IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

ELP111

Fonctions électroniques logiques et analogiques

Fiche séance

PC3 ARITHMETIQUE BINAIRE

IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom	ELP111
	Fonctions électroniques logiques et analogiques
	Fiche séance PC3 Arithmétique binaire

1. LES OBJECTIFS D'APPRENTISSAGE

OA1 Représenter un nombre dans n'importe quelle base arithmétique.
Enjeu : Notion de complexité dans un processeur matériel de traitement de l'information.

2. LES CONCEPTS

2.1 REPRESENTATION POLYNOMIALE D'UN NOMBRE, EXPRESSION D'UN NOMBRE DANS UNE BASE

De manière générale tout nombre *N* exprimé dans une base *b* peut se décomposer sous la forme polynomiale suivante :

$$N_{(b)} = S(a_n b^n + a_{n-1} b^{n-1} + \ldots + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + \ldots + a_{-m} b^{-m}) \qquad \text{(équation 1)}$$

avec

- S est le **signe** du nombre
- a_i est le **symbole de rang *i***, $a_i \in \mathbf{N}$ et $0 \leq a_i < b$
- a_n est le **symbole de poids le plus fort** (MSB : Most Significant Bit si $b = 2$), et a_{-m} est le **symbole de poids le plus faible** (LSB : Least Significant Bit si $b = 2$)

Le nombre $N_{(b)}$ s'exprime en numérotation de position par $S\, a_n a_{n-1} \ldots a_0 , a_{-1} \ldots a_{-m}$. Les symboles $a_n a_{n-1} \ldots a_0$ et $a_{-1} \ldots a_{-m}$ représentent respectivement la **partie entière** et la **partie fractionnaire** de *N*.

On appelle **dynamique** ou **amplitude de codage** d'une représentation la différence entre le plus grand nombre et le plus petit nombre représentables. On appelle **résolution** ou **précision** d'une représentation la différence entre deux nombres consécutifs. A titre d'exemple pour une représentation décimale d'entiers positifs sur 5 chiffres, la dynamique est égale à 99999 et la résolution est égale à 1.

2.2 REPRESENTATION D'UN NOMBRE POSITIF ET CONVERSION ENTRE BASES

Les bases de numération les plus utilisées sont la base **décimale** ($b = 10$), la base **binaire** ($b = 2$), et les bases dérivées de la base binaire : base **octale** ($b = 8$) et base **hexadécimale** ($b = 16$).

La numération binaire utilise les 2 bits 0 et 1, la numération octale utilise 8 chiffres : 0, 1, 2, 3, 4, 5, 6, 7, et la numération hexadécimale utilise 16 symboles : 0, 1, 2, ..., 9, A, B, C, D, E, F (les symboles A à F ont pour équivalents décimaux les nombres 10 à 15).

Le système binaire et ses dérivés sont ceux utilisés pour le codage des informations dans les systèmes numériques. La base 16 est couramment utilisée car elle peut être considérée comme une écriture condensée de l'écriture binaire et, par conséquent, sa conversion vers le binaire est particulièrement aisée.

Les conversions les plus utilisées sont les suivantes

- base *b* vers base 10
- base 10 vers base *b*
- base 2 vers base 2^n (8 ou 16)
- base 2^n (8 ou 16) vers base 2

2.2.1 Base b vers base 10

Pour convertir un nombre d'une base *b* vers la base décimale, on utilise la **méthode** dite **des additions** qui consiste à utiliser la représentation du nombre sous forme polynomiale (équation 1).

Exemple 1 : conversion du nombre binaire entier $N_{(2)} = 1101\ 0011_{(2)}$ en base 10.

$$N = 1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 128 + 64 + 16 + 2 + 1 = 211_{(10)}$$

Exemple 2 : conversion du nombre binaire fractionnaire $N_{(2)} = 110011,1001_{(2)}$ en base 10

$$N = 1.2^5 + 1.2^4 + 1.2^1 + 1.2^0 + 1.2^{-1} + 1.2^{-4} = 51,5625_{(10)}$$

Exemple 3 : conversion du nombre octal entier $N_{(8)} = 4513_{(8)}$ en base 10

$$N = 4.8^3 + 5.8^2 + 1.8^1 + 3.8^0 = 2379_{(10)}$$

Exemple 4 : conversion du nombre hexadécimal fractionnaire $N_{(16)} = 1B20,8_{(16)}$ en base 10

$$N = 1.16^3 + 11.16^2 + 2.16^1 + 8.16^{-1} = 6944,5_{(10)}$$

N.B. : La méthode des additions requiert la connaissance des puissances successives de la base de départ.

2.2.2 Base 2 vers base 8 ou 16

Il s'agit d'utiliser la représentation du nombre sous sa forme polynomiale (équation 1), et à factoriser par des puissances de 8 ou de 16.

Exemple 1 : conversion du nombre entier $N_{(2)} = 1101\ 0011$ en base 8

$N_{(2)} = 1101\ 0011 = 1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0$

$N_{(2)} = 8^2.(1.2^1 + 1.2^0) + 8^1.(0.2^2 + 1.2^1 + 0.2^0) + 8^0.(0.2^2 + 1.2^1 + 1.2^0)$

$N_{(2)} = 3.8^2 + 2.8^1 + 3.8^0 = 323_{(8)}$

Visuellement, cela peut se représenter par des regroupements de bits de taille 3 :

$N_{(2)} = \underbrace{1101}_{3}\ \underbrace{0011}_{2\quad 3}$

2.2.3 Base 10 vers base b

2.2.3.1. Nombres entiers positifs (2 méthodes disponibles)

Il existe deux méthodes pour convertir un nombre entier positif :

- Méthode des divisions successives.
- Méthode des soustractions successives : on travaille directement à partir des puissances de la base d'arrivée. Cette méthode est rapide à appliquer en base 2 où les puissances de 2 sont connues facilement.

