

Computer Systems

UNIT 02. INFORMATION REPRESENTATION



Computer systems
CFGS DAM/DAW

Sergio García / Alfredo Oltra

2022/2023

Versión:220911.2030

Licencia



Reconocimiento - NoComercial - CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

INDEX

1. Introduction.....	4
1.1 Piece of information and information.....	4
1.2 Data internal representation.....	4
2. Numeral systems.....	5
2.1 Binary code.....	5
2.1.1 How to convert a decimal number into a binary number.....	5
2.1.2 How to convert a binary number into a decimal number.....	7
2.1.3 Maximum number of values to represent.....	8
2.1.4 Operations with binary numbers.....	8
2.1.5 Negative numbers.....	9
2.1.6 Real numbers.....	11
2.1.7 Boolean algebra.....	13
2.2 Octal.....	15
2.2.1 How to convert a binary number into octal.....	15
2.2.2 How to convert an octal number into binary.....	15
2.3 Hexadecimal.....	15
2.3.1 How to convert binary numbers into hexadecimal.....	16
2.3.2 How to convert hexadecimal numbers into binary.....	16
2.3.3 How to convert hexadecimal numbers into octal numbers.....	16
3. Alphanumeric Representation.....	17
3.1 Numeric and alphanumeric data.....	17
3.2 Internal representation.....	17
4. Unit system.....	18
5. Additional material.....	19
6. Bibliography.....	19

UNIT 02. INFORMATION REPRESENTATION

1. INTRODUCTION

1.1 Piece of information and information

Computers (or more correctly information systems) are machines designed for processing information or, in other words, to get results from the application of operations on a data set. But, what is information? What is a piece of information? And what is an operation?. Take an example:

The temperature is 30°

- Piece of information: formal representation of a concept, in this case: “30”
- Information: the result of the interpretation of the data: “It's hot”
- Operation: rule applied to get information: “As the temperature is higher than 23, it's hot”

1.2 Data internal representation

Therefore, we need to store and to handle in computers data and operations. And for that, they need to use the binary code.

⚡ All kind of data, both numbers or letters, are stored using this system.

This system is based on the use of only two digits, 0 and 1, unlike the decimal system that uses ten (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). This is because computers only know these two numerical values resulting from the detection or not of some potential, of a number of volts. Thus, a computer knows there's a 0 when the potential measured in an inner member has a value close to 0 volts. Otherwise, it detects a 1.

🔊 In electrical terms, the potential could be assimilated to the strength in which the electric current passes through a wire.

🔊 In general, values of 1 usually correspond to potential around 3 or 5 volts.

All computer elements handle this numbering and interpretation system of information. It might be said that computers actually know nothing at all. They only know about 0's and 1's and how to perform some basic operations with them (+, -, * ...), although faster.

2. NUMERAL SYSTEMS

A numeral system is a set of **sorted symbols** used to represent quantities. The number of symbols is called **system base**.

In the real world, we are used to use decimal system (base 10) which set of sorted symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Any number, represented in any numeral system, can be split in digits. For instance, 128 can be split in 1, 2, 8 or 34,76 in 3, 4, 7, 6. From these digits and with their position and the system base is possible to get again the number:

$$128 = 1 * 10^2 + 2 * 10^1 + 8 * 10^0$$

$$34,76 = 3 * 10^1 + 4 * 10^0 + 7 * 10^{-1} + 6 * 10^{-2}$$

We can see that a decimal number can be represented as additions of powers of 10 (the decimal system base).

If we generalize, a number **N** expressed in a numeral system **B** would be like:

$$N = a_{n-1} a_{n-2} \dots a_1 a_0 , a_{-1} a_{-2} \dots a_{-p+1} a_{-p}$$

where:

N: number to represent


a: the symbols that our numeral system includes (integers from 0 to B-1)


The digits before the comma (,) ¹ are the integer part.

The digits after the comma (,) are the fractional part.

2.1 Binary code

The binary code is a numeral system which system base is 2 and its symbols are 0 and 1.

 Each digit of a binary number is called **bit** and it is the smallest unit of information, in other words, it is the least that can be represented

 To avoid confusion, it is usual to indicate the base system number to be represented by a subscript to the right. For example $101_{(10)}$ or $101_{(2)}$

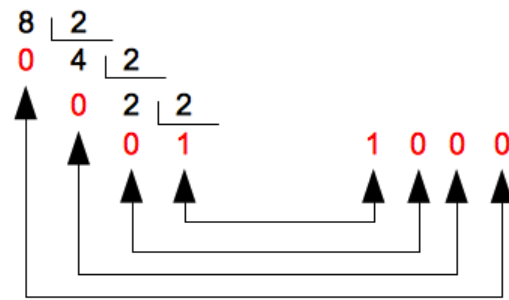
2.1.1 How to convert a decimal number into a binary number

In general, to convert a decimal number into another base, we have to perform successive divisions of the number by the base. At the end, we have to get the remainders and the last quotient and sorted them in the opposite direction.

