REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieure et de la Recherche Scientifique
Université Djilali Bounaâma de Khemis Miliana
Faculté des sciences et techniques
Département de maths et informatique
Niveau : Licence 1ère année MI

Rédigé par Dr. MAHROUG RABIAA
E-mail : r.mahroug@univ-dbkm.dz

Chapitre III : La représentation de l'information

Année universitaire 2022-2023

Table des matières

Abréviation

Bit	: Binary Digit
B ou b	: Base
MSB	: Most Significant Bit
LSB	: Least Significant Bit
SVA	: Signe et Valeur Absolue
Cà1	: Complément à 1
Cà2	: Complément à 2
BCD	: Binary Coded Decimal
BCD+3	: Binary Coded Decimal+3
IEEE 754	: IEEE Standard for Floating-Point Arithmetic 754
ASCII	: Americain Standard Code Information Interchange
EBCDIC	: Extended Binary Coded Decimal Interchange Code
UTF	: (Unicode Transformation Format
F ou f	: Fonction
FND	: La première forme canonique disjonctive
FNC	: La deuxième forme canonique conjonctive

Chapitre III : La représentation de l'information

3.1. Introduction

Toute information en clair est constituée de 3 types de caractères :

-Des textes ensemble de mots eux même constitués de lettres.

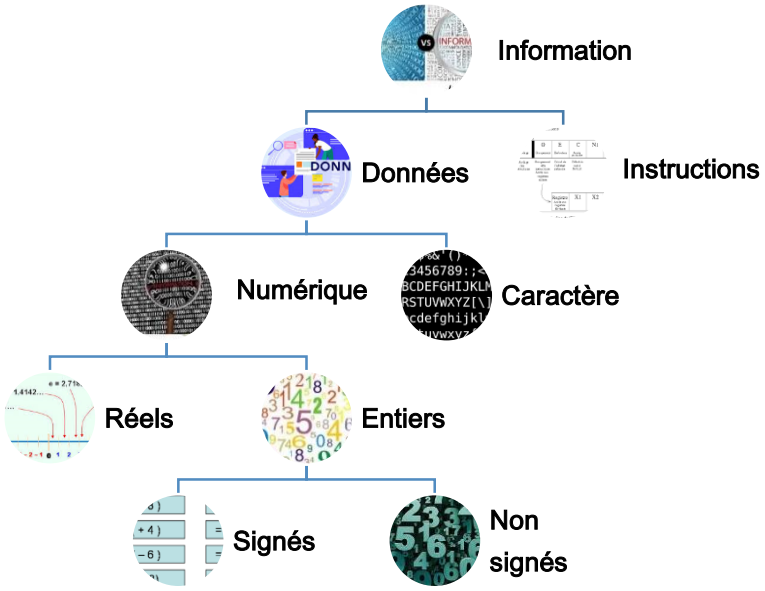
-Des nombres représentés par des chiffres.

-Des symboles qui peuvent être de nature très diverse (signes de ponctuation ; opérateurs arithmétiques opérateurs logiques ; commandes auxiliaires …)

On distingue deux catégories de codes :

-Les codes numériques qui permettent seulement le codage des nombres.

-Les codes alphanumériques qui permettent le codage d'une information quelconque (ensemble de lettre ; de chiffres et de symboles).



3.2. Codage de l'information :

Le codage de l'information permet d'établir une correspondance qui permet sans ambiguïté de passer d'une représentation (dite externe) d'une information à une autre représentation (dite interne : sous forme binaire) de la même information, suivant un ensemble de règle précise.

En informatique, Le codage de l'information s'effectue principalement en trois étapes :

- L'information sera exprimée par une suite de nombres (Numérisation)
- Chaque nombre est code sous forme binaire (suite de 0 et 1)
- Chaque élément binaire est représenté par un état physique.

Exemples :

- Charge électrique (RAM : Condensateur-transistor) : Charge (bit 1) ou non charge (bit 0).
- Magnétisation (Disque dur, disquette) : polarisation : Nord (bit 1) ou Sud (bit 0).
- Alvéoles (CDROM): réflexion (bit 1) ou pas de réflexion (bit 0).
- Fréquences (Modem) : dans un signal sinusoïdal : Fréquence f1 (bit 1) : Fréquence f2 (bit 0).

2.3. Codage des nombres

3.3.1. Codage des nombres entiers non signés

3.3.1.1. Code binaire pur (Code binaire naturel)

Ce code est utilisé uniquement pour représenter les chiffres décimaux. L'équivalent binaire d'un nombre décimal s'obtient en divisant successivement le nombre décimal par 2. Ce code est encore appelé code 8421.

N	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Table 1 : code 8421

Exemple :

$N1=(29)_{10}=(11101)_2$

$N2=(36)_{10}=(100100)_2$

$N1+N2=(36)_{10}+(29)_{10}=(65)_{10}=(1000001)_2$

3.3.1.2. Code binaire réfléchi (ou code GRAY)

Ce code ne permet que la représentation des chiffres décimaux à chaque augmentation d'une unité du chiffre décimal. Le code Gray est un code adjacent (ou un seul bit change quand on passe d'une valeur à la valeur suivante). On l'appelle aussi le code binaire réfléchi. On l'utilise dans les tableaux de Karnaugh mais aussi en conception numérique.

Première méthode :

La construction de code gray peut se faire graphiquement par itération ou bien à partir du code en binaire naturel [4]:

-On commence par 0 puis on crée un axe de symétrie,

-On ajoute le bit de poids fort en binaire naturel,

-Pour prolonger le code (passer sur N bits),

-On recrée un nouvel axe de symétrie sur les deux bits faibles,

-Puis on ajoute un bit supplémentaire en binaire naturel,

-On recommence si on veut ajouter un bit supplémentaire.

0	0	00	00	000	000	0000	0000	00000
1	1	01	01	001	001	0001	0001	00001
	1	11	11	011	011	0011	0011	00011
	0	10	10	010	010	0010	0010	00010
			10	110	110	0110	0110	00110
			11	111	111	0111	0111	00111
			01	101	101	0101	0101	00101
			00	100	100	0100	0100	00100
					100	1100	1100	01100
					101	1101	1101	01101
					111	1111	1111	01111
					110	1110	1110	01110
					010	1010	1010	01010
					011	1011	1011	01011
					001	1001	1001	01001
					000	1000	1000	01000
							1000	11000
							1001	11001
							1011	11011
							1010	11010
							1110	11110
							1111	11111
							1101	11101
							1100	11100
							0100	10100
							0101	10101
							0111	10111
							0110	10110
							0010	10010
							0011	10011
							0001	10001
							0000	10000

Deuxième méthode :

On cherche le suit, pour trouver le suit il faut suit les étapes suivantes :

1. On compte le nombre de 1
2. Si le nombre de 1 est pair on inverse le 1 suintant après le premier 1 à droit de nombre et on garde les autres bits du nombre.
3. Si le nombre de 1 est impair on inverse le premier bit faible et on garde les autres bits du nombre.

0
1
11
10
110
111
101
100
1100
1101
1111
1110
1010
1011
1001
1000

Exemple :

$(10001010001)_{Gray} \xrightarrow{\text{Suivant}} (10001010000)_{Gray}$

$(10101010111)_{Gray} \xrightarrow{\text{Suivant}} (10101010101)_{Gray}$

$(11101011010)_{Gray} \xrightarrow{\text{Suivant}} (11101011110)_{Gray}$

En comparant le code binaire naturel et le code Gray :

Décimal	Binaire	Gray
0	0	0
1	1	1
2	10	11
3	11	10
4	100	110
5	101	111
6	110	101
7	111	100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Table 2 : Comparaison entre le code binaire naturel et le code Gray

➲ Conversion binaire pur en code Gray

Pour passer du binaire au code Gray, on cite trois méthodes :

➢ Première méthode :

Dans cette première méthode, pour passer du binaire au code Gray, il faut ajouter au nombre N la valeur de N sans le bit de poids faible. Autrement dit si **$N= X_4X_3X_2X_1X_0$**

1. On garde **X_4** le bit du poids fort
2. On élimine **X_0** le bit du poids faible
3. On fait l'addition suivante : **$(X_4X_3X_2X_1X_0 + X_4X_3X_2X_1)$**
4. Le résultat trouvé est en **gray**

Remarque : On ne tient pas compte de la retenue lorsqu'on a 1+1=0 et retenue=1 à ignorer. Donc 1+1=0.