(a) Méthode des divisions successives

Pour effectuer une conversion d'un entier décimal dans une autre base on applique la **méthode des divisions successives** : on effectue des divisions successives du nombre par cette base, les restes successifs forment alors le nombre converti.

A titre d'exemple, dans le cas d'une conversion d'un nombre décimal en son équivalent binaire, on réalisera des divisions successives par 2. Les restes de ces divisions formeront le nombre converti dans la base 2.

Exemple 1 : conversion de $N_{(10)} = 52$ en base 2

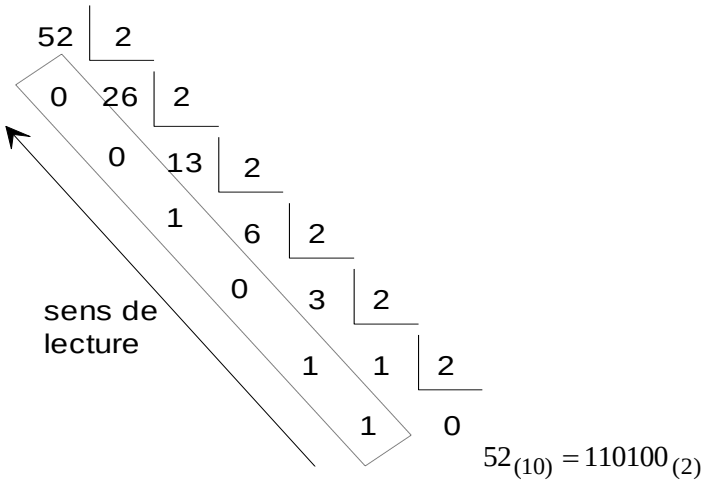


figure 1.1 : conversion de $52_{(10)}$ en base 2 par divisions successives par 2

Exemple 2 : conversion de $N_{(10)} = 90$ en base 8

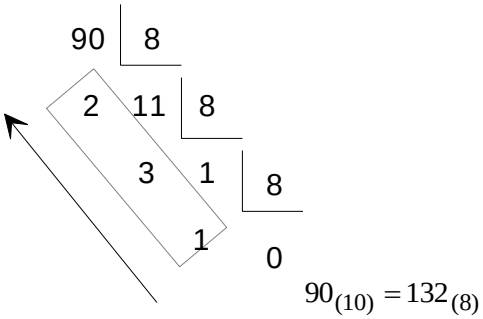


figure 1.2 : conversion de $90_{(10)}$ en base 8 par divisions successives par 8

Chaque division revient à opérer un décalage à droite d'une position et permet ainsi d'isoler un bit dans la partie fractionnaire.

(b) Méthodes des soustractions successives

Si les puissances successives de la base d'arrivée sont connues, on peut également, plutôt que d'utiliser la méthode précédente, effectuer la transformation par **soustractions successives** de ces puissances. Cette méthode est illustrée sur les deux exemples traités précédemment.

Exemple 1 : conversion de $N_{(10)} = 52$ en base 2

$32(=2^5) \le N < 64(=2^6)$, on peut donc retrancher 32 à N : $N = 32 + 20$,

$16(=2^4) \le 20 < 32(=2^5)$, on peut retrancher 16 : $N = 32 + 16 + 4$,

4 est une puissance de 2, l'itération est donc terminée. On en déduit :

$N = 2^5 + 2^4 + 2^2 = 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 0.2^0 = 110100_{(2)}$.

Exemple 2 : conversion de $N_{(10)} = 90$ en base 8

$64(=8^2) \le N < 512(=8^3)$, on retranche 64 à N : $N = 64 + 26$,

$8 \le 26 < 64(=8^2)$, on retranche 8 : $N = 64 + 8 + 18$,

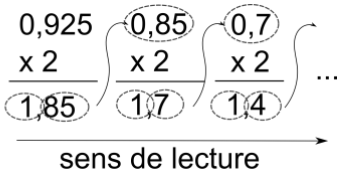
on peut de nouveau retrancher 2 fois 8 à 18, on obtient alors : $N = 64 + 3.8 + 2$. Puisque 2 est inférieur à 8, l'itération est terminée, d'où $N = 1.8^2 + 3.8^1 + 2.8^0 = 132_{(8)}$.

2.2.3.2. Nombres réels positifs (ou fractionnaires)

Pour convertir un nombre fractionnaire de la base 10 vers une autre base, il faut procéder en deux étapes. La partie entière du nombre est convertie comme indiqué précédemment ; la partie fractionnaire du nombre est convertie par **multiplications successives** : on multiplie successivement la partie fractionnaire par la base cible, en retenant les parties entières qui apparaissent au fur et à mesure.

Exemple 1 : conversion de $N_{(10)} = 12,925$ en base 2

- partie entière : $12_{(10)} = 1100_{(2)}$
- partie fractionnaire :

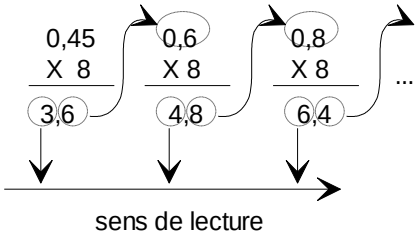


$0,925_{(10)} = 0,111\ldots_{(2)}$

figure 1.3 : conversion de $0,925_{(10)}$ en base 2 par multiplications successives

Finalement $12,925_{(10)} = 1100,111\ldots_{(2)}$

Exemple 2 : conversion de $N_{(10)} = 0,45$ en base 8



$0,45_{(10)} = 0,346\ldots_{(8)}$

figure 1.4 : conversion de $0,45_{(10)}$ en base 8 par multiplications successives

Il est visible sur les deux exemples précédents que la conversion peut ne pas se terminer et que l'on obtient, en s'arrêtant à un nombre fini de positions une approximation de la représentation du nombre.

