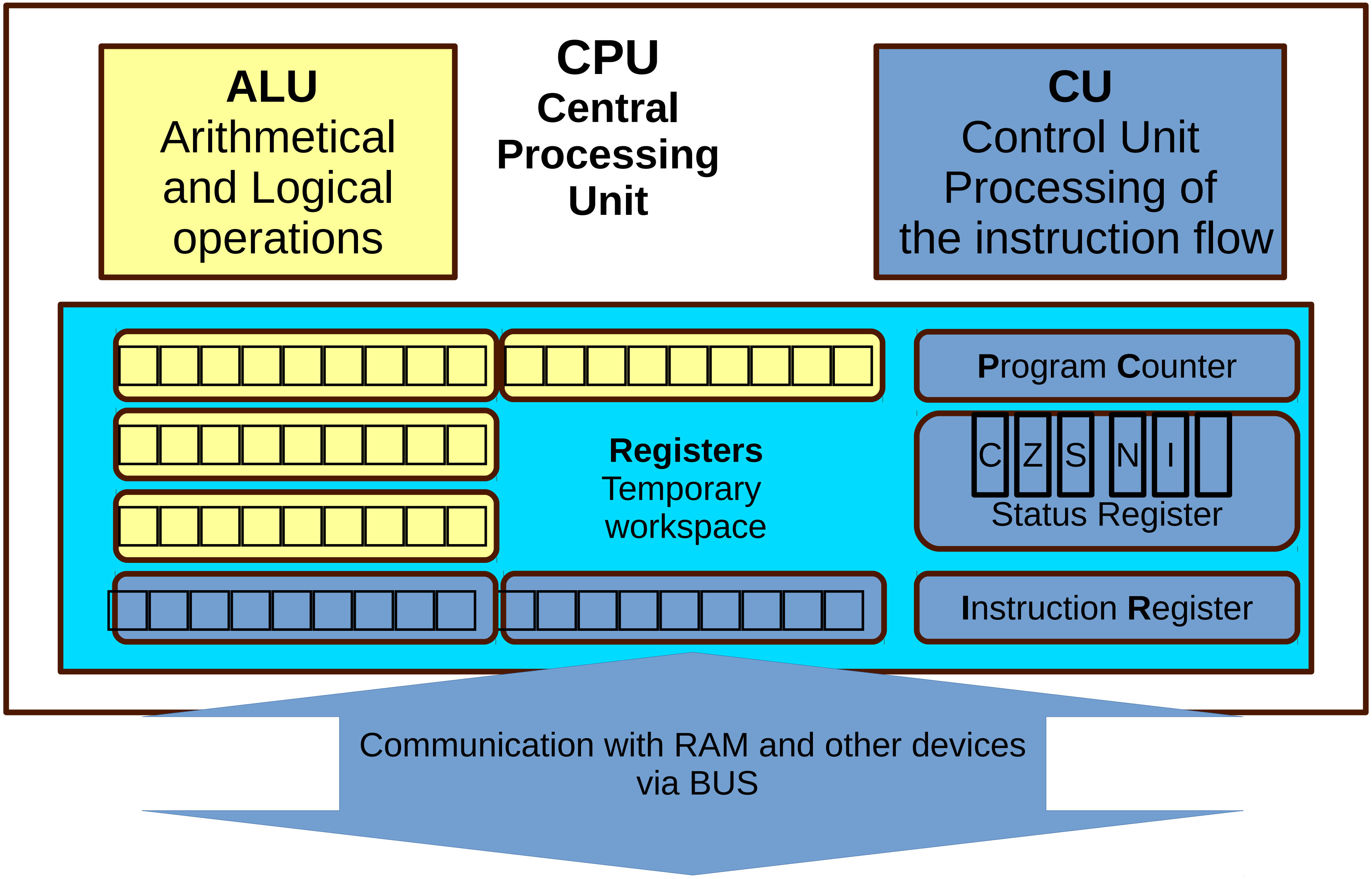
Machine Language: Binary Code

- A computer instruction is a **sequence of bits**, that is, zeroes and ones.
- A binary instruction is also called **opcode**, **Op**eration **Code**
- For simplicity, each instruction corresponds to a human-readable string, called **Assembly Instruction**
- The following table shows shows examples of instructions, where the letters identified by dollars denote an operand.
- Operands **are not values**, but identify **one Processor Register**. Processor registers are small memory inside the CPU itself that the CPU uses to work; each has a number that identifies it.
  A register contains the actual values that the operation will use.

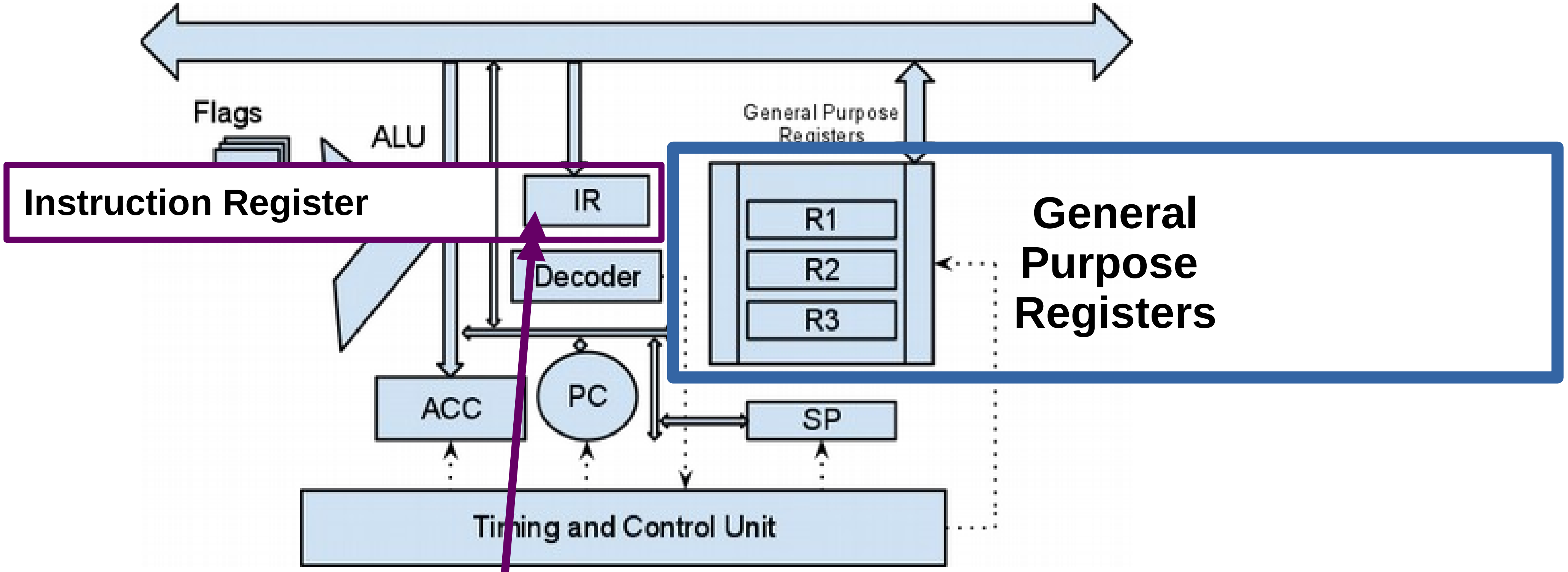
Instruction	Opcode/Function	Syntax	Operation
add	100000	ArithLog	$d = $s + $t
addu	100001	ArithLog	$d = $s + $t
and	100100	ArithLog	$d = $s & $t

$d	ID of destination register
$s	ID of source register
$t	ID of second source register

CPU components



Machine Language: Binary Code



Instruction	Opcode/ Function	Syntax	Operation
add	100000	ArithLog	$d = $s + $t
addu	100001	ArithLog	$d = $s + $t
and	100100	ArithLog	$d = $s & $t

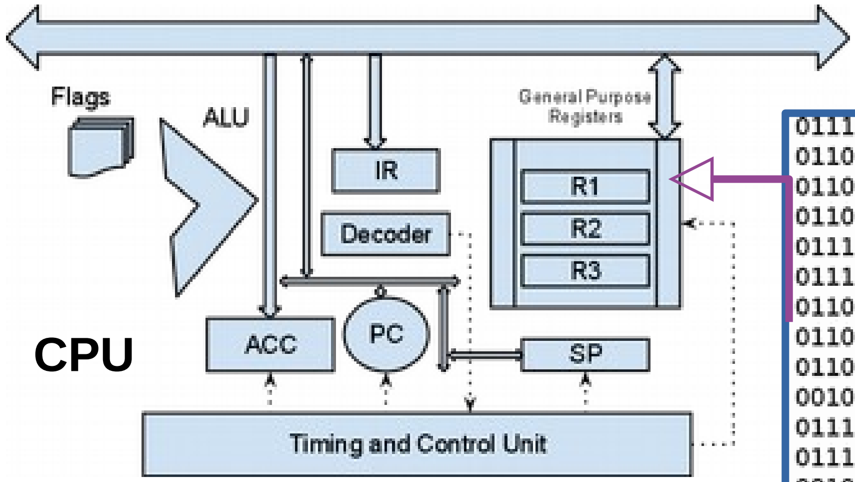
$d	ID of destination register
$s	ID of source register
$t	ID of second source register

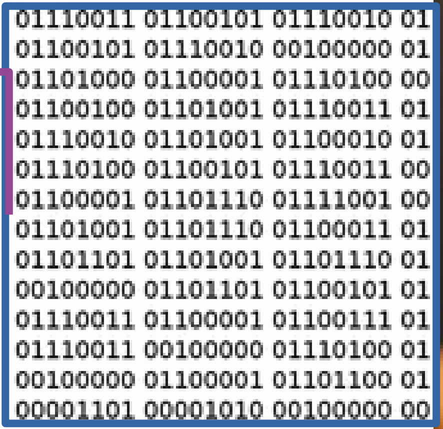
Programming languages: A brief history

Modern classification of programming languages is based on generations. As generation increases, the languages are closer to the human way of expressing concepts.

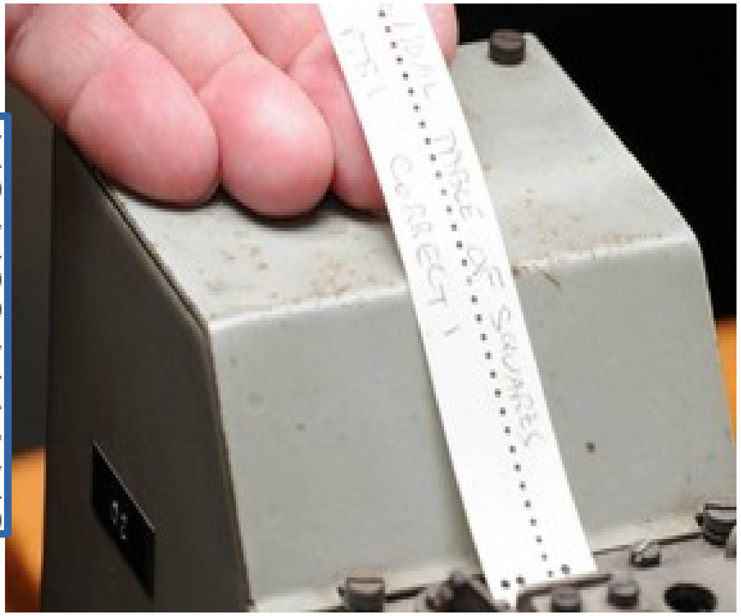
- 1st generation. **Machine code** language. This includes punchboards and **binary code**. Machine dependent.
- 2nd generation. **Assembly** or instruction-based languages. Still used in embedded programming, but through 3rd generation ones. Machine dependent. Hard to use for complex things.
- 3rd generation. Also called **High-Level** programming languages. Mostly use **English** to describe commands. **Machine independent. General Purpose: you can use them for EVERYTHING.**
  These include: C, C++, C#, Java, Javascript, Python, Bash, PHP, Pascal, Fortran...
- 4th generation. **Domain specific** languages. Report or Form generator, or Data manipulation. Examples: Mathematica, Matlab, SPSS, R (statistics). Targeted to a specific set of tasks.
- 5th generation. Mathematical or logical languages. Solving problem by specifying constraints, **without focusing on the algorithm**. Mainly used in artificial intelligence research. Examples: Prolog, NetLogo. Very narrow scope.

