CHAPITRE 2

LA REPRÉSENTATION DES DONNÉES

1. LES SYSTEMES DE NUMÉRATION

Dans la vie de tous jours, nous avons pris l'habitude de représenter les nombres en utilisant dix symboles différents, à savoir les chiffres suivants:

0 1 2 3 4 5 6 7 8 9

Ce système est appelée le système décimal (déci signifie dix). Il existe cependant d'autres formes de numération qui fonctionnent en utilisant un nombre de symboles distincts, par exemple le système binaire (bi: deux), le système octal (oct: huit), le système hexadécimal (hexa: seize).

En fait, on peut utiliser n'importe quel nombre de symboles différents (pas nécessairement des chiffres) dans un système de numération; ce nombre de symboles distincts est appelé la base du système de numération. Le schéma suivant montre les symboles utilisés des principaux systèmes rencontrés.

Tableau 1 :Systèmes de numération

Système	Base	Symboles utilisés
Binaire	2	0 1
Ternaire	3	0 1 2
Octal	8	0 1 2 3 4 5 6 7
Décimal	10	0 1 2 3 4 5 6 7 8 9
Hexadécimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Le système binaire est utilisé en informatique parce qu'il n'a besoin que de deux symboles le rendant tout à fait approprié pour les circuits électriques qui, eux-mêmes, ne présentent généralement que deux états possibles: le circuit est allumé (1) ou éteint (0). On peut aussi les représenter avec les variables logiques VRAI ou FAUX.

Dans le système binaire, les tables d'addition et de multiplication sont très simples. Cela facilite l'implantation des algorithmes de calcul:

Figure 1: Calcul en binaire

Addition Binaire		
	1	
+	0	1
	0	1
	1	0

Multiplication Binaire		
	0	
X	0	1
	0	1
	0	1
0	0	
0	0	1

Les informations manipulées par une machine ne sont pas tous des nombres. Il faut donc donner à tous les caractères utilisés (lettres de l'alphabet, espaces, caractères de contrôle, instructions, etc.), une forme numérique (codification) avant de pouvoir les traduire en binaire.

L'ordinateur ne reconnaît et ne traite que des chaînes binaires. Les principaux systèmes de codification en chaînes binaires seront abordés plus loin. Disons simplement pour l'instant qu'au lieu de coder tous ces caractères sous forme de nombres décimaux, on utilise le système binaire et d'autres systèmes de numération, surtout les systèmes octal et hexadécimal. Ces derniers systèmes sont plus commodes à utiliser, leur base respective étant plus grande que 2, tout en étant un multiple de 2. Ils produisent des chaînes de caractères plus courtes qu'en binaire pour une même quantité d'information, et ces chaînes sont plus faciles à traduire en chaînes binaires que les chaînes correspondantes en notation décimale.

Commençons donc par regarder de plus près ces différents systèmes de numération, et examinons les moyens de passer d'un système à un autre.

1.1. Bases et exposants

Il est très utile de noter que, lorsqu'on utilise le système décimal, on compte de la façon suivante: on énumère tous les symboles possibles, 0, 1, 2, jusqu'à 9. Une fois la liste de symboles épuisée, on ajoute une position à gauche pour former le nombre suivant: 10. La valeur du 1 de 10 est cependant 10 fois plus grande que celle du simple 1 en première position. C'est que le poids associé au symbole diffère selon la position où il est situé. Bref, la notion de dizaine, centaines, milliers, etc. exprime en fait l'exposant que prend la base 10 donnant le poids de la deuxième, troisième, etc. position du chiffre dans la représentation du nombre.

En binaire, le nombre de symboles se limite à deux. Une fois les deux symboles épuisés, il faut déjà ajouter une position à gauche. Ainsi, après 0 et 1, il faut passer à 10 dont le 1 de la deuxième position vaut cependant deux fois plus que le simple 1 de la première position. Une fois les possibilités de deux positions épuisées, une troisième position est ajoutée, et celle-ci vaut deux fois plus que la précédente, et ainsi de suite; puisque la base est 2, le poids associé à chaque position augmente d'un facteur de deux à chaque fois.

On peut indiquer la base d'un nombre en écrivant l'indice correspondant à droite du nombre :

Exemple: 734_8 signifie que 734 est la représentation d'un nombre exprimé en base 8.

Lorsque la base n'est pas indiquée, cela voudra dire, par convention, que le nombre représenté est exprimé en base 10, notre système familier à tous. Le tableau suivant montre la correspondance entre divers systèmes de bases différentes.

Tableau 2 : Valeurs équivalentes exprimées dans les principales bases

BINAIRE	DÉCIMAL	OCTAL	HEXADÉCIMAL
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F
10000	16	20	10
10001	17	21	11
10010	18	22	12
10011	19	23	13
10100	20	24	14
…	…	…	…
11001	25	31	19
11010	26	32	1A
11011	27	33	1B
11100	28	34	1C
11101	29	35	1D
11110	30	36	1E
11111	31	37	1F
100000	32	40	20

On a par exemple, $10011_2 = 19_{10} = 23_8 = 13_{16}$

On peut aussi décomposer un nombre en mettant en évidence les facteurs qui multiplient les différentes puissances auxquelles est élevée la base.

Par exemple, 125_{10} peut se réécrire:

$(1 \times 10^2) + (2 \times 10^1) + (5 \times 10^0) =$
$100 + 20 + 5 = 125$

L'évaluation de l'expression, calculée en base 10, donne la valeur (en base 10) du nombre considéré.

