ARITHMÉTIQUE : SYSTÈMES DE NUMÉRATION

Une quantité d'objets est représentée par un nombre. L'homme, en utilisant ses dix doigts (digit) pour compter, est venu à utiliser la numération en base 10 ou numération décimale. On utilise aussi de manière épisodique la base 12 (les douzaines) la base soixante (heures, minutes et secondes).
Les informaticiens utilisent les bases **2**, **16** et **8**.

I. <u>LA NUMÉRATION DÉCIMALE</u>

Elle repose sur 3 codes :

- Le **premier code** consiste à adopter des **graphismes** pour représenter des quantités différentes. Les graphismes qui se sont généralisés pour nous constituent les chiffres arabes :{**0, 1, 2, 3, 4, 5, 6, 7, 8, 9**}.
- Le **second code** porte sur la **position** de chacun de ces chiffres. En effet, ces graphismes, alignés à la suite les uns des autres, seront affectés d'un poids en fonction de leur rang (le poids le plus faible étant affecté au rang le plus à droite). Dans l'écriture du nombre 22, le 1er 2 n'a pas le même sens que le second.
- Le **troisième code** détermine la **base** de numération.

Il faut faire la distinction entre chiffre et nombre. Un chiffre est un graphisme, le nombre représente un rapport ou une quantité et est constitué d'un ou plusieurs chiffres.

Dans le cas présent, il s'agit du système à base 10.

<u>Par exemple</u>, le nombre 2048 (deux mille quarante-huit) exprimé en base 10 est tel que :
2048 = 2 milliers + 0 centaine + 4 dizaines + 8 unités = $2*10^3 + 0*10^2 + 4*10^1 + 8*10^0$
(Rappel : $b^0 = 1$ pour toute valeur de b non nulle).

Dans le cas des nombres décimaux par exemple 12,74, on a :
12,74 = 1 dizaine + 2 unités + 7 dixièmes + 4 centièmes = $1*10^1 + 2*10^0 + 7*10^{-1} + 4*10^{-2}$
La **virgule** décimale se place entre les puissances positives et les puissances négatives de la base.
Cette limite **sépare la partie entière de la partie fractionnaire**.

Le déplacement de cette virgule d'un rang vers les puissances positives de la base, correspond à une division du nombre par 10 (la base).
A l'opposé, le déplacement de la virgule d'un rang vers les puissances négatives, correspond à une multiplication du nombre par 10 (la base).

Cette numération de position distribue un **poids** à chaque rang : on la dit **pondérée.**

Le système de numération à base 10 est un cas parmi bien d'autres, car nous pouvons utiliser d'autres bases pourvu que celle que l'on choisit soit un nombre entier au moins égal à **2**.
Par conséquent, pour interpréter correctement un nombre il faut connaître sa base. On indique donc en indice la base employée.
<u>Exemples :</u> $\mathbf{1024_{10}}$ représente le nombre mille vingt quatre en base **10**.
$\mathbf{1000_2}$ représente le nombre **un, zéro, zéro, zéro** en base **2** (ce nombre correspond à **8** en base **10**).

Dans la vie courante, nous n'utilisons pratiquement que des nombres en numération décimale et, de ce fait, l'indice précisant la base disparaît.
Il faut noter également qu'un nombre représenté dans un autre système (autre que la base 10), ne doit pas être prononcé de la même manière, mais en énumérant, du poids le plus fort vers le poids le plus faible, chaque chiffre ou graphisme constituant ce nombre.

II. LA NUMÉRATION BINAIRE

Définition : Ce système est aussi un système à position, il reprend les mêmes règles que la numération décimale. La numération en base **2** est le système le plus adapté aux machines électroniques : les deux symboles 1 et 0 correspondent à deux états : le courant passe ou ne passe pas.
Comme son nom l'indique, il est fondé sur deux valeurs représentées par les graphismes: {**0**, **1**}.

La représentation d'un nombre en base 2 suit le même principe qu'en base 10.