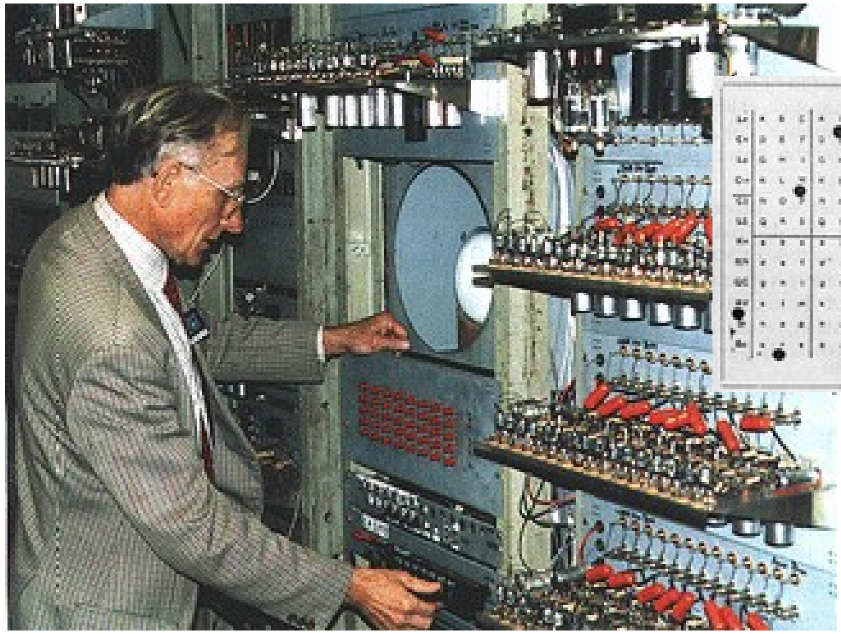
1st generation: Machine Language

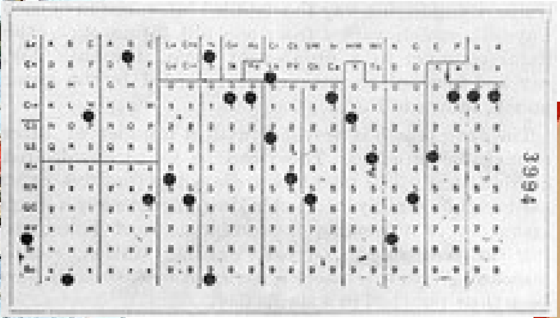


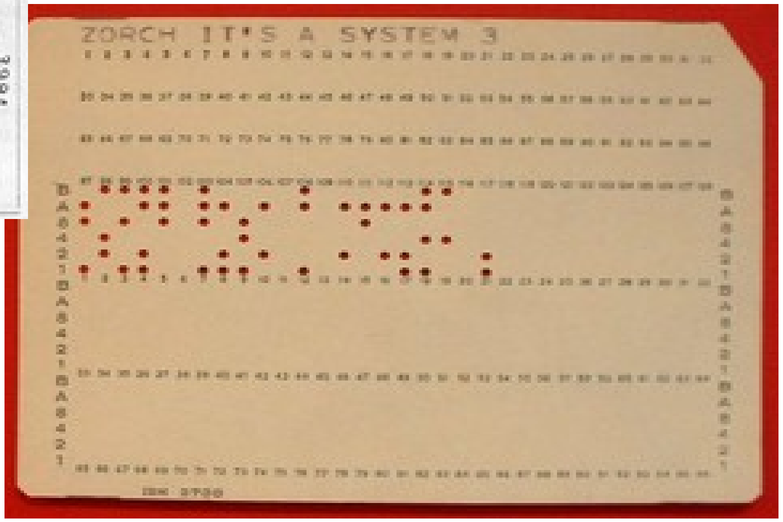


```
01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00
01100001 01101110 01111001 00
01101001 01101110 01100011 01
01101101 01101001 01101110 01
00100000 01101101 01100101 01
01110011 01100001 01100111 01
01110011 00100000 01110100 01
00100000 01100001 01101100 01
00001101 00001010 00100000 00
```





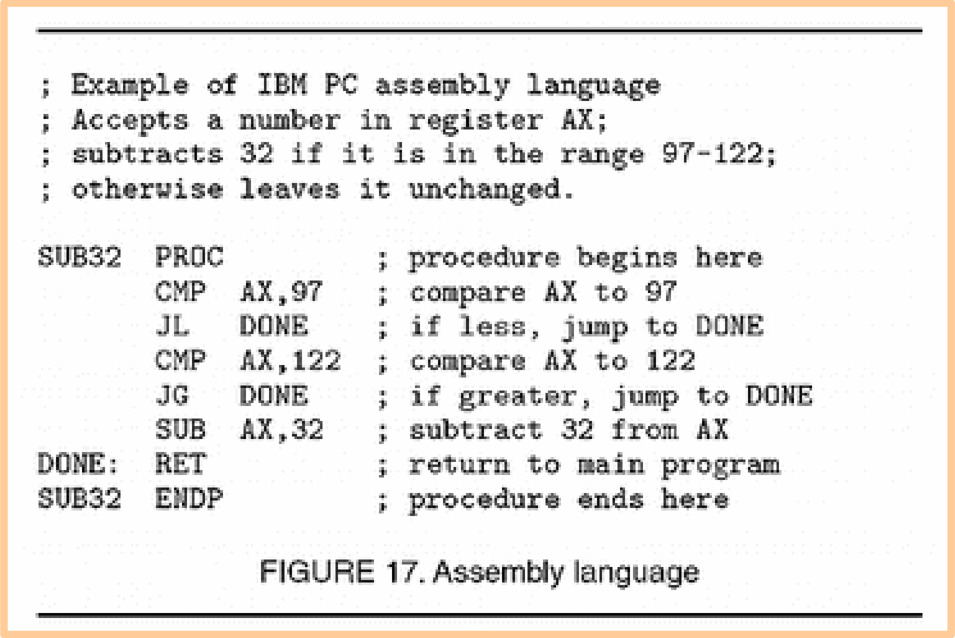




1st generation: Machine Language

- Minimal instructions set in binary code: binary sequences corresponding to operations like **move**, **read**, **sum**, **multiply**...
- Direct edit of CPU Registers, Memory Pointers, Start of Program Counter
- Direct programming, not portable, specific for a determined machine.

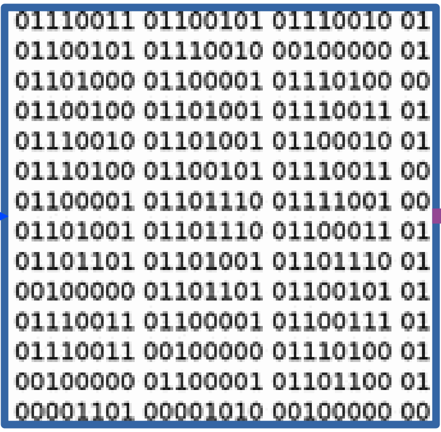
2nd generation: Assembly Code



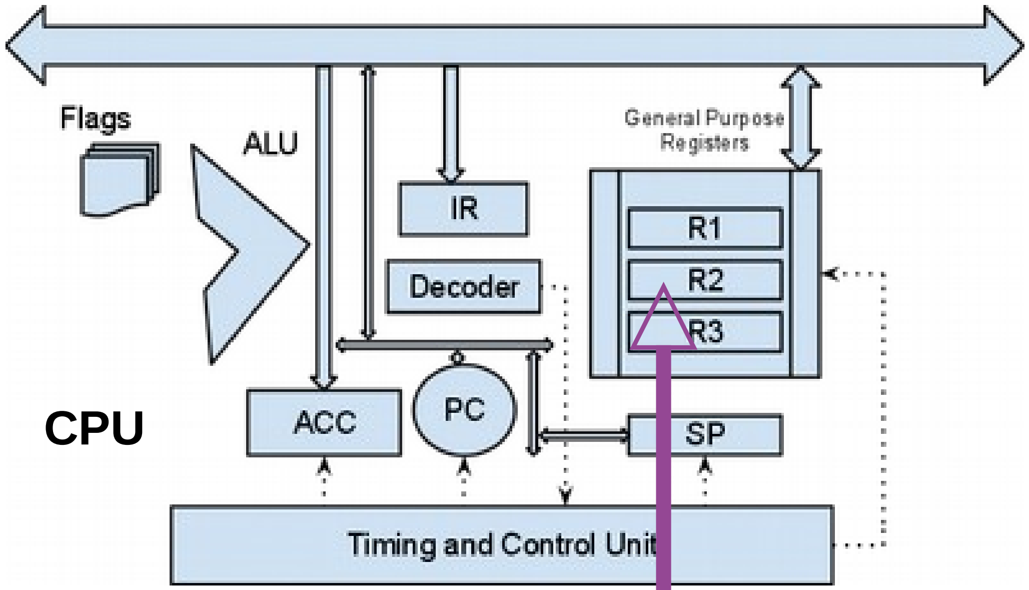
```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32   PROC            ; procedure begins here
        CMP   AX,97     ; compare AX to 97
        JL    DONE      ; if less, jump to DONE
        CMP   AX,122    ; compare AX to 122
        JG    DONE      ; if greater, jump to DONE
        SUB   AX,32     ; subtract 32 from AX
DONE:   RET             ; return to main program
SUB32   ENDP            ; procedure ends here
```

FIGURE 17. Assembly language

Assembler

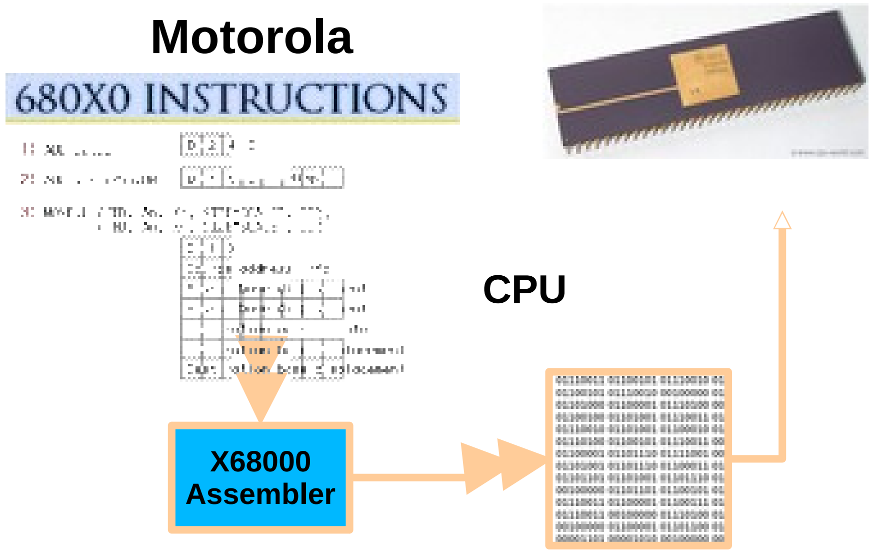


```
01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00
01100001 01101110 01111001 00
01101001 01101110 01100011 01
01101101 01101001 01101110 01
00100000 01101101 01100101 01
01110011 01100001 01100111 01
01110011 00100000 01110100 01
00100000 01100001 01101100 01
00001101 00001010 00100000 00
```

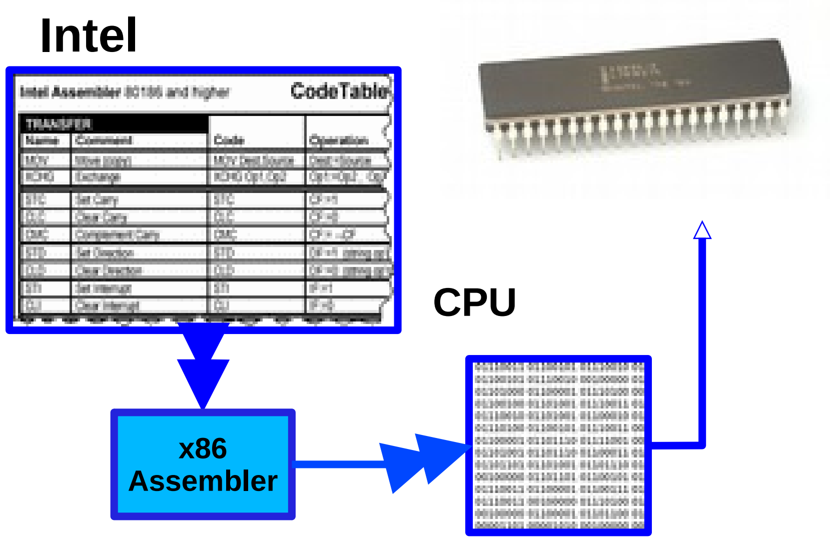


2nd generation: Assembly Code and Microcode

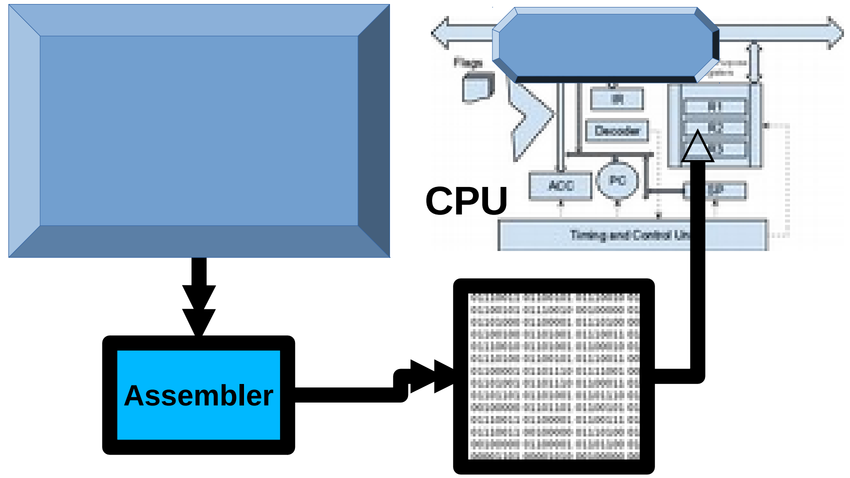
Motorola



Intel



Other Architecure



Not Portable!

2nd generation: Assembly Code and Microcode

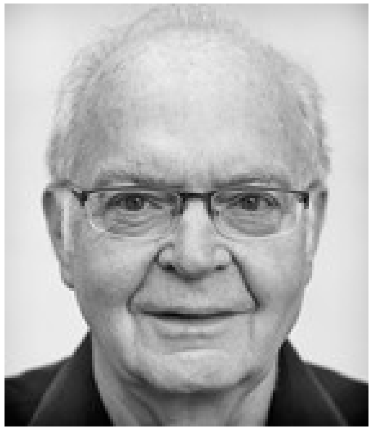
- Each instruction is represented by an **opcode** and its **arguments**.
- A more human readable language is introduced, **assembly**, that maps each opcode and arguments to a human readable syntax.
  - The program used to code is called **assembler**, takes in input a sequence of assembly statements and translates them into binary code
- New CPUs emerge that contain a more complex instruction set called **microcode**, stored physically in a ROM inside the CPU: a single instruction can do more than a single operation. Different assembly for different architectures.
  - **Not portable**: code can only be used for a specific machine.
- Used for home computers, nowadays for small devices.