2.2.3.3. Maintien de la résolution pour les nombres réels positifs

Soit $(a_n a_{n-1} \ldots a_0, a_{-1} \ldots a_{-m})_{(10)}$ et $(a_p a_{p-1} \ldots a_0, a_{-1} \ldots a_{-k})_{(b)}$ les numérotations de position d'un même nombre *N* exprimé respectivement en base 10 et en base *b*. La résolution d'une base *b* est définie comme étant la différence entre 2 nombres consécutifs dans cette base. Elle est donc égale à b^{-k}.

Ainsi, la résolution est conservée lors du passage de la base 10 à la base b si et seulement si la résolution du nombre transformé en base *b* est inférieur ou égal à la résolution de ce nombre en base 10. $b^{-k} \leq 10^{-m}$, c'est-à-dire si $k \log b \geq m \log 10$, soit

$$\boxed{k \geq m \frac{\log 10}{\log b}}$$

Exemple 1 : pour conserver la résolution lors du passage de $0{,}925_{(10)}$ en base 2, il faut garder $k \geq 3 \frac{\log 10}{\log 2} \approx 9{,}97$, soit 10 bits après la virgule.

Exemple 2 : pour conserver la résolution lors de la conversion de $0{,}45_{(10)}$ en base 8, il faut garder $k \geq 2 \frac{\log 10}{\log 8} \approx 2{,}2$, soit 3 bits après la virgule.

2.3 REPRESENTATION BINAIRE DES NOMBRES SIGNES (COMPLEMENT A 2)

Les systèmes numériques doivent être capables de traiter des nombres positifs et négatifs. L'utilisation d'une représentation signée suppose l'utilisation d'un **format** (nombre de bits) fixé au préalable.

2.3.1 Représentation en complément à 2

Le **complément à 2** est le mode de représentation le plus utilisé en arithmétique binaire et donc dans les ordinateurs pour coder les nombres entiers.

Dans cette représentation, les **nombres positifs** se représentent par leur valeur binaire naturelle. Par exemple +6 est représenté par 0000 0110 sur un format de 8 bits.

La représentation **des nombres négatifs** s'obtient comme suit :

- On part de la représentation binaire naturelle de l'opposé arithmétique du nombre à coder (nombre positif),
- On calcule son **complément à 1** (CA1) ou **complément restreint**. Celui-ci est obtenu en inversant tous ses bits,
- On en déduit son **complément à 2** (CA2) ou **complément vrai** en ajoutant 1 au niveau du LSB, c'est-à-dire en réalisant l'addition binaire du complément à 1 et de 1. L'addition binaire est décrite succinctement à la section 2.4.

Exemple : représentation de -5 en CA2 sur un format de 8 bits

- Représentation binaire naturelle de +5 : 5 = 0000 0101,

- CA1 de +5 : $\overline{5} = 1111\ 1010$,
- CA2 de +5 : $-5 = 1111\ 1011$.

On identifie le CA2 d'un nombre à son opposé arithmétique, ainsi CA2(A)=-A.

En effet, soit $A = a_{n-1}\ldots a_1 a_0$, alors $\overline{A} = \overline{a_{n-1}}\ldots\overline{a_1}\ \overline{a_0}$, et donc $A + \overline{A} = 11\ldots11$, soit $A + \overline{A} = 2^n - 1$, et $-A = \bar{A} + 1 = CA2(A)$. L'addition de 2 nombres sur un format fixé de *n* bits étant toujours réalisée modulo 2^n (cf. section 1.4).

La représentation en complément à 2 présente les caractéristiques suivantes :

- Le principe d'obtention de l'opposé d'un nombre négatif est le même que celui permettant d'obtenir l'opposé d'un nombre positif,
- Le nombre 0 a une représentation unique,
- Un format sur *n* bits permet de coder en CA2 les nombres *N* vérifiant

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

Par exemple, pour *n* = 4,

$N_{(10)}$	$N_{(2)}$	$\overline{N}_{(2)}$	$-N_{(2)}$	$-N_{(10)}$
0	0000	1111	0000	0
1	0001	1110	1111	-1
2	0010	1101	1110	-2
3	0011	1100	1101	-3
4	0100	1011	1100	-4
5	0101	1010	1011	-5
6	0110	1001	1010	-6
7	0111	1000	1001	-7
			1000	-8

tableau 1 : représentation en complément à 2 sur 4 bits

On peut ainsi représenter des nombres compris entre -4 et +3 sur un format de 4 bits, entre -16 et +15 sur un format de 5 bits, entre -32 et +31 sur un format de 6 bits, entre -64 et +63 sur un format de 7 bits, etc.

Remarques :

- Le bit de poids fort (MSB) est représentatif du bit de signe, mais il est traité comme les autres bits dans les opérations arithmétiques : si MSB = 0 le nombre est positif, si MSB = 1 le nombre est négatif.
- Le nombre $+2^{n-1}$ n'est pas représenté. En effet, dans le cas où *n* = 4, le calcul du CA2 de 1000 donne CA2(-8)=-(-8)=CA2(1000)=0111 + 1 = 1000 = $(-8)_{(10)}$. Ce qui est arithmétiquement incorrect car 0 est le seul entier à être son propre opposé. On a donc choisi de supprimer la représentation du nombre $+2^{n-1}$, un MSB à 1 étant représentatif d'un nombre négatif.
- La représentation des nombres négatifs peut être recalculée rapidement en observant que les nombres négatifs entre 1000 (-8) et 1111 (-1) correspondent à une translation du bloc des nombres codés en binaire naturel de 1000 (8) à 1111 (15) de 16 positions vers le bas. C'est la conséquence de l'addition modulo 2^n .