A. Transposition de la base 2 vers la base 10

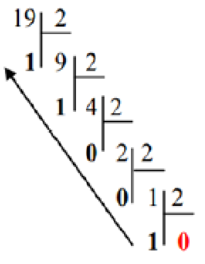
On applique la formule !

Exemple **:**
$\mathbf{10011_2 = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 19_{10}}$

B. Transposition de la base 10 vers la base 2

$29 = 2*14 + 1$
$= 2\ (2*7) + 1$
$= 2\ (\ 2(\ 2*3 + 1) + 1$
$= 2(2(2(2 + 1) + 1) + 1$
$= 2^4 + 2^3 + 2^2 + 1$
$= 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
$= 11101_2$

La méthode est efficace, et c'est la plus proche de la définition de la numération en base 2. Elle est cependant un peu longue. On opère par divisions successives par 2.



On peut plus simplement présenter ces divisions sous forme de tableau :

	29	1
division entière de 29 par 2 –>	14	0
division entière de 14 par 2 –>	7	1
	3	1
	1	1
Arrêt lorsque le résultat est 0	0	

$19_{10} = 10011_2$

C. Opérations en base 2

Les opérations en base 2 ne posent pas de problème particulier, il suffit de savoir que 1+1 = 10 (en base 2) et 1 + 1 + 1 = 11.

Addition :

$$
\begin{array}{lcccccccc}
\text{Retenues :} & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\
 & & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
+ & & & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0
\end{array}
$$

Multiplication :

$$
\begin{array}{cccccccccc}
 & & & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
 & \text{x} & & & & & & 1 & 0 & 1 \\
\hline
 & & & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
+ & & & & & & & & 0 & \\
+ & 1 & 1 & 0 & 1 & 0 & 1 & 1 & & \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1
\end{array}
$$

III. IMPLÉMENTATION DES ENTIERS EN MACHINE

A. Codage des nombres en informatique

En informatique, les informations sont codées sous forme d'octets, un octet = un nombre binaire de 8 chiffres.
Un byte est la plus petite unité de mémoire adressable.
Dans les ordinateurs actuels, 1 Byte = 1 Octet = 8 bits.
On comprend donc l'intérêt de pouvoir représenter simplement le contenu d'une zone mémoire :
L'octet : 11010011 se transcrit en Hexadécimal par D3. (voir plus loin)

B. Les multiples de l'octet

Usuellement on utilise Kilo-Octets, Méga-Octets. A ce sujet, on fait en réalité une erreur de vocabulaire car 1 kilo-Octet = 1024 octets et non 1000 octets comme le veulent les règles du système international (SI) des unités de mesure.

Nom officiel	Nom usuel	Symbole	Valeur
kibioctet	*kilo-octet*	Ko	2^{10} octets = 1024 octets
mébioctet	*méga-octet*	Mo	2^{20} octets = 1024 Ko = 1 073 741 824 octets
gibioctet	*giga-octet*	Go	2^{30} octets = 1024 Mo = 1 099 511 627 776 octets
tébioctet	*téra-octet*	To	2^{40} octets = 1024 Go
pébioctet	*péta-octet*	Po	2^{50} octets = 1024 To
exbioctet	*exa-octet*	Eo	2^{60} octets = 1024 Po
zébioctet	*zetta-octet*	Zo	2^{70} octets = 1024 Eo
yobioctet	*yotta-octet*	Yo	2^{80} octets = 1024 Zo

C. Implémentation des entiers positifs

Les entiers sont généralement codés sur 1, 2, 4, 8 ou 16 octets. Le choix de la taille est un choix du concepteur du programme et dépend du langage de programmation utilisé.
Sur 1 octet, il est possible de coder de 00000000 à 11111111 c'est-à-dire de 0 à 255 soit 256 valeurs.
Sur 2 octets de 0 à $2^{16} - 1 = 65536_{10}$
Sur 4 octets de 0 à $2^{32} - 1 = 4\,294\,967\,296_{10}$

Quelle que soit la taille de stockage choisie, on est toujours confronté au problème du dépassement de capacité (over flow).