Live example: https://schweigi.github.io/assembler-simulator/

3rd generation: Human-oriented

- **Algorithm oriented**: the user translates an algorithm into language commands
- Introduces programming *paradigms*:
  - **Imperative**
  - **Object Oriented**
  - Functional
  - ... more!
- Introduces various **translation to machine language** methods:
  - Compiled
  - Interpreted
  - Bytecode interpreted

Grace Hopper
1959 invents COBOL


Donald Knuth
1970 Writes
“The Art of Computer Programming”

Imperative languages

- Programming style that describes computation in terms of a **program state** and **statements** that **change** the program state.
- Adheres to the *separation of code and data* principle.
- Examples: C, FORTRAN, Python, Bash
- Remember `printf ("%s \n", "Hello World!");` ?

Code

Data

Separation of Code and Data principle

- Code is information about **logic**, **arithmetics** and **algorithms**.
  - One can think of it like a mathematical function, that defines a domain and co-domain in generic terms.
- Data is information that is **to be read, processed, written**.
  - **Input** data **should be left untouched and not modified**. Think about is as a science fact or empirical/experimental data.
    - One does modify it in memory while running a program, but the changes should never be written back to the original data (would pollute science facts!)
  - **Output** Data is usually **the result** of something code did on it. For ease of use, it might be represented the same way as Input Data.

Ingredients of programming: Data

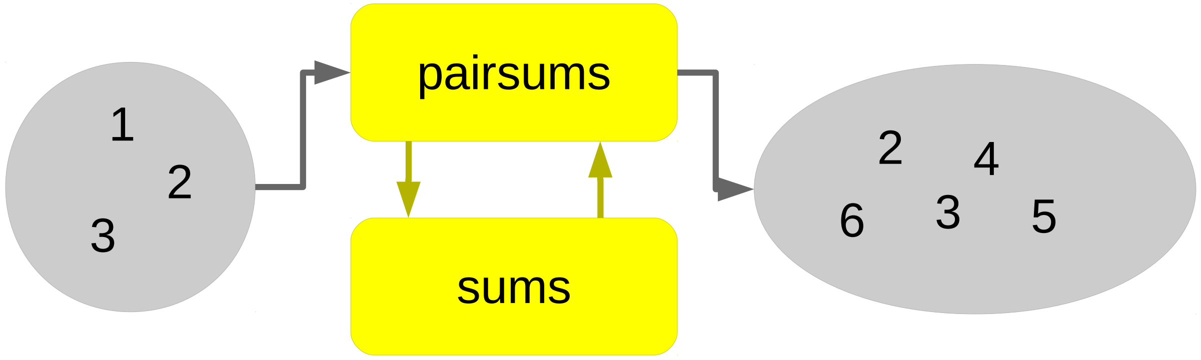
- Often provided by the user
- NOT code, but *used* by code to do things
- Carries **information**, most likely understandable by a scientist.
- **Input data**: provided in input **to** the code to process information.
  - Example: the formatting information `"%s \n"`, and the text string ”`Hello World!`”
- **Output data**: the result of the code execution, that will be generated as output **from** the code execution.
  - Example: the output string `Hello World!`

Separation of Code and Data Mathematical example

- Goal: Given a set of positive integer numbers, give all the possible sums of each pair of such numbers (including the a number and self, i.e. (a + a)).
- Input data:
  - The set of numbers I={1,2,3}.
- algorithm using math syntax and natural language:

1. $Define: sums(x,y) = x + y; x,y \in \mathbb{N}$
2. $Define: pairsums(I) = n \in \mathbb{N}\, such\ that\ sums(i,j) = n, for\ all\ i,j \in I$
3. $Compute: pairsums(\{1,2,3\})$

- Output data:
  - O={2,3,4,5,6}

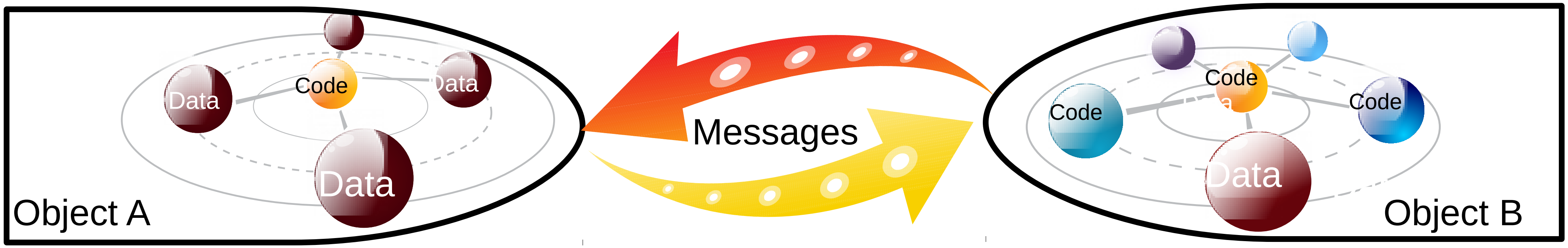


Object-oriented languages

- A computer program is a **collection of objects** that act on each other.
- Each object is capable of **sending and receiving messages** and **processing data**. Each object is independent.
- An object is a ‘black box’ which sends and receives messages, and consists of **code** (computer instructions) and **data** (information which these instructions operate on).

⚠ Breaks the *separation of code and data* principle.

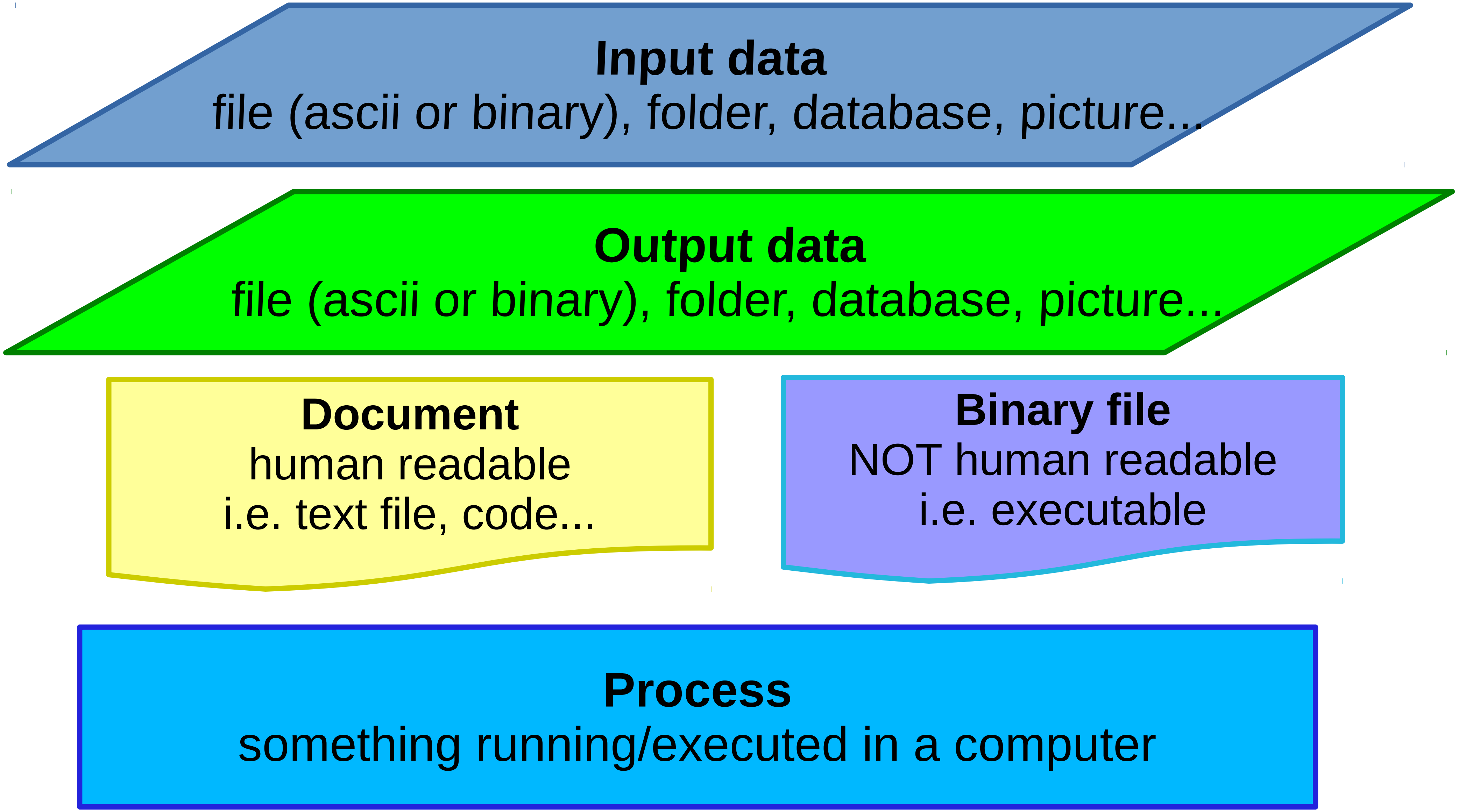
- Examples: Java, C++, Python



Object-Oriented Languages Concepts

- A **Class** is a piece of code that defines the object's:
  - **properties** (also called **attributes**) usually data that the object can handle or carries
  - **Methods** (Usually **functions** or **procedures**) that is usually code used to *modify* the properties of the object or other objects.
- An **instance** of a class is an object, that is, something that a computer can run.
- Classes can **inherit** properties and methods from each other. If *class A is inherits from class B*, B is said to be the **parent** class of the ***child*** class A.
- Classes can **override** properties and methods that belong to their parent class by reusing the same names of properties and methods
- More practical stuff during Katja's tutorials

Flow chart like notation


Input data
file (ascii or binary), folder, database, picture...
Output data
file (ascii or binary), folder, database, picture...
Document
human readable
i.e. text file, code...
Binary file
NOT human readable
i.e. executable
Process
something running/executed in a computer

The information flow

DIGITALIZATION

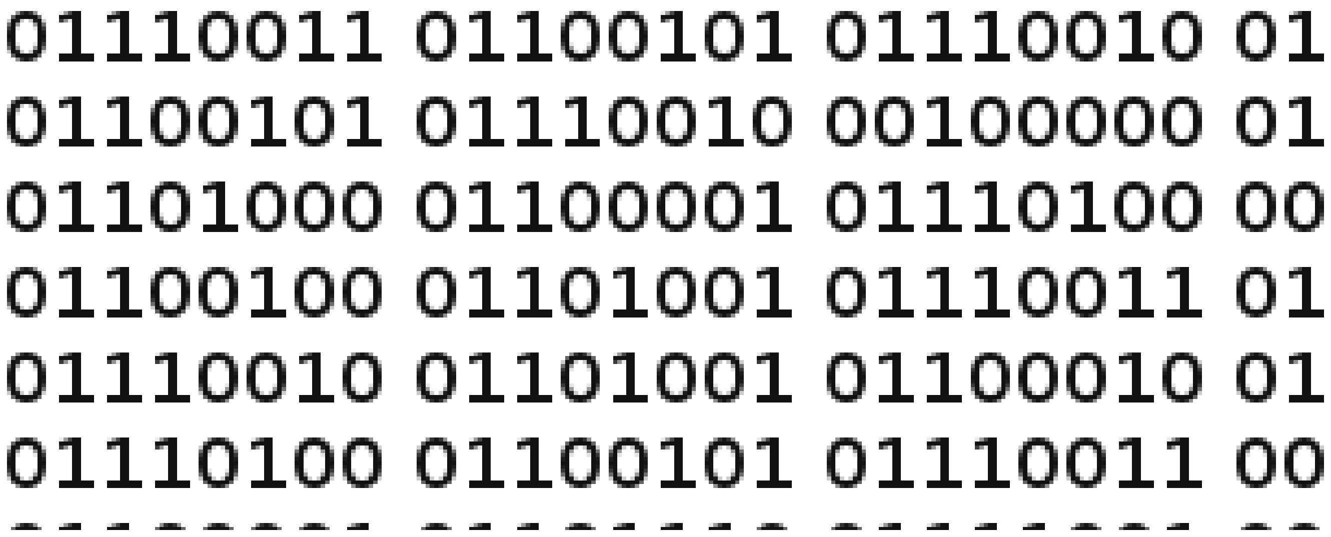
MAY CAUSE LOSS OF INFORMATION!!

From code to machine language

- A **process** is a program that is *executing* in a computer.

Process

- To be executed by a computer, a program must be written in **machine language**.
- Machine language is **binary code**:

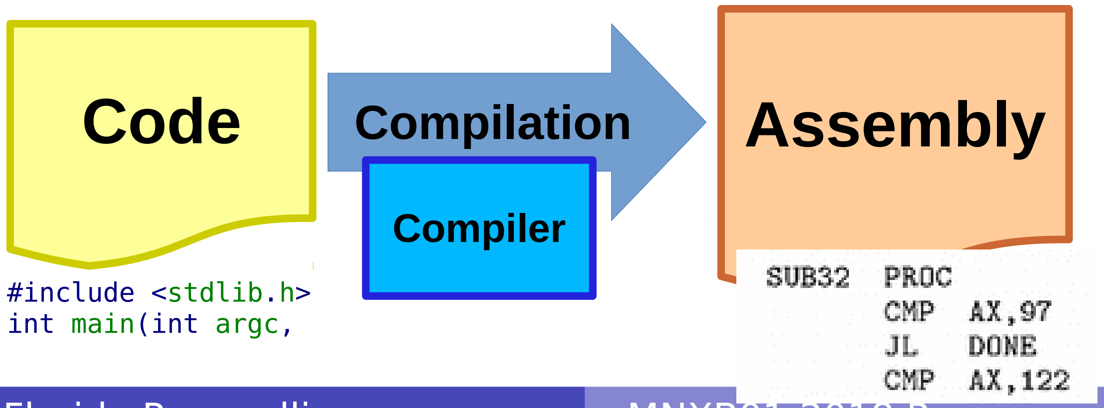


01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00

How does one go from **code** to **machine language**?