La représentation sur 4 bits (en CA2 signé et en binaire naturel non signé) est expliquée sur la figure 1.5.

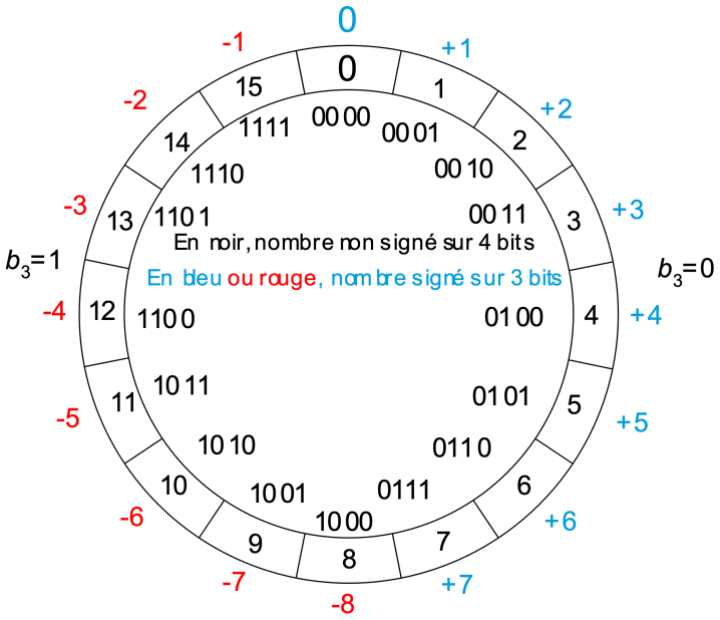


figure 1.5 : Représentation d'un nombre signé et non signé sur 4 bits.

2.3.2 Extension d'un nombre codé en CA2

L'extension d'un nombre codé sur *n* bits à un format sur *n+k* bits est réalisée comme suit :

- Si le nombre est positif, on complète les *k* bits de poids forts par des 0. Par exemple,

$$\underbrace{0110_{(CA2)}}_{6_{(10)}\text{ codé sur 4 bits}} \xrightarrow{6\text{ bits}} \underline{00}0110_{(CA2)}$$

.

- Si le nombre est négatif, on complète les *k* bits de poids forts avec des 1. Par exemple,

$$\underbrace{1010_{(CA2)}}_{-6_{(10)}\text{ codé sur 4 bits}} \xrightarrow{6\text{ bits}} \underline{11}1010_{(CA2)}$$

.

2.4 ADDITION ET SOUSTRACTION BINAIRE

Dans toutes les bases, le principe de **l'addition** est similaire à celui de l'addition décimale : on additionne symbole par symbole en partant des poids faibles, et en propageant éventuellement une retenue. Des exemples sont donnés en fin de section.

Si le nombre de bits est fixé à *n*, alors on réalise une addition modulo 2^n (puisqu'on ne peut pas aller au-delà de $2^n - 1$) .

Par exemple, $A + (-A) = 2^n = 0 \bmod 2^n$

En effet, $A + (-A) = A + CA2(A) = A + \bar{A} + 1 = \underbrace{(1 \dots 1)}_{n\ bits} + 1 = 10 \dots 0 = 2^n = 0\ mod\ 2^n$ car 2^n n'est pas représentable sur *n* bits.

Exemple sur 3 bits : $+2 + (-2) = 010 + 101 + 1 = 111 + 1 = 1000 = 000\ mod\ 2^n$

La **soustraction**, en arithmétique binaire, est le plus souvent appliquée sur des nombres signés. Dans ce cas, cette opération se ramène dans tous les cas à une addition.

Si le format des nombres est fixe, le résultat de l'addition peut donner lieu à **un dépassement de capacité**.

Dans le cas où les nombres **sont codés en binaire naturel**, le résultat de l'addition de 2 nombres binaires codés sur *n* bits peut dépasser la plus grande valeur codable sur *n* bits, qui est $2^n - 1$ en binaire naturel.

Dans le cas où les nombres **sont codés en complément à 2**, l'addition de 2 nombres exprimés sur *n* bits fournit toujours un résultat correct, sauf dans le cas où le résultat n'est pas représentable sur les *n* bits. Il y a alors **dépassement de capacité** lorsque les deux opérandes sont de même signe et que le résultat est de signe opposé.

Dans le registre d'état d'un ordinateur, deux indicateurs viennent renseigner le programmeur (ou le système d'exploitation) sur la **validité des résultats obtenus** : la **retenue** (carry *C*) et le **débordement** (overflow *OVF*). L'indicateur *C* signale la présence d'une retenue au niveau des poids forts; l'indicateur *OVF* indique que le résultat de l'opération n'est pas représentable dans le système du complément à 2 sur le format défini au départ.

Nous allons illustrer le positionnement de la retenue et du débordement par quatre exemples pour des nombres signés, codés en CA2 sur 8 bits :

0000 0110 (+6)	0111 1111 (+127)	0000 0100 (+4)	0000 0100 (+4)
+ 0000 0100 (+4)	+ 0000 0001 (+1)	+ 1111 1110 (-2)	+ 1111 1100 (-4)
0000 1010 (+10)	1000 0000 (-128)	1 0000 0010 (+2)	1 0000 0000 (0)
OVF=0	OVF=1	OVF=0	OVF=0
C=0	C=0	C=1 ignoré	C=1 ignoré
résultat correct	résultat incorrect	résultat correct	résultat correct

Exemple 1 : le résultat (+10) est codable sur 8 bits, le résultat est correct.

Exemple 2 : le résultat (+128) n'est pas codable sur 8 bits. Du fait de l'addition modulo 2^n, on obtient -128, le résultat est incorrect et OVF est positionné à 1 pour indiquer au système d'exploitation que le résultat est incorrect.

