Chapitre II : La représentation de l'information

1. introduction

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

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

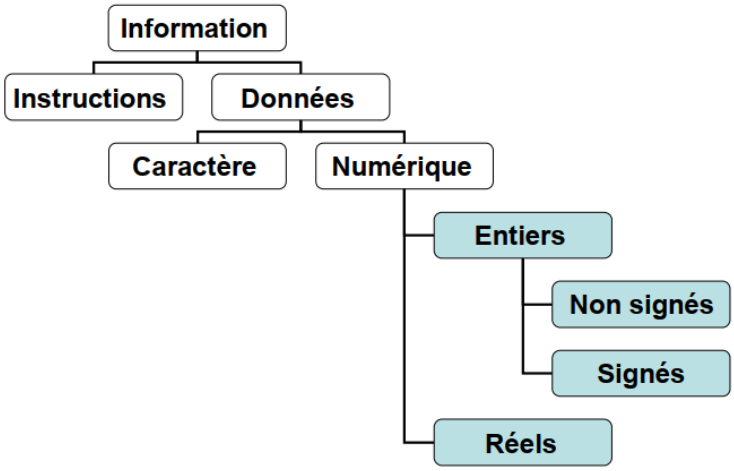
-Des nombres représentés par des chiffres.

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

On distingue deus 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).



2. Le codage binaire :

2.1. Le codage 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

Exemple :

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

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

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

2.2. Le code binaire réfléchi (ou code DE 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. Voici un exemple de code Gray sur 3 bits :

g2	g1	g0
0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

La construction peut se faire graphiquement par itération ou bien à partir du code en binaire naturel. Graphiquement :

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

1

1

0

On ajoute le bit de poids fort en binaire naturel.

0 0

0 1

1 1

1 0

Pour prolonger le code (passer sur 3 bits), on re-crée un nouvel axe de symétrie sur les deux bits faibles, puis on ajoute un bit supplémentaire en binaire naturel :

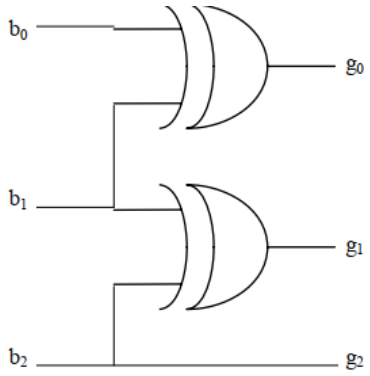
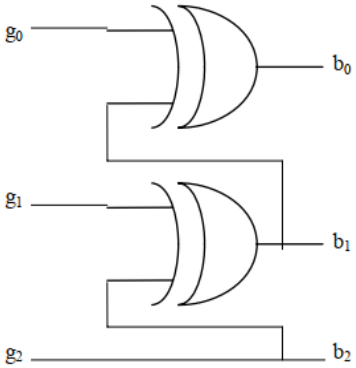
000
001
011
010
110
111
101
100

Et on recommence si on veut ajouter un bit supplémentaire.

En comparant le code binaire naturel et le code Gray sur 3 bits :

binaire			Gray		
b2	b1	b0	g2	g1	g0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

On déduit les équations et les schémas suivants :

De binaire naturel vers code Gray	De code Gray vers binaire naturel
g0 = b0⊕b1 g1 = b1⊕b2 g2 = b2	b0 = g0⊕b1 b1 = g1⊕b2 b2 = g2
	

Sur un nombre quelconque de bits, on obtient les équations :

De binaire naturel vers code Gray	De code Gray vers binaire naturel
gi = bi XOR bi+1 pour 0 < i < N-2 gN-1 = bN-1	bi = gi XOR bi+1 pour 0 < i < N-2 gN-1 = bN-1

Conversion binaire pur en code Gray

Il y a une autre méthode pour passer du code binaire normal au code gray, pour convertir un nombre binaire pur 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
3. Répéter le processus, Continuer l'addition de chaque bit avec le bit suivant jusqu'au bit du pois le plus faible.

