Chapter 2: Coding and Representation of data / information

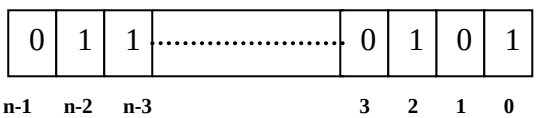
II.1. Introduction

Data and instructions cannot be entered and processed directly into computers using human language. Any type of data be it numbers, letters, special symbols, sound or pictures must first be converted into machine-readable form i.e. binary form. Due to this reason, it is important to understand how a computer together with its peripheral devices handles data. Data Representation refers to the form in which data is stored, processed, and transmitted.

II.2. Representation of the numbers in the machine

We call representation of a number the manner it is written in binary form. The representation of numbers in the computer is indispensable because it allows their storing and manipulation.

The main problem is the limitation of the size of coding because the coding of numbers is carried out in the computer using a fixed number n of bits.



II.2.1. Natural numbers

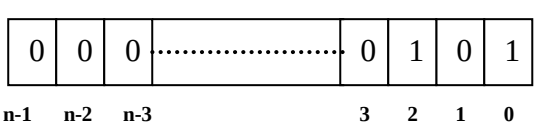
The natural numbers (without sign) are coded using fixed number of bytes. Typically, we find 1, 2, 4, bytes. A coding using n bits allows the representation of all natural numbers from **0 to $2^n - 1$**.

Example:

Using 1 byte, we can represent the numbers from **0 to $2^8 - 1 = 255$**.

We represent the number in base 2 (binary system) and we arrange the bits in the binary cells corresponding to their binary weight. If necessary, we complete on the left with zeros (most significant bits).

$(5)_{10}$=



II.2.2. The integer numbers

A signed integer is an integer with a positive '+' or negative sign '-' associated with it. Since the computer only understands binary, it is necessary to represent these signed integers in

binary form. Example: -52, +6987. There are three methods of writing and representing the integer numbers (positive and negative):

a) Signed magnitude.
b) One's (1's) complement.
c) Two's (2's) complement.

II.2.2.a. Signed magnitude representation

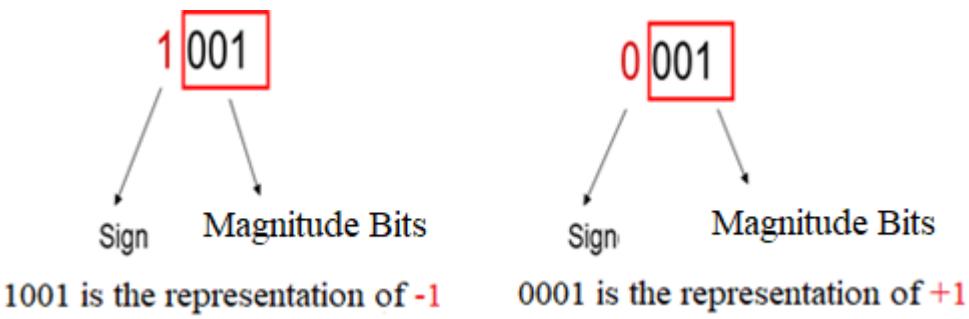
Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers. In this representation, a number consists of a magnitude and a symbol which indicates whether the magnitude is positive or negative. In the signed magnitude representation method, the following rules are followed:

- The MSB (Most Significant Bit) represents the sign of the Integer.
- Magnitude is represented by other bits other than MSB i.e. (n-1) bits where n is the no. of bits i.e the other bits (n -1) are used to represent the absolute value of the number.
- If we use n bits, then the most significant bit is used for indicate the sign:

  1: negative sign

  0: positive sign

Example: we have 4 bits to represent a signed binary number, (1-bit for the **Sign bit** and 3-bits for the **Magnitude bits**):



The following table shows the representation of the values betwing -3 and +3.