Exemple :

➦ $(101101)_2=(?)_{Gray}$

101101
+ 10110
= 111011

$(101101)_2=(111011)_{Gray}$

➦ $(11110001)_2=(?)_{Gray}$

11110001
+ 1111000
= 10001001

$(11110001)_2=(10001001)_{Gray}$

➢ Deuxième méthode :

Dans cette deuxième méthode, pour convertir un nombre binaire en nombre binaire réfléchi, nous allons observer les 3 étapes suivantes :

1. Écrire le nombre binaire pur et laisser le MSB comme tel
2. Additionner le MBS au nombre suivant en négligeant la retenue c'est-à-dire lorsqu'on a 1+1=0 et retenue=1 à ignorer. Donc 1+1=0.
3. Répéter le processus, Continuer l'addition de chaque bit avec le bit suivant jusqu'au bit du pois le plus faible.

Exemple

$(1011)_2$ ↓ $(1110)_{Gray}$

➲ $(1011)_2=(1110)_{Gray}$

$(1101011000)_2$ ↓ $(1011110100)_{Gray}$

➲ $(1101011000)_2=(1011110100)_{Gray}$

➢ **Troisième méthode :**

Il existe troisième méthode pour passer du Binaire au code Gray. Il suffit de changer le bit qui précède directement le bit 1.

➦ $(101101)_2=(\ ?)_{Gray}$ ➦ $(11110001)_2=(\ ?)_{Gray}$

$(101101)_2=(111011)_{Gray}$ $(11110001)_2=(10001001)_{Gray}$

➲ **Conversion d'un binaire réfléchi (Gray) en binaire pur**

si **N= $X_4X_3X_2X_1X_0$** , pour passer du code Gray au binaire, il faut procéder comme suite :

1. On garde **X_4** le bit du poids fort
2. On ajoute le bit du poids fort **X_4** à **X_3**
3. Le résultat trouvé dans la 2ème étape est ajouté à **X_2**
4. Le résultat trouvé dans la 3ème étape est ajouté à **X_1**
5. Le résultat trouvé dans la 4ème étape est ajouté à **X_0**

$(1011)_{Gray}$

$(1101)_2$

➲ $(1011)_{Gray}=(1101)_2$

$(11010110110)_{Gray}$

$(10011011011)_2$

➲$(11010110110)_{Gray}=(10011011011)_2$

$(111011110)_{Gray}$

$(100110111)_2$

➲$(11010110110)_{Gray}=(10011011011)_2$

3.3.1.3. Code DCB (Décimal codé binaire)

Le code BCD (**Binary Coded Decimal**) est aussi un code très utilisé qui permet de représenter des nombres sous leur écriture décimale. Il s'agit de coder les chiffres de 0 à 9 en binaire sur 4 bits.

Mais ici, il n'y a pas de valeur supérieure à 9. Le code est le suivant :

• Pour passer du décimal au binaire, il faut effectuer des divisions successives. Il existe une autre méthode simplifiée pour le passage du décimal au binaire.

• Le principe consiste à faire des éclatements sur 4 bits et de remplacer chaque chiffre décimal par sa valeur binaire correspondante sur 4bits.

• Les combinaisons supérieures à 9 sont interdites

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

Table 3 : Représentation des chiffres en BCD

On voit que c'est un code incomplet car il n'utilise que 10 combinaisons sur les 16 possibles. Ce code est utile quand on veut traiter des nombres décimaux en électronique numérique (par exemple pour dialoguer avec un être humain).

Exemples :

$(5041)_{10}=(0101\ 0000\ 0100\ 0001)_{BCD}$

$(238)_{10}=(0010\ 0011\ 1000)_{BCD}$

$(7694)_{10}=(0111\ 0110\ 1001\ 0100)_{BCD}$

Le BCD n'est pas un système de numération mais un code. Le code BCD est utilisé dans les systèmes d'affichage de chiffres décimaux.

La représentation en BCD a toutefois des inconvénients [5]. L'un des inconvénients de ce codage est qu'il ne se prête pas directement aux opérations arithmétiques : en effet, l'addition de deux valeurs dont la somme est comprise entre $(10)_{10}$ et $(15)_{10}$ donne un code binaire sans signification. Aussi, lorsque la somme de deux chiffres décimaux est supérieure à $(9)_{10}$, la machine doit effectuer un ajustement décimal, consistant à ajouter la valeur $+(6)_{10}$ à la somme des deux chiffres [6].

Addition et soustraction en BCD

Pour additionner deux nombre BCD, il faut considérer les 3 cas suivants :

- La somme ne dépasse pas 9 et la retenue est égale à 0, dans ce cas l'addition se fait comme en binaire pur et le résultat obtenu est correct.

➦ **$(8)_{10}+(1)_{10}$ en BDC**

$$\begin{array}{rcl} 8 & \xrightarrow{BCD} & 1000 \\ + & & + \\ 1 & \xrightarrow{BCD} & 0001 \\ = 9 & & = \underline{1001} \\ & & 9 \end{array}$$

➦ **$(5)_{10}+(3)_{10}$ en BDC**

$$\begin{array}{rcl} 5 & \xrightarrow{BCD} & 0101 \\ + & & + \\ 3 & \xrightarrow{BCD} & 0011 \\ = 8 & & = \underline{1000} \\ & & 8 \end{array}$$

➦ **$(33)_{10}+(52)_{10}$ en BDC**

$$\begin{array}{rcl} 33 & \xrightarrow{BCD} & 0011\ 0011 \\ + & & + \\ 52 & \xrightarrow{BCD} & 0101\ 0010 \\ = 85 & & = \underline{1000\ 0101} \\ & & 8\quad 5 \end{array}$$

➢ La somme ne dépasse pas 9 et la retenue est égale à 1, dans ce cas l'addition se fait comme en binaire pur mais le résultat obtenu est incorrect. Pour arriver au bon résultat, il faut additionner 6 (0110) au résultat provisoire.

➦ **$(7)_{10}+(9)_{10}$ en BDC**

$$\begin{array}{rcl} 7 & \xrightarrow{BCD} & 0111 \\ + & & + \\ 9 & \xrightarrow{BCD} & 1001 \\ = 16 & & = 0001\ 0000 \\ & & +\ 0110 \\ & & = \underline{0001\ 0110} \\ & & 1\quad 6 \end{array}$$

➥ **$(27)_{10}+(19)_{10}$ en BDC**

27 —BCD→ 0010 0111
\+ +
19 —BCD→ 0001 1001
= 46 = 0100 0000
+ 0110
= 0100 0110
4 6

- 7+9 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 16, Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0001 0000, il faut donc pour corriger ajouter 6 (0110), ce qui va nous donner (0001 0000) + (0110) = nous obtenons 0001 0110 qui l'équivalent de 16 en BCD.
- 27+19 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 46, Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0100 0000, il faut donc pour corriger ajouter 6 (0110) sur la partie de droite qui a la retenue de 1, ce qui va nous donner (0100 0000)+(0110) = nous obtenons 0100 0110 qui l'équivalent de 46 en BCD.

➢ La somme dépasse 9 et la retenue est égale à 0, dans ce cas l'addition se fait comme en binaire pur mais le résultat obtenu est incorrect. Pour arriver au bon résultat, il faut additionner 6 (0110) au résultat provisoire.

➥ **$(8)_{10}+(4)_{10}$ en BDC**

8 —BCD→ 1000
\+ +
4 —BCD→ 0100
= 12 = 1100
+ 0110
= 0001 0010
1 2

➥ **$(25)_{10}+(19)_{10}$ en BDC**

25 —BCD→ 0010 0101
\+ +
19 —BCD→ 0001 1001
= 44 = 0011 1110
+ 0110
= 0100 0100
4 4

- 8+4 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 12, il faut donc pour corriger ajouter 6 (0110), ce qui va nous donner 1100+0110 = 10010. Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0001 0010 qui l'équivalent de 12 en BCD.

- 25+19 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 44, il faut donc pour corriger ajouter 6 (0110) à la partie se trouvant à droite, sachant que cette dernière qui est égale à (1110) est supérieur (1001), ce qui va nous donner (0011 1110) + (0110)=(0100 0100) qui l'équivalent de 44 en BCD.

