Introduction

The information processed by a computer can be of **different types** (text, numbers, etc.) but it is always represented and manipulated by the computer in **binary form**. All information will be processed as a series of 0 and 1. The coding of information consists in establishing correspondence between the external (usual) information representation of information and its internal representation in the machine, which is a series of bits. We use binary representation because it is simple and easy to make.

Data representation is defined as the methods used to represent data in computers. In other words, it refers to the form in which data is stored and processed.

1. Binary Coding :

There are many binary coding:

1.1 Binary coding pure (natural binary code)

Binary coding pure binary is also qualified as natural binary. This coding has already been discussed in Chapter 1. Indeed, in this coding we associate with each positive integer the value which corresponds to it according to the binary number system. Thus, by having ***n bits*** we can code the values between **[0 and 2^n[.**

Example:

On 6 bits: $\mathbf{(35)_{10} = (100011)_2}$

Please note the value $\mathbf{(35)_{10}}$ is not representable on 5 bits. Indeed, remember that to code a value on ***n bits***, it must be between **[0 and 2^n[**.

In our case (***5 bits***) between ***[0 et 2^5[= [0 et 32[.***

In this code, numbers are represented in straight binary (b=2). This code is simple and easy to implement, but it has the disadvantage of changing more binary digits between two consecutive numbers.

For **example** the case for the transition from the decimal 3 to 4 for which the bits of weight 1 and 2 go from 1 to 0 and the weight bit 3 pass from 0 to 1. (The Passage from 011 to 100 implies the modification of 3 bits).

To avoid this disadvantage, the **Gray code** is the most used.

1.2 Gray code (the reflected binary code)

In Gray code two consecutive numbers differ from each other by only one bit. Gray code is not suitable for arithmetic operations but it is widely used in digital transmission systems to aid in error correction (minimizes the occurrence of errors), improves the signal's quality and consumes less power.

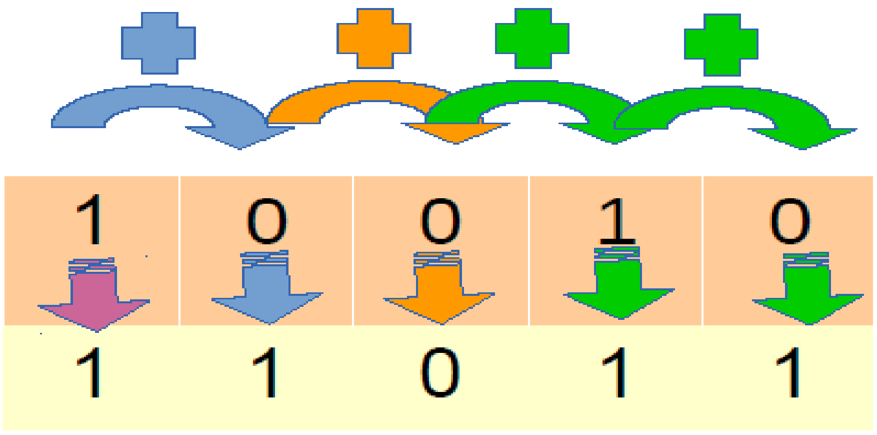
Like natural binary, Gray code can code any natural integer number.

Note: a reflected code cannot be used for arithmetic operations.

The link between a code **n** coded in Gray code and a code **N** coded in binary is as follows:

$$\mathbf{n} = \frac{\mathbf{N \oplus 2N}}{\mathbf{2}}$$

There is another method to build the **Gray code**:



The first bit on the left remains the same, then from left to right to make the sum of the adjacent bits without restraint

1. $(10010)_2 = (11011)_{Gray}$

- **Binary-to-Gray Code Conversion**

Conversion between binary code and Gray code is sometimes useful. The following rules explain how to convert from a binary number to a Gray code word:

1. The most significant bit (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

Note: the Gray code always has the same number of bits as the natural binary representation.

Example: consider a 4-digit number written *in natural binary* $(b_3b_2b_1b_0)_2$, the Gray code $(g_3g_2g_1g_0)_{gc}$ is obtained as follows :

$$\begin{tikzcd}
\text{Binary code} & b_3 \arrow[r, bend left, "+"] \arrow[d] & b_2 \arrow[r, bend left, "+"] \arrow[d] & b_1 \arrow[r, bend left, "+"] \arrow[d] & b_0 \arrow[d] \\
\text{Gray code} & g_3 & g_2 & g_1 & g_0
\end{tikzcd}$$

For example, the conversion of the binary number 10110 to Gray code is as follows:

$$\begin{tikzcd}
1 \arrow[r, "+"] \arrow[d] & 0 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 0 \arrow[d] & \text{Binary} \\
1 & 1 & 1 & 0 & 1 & \text{Gray}
\end{tikzcd}$$

The Gray code is 11101.

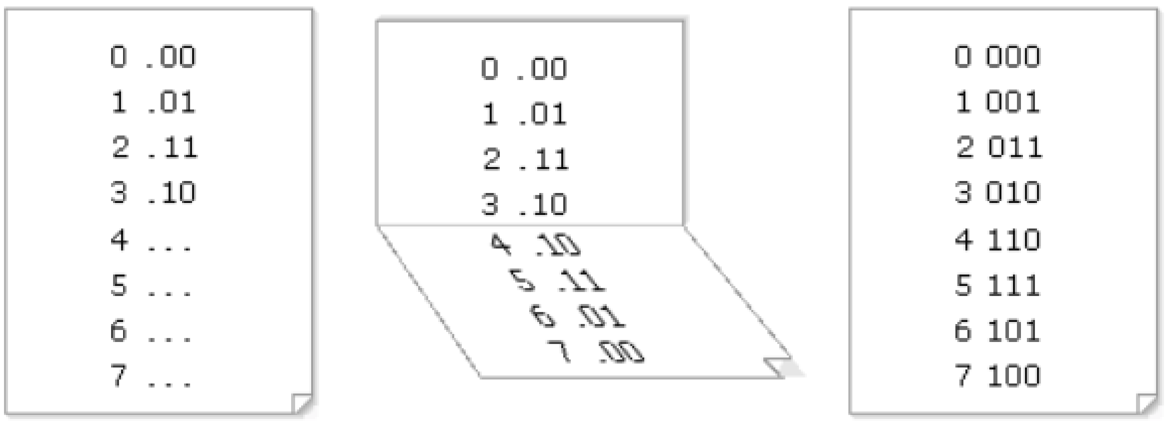
Example: convert values 35 and 36 to straight binary and Gray code