From code to machine language

- The *translation* of **code** written in a certain **programming language** is called **compilation**.
- Is performed by a special program called the **compiler.**
- The first step of compilation transforms Code into Assembly Code.

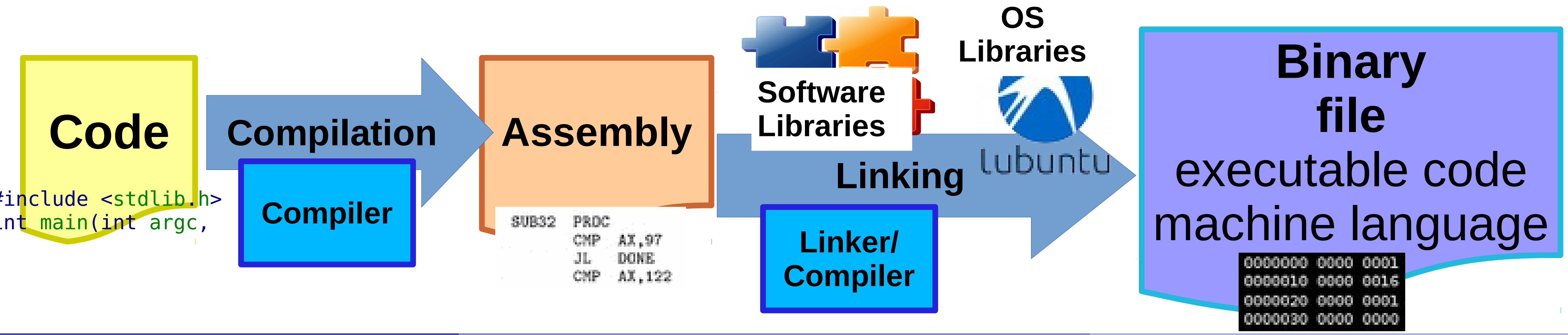


```
#include <stdlib.h>
int main(int argc,
```

```
SUB32   PROC
        CMP   AX,97
        JL    DONE
        CMP   AX,122
```

From code to machine language

- The *translation* of **assembly code** to **executable code** or **machine language** is called **linking**.
- The **Linker**:
  - Binds the software to specific Operating System functions, the **system libraries**
  - Adds **external libraries** to the written code (i.e. scientific libraries for advanced computation)
  - Translates the Assembly code into machine language.
- The result of linking is also called **binary file**



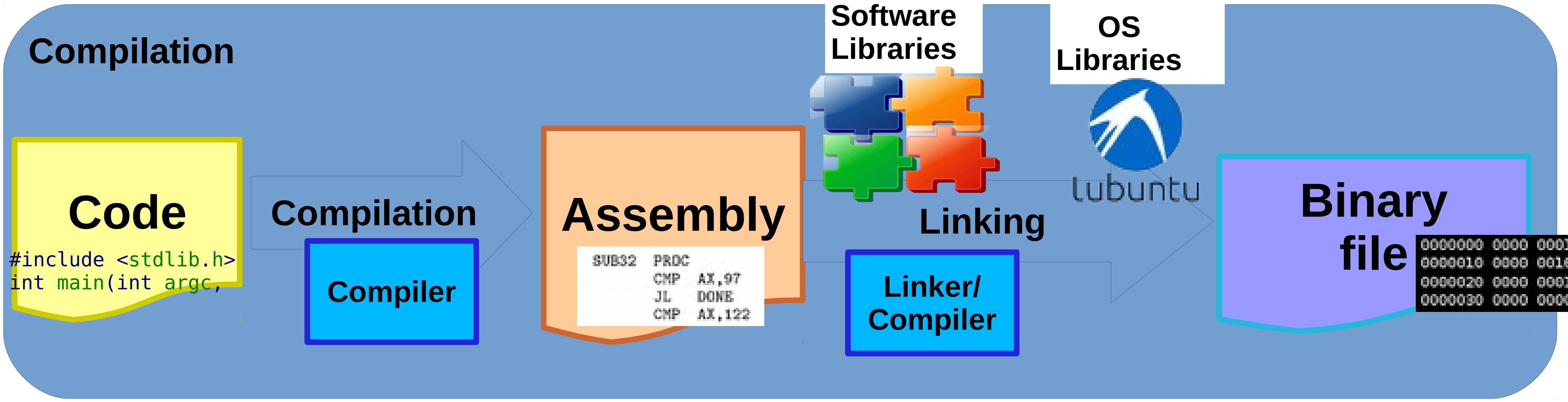
```
include <stdlib.h>
nt main(int argc,
```

```
SUB32  PROC
       CMP   AX,97
       JL    DONE
       CMP   AX,122
```

```
0000000 0000 0001
0000010 0000 0016
0000020 0000 0001
0000030 0000 0000
```

From code to machine language

- The term **compilation** is commonly used for both the process of Compiling and Linking, as it is very hard to decouple them in practice.



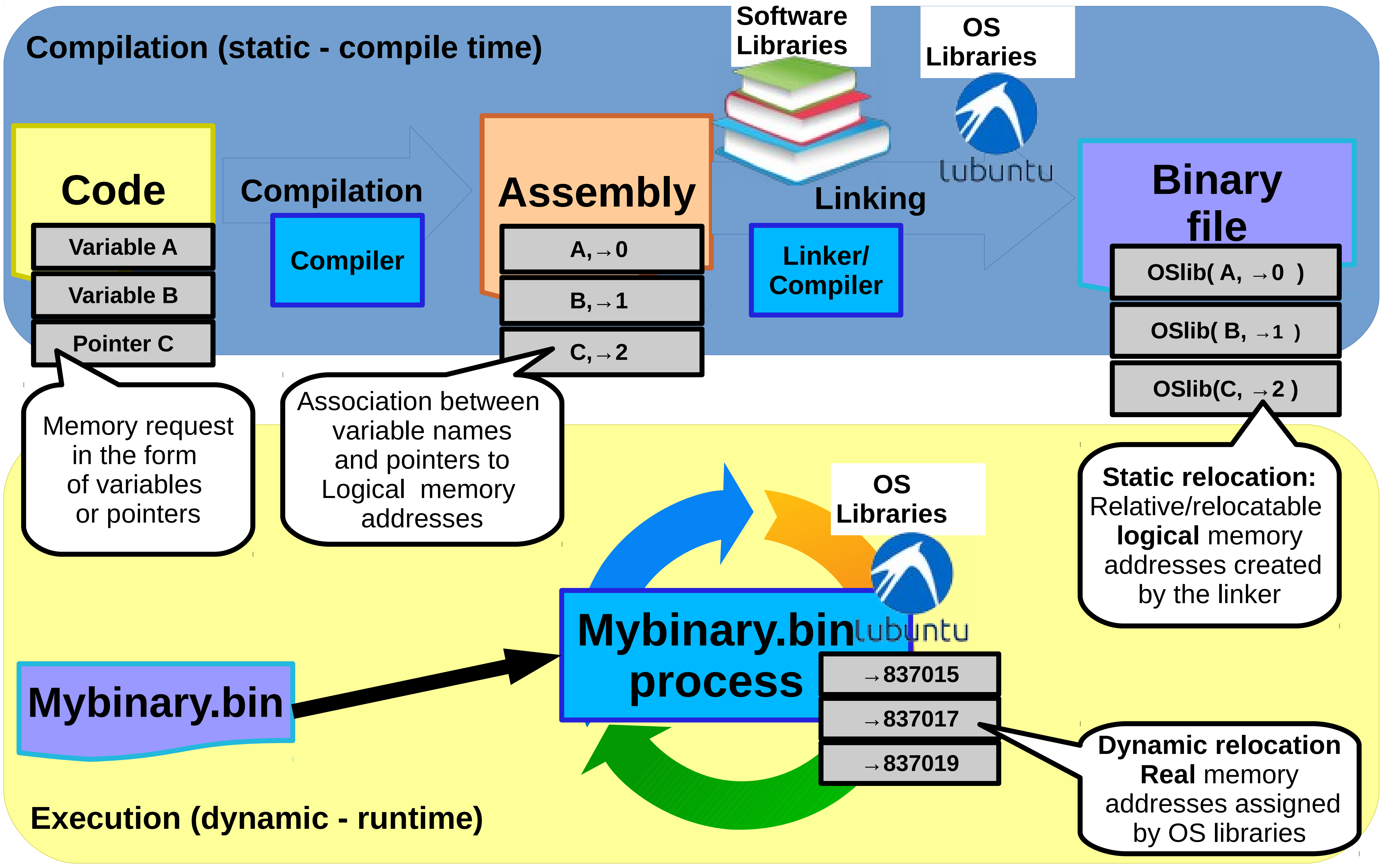
Steps to compilation

- Scientist write their own code, also called **source code**.
- Source code is provided as Input data to the **compiler**.
- The compiler process runs, compiles and links the code and then generates **compiled and linked binary code**.
- The binary code is written to a file as Output data of the compilation process, the result of the compilation process is hence a **binary file**.

Execution

- **Execution** of a binary file is the task of
  1) *Loading* it into the computer memory (RAM)
  2) *Tell* the processor (CPU) to *start processing* the instructions just loaded in memory (commonly said **run**)
- In modern machines this is simplified by
  - touching an app icon (phones)
  - double clicking on an icon (most of graphical interfaces)
  - explicitly writing the name of the program to run using command line interfaces (e.g. BASH).
- When the code is being compiled, all the decisions taken by the compiler are said to be at "**compile time**"
- When a process is running, all the actions taken by the process and the operating system are said to be at "**runtime**"

Memory Relocation

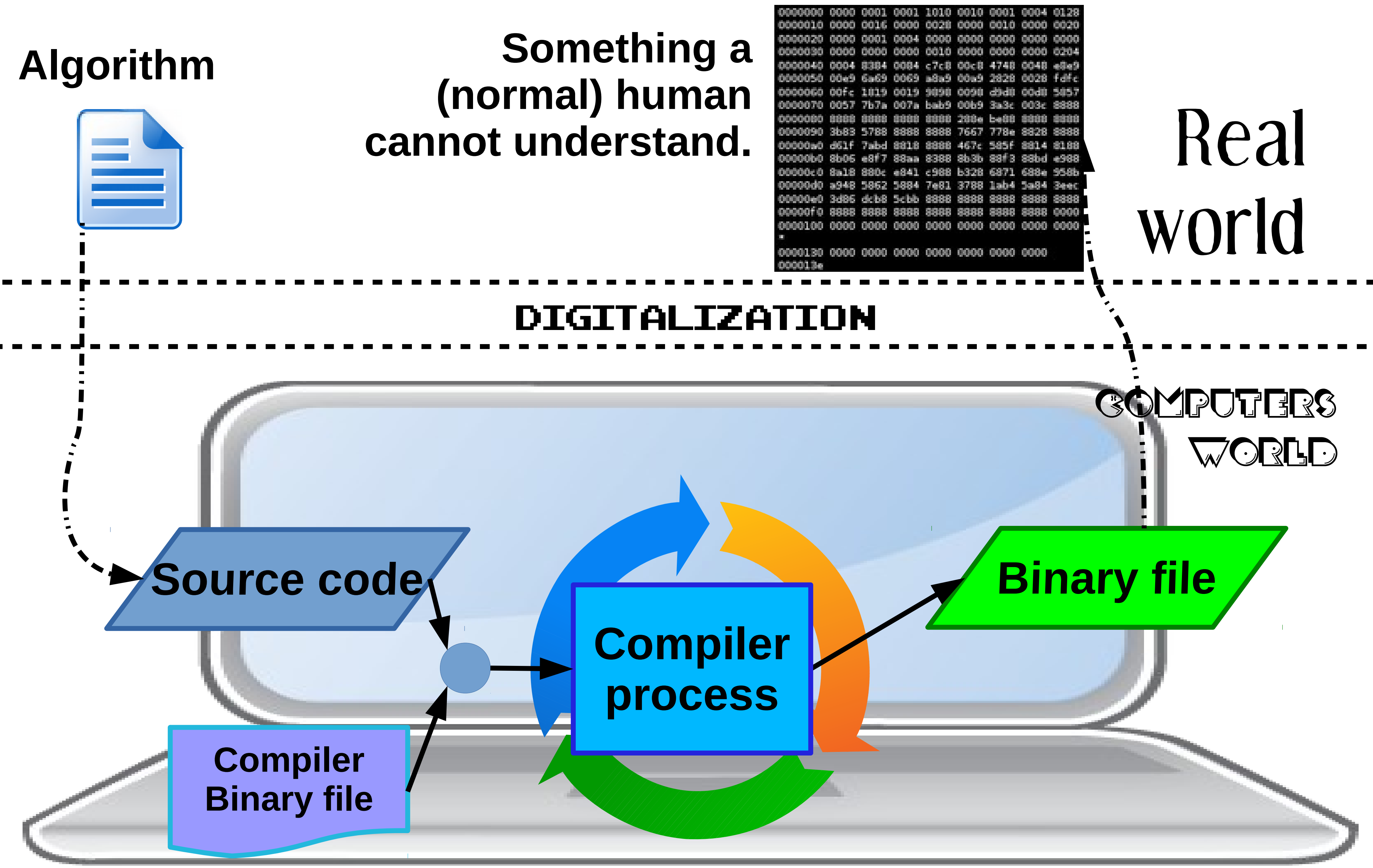


Compilation (static - compile time)
Software Libraries
OS Libraries
Code
Variable A
Variable B
Pointer C
Compilation
Compiler
Assembly
A, →0
B, →1
C, →2
Linking
Linker/ Compiler
lubuntu
Binary file
OSlib(A, →0)
OSlib(B, →1)
OSlib(C, →2)
Memory request in the form of variables or pointers
Association between variable names and pointers to Logical memory addresses
Static relocation: Relative/relocatable logical memory addresses created by the linker
OS Libraries
lubuntu
Mybinary.bin
Mybinary.bin process
→837015
→837017
→837019
Dynamic relocation Real memory addresses assigned by OS libraries
Execution (dynamic - runtime)

Memory Relocation

- Not all the memory is available for users programs, because also the operating system uses it.
- The programmer doesn't want to care about the specific memory address. **He/She/Ze just wants some memory!**
  - At **compile time**:
    - The developer memory space starts from a *virtual* location 0, that is actually mapped to some *physical* location
    - The linker **statically assigns virtual memory addresses** relative to some feature that the operating system offers to the compilation process.
  - At **runtime**:
    - The Operating System will **dynamically relocate memory addresses** for the program to execute.