Exemple :

➦ **$(357)_{10}+(579)_{10}$ en BDC**

357 —BCD→ 0011 0101 0111
\+ \+
579 —BCD→ 0101 0111 1001
= 936 = 1000 1101 0000
+ 0110
= 1000 1101 0110
+ 0110
= 1001 0011 0110
9 3 6

-Le premier résultat est incorrect, on ajoute au groupe de droite (0110), on garde le nibble et on additionne la retenue au suivant,

-Le suivant, après addition, donne une retenue, on garde le nibble et on additionne la retenue au suivant.

Principe de la soustraction en BCD

Lors d'une soustraction en BCD ; il faut tenir compte du classement relatif de leurs valeurs absolues ; le signe final est le signe du plus grand nombre ; une correction est nécessaire lorsqu'on détecte une retenue terminale.

La correction consiste à soustraire par le nombre 6 (0110 en binaire) ; si une retenue terminale du dernier quartet apparaît, le résultat est faux parce que le classement des valeurs absolues n'a pas été respecté.

➦ **$(92)_{10}-(48)_{10}$ en BDC**

92 —BCD→ 1001 0010
\- \-
48 —BCD→ 0100 1000
= 44 = 0100 1010
- 0110
= 0100 0100
4 4

➦ $(244)_{10}$-$(89)_{10}$ en BDC

244 —BCD→ 0010 0100 0100

\- -

89 —BCD→ 0000 1000 1001

= 155 = 0001 1011 1011

-0110-0110

= 0001 0101 0101

1 5 5

3.3.1.4. Code excède de trois (Le codage EXCESS3 ou BCD+3)

Le code décimal binaire Excess3 (XS3), ou code de Stibitz, est un système numérique non pondéré, utilisé principalement par d'anciens processeurs pour la représentation des nombres en base 10. En XS3, les nombres sont représentés par 4 bits pour chaque chiffre décimal, chaque chiffre étant représenté par les quatre bits de sa représentation binaire, additionné de 3. Le code XS3 d'un nombre est donc similaire à son code BCD, à la différence que chaque groupe de quatre bits est incrémenté de 3.

Le code Excess3 est le seul code non pondéré auto réfléchi. Vous remarquerez que pour passer de 0 à 9, il suffit de remplacer les 1 par des 0 et les 0 par des 1 comme pour faire le complément de 1 en binaire pur. Pour avoir 7, il suffit de prendre 2 et chercher son complément de 1. Pour convertir un nombre décimal en XS-3 code,

décimal	BCD	BCD+3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Table 4 : Représentation des chiffres en BCD+3

Exemple :

$(238)_{10}=(0101\ 0110\ 1011)_{BCD+3}$

$(7694)_{10}=(1010\ 1001\ 1100\ 0111)_{BCD+3}$

Addition en BCD+3

Additionner deux nombre Excess XS3 Pour additionner 2 nombres XS3, il faut commencer par convertir les nombres en BCD+3, ensuite additionner. Pour additionner deux nombre BCD+3, Le résultat n'est pas correct, pour corriger il faut considérer les 3 cas suivants :

1. Il faut enlever 3 si la retenue=0
2. Il faut ajouter 3 si la retenue=1
3. Il faut ajouter 3 si on a un nouveau groupe de 4

➥ **$(6)_{10}+(2)_{10}$ en BDC+3**

6 —BCD+3→ 1001

\+ +

2 —BCD+3→ 0101

= 8 = 1110

-0011

=1011

8

Le résultat n'est pas correct, pour corriger, il faut soustraire 3 si la

➥ **$(72)_{10}+(19)_{10}$ en BDC+3**

72 —BCD+3→ 1010 0101

\+ +

19 —BCD+3→ 0100 1100

= 91 = 1 1 1 1 0001

-001 1 +0011

= 1100 0100

9 1

Le résultat n'est pas correct, pour corriger, il faut enlever 3 si la retenue=0 et il faut ajouter 3 si la retenue=1.

3.3.2. Codage des nombres entiers signés

Il y'a 3 représentations possibles pour réaliser les opérations de base (ADD ; Soust ; DIV ;Mult)

- Représentation en module et signe MS (signe et valeur absolue SVA)
- Représentation en complément à « 1 » → (Cà1)
- Représentation en complément à « 2 » → (Cà2)

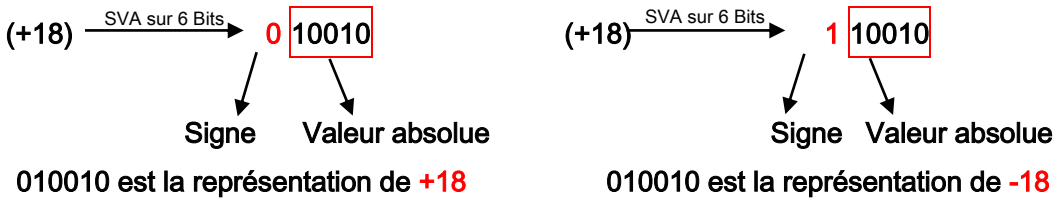
3.3.2.1. Représentation avec signe et valeur absolue

• Si on travaille sur n bits, alors le bit du poids fort est utilisé pour indiquer le signe :

$$\begin{cases} S = 0 & \text{si le nombre} > 0 \\ S = 1 & \text{si le nombre} < 0 \end{cases}$$

• Les autres bits (n -1) désignent la valeur absolue du nombre.

Exemple : Si on travaille sur 6 bits



Exemple : Représenter le nombre (-15) en signe et valeur absolue sur 8 bits

$(-15)_{10} = (-1111)_2$ SVA sur 8 Bits → | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Il est impossible de représenter le chiffre -15 sur 4 bits car sa valeur absolue $|-(15)_{10}|$ qui est égale à $(1111)_2$ prends déjà 4 bits et donc on aura besoin au minimum de 5 bits pour pouvoir représenter son bit de signe.

S	VA				Nombre en décimal
0	0	0	0	0	+0
0	0	0	0	1	+1
0	0	0	1	0	+2
0	0	0	1	1	+3
0	0	1	0	0	+4
0	0	1	0	1	+5
0	0	1	1	0	+6
0	0	1	1	1	+7
0	1	0	0	0	+8
0	1	0	0	1	+9
0	1	0	1	0	+10
0	1	0	1	1	+11
0	1	1	0	0	+12
0	1	1	0	1	+13
0	1	1	1	0	+14
0	1	1	1	1	+15

S	VA				Nombre en décimal
1	0	0	0	0	-0
1	0	0	0	1	-1
1	0	0	1	0	-2
1	0	0	1	1	-3
1	0	1	0	0	-4
1	0	1	0	1	-5
1	0	1	1	0	-6
1	0	1	1	1	-7
1	1	0	0	0	-8
1	1	0	0	1	-9
1	1	0	1	0	-10
1	1	0	1	1	-11
1	1	1	0	0	-12
1	1	1	0	1	-13
1	1	1	1	0	-14
1	1	1	1	1	-15

Tableau 5 : Représentation des nombres par la méthode Signe et Valeur Absolue

D'après le tableau 5, on peut représenter sur 5 bits, l'intervalle de nombres entiers : De $[-(2^4-1)\ ,\ (2^4-1)]$ soit de [-15 , +15].

Plus généralement, si on travaille sur n bits, l'intervalle des valeurs qu'on peut représenter en SVA est : $[-(2^{n-1} - 1),\ +(2^{n-1} - 1)]$.

Cette méthode présente deux inconvénients :

• Le zéro possède deux représentations distinctes (00000) et (10000) soit +0 et -0 [7];

• Les tables d'additions et de multiplication sont compliquées, à cause du bit de signe qui doit être traité à part.

3.3.2.2. Complément à 1 (ou Complément restreint)

Cette représentation concerne les nombres négatifs

$$A = A_n A_{n-1} A_{n-2} \cdots\cdots A_2 A_1 A_0 \quad \Rightarrow \quad (A)_{Cà1} = \overline{A}_n \overline{A}_{n-1} \overline{A}_{n-2} \cdots\cdots \overline{A}_2 \overline{A}_1 \overline{A}_0$$

• Dans cette représentation, le bit du poids fort nous indique le signe (0 : positif, 1 : négatif).