Exemple

Nombre en binaire pur 1 0 1 1

Nombre en binaire réfléchi 1 1 1 0

$(1011)_2$ = $(1110)_{réfléchi}$

$(1011)_2=(1110)_{Gray}$

$(1101011000)_2 =(\mathbf{1011110100})_{Gray}$

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

Nombre en binaire réfléchi 1 0 1 1

Nombre en binaire pur 1 1 0 1

$(1011)_{réfléchi}$ = $(1101)_2$

Nombre en binaire réfléchi 1 1 0 1 0 1

Nombre en binaire pur 1 0 0 1 1 0

$(110101)_{réfléchi}$ = $(100110)_2$

$(11010110110)_{Gray}=(\mathbf{10011011011})_2$

2.3. Le 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 :

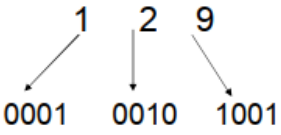
• Pour passer du décimal au binaire, il faut effectuer des divisions successives. Il existe une autre méthode simplifiée pour le passage du décimal au binaire.

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

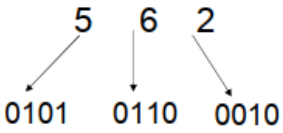
• Les combinaisons supérieures à 9 sont interdites

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

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).



$129 = (0001\ 0010\ 1001)_2$



$562 = (0101\ 0110\ 0010)_2$

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.

```
6   0110          5   0101
+                 +
3   0011          2   0010
9   1001          7   0111
```

```
4   0100      22      0010  0010
+             +
4   0100      12      0001  0010
8   1000      34      0011  0100
```

• **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.**

8		1000		18	0001	1000	
+					+		
9		1001		19	0001	1001	
17	1	0001		37	0010	10001	
	+ 0110 (6)					+ 0110 (6)	
0001	**0111**				**0101**	**0111**	

8+9 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 17, il faut donc pour corriger ajouter 6 (0110), ce qui va nous donner 10001+0110 = 10111. Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0001 0111 qui l'équivalent de 17 en BCD.

18+19 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 37, il faut donc pour corriger ajouter 6 (0110) sur la partie de droite qui a la retenue de 1, ce qui va nous donner **0010 10001**+0110 = 001010111. Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0101 0111 qui l'équivalent de 37 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**.

3		0011		57		0101	0111
+					+		
7		0111		26		0010	0110
10		1010		83		0111	1101
	+ 0110 (6)					+ 0110 (6)	
0001	0000					**1000**	**0011**

3+7 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 10, il faut donc pour corriger ajouter 6 (0110), ce qui va nous donner **1010**+0110 = 10000. Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 0001 0000 qui l'équivalent de 10 en BCD.

57+26 : Nous voyons que le résultat obtenu est incorrect car il n'est pas égal à 83, il faut donc pour corriger ajouter 6 (0110) à la partie se trouvant à droite, sachant que cette dernière qui est égale à 13 est supérieur 9, ce qui va nous donner **0111 1101**+0110=10000011. Si nous éclatons ce nombre en deux groupes de 4 à partir de la gauche, nous obtenons 10000 0011 qui l'équivalent de 83 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.

168	0001	0110	1000	
+				
779	0111	0111	1001	
947	1000	1101	**10001**	**Le premier résultat est incorrect.**
	1	**1**	**0110 (6)**	On ajoute au groupe de droite 6 (0110)
		1110	***10111***	On garde le nibble et on additionne la retenue au suivant
		0110 (6)		Le suivant, après addition, donne une retenue
	1001	***10100***		On garde le nibble et on additionne la retenue au suivant
	1001	0100	0111 = 947	

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.