De la même façon, 0.528_{10} peut se réécrire:

$(5 \times 10^{-1}) + (2 \times 10^{-2}) + (8 \times 10^{-3}) =$
$.5 + .02 + .008 = .528$

D'une manière plus générale, un nombre N écrit en base b s'exprime de la manière suivante:

$$\mathbf{N_b = (C_{n-1} \times b^{n-1}) + (C_{n-2} \times b^{n-2}) + ... + (C_1 \times b^1) + (C_0 \times b^0)}$$
$$\mathbf{+ (C_{-1} \times b^{-1}) + (C_{-2} \times b^{-2}) + ... + (C_{-m} \times b^{-m})}$$

où

- N est un nombre
- C un chiffre ou un symbole composant le nombre
- b la base
- n le rang du symbole le plus à gauche du point
- -m le rang du symbole le plus à droite du point

1.2. Conversion d'une base à l'autre

1.2.1 Conversion d'un système de base b au système décimal

On peut aussi décomposer ainsi les nombres de n'importe quelle base en somme de facteurs de la base élevée en puissances. Par exemple, 237_8 peut se réécrire:

$(2 \times 8^2) + (3 \times 8^1) + (7 \times 8^0) = 237_8$

Or, comme on sait que:

$\mathbf{8^2 = 64_{10}}$
$\mathbf{8^1 = 8_{10}}$
$\mathbf{8^0 = 1_{10}}$

On peut faire la conversion suivante:

$$
\begin{aligned}
237_8 &= (2 \times 8^2) + (3 \times 8^1) + (7 \times 8^0) \\
&= (2 \times 64) + (3 \times 8) + (7 \times 1) \\
&= 159_{10}
\end{aligned}
$$

On rappelle que l'évaluation des calculs effectués en base 10 donne l'expression en base 10 du nombre. Le procédé est le même quelque soit la base. Ainsi, pour trouver ce que vaut 11011.1_2 en base 10 on décompose ce nombre comme suit:

$$
\begin{aligned}
1011.1_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) \\
&= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1) + (1 \times 0.5) \\
&= 8 + 0 + 2 + 1 + 0.5 \\
&= 11.5_{10}
\end{aligned}
$$

Exemple 1 : Trouver la valeur en base 10 de $D3F4_{16}$

$$
\begin{aligned}
D3F.4_{16} &= (D \times 16^2) + (3 \times 16^1) + (F \times 16^0) + (4 \times 16^{-1}) \\
&= (13 \times 256) + (3 \times 16) + (15 \times 1) + (4 \times 0.625) \\
&= 3328 + 48 + 15 + 0.25 \\
&= 3391.25_{10}
\end{aligned}
$$

1.2.2 Conversion du système décimal au système de base b

1.2.2.1 Les entiers

Si on divise un nombre décimal, par exemple 125, par 10 (la base) on obtient:

$$\frac{125}{10} = 12 \quad \text{reste } 5$$

On peut également dire que 125 modulo 10 égale 5. Or 5 est le dernier chiffre du nombre 125. Soit ce qui reste lorsqu'on a formé toutes les dizaines possibles. De plus, la partie entière du résultat, c'est-à-dire le nombre de dizaines, peut être obtenue par l'opération 125 DIV 10 .

Si on reprend le résultat de la division, soit 12, et qu'on l'on divise à nouveau par la base, on obtient:

$$\frac{12}{10} = 1 \quad \text{reste } 2$$

Qui pourrait aussi se dire 12 modulo 10 = 2. On constate que 2 est le second chiffre du nombre 125. Divisons encore une fois 1 par 10. On obtient:

$$\frac{1}{10} = 0 \quad \text{reste } 1$$

On s'aperçoit que le reste, 1, est également le premier chiffre de 125, le nombre de départ. Comme le résultat de cette dernière division est 0, toute autre division par 10 par la suite donnera comme résultat 0. Maintenant si on regroupe les restes des divisions, en commençant par le dernier, on obtient 1, 2 et 5, soient les trois chiffres constituant le nombre de départ. Cela s'explique par le fait que chaque chiffre composant le nombre a comme poids un multiple de 10, la base. Puisque:

$$125 = (1 \times 10^2) + (2 \times 10^1) + (5 \times 10^0),$$

Il est normal que les divisions successives par 10 nous donnent 5, 2 et 1 comme restes On peut récapituler les opérations à l'aide d'un tableau:

DÉPART	125	12	1
DIV 10 =	12	1	0
Reste =	5	2	1

Supposons maintenant que nous voulions connaître l'expression en base 8 de 159_{10} . On sait que ce nombre devra se décomposer tel que:

$$159_{10} = \ldots + (C_2 \times 8^2) + (C_1 \times 8^1) + (C_0 \times 8^0)$$

Comme on sait que le nombre de symboles en base 8 est limité à 8, on peut être certain que C_0 sera inférieur à 8, c'est-à-dire un nombre parmi l'ensemble {0, 1, 2, 3, 4, 5, 6, 7}. C'est donc dire que C_0 est le reste de la division du nombre de départ par 8, la base désirée:

$$\frac{159}{8} = 19 \quad \text{reste } 7$$

donc $C_0 = 7$

Par la méthode de divisions successives, on obtient:

DÉPART	159	19	2
DIV 8 =	19	2	0
Reste =	7	3	2

En regroupant les restes, à partir du dernier, on obtient 237_8 soit la valeur de 159_{10} en base 8.

Exemple 2 : Trouver la valeur en base 16 de 3391_{10} :

DÉPART	3391	211	13
DIV 16 =	211	13	0
Reste =	15	3	13

Si on regroupe dans l'ordre inverse les restes obtenus en sachant que dans la base 16, 13 s'écrit D et 15 est représenté par le symbole F, on obtient $3391_{10} = D3F_{16}$.

On peut ainsi trouver l'équivalent d'un entier de base 10 dans n'importe laquelle base b. Il suffit de faire des divisions par b les parties entières obtenues de façon récursive jusqu'à ce qu'on obtienne toujours un zéro comme quotient, et de rassembler ensuite les restes des divisions trouvés dans l'ordre inverse.

Notons que l'on a toujours

$$\mathbf{N_{10} = (N_{10}\ DIV\ b) \times b \quad + \quad N_{10}\ MOD\ b} ,$$

expression utile servant à trouver la représentation en base b du nombre N_{10} par applications successives.