Sur un octet : $255 + 1 = 11111111_2 + 1_2 = 100000000$ qui est alors codé sur 9 bits. Avec un codage sur 8 bits on arrive donc au résultat surprenant (et très gênant) 255 + 1 = 0 !

D. Implémentation des entiers signés (négatifs ou positifs)

Pour manipuler les négatifs, on a été tenté de garder le bit de poids fort (le premier chiffre) pour le signe, ainsi 00000001 représente +1 et 10000001 représente -1, et l'on code ainsi le entiers de $-(2^7 - 1)$ à $2^7 - 1$ soit de -127 à 127, le 0 étant alors codé 00000000 ou 10000000.

De plus, avec cette représentation, les opérations binaires classiques ne peuvent plus se faire simplement. Finalement l'implémentation choisie est celle du **complément à 2**.

Exemple sur 8 bits
Cette méthode consiste à représenter un entier relatif par un entier naturel.
Si on utilise des mots de 8 bits, on ne peut représenter que les nombres compris entre -2^7 et $2^7 - 1$ (donc entre -128 et 127)

- si le nombre x est compris entre 0 et $2^7 - 1$ on le représente "normalement"
- si le nombre x est compris entre -2^7 et 0 on le représente par la valeur binaire de $x + 2^8$, c'est à dire x + 256

Pratiquement
Pour connaître la représentation d'un nombre négatif par le complément à 2 :
Pour coder (-4):
On prend le nombre positif 4 écrit en binaire : 0000 0100
On inverse tous les bits : 1111 1011
On ajoute 1: + 1
Le codage de -4 est alors 1111 1100

Inversement
Lorsque l'on connaît la représentation d'un nombre par le complément à 2,

- si le bit de poids fort est 0, il s'agit d'un nombre positif :
  x représenté par 0001 0011 donc x = 19
- Si le bit de poids fort est 1, il s'agit d'un nombre négatif, il faut alors prendre son complément à 2 :
  x représenté par 1001 1011
  Complément à 2 : 0110 0101 = 101
  donc x = -101

Sur n bits

Si on utilise des mots de n bits, on ne peut représenter que les nombres compris entre -2^{n-1} et $2^{n-1}-1$

- si le nombre x est compris entre 0 et $2^{n-1}-1$ on le représente "normalement"
- si le nombre x est compris entre -2^{n-1} et 0 on le représente par la valeur binaire de $x+2^n$

Les opérations sur les entiers relatifs

Avec la représentation par le complément à 2 des entiers signés, les opérations en binaire se font alors simplement.
5 – 4 = 5 + (-4) = 0000 0101 + 1111 1100 = 1 0000 0001 **sur 9 bits** donc le résultat sur 8 bits sera 0000 0001 = 1

IV. LA NUMÉRATION HEXADÉCIMALE

L'utilisation du système binaire est particulièrement délicate (risque d'erreurs) et la conversion systématique en décimal est lourde.
Les informaticiens utilisent le système Hexadécimal (base 16), qui est relativement simple, de même que le passage de la base 2 à la base 16. Le système hexadécimal est comme le binaire et le décimal un système de numération de position pondéré.
Il nécessite 16 symboles : **{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }**
$A_{16} = 10_{10}$; $B_{16} = 11_{10}$; $C_{16} = 12_{10}$; $D_{16} = 13_{10}$; $E_{16} = 14_{10}$; $F_{16} = 15_{10}$.
Le nombre $B2_{16} = B*16^1 + 2*16^0 = 11*16 + 2 = 178_{10}$
Ce nombre $178_{10} = 10110010_2$

A. Transposition Hexadécimal – Décimal

On applique la formule !
$5AB_{16} = 5*16^2 + 10*16^1 + 11*16^0 = 5*256 + 10*16 + 11*1 = 1451_{10}$

B. Transposition Décimal – Hexadécimal

De même qu'en binaire nous divisions successivement par 2, nous divisons ici par 16
$178 = 16*11 + 2$ donc $178_{10} = B2_{16}$
$235 = 16*14 + 11$ donc $235_{10} = EB_{16}$

C. Transposition Binaire Hexadécimal

$10110010_2 = 1011\ 0010_2$ (regroupé par bloc de quatre chiffres à partir de la droite)
on a $1011_2 = 8 + 2 + 1 = 11_{10} = B_{16}$ et $0010_2 = 2_{10} = 2_{16}$
On a alors : $1011\ 0010_2 = B2_{16}$
On peut donc transposer des nombres de la base 2 à la base 16 et réciproquement sans passer par la base 10.