- **91-37 en code BCD.**

```
 91 --BCD-->  1001  1001
-37 --BCD-->  0011  0111
 54           0101  1010
                  -0110
              0101  0100
               5     4
```

- **112-99 en code BCD**

```
112 --BCD-->  0001  0001  0010
- 99 --BCD-->  0000  1001  1001
 +13          0000  0111  1001
                   -0110 -0110
              0000  0001  0011
               0     1     3
```

2.4. Le code excède de trois.(Le codage EXCESS3 ou **BCD+3)**

Le code décimal binaire Excess-3 (XS-3), 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 XS-3, 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 XS-3 d'un nombre est donc similaire à son code BCD, à la différence que chaque groupe de quatre bits est incrémenté de 3

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

4 décimal codé binaire 0100, Pour avoir 4 en XS-3 on ajoute 011 à 0100 qui donne 0111
5 décimal codé binaire 0101, Pour avoir 5 en XS-3 on ajoute 011 à 0101 qui donne 1000

Le code Excess-3 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, il faut le convertir en DCB et ajouter à chaque élément 001 (3)

127 en BCD donne (0001 0010 0111)

Pour avoir 127 en XS-3 **on ajoute 011 à chaque élément**

127 en XS-3 donne (0001+011 0010+011 0111+011)

127 en XS-3 donne (0100 0101 1010)

Additionner deux nombre Excess XS-3

Pour additionner 2 nombres XS-3, il faut commencer par convertir les nombres en BCD, ajouter 3 ensuite additionner.

Décimal	BCD		XS-3	
2	0010 +	011	0101 (2 XS3)	
5	0101 +	011	1000 (5 XS3)	
			1101	le résultat n'est pas correct,
		-	0011	pour corriger, il faut soustraire 3 si retenu =0
			1010 (7 XS3)	résultat correct = $7+3=10_{10}$

Décimal	BCD			XS-3		
27	0010	0111	0010+011 0111+011	0101	1010	27
39	0011	1001	0011+011 1001+011	0110	1100	+ 39
			le résultat n'est pas correct,	1100	0110	66
pour corriger, il faut ajouter 3 (011) si retenu =1				-0011	+0011	+ 33
pour corriger, il faut enlever 3 (011) si retenu =0				1001	1001	99

Exemple : 72+19 en code BCD+3

72	BCD →	1010 0101
+19	BCD →	0100 1100
91		1111 0001
		-0011 +0011
		1100 0100
		9 1

3. Représentation des caractères :

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

• Le codage le plus utilisé est le ASCII (American Standard Code for Information Interchange)

• Dans ce codage chaque caractère est représenté sur 8 bits .

• Avec 8 bits on peut avoir 28 = 256 combinaisons

• Chaque combinaison représente un caractère

Exemple :

• Le code 65 (01000001)2correspond au caractère **A**

• Le code 97 (01100001) correspond au caractère **a**

• Le code 58 (00111010)correspond au caractère **:**

• Actuellement il existe un autre code sur 16 bits , se code s'appel **UNICODE** .

3.1. 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

EBCDIC character codes

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

3.2. Code ASCII

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).

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

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

Chaque caractère est représenté par un code hexadécimal :

A → 41H
B → 42H
a → 61H
…

De plus, la codification ASCII a respecté l'ordre de l'alphabet, ce qui permet de manipuler les données par leur codification et d'en effectuer un tri alphanumérique. Il est possible dans les logiciels d'utiliser le caractère ASCII ou le code correspondant.

Exemple : la représentation en ASCII hexadécimal du texte: STRUCTURE machine I

53 54 52 55 43 54 55 52 45 20 6D 61 63 68 69 6E 65 20 49

3.3. Code UTF

Unicode (1991) :

Unicode a pour vocation d'uniformiser le codage en donnant à tout caractère de n'importe quelle langue un identifiant unique au niveau mondial. 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