• Le complément à un du complément à un d'un nombre est égale au nombre lui-même : Cà1(Cà1(N))= N.

• La somme d'un nombre binaire et de son complément à 1 est un nombre binaire composé uniquement de 1.

•Le bit de poids fort est utilisé pour représenter le signe du nombre :

Exemple : Représenter les nombres suivants en complément à 1 sur 8 bits

(-20) $\xrightarrow{\text{SVA sur 8 Bits}}$ 1\|0010100 $\xrightarrow{\text{Cà1}}$ 1\| 1101011

(+20) $\xrightarrow{\text{SVA sur 8 Bits}}$ 0\|0010100 $\xrightarrow{\text{Cà1}}$ 0\|0010100

Nombre en Cà1					Nombre en SVA	Nombre en décimal
0	0	0	0	0	00000	+0
0	0	0	0	1	00001	+1
0	0	0	1	0	00010	+2
0	0	0	1	1	00011	+3
0	0	1	0	0	00100	+4
0	0	1	0	1	00101	+5
0	0	1	1	0	00110	+6
0	0	1	1	1	00111	+7
0	1	0	0	0	01000	+8
0	1	0	0	1	01001	+9
0	1	0	1	0	01010	+10
0	1	0	1	1	01011	+11
0	1	1	0	0	01100	+12
0	1	1	0	1	01101	+13
0	1	1	1	0	01110	+14
0	1	1	1	1	01111	+15

Nombre en Cà1					Nombre en SVA	Nombre en décimal
1	0	0	0	0	11111	-15
1	0	0	0	1	11110	-14
1	0	0	1	0	11101	-13
1	0	0	1	1	11100	-12
1	0	1	0	0	11011	-11
1	0	1	0	1	11010	-10
1	0	1	1	0	11001	-9
1	0	1	1	1	11000	-8
1	1	0	0	0	10111	-7
1	1	0	0	1	10110	-6
1	1	0	1	0	10101	-5
1	1	0	1	1	10100	-4
1	1	1	0	0	10011	-3
1	1	1	0	1	10010	-2
1	1	1	1	0	10001	-1
1	1	1	1	1	10000	-0

Tableau 6 : Représentation des nombres par la méthode Cà1

D'après le tableau 6, on peut déduire que sur 5 bits :

- Le plus grand nombre positif représentable est donc 01111 ce qui représente (2^4–1) soit (+15)
- Le plus petit négatif est 11111. Ce qui donne –(2^4-1) soit (-15).

Donc, on constate que sur 5 bits, on peut représenter les nombres qui sont dans l'intervalle [-15 ,+15], soit [-(2^4-1) , +(2^4-1)].

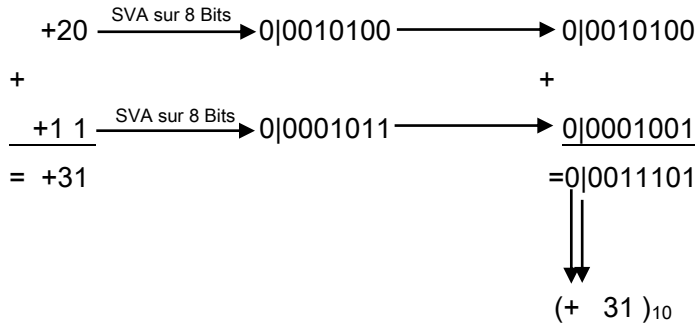
Plus généralement, si on travaille sur n bits, l'intervalle des valeurs qu'on peut représenter en Cà1 est : [-(2^{n-1} -1), +(2^{n-1} -1)].

Cette méthode présente un inconvénient :

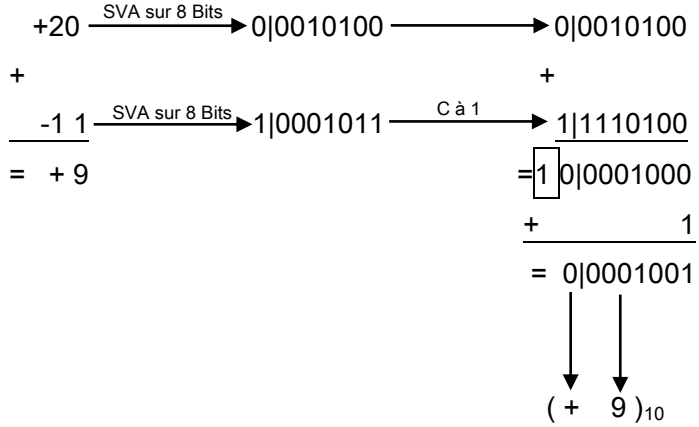
Le zéro possède deux représentations distinctes. Par exemple sur 8 bits $+(0)_{10}=(00000000)_2$, $-(0)_{10}$= Cà1(10000000)= $(11111111)_2$.

Addition en complément à 1 :

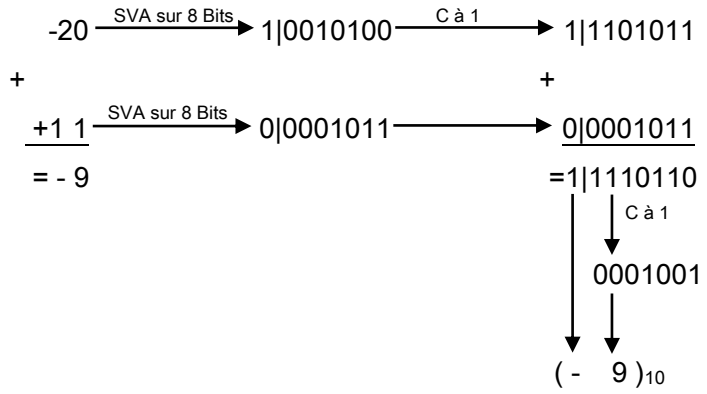
➦ **$(+20)_{10}$ + $(+11)_{10}$ en complément à 1 sur 8 bits**



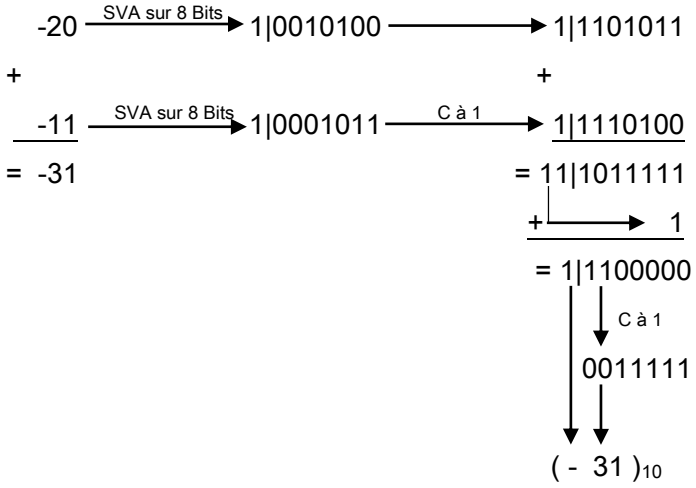
➦ **$(+20)_{10}$ + $(-11)_{10}$ en complément à 1 sur 8 bits**



➦ $(-20)_{10} + (+11)_{10}$ en complément à 1 sur 8 bits



➦ $(-20)_{10} + (-11)_{10}$ en complément à 1 sur 8 bits



3.3.2.3. Complément à 2 (ou Complément Vrai)

Le complément vrai d'un nombre négatif est le complément à « 1 » de ce nombre auquel on ajoute un « 1 » au bit de poids faible. $(A)_{Cà2} = (A)_{Cà1} + 1$. Un nombre positif est représenté de façon standard par son écriture binaire. On représente un nombre négatif en ajoutant 1 à son complément à 1 (obtenu en inversant tous les bits) et en laissant tomber une éventuelle retenue finale [8].

Exemple : Représenter les nombres suivants en complément à 1 sur 8 bits

-20 —SVA sur 8 Bits→ 1|0010100 —Cà1→ 1| 1101011 —Cà2→ 1| 1101100

+20 —SVA sur 8 Bits→ 0|0010100 —Cà1→ 0|0010100 —Cà2→ 0|0010100

Il existe quatre méthodes pour calculer le complément à 2 d'un nombre binaire.