Exemple 3 : le résultat (+2) est codable sur 8 bits, le résultat est correct.

Exemple 4 : le résultat (0) est codable sur 8 bits , le résultat est correct.

Remarques :

- La retenue sortante (C) est toujours ignorée lorsque l'on manipule des nombres signés en CA2 et que le format des nombres est fixé
- La sortie OVF indique si le résultat est correct ou non ; si les 2 nombres sont positifs (resp. négatifs) et le résultat négatif (resp. positif) alors OVF = 1.
- Pour que le résultat d'une opération sur *n* bits soit correct dans la méthode du complément à 2, il faut que les retenues de rang *n* et de rang *n*+1 soient identiques (cf. Séance sur l'addition binaire).

2.5 REPRESENTATION BINAIRE DES NOMBRES REELS

Dans les calculateurs, deux représentations sont utilisées pour représenter les nombres fractionnaires : le codage en virgule fixe et le codage en virgule flottante.

2.5.1 Codage en virgule fixe

Dans cette représentation, les nombres réels sont représentés par des entiers, après avoir décidé d'un facteur d'échelle *k* qui est une puissance de la base dans laquelle on écrit les entiers. Autrement dit, un bloc de *n* bits est considéré comme un nombre dont la partie entière et le signe sont codés sur $n-k$ bits, et dont la partie fractionnaire est codée sur *k* bits. La résolution d'une telle représentation est de 2^{-k}. L'addition de deux nombres réels en virgule fixe (s'ils possèdent le même facteur d'échelle *k*) revient à additionner les deux entiers qui représentent ces nombres. Ce codage est surtout utilisé dans les processeurs de traitement de signal (DSP) où les exigences de rapidité sont primordiales. En revanche, la dynamique de cette représentation est assez limitée : pour un format sur *n* bits avec *k* bits après la virgule, la dynamique est de $(2^n - 1)/2^k$.

2.5.2 Codage en virgule flottante

Dans le cas précédent, le facteur d'échelle était fixe. Dans le cas du codage en virgule flottante, le facteur d'échelle peut varier au cours du calcul. On utilise l'écriture semi-logarithmique suivante :

$$N = S.M.b^e \qquad \text{(équation 2)}$$

avec :

- S : signe du nombre,
- M : mantisse,
- b : base de numération (ici $b = 2$),
- e : exposant.

L'exposant *e* est représentatif de l'ordre de grandeur du nombre et la mantisse *M* est représentative de sa précision. Le terme b^e joue le rôle de facteur d'échelle de la représentation, mais il est ici *explicite*, contrairement au cas précédent.

Exemple : représentation normalisée IEEE simple précision des nombres flottants en machine sur 32 bits (Standard IEEE 754-1985)

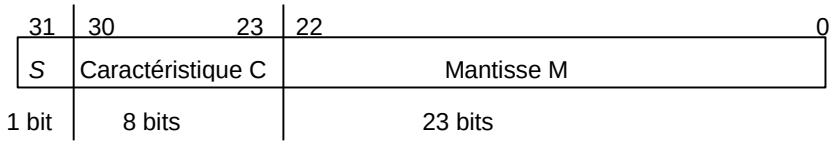


figure 1.6 : norme IEEE, simple précision sur 32 bits

Chaque valeur à représenter est dans ce cas déterminée par l'expression suivante :

$$N = (-1)^S \times 2^{C-127} \times 1,M \qquad \text{(équation 3)}$$

où la valeur de l'exposant *e* est donnée par $e = C - 127$. Ainsi, C peut s'exprimer en binaire naturel en s'affranchissant du problème de son signe. C est toujours positif. Le 1 précédent la mantisse n'est pas représenté en machine. On l'appelle le bit caché.

Cette représentation présente des singularités

- $C =' 0..0'$ et $M = '0..0'$: nombre zéro,
- $C \neq' 0..0'$ et $C \neq' 1..1'$, float normalisé (ou normal)
  $N = (-1)^S \times 2^{C-127} \times 1,M$
- $C =' 0..0'$ et $M \neq '0..0'$, float dénormalisé (très petit)
  $N = (-1)^S \times 2^{-127} \times 0,M$
- $C =' 1..1'$ (255) nombre infini ($+\infty$ si $S = 0$, $-\infty$ si $S = 1$) si $M = '0..0'$, *NaN* (not a number) sinon.
  $N = (-1)^S \times 2^{128} \times 1,0 = \pm\infty$

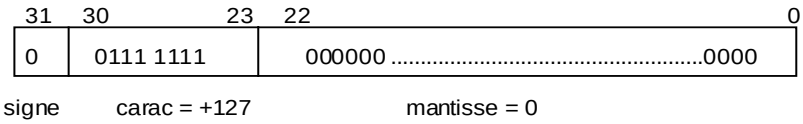
Exemple 1 : traduire le nombre décimal $1_{(10)}$ dans le format précédemment présenté.

Il faut le mettre sous la forme normalisée de l'équation 3, soit

$$1 = (-1)^0 \times 2^{127-127} \times 1,00 \cdots 0$$

d'où *S* = 0, *C* = 127, et *M* = 000.....0.

On en déduit la représentation en virgule flottante :



soit $3F800000_{(16)}$.

Exemple 2 : traduire le nombre décimal $N = +4,75_{(10)}$dans le format précédemment présenté.

On commence par convertir en binaire → $N = 100,11$ sur 5 bits.

Puis on décale la virgule jusqu'à obtenir un nombre en 1,.. →$N = 1,0011 \times 2^2$

Afin de conserver le nombre N, un décalage de la virgule de 2 positions à gauche doit être compensé par une multiplication du nombre par 2^2.

On peut ainsi récupérer la mantisse $M = 00110..0$ sur 23 bits. Puis l'exposant $e = C - 127 = 2$ soit $C = 129 = 10000001$.