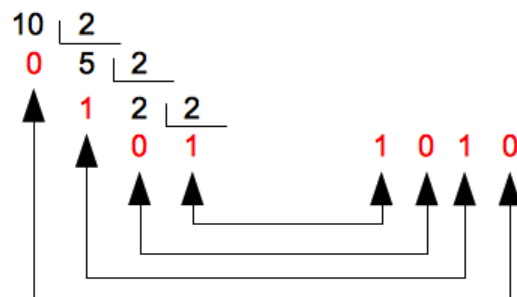
¹ In the English culture, the separation between the decimal part and fractional part is a decimal point (.)

Consider the case of convert a decimal into binary with some examples:

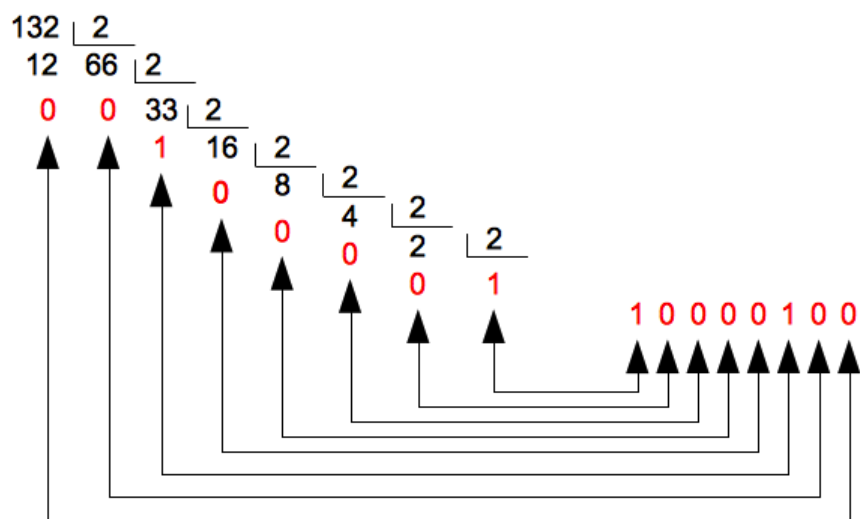
$8_{(10)} \Rightarrow ?_{(2)}$




$10_{(10)} \Rightarrow ?_{(2)}$

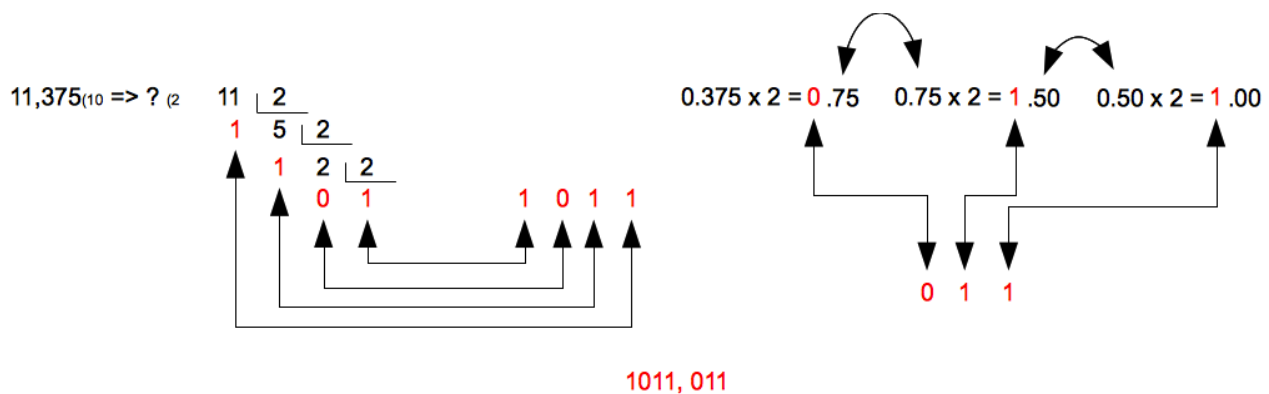


$132_{(10)} \Rightarrow ?_{(2)}$



In numbers with fractional part, the process is the same for the integer part, but the fractional part is calculated multiplying by 2 successively and to take the integer part (in this case in right order).