Compilation workflow



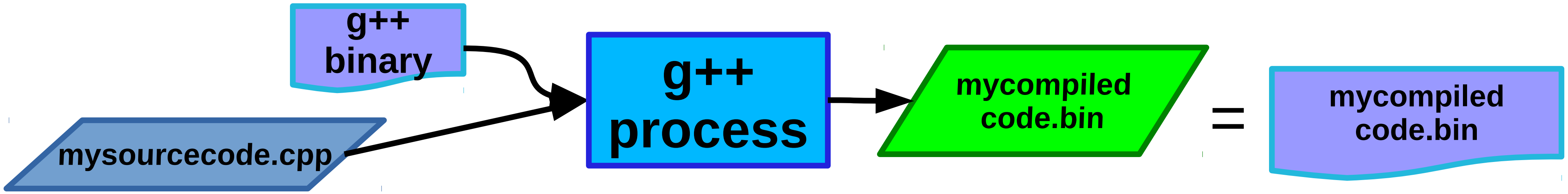
```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1019 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Compiled languages

- Classic programming languages like C or C++ are said to be **compiled** as the creation of an executable works as shown in the previous slides.
  - The developer will have to

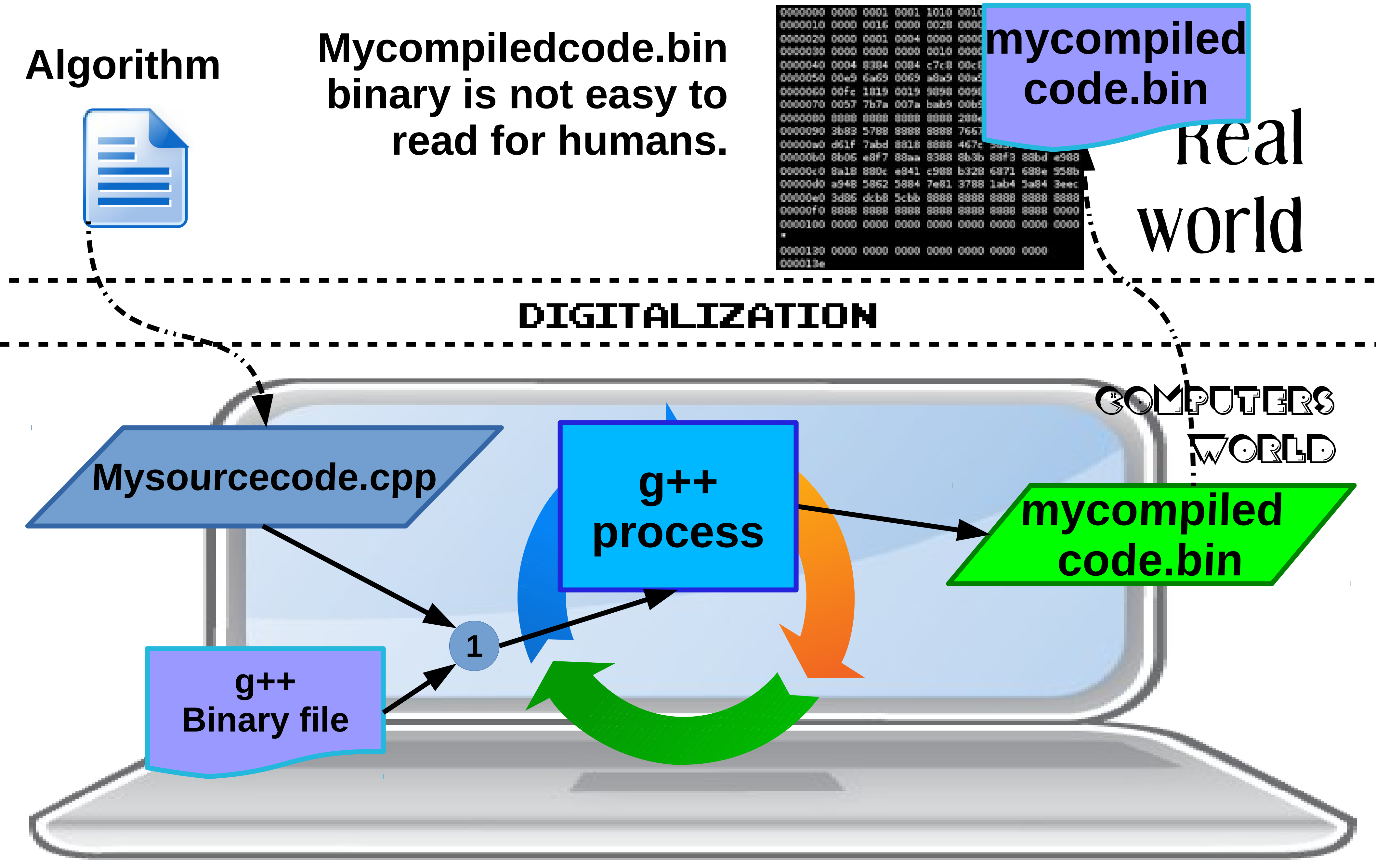
  1) **Compile** her *source code*
  Example: compile a C++ source file and generate a binary file `mycompiledcode.bin`:
  `g++ -o mycompiledcode.bin mysourcecode.cpp`

  - **run** or **execute** the binary code to see his program in action.
  Example: run `mycompiledcode.bin` binary file
  `./mycompiledcode.bin`
- Note: `mycompiledcode.bin` is an output file. g++ and `mycompiledcode.bin` are binary files. g++ is a program that generates binary files as its output.



Compilation workflow: C++

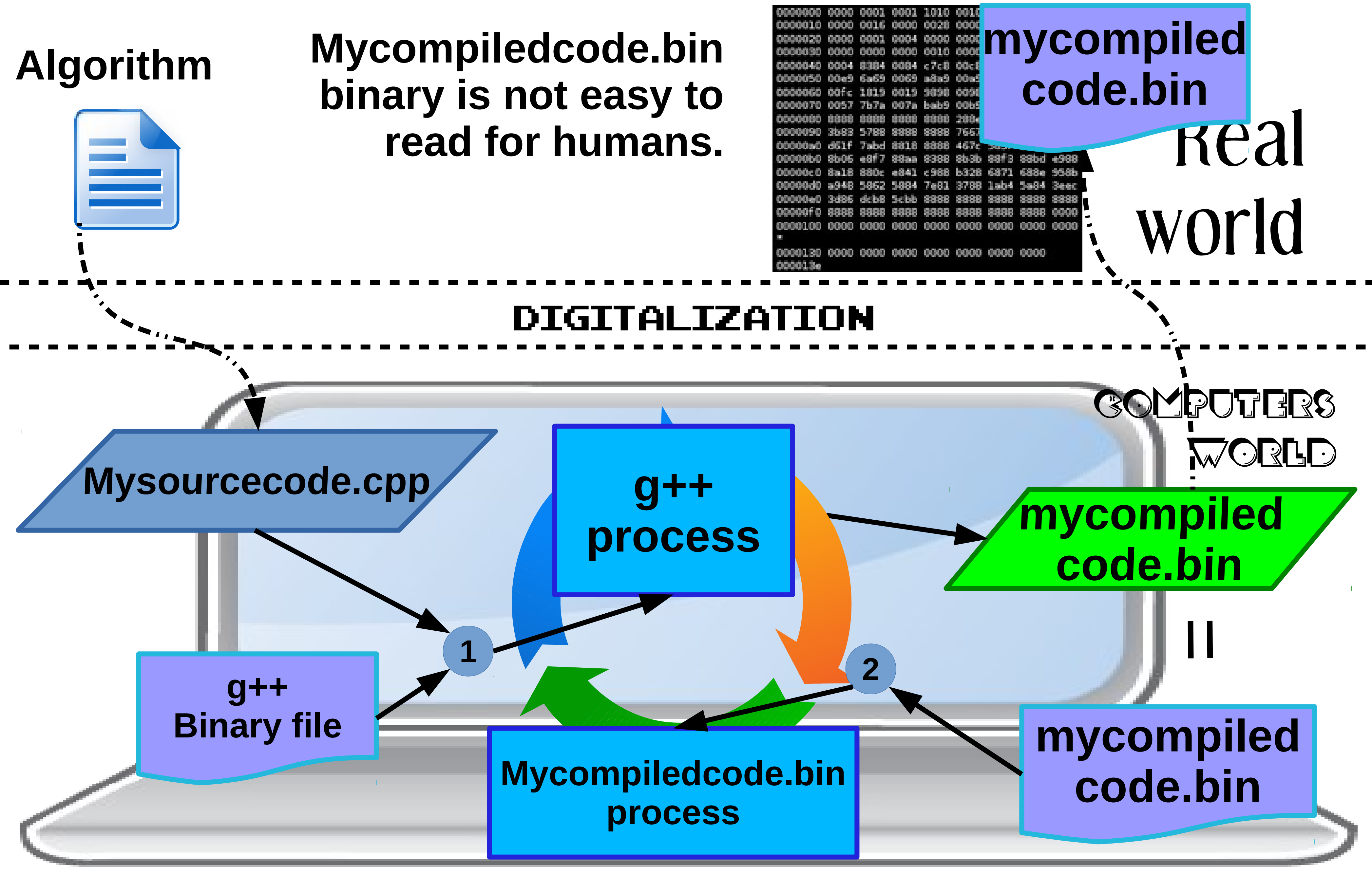
Mycompiledcode.bin binary is not easy to read for humans.



```
0000000 0000 0001 0001 1010 0010
0000010 0000 0016 0000 0028 0000
0000020 0000 0001 0004 0000 0000
0000030 0000 0000 0000 0010 0000
0000040 0004 8384 0084 c7c8 00c
0000050 00e9 6a69 0069 a8a9 00a
0000060 00fc 1019 0019 9098 009
0000070 0057 7b7a 007a bab9 00b
0000080 8888 8888 8888 8888 288
0000090 3b83 5788 8888 8888 766
00000a0 d61f 7abd 8818 8888 467c
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Compilation workflow: C++