1.2.2.2 Les fractions

Quelle que soit la base utilisée, la quantité représentée est la même; ce ne sont que les symboles avec laquelle cette quantité est représentée qui changent. Comme une fraction est un nombre inférieur à 1 et que le symbole 1 a la même signification dans toutes les bases, la partie fractionnaire d'un nombre dans une base donnée demeure fractionnaire dans toute autre base.

On sait aussi que la partie fractionnaire d'un nombre exprimée en base b peut se réécrire:

$$\mathbf{C_{-1} \times b^{-1} + C_{-2} \times b^{-2} + \quad \ldots \quad + C_{-m} \times b^{-m}}$$

Par exemple, 0.35601_8 peut s'écrire sous la forme

$$(3 \text{ x } 8^{-1}) + (5 \text{ x } 8^{-2}) + (6 \text{ x } 8^{-3}) + (0 \text{ x } 8^{-4}) + (1 \text{ x } 8^{-5})$$

et de même: 0.1250_{10} se réécrit

$$(1 \text{ X } 10^{-1}) + (2 \text{ X } 10^{-2}) + (5 \text{ X } 10^{-3}) .$$

Le problème posé maintenant est de trouver la valeur des coefficients :

C_{-1}, C_{-2}, C_{-3}, ... , C_{-m},

quelque soit la base choisie b. On remarque que si on multiplie la dernière expression par 10 (la base), on obtient:

$(1 \text{ x } 10^{0}) + (2 \text{ x } 10^{-1}) + (5 \text{ x } 10^{-2}) = 1.25_{10}$.

Si on multiplie de nouveau par 10 la nouvelle partie fractionnaire obtenue, soit la fraction 0.25_{10}, on obtient:

$(2 \text{ x } 10^{0}) + (5 \text{ x } 10^{-1}) = 2.5_{10}$.

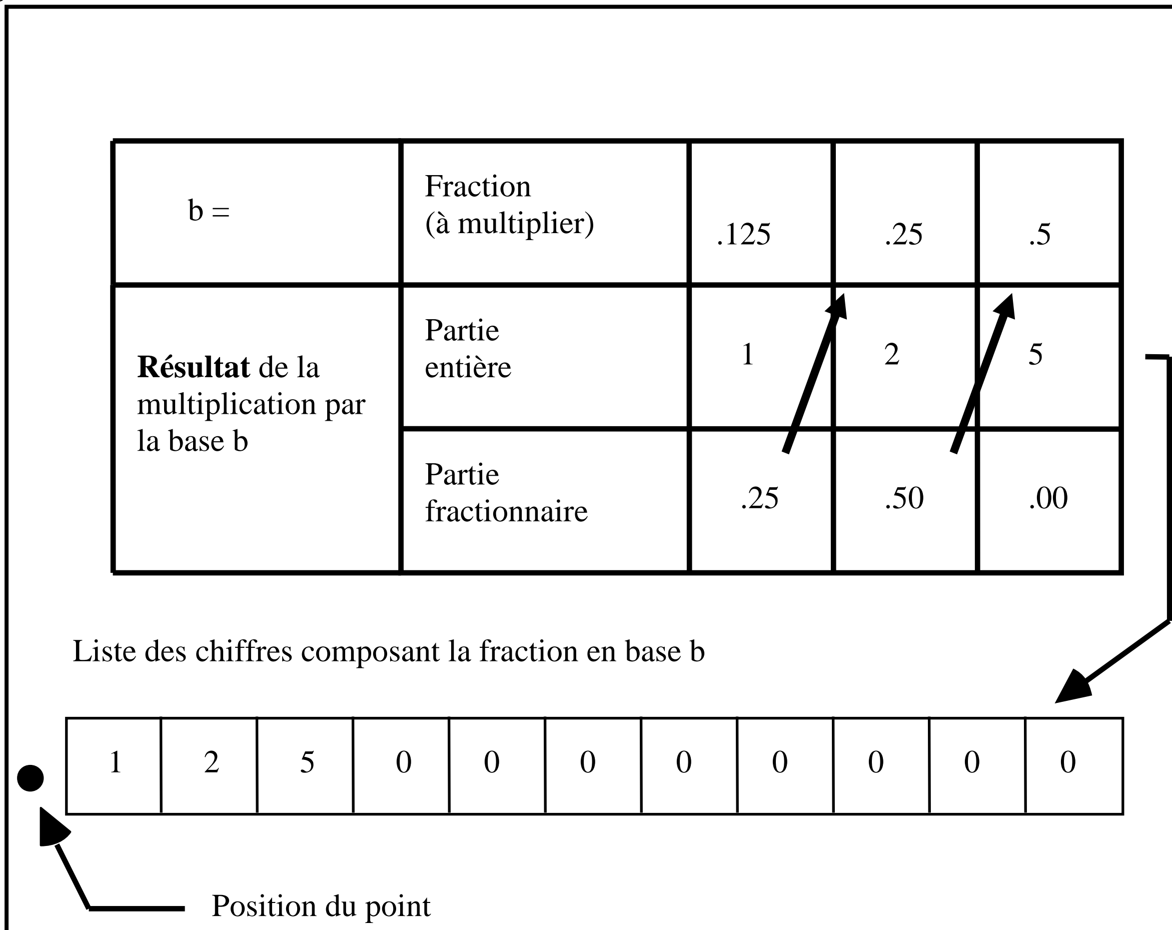
En multipliant de nouveau par 10, la partie fractionnaire de l'expression précédente, soit 0.5_{10}, on obtient:

$(5 \text{ x } 10^{0}) = 5_{10}$.

En regroupant dans l'ordre les parties entières obtenues à chacune des multiplications précédentes (c'est-à-dire 1, 2 et 5), on retrouve les coefficients C_{-1}, C_{-2} et C_{-3}.