Binary	$35 = (1\,0\,0\ \ 0\,1\,1)_2$	$36 = (1\,0\,0\ \ 1\,0\,0)_2$
Gray code	**$35 = (1\,1\,0\ \ 0\,1\,0)_{gc}$**	**$36 = (1\,1\,0\ \ 1\,1\,0)_{gc}$**

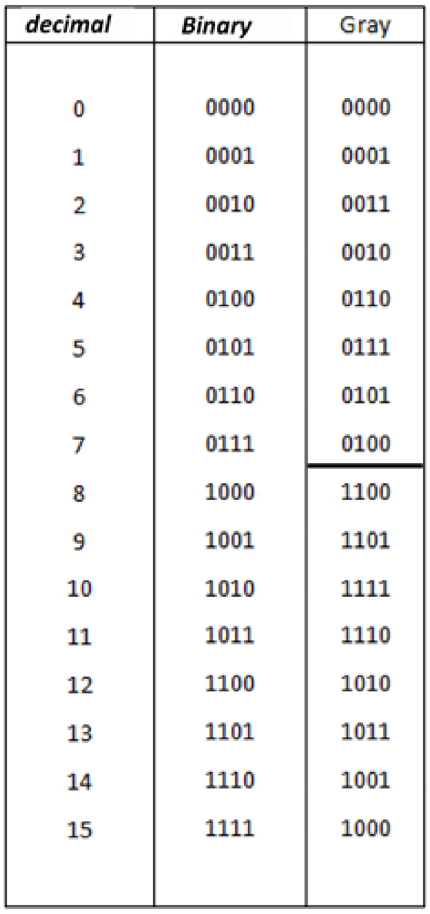
Note: the Gray code is called reflected binary, because the **n-1** bits are generated by reflection (mirror).

Where the **Gray code** can be created as follows:

1. A starting code is established: zero is coded 0 and one is coded 1.
2. Then, each time you need an additional bit, we symmetrize the numbers already obtained (like a reflection in a mirror).
3. We add a 1 at the start of the new numbers and a zero on the old ones.



Example: write numbers from 0 to 15 in natural binary and Gray code

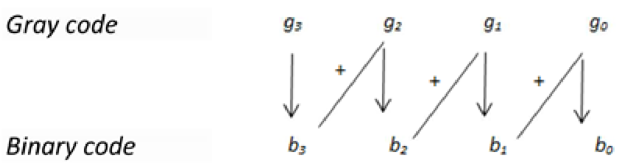


decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

- **Gray Code-to-Binary Conversion**

- The MSB remains the same as in Gray code (unchanged)

- Starting from left to right, each bit of the binary code is added to its diagonal neighbor in Gray code. The sum is carried over to the lower line which corresponds to the binary code. Carries are neglected.



For example, the conversion of the Gray code word 11011 to binary is as follows:

$$\begin{tikzcd}
1 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & \text{Gray} \\
1 \arrow[ur, "+"] & 0 \arrow[ur, "+"] & 0 \arrow[ur, "+"] & 1 \arrow[ur, "+"] & 0 & \text{Binary}
\end{tikzcd}$$

The binary number is 10010.

Example: convert the Gray $(1\ 1\ 0\ \ 0\ 1\ 0)_{gc}$ into decimal

$$(1\ 1\ 0\ \ 0\ 1\ 0)_{gc} = (1\ 0\ 0\ \ 0\ 1\ 1)_2 = \mathbf{35}$$

1.3 BCD code (Binary Coded Decimal)

To move from decimal to binary, successive divisions must be made. There are other simplified methods for the transition from decimal to binary (the BCD code, Excess 3 ...). The BCD is the most used code.

In BCD, each decimal digit is replaced by its 4-bit binary equivalent.

decimal	0	1	2	3	4	5	6	7	8	9
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Example: convert the number 275 into BCD

$2 \Rightarrow (0010)_2$

$7 \Rightarrow (0111)_2$

$5 \Rightarrow (0101)_2$

$275 = (\mathbf{0010\ 0111\ 0101})_{BCD}$

$$\begin{tikzcd}
 & 1 \arrow[dl] & 2 \arrow[d] & 9 \arrow[dr] & \\
0001 & & 0010 & & 1001
\end{tikzcd}$$

$129 = (\ 0001\ 0010\ 1001)_{BCD}$

$$\begin{tikzcd}
 & 5 \arrow[dl] & 6 \arrow[d] & 2 \arrow[dr] & \\
0101 & & 0110 & & 0010
\end{tikzcd}$$

$562 = (0101\ 0110\ 0010)_{BCD}$

Decimal	**BCD code**
0	**0000**
1	**0001**
2	**0010**
3	**0011**
4	**0100**
5	**0101**
6	**0110**
7	**0111**
8	**1000**
9	**1001**

Remark:

In the BCD code: For the addition, If there is a restraint or the result obtained not belonging to the BCD (**the result is > 9**) we must add $6 = (0110)_2$

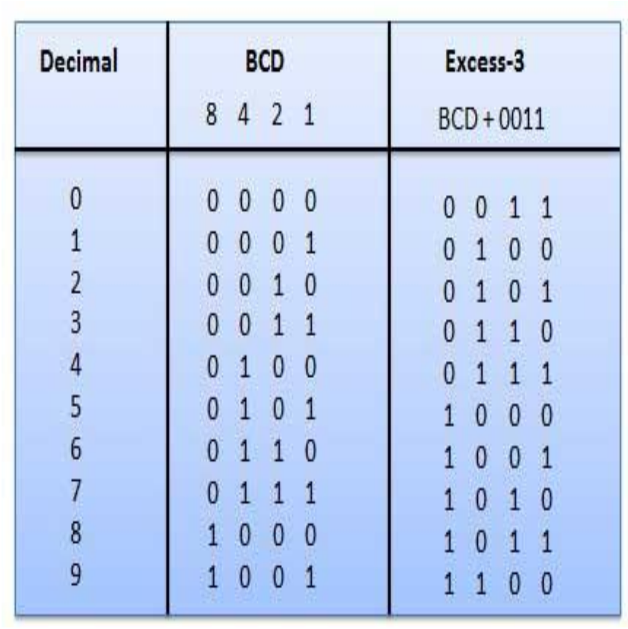
Examples:

$$\begin{array}{llrl}
 & 16 & 0001\ 0110 & \\
+ & \underline{25} & \underline{0010\ 0101} & \\
= & & 0011\ 1011 & >9 \\
 & + & \underline{0110} & +6 \\
 & & 0100\ 0001 &
\end{array}$$