D. Transposition Hexadécimal Binaire

Le principe est symétrique !
$5B_{16}$
$5 = 2^2 + 1 = 101_2$
$B = 11 = 2^3 + 2^1 + 1 = 1011_2$
$5B_{16} = 0101\ 1011_2$

E. Additions en Hexadécimal

Elles suivent le même principe qu'en binaire et en décimal (attention une retenue quand la somme de deux chiffres est supérieure ou égale à… 16 !)

Ex :

$$
\begin{array}{ccccc}
 & ^{1} & & ^{1} & \\
 & 2 & A & 3 & B_{16} \\
+ & 5 & 8 & 9 & C_{16} \\
\hline
= & 8 & 2 & D & 7_{16}
\end{array}
$$

F. Tableau de transposition Décimal/Binaire/ Hexadécimal

Décimal	Binaire	Hexadécimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

V. LES NOMBRES FRACTIONNAIRES EN BASE 2

Comme dans le système décimal, les nombres fractionnaires doivent être représentés. La partie entière est séparée de la partie fractionnaire par une virgule (les anglo-saxons utilisent un point).

Exemple :
En base 10 : $213{,}45 = 2 * 10^2 + 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2}$
En base 2 : $1101{,}01_2 = 2^3 + 2^2 + 2^0 + 2^{-2} = 8 + 4 + 1 + 0{,}25 = 13{,}25_{10}$

Les exposants négatifs correspondent à des fractions :
$2^{-1} = 1/2 = 0{,}5$
$2^{-2} = 1/4 = 0{,}25$
$2^{-3} = 1/8 = 0{,}125$,
… d'où l'appellation de **nombres fractionnaires**.

Transposition des nombres décimaux en binaires fractionnaires base 2

Exemple 1 **$13{,}375_{10}$ à transposer en base 2**
$13{,}375_{10} = 13_{10} + 0{,}375_{10}$
$13_{10} = 1101_2$
Transposons $0{,}375_{10}$ en base 2 par multiplications successives par 2

partie entière	partie fractionnaire	
0	375	
0	75	$0{,}375 \times 2 = 0{,}75$
1	5	$0{,}75 \times 2 = 1{,}5$
1	0	$0{,}5 \times 2 = 1{,}0$

Chaque ligne se déduit de la précédente en multipliant la partie fractionnaire par 2.
On s'arrête lorsque la partie fractionnaire est nulle.

$0{,}375_{10} = 0{,}011_2$

$13{,}375_{10} = 13_{10} + 0{,}375_{10} = 1101{,}011_2$

Exemple 2 **$7{,}325_{10}$ à transposer en base 2**
$7{,}325_{10} = 7_{10} + 0{,}325_{10}$
$7_{10} = 111_2$
Transposons $0{,}325_{10}$ en base 2 par multiplications successives par 2

partie entière	partie fractionnaire	
0	325	
0	65	$0{,}325 \times 2 = 0{,}65$
1	3	$0{,}65 \times 2 = 1{,}3$
0	6	$0{,}3 \times 2 = 0{,}6$
1	**2**	$0{,}6 \times 2 = 1{,}2$
0	4	$0{,}2 \times 2 = 0{,}4$
0	8	$0{,}4 \times 2 = 0{,}8$
1	6	$0{,}8 \times 2 = 1{,}6$
1	**2**	$0{,}6 \times 2 = 1{,}2$

Chaque ligne se déduit de la précédente en multipliant la partie fractionnaire par 2.