On peut ici encore présenter les résultats obtenus, comme on l'a fait pour les coefficients d'indices positifs pour les entiers, dans un tableau:

Figure 2: Transformation de fraction décimale en base b

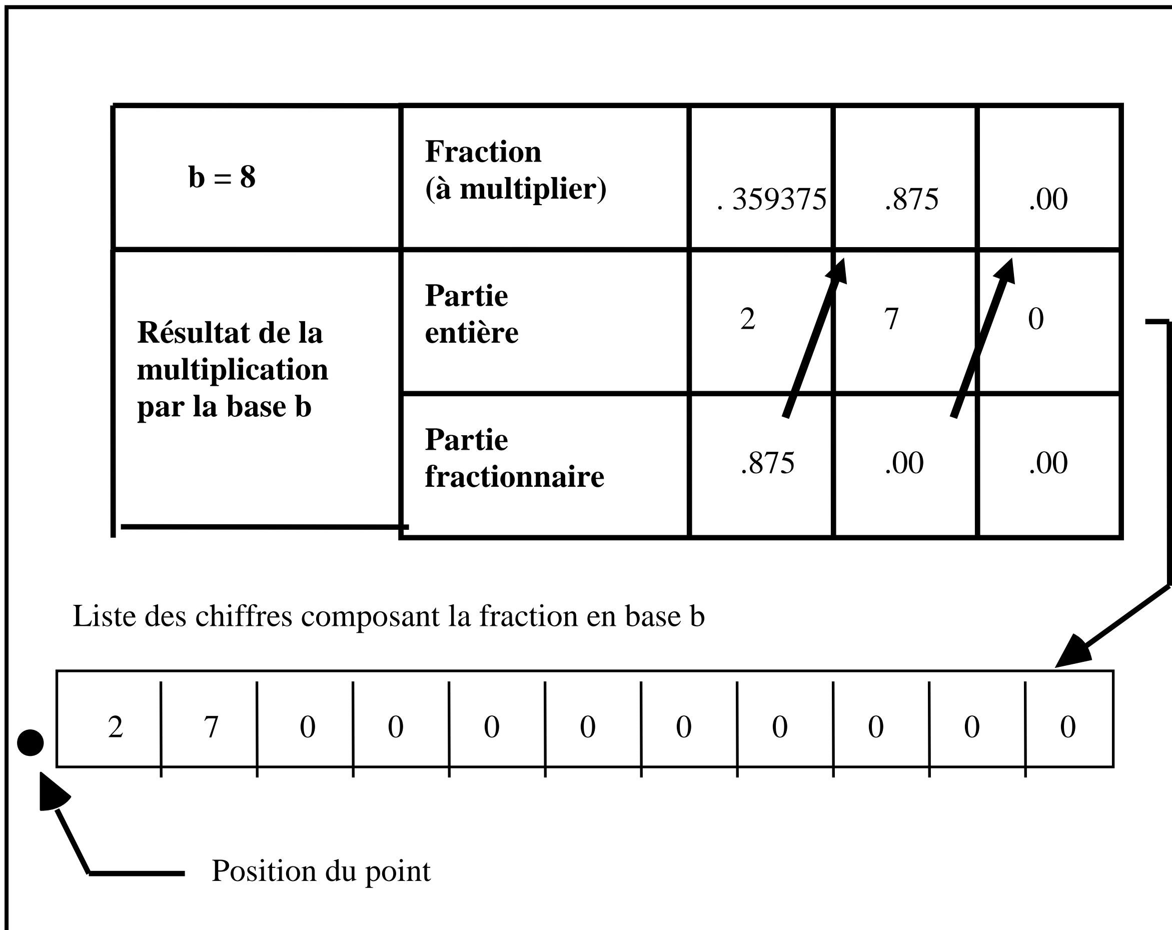


On retrouve, bien sûr, la fraction de départ 0.125_{10}.

Pour trouver la valeur dans une base b quelconque d'un nombre fractionnaire exprimé au départ en base 10, on procède de la même manière, c'est à dire par des multiplications successives de la partie fractionnaire des résultats à chaque étape par b.

Ainsi, pour trouver la valeur en base 8 de 0.359375_{10}, on fait une série de multiplications par 8. Les résultats de ces multiplications successives des parties fractionnaires sont résumés dans le tableau suivant:

Figure 3: Transformation de fraction décimale en base b



Regroupant les entiers obtenus dans la deuxième ligne du tableau, on trouve alors l'égalité $0.359375_{10} = 0.27_8$

Exemple 3 :. Trouver la valeur binaire de 0.3125_{10}:

b = 2	Fraction (à multiplier)	.3125	.625	.25	.50
Résultat De la multiplication par la base b	Partie entière	0	1	0	1
	Partie fractionnaire	.625	.25	.50	.00