 The leftmost bit it is called most significant bit (MSB) and the rightmost bit it is called least significant bit (LSB).



2.1.2 How to convert a binary number into a decimal number

In this case the process is very easy. As explained above, a decimal number can be represented as additions of powers of ten.

On the whole, it can convert the value of a number represented in a numeral system B^2 into decimal system using the next formula:

$$N = a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + \dots + a_1B^1 + a_0B^0 + a_{-1}B^{-1} + \dots + a_{-p}B^{-p} = \sum_{i=-p}^{n-1} a_i B^i$$

We are going to use it to convert into base 2

$$101001_{(2)} \Rightarrow ?_{(10)}$$

$$101001 \Rightarrow 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 41$$

The process involves four steps

1. To write the binary number figures multiplied by 2.
2. To write a plus sign (+) between each of products.
3. To write an exponent in each 2, starting from zero and from the last number of the integer part (on the far right if there is not fractional part) and increasing it one by one to the left and decreasing to the right.
4. To perform the operation

$$10,01_{(2)} \Rightarrow ?_{(10)}$$

$$10,01 \Rightarrow 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2,25$$

2 As we will see later, B can be any numeral system

2.1.3 Maximum number of values to represent

One of the typical questions when handling a binary number is to know what is the maximum decimal value that can be represented by a certain bits number. The answer is easy: 2^n , where n is the bit number. For instance, with 4 bits we can represent 16 values, from 0 to 15 (0000-1111)

2.1.4 Operations with binary numbers

Binary addition and subtraction follow the next rules:

Addition:

$$\begin{aligned} 0 + 0 &= 0 \\ 1 + 0 &= 1 \\ 0 + 1 &= 1 \\ 1 + 1 &= 0 \text{ (carry 1)} \end{aligned}$$

Subtraction:

$$\begin{aligned} 0 - 0 &= 0 \\ 1 - 0 &= 1 \\ 0 - 1 &= 1 \text{ (carry 1 to subtrahend)} \\ 1 - 1 &= 0 \end{aligned}$$

The operations result is the same as their related decimal operations, except for the cases where the result don't have a value in the binary system, that is $1+1$, which can not be represented by 2 and $0-1$, which can not represent by -1 . This is where carry-over is important.

Some examples:

$$\begin{array}{r} \textcolor{red}{11} \\ 10011010 \\ + 01001100 \\ \hline 11100110 \end{array}$$

$$\begin{array}{r} \textcolor{red}{111111} \\ 1011 \\ + 111101 \\ \hline 1001000 \end{array}$$

⚡ If we want to add two binary numbers which addition is greater than the maximum number to represent the computer throws an *overflow* warning. For instance, if we have a computer who works with 8 bits, it can represent from $0_{(10)}$ to $255_{(10)}$. If we want to add 10000000_2 ($128_{(10)}$) plus 10000000_2 ($128_{(10)}$) we have a problem because the result is 100000000_2 ($256_{(10)}$) greater than 255. So an *overflow* occurs.

$$\begin{array}{r} \textcolor{red}{1} \\ 10000000 \\ + 10000000 \\ \hline \textcolor{red}{1}00000000 \end{array}$$

$$\begin{array}{r} 101101 \\ \textcolor{red}{1} \\ - 10101 \\ \hline 011000 \end{array}$$

$$\begin{array}{r} 11101 \\ \textcolor{red}{11} \\ - 00111 \\ \hline 10110 \end{array}$$

⚡ In the subtraction, the carry-over don't add to the minuend, but subtrahend.

Multiplication:

$$0 * 0 = 0$$

$$1 * 0 = 0$$

$$0 * 1 = 0$$

$$1 * 1 = 1$$

Division:

0 / 0 = Undefined

1 / 0 = Boundless

$$1 / 1 = 1$$

$$0 / 1 = 0$$

Both, multiplication and division, presents no difference from the related operations in decimal, unless the auxiliary operations are performed in binary.

✦ In multiplication, when we add, it may be that we have in the same column more than two 1's. In this case, we perform the additions in groups of two and are going to carry the 1's in the next column.

⚡ In division, we start getting in dividend and divisor the same number of figures. If you can not be divided, we try getting one figure more in the dividend.