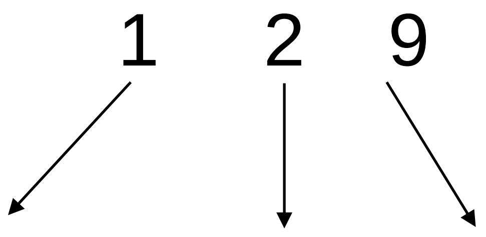
$$\begin{array}{llr}
 & 137 = & 0001\ |\ 001\overset{\textcircled{1}}{1}\ |\ 0\overset{1}{1}\overset{1}{1}\overset{1}{1} \\
+ & 99 & +\ \underline{0000\ |\ 1001\ |\ 1001} \\
 & & 0001\ |\ 1101\ |\ 0000 \\
 & & +\ \underline{0110\ |\ 0110} \\
 & & 0010\ |\ 0011\ |\ 0110 \\
 & = & 2 \qquad 3 \qquad 6
\end{array}$$

1.4 Excess3 code (BCD+3)

In Excess3, each decimal digit is coded separately in its **binary + 3** equivalent.



Decimal	BCD 8 4 2 1	Excess-3 BCD+0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0



Remark: In excess3 code 3: For the addition, if there is a restraint we must add 3 **$(+0011)_2$** if not we subtract 3 **$(-0011)_2$**

Examples:

```
            1    1
 45  =    0011 | 0111| 1000
+90     + 0011 | 1100| 0011
          0111 | 0011| 1011
         -0011 |+0011|- 0011
        = 0100 | 0110| 1000
  =        1      3     5
```

```
               1     1  1
137 =        0100 | 0110|  1010
+  90      + 0011 | 1100|  1100
             1000 | 0011|  0110
            -0011 |+0011|+ 0011
           = 0101 | 0110|  1001
  =           2      3      6
```

2. Data types :

2.1 Alphanumeric Data (Characters):

Set of characters including letters, special characters and numerals that are not used in calculations. ***E.g*** : address, name ….

2.2 Numeric Data (Numbers)

Is a data in the form of numbers that can be used in arithmetic calculations.

- Unsigned Integers (positive numbers). ***E.g*** : age, number of students in class,…
- Signed Integers. ***E.g*** : temperature,…
- Real numbers. ***E.g*** score, weight, size…..

3. Characters representation :

Though computers deal use binary to represent data, humans usually deal with information as symbolic alphabetic and numeric data. So to allow computers to handle user readable alpha/numeric data, a system to encode characters as binary numbers was created.

The set of alphanumeric characters includes letters 'a'……'z', 'A…….'Z', numerals 0…..9, and special characters +,-,*, ? #. The character representation is done by a correspondence table specific to each code used.

There are a number of standards, we can cite examples:

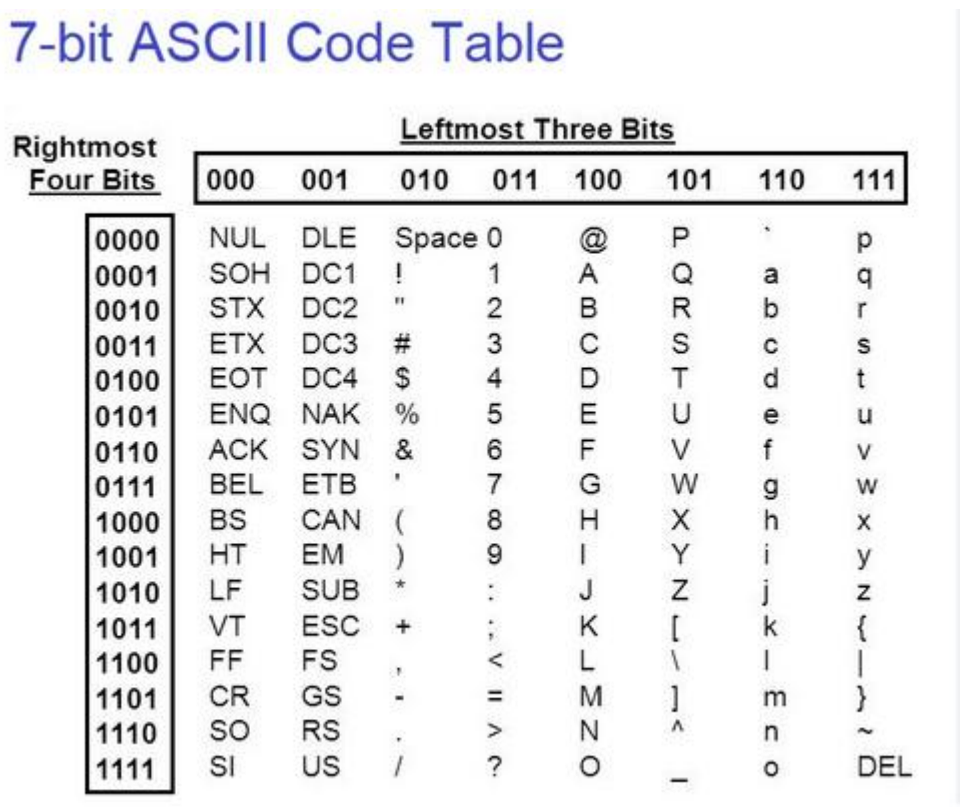
- **EBCDIC** (Extended Binary Coded Decimal Interchange Code) : 8-bit encoding, mainly used by IBM.
- **ASCII** (American Standard Code for Information Interchange): each character on 7 bits → 128 characters
- **Extended ASCII**: each character on 8 bits → 256 characters
- **Unicode :** it is the coding of most alphabets: Arabic, Chinese, Hebrew, Cyrillic ……..
  It includes several standards: UTF-8, UTF-16, UTF-32…...