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

Sign	AV	Value
0	00	+0
0	01	+1
0	10	+2
0	11	+3
1	00	-0
1	01	-1
1	10	-2
1	11	-3

If we use n bits, the range (interval) of the values that we can represent using Signed magnitude representation is:

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

Advantages and drawbacks of signed magnitude representation are:

- It is a simple representation.
- We remark that the zero has two representations +0 and -0, which should not be the case as 0 is neither negative nor positive and which causes difficulties in the arithmetic operations.
- For the arithmetic operations, it is necessary to use two circuits: one for the addition and the second for the subtraction. The idea is to use only one circuit to perform both operation, because: a - b = a + (-b)

II.2.2.b. Representation of integer numbers using One's (1's) complement

First of all, we should define the notions of 1's complement. 1's complement of a number N is another number N' such as:

$$N+N'=2^n-1$$

$$N'=2^n-1-N$$

n : is the number of bits for representing the number N .

Example: Consider N=1010 represented in 4 bits, so its 1's complement is: $N'=(2^4-1)-N$

$$N'=(16-1)-(1010)_2=(15)-(1010)_2=(1111)_2-(1010)_2=0101$$

In 1's complement representation the following rules are used:

- For positive numbers the representation rules are the same as signed magnitude representation.
- For negative numbers, we can follow any one of the two approaches:
    1. Write the positive number in binary and take 1's complement of it. 1's complement of 0 = 1 and 1's complement of 1 = 0
    2. Write Unsigned representation of (2^n-1-X) for –X.

Simply, to find the 1's complement of a number, we toggle all the bits of this number: transforming the 0 bit to 1 and the 1 bit to 0.

N= 1010 thus N' = 0101

N= 10110 thus N' =01001

Example: Using 8 Bits

(+5) = **00000101** by toggling all the bits, we obtain 11111010, so

(-5) = **11111010**

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

Remarks:

- In this representation, the most significant bit indicates the sign (0: positive, 1: negative).
- The 1's complement of 1's complement of a number equals the same number.
- We remark that the zero has also two representations +0 and -0

The range of 1's complement integer representation of n-bit number is given as:

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

II.2.2.c. Representation of integer numbers using Two's (2's) complement

To obtain the 2's complement of a number is by calculating its 1's complement and adding one. Two's complement of a number is toggling all the bits of the number and add one. To take 2's complement simply take 1's complement and add 1 to it.

Two's complement = One's complement + 1

Example: in 8 Bits

(+5) = **00000101** toggling all the bits 11111010 then add 1 : 11111010 +1 = 11111011

(-5) = **11111011**

If we use n bits, the rage (interval) of the values that we can represent using 2's Complement is:

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

Remarks:

- In this representation, the most significant bit indicates the sign (0: positive, 1: negative).
- The advantage of this representation is the zero presents only one representation. Therefore; no ambiguity in the representation of 0.
- Numbers are in cyclic order i.e. after +7 comes -8.
- Signed Extension works.

- The range of numbers that can be represented using 2's complement is very high.
- Two's complement representation facilitates the arithmetic operations (using the same manner for positive and negative numbers).

Due to all of the above merits of 2's complement representation of a signed integer, binary numbers are generally represented using 2's complement method instead of signed bit and 1's complement.

II.2.3. Representation of the real numbers

Real numbers are numbers that include fractions/values after the decimal point. For example, 123.75 is a real number. In the decimal system, we write :

$\mathbf{(12, 346)_{10}} = 1* 10^1 + 2 *10^0 + 3* 10^{-1} + 4* 10^{-2} + 6* 10^{-}3.$

In general, in base b, we write: $(a_{n-1}...a_1a_0 \ , \ a_{-1}a_{-2}...a_{-p})_b$.

The equivalent to the decimal: $N_{10}=[(a_{n-1}*b^{n-1}+...+a_0*b^0)+(a_{-1}*b^{-1}+...+a_{-p}*b^{-p})]_{10}$

To represent the real numbers, we have two methods:

II.2.3.a. Fixed-Point

A *fixed-point* is used to represent a *real number* (one that has a fractional part) using a fixed number of digits after the radix point. The radix point is called the *decimal point* for real numbers to base ten. In binary number systems it would be called the *binary point*. Fixed-point numbers are sometimes used where the processor employed does not have a floating-point unit (FPU), which is often the case in low-cost microcontrollers.