If the division is possible, then, the divisor can only be contained once in the dividend, that is, the first quotient figure is 1. In this case, the result of multiplying the divisor by 1 is the divisor itself (the value that we will subtract).

$$\begin{array}{r} 101010 \\ -110 \\ \hline 1001 \\ 1 \\ -1110 \\ \hline 00110 \\ 110 \\ \hline 000 \end{array}$$

2.1.5 Negative numbers

When we need to represent a negative binary number, we have several options although three are the most important. This range indicates that the way to express it should be an agreement between two sides: the one that generates the number and one to read it. If not, the real value to be expressed would be wrong.

Signed magnitude

Perhaps it is the easiest approach to understand. The idea is to keep the MSB to indicate the sign of the number: 0 positive, 1 negative. The remaining bits indicate the number value in absolute value. For example:

Decimal	Binary	Positive Binary	Negative Binary
5	101	0101	1101

As can be seen, we need a bit to indicate the sign, so that what in normal representation values would be 0 to 15 in this case, to use the sign, becomes of -7 to +7 (1111 - 0111).

This system is simple to understand but complex to use when performing mathematical operations. Besides it has a problem: there are two ways to define the $0_{(10)}$: 0000_2 and 1000_2

Ones' complement

The second option also uses the first bit as sign indicator, but in this case the negative number is achieved complemented positive number (changing ones' by zeros and vice versa).

Decimal	Binary	Positive Binary	Negative Binary
5	0101	0101	1010

In this option is required to give the number of bits to encode, in such a way that if in the previous example we use 8 bits to encode:

Decimal	Binary	Positive Binary	Negative Binary
5	101	00000101	11111010

This method has the same problem that signed magnitude: there are two ways to define the $0_{(10)}$: 0000_2 and 1111_2

Two's complement

Although ones' complement simplifies the mathematical operations, they do much more with the use of two's complement. That is why it is the most used method.

Two's complement consists in to apply a ones' complement and then, to add 1. For instance, two's complement of 5 encoded with 8 bits is:

$$5_{(10)} \rightarrow 101_2 \rightarrow (\text{encoded in 8 bits}) 00000101_2 \rightarrow (1's \text{ complement}) 11111010_2 \rightarrow (+1) 11111011$$

Decimal	Binary	Positive Binary	Negative Binary
5	101	00000101	11111011

What decimal number represents a number in two's complement?. Easy. We have to perform the same process:

$$11111011_{(2)} \rightarrow (1's \text{ complement}) \rightarrow 00000100_{(2)} \rightarrow (+1) 00000101_{(2)} \rightarrow 5_{(10)}$$

The great advantage of the two's complement method is that it allows subtraction as if they were adds. This is because subtract two binary numbers is the same as adding to the minuend the complement of the subtrahend..

$$101101_{(2)} (45_{(10)}) - 010101_{(2)} (21_{(10)}) \Rightarrow 010101_{(2)} (1's \text{ complement}) \rightarrow 101010_{(2)} \rightarrow \\ \rightarrow (+1) 101011 \Rightarrow 101101 + 101011$$

$$\begin{array}{r} 101101 \\ + 101011 \\ \hline \end{array}$$

$$1011000_{(2)} \quad (24_{(10)}) \text{ the last carry-over } 1 \text{ is rejected}$$

Excess-K or offset binary

Depending on the number of bits available, mid-range is dedicated for negative numbers and the other half (minus 1) to the positives (the zero value is in the middle). The new range will be $[-K, K-1]$, where we can calculate by $K = 2^{n-1}$. Once we have the permissible range, the smallest number is who has all its bits to 0. Let us see an example:

We have 3 bits for representing the number so we can represent 2^3 numbers, the range $[0, 7]$. In this case, K will be $2^{3-1} = 2^2 = 4$, so the range with negative numbers will be $[-4, 3]$. The smallest number -4 will be 000 and the biggest 3 will be 111. The complete board will be:

-4	-3	-2	-1	0	1	2	3
000	001	010	011	100	101	110	111

If we have a number in *Excess-K* and we know its decimal value, we need to subtract the value of the excess to the decimal value. For instance, if $n = 8$ and is $K = 2^{n-1} = 128$,

$$11001100 \rightarrow 204 ; 204 - 128 = 76 \quad \text{or} \quad 00111100 \rightarrow 60 ; 60 - 128 = -68$$

2.1.6 Real numbers

When we write a real number in a paper we use a decimal comma (or decimal point, it depends of the culture) to distinguish between integer part and fractional part. In a computer, the space to represent this kind of numbers is divided in two areas: one for the integer part and one for the fractional part. There are two ways to denote the size of this areas (fields), and therefore, the comma position: *fixed point* and *floating point*.