Example: use the Extended ASCII table to code the following characters *‘A’ ‘?’ ‘a’*

‘A’ → *Code* $(41)_{16} = (01000001)_2$

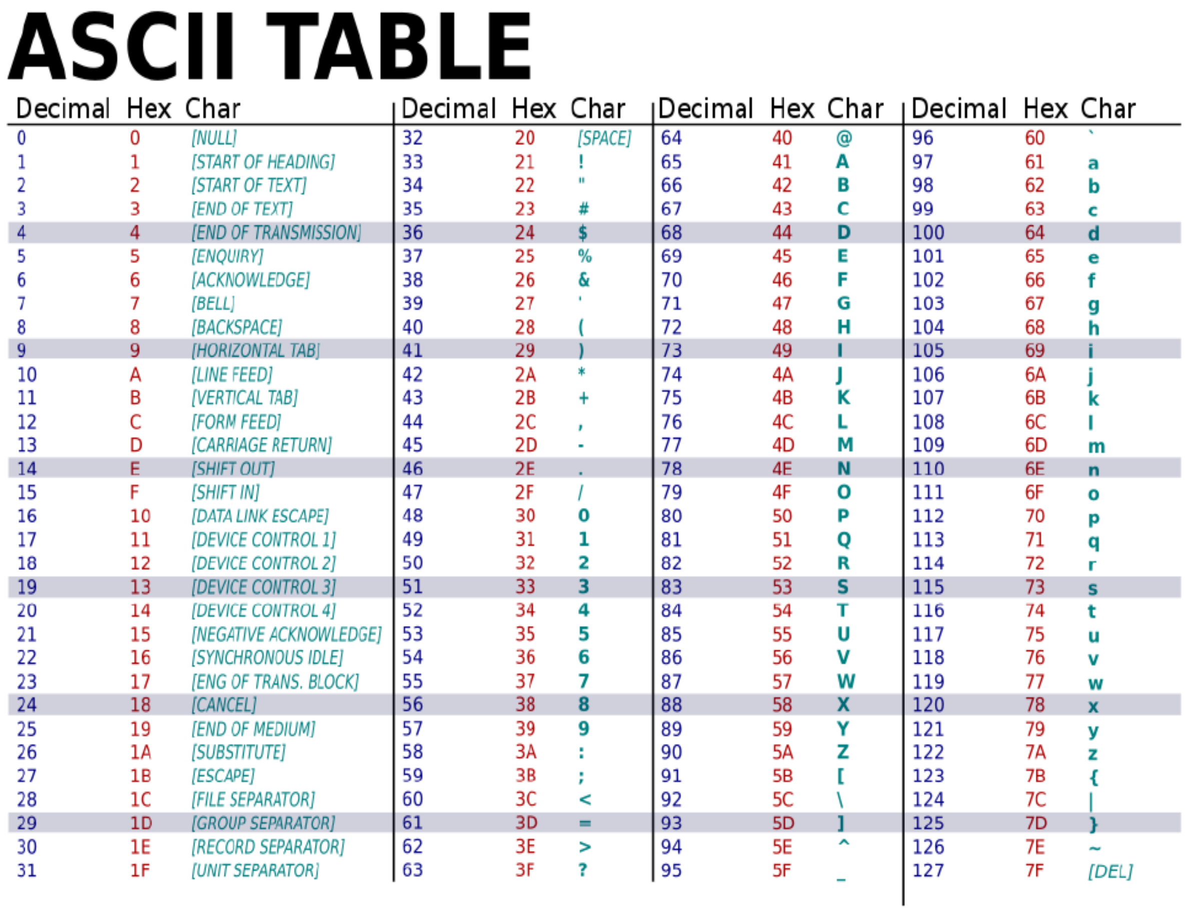
‘a’ → *Code* $(61)_{16} = (01100001)_2$

‘?’ → *Code* $(3F)_{16} = (00111111)_2$



7-bit ASCII Code Table

Rightmost Four Bits	Leftmost Three Bits 000	001	010	011	100	101	110	111
0000	NUL	DLE	Space	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

4. Integer representation :

4.1 Unsigned Integers

Unsigned integers can represent zero and positive integers. Natural binary is used to represent these numbers.

The range of unsigned integers for **n** bits register is **$[0, 2^n\text{-}1]$**

Example: write the value 35 as an 8-bits unsigned integer.

$35 = \mathbf{(00100011)_2}$

4.2 Signed Integers

Signed integers can represent zero, positive and negative integers.

Problem: how to indicate the machine that a number is negative or positive?

Solution: Three (3) representations (3 methods) had been proposed for signed integers:

1. **Sign-Magnitude representation (Sign + Absolute value)**
2. **One's Complement representation (1's Complement)**
3. **Tow's Complement representation (2's Complement)**

In all three (3) representations, *leftmost bit* (the *most-significant bit* (msb)) is called the ***sign bit***. The sign bit is used to represent the *sign* of the integer, with 0 for positive integers and 1 for negative integers.

4.2.1 Signed-Magnitude

- The most significant bit (MSB) is the sign bit, with value of 0 representing positive integer and 1 negative integer.

- The remaining **n-1** bits represent the magnitude (absolute value) of the integer.