In the fixed-point representation (the one that we humans use in everyday life), a fixed predetermined number of bits is allocated for the integer part and for the fractional part of the real number, and the radix point is assumed to lie between the two. Since we know where the radix point is assumed to lie, we don't need to store it *explicitly* in the computer, and we can treat (internally) fixed-point numbers just like integers, knowing that there is a radix point sitting at a given place in the middle of the number.

The calculations in fixed-point and integer arithmetic are entirely identical, all that we need to do is to keep track of the location of the decimal point. The computer can therefore represent internally real numbers as integers, carry out integer arithmetic on them, and simply scale the end results before outputting them.

The advantage of fixed-point representation is that **it requires no complex software or hardware to be implemented**. However; this method is not convenient to represent real numbers. With fixed-point representation we reserve a fixed number of bits on the left and on the right of the binary point. In many cases, we would need to reserve several words to represent a large range of values.

Convert a fractional decimal number to binary system

We convert the integer part and fractional part separately and then combine the results. We multiply the **fractional part** by the base "2", by repeating the operation on the fractional part of the product until it becomes zero (or until the desired precision is reached). For the **integer part**, we proceed by divisions as for an integer.

Example*:* Convert the number N= $(85.375)_{10}$ to binary number (base 2) using 16 bits: 6 bits for fractional part and 10 bits for integer part.

Integer part

The integer part of 85.375 is 85. Divide this number repeatedly by 2 until the quotient becomes 0. Write the remainders **from bottom to top**:

$$(\mathbf{85})_{10} = (\mathbf{1010101})_2$$

Fractional part

The fractional part of 85.375 is 0.375. Multiply the fractional part repeatedly by 2 until it becomes 0.

- **0.375 × 2 = 0.750**
- **0.750 × 2 = 1.500**
- **0.500 × 2 = 1.000**

From top to bottom, write the integer parts of the results to the fractional part of the number in base 2: $(\mathbf{0.375})_{10} = (\mathbf{0.011})_2$

Overall result

Combine the whole number and fractional parts to obtain the overall result.

$$N = (\mathbf{85.375})_{10} = (\mathbf{1010101})_2 + (\mathbf{0.011})_2 = (\mathbf{1010101.011})_2$$

N = $(\mathbf{85.375})_{10}$ **= 0001010101011000**

II.2.3.2. Floating point representation

Real numbers are stored in a computer as **floating-point** numbers using a **mantissa (M)**, a **base (b)** and an **exponent (E)** in this format: $\mathbf{M * b^E}$

Sign	Exponent	Mantissa

Example:

$(1.011011)_2 * 2^3$ is equivalent to $(1011.011)_2$. The mantissa is 1.011011, the radix is 2, and the exponent is 3.

The scientific notation of real numbers is: In "scientific notation" called "floating point": -.006234 is written: - 6.234 e^{-3} or - 6.234 E-3 This notation is the equivalent of: 6.234 * 10^{-3}

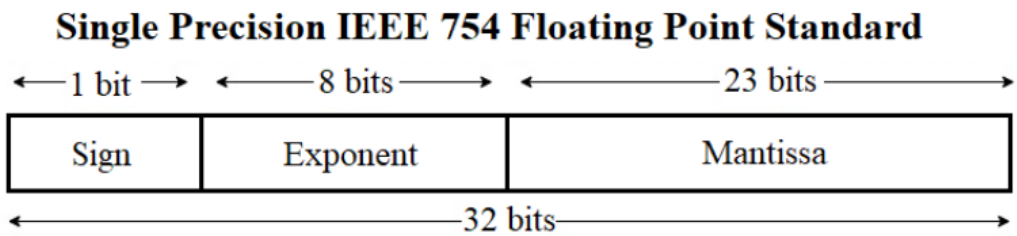
This example gives a general idea of the role of the mantissa, base and exponent. It does not fully reflect the computer's method for storing real numbers. The radix **r** is understood to be 2 and the computer doesn't need to store it explicitly.