Fixed point

In this notation, we assign a fixed size to the integer part and the fractional part of the number, in other words, a fixed place to the comma.

The advantage is that the process to perform basic operations is the same that integers numbers. However, this method does not take advantage of the capacity of representation format used. For instance, a computer with 8 bits to represent numbers, could use 5 bits for integer part and 3 for

3 There is another version of this method with $K = 2^{n-1} - 1$

fractional part $b_7b_6b_5b_4b_3b_2b_1b_0$.

In this case the maximum number to represent will be 01111,111 and the minimum (positive) 00000,001. If the decimal comma would be in a floating position, the range of positive numbers to represent would be 011111111 – 0,0000001

Floating point

The range of numbers that can be represented in the fixed point format is insufficient for many applications, particularly for scientific applications which often use very large and very small numbers. To represent a wide range of numbers using relatively few digits, is well known in the decimal system, the scientific representation or exponential notation. For example, 0,00000025 = $2,5 \cdot 10^{-7}$. in general, for any numbering system, a real number can be expressed as:

$$N = M \cdot B^E \text{ or } N = (M;B;E)$$

where:

M: mantissa

B: base

E: exponent

The internal representation that makes this format on computers is known as floating point.

For instance in decimal (B=10) $259,75_{(10)} = 0,25975 \cdot 10^3$ or (0,25975;10;3) or, in binary code (B=2)

$$259,75_{(10)} \rightarrow 10000011,11_{(2)} \rightarrow 0,1000001111 \cdot 2^9_{(2)} \rightarrow 0,1000001111 \cdot 2^{1001}_{(2)} \rightarrow (0,1000001111;1001)$$

The representable numbers range for a given value of B, is fixed by the number of bits of the exponent E, while the accuracy is determined by the number of bits of M.

Normalization

The same real value can be represented in infinite ways by exponential notation. For example, 2,5 can be represented as $0,25 \cdot 10^1$, $0,025 \cdot 10^2$, $250 \cdot 10^{-2}$,... To avoid confusion, we should choose one of these formats as the standard for floating point representation of a real number. The form chosen is called *Normalized form* and it is one that maintains the highest accuracy in the representation of numbers. This is achieved when the binary point is located immediately to the left of the first significant digit, so that no space is wasted representing no significant digits.

For instance:

- 2,5 represented in normalized form is $0,25 \cdot 10^1$

- $(0,000011101 ; 2 ; 0111) \rightarrow (0,11101 ; 0011) \xrightarrow{\text{Exponent excess-k}} (0,11101 ; 1011)$

 In general, to represent negative exponents the *excess-k* method is used. In the other hand, for represent negative *mantissas* signed magnitude method

- $(100,11110 ; 2 ; 0010) \rightarrow (0,10011110 ; 2 ; 0101) \xrightarrow{\text{Exponent excess-k}} (0,10011110 ; 1101)$

- $(101,001 ; 2 ; 0100) \rightarrow (0,1010010 ; 2 ; 0111) \xrightarrow{\text{Exponent excess-k}} (0,10011110 ; 1111)$

4 We remove the base value because we assume that the system base is 2

IEEE754

The most popular format for representing floating points in binary was developed by the *Institute of Electrical and Electronics Engineers* (IEEE) and it is called the IEEE754. This format can represent special cases such as infinite values and undefined results, *NaN* or *Not a Number* results. It proposes 3 formats:

Half precision. It uses 16 bits



It uses 16 bits: One bit for the sign, 5 for the exponent, 10 for the mantissa

Simple precision.

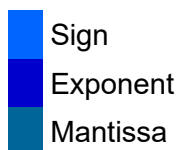



It uses 32 bits: One bit for the sign, 8 for the exponent, 23 for the mantissa

Double precision.




It uses 64 bits: One bit for the sign, 11 for the exponent, 52 for the mantissa




 All three formats use normalized mantissa, so the first bit on the left hand on the mantissa will be a 1 (the first significant digit have to be a 1). Because of this, three formats do not encode this 1 in the mantissa, although it is taken into account when operating with the number. In other words, in these formats the MSB is on the left of the decimal comma and they only save the bits on right side.

To represent the exponent the standard uses the Excess-K method con $K = 2^{n-1} - 1$

 To better understand this operation is highly recommended to watch the pill 02 [Convert real number to binary code URL](#)

2.1.7 Boolean algebra

Besides mathematical operations (+, -, *, /), on binary numbers can apply boolean or logical operations: and, or, xor, not...