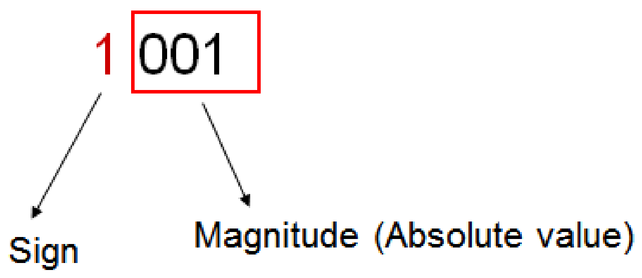
The range of numbers that can be represented by n-bit signed-magnitude is

$$\mathbf{[-(2^{n-1}-1), + (2^{n-1}-1)]}$$

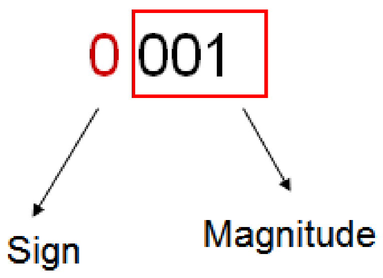
In Signed-Magnitude representation (Sign + Absolute value), if we work on ***n bits***, then:

- the *leftmost bit* (the *most-significant bit* (msb)) is used to indicate the sign:
  - **1** : negative sign
  - **0** : positive sign
- The other (*n -1*) bits represent the absolute value (magnitude) of the integer.

Example: if we work on 4 bits.

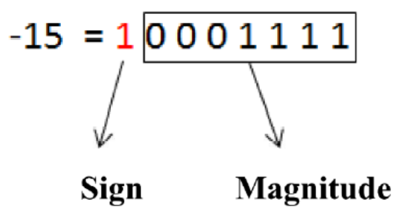


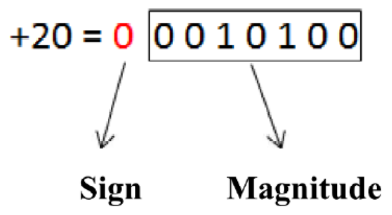
1001 is the representation of – 1



0001 is the representation of + 1

Example: convert the following values -15 and +20 into 8-bit Signed-Magnitude numbers.





- ***Conversion Signed-Magnitude →decimal***

Convert the **n-1** bits to decimal and introduce (+) if the MSB=0 or (-) otherwise.

Example: the following binary numbers are 8-bit Signed-magnitude numbers. What are the decimal values?

$(00001001)_2 = $ **+9** $(10000101)_2 = $ **-5**

If using 3 bits, (on 3 bits) we obtain:

Sign-M	Decimal
000	+ 0
001	+ 1
010	+ 2
011	+ 3
100	- 0
101	- 1
110	- 2
111	- 3

Sign	Mag	Decimal
0	00	+ 0
0	01	+ 1
0	10	+ 2
0	11	+ 3
1	00	- 0
1	01	- 1
1	10	- 2
1	11	- 3

Decimal numbers are between -3 and +3

$$-3 \leq N \leq +3$$
$$-(4-1) \leq N \leq +(4-1)$$
$$-(2^{2}-1) \leq N \leq +(2^{2}-1)$$
$$-(2^{(3-1)}-1) \leq N \leq +(2^{(3-1)}-1)$$

If we work on *n* bits, the range (interval) of the decimal numbers that we can represent in Sign-Magnitude is:

$$-(2^{(n-1)}-1) \leq N \leq +(2^{(n-1)}-1)$$

Question: perform in 4-bit Signed-Magnitude representation the following operations and give the decimal results. 5+2 5-2

```
 5+2              5-2=5+(-2)

  0101              0101
+ 0010            + 1010
------------      ------------
  0111              1111
Correct result +7   incorrect result -7
```

- **Signed-Magnitude drawbacks**

Signed-Magnitude method is very simple, but it has some drawbacks:

- There are two representations of zero (00………00) and (10………00) which could lead to inefficiency and confusion. (***E.g***. to test if a number is 0 or not, the CPU will need two tests).
- The difficulty of arithmetic operations which are complicated, because of the sign bit which must be treated separately (designing an appropriate circuit is difficult).
- The sign of both numbers have to be examined before the operation is determined (addition or subtraction).
- Two separate circuits are required to do the addition and subtraction operations (the ideal is to use a single adder-subtractor circuit that does both addition and subtraction.)

4.2.2 One's complement

- Positive numbers are obtained by conversion to natural binary.

- Negative numbers are obtained by inverting each bit of the positive opposite (0 becomes 1 and 1 becomes 0).

Example: convert the following values to 8-bit 1's complement +12 and -23

- +12= **(00001100)2**
- Positive value of -23 is +23 =$(00010111)_2$, 1's complement ➔ $\mathbf{(11101000)_2}$

Question*:* find the same values into 16-bit 1's complement

Answer:

- +12= $\mathbf{(0000000000001100)_2}$
- Positive value of -23 is +23 =$(0000000000010111)_2$,

  1's complement ➔ $\mathbf{(1111111111101000)_2}$

The range of numbers that can be represented by n-bit 1's complement is

$$[-(2^{n-1}-1), +(2^{n-1}-1)]$$

- ***Conversion 1's complement ➔ decimal***

- Determine whether the number is positive or negative (look at the MSB)
- If the number is positive, convert to decimal
- If the number is negative, complement the number, convert to decimal and introduce the sign (-)

Example:

What is the decimal value represented by the value **101010** in **1's complement** of **6** bits?

- The MSB = **1** indicates that this is a ***negative*** number. (Sign bit is 1 ⇒ negative)
- Value = $-(010101)_2 = -(21)_{10}$

Example*:* convert the following 8-bit 1's complement numbers to decimal

$(00001111)_2$ $(11110011)_2$

$(00001111)_2 = \mathbf{+15}$ $(11110011)_2 = \mathbf{-12}$

Example*:* convert the following 16-bit 1's complement numbers to decimal

$(1111111111100110)_2$ $(1111111111111111)_2$

$(1111111111100110)_2 = \mathbf{-25}$ $(1111111111111111)_2 = \mathbf{-0}$

If we work on 3 bits:

1's C	Binary	Decimal
000	000	+ 0
001	001	+ 1
010	010	+ 2
011	011	+ 3
100	-011	- 3
101	-010	- 2
110	-001	- 1
111	-000	- 0

Decimal numbers are between -3 and +3

$$-3 \leq N \leq +3$$
$$-(4-1) \leq N \leq +(4-1)$$
$$-(2^2-1) \leq N \leq +(2^2-1)$$
$$-(2^{(3-1)}-1) \leq N \leq +(2^{(3-1)}-1)$$

If we work on *n* bits, the range (interval) of the decimal numbers that we can represent in **1's complement** is:

$$-(2^{(n-1)}-1) \leq N \leq +(2^{(n-1)}-1)$$

- ***1's complement addition/ subtraction***

Addition in 1's complement is done as follows:

- o Add the two numbers.
- o If an end carry occurs, add the carry to the result.