Some examples of floating-point binary formats

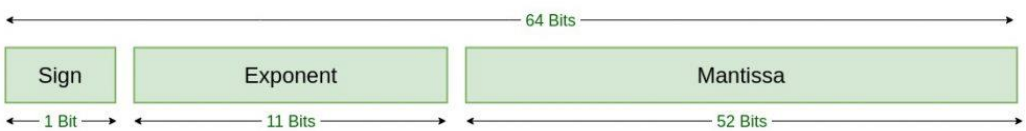
IEEE 754 has 3 basic components:

1. **The Sign of Mantissa –**This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
2. **The Biased exponent (stored exponent) –** The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
3. **The Normalized Mantissa –** The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. 0 and 1. So a normalized mantissa is one with only one 1 to the left of the binary point.

- 32-bit binary floating point formats IEEE 754 standard: used for the "float" type (single precision)



- 64-bit binary floating point formats IEEE 754 standard: used for the "float" type (double precision)



Double Precision
IEEE 754 Floating-Point Standard

The adopted format of floating-point numbers at the level of their binary representation. IEEE Standard is as follows:

$$\textbf{(Sign) 1,Mantissa} * 2^{\textbf{True Exponent}}$$

In binary scientific notation, the number before the binary point is always 1: it can therefore be omitted in the coding of floats concerning the mantissa. We are talking about a hidden 1.

$$\textbf{Stored Exponent = True Exponent + Bias}$$

$$E_s = E_t + B$$

We have **Bias = $(2^{n-1})-1$**
with **n:** the number of bits assigned to the exponent

$$\textbf{True Exponent = Stored Exponent - Bias}$$

$$E_t = E_s - B$$

Example: Represent the number N= +85.125 using single precision IEEE 754 floating-point standard

1) Sign Bit Calculation:

The given number is Positive**.** Therefore, sign = 0

2) Mantissa or Significand Bits Calculation and Normalizing:

First, Convert 85 into binary format

85 = 1010101

Second, look at the fraction part now that is **0.125**

$0.125 = (001)_2$

$85.125 = 1010101.001 = 1.010101001 \times 2^6$

Normalized mantissa = 010101001

3) Exponent Bits Calculation (Biasing the exponent):

$E_s = E_t + B = 6 + 2^{8-1}-1 = 6 + 2^7 = 6+127 = 133$

So biased exponent = $(133)_{10}$

$(133)_{10} = (10000101)_2$

4) Complete Single-Precision Format:

Hence, the Complete Single-Precision IEEE 754 Floating-Point Representation of 85.125 looks as follows:

Sign Bit+Exponent Bits+Mantissa Bits

Normalized mantissa = 010101001

we will add 0's to complete the 23 bits

Mantissa = 01010100100000000000000

The IEEE 754 Single precision is:= **0 10000101 01010100100000000000000**

This can be written in hexadecimal form $(42AA4000)_{16}$

II.3. Character representation

Characters are non-numeric data: there is no meaning in adding or multiplying two characters. On the other hand, it is often useful to compare two characters, for example to sort them in alphabetical order. The characters, called alphanumeric symbols, include upper and lower case letters, punctuation symbols (& ~, . ; # " - etc), and the numbers. A text, or character string, will be represented as a sequence of characters.

Everything in a computer is binary and can be represented as a binary value. 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 two best-known codes are EBCDIC (Extended Binary Coded Decimal Interchange Code) and ASCII (American Standard Code for Information Interchange).

In ASCII all characters are represented by a number from 1 - 127, stored in 8 bits. The ASCII encodings are shown in the following table.

Numbers as character data are also represented in ASCII. Note the number 13 is 0xD or $(1101)_2$. However; the value of the character string "13" is 0x3133 or $(001100100110011)_2$. Character numbers are represented using binary values, but are very different from their binary numbers.

ASCII is limited to just 127 characters, and is thus too limited for many applications that deal with internationalization using multiple languages and alphabets so that it does not allow the representation of accented characters (é, è, à, ù,...), and even of Chinese or Arabic characters. Representations, such as Unicode, have been developed to handle these character sets.

Several important points about ASCII code:

- Codes between 0 and 31 do not represent characters, they cannot be displayed. These codes, often called control characters, are used to indicate actions such as moving to the line

- The letters follow each other in alphabetical order (codes 65 to 90 for uppercase, 97 to 122 for lowercase), which simplifies comparisons.
- We go from upper case to lower case by modifying the 5th bit, which amounts to adding 32 to the decimal ASCII code.
- The digits are arranged in ascending order (codes 48 to 57), and the 4 least significant bits define the binary value of the digit.