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1110 0000 10 1xxxxx 10xxxxxx	3 octets codant 12 à 16 bits
1110 0001 10xxxxxx 10xxxxxx	
1110 001x 10xxxxxx 10xxxxxx	
1110 01xx 10xxxxxx 10xxxxxx	
1110 10xx 10xxxxxx 10xxxxxx	
1110 1100 10xxxxxx 10xxxxxx	
1110 1101 10 0xxxxx 10xxxxxx	
1110 111x 10xxxxxx 10xxxxxx	
11110 000 10 01xxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits
11110 000 10 1xxxxx 10xxxxxx 10xxxxxx	
11110 001 10xxxxxx 10xxxxxx 10xxxxxx	
11110 01x 10xxxxxx 10xxxxxx 10xxxxxx	
11110 100 10 00xxxx 10xxxxxx 10xxxxxx	

reprise ASCII

reprise Unicode

UTF-8

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 que 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 de 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	5 ko	*non supporté*
Unicode	10 ko	10 ko
UTF-8	5 ko	15 ko

4. Représentation des nombres :

4.1. Nombres entiers :

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 » → (C1)
- Représentation en complément à « 2 » → (C2)

4.1.1. Représentation non signée.

4.1.2. Représentation avec signe et valeur absolue.

N : nombre → Représente le nombre en signe et valeur absolue ± N

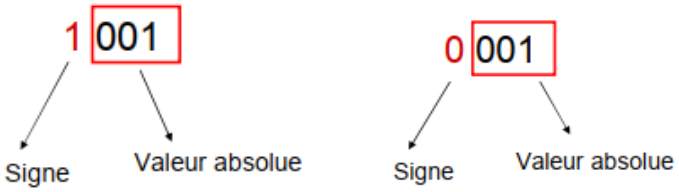
• Si on travail sur n bits, alors le bit du poids fort est utilisé pour indiquer le signe :

1 : signe négatif

0 : signe positif

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

Exemple : Si on travail sur 4 bits



1001 est la représentation de - 1 0001 est la représentation de + 1

Exemple : Représenter le nombre en signe et valeur absolue sur 8 bits

$(-15)_{10} = (-1111)_2$

=

1	0	0	0	1	1	1	1

4.1.3. Complément à 1 (ou Complément restreint)

Cette représentation concerne les nombres négatifs

$A = A_n A_{n-1} A_{n-2} \ldots\ldots\ldots A_2 A_1 A_0$

$(A)_{C1} = \overline{A}_n \overline{A}_{n-1} \overline{A}_{n-2} \ldots\ldots\ldots \overline{A}_2 \overline{A}_1 \overline{A}_0$

• 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.

CA1(CA1(N))= N

Exemple : Représenter le nombre en complément à 1 sur 8 bits

A = $(-17)_{10}$ → $(1\ 0010001)_{SVA}$ → $(1\ 1101110)_{C1}$

B= $(01101101)_2$ ⟶ $(10010010)_{Cà1}$

- **Sur 6 bits (-12)+(-14) en complément à 1**

-12 —SVA sur 6bits→ 1/01100 ⟶ 1/10011

-14 —SVA sur 6bits→ 1/01110 —C à 1→ 1/10001

-26 1 1/00100

\+ 1

00101

C à 1

11010

(- 26)$_{10}$

- **Sur 8 bits (-19)+(-14) en complément à 1**

-19 —SVA sur 6bits→ 1/0010011 —C à 1→ 1/1101100

-14 —SVA sur 6bits→ 1/0001110 —C à 1→ 1/1110001

-33 1 1/1011101

\+ 1

1011110

C à 1

0100001

(- 33)$_{10}$

- **Sur 8 bits (+11)+(-22) en complément à 1**

11 —SVA sur 8bits→ 0/0001011 ⟶ 0/0001011

-22 —SVA sur8bits→ 1/0010110 —C à 1→ 1/1101001

-11 1/1110100

C à 1 ↓

0001011

↓

(- 11)$_{10}$

4.1.4. 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)c2= (A)c1 +1

Exemple :

X=(18) —SVA sur 6bits→ **0/10010**

—C à 1→ **0/01101**

—C à 2→ **0/01110**

Y=(-30) —SVA sur 6bits→ **1/11110**

—C à 1→ **1/00001**

—C à 2→ **1/00010**

Addition et soustraction 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.