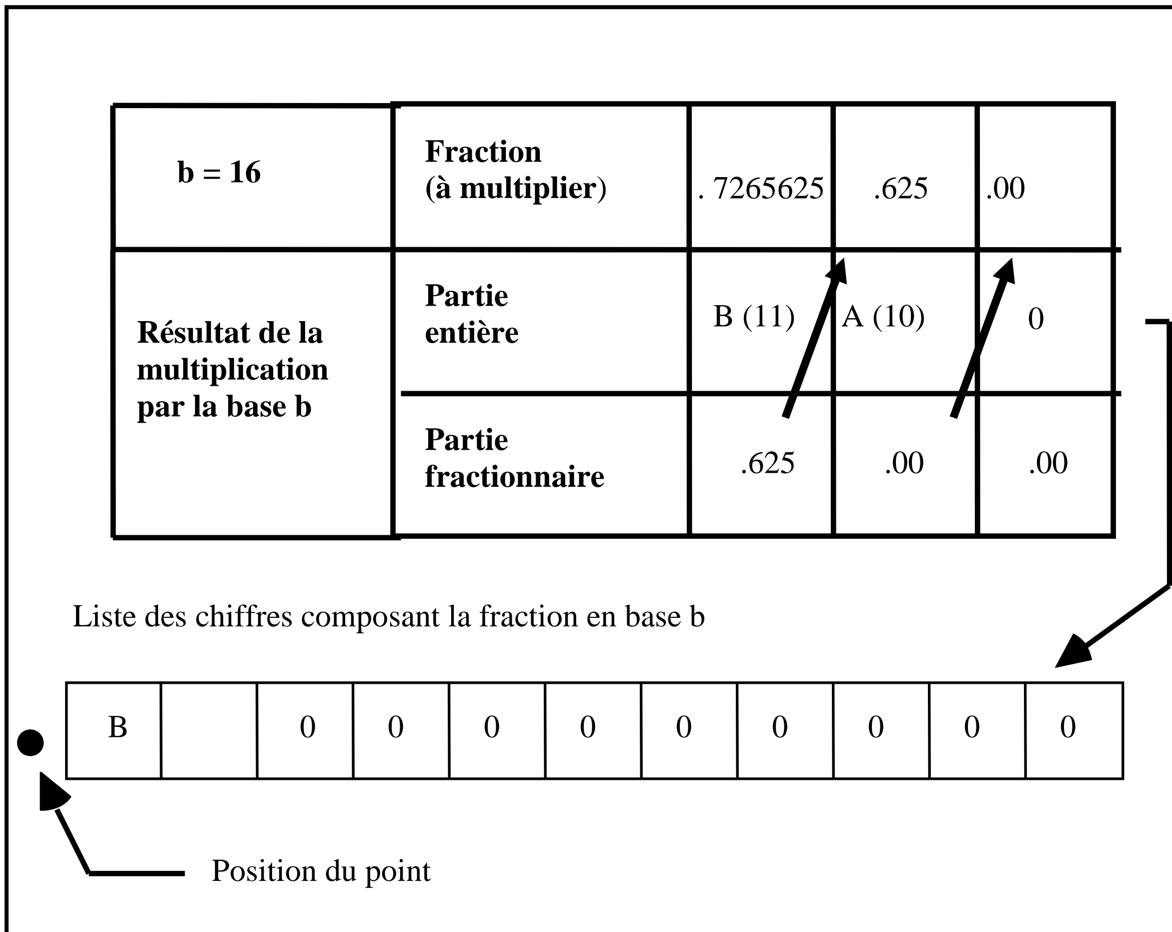
Liste des chiffres composant la fraction en base b

●	0	1	0	1	0	0	0	0	0	0	0

Position du point décimal

donc $0.3125_{10} = 0.0101_2$

Exemple 4 : Trouver la valeur 0.7265625_{10} en hexadécimal:



b = 16	**Fraction (à multiplier)**	. 7265625	.625	.00
Résultat de la multiplication par la base b	**Partie entière**	B (11)	A (10)	0
	Partie fractionnaire	.625	.00	.00

C'est-à-dire $0.7265625_{10} = 0.BA_{16}$

Exemple 5 :. Trouver la valeur octale de 0.1_{10}:

Transformation de fractions décimales en base b

b = 8	.1	.8	.4	.2	.6	.8	.4	.2	.6	.8
entier	0	6	3	1	4	6	3	1	4	6
fraction	.8	.4	.2	.6	.8	.4	.2	.6	.8	.4

Liste des chiffres composant la fraction en base b

0	6	3	1	4	6	3	1	4	6	....

Position du point décimal

On obtient $0.1_{10} = 0.0[6314][6314][6314][...]_8$. On se rend compte que le nombre fractionnaire 0.1_{10}, représenté dans la base 10 par une suite finie de décimales non nulles, est représenté par une suite infinie de décimales non nulles dans la représentation en octal (les chiffres 3146 sont répétés indéfiniment).

Pour trouver la valeur en base b d'un nombre qui comprend à la fois une partie entière et une partie fractionnaire, on utilise les deux méthodes exposées plus haut: d'abord on trouve la valeur de la partie entière en faisant une suite de divisions par la base et en regroupant dans l'ordre inverse les restes obtenus. Ensuite, on convertit la partie fractionnaire en effectuant une suite de multiplications par la base, les parties entières des résultats intermédiaires obtenus successivement nous fournissant les chiffres de la représentation désirée.

1.2.3 Conversion du binaire vers l'octal

Puisque $8 = 2^3$, la base 8 est un multiple de la base 2. Cette constatation simplifie grandement la conversion d'un nombre octal en binaire et vice-versa. En effet, en binaire, 7_8 s'écrit 111_2. C'est donc dire qu'il ne faut jamais plus de trois positions binaires pour représenter un symbole octal, tous forcément plus petits que 7. Mais voyons de plus près la relation qui existe entre un nombre en binaire et son équivalent octal à l'aide d'un exemple.

Soit un nombre binaire quelconque, par exemple 101010_2. Ce nombre peut se réécrire ainsi:

$$101010_2 = (1 \text{ x } 2^5) + (0 \text{ x } 2^4) + (1 \text{ x } 2^3) + (0 \text{ x } 2^2) + (1 \text{ x } 2^1) + (0 \text{ x } 2^0)$$

Comme les exposants des trois premiers membres de l'expression sont supérieurs à 3, on pourrait réécrire l'expression en mettant ($2^3 = 8$) en évidence:

$101010_2 = 2^3$ x [(1 x 2^2) + (0 x 2^1) + (1 x 2^0)] + 1 x [(0 x 2^2) + (1 x 2^1) + (0 x 2^0)]

Comme tout nombre élevé à la puissance 0 donne 1 , on pourrait aussi écrire:

$101010_2 = 8^1$ x [(1 x 2^2) + (0 x 2^1) + (1 x 2^0)] + 8^0 x [(0 * 2^2) + (1 * 2^1) + (0 * 2^0)]

en évaluant les expressions entre parenthèses:

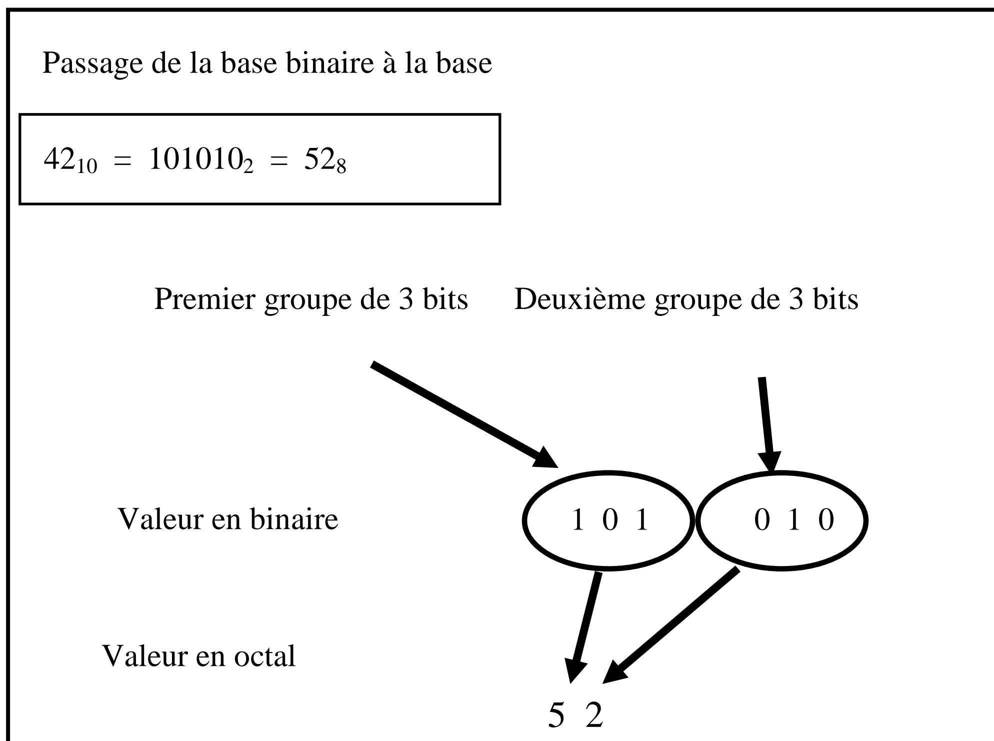
$101010_2 = 8^1$ X [4 + 0 + 1] + 8^0 X [0 + 2 + 0]
= (5 X 8^1) + (2 X 8^0)
= 52_8

Autrement dit, chaque tranche de trois bits d'un nombre binaire correspond à un symbole octal.

101010_2 = $[101]_2$ $[010]_2$ = 52_8

Il suffit donc de séparer le nombre binaire en tranches de trois bits et de trouver la valeur équivalente en décimal de chaque tranche, donnant ainsi le symbole en octal pour chaque tranche.

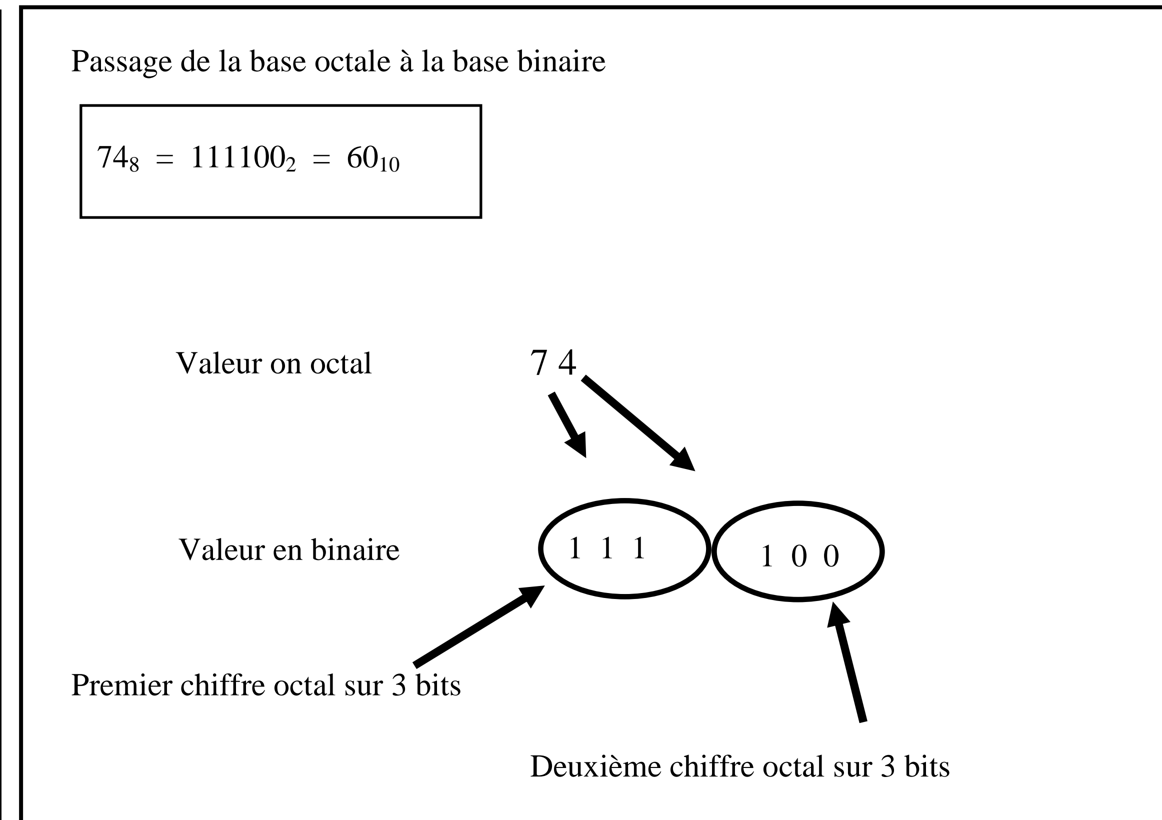
Exemple 6 :



La conversion d'un nombre octal en binaire suit le processus inverse, c'est-à-dire qu'il s'agit de trouver la valeur binaire (sur trois bits) de chaque symbole octal.