There are other codes known in computing, we can cite the codes:

- BCD "Binary Coded Decimal"
- EBCDIC "Extended Binary Coded Decimal Interchange Code" a character is coded on 8 bits.
- UNICode (16 bits); ISO/IEC (32 bits): appeared in the 90s to represent all the characters of all the languages on the planet.

Character	ASCII code	Hexadecimal code	Character	ASCII code	Hexadecimal code	Character	ASCII code	Hexadecimal code
NUL	0	00	+	43	2B	V	86	56
SOH	1	01	,	44	2C	W	87	57
STX	2	02	-	45	2D	X	88	58
ETX	3	03	.	46	2E	Y	89	59
EOT	4	04	/	47	2F	Z	90	5A
ENQ	5	05	0	48	30	[	91	5B
ACK	6	06	1	49	31	\	92	5C
BEL	7	07	2	50	32	]	93	5D
BS	8	08	3	51	33	^	94	5E
TAB	9	09	4	52	34	_	95	5F
LF	10	0A	5	53	35	`	96	60
VT	11	0B	6	54	36	a	97	61
FF	12	0C	7	55	37	b	98	62
CR	13	0D	8	56	38	c	99	63
SO	14	0E	9	57	39	d	100	64
SI	15	0F	:	58	3A	e	101	65
DLE	16	10	;	59	3B	f	102	66
DC1	17	11	<	60	3C	g	103	67
DC2	18	12	=	61	3D	h	104	68
DC3	19	13	>	62	3E	i	105	69
DC4	20	14	?	63	3F	j	106	6A
NAK	21	15	@	64	40	k	107	6B
SYN	22	16	A	65	41	l	108	6C
ETB	23	17	B	66	42	m	109	6D
CAN	24	18	C	67	43	n	110	6E
EM	25	19	D	68	44	o	111	6F
SUB	26	1A	E	69	45	p	112	70
ESC	27	1B	F	70	46	q	113	71
FS	28	1C	G	71	47	r	114	72
GS	29	1D	H	72	48	s	115	73
RS	30	1E	I	73	49	t	116	74
US	31	1F	J	74	4A	u	117	75
Espace	32	20	K	75	4B	v	118	76
!	33	21	L	76	4C	w	119	77
"	34	22	M	77	4D	x	120	78
#	35	23	N	78	4E	y	121	79
$	36	24	O	79	4F	z	122	7A
%	37	25	P	80	50	{	123	7B
&	38	26	Q	81	51	|	124	7C

'	39	27	R	82	52	}	125	7D
(	40	28	S	83	53	~	126	7E
)	41	29	T	84	54	Touche de suppression	127	7F
*	42	2A	U	85	55			

II.4. Gray code

The reflected binary code (RBC), also known as reflected binary (RB) or Gray code, is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit). This property is important for several applications. The name of the code comes from the American engineer Frank Gray.

Gray codes are widely used to prevent spurious output from electromechanical switches and to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems. The use of Gray code in these devices helps simplify logic operations and reduce errors in practice.

The construction of the Gray code for numbers 0 to 15 is represented by the following table:

Decimal	Binary	Gray Code
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

II.4.1. Binary Code to Gray Code conversion method

To convert Binary into Gray, use the following method:

1. We add the binary number to another like it. The second number need to be moved one digit to the right.

2. We do a binary addition digit by digit, and we discard the carry.
3. We remove the last digit on the right side of the result on step 2 (we remove the zero, which is in red color). The resulting code is the GRAY code.

```
1100110
 1100110
---------
10101010
1010101
```

II.5. Binary-Coded Decimal (BCD)

In electronic control systems, *binary-coded decimal* (BCD) is a method for representing decimal numbers in which each decimal digit is represented by a sequence of 4 binary digits. This makes it relatively easy for the system to convert the numeric representation for printing or display purposes, and speeds up decimal calculations. The main disadvantage is that representing decimal numbers in this way takes up more space in memory than using a more conventional binary representation. The decimal digits 0-9 are each represented using four bits. The table below shows the standard BCD encodings for the decimal digits 0-9. Note that values greater than 1001 (1010, 1011, 1100, 1101, 1110, or 1111) are *not* valid BCD decimal values.