Subtraction in 1's complement is done as follows:

- o Transform the subtraction to the addition of the 1's complement.
- o Use 1's complement addition rules.

Example: calculate the following operations using 1's complement form (4-bit)

5 + 2 -5-2 5 -2 -5 +2

5+2

0101
+ 0010

0111
= +7

(-5)+(-2)

^{1}1010
+ 1101

0111
+ 1

1000
= -7

5+(-2)

^{1}0101
+ 1101

0010
+ 1

0011
=+3

(-5)+2

1010
+ 0010

1100
=-3

Note :

The disadvantage of 1's complement method is the double representation of zero +0 and – 0.

Question: perform the following operations using 1's complement form (4-bit)

5+4 -5 -7

Answer:

5+4

^{1}0101
+ 0100

1001
= -6
incorrect result

(-5)+(-7)

^{1}1010
+ 1000

0010
+ 1

0011
= +3 incorrect

The results are incorrect because +9 and -12 are outside the range [-7, +7]

- **Overflow problem**

- The overflow occurs because the width of registers is finite.
- The overflow occurs when two numbers of **n** bits each are added and the resultoccupies **n+1** digits.
- An overflow can be detected when the sign of the result is different from the sign ofthe two numbers.
- The overflow cannot occur when the two numbers have different signs.

Note:

In unsigned integer addition, overflow occurs if there is end carry.

4.2.3 Two's complement

- Positive number is obtained by conversion to natural binary.
- The 2's complement of negative number equals its **1's complement + 1**

Example: convert the following values to 8-bit 2's complement +12 and -35

- $+12=\mathbf{(00001100)_2}$
- Positive value of -35 is $+35 = (00100011)_2$, 1's complement of -35 ➔ $(11011100)_2$

The 2's complement of $-35 = (11011100)_2 + 1 = \mathbf{(11011101)_2}$

Question*:* find the same values into 16-bit 2's complement

Answer:

- $+12=\mathbf{(0000000000001100)_2}$
- Positive value of -35 is $+35 = (0000000000100011)_2$,

  1's complement of -35 ➔ $(1111111111011100)_2$

The 2's complement of $-35 = (1111111111011100)_2 + 1 = \mathbf{(1111111111011101)_2}$

Note:

The easiest way to obtain the 2's complement of a negative number is by starting from LSB (rightmost bit) of the positive value, leaving all the 0s and the first 1 unchanged and complements all the remaining bits.

Example: convert the following values to 8-bit 2's complement -24 and -19

$+24= (00011000)$ $+19=(00010011)_2$

$-24=$ $\mathbf{(11101000)_2}$ $-19=\mathbf{(11101101)_2}$

The range of numbers that can be represented by n-bit 2's complement is:

$$\mathbf{[-2^{n-1}, +(2^{n-1}-1)]}$$

- ***Conversion 2's complement ➔decimal***

- Determine whether the number is positive or negative (look at the MSB).
- If the number is positive, convert to decimal.
- If the number is negative, flip all the bits and add 1 then convert to decimal (introduce the sign -)

Note:

Another method to obtain the decimal value of 2's complement negative number is to leave (from the LSB) all the 0s and the first 1 unchanged, flip all the remaining bits then convert to decimal (introduce the sign -)

Example: convert the following 8-bit 2's complement numbers to decimal

$(00001110)_2$ $(10001000)_2$ $(11101110)_2$

- $(00001110)_2 =$ **+14**
- $(10001000)_2 = -(01111000)_2 =$ **- 120**
- $(11101110)_2 = -(00010010)_2 =$ **-18**

Example: convert the following 16-bit 2's complement numbers to decimal

$(1111111111101111)_2$ $(1111111111111000)_2$

- $(1111111111101111)_2 = -(0000000000010001)_2 =$ **-17**
- $(1111111111111000)_2 = -(0000000000001000)_2 =$ **- 8**

If we work on 3 bits:

2's C	Binary	Decimal
000	000	+ 0
001	001	+ 1
010	010	+ 2
011	011	+ 3
100	- 100	- 4
101	- 011	- 3
110	- 010	- 2
111	- 001	- 1

Decimal numbers are between -4 and +3

$$-4 \leq N \leq +3$$
$$-(4) \leq N \leq +(4-1)$$
$$-(2^2) \leq N \leq +(2^2-1)$$
$$-(2^{(3-1)}) \leq N \leq +(2^{(3-1)}-1)$$

If we work on *n* bits, the range (interval) of the decimal numbers that we can represent in **2's complement** is:

$$-(2^{(n-1)}) \leq N \leq +(2^{(n-1)}-1)$$

- ***2's complement addition/ subtraction***

Addition / subtraction in 2's complement is similar to 1's complement with the following difference:

- o If an end carry occurs it'll be dropped rather than added to the result

Example: Calculate the following operations using 2's complement form (4-bit)

5 + 2 -5-2 5 -2 -5 +2

```
5+2              -5+(-2)            5+(-2)             -5+2

 0101               1011               0101               1011
+0010              +1110              +1110              +0010
------------       ------------       ------------       ------------
 0111               1001               0011               1101
= +7               = -7               = +3               = -3
```

Question: perform the following operations using 2's complement form (4-bit)

5+4 -5 -7

Answer:

```
   5+4                                      ( -5 )+(-7)

  ¹0101                                       10¹11
+  0100                                   +   1001
------------                              ------------
   1001                                       0100
= - 7                                     = +4    incorrect result
incorrect result
```

Overflow problem!!!

The results are incorrect because +9 and -12 are outside the range [-8,+7]

• **Advantages of 2's complement**

- The most used representation for negative numbers in computers
- One representation of zero
- One additional number **- 2^{n-1}**

Note :

In 1's complement or 2's complement, arithmetic operations are advantageous. The subtraction of a number is reduced to the addition of its complement. This allows the machine to use a single adder-subtractor circuit that does both addition and subtraction.