On en déduit la représentation en virgule flottante :

0	10000001	00110……..0

La représentation en virgule flottante autorise une dynamique plus grande qu'en virgule fixe. Dans le format précédent, on peut coder des nombres dont la valeur absolue va de 10^{-38} à

10^{38}, et le standard d'écriture en virgule flottante double précision (sur 64 bits) permet de représenter des nombres atteignant 10^{300}.

2.6 REPRESENTATION DES CARACTERES ALPHANUMERIQUES

Certains codes peuvent avoir une signification non numérique. Le plus connu d'entre eux est le code ASCII (American Standard Code for Information Interchange), qui est utilisé pour représenter les caractères alphanumériques. Dans ce code, 128 combinaisons (lettres, chiffres, signes de ponctuation, caractères de contrôle, etc.) sont codées à l'aide de 7 bits de 00000000 à 01111111. Les transmissions asynchrones entre machines s'effectuant souvent sur un format de 8 bits, le dernier bit est alors utilisé pour contrôler la parité du message. Ainsi, sur les 8 bots de l'octet, celui de poids fort est placé à zéro.

La plus grande limitation de l'ASCII est donc le nombre de caractères codables. Un système de codage plus performance a donc été proposé : UCS (*Universal Character Set*), également appelé Unicode. Ce code permet de représenter n'importe quel caractère de n'importe quel système d'écriture de langue par un identifiant numérique et un nom unique, et ce de manière unifiée quelle que soit la machine utilisée. Une fois le code défini, il faut en déduire le formatage, c'est à dire son association à un code binaire, c'est l'objectif de UTF-8 (UCS transformation Format 8 bits). Le tableau 2 présente trois exemples.

Caractère	Id. Unicode	Codage binaire UTF-8
A	65	**0**1000001
é	233	**110**00011 **10**101001
€	8364	**1110**0010 **10**000010 **10**101100

tableau 2 : Exemples de format UTF-8.

Pour le caractère A, le bit de poids fort à '0' indique que le caractère n'est représenté que par un unique octet (c'est de l'ASCII). Pour le caractère é, l'entête 110 indique que le caractère est constitué d'un train de 2 octets. Pour un train de 3 octets et 4 octets, nous avons l'entête 1110 et 11110, respectivement. Enfin, un entête 10 indique que le l'octet fait partie d'un train d'octets.

3. LES EXERCICES D'APPLICATION

3.7 LES NOMBRES ENTIERS, REPRESENTATION ET CONVERSION DE BASES

Effectuer les conversions suivantes:

a) $1030_{(10)}$ = _____$_2$

b) $1030_{(10)}$ = _____$_{(8)}$ (à déduire du a))

c) $1030_{(10)}$ = _____$_{(16)}$ (à déduire du a))

Expliquer pour chaque conversion la méthode utilisée.

3.8 LES NOMBRES ENTIERS SIGNES, COMPLEMENT A 2

3.8.1 Représentation binaire en CA2

Donner la représentation en Complément à 2 (CA2) des nombres décimaux suivants:

+ 34 + 4,75 - 57 - 23, 75

3.8.2 Conversion d'un nombre en CA2 en sa valeur décimale

Calculer la valeur décimale des nombres suivants codés en CA2:

0 1 0 1 1 1 0 1 1 1 1, 1 1 0 0 0, 0 1

0 1 0 0 1 0 0 0 1 0 0 1

3.9 LES NOMBRES REELS

3.9.1 Conversion de bases

Effectuer les conversions suivantes:

a) $ABCD,1F_{(16)}$ = _____$_{(8)}$

b) $ABCD,1F_{(16)}$ = _____$_{(10)}$

c) 31,35 = _____$_{(2)}$ avec 8 bits en tout

d) $34,703125_{(10)}$ = _____$_{(2)}$ sans approximation

Expliquer pour chaque conversion la méthode utilisée.

3.9.2 Représentation de nombres en virgule flottante dans les processeurs (norme IEEE 754)

Donner la représentation en virgule flottante simple précision (sur 32 bits) du nombre décimal:

+ 35, 5703125

4. POUR ALLER PLUS LOIN

REPRESENTATION SIGNE+AMPLITUDE (OU MODULE)

Il s'agit d'une représentation parfois utilisée car plus simple que celle du CA2, mais qui est moins bien adaptée aux opérations arithmétiques.

Dans cette représentation, le bit de poids le plus fort représente le signe (MSB = 0 => nombre positif, MSB = 1 => nombre négatif), et les autres bits la valeur absolue du nombre.

Ainsi, un format de *n* bits permet de coder les nombres compris entre $-(2^{n-1}-1)$ et $2^{n-1}-1$.

Dans cette représentation, le zéro possède deux notations possibles.

Par exemple, pour *n* = 4,

$N_{(10)}$	$N_{(2)}$	$-N_{(2)}$	$-N_{(10)}$
0	0000	1000	0
1	0001	1001	-1
2	0010	1010	-2
3	0011	1011	-3
4	0100	1100	-4
5	0101	1101	-5
6	0110	1110	-6
7	0111	1111	-7

tableau 3 : représentation "module + signe" sur 4 bits

N. B. Extension d'un nombre en représentation "module + signe"

L'extension d'un nombre codé sur *n* bits à un format sur *n+k* bits consiste à décaler le bit de signe à la position du MSB et à compléter les autres positions par des 0, que le nombre soit positif ou négatif. Par exemple

$$\underbrace{0110_{(M+S)}}_{6_{(10)}\text{ codé sur 4 bits}} \xrightarrow{6\text{ bits}} 0\underset{S}{0}0110_{(M+S)} \quad \text{, et} \quad \underbrace{1110_{(M+S)}}_{-6_{(10)}\text{ codé sur 4 bits}} \xrightarrow{6\text{ bits}} 1\underset{S}{0}0110_{(M+S)} \text{ .}$$

REPRESENTATION BINAIRE DECALEE

Cette représentation peut être déduite du complément à 2 par une simple inversion du bit de signe (MSB = 0 => nombre négatif, MSB = 1 => nombre positif). Cette représentation est commode pour la conversion numérique/analogique car la valeur maximale positive est codée par tous les bits à 1 et la valeur minimale négative par tous les bits à 0.