Première méthode

La première consiste à le soustraire à la puissance de 2 immédiatement supérieure. Par exemple, le complément à 2 du nombre $N=(10001110)_2$.

```
  100000000
- 10001110
= 01110010        Donc Cà2(10001110) = 01110010
```

Deuxième méthode

Elle consiste à trouver d'abord le complément à 1 et à ajouter 1 au résultat. Par exemple :

N=$(10001110)_2$, Cà1(10001110) = 01110001

```
  01110001
+         1
= 01110010          Donc Cà2(10001110) = 01110010
```

Troisième méthode

Consiste à conserver tous les bits à partir de la droite jusqu'au premier 1 compris et de changer les autres bits de 1 à 0 ou de 0 à 1. Pour l'exemple précédent, on obtient :

Cà2(1000111**10**) = 011100**10**

bits conservés

Quatrième méthode

Consiste à conserver tous les bits nuls à partir de la droite jusqu'au premier 1 compris, trouver le Cà2 de ces bits et le Cà1 pour les autres. Pour l'exemple précédent, on obtient :

Cà2(100011 | 1 | 0) = 011100 | 1 | 0

Cà1 Cà2 bits conservés

Nombre en Cà2					Nombre en SVA	Nombre en décimal
1	0	0	0	0	10000	-16
1	0	0	0	1	11111	-15
1	0	0	1	0	11110	-14
1	0	0	1	1	11101	-13
1	0	1	0	0	11100	-12
1	0	1	0	1	11011	-11
1	0	1	1	0	11010	-10
1	0	1	1	1	11001	-9
1	1	0	0	0	11000	-8
1	1	0	0	1	10111	-7
1	1	0	1	0	10110	-6
1	1	0	1	1	10101	-5
1	1	1	0	0	10100	-4
1	1	1	0	1	10011	-3
1	1	1	1	0	10010	-2
1	1	1	1	1	10001	-1

Nombre en Cà2					Nombre en SVA	Nombre en décimal
0	0	0	0	0	00000	+0
0	0	0	0	1	00001	+1
0	0	0	1	0	00010	+2
0	0	0	1	1	00011	+3
0	0	1	0	0	00100	+4
0	0	1	0	1	00101	+5
0	0	1	1	0	00110	+6
0	0	1	1	1	00111	+7
0	1	0	0	0	01000	+8
0	1	0	0	1	01001	+9
0	1	0	1	0	01010	+10
0	1	0	1	1	01011	+11
0	1	1	0	0	01100	+12
0	1	1	0	1	01101	+13
0	1	1	1	0	01110	+14
0	1	1	1	1	01111	+15

Tableau 7 : Représentation des nombres par la méthode Cà2

Sachant que le bit du poids fort est utilisé pour représenter le signe du nombre, on peut déduire que sur 5 bits :

- Le plus grand nombre positif représentable est donc 01111 ce qui représente 2^4–1 soit +15
- Le plus petit négatif est codé par 10000, ce qui donne la valeur binaire -1000, soit -16.

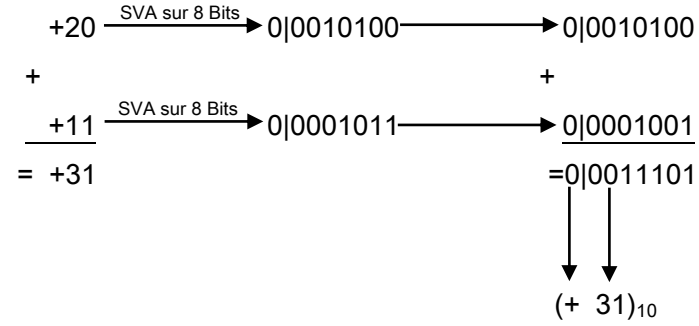
Donc, d'après le tableau 7, on constate que sur 5 bits, on peut représenter les nombres qui sont dans l'intervalle [-16, +15], soit [-2^4, $+(2^4 -1)$].

- Plus généralement, si on travaille sur n bits, l'intervalle des valeurs qu'on peut représenter en Cà2 est : [-2^{n-1}, $+(2^{n-1} - 1)$].

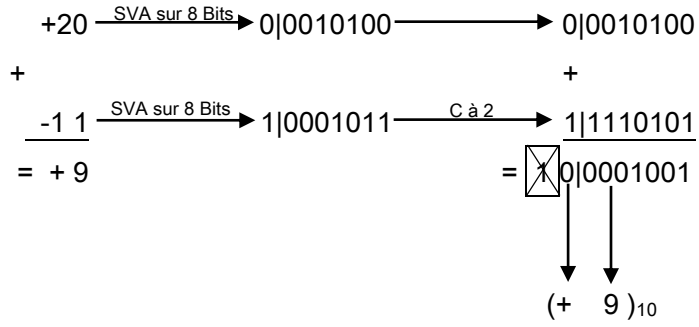
Addition en complément à 2

L'avantage de la représentation en complément à 2 est que l'on ne s'occupe pas des signes lors des opérations arithmétiques. On additionne ou l'on soustrait les nombres, bit de signe compris. Le résultat est obtenu en complément à 2 avec le signe correct.

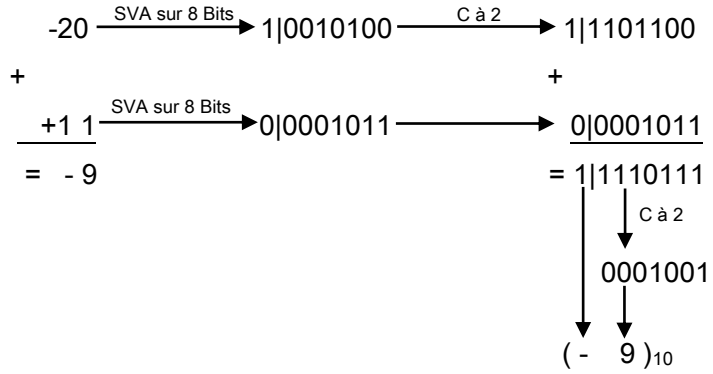
➦ **$(+20)_{10} + (+11)_{10}$ en complément à 2 sur 8 bits**



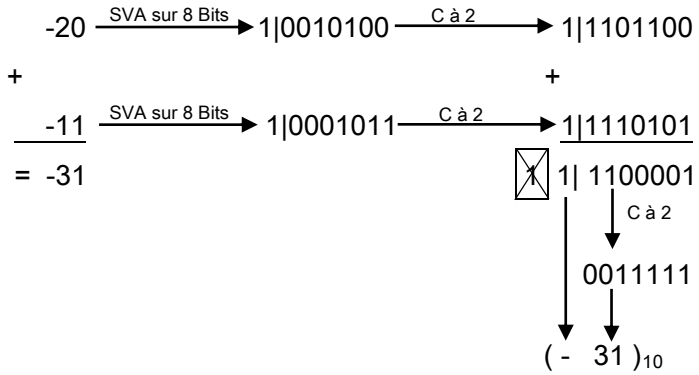
➦ **$(+20)_{10} + (-11)_{10}$ en complément à 2 sur 8 bits**



➦ **$(-20)_{10} + (+11)_{10}$ en complément à 2 sur 8 bits**



➦ $(-20)_{10} + (-11)_{10}$ en complément à 2 sur 8 bits



3.3.3. Codage des nombres fractionnaires

Les nombres fractionnaires sont ceux qui comportent des chiffres après la virgule [9]. Dans le système décimal, on écrit par exemple :

$78,25 = 7*10^{1}+8*10^{0}+2*10^{-1}+5*10^{-2}$

En général, en base b, on écrit :

$a_n a_{n-}1 \ldots a_1 a_0, a_{-1}a_{-2}\ldots..a_{-p} = a_n\, b^n + a_{n-1}b^{n-1} + \ldots. + a_0\, b^0 + a_{-1}b^{-1} + \ldots.... + a_{-p}b^{-p}$

3.3.3.1. Virgule fixe

Un nombre de m éléments binaire est partagé en deux parties dont l'une correspond aux valeurs entières et l'autre aux valeurs fractionnaires la position de la virgule fixe est fictive ; fixée par l'utilisateur Par convention :

• Si on travaille sur n bits, alors le bit du poids fort est utilisé pour indiquer le signe :

$$\begin{cases} S = 0 & \text{si le nombre} > 0 \\ S = 1 & \text{si le nombre} < 0 \end{cases}$$

• Les autres bits (n -1) désignent les valeurs entière et fractionnaire du nombre.