Cette fois on retrouve la même configuration : 1,2.
En poursuivant la multiplication, on retrouvera indéfiniment les mêmes résultats.
L'écriture fractionnaire en base 2 de $0{,}325_{10}$ est donc :

0,010**1001**1001**1001**1001… que l'on note $0{,}010\overline{1001}$

$0{,}325_{10} = 0{,}010\overline{1001}_2$

$7{,}325_{10} = 7_{10} + 0{,}375_{10} = 111_2 + 0{,}0101001_2$

Important : On voit donc ici qu'un nombre aussi simple que $7{,}235_{10}$ ne peut être représenté exactement en machine puisqu'en base 2 il s'écrit avec une infinité de chiffres !
Les conséquences sont multiples et sources de nombreuses erreurs de programmation.

VI. LA REPRÉSENTATION MACHINE DES NOMBRES RÉELS (*Hors programme*)

En **mathématiques**, considère plusieurs ensembles de nombres : les entiers naturels ℕ,
les entiers relatifs (ou entiers signés) ℤ, les décimaux *ID*: les nombres (positifs ou négatifs) qui s'écrivent avec un nombre fini de chiffres après la virgule, les rationnels ℚ : les nombres qui peuvent s'écrire sous forme de fraction, et les réels ℝ. Ils sont tous inclus les uns dans les autres.
En **informatique**, la place disponible pour stocker un nombre est par nature limitée (de 1 à plusieurs octets), et l'on perçoit vite les difficultés que l'on va rencontrer à exprimer des nombres très grands, ou simplement des nombres qui s'expriment avec trop de chiffres. Le nombre π est sans doute le plus célèbre d'entre eux.
Les calculs sur ces nombres sont alors par essence inexacts dans le sens où l'on ne peut représenter ces nombres en machine dans leur intégralité.

- **La représentation des nombres en virgule flottante.**

Rappel : la notation scientifique
La notation scientifique d'un nombre est l'écriture de ce nombre en utilisant les puissances de 10.

La représentation à virgule flottante
Les nombres non entiers (float par exemple) sont représentés en machines par leur valeur en binaire en codant, séparément, le signe, l'exposant et la mantisse.
Le signe est toujours codé sur un bit, le bit de poids fort.
Le nombre de bits réservé l'exposant et à la mantisse dépendent des choix de conceptions.

Sur 32 bits, avec la norme IEE754 on a :
· 1 bit pour le signe,
· 8 bits pour l'exposant,
· 23 bits pour la mantisse.
· le signe est 0 pour les positifs et est 1 pour les négatifs,
· Sur les 8 bits de l'exposant on code l'exposant (négatif ou positif) par un décalage de 2^7-1 = 127.
C'est-à-dire que l'exposant -25 sera codé par le nombre -25 + 127 = 102
Sur les 8 bits de l'exposant on peut coder des entiers de 0 à 255 donc des exposants de 0 – 127 à 255 – 127 soit des nombres dont l'exposant va de -127 à 128. En fait, l'exposant -127 et 128 sont réservés à des nombres particuliers.
· Sur les 23 bits de la mantisse on ne code que les chiffres après la virgule puisqu'en base 2 le chiffre avant la virgule est toujours 1

Exemple :
Pour coder x = -2.75_{10}

- x est négatif, le bit de poids fort est 1
- $2.75_{10} = 10.11_2 = 1.011 * 2^1$ (l'exposant est 1)
- l'exposant sera codé par 1+127 = 128 = 1000 0000
- la mantisse a = 1.011 sera codée par 01100000000000000 (le 1 avant la virgule n'est pas indiquée)
- le nombre x est donc codé sur 32 bits norme IEEE754 par 1 1000 0000 0110 0000 0000 0000 0000 000

Inversement le nombre codé : 0 01111100 01000000000000000000000 :

- le signe est 0, le nombre est positif,
- l'exposant est codé par 01111100 = 124_{10} l'exposant e = 124 – 127 = -3,
- la mantisse m est codée 01000000000000000000000 ce qui donne, puisque le 1 avant la virgule n'est pas indiqué, a = 1.012 = 1.2510 (1×20 + 0×2−1 + 1×2−2)
  Le nombre représenté est donc $1{,}25\times2^{-3}$ soit +0,15625.