 To better understand these operations is worth to name 1 as *true* and 0 as *false*

NOT:

It can be represented in various ways: NOT, \neg

$$\text{NOT } 0 = 1$$

$$\text{NOT } 1 = 0$$

AND:

It can be represented in various ways: AND, \wedge , $*$

$$0 \text{ AND } 0 = 0$$

$$1 \text{ AND } 0 = 0$$

$$0 \text{ AND } 1 = 0$$

$$1 \text{ AND } 1 = 1$$

In other words, the result will be *true* (1) only when both digits were *true*. As can be see, the result is the same that the multiplication.

$$\begin{array}{r} \phantom{\text{AND }} 10011010 \\ \text{AND } 01001100 \\ \hline 00001000 \end{array} \qquad \begin{array}{r} \phantom{\text{AND }} 1011 \\ \text{AND } 111101 \\ \hline 001001 \end{array}$$

OR:

It can be represented in various ways: OR, \vee

$$0 \text{ OR } 0 = 0$$

$$1 \text{ OR } 0 = 1$$

$$0 \text{ OR } 1 = 1$$

$$1 \text{ OR } 1 = 1$$

In this case, the result will be *true* as soon as one of the digits was *true*.

$$\begin{array}{r} \phantom{\text{OR }} 10011010 \\ \text{OR } 01001100 \\ \hline 11011110 \end{array} \qquad \begin{array}{r} \phantom{\text{OR }} 1011 \\ \text{OR } 111101 \\ \hline 111111 \end{array}$$

XOR:

$$0 \text{ XOR } 0 = 0$$

$$1 \text{ XOR } 0 = 1$$

$$0 \text{ XOR } 1 = 1$$

$$1 \text{ XOR } 1 = 0$$

In this case, the result will be *true* when one and only one of the digits were *true*.

$$\begin{array}{r} \phantom{\text{XOR }} 10011010 \\ \text{XOR } 01001100 \\ \hline \end{array} \qquad \begin{array}{r} \phantom{\text{XOR }} 1011 \\ \text{XOR } 111101 \\ \hline \end{array}$$

11010110

110110

2.2 Octal

Beside binary, there are two other interesting numeral systems when working on issues related to information technology: the octal and hexadecimal. This is because from them are easy to convert to binary.

The octal is a numeral system with a system base equal to 8 (symbols 0, 1, 2, 3, 4, 5, 6, 7). Its base is an exact power of binary system base $2^3=8$ or, in other words, with three binary digits (with three bits) we can represent all the octal digits.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

2.2.1 How to convert a binary number into octal

The process lies in creating groups of threes bits, starting on the right hand, and replace them for the related octal value

$$1101011_{(2)} \Rightarrow \textcolor{red}{1} \textcolor{violet}{101} \textcolor{green}{011} \Rightarrow \textcolor{red}{153}_{(8)}$$

2.2.2 How to convert an octal number into binary

The process is reversed to the previous: it is converted to binary each of the numbers of octal number

$$\textcolor{red}{7}\textcolor{blue}{4}\textcolor{green}{0}\textcolor{blue}{2}_{(8)} \Rightarrow \textcolor{red}{111} \textcolor{violet}{100} \textcolor{green}{000} \textcolor{blue}{010}_{(2)} = \textcolor{red}{111}\textcolor{violet}{100}\textcolor{green}{000}\textcolor{blue}{010}_{(2)}$$

2.3 Hexadecimal

Its system base is 16. As the number of symbols used in the system is greater than 10, 6 characters must be used, in this case from A to F. Thus, the ordered set of symbols is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2

3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

2.3.1 How to convert binary numbers into hexadecimal

The process is similar to the binary-octal process, except in this case groups are in fours.

$$1101011_{(2)} \Rightarrow \text{110} \text{ 1011} \Rightarrow \text{6B}_{(16)}$$

2.3.2 How to convert hexadecimal numbers into binary

The process is reversed to the previous: it is converted to binary each of the numbers of octal number

$$\text{7F0A}_{(16)} \Rightarrow \text{0111} \text{ 1111} \text{ 0000} \text{ 1010}_{(2)} = \text{111111100001010}_{(2)}$$

2.3.3 How to convert hexadecimal numbers into octal numbers

We convert into binary and group the bits in fours or threes, whichever is the numeral system destination

$$\text{6B}_{(16)} \Rightarrow \text{110} \text{ 1011} \Rightarrow 1101011_{(2)} \Rightarrow \text{1} \text{ 101} \text{ 011} \Rightarrow \text{153}_{(8)}$$

🔊 The conversion between octal or hexadecimal and decimal or vice versa, can be performed following the methods to convert between binary and decimal, but multiplying by power of 8 or 16 or dividing by these numbers and to get the remainders (in hexadecimal if the remainder is greater than 9 we get its related values A..F).