On trouve donc $52_8 = 101\ 010_2$

Exemple 7 :

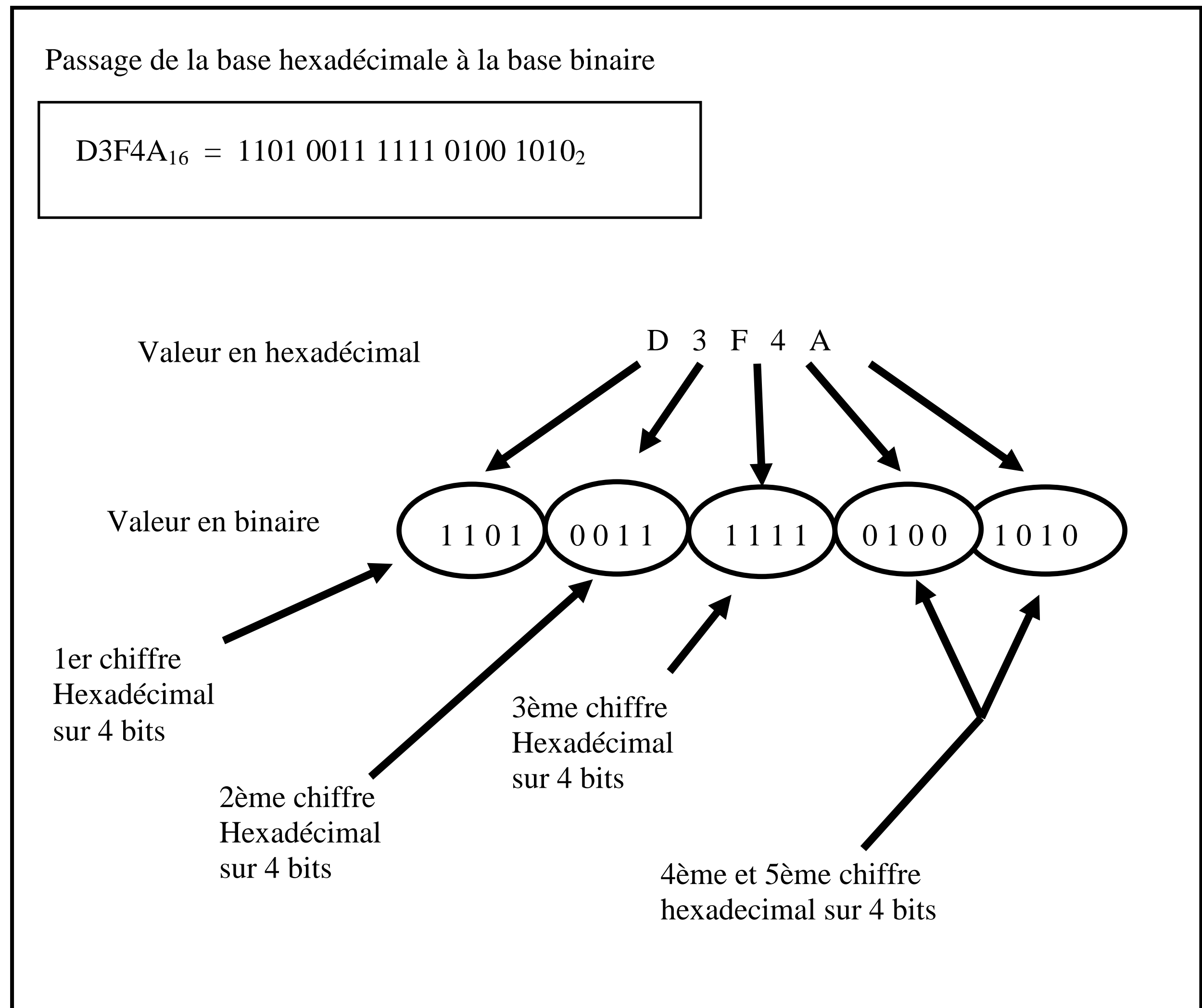


On trouve donc $74_8 = 111\ 100_2$

1.2.4 La conversion du binaire à l'hexadécimal

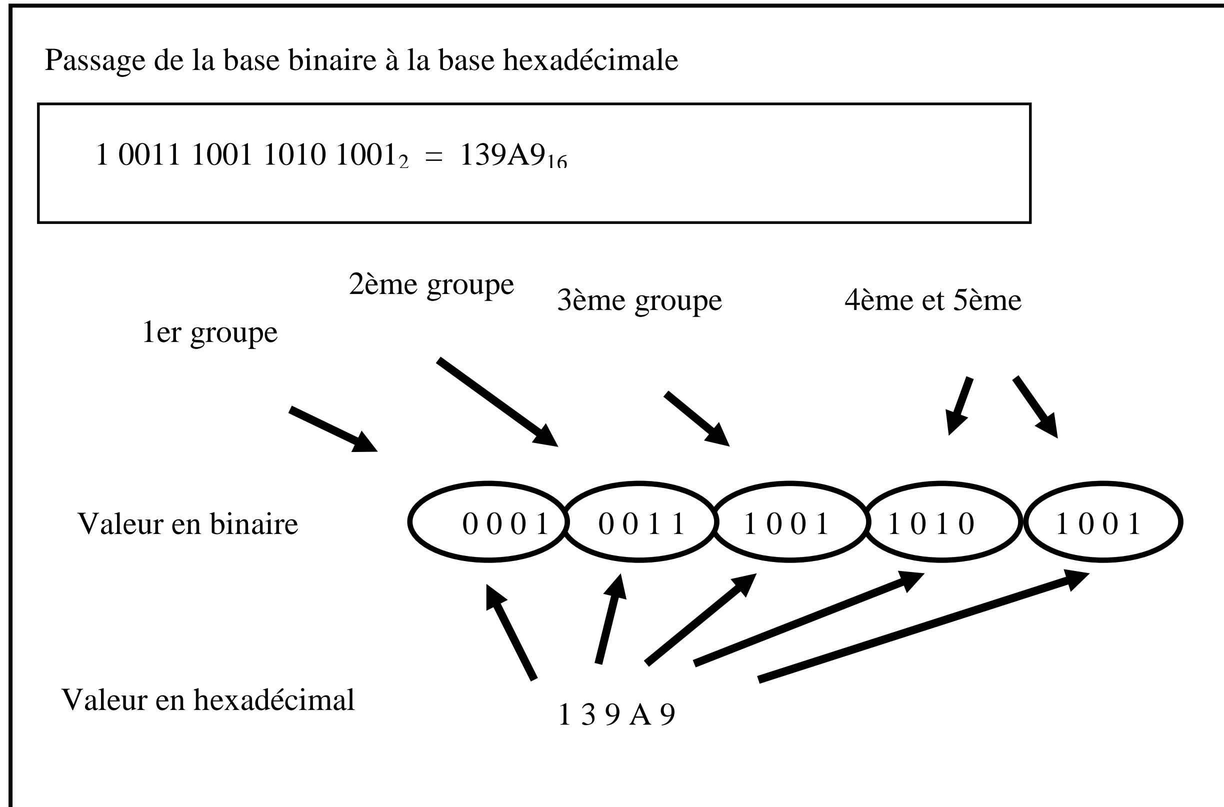
Pour passer du binaire à l'hexadécimal, le principe est le même que pour passer du binaire à l'octal. Cependant, comme $16 = 2^4$, on sépare cette fois-ci le nombre binaire en tranches de 4 bits et on trouve la valeur hexadécimale correspondant à chaque tranche. Inversement, la conversion d'un nombre hexadécimal en binaire se fait en juxtaposant les valeurs binaires (sur 4 bits) de chaque symbole hexadécimal.