- **Sur 6 bits (-12)+(-14) en complément à 2**

-12 ⟶ 1/01100 ⟶ 1/10100

-14 —SVA sur 6bits→ 1/01110 —C à 2→ 1/10010

-26 11/0010

C à 2 ↓

11010

↓

(- 26)$_{10}$

- **X+Y en C à 2**

19 —SVA sur 8bits→ 0/0010011 ⟶ 0/0010011

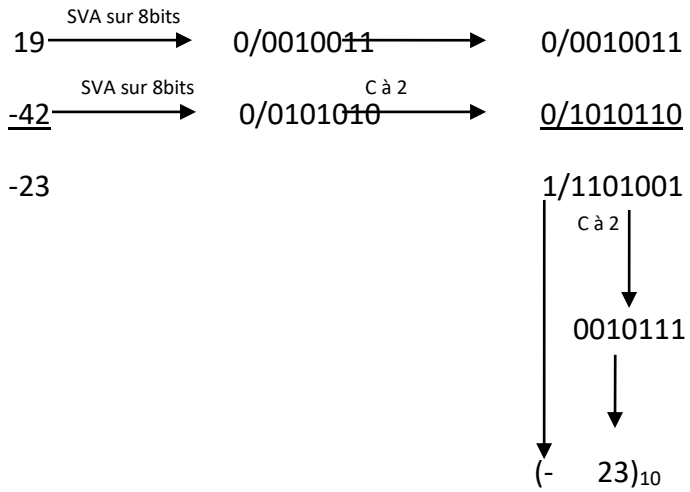
+42 —SVA sur 8bits→ 0/0101010 ⟶ 0/0101010

+61 0/0111101

↓ ↓

(+ 61)$_{10}$

- **X-Y en C à 2**



4.2. Les nombres fractionnaires :

4.2.1. 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 travail sur n bits, alors le bit du poids fort est utilisé pour indiquer le signe :

1 : signe négatif

0 : signe positif

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

D- 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.

+ 5.25	0 1 0 1. 0 1
+ 3.75	0 0 1 1. 1 1
= 9.00	1 0 0 1. 0 0

4.2.2. Virgule flottante (norme IEEE 754)

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

● la norme IEEE 754 est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique.

● il existe 2 formats associés à la norme IEEE 754 : le format simple précision (le nombre est représenté sur 32 bits) et le format double précision (le nombre est représenté sur 64 bits)

● que cela soit en simple précision ou en double précision, la norme IEEE754 utilise :

▪ 1 bit de signe (1 si le nombre est négatif et 0 si le nombre est positif)

▪ des bits consacrés à l'exposant (8 bits pour la simple précision et 11 bits pour la double précision)

▪ des bits consacrés à la mantisse (23bits pour la simple précision et 52bits pour la double précision)

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

– M : mantisse ,

– b : la base ,

– e : l'exposant

– S : signe

• **Exemple :**

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

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

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

• 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.

Les nombres flottants permettent de représenter, de **manière approchée**, une partie des nombres réels. La valeur d'un réel ne peut être ni trop grande, ni trop précise. Cela dépend du nombre de bits utilisés (en général 32 ou 64 bits). C'est un domaine très complexe et il existe une norme, IEEE-754, pour que tout le monde obtienne les mêmes résultats. Le codage en binaire est de la forme :

signe mantisse	exposant (signé)	partie fractionnaire de la mantisse (non signé)
1bits	**pbits**	**Kbits**

C'est-à-dire que 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 et zzzzz… est la partie fractionnaire de la mantisse. C'est, en binaire, la notation scientifique traditionnelle.