Soit un nombre N=-(1101010,10101)$_2$. Le codage du nombre en virgule fixe consiste à définir la position de la virgule selon un format donné, c'est-à-dire la taille de la partie entière ainsi que la taille de partie fractionnaire.

1	0	0	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	0	0
Signe	Partie entière sur 11bits											Partie fractionnaire sur 8bits						

Tableau 8 : Représentation d'un nombre réel par la méthode de la virgule fixe

La représentation des nombres à virgule fixe présente des avantages en terme de simplicité d'implémentation (elle est souvent privilégiée pour cela dans les processeurs embarqués de faible puissance) [10].

Addition et soustraction en virgule fixe

L'addition et la soustraction des nombres binaires en virgule fixe (VF) ainsi que les problèmes d'overflow sont exactement les mêmes comme pour des nombres signés en complément à 2.

+6,25	0110,01
+2,75	0011,11
=9,00	1001,00

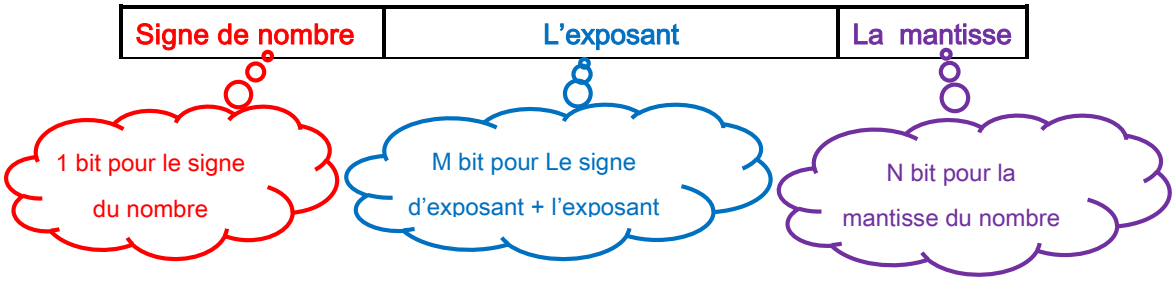
3.3.3.2. Virgule flottante

La représentation en machine des nombres réels (on parle souvent en informatique de nombres flottants) diffère de la représentation en machine des entiers.

• Chaque nombre réel peut s'écrire de la façon suivante : $N= (-1)^S*M*B^e$

- M : mantisse
- B : la base
- e : l'exposant
- S : signe

Le format de représentation des nombres flottants est comme suite :



Exemple :

$(15,6)_{10} =(-1)^0* 0,156 * 10^{+2}$

$-(110,101)_2 = (-1)^1* 0,110101 * 2^{+3}$

$(0,00101)_2=(-1)^0* 0,101* 2^{-2}$

La représentation de nombre (1101,101)2 selon le format virgule flottante donnée suivante : 1 bit de signe, 6 bits pour l'exposant plus son signe et 9bits pour la mantisse. On a :

$(1101,101)_2=(-1)^0* 0,1101101* 2^4 \Rightarrow$

0	0	0	0	1	0	0	1	1	0	1	1	0	1	0	0
Signe de nombre	Le signe d'exposant + l'exposant						La mantisse								

• Dans cette représentation sur **n** bits :

– La mantisse est sous la forme signe/valeur absolue

• 1 bit pour le signe

• et **K** bits pour la valeur.

– L'exposant (positif ou négatif) est représenté sur **P** bits.

La mantisse m est exprimée dans la base b. La précision est le nombre de chiffres de la mantisse, l'exposant est un nombre entier signé. En machine, b = 2. Un nombre décimal a une infinité de représentations mantisse–exposant [11]. La représentation en virgule flottante a été normalisée (norme IEEE 754) afin que les programmes aient un comportement identique d'une machine à l'autre [12].

- **Virgule flottante selon la norme IEEE 754**

La norme IEEE 754 (IEEE Standard for Floating-Point Arithmetic) est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique. La Représentation en virgule flottante selon la Norme IEEE -75 est un cas particulier de la représentation en virgule flottante des nombres réels [13], pour que tout le monde obtienne les mêmes résultats. Pour normaliser une représentation selon la norme IEEE-754 on décale la virgule jusqu'à avoir 1, c'est-à-dire si X est un nombre flottant:

$$X = (-1)^{S} * 1,ZZZZZ\ldots * 2^{e}$$

Où :

S :est le signe de X,

e :est l'exposant signé

ZZZZZ…: est la partie fractionnaire de la mantisse.

Il existe quatre formats : simple et double précision, et simple et double précision étendue [14]. Voici quelques exemples de formats usuels :

Nombre de bits	format	valeur max	précision max
32	1 + 8 + 23	$2128 \approx 10^{+38}$	$2^{-23} \approx 10^{-7}$
64	1 + 11 + 52	$21024 \approx 10^{+308}$	$2^{-52} \approx 10^{-15}$
80	1 + 15 + 64	$216384 \approx 10^{+4932}$	$2^{-64} \approx 10^{-19}$

- **Représentations en virgule flottante sous format IEEE754 simple précision**

Représentation simple précision est sur **32 bits** organisés comme suite :

- **1** bit pour le signe
- **8** bits pour l'exposant décalé
- **23** bits pour la mantisse

signe mantisse	Exposant décalé	partie fractionnaire de la mantisse (non signé)
1bit	8 bits	23 bits

Pour calculer l'exposant décalé on a : **Exposant Décalé = Exposant Réel + Décalage**

Décalage= $2^{n-1}- 1$

n = nombre de bit de l'exposant décalé (n= 8 pour la représentation sous format IEEE754 simple précision) , Décalage = $2^{8-1} - 1 = 2^{7} - 1 = 128 – 1 = 127$

Exemples :

- $A=(-42,75)_{10}=(-101010,11)_2$

$=(-1)^1.1,0101011.2^{+5}$

$S=1$

$M=(01010110……..0)_2$

$E-127=e \Rightarrow E=(127+5)_{10}$

$=(132)_{10}$

$=(10000100)_2$

$A=(-42,75)_{10} \xrightarrow[\text{Précision}]{\text{VF Simple}} \underbrace{1|\ 10000100|\ 010101100……0}_{32\ bits}$

- $B=(-13,125)_{10}=(-1101,001)_2$

$=(-1)^1.1,101001.2^{+3}$

$S=1$

$M=(1010010……..0)_2$

$E-127=3 \Rightarrow E=(127+3)_{10}$

$=(130)_{10}$

$=(10000010)_2$

$B=(-13,125)_{10} \xrightarrow[\text{Précision}]{\text{VF Simple}} \underbrace{1|10000010|\ 101001000……0}_{32\ bits}$

- $C=(+30,0625)_{10}=(+11110,0001)_2$

$=(-1)^0\ x1.1100001x2^{+4}$

$S=0$

$M=(111000010……0)_2$

$E-127=+4 \Rightarrow E=127+4=(131)_{10}$

$=(10000011)_2$

$C=(27,25)_{10} \xrightarrow[\text{Précision}]{\text{VF Simple}} \underbrace{0|\ 10000011\ |\ 11100001000000000000000}_{32\ bits}$

- $D=(+0,21875)_{10}=(+0,00111)_2$

$=(-1)^0\ x1.11x2^{-3}$

$S=0$

$M=(11000……0)_2$

$E-127=-3 \Rightarrow E=127-3=(124)_{10}$

$=(1111100)_2$

$D=(+0,21875)_{10} \xrightarrow[\text{Précision}]{\text{VF Simple}} \underbrace{0|01111100|\ 11000000000000000000000}_{32\ bits}$

- $E=(+0,375)_{10}=(-0,011)_2$

$=(-1)^0\ x1.1x2^{-2}$

S=0

M=(100……0)$_2$

E-127=-2 ⇒ E=127-2=(125)$_{10}$

=(01111101)$_2$

$E=(+0.375)_{10} \xrightarrow[\text{Précision}]{\text{VF Simple}} \underbrace{0|01111101|\ 10000000000000000000000}_{32\text{ bits}}$