Mycompiledcode.bin binary is not easy to read for humans.



```
0000000 0000 0001 0001 1010 0010
0000010 0000 0016 0000 0028 0000
0000020 0000 0001 0004 0000 0000
0000030 0000 0000 0000 0010 0000
0000040 0004 8384 0084 c7c8 00c8
0000050 00e9 6a69 0069 a8a9 00a
0000060 00fc 1819 0019 9898 0098
0000070 0057 7b7a 007a bab9 00b
0000080 8888 8888 8888 8888 288
0000090 3b83 5788 8888 8888 766
00000a0 d61f 7abd 8818 8888 467c
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Interpreted languages

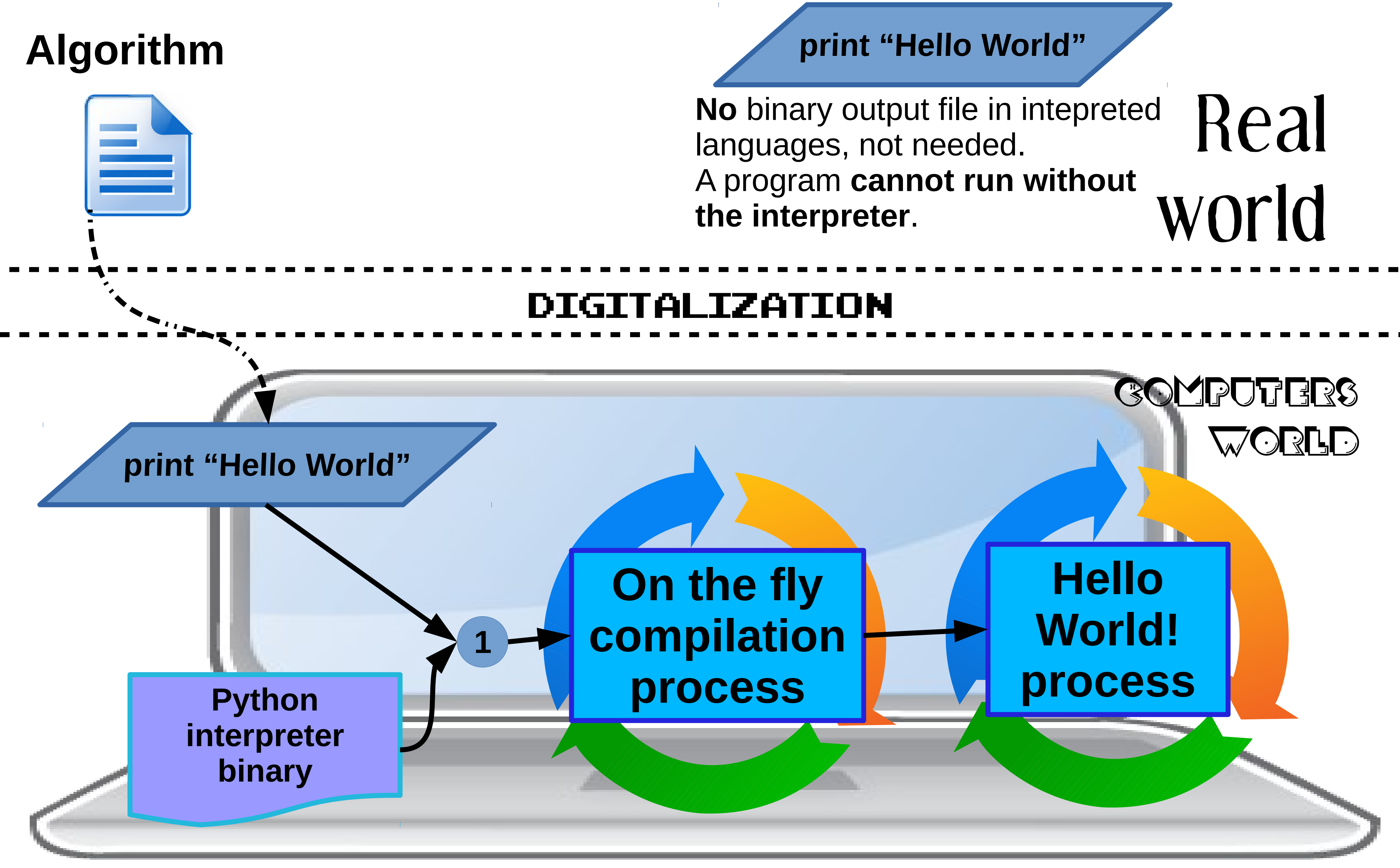
- Some languages like Python or PHP have another approach, where compilation **is done on the fly** by an helper compiler process. In this case the compiler process is called **interpreter**.
- The developer can just write a line of code inside the interpreter command line interface and this is **immediately executed**. Compilation is transparent.
- Example: Write “Hello World” in Python:
  - Run the python interpreter

    ```
    python
    Python 2.4.3 (#1, Jun 18 2012, 09:40:07)
    [GCC 4.1.2 20080704 (Red Hat 4.1.2-52)] on linux2
    Type "help", "copyright", "credits" or "license" for more information.
    ```
  - Execute a python command

    ```
    >>> print "hello world"
    hello world
    >>>
    ```
- The source code in this case is a **list of commands** to be *passed* to the interpreter to be executed.
  Example:
  `python mysourcecode.py`
- Question: what about BASH from the Tutorials? Discuss.

Steps to interpretation: Python

No binary output file in intepreted languages, not needed.
A program **cannot run without the interpreter**.



Compiled VS Intepreted

	Compiled	Interpreted
Performance	High	Low
Coding Complexity	High	Low
Portability	Low	High
Learning Curve	High	Low
Performance Tuning	Very High	Very Low
Capacity requirements	Very Low	Very High
Debugging features	Medium (depends on platform/compiler)	High

Compiled, use if:

- Need performance on intensive calculations
- Require specific technologies
- Small devices with limited memory or CPU

Intepreted, use if:

- Need to quickly create a prototype
- Require easy portability on different platforms
- Only on powerful computers

Compiled vs Interpreted in scientific computation

- **Compiled** languages are used when in need of **performance**, **precision** or **optimization**:
  - machine-consuming tasks that require lots of memory and time, to minimize memory and cpu consumption:
    - Intensive computation (when it takes days or weeks to obtain a result)
    - Complex simulation models (montecarlo, data reconstruction)
    - Parallel computing
  - Dedicated hardware tasks:
    - To take such hardware features to the limit
  - Dedicated hardware with limited resources:
    - Detectors
    - Mobile phones
    - Embedded devices

Compiled vs Interpreted in scientific computation

- **Interpreted** languages are used for **tedious tasks** that are not going to be executed too frequently, and **quick development**:
  - Creation of quick proof-of-concept prototypes
  - Submission of multiple computing jobs with multiple parameters
  - Streamlining/orchestration of complex computing tasks carried on with compiled languages binary code
  - Scripts that cannot be easily written in BASH.

Comparison between languages and when they work best

- Every language is usually designed for a **specific purpose**, and then extended to serve other purposes.
- Sometimes a language is to tightly close to its designed purpose that no extension really changes a programmer way of thinking
- Sometimes the practical use of a language goes **very very far from the purpose of which it was designed**

Bash

Features:

- Interpreted
- Runs commands, executables
- Imperative paradigm
- Not explicitly typed
- No memory pointers: only environment

Preferred use:

- Scripting
- Automation of command tasks
- Combine several commands

Pros:

- Use existing commands to do tasks
- Lots of community experience
- Very low learning curve
- Very intuitive approach

Cons:

- Not portable; code depends on installed software
- Lack of types might cause unexpected results
- No memory management, only environment variables might cause scope issues: all variables are global!
- Not rich in native datastructures, that are hard to use and very rarely used in practice

Bash example

Reading and printing a file to screen – executing the script

```
#!/bin/bash
# script readgames.sh
#

DATAFOLDER='../../data'
FILECONTENTS=$(cat ${DATAFOLDER}/nintendowiigames.xml)
echo "$FILECONTENTS"
```

1 Make the script executable and execute it:

```
pflorido@tjatte:~> chmod +x ./readgames.sh
pflorido@tjatte:~> ./readgames.sh
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
```

C

Features:

- Compiled
- Imperative paradigm
- Functions
- Types and type creation
- Memory Pointers
- Based on standards

Preferred use:

- System development
- Embedded devices
- Low-level coding, i.e. hardware drivers
- Performance

Pros:

- Very efficient
- Can directly use Assembly
- Lots of community experience
- Good debugging tools
- Control on the code preprocessor (for efficiency)

Cons:

- Requires deep knowledge of pointers and memory handling – developer has to free memory by herself
- Has high learning curve
- No object oriented approach: if new features need to be added, code needs to be rewritten or revised
- Hard to foresee runtime errors at compile time
- Control on the code preprocessor (hard to debug and understand)

C example

Reading and printing a file to screen

```
/*
 * readgames.c
 *
 * Copyleft 2016 Florido Paganelli<florido.paganelli@hep.lu.se>
 *
 */

// standard library to allocate memory
#include <stdlib.h>
// input/output library
#include <stdio.h>

int main(int argc, char **argv)
{
    // a sequence of chars will contain the file
     char *filecontents;
    // C doesn't automatically know the size of a file
    long input_file_size;
    // opening the file nintendowiigames.xml for reading
    FILE * input_file = fopen("../../data/nintendowiigames.xml", "rb");
    // Calculating the size of the file:
    // reach the end of the file
    fseek(input_file, 0, SEEK_END);
    // get the position of the pointer: will give us how big is the file
    input_file_size = ftell(input_file);
    // go back at the beginning of the file
    rewind(input_file);
    // allocate memory for file contents
    filecontents = malloc(input_file_size * (sizeof(char)));
    // read the file regardless of newlines
    fread(filecontents, sizeof(char), input_file_size, input_file);
    // close the file
    fclose(input_file);

    //print the content of the variable
    printf("%s",filecontents);
     return 0;
}
```

C example

Reading and printing a file to screen – compile and execute

1 Compile:

```
pflorido@tjatte:~> gcc -o readgames.c.bin readgames.c
```

2 Execute (the object file is already executable!):

```
pflorido@tjatte:~> ./readgames.c.bin
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

C++

Features:

- Compiled
- Imperative paradigm
- Object oriented paradigm
- Types and type creation
- Templating
- Memory Pointers
- Based on standards

Preferred use:

- System development
- Embedded devices
- Low-level coding, i.e. hardware drivers
- Performance

Pros:

- Very efficient
- Empowers C with objects, allowing extending existing code
- Can directly use Assembly
- Lots of community experience
- Good debugging tools
- Good coding environments
- Control on the code preprocessor (for efficiency)

Cons:

- Requires deep knowledge of pointers and memory handling – developer has to free memory by herself
- Has high learning curve
- Not suitable for fast prototyping
- Hard to foresee runtime errors at compile time
- Control on the code preprocessor (hard to debug and understand

C++ example

Reading and printing a file to screen

```cpp
/*
 * readgames.cpp
 *
 * Copyleft 2016 Florido Paganelli <florido.paganelli@hep.lu.se>
 *
 */

// library for basic input/output
#include <iostream>
// library for files stream
#include <fstream>
// library for strings stream
#include <sstream>
// library for strings
#include <string>
// if not specified, the functions belong to the std namespace
using namespace std;

int main(int argc, char **argv)
{
    // create a stream of strings
    std::stringstream filecontents;
    // create an input file stream
      ifstream myfile;
    // open the nintendowiigames.xml file as a file stream
      myfile.open ("../../data/nintendowiigames.xml");
    // if the open was successfull
      if (myfile.is_open())
      {
       // stream the contents of the file inside the string stream
       filecontents << myfile.rdbuf();
      }
    // close the file
      myfile.close();
    // convert the stream to a string
    string contents(filecontents.str());
    // print out the string
    cout << contents;
      return 0;
}
```

C++ example

Reading and printing a file to screen – compile and execute

1 Compile:

```
pflorido@tjatte:~> g++ -o readgames.cpp.bin readgames.cpp
```

2 Execute:

```
pflorido@tjatte:~> ./readgames.cpp.bin
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

Python

Features:

- Interpreted
- Portable
- Imperative paradigm
- Object oriented paradigm
- Not typed
- Templating
- No memory pointers: memory is managed by the interpreter

Preferred use:

- Scripting
- Application prototype development
- Cross platform development
- Very High level coding

Pros:

- Portable, given one has the same verison of the interpreter
- Objects allowing reuse and extension of existing code
- No need to care about freeing memory, locations are cleared by Python Garbage Collector
- Lots of community experience
- Very low learning curve
- Very intuitive approach
- Can use C/C++ code

Cons:

- Portability depends on interpreter version
- Automatic memory management imposes huge memory requirements on the machine: not efficient
- Enviroment and scope models not very intuitive, runtime behaviour might be unexpected
- Lack of types might cause unexpected results
- Semantic not well defined: references, pointer like datatypes, can be hard to see looking at the code

Python example

Reading and printing a file to screen

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  readgames.py
#
#  Copyleft 2016 Florido Paganelli <florido.paganelli@hep.lu.se>
#
#
#

def main():
    # open the file as f
    with open('../../data/nintendowiigames.xml','r') as f:
        # read the whole contents
        contents = f.read();
    # close the file
    f.close();
    # output the contents
    print contents;
    return 0

if __name__ == '__main__':
    main()
```

Python example

Reading and printing a file to screen – pass to interpreter or run script

1 Pass the file to the intepreter to be executed:

```
pflorido@tjatte:~> python readgames.py
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

1 Alternatively, since we specified the intepreter at the beginning of the script, make the file executable and execute the file:

```
pflorido@tjatte:~> chmod +x readgames.py
pflorido@tjatte:~> ./readgames.py
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

Golden rules of a scientific programmer

(1) Never trust the computer, but trust your scientific intuition

- Remember the digitalization problem: a computer reduces precision

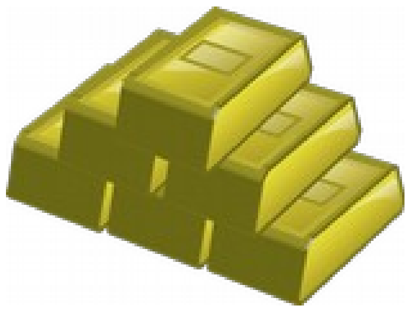
(2) Keep your code simple and functionalities separate in your code

- Write and test each functionality
- Will help you figure out what is wrong

(3) Write many (significant) comments

- Science is knowledge sharing: others will read your code sooner or later

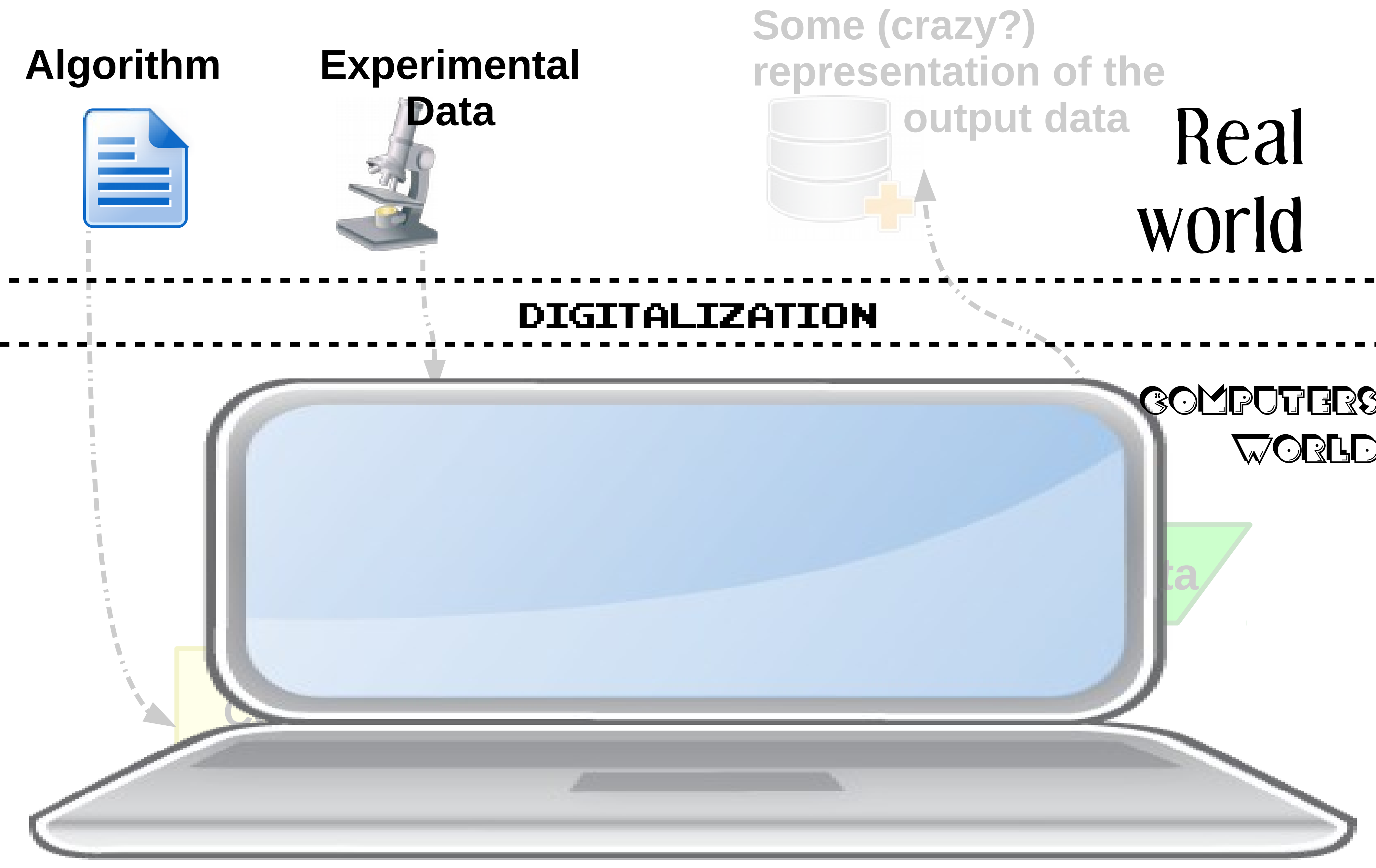
(4) Don't blame the sysadmin until you're sure it's his/her fault! ;-)