However, it is usually more practical to perform directly conversion into binary and then, convert into the system requested.

3. ALPHANUMERIC REPRESENTATION

3.1 Numeric and alphanumeric data.

A data is numeric if it is possible to perform mathematical operations. In contrast, a data is alphanumeric if you can NOT perform mathematical operations on it.

numeric : *how old are you?* **45**

alphanumeric: *What is your name?* **"Roberto"**

⚡ In order to clearly differentiate between the two types of data, it is common to use single or double quotes to indicate that data is alphanumeric.

It is usual to think that numeric data are numbers and alphanumeric data are only letters. But this is not correct. For instance:

What is your address? **"Avenida de las Palmeras 34"**

What is your mobile number? **"555341273"**

In the first case, **Avenida de las Palmeras 34**, is composed of letters and numbers, and in the second, **555341273**, only by numbers, but not operable (it makes no sense to add or multiply two phone numbers).

3.2 Internal representation

Alphanumeric characters to represent computers rely on tables, such that each of the table entries (each number) corresponds to an alphanumeric symbol.

Throughout the history of computing, there have been several tables that have always been characterized by the number of bits used to represent each character. One of the best examples is the ASCII table. The number of bits is 7, which left room for 128 characters ($2^7=128$)

As can be seen in the following table, each number is related with a character. For example, $73_{(10)}$ is a "I", $105_{(10)}$ is a "i" or $50_{(10)}$ is a "2". The first entries are reserved for non-printable characters, those that are not visible, such as tabulator ($9_{(10)}$) or carriage return ($15_{(10)}$).

⚡ The space is also a character: $32_{(10)}$

The problem of this table is its limited space. As you can see, it has room for all the Latin spellings used in Anglo-Saxon languages, but we can not find spellings like the ñ, ç or accented vowels. Therefore, the extended ASCII table of 8 bits (256 characters) was created. This new table can incorporate all of Latin spellings plus some graphic symbols.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	{null}	32	20	040	Space	64	40	100	0	96	60	140	96	60	140
1	1	001	SOH	{start of heading}	33	21	041	!	65	41	101	A	97	61	141	97	61	141
2	2	002	STX	{start of text}	34	22	042	"	66	42	102	B	98	62	142	98	62	142
3	3	003	ETX	{end of text}	35	23	043	#	67	43	103	C	99	63	143	99	63	143
4	4	004	EOT	{end of transmission}	36	24	044	\$	68	44	104	D	100	64	144	100	64	144
5	5	005	ENQ	{enquiry}	37	25	045	%	69	45	105	E	101	65	145	101	65	145
6	6	006	ACK	{acknowledge}	38	26	046	&	70	46	106	F	102	66	146	102	66	146
7	7	007	BEL	{bell}	39	27	047	'	71	47	107	G	103	67	147	103	67	147
8	8	010	BS	{backspace}	40	28	050	(72	48	110	H	104	68	150	104	68	150
9	9	011	TAB	{horizontal tab}	41	29	051)	73	49	111	I	105	69	151	105	69	151
10	A	012	LF	{NL line feed, new line}	42	2A	052	*	74	4A	112	J	106	70	152	106	70	152
11	B	013	VT	{vertical tab}	43	2B	053	+	75	4B	113	K	107	71	153	107	71	153
12	C	014	FF	{NP form feed, new page}	44	2C	054	,	76	4C	114	L	108	72	154	108	72	154
13	D	015	CR	{carriage return}	45	2D	055	-	77	4D	115	M	109	73	155	109	73	155
14	E	016	SO	{shift out}	46	2E	056	.	78	4E	116	N	110	74	156	110	74	156
15	F	017	SI	{shift in}	47	2F	057	/	79	4F	117	O	111	75	157	111	75	157
16	10	020	DLE	{data link escape}	48	30	060	0	80	50	120	P	112	76	158	112	76	158
17	11	021	DC1	{device control 1}	49	31	061	1	81	51	121	Q	113	77	159	113	77	159
18	12	022	DC2	{device control 2}	50	32	062	2	82	52	122	R	114	78	160	114	78	160
19	13	023	DC3	{device control 3}	51	33	063	3	83	53	123	S	115	79	161	115	79	161
20	14	024	DC4	{device control 4}	52	34	064	4	84	54	124	T	116	80	162	116	80	162
21	15	025	NAK	{negative acknowledge}	53	35	065	5	85	55	125	U	117	81	163	117	81	163
22	16	026	SYN	{synchronous idle}	54	36	066	6	86	56	126	V	118	82	164	118	82	164
23	17	027	ETB	{end of trans. block}	55	37	067	7	87	57	127	W	119	83	165	119	83	165
24	18	030	CAN	{cancel}	56	38	070	8	88	58	130	X	120	84	166	120	84	166
25	19	031	EM	{end of medium}	57	39	071	9	89	59	131	Y	121	85	167	121	85	167
26	1A	032	SUB	{substitute}	58	3A	072	:	90	5A	132	Z	122	86	168	122	86	168
27	1B	033	ESC	{escape}	59	3B	073	;	91	5B	133	[123	87	169	123	87	169
28	1C	034	FS	{file separator}	60	3C	074	<	92	5C	134	\	124	88	170	124	88	170
29	1D	035	GS	{group separator}	61	3D	075	=	93	5D	135]	125	89	171	125	89	171
30	1E	036	RS	{record separator}	62	3E	076	>	94	5E	136	^	126	90	172	126	90	172
31	1F	037	US	{unit separator}	63	3F	077	?	95	5F	137	_	127	91	173	127	91	173