➦ **Représentations en virgule flottante sous format IEEE754 double précision**

Représentation simple précision est sur **64 bits** organisés comme suite;

- **1** bit pour le signe
- **11** bits pour l'exposant décalé
- **52** bits pour la mantisse

signe mantisse	Exposant décalé	partie fractionnaire de la mantisse (non signé)
1bit	11 bits	52 bits

Pour calculer l'exposant décalé on a : **Exposant Décalé = Exposant Réel + Décalage**

Décalage= 2^{n-1}- 1

n=nombre de bit de l'exposant décalé (n=11 pour la représentation sous format IEEE754 simple précision), Décalage = 2^{11-1} - 1 = 2^{10} - 1 = 1024 – 1 = 1023

Exemples :

• X=(-42,75)$_{10}$ =(-101010,11)$_2$

=$(-1)^1$.1,0101011.2^{+5}

S=1

M=(01010110……..0)$_2$

E-1023=e ⇒ E=(1023+5)$_{10}$

=(1028)$_{10}$

=(10000000100)$_2$

$X= (-42,75)_{10} \xrightarrow[\text{Précision}]{\text{VF Double}} \underbrace{1|\ 10000000100|\ 010101100\ldots\ldots\ldots\ldots 0}_{64\text{ bits}}$

• Y=(+0,21875)$_{10}$=(+0,00111)$_2$

=$(-1)^0$ x1.11x2^{-3}

S=0

M=(11000……0)$_2$

E-1023=-3 ⇒ E=1023-3=(1020)$_{10}$

=(1111111100)$_2$

$Y=(+0,21875)_{10} \xrightarrow[\text{Précision}]{\text{VF Double}} \underbrace{0|01111111100|\ 11000000000000000000000}_{64\text{ bits}}$

➦ **La représentation décimale**

A/ Conversion de la représentation en virgule flottante sous format IEEE754 simple précision vers la représentation décimale

• $Z=(43C00000)_{16}=(0|10000111|10000000000000000000000)_2$

$S=0$

$M=(1000……..0)_2$

$E=(10000111)_2=(135)_{10}\Rightarrow e=E-127$

$=135-127$

$=+8$

$Z=(-1)^0x1,1x2^{+8}$

$=+1,1x2^{+8}$

$=(+11000000)_2$

$=+(2^8+2^7)_{10}$

$=(+384)_{10}$

• $W=(C29A0000)_{16}=1|10000101|00110100000000000000000$

$S=1$

$M=(001101000……0)_2$

$E=(10000101)_2=(133)_{10}\Rightarrow e=E-127$

$=133-127$

$=+6$

$W=(-1)^1x1,001101x2^{+6}$

$=(-1,001101\ x2^6)_2$

$=(-1001101)_2$

$=-(2^6+2^3+2^1+2^0)_{10}$

$=(-75)_{10}$

B/ Conversion de la représentation en virgule flottante sous format IEEE754 double précision vers la représentation décimale

• $Z=(C038\ 0000\ 0000\ 0000)_{16}=(1100\ 0000\ 0011\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000……)_2$

$S=1$

$M=(100……..0)_2$

$E=(10000000011)_2=(1027)_{10}\Rightarrow e=E-1023$

$=1027-1023$

$=+4$

$Z=(-1)^1.1,1.2^{+4}$

$=-1,1.2^{+4}=(-11000)_2=(-24)_{10}$

•$W=(BF8C\ 0000\ 0000\ 0000)_{16}=(1011\ 1111\ 1000\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ldots\ldots\ldots\ldots)_2$

S=1

$M=(1100\ldots\ldots..0)_2$

$E=(01111111000)_2=(1016)_{10}\Rightarrow e=E-1023$

$=1016-1023$

$=-7$

$W=(-1)^1x1,11x2^{-7}$

$=-1,11.2^{-7}=(-0.000000111)_2$

$=-(2^{-7}+2^{-8}+2^{-9})_{10}$

$=(-0.013671875)_{10}$

3.4. Codage des caractères

Les caractères englobent : les lettres alphabétiques (A, a, B , b,..) , les chiffres , et les autres symboles (> , ; / : ….) .

3.4.1. Code ASCII (American Standard Code for Information Interchange)

Il n'y a pas que les nombres qui doivent être codé en binaire, mais aussi les caractères comme les lettres de l'alphabet, les signes de ponctuation, … Le code le plus connu et le plus utilisé est le code ASCII (American Standard Code for Information Interchange).

Le codage ASCII affecte un code sur 7 bits aux principaux caractères mais pas aux caractères accentués [15]. L'ASCII définit seulement 128 caractères numérotés de 0 à 127 et codés en binaire de 0000000 à 1111111. Sept bits suffisent donc pour représenter un caractère codé en ASCII. Toutefois, les ordinateurs travaillant presque tous sur un multiple de huit bits (multiple d'un octet) depuis les années 1970, chaque caractère d'un texte en ASCII est souvent stocké dans un octet dont le 8ème bit est 0.

Les caractères de numéro 0 à 31 et le 127 ne sont pas affichables ; ils correspondent à des commandes de contrôle de terminal informatique. Le caractère numéro 127 est la commande pour effacer. Le caractère numéro 32 est l'espace. Les autres caractères sont les chiffres arabes, les lettres latines majuscules et minuscules sans accent, des symboles de ponctuation, des opérateurs mathématiques et quelques autres symboles. Des fois le 8ème bit est utilisé pour donner le code ASCII étendu avec la possibilité de faire 28, soit 256 symboles.

Des représentations plus évoluées comme l'Unicode qui comporte 16 bits peuvent représenter jusqu' 216, soit 65536 symboles. Avec ce code plusieurs alphabets peuvent être représentés. L'absence des caractères des langues étrangères à l'anglais rend ce standard insuffisant à lui seul pour des textes étrangers comme le français, ce qui rend nécessaire l'utilisation d'autres encodages.

		000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	10	LF	SUB	*	:	J	Z	j	z
1011	11	VT	ESC	+	;	K	[	k	{
1100	12	FF	FS	,	<	L	\	l	!
1101	13	CR	GS	-	=	M	]	m	}
1110	14	SO	RS	.	>	N	^	n	~
1111	15	SI	US	/	?	O	-	o	DEL

Table du codage ASCII

Exemple :

•Le code de caractère A =(41)$_{\text{ASCII-hexadécimal}}$

=(01000001)$_{\text{ASCII-binaire}}$

=(65)$_{\text{ASCII-décimal}}$

•Le code de caractère a =(61)$_{\text{ASCII-hexadécimal}}$

=(01000001)$_{\text{ASCII-binaire}}$

=(97)$_{\text{ASCII-décimal}}$

•Le code de caractère X =(58)$_{\text{ASCII-hexadécimal}}$

=(01011000)$_{\text{ASCII-binaire}}$

=(88)$_{\text{ASCII-décimal}}$

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	NP	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	Del

Table du codage ASCII

NUL	Absence de caractère,blanc,espace
SOH	Start of Heading : début en-tête
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquiry Demande
ACK	Acknowledge, accusé réception
BEL	Bell, sonnette
BS	Backspace marche arrière 1 caractère
HT	Horizontale Tabulation
LF	Line Fed retour à la nvelle ligne
VT	Vertical Tabulation
FF	Form Fed, passage page suivante
CR	Carriage Return, retour chariot
SO	Shift Out caractère suivant non std
SI	Shift In retour au caractères std
DLE	DataLink Escape chgmt de signific.
NAK	Negative Acknoledgment
SYN	Synchronous, caractère de synchro.
ETB	End Of Transmission Block
CAN	Cancel annulation de la donnée précédente
SUB	Substitute remplacement
ESC	Escape caractère de ctrl d'extension
FS	File Separator
GS	Groupe Separator
RS	Record Separator
US	United Separator
SP	Space Espace
DEL	Delete, suppression

DC1 à DC4 : caractères de commandes

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Table du codage ASCII

(BONJOUR)$_{caractère}$= (66 79 78 74 79 85 82)$_{ASCII-décimal}$

=(42 4F 4E 4A 4F 55 52)$_{ASCII-hexadécimal}$

=(01000010 01001111 01001110 01001010 01001111 01010101 01010010)$_{ASCII-binaire}$

(STRUCTURE Machine 1)$_{caractère}$

=(83 84 82 85 67 84 85 82 69 32 109 97 99 104 105 110 101 32 73)$_{ASCII-décimal}$

=(53 54 52 55 43 54 55 52 45 20 6D 61 63 68 69 6E 65 20 49)$_{ASCII-hexadécimal}$

=(01010010 01010100 01010010 01010101 01000011 01010100 01010101 01010010 01000101 00100000 01101101 01100001 01100011 01101000 01101001 01101110 01100101 00100000 01001001)$_{ASCII-binaire}$

3.4.2. Code EBCDIC

L'Extended Binary Coded Decimal Interchange Code (EBCDIC) est un mode de codage des caractères sur 8 bits créé par IBM à l'époque des cartes perforées. Il existe au moins 6 versions différentes bien documentées, incompatibles entre elles. Ce mode de codage a été critiqué pour cette raison, mais aussi parce que certains caractères de ponctuation ne sont pas disponibles dans certaines versions. Ces disparités ont parfois été interprétées comme un moyen pour IBM de conserver ses clients captifs.