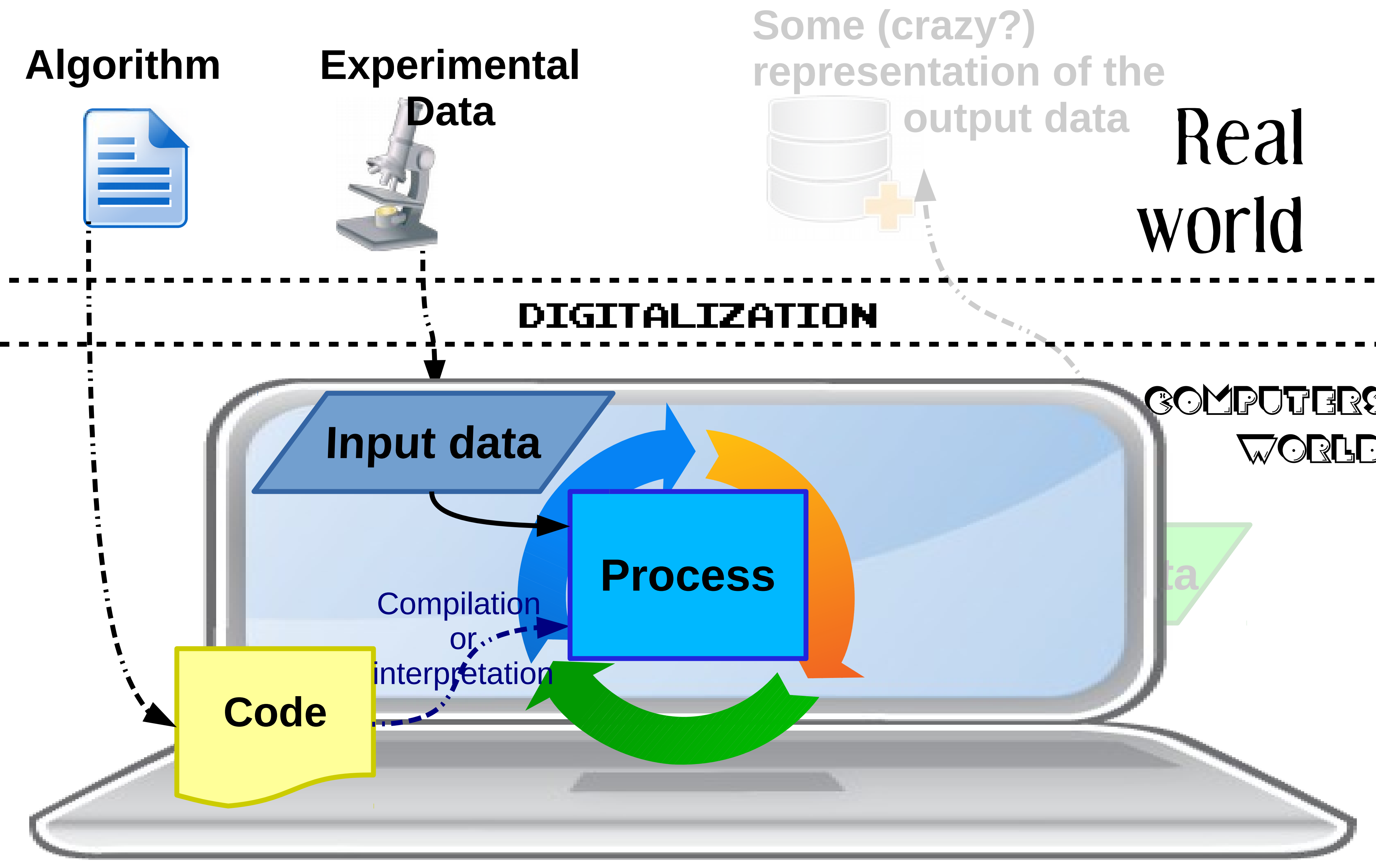
Questions?