Par exemple, pour *n* = 4,

$N_{(10)}$	$N_{(2)}$	$-N_{(2)}$	$-N_{(10)}$
0	1000	1000	0
1	1001	0111	-1
2	1010	0110	-2
3	1011	0101	-3

4	1100	0100	-4
5	1101	0011	-5
6	1110	0010	-6
7	1111	0001	-7
		0000	-8

tableau 4 : représentation binaire décalée sur 4 bits

CLASSIFICATION DES CODES BINAIRES

Le champ d'application des systèmes numériques est très étendu. Lorsque l'application ne nécessite pas de calculs arithmétiques, les codages précédents sont inutiles ou peu adaptés. On utilise alors des codages possédant d'autres propriétés. On emploie ainsi dans certains systèmes des codes permettant d'éviter des états transitoires parasites lors de la saisie de données, ou de visualiser facilement des chiffres ou des lettres, ou bien encore de détecter des erreurs et/ou de les corriger dans un résultat susceptible d'être erroné. Nous présentons ci-après quelques codes fréquemment utilisés.

L'ensemble des codes binaires peuvent être regroupés en deux classes : les **codes pondérés** et les **codes non pondérés**.

Codes pondérés

Un code est dit pondéré si la position de chaque symbole dans chaque mot correspond à un poids fixé : par exemple 1, 10, 100, 1000 ... pour la numération décimale, et 1, 2, 4, 8, ... pour la numération binaire. Les codes pondérés ont, en général, des propriétés intéressantes du point de vue arithmétique.

Le code binaire pur et ses dérivés (octal, hexadécimal)

Ce sont les codes utilisés en arithmétique binaire et qui ont été étudiés dans la première partie de ce chapitre.

Le code DCB (Décimal Codé Binaire) ou BCD (Binary-Coded Decimal)

Ce code est utilisé dans de nombreux systèmes d'affichage, de comptage ou même les calculatrices de poche. Dans le code BCD chaque chiffre d'un nombre décimal (de $0_{(10)}$ à $9_{(10)}$) est codé à l'aide de 4 bits (de $0000_{(2)}$ à $1001_{(2)}$). Ainsi le code BCD n'utilise que 10 **mots de codes** de 4 bits. Par exemple la représentation du nombre $1995_{(10)}$ est : $(0001\ 1001\ 1001\ 0101)_{(BCD)}$. Il est possible d'effectuer des opérations arithmétiques en BCD, mais celles-ci sont plus complexes qu'en binaire classique. Ce code est pondéré avec les poids 1, 2, 4, 8, 10, 20, 40, 80, 100, ...

Codes non pondérés

Dans le cas des codes non pondérés, il n'y a pas de poids affecté à chaque position des symboles. On convient simplement d'un tableau de correspondance entre les objets à coder et une représentation binaire. De tels codes peuvent néanmoins parfois posséder des propriétés arithmétiques intéressantes, comme le code **excédent 3**.

Code excédent 3 ou excess 3

Le code excédent 3 utilise, tout comme le code BCD, 10 mots de codes, auxquels on fait correspondre les 10 chiffres décimaux.

$N_{(10)}$	$N_{(XS3)}$
0	0 0 1 1
1	0 1 0 0
2	0 1 0 1
3	0 1 1 0
4	0 1 1 1
5	1 0 0 0
6	1 0 0 1
7	1 0 1 0
8	1 0 1 1
9	1 1 0 0

tableau 5 : code excédent 3

Il est obtenu en décalant le code binaire de trois lignes vers le haut. Ce code peut être intéressant pour effectuer des soustractions car le complément à 1 de la représentation binaire d'un chiffre correspond au complément à 9 de ce chiffre. Ainsi, toute opération de soustraction se ramène à une addition.

Par exemple $5_{(XS3)} = 1000$, son complément à 1 est obtenu par inversion des bits, $\overline{5}_{(XS3)} = 0111 = 4_{(XS3)}$, le résultat en excédent 3 est le nombre 4, qui est le complément à 9 de 5 en décimal. Ainsi, pour faire une soustraction, il suffit d'ajouter le complément à 1 du nombre à retrancher, puis 1. Par exemple, $(7-5)_{(XS3)} = 7_{(XS3)} + \overline{5}_{(XS3)} + 1_{(XS3)} = 1010 + 0111 + 0100 = 0101 = 2_{(XS3)}$.