Voici quelques exemples de formats usuels :

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

Exemples :

❖ <u>représentations en virgule flottante sous format IEEE754 simple et double précision</u>

- **A=**$(-12.75)_{10} = (-1100.1)_2$

  $= (-1)^1.1,10011.2^{+3}$

S=1

$M=(100110\ldots\ldots.0)_2$

$E-127=3 \rightarrow E=(130)_{10}=(10000010)_2$

$A= (-12.75)_{10} \xrightarrow[\text{précision}]{\text{Simple}}$ (1 10000010 10011000……0)

32 bits

- **B=**$(-10.125)_{10} = (-1010.001)_2$

  $= (-1)^1.1,010001.2^{+3}$

S=1

$M=(0100010\ldots\ldots.0)_2$

$E-127=3 \rightarrow E=(130)_{10}=(10000010)_2$

$B=(-10.125)_{10} \xrightarrow[\text{précision}]{\text{Simple}}$ (1 10000010 010001000……0)

32 bits

- **C=$(27,25)_{10}=(+11011,01)_2$**

  $= (-1)^0 x1.101101x2^{+3}$

S=0

$M=(1011010\ldots\ldots0)_2$

$E-127=+3 \rightarrow E=127+3=(130)_{10}=(10000010)_2$

$C=(27,25)_{10} \xrightarrow[\text{précision}]{\text{Simple}}$ (0 10000010 10110100000000000000000)

32 bits

- **$D=(-13,5)_{10}=(-1101,1)_2$**

$=(-1)^1 x1.1011x2^{+3}$

$S=1$

$M=(10110\ldots\ldots0)_2$

$E-127=+3 \rightarrow E=127+3=(130)_{10}=(10000010)_2$

$D=(-13.5)_{10} \xrightarrow[\text{précision}]{\text{Simple}}$ (1 10000010 10110000000000000000000)

32 bits

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

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

$S=0$

$M=(100\ldots\ldots0)_2$

$E-127=-2 \rightarrow E=127-2=(125)_{10}=(01111101)_2$

$X=(+0.375)_{10} \xrightarrow[\text{précision}]{\text{Simple}}$ (0 01111101 10000000000000000000000)

32 bits

❖ <u>Les représentations en virgule flottante sous format IEEE754 double précision</u>

- **$X=(-12.75)_{10}=(-1100.1)_2$**

$=(-1)^1.1,10011.2^{+3}$

$S=1$

$M=(100110\ldots\ldots..0)_2$

$E-1023=3 \rightarrow E=(1026)_{10}=(10000000010)_2$

$x=(-12.75)_{10} \xrightarrow[\text{précision}]{\text{Double}}$ (1 1000000001010011000……0)

64 bits

- **$Y=(-10.125)_{10}=(-1010.001)_2$**

$=(-1)^1.1,010001.2^{+3}$

$S=1$

$M=(0100010\ldots\ldots..0)_2$

$E-1023=3 \rightarrow E=(1026)_{10}=(10000000010)_2$

$y=(-10.125)_{10} \xrightarrow[\text{précision}]{\text{Double}}$ (1 1000000010010001000……0)

64 bits

❖ <u>La représentation décimale</u>

- $A=(C038\ 0000\ 0000\ 0000)_{16}=(1100\ 0000\ 0011\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ldots\ldots\ldots\ldots\ldots\ldots)_2$

$S=1$

$M=(100\ldots\ldots..0)_2$

$E=(10000000011)_2 =(1027)_{10}$

$A=(-1)^1.1,1.2^{1027-1023}$

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

- $B=(0001\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000)_2$

$S=1$

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

$E=(10000010)_2 =(130)_{10}$

$B=(-1)^1.1,11.2^{130-127}$

$=-1,11.2^{+3}=(-1110)_2=(-14)_{10}$

- C=11011000011010110000000000000000

  S=1

  M=$(11010110\ldots\ldots\ldots0)_2$

  E=$(10110000)_2=(176)_{10}$

$C=(-1)^1x1.1101011x2^{176-127}$

$=(-1,1101011\ x2^{49})_2$

$=(11101011000000000000000000000000000000000000000000000000)_2$

$=(2^{49}+2^{48}+2^{47}+2^{45}+2^{43}+2^{42})_{10}$