Additional Material

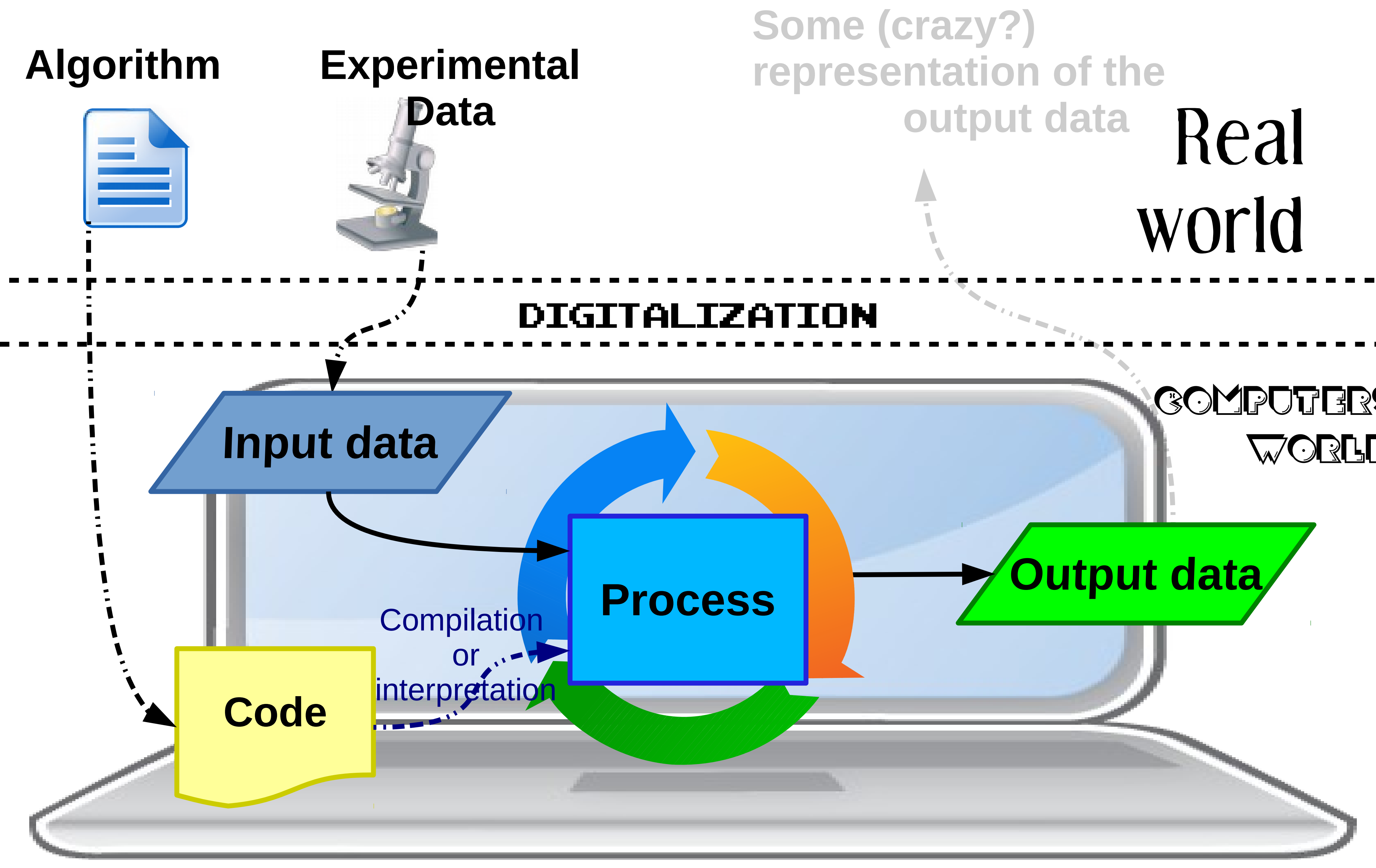
The information flow



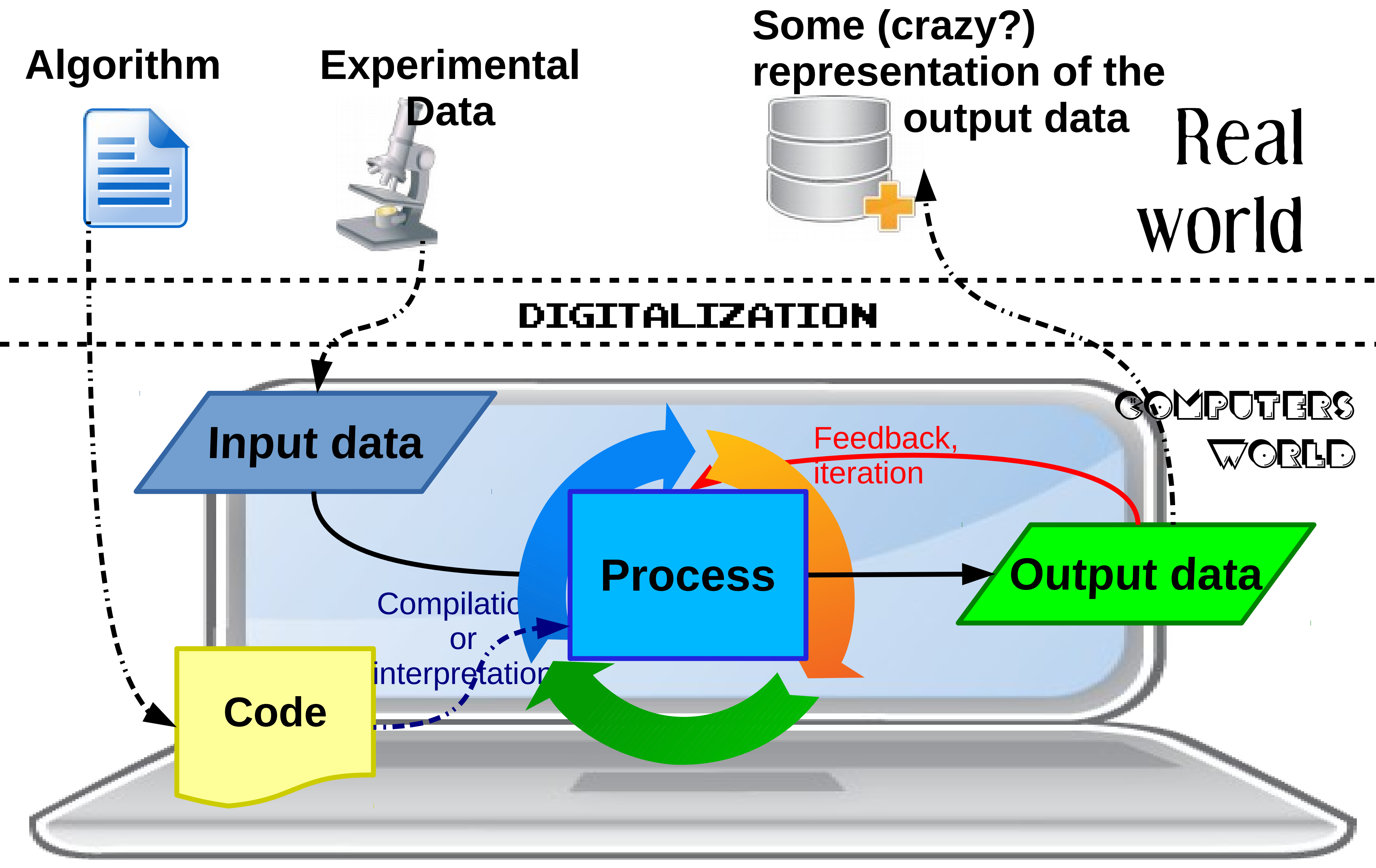
The information flow



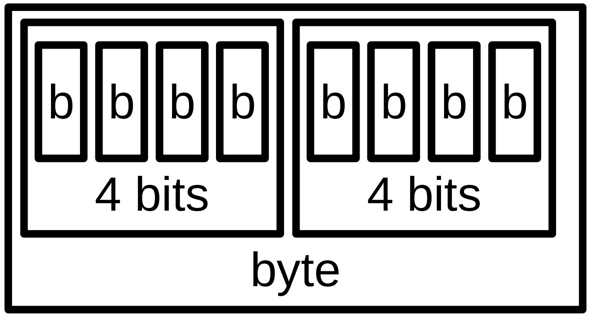
The information flow



The information flow

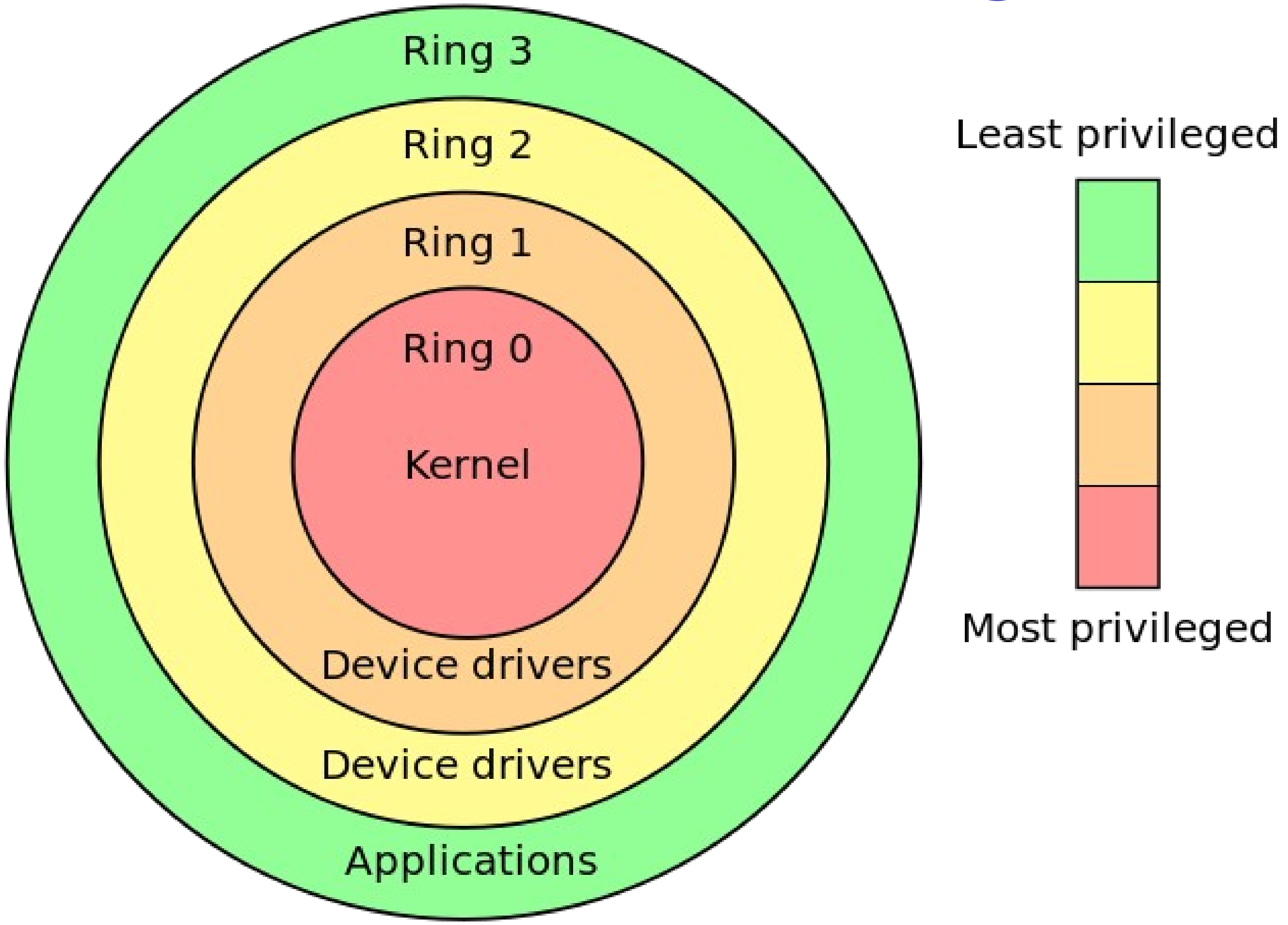


Memory size detailed



- Memory is measured in **bytes**.
- Since we know how many values we can have in a register made of 32 or 64 bits, it's handy to use the binary system (base 2) to identify the size of a memory bank.
- Byte unit of measure follows the base 2 we presented before. The concept behind this weird choice is historically related to **counting groups of 4 bits**. So:
- 1 byte = 1 byte * 2^0 = **2 groups of 4 bits each**, 2*4 = 8 bits is the fundamental “quantity” of memory information.
- 2 bytes = 1 byte * 2^1 = 4 groups of 4 bits, 4*4 = 2*8 = 16 bits
- 1024 bytes = 1 byte * 2^{10} is called a Kilobyte. Often noted as Kb or kb or KB (unfortunately producers never agreed on the notation). Conversion to the different orders is done by dividing/multiplying for 1024 in decimal notation. Examples:
  - 1 Kilobyte = 1Kb = 2^{10} bytes = 1024 bytes
  - 1 Megabyte = 1Mb = 2^{20} bytes = 1048576 bytes = 1024 KB
  - 1 Gigabyte = 1Gb = 2^{30} bytes = 1073741824 bytes = 1048576 KB = 1024 MB
- A 4GB memory bank contains 4*1073741824 bytes = 4294967296 bytes = 2^{32} bytes = 4194304 KB = 4*1048576 KB = 4096 MB = 4*1024 MB

Protection Rings



Protection Rings

An operating system is organized such that an application cannot write on the other application's memory.
A **three-layered architecture** where memory access is controlled according to protection rings:

- the core **Ring 0** belongs to the kernel, who orchestrates the system. Nobody but the kernel can access its memory
- **Ring 1 and 2** are for programs that access the hardware and interact with the kernel directly for performance reasons. Some may write the kernel memory directly, some not.
  - **Ring 1**, Kernel modules usually write directly
  - **Ring 2**, Device drivers interact with the modules
- **Ring 3**, The external layer which is the one where we run our programs.

Bytecode-based languages

- Some languages like Java have an intermediate representation called **bytecode**.
- Bytecode is some sort of compiled code that cannot be executed by a real machine, but by a **Runtime Virtual Machine**. (NOTE: it is NOT like the virtual machine we saw in tutorials!).
- A **Runtime Virtual Machine** is a program that takes in *input* a bytecode file and *translates* it into a real machine binary code.
- The developer must:
  - Compile her *source code* to bytecode

    Example: generate bytecode file from source

    ```
    javac mysourcecode.java
    ```

    Output will be a `musourcecode.class` bytecode file

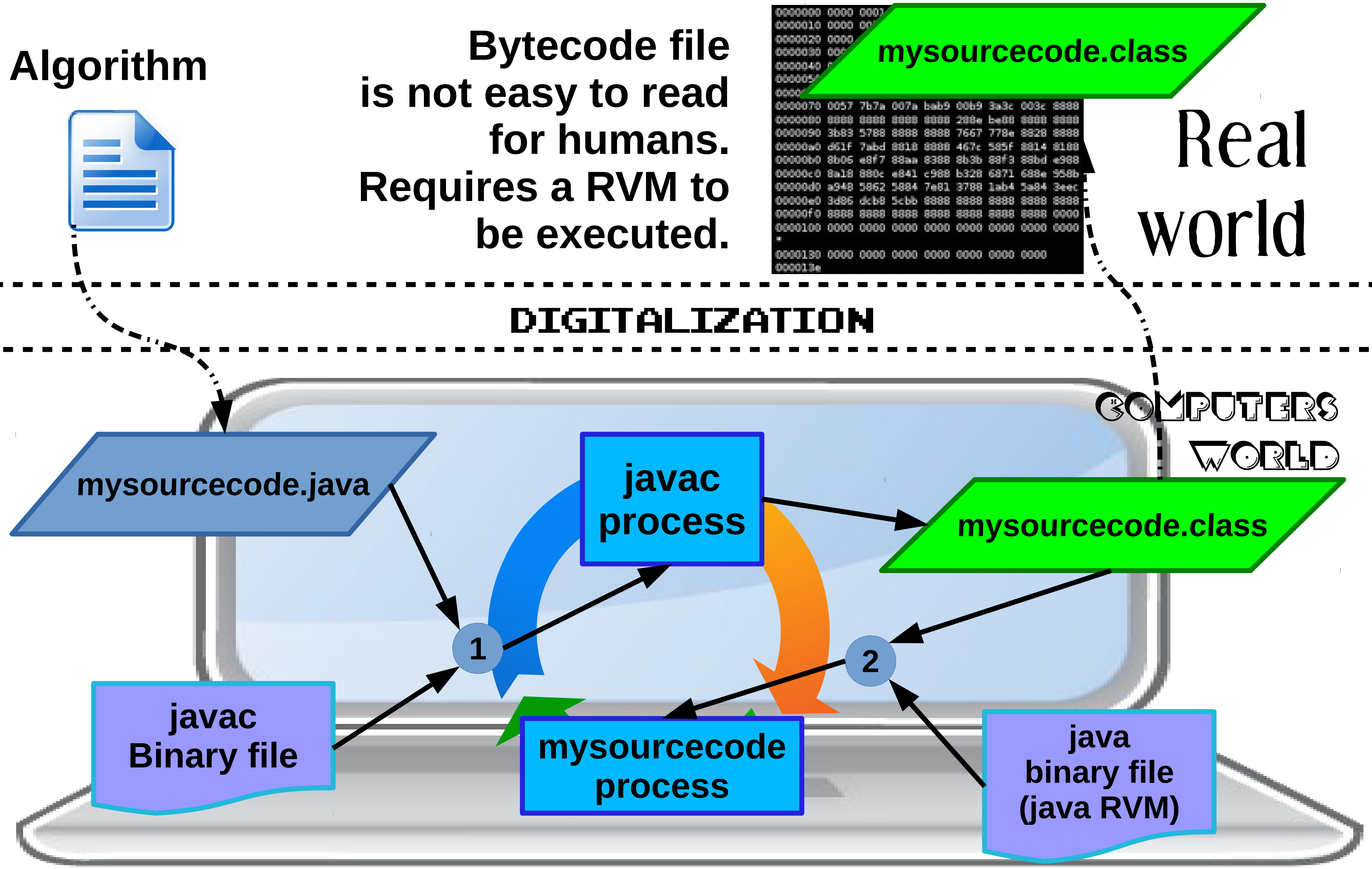
  1) *Pass* the bytecode as *input file* to a *runtime virtual machine* for it to run.

     Example: execute a generated bytecode file

     ```
     java mysourcecode.class
     ```

     The RVM will be started and the execution of the program will start.

Steps to bytecode compilation: Java



Bytecode file is not easy to read for humans. Requires a RVM to be executed.

```
0000000 0000 0001
0000010 0000 00
0000020 0000
0000030 00
0000040
00000
0000
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Dream and reality of Java

- Java's bytecode and Virtual Machine goal was to create a **type-safe**, object oriented **portable** language.
- **Type-safe**: means that the languages always enforces that data types are correct. This is also done by requesting the programmer to take care of eventual bad situations at compile time. This has actually been achieved; but if the programmer fails to do that the code dies badly.
- **Portability**: Bytecode was an attempt to **decouple the physical machine from the computation model**. Unfortunately, in the end the Virtual Machine must "talk" with the actual machine, and that's where portability **failed**.
  - **Different versions of the virtual machine** for Windows, Linux and Mac, not always compatible. Moreover, there are **different implementations** of the JavaVM that are not always compatible
  - **Software Development Kit changes all the time,** making it impossible to write an application that can work with a newer version of the virtual machine. One needs to update both the libraries and the VM.
  - **Efficiency drop**: The virtual machine is usually slower than the real machine; Automatic garbage collection (that allows the programmer not to care about memory problems) causes high memory consumption and makes this language **a bad choice for intensive scientific computation – performance will quickly drop and one will need more powerful hardware.**

Java

Features:

- Bytecode Compiled for a Runtime Virtual Machine (RVM)
- Portable
- Imperative paradigm
- Object oriented paradigm
- Types and type creation
- Templating
- No memory pointers: memory is managed by the RVM

Preferred use:

- Application development
- Cross platform development
- Embedded devices
- High level coding
- Server-Client architectures
- Big projects

Pros:

- Portable, given the RVM can run it
- Objects allowing reuse and extension of existing code
- Developers do not need to care about freeing memory, all is taken care by the RVM *Garbage Collector*
- Lots of community experience
- Very good debugging tools and coding environments

Cons:

- Portability depends on RVM version, in reality is not really achieved; RVM and SDK updates may break code compatibility
- Has high learning curve
- Not suitable for fast prototyping
- Automatic memory management imposes huge memory requirements on the machine: not efficient
- In the last years a lot of security holes have been discovered in the RVM, needs continuous update

Java example

Reading and printing a file to screen

```
/*
 * readgames.java
 *
 * Copyleft 2016 Florido Paganelli <florido.paganelli@hep.lu.se>
 *
 */

// import basic input/output java libraries
import java.io.*;
// import java utility Scanner
import java.util.Scanner;

// everything is a class in java
public class readgames {
    // cause specific file errors in case of problems
     public static void main (String args[]) throws FileNotFoundException, IOException {

        String text = new Scanner( new File("../../data/nintendowiigames.xml") ).useDelimiter("\\A").next();
        // try this code
        try {
            // create an output buffer to standard output
            BufferedWriter output = new BufferedWriter(new OutputStreamWriter(System.out));
            // write the content of text on output
            output.write(text);
            // empty the content of standard out to screen
            output.flush();
        }
        // print an error if it fails
        catch (Exception e) {
            e.printStackTrace();
        }
     }
}
```

Java example

Reading and printing a file to screen – compile to bytecode and launch RVM

Compile and generate a class file:

```
pflorido@tjatte:~> javac readgames.java
pflorido@tjatte:~> ls
readgames.class  readgames.java
```

Launch the Java Virtual Machine and execute the class file:

```
pflorido@tjatte:~> java readgames
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

References

- Binary code:
  http://www3.amherst.edu/~jcook15/binarycode.html
- A brief history of computing
  http://ludwig.lub.lu.se/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat01310a&AN=lovisa.003214669&lang=sv&site=eds-live&scope=site
- Example data taken from The Game Database:
  http://www.thegamesdb.net/
  http://wiki.thegamesdb.net/index.php/API_Introduction
- Numbers representation
  https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Pictures references (not complete)

- http://www.jegerlehner.ch/intel/
- http://www.cpu-world.com/CPUs/68000/
- http://en.wikipedia.org/wiki/X86
- http://en.wikipedia.org/wiki/Protection_ring