EBCDIC est encore utilisé dans les systèmes AS/400 d'IBM ainsi que sur les mainframes sous MVS, VM ou DOS/VSE. Ce code qui était le précurseur du code ASCII a été dévoilé entre 1963 et 1964 lors du lancement de la série 360 d'IBM. Il été crée pour améliorer le code BCD.

b8-b5		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
b4-b1		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL	DLE	DS		SP	&	-						{	}	⁄	0
0001	1	SOH	DC1	SOS				/		a	j	~		A	J		1
0010	2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	3	ETX	DC3	WUS	IR					c	l	t		C	L	T	3
0100	4	SEL	RES	BYP	PP					d	m	u		D	M	U	4
0101	5	HT	NL	LF	TRN					e	n	v		E	N	V	5
0110	6	RNL	BS	ETB	NBS					f	o	w		F	O	W	6
0111	7	DEL	POC	ESC	EOT					g	p	x		G	P	X	7
1000	8	GE	CAN	SA	SBS					h	q	y		H	Q	Y	8
1001	9	SPS	EM	SFE	IT				`	i	r	z		I	R	Z	9
1010	A	RPT	UBS	SM	REF	¢	!		:								
1011	B	VT	CU1	CSP	CU3	.	$	,	#								
1100	C	FF	FS	MFA	DC4	<	*	%	@								
1101	D	CR	GS	ENQ	NAK	(	)	_	'								
1110	E	SO	RS	ACK		+	;	>	=								
1111	F	SI	US	BEL	SUB	|	¬	?	"								EO

Le code EBCDIC Standard

$(BONJOUR)_{caractère}$= $(C2\ D6\ D5\ D1\ D6\ E4\ D9)_{EBCDIC\text{-}hexadécimal}$

$=(11000010\ 11010110\ 11010101\ 11010001\ 11010110\ 11100110\ 11011001)_{EBCDIC\text{-}binaire}$

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	PF	HT	LC	DEL	GE	RLF	SMM	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
2	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
3			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
4											¢	.	<	(	+	|
5	&										!	$	*	)	;	¬
6	-	/									¦	,	%	_	>	?
7										‘	:	#	@	'	=	"
8		a	b	c	d	e	f	g	h	i						
9		j	k	l	m	n	o	p	q	r						
A		~	s	t	u	v	w	x	y	z						
B																
C	{	A	B	C	D	E	F	G	H	I			∫		Ч	
D	}	J	K	L	M	N	O	P	Q	R						
E	\		S	T	U	V	W	X	Y	Z			⊦			
F	0	1	2	3	4	5	6	7	8	9	|					EO

Le code EBCDIC Standard

3.4.3. Code UTF

Unicode (1991)

Unicode créé en 1991 a pour vocation de rassembler tous ces codes afin d'avoir un code commun et unique pour toutes les langues au niveau mondial [16]. Il contient plus d'un million de caractères. Unicode utilise des points de code jusqu'à 4 octets soit plus de 24x8 =232 = 4,3 milliard de possibilités ! En 2015 Unicode définit plus de cent mille caractères. Chaque caractère abstrait est identifié par un nom unique (un en anglais et un en français) et associé à un point de code (= position de code).

Remarque : l'alphabet Chinois Kanji comporte à lui seul 6 879 caractères.

Gros **avantage** de l'Unicode : tous les caractères sont codés de façon unique et universelle.

Inconvénient : un caractère prend jusqu'à 4 octets soit 4 fois plus de place qu'en ASCII.

UTF-8 (1996)

UTF-8 (Unicode Transformation Format 8 bits) est dérivé de l'Unicode, et rétro-compatible avec la norme ASCII (codes de 0 à 127). Tout caractère ASCII est codé avec un seul octet, identique au code ASCII, alors que les autres caractères sont codés selon Unicode, sur 2 à 4 octets. UFT-8 donne le moyen de différencier les codes ASCII des codes Unicode.

Pour ASCII, le bit de poids fort est nul et les 7 autres bits (surlignés en vert ci-dessous) donnent le code ASCII. Pour Unicode, le bit de poids fort est non nul et suit la règle : 110, 1110, 11110... Le reste (surligné en vert ci-dessous) étant le code Unicode.

Ainsi, l'UTF-8 cumule à la fois les avantages de l'ASCII (taille des fichiers faible) et de l'Unicode (universalité des caractères).

UTF-16 et UTF-32

UTF-16 et UTF-32 suivent le même principe que UTF-8 mais le codage se fait sur 16 bits (UTF-16) ou 32 bits (UTF-32) au minimum, au lieu de 8 bits. Du coup, pour la plupart des langues latines qui utilisent abondamment les caractères ASCII, UTF-8 nécessite moins d'octets qu'UTF-16 ou UTF-32. Par contre pour les langues utilisant beaucoup de caractères extérieurs à ASCII, UTF-8 occupe sensiblement plus d'espace. En effet les idéogrammes employés dans les textes de langues asiatiques comme le chinois, le coréen ou le japonais utilisent 3 octets en UTF-8, contre 2 octets en UTF-16.

UTF-32 sera plus efficace uniquement pour les textes utilisant majoritairement des écritures anciennes ou rares codées hors du plan multilingue de base, c'est-à-dire à partir de U+10000, mais il peut aussi s'avérer utile localement dans certains traitements pour simplifier les algorithmes, car les points de code y ont toujours une taille fixe, la conversion des données d'entrée ou de sortie depuis ou vers UTF-8 ou UTF-16 étant triviale.

Utilisation actuelle d'UTF-8 et UTF-16

Windows. UTF-16 est le standard dans Windows (et donc NTFS) depuis Windows NT. Mais UTF-8 est totalement pris en charge depuis Windows 2000 et Windows XP. UEFI et GPT. Comme Windows, c'est UTF-16 qui a été choisi comme standard pour UEFI et GPT. Mac. Avec l'avènement

Mac OS X, le codage MacRoman a été remplacé par UTF-8 en tant que codage par défaut sur les systèmes d'exploitation Macintosh.

Remarque : le codage MacRoman inclut un glyphe correspondant à la pomme du logo Apple (point de code F0). Ce caractère n'existe pas en Unicode et doit donc posséder une correspondance dans la Zone d'Usage Privée. Apple utilise le point de code U+F8FF à cet effet. Internet. Depuis 2003, c'est l'UTF-8 qui est un des standards de l'Internet. Il est utilisé par 82,2 % des sites web en décembre 2014. Messagerie mail. A l'heure actuelle, tous les logiciels de messagerie (mail) supportent UTF-8.

Taille d'un fichier texte contenant 5 000 caractères, et sauvegardé dans différents formats :

	Caractères latins	**Kanjis japonais**
ANSI	5KO	Non supporté
Unicode	10KO	10KO
UTF-8	5KO	15KO

Chaque symbole d'écriture est représenté par un nom et une valeur hexadécimale préfixée par «**U+** ».

A « lettre majuscule latine A » **U+0041**

é « lettre minuscule latine e accent aigü » **U+00E9**

€ « symbole euro » **U+20AC**