5. Real number representation:

A real (fractional) number has two parts: **integer part** and **fractional (decimal) part** these parts are separated by a dot (**.**) called the decimal point. The problem in real (fractional) number representation is how to indicate to the machine the position of the decimal point? There are two ways to represent real numbers:

5.1 Fixed-Point Representation

In this representation, the most significant bit represents the sign **(+/-)**, the integer part is represented on a fixed *n* bits and the fractional part on a fixed *p* bits.

Sign	Integer part	Fractional part
1bit	**n bits**	**p bits**

Example: assume a 6-bit real number with 3 bits for integer part and 2 bits for the fractional part.

- Find the representation of the following values +5.75 -0. 25 -4.125
- Calculate the range of the possible values

Answer:

- +5.75 = $(010111)_2$
  -0. 25 = $(100001)_2$
  -4.125 impossible
- The range is [-7.75, +7.75]

[-7.75, -7.50, -7.25,………………………+0.00……………………..,+7.25, +7.50, +7.75]

- **Fixed-Point advantages**

- Fixed point representation is easy to implement.
- Arithmetic operations are simple and can be performed faster (like integers).

- **Fixed-Point drawbacks**

- In fixed point representation, range of representable numbers is limited.
- Loss of precision.
- There is no standard for fixed point representation.

5.2 Floating-Point Representation

Floating point representation is similar to scientific notation. The proper format for **normalized** scientific notation of a number N is $\pm\, a \times 10^b$ where $1 \leq a < 10$ and b is the power of 10. In binary $N = \pm\, a \times 2^b$ where $1 \leq a < 2$ and b is the power of 2.

Question: express the following numbers in **normalized** scientific notation

637.8 -0.0475 89 $-(10101)_2$ $(1101.011)_2$

$(0.01101)_2$ $(33.476)_8$ $(279.DE3)_{16}$

Answer:

$637.8 = +6.378 \times 10^{+2}$ $-0.0475 = -4.75 \times 10^{-2}$ $89 = +8.9 \times 10^{+1}$

$-(10101)_2 = -(1.0101)_2 \times 2^{+4}$ $(1101.011)_2 = +(1.101011)_2 \times 2^{+3}$

$(0.01101)_2 = +(1.101)_2 \times 2^{-2}$ $(33.476)_8 = (3.3476)_8 \times 8^{+1}$

$(279.DE3)_{16} = (2.79DE3)_{16} \times 16^{+2}$

In the early days of computing, each computer constructor (brand) had its own floating point format. This had the unfortunate effect that a code that worked perfectly well on one machine could crash on another one (portability limited).

- **IEEE 754 Floating Point representation standard**

IEEE 754 is a technical standard for floating-point representation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**.

IEEE 754 has 3 basic components:

1. **The Sign**
2. **The Biased Exponent**
3. **The Mantissa**

The IEEE754 Standard defines three (3) formats for representing floating point numbers:

- **Single precision** on 32 bits
- **Double precision** on 64 bits
- **Extended precision** on 80 bits

  - **IEEE 754 single precision**

- **Normalized Form:**

The number is expressed as follows:

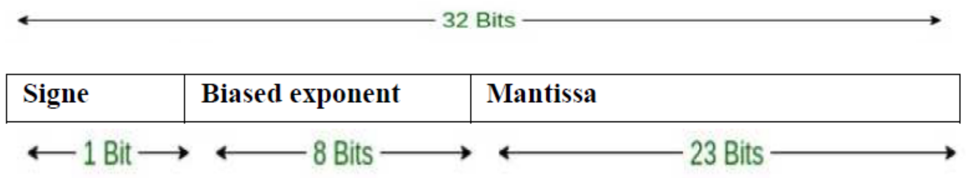
• 1 bit for sign

• 8 bits for the biased (shifted) exponent. **Biased exponent = Real exponent +127**. The exponent is shifted by $2^{8-1}–1=127$. This shift is useful because the exponent can be positive or negative.

• 23 bits for the mantissa

The following steps provide the method to convert a real number to floating point format:

1. Convert the number to Binary
2. **Normalize** the number under the form **N= (+/-)$(1.m)_2.2^{RE}$ (m: mantissa)**
3. **Calculate** the **Biased exponent, BE=RE+ bias= RE+127**
4. Store the **sign**, **BE** and the **mantissa (m)** in 32 bits



Note:
Since the first bit of a normalized binary floating point number is always 1 (always exists), we don't need to store it explicitly in the memory. This bit is called ***"hidden bit"***.

Examples: convert the following numbers to IEEE 754 single-precision.

-3.625 +65.75 -0.125

- N = - 3.625

1. Convert to binary N = - $(11.101)_2$
2. Normalize N = $-(1.1101)_2 \times 2^{+1}$ (**Mantissa (M)** = 1101)
3. Calculate the **Biased exponent , BE= RE+127**= 1+127 = 128 = $10000000_{(2)}$
4. Store S, BE and M in 32 bits

1	10000000	11010000000000000000000

- N=+65.75

1. Convert to binary $N = +(1000001.11)_2$
2. Normalize $N = +(1.00000111)_2 x2^{+6}$
3. Calculate the **Biased exponent**, **BE= RE+127**= 6+127 = 133
4. Store S, BE and M in 32 bits

$\boxed{0\ 10000101\ 00000111000000000000000}$

- N= -0.125

1. Convert to binary $N = -(0.001)_2$
2. Normalize $N = +(1.0)_2 x2^{-3}$
3. Calculate the **Biased exponent**, **BE= RE+127**= -3+127 = 124
4. Store S, BE and M in 32 bits

$\boxed{1\ 0111100\ 00000000000000000000000}$

- **IEEE 754 single precision → decimal**

To convert a number written in IEEE754 single precision to decimal:

1. Calculate the **Real exponent**, **RE= BE-127**

2. Calculate value = sign ×(1, mantissa)$_2$ × $2^{\mathbf{RE}}$, with sign = ±1

3. Convert the value to decimal (polynomial form)

Examples: convert to decimal the following IEEE 754 single-precision numbers.