Source: www.LookupTables.com

Today the ASCII tables are almost obsolete. The expansion of the Internet and globalization make necessary tables that incorporate not only Latin characters, but Chinese, Arabic, Korean, Russian, Hebrew...

4. UNIT SYSTEM

As discussed above, the bit is the smallest unit of information. Today we do not work at bit level, but in groups of bits (in the previous section we have seen that a character is encoded using 7 or 8 bits).

Because inside the computer everything is in binary code, an easy way to handle groups is to use some value that is a power of 2, being the most basic $2^3 = 8$. A group of 8 bits is named **byte**.

Today is not usual to use the byte group, but some multiple of it: *Kilobyte* (kB), *Megabyte* (MB), *Gigabyte* (GB), *Terabyte* (TB)... . In the *International System* these multiples are powers of 10 (they are based on the decimal system), but in computing powers of 2 are used. However, the trend is to use the International System, although it should be noted that the values are similar but not the same.

We use two different kinds of words to differentiate between the system units. When we talk about kilobyte we refer to decimal system and when we talk about *kibibytes* to the binary system.

The equivalences can be seen in the following table:

Name SI	SI	Binary	Name binary
Kilobyte (kB)	10^3 bytes = 1000 bytes	2^{10} bytes = 1024 bytes	Kibibyte (KiB)
Megabyte (MB)	10^6 bytes = 1000 kB	2^{20} bytes = 1024 ² bytes	Mebibyte (MiB)
Gigabyte (GB)	10^9 bytes = 1000 MB	2^{30} bytes = 1024 ³ bytes	Gibibyte (GiB)
Terabyte (TB)	10^{12} bytes = 1000 GB	2^{40} bytes = 1024 ⁴ bytes	Tebibyte (TiB)
Petabyte (PB)	10^{15} bytes = 1000 TB	2^{50} bytes = 1024 ⁵ bytes	Pebibyte (PiB)
Exabyte (EB)	10^{18} bytes = 1000 PB	2^{60} bytes = 1024 ⁶ bytes	Exbibyte (EiB)
Zetabyte (ZB)	10^{21} bytes = 1000 EB	2^{70} bytes = 1024 ⁷ bytes	Zebibyte (ZiB)



Although usually is used indifferently and, even today is more common *International System*, the values that are represented are different: 1 MB are 1,000,000 bytes (one million bytes), while 1 MiB are 1,048,576 bytes



It is important to differentiate between kB and kb. The first refers to kilobyte, while the second kilobit, 8 times less.



Although most of the names and abbreviations of multiples have capital letters, the kilo is defined with a lowercase.

5. ADDITIONAL MATERIAL

- [1] Glossary.
- [2] Videos about binary operations.
- [3] Exercises.

6. BIBLIOGRAPHY

- [1] Wikipedia. Signed Number Representations
https://en.wikipedia.org/wiki/Signed_number_representations
- [2] Wikipedia. Signed Number Representations
https://en.wikipedia.org/wiki/Signed_number_representations
- [3] Sistemas Informáticos. Isabel Mª Jimenez Cumbreiras. *Garceta*. 2012