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

Example :

The BCD encoding for the number 123 would therefore be: 0001 0010 0011

As opposed to the normal binary representation: 1111011

II.6. Excess 3 code (or code of Stibitz)

shifted binary or Stibitz code (after George Stibitz, who built a relay-based adding machine in 1937) is obtained by adding 3 to each code word of the BCD code. Just like BCD, the code with excess of 3 is a decimal code, its conversion table therefore only concerns the digits from 0 to 9. Excess-3 code was used on some older computers as well as in cash registers and hand-held

portable electronic calculators of the 1970s, among other uses. In excess-3 code, numbers are represented as decimal digits, and each digit is represented by four bits as the digit value (BCD) plus 3 (the "excess" amount):

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

II.7. Binary arithmetic for signed numbers

II.7.1. Signed magnitude method

The sign is treated separately from the magnitude

Consider N_1, N_2: two binary numbers of n bits

II.7.1.1. The addition

Rule 1: N_1 and N_2 have the same sign

If ($N_1 \geq 0$) and ($N_2 \geq 0$) (The sign bit of the two numbers contains 0)

Or if ($N_1 \leq 0$) and ($N_2 \leq 0$) (The sign bit of the two numbers contains 1)

- The magnitude of N1 is added to the magnitude of N2 like the natural binary
- The sign bit of the result has the sign of the value N1 (or N2)
- Overflow: when the number of the magnitude bits of the result ≥ n bits

Rule 2 : N_1 and N_2 have different sign

If ($N_1 \geq 0$) et ($N_2 \leq 0$)

Or if ($N_1 \leq 0$) et ($N_2 \geq 0$)

- The smaller magnitude is subtracted from the greater magnitude like the natural binary
- The sign bit of the result has the sign of the greater magnitude (value)
- No overflow in this

II.7.1.2. Subtraction

- change the sign of the value you want to subtract
- the subtraction operation transforms into a simple binary addition operation [A-B = A+(-B)].
- we apply the rules of addition

II.7.1.3. Multiplication

- The magnitude of N1 is multiplied to the magnitude of N2 like the natural binary
- The result sign bit (bit_{n-1}) includes:
  - $bit_{n-1} = 1$ if ($N_1 \times N_2$) ≤ 0 (N_1 , N_2 have different signs)
  - $bit_{n-1} = 0$ if ($N_1 \times N_2$) ≥ 0 (N_1 , N_2 have same sign)
- Overflow: when the number of magnitude bits of the result ≥ n bits

II.7.2. One's complement method

Consider N_1, N_2: two binary numbers of n bits

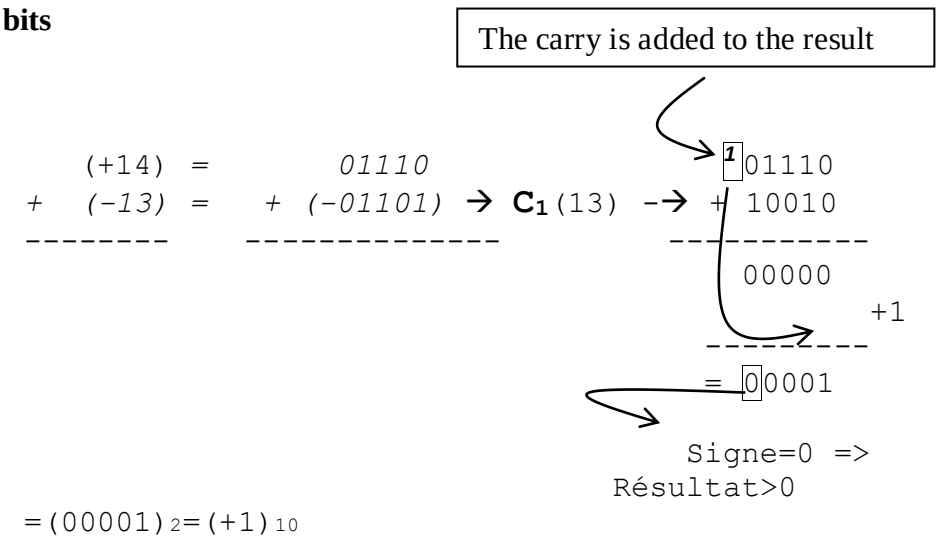
II.7.2.1. The addition

Whatever the sign of the two binary values:

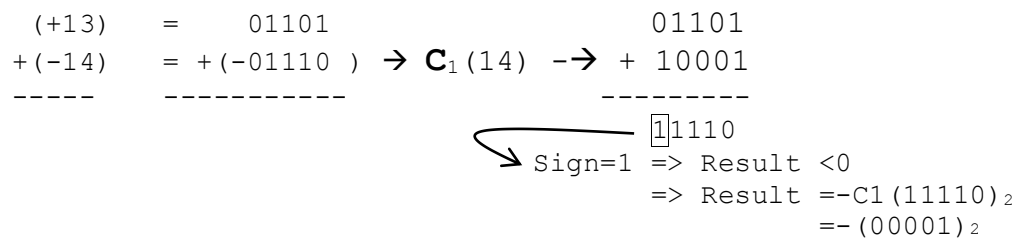
- The binary sequence of N1 is added to the binary sequence of N2 like the natural binary
- If there is a carry at the end, this is added to the least significant bit of the result

- The result of the addition can be positive or negative and must be less than 2^n, i.e. its binary representation has n bits at most
  - If the most significant bit (sign) of the result is 0, the number obtained is positive and coded in natural binary on the n bits.
  - If the most significant bit (sign) of the result is 1, the number obtained is negative and coded in one's complement to 1.
- **Overflow:** If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign i.e. When the sum of two positive numbers gives a negative number or the sum of two negative numbers gives a positive number. Overflow never occurs when adding operands with different signs.

Exemple : in 5 bits

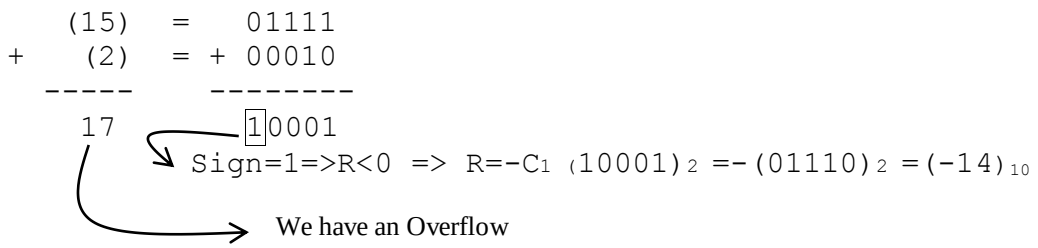


$=(00001)_2=(+1)_{10}$



$=(-1)_{10}$

Example of overflow



II.7.2.2. Subtraction

Subtrahend: what is being subtracted

Minuhend: what it is being subtracted from

- change the sign of the value of the subtrahend: value you want to subtract (Invert the bits of the second number)
- the subtraction operation transforms into a simple binary addition operation [A-B = A+(-B)].
- we apply the rules of addition

II.7.2.3. Multiplication

- Calculate the 1's complement of the negative numbers (we invert the bits of negative values)
- we apply multiplication like the natural binary
- if one and only one of the two values N1 or N2 is negative, we invert the result bits (1's complement of the result).
- Overflow: when the number of result bits ≥ n bits

II.7.2.4. Division

- we invert the bits of negative values
- we apply division like the natural binary
- we invert the result bits if one and only one of the two values N1 or N2 is negative
- No overflow