$(C0980000)_{16}$ $(42484000)_{16}$

- N= $(C0980000)_{16}$

Convert to binary $N = (1\ 10000001\ 00110000000000000000000)_2$

1. Calculate the **Real exponent**, **RE= BE-127 =129-127=+2**

2. $N = -(1.0011)_2 \times 2^{+2}$

3. Convert the value to decimal $N = -(1.0011)_2 \times 2^{+2} = -(100.11)_2 = -4.75$

- $(42484000)_{16}$ = +50.0625

- **Denormalized Form**

Normalized form has a serious problem, with an implicit leading 1 for the fraction; it cannot represent the number zero! And the numbers close to zero.

Denormalized form was devised to represent zero and other numbers.

For BE=0, the numbers are in the Denormalized form. An implicit leading 0 (instead of 1) is used for the Mantissa (fraction); and the actual exponent is always -126. Hence, the number zero can be represented with E=0 and M=0 (because $0.0\times2^{-126}=0$).

We can also represent very small positive and negative numbers in de-normalized form with E=0. For example, if S=1, BE=0, and M=011 0000 0000 0000 0000 0000. The actual Mantissa (fraction) is: $0.011=1\times2^{-2}+1\times2^{-3}=0.375$. Since S=1, it is a negative number. With BE=0, the actual exponent is -126. Hence the number is -0.375×2^{-126}, which is an extremely small negative number (close to zero).

- **Special Values**

***Zero**:

SIGNE = 0/1	EXPONENT = 000…..0	MANTISSA =000…..0

***Infinity** (+infinity, -infinity): result of overflow or division by 0

SIGNE = 0/1	EXPONENT = 111…..1	MANTISSA =000…..0

***NAN (Not A Number):** undefined values such as:

$(\pm0) / (\pm0)$ $(\pm\infty) / (\pm\infty)$ $(\pm0)\times(\pm\infty)$ $0*(\pm\infty)$

$\infty - \infty$ $-\infty + \infty$ square root of a negative number

SIGNE = 0/1	EXPONENT = 111…..1	MANTISSA ≠ 000……0

In summary, the value (N) is calculated as follows:

- For $1 \le BE \le 254$, $N = (-1)^S \times 1.M \times 2^{(EB-127)}$. These numbers are in the so-called ***Normalized*** form. The sign-bit represents the sign of the number. Fractional part (1.M) is 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 $E = 0$, $N = (-1)^S \times 0.M \times 2^{(-126)}$. 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 M=0 and BE=0). It can also represent very small positive and negative number close to zero.
- For $E = 255$, it represents special values, such as ±INF (positive and negative infinity) and NaN (not a number).

Summary of the different possible representations on the IEEE 754 standard (Simple Precision)

Sign	BE	M	Value
1	11111111 = 255	$= 0$	$-\infty$
0			$+\infty$
$\forall$	11111111 = 255	$\neq 0$	NaNs
1	00000000 = 0	$= 0$	-0
0			$+0$
1	$0 < BE < 255$ Normalized	$\forall\ M$	$N = -1.M \times 2^{BE-127}$
0			$N = +1.M \times 2^{BE-127}$
1	00000000 = 0 Denormalized	$\neq 0$	$N = -0.M \times 2^{-126}$
0			$N = +0.M \times 2^{-126}$

Question: What is the largest and smallest positive number in normalized form?

Answer: We have in normalized form:

$$1 \leq BE \leq 254 \;\Rightarrow\; -126 \leq RE \leq +127$$
$$1.00\ldots0_{(2)} \leq 1.M \leq 1.11\ldots1_{(2)} \;\Rightarrow\; 1_{(10)} \leq RE \leq (2-2^{-23})_{(10)}$$

Smallest normalized positive number = $1{,}0 \times 2^{-126} = 2^{-126}$

Largest normalized positive number = $(2-2^{-23}) \times 2^{+127}$

Note:

There is a compromise between the size of the mantissa (23 bits) representing the accuracy and the size of the exponent (8 bits) representing the range.

- More digits assigned for the mantissa → higher precision and lower range
- More digits assigned for the exponent→ higher range and lower precision

Note:

The IEEE754 Standard defines three (3) formats for representing floating point numbers:

- **Single precision** on 32 bits (1bit 8bits 23 bits Bias= 127)
- **Double precision** on 64 bits (1 bit 11bits 52 bits Bias= 1023)
- **Extended precision** on 80 bits (1 bit 15 bits 64 bit Bias= 16383)

○ IEEE 754 Addition/Subtraction operations

Floating point arithmetic is more complicated than the fixed point. To calculate A+B (or A-B) we have to follow these steps:

1- Write the two numbers in normalized form

2- Align the exponents (smaller exponent aligned to the larger)

3- Add/subtract the mantissas

4- Renormalize if necessary

Example:

Assume A and B two IEEE 754 single precision numbers. Calculate A+B , A-B and B-A

A = 0 10000010 10101000000000000000000

B = 0 10000001 00100000000000000000000

- A+B

  A+B= $+(1{,}10101)_2 \text{ x}2^3 + (1{,}001)_2 \text{ x}2^2$

  $= + ((1{,}10101)_2 + (0{,}1001)_2)\text{x}2^3 = (10{,}00111)_2 \text{ x}2^3 = \mathbf{(1{,}000111)_2 \text{ x}2^4}$

  $\boxed{\text{A+B= 0 10000011 00011100000000000000000}}$

- A-B

  A-B= $+(1{,}10101)_2 \text{ x}2^3 - (1{,}001)_2 \text{ x}2^2$

  $= + ((1{,}10101)_2 - (0{,}1001)_2)\text{x}2^3 = (1{,}00011)_2 \text{ x}2^3$

  $\boxed{\text{A-B= 0 10000010 00011000000000000000000}}$

- B-A

  B-A= $(1{,}001)_2 \text{ x}2^2 - (1{,}10101)_2 \text{ x}2^3$

  $= ((0{,}1001)_2 - (1{,}10101)_2)\text{x}2^3 = - ((1{,}10101)_2 - (0{,}1001)_2)\text{x}2^3 = - (1{,}00011)_2 \text{ x}2^3$

  $\boxed{\text{B-A= 1 10000010 00011000000000000000000}}$

Note :

B-A can be obtained directly by flipping the sign of the operation A-B.