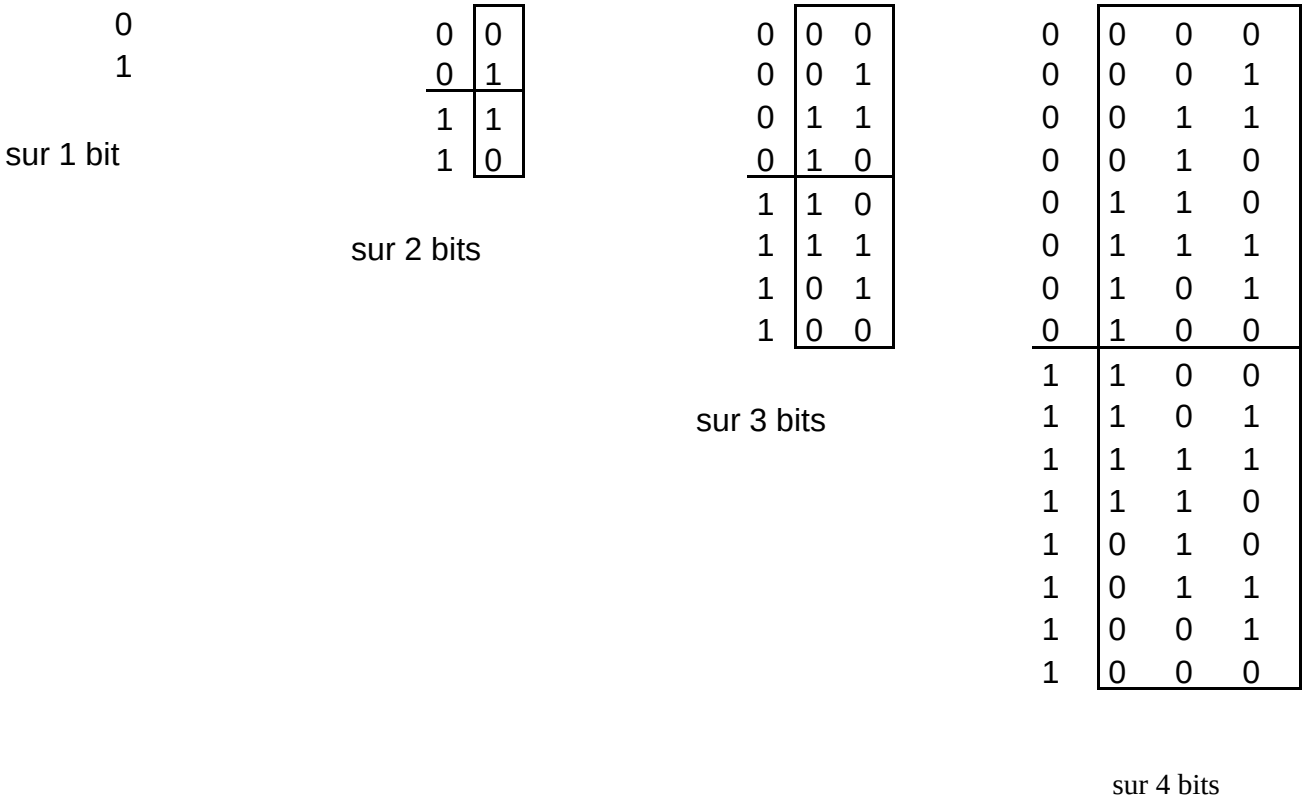
Le code excédent 3 ne présente pas d'intérêt en addition.

Code binaire réfléchi ou code de Gray

Ce code numérique n'étant pas pondéré, il est peu employé pour les opérations arithmétiques. Il est, par contre, utilisé pour le codage des déplacements angulaires, linéaires ou pour la réalisation des tableaux de Karnaugh (cf. chapitre « Propriétés des variables et fonctions logiques »). La propriété principale de ce code est que **deux mots successifs du code ne diffèrent que par un élément binaire**. Ceci permet, d'une part d'éviter la génération d'aléas (états parasites) au passage de deux combinaisons successives, et d'autre part de tirer parti de cette adjacence du codage pour simplifier les fonctions.

L'appellation "binaire réfléchi" provient de sa technique de construction : on peut construire un code de Gray sur *n* bits à partir d'un code de Gray sur *n*-1 bits en procédant comme suit : on copie les mots du code de départ, précédés d'un 0, suivis des mots du même code, pris dans **l'ordre inverse** et précédés d'un 1. Ceci permet de construire un code de Gray de n'importe quel format (cf. figure 1.7).

figure 1.7 : Construction du code de Gray sur 1, 2, 3, et 4 bits



Codes redondants

Il existe un ensemble de codes conçus pour pouvoir détecter, voire corriger des erreurs dans des messages binaires. Leur principe repose sur l'insertion de données redondantes dans l'information initiale. Leur étude approfondie relève du domaine des communications numériques, et ne sera pas traité dans ce cours. Nous citerons cependant quelques exemples simples de codes redondants, les codes *p* parmi *n* et les codes de contrôle de parité.

(a) Code *p* parmi *n*

Ce code est constitué de $\mathbf{C}_n^p = \frac{n!}{p!(n-p)!}$ mots de code. Chaque mot de code est codé sur *n* bits et contient exactement *p* "1" et (*n* - *p*) "0". Par exemple, le code 2 parmi 5 (tableau 6) est constitué de 10 mots de codes et permet de coder les chiffres décimaux.

$N_{(10)}$	$N_{(2\ parmi\ 5)}$
0	0 0 0 1 1
1	1 1 0 0 0
2	1 0 1 0 0
3	0 1 1 0 0
4	1 0 0 1 0
5	0 1 0 1 0
6	0 0 1 1 0
7	1 0 0 0 1
8	0 1 0 0 1
9	0 0 1 0 1

tableau 6 : code 2 parmi 5

L'utilisation de ce code permet, à la réception d'une information, de vérifier par comptage du nombre de 1 si une erreur s'est introduite dans l'information transmise. Dans le cas où plus d'une erreur s'est glissée dans un mot de code, la détection n'est pas assurée dans tous les cas. Ce code ne permet pas non plus de trouver la place de l'erreur, donc de la corriger. D'autre part, le décodage des combinaisons est particulièrement simple, car il ne porte que sur 2 bits par combinaison.

(b) Code contrôle de parité

Le codage d'un mot de *n* bits par contrôle de parité consiste à y adjoindre un $(n+1)^{ème}$ bit dont le rôle est de rendre systématiquement pair ou impair le nombre total de 1 contenus dans l'information codée. Si une erreur se glisse dans l'information, le nombre de 1 devient impair et l'erreur est détectée. Ce code ne permet pas non plus de corriger les erreurs.