II.7.3. Two's complement method

Consider N_1, N_2: two binary numbers of n bits

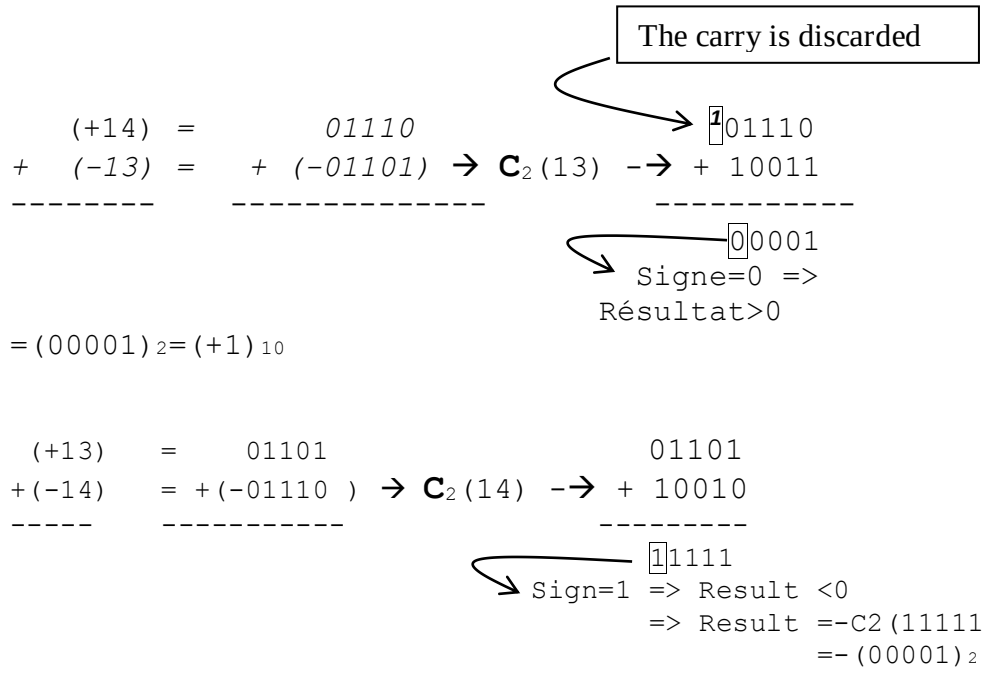
II.7.3.1. The addition

Whatever the sign of the two binary values:

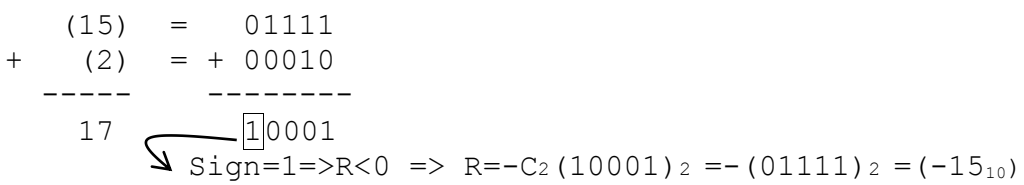
- The binary sequence of N1 is added to the binary sequence of N2 like the natural binary
- If there is a carry at the end, this should be discarded (ignored) in any cases
- The result of the addition can be positive or negative and must be less than 2^n, i.e. its binary representation has n bits at most
    - If the most significant bit (sign) of the result is 0, the number obtained is positive and coded in natural binary on the n bits.
    - If the most significant bit (sign) of the result is 1, the number obtained is negative and coded in one's complement to 2.
- **Overflow:** If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign i.e. When the sum of two positive numbers gives a negative number or the

sum of two negative numbers gives a positive number. Overflow never occurs when adding operands with different signs.

Example: in 5 bits



Example of overflow



We have an Overflow

II.7.3.2. Subtraction

- Calculate the C2 (N2) i.e. change the sign of the value you want to subtract (Invert the bits of the second number and add 1 to the least significant bit)
- the subtraction operation transforms into a simple binary addition operation [A-B = A+(-B)].
- we apply the rules of addition

II.7.3.3. Multiplication

- calculate the 2's complement of the negative numbers (we invert the bits of negative values and add 1 to the least significant bit)
- we apply multiplication like the natural binary
- if one and only one of the two values N1 or N2 is negative, then we calculate the 2's complement of the result.
- Overflow: when the number of result bits ≥ n bits

II.7.3.4. Division

- Calculate the 2's complement of the negative numbers (we invert the bits of negative values and add 1 to the least significant bit)
- we apply division like the natural binary
- if one and only one of the two values N1 or N2 is negative, then we calculate the 2's complement of the result.
- No overflow