Exemple 8 : Trouver la valeur binaire de $D3F4A_{16}$



On trouve donc $D3F.4A_{16} = 110100111111.01001010_2$

Exemple 9 : Exprimer 10011100110101001_2 en hexadécimal.

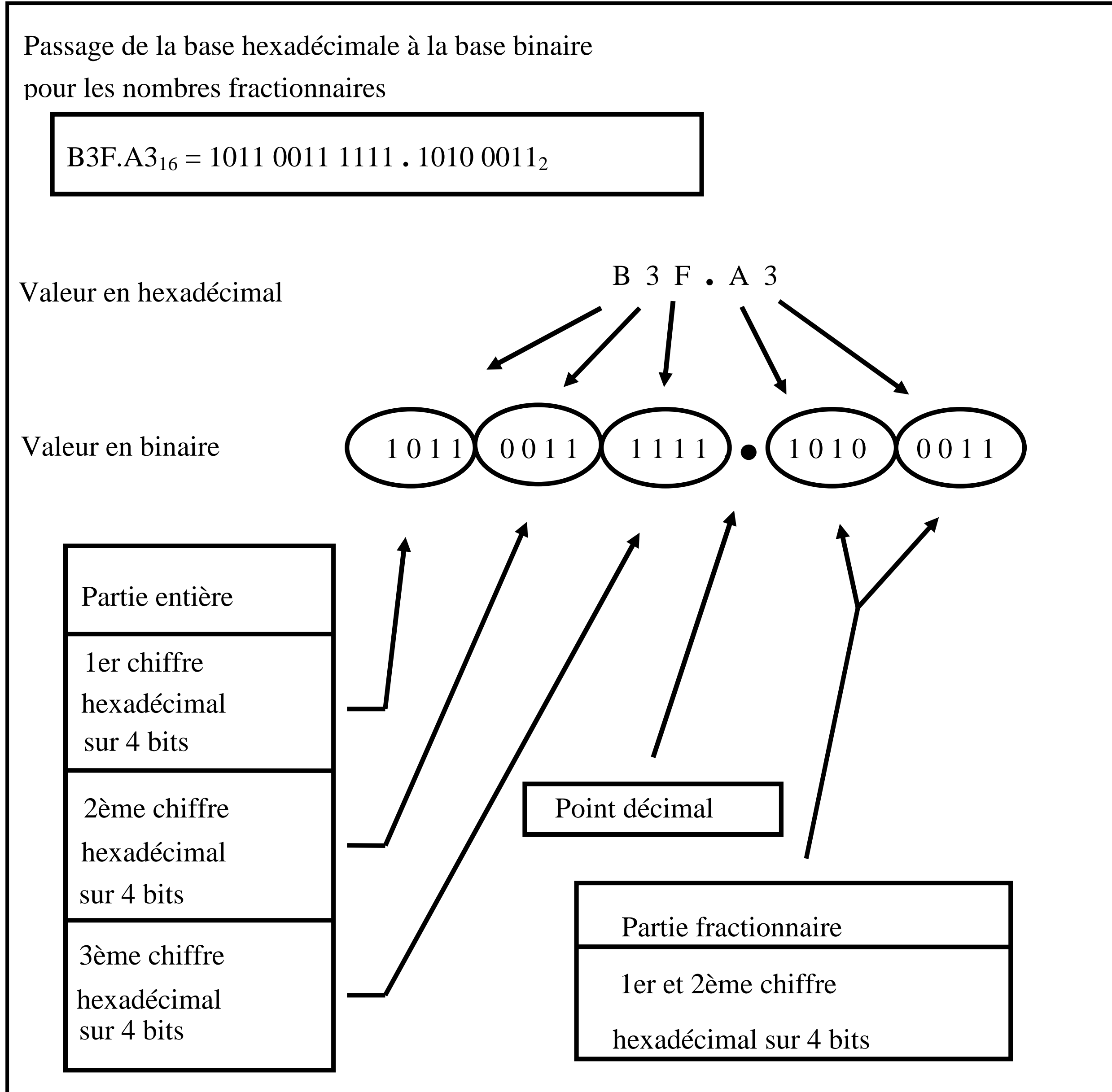


On trouve alors: $10011100110101001_2 = 139A9_{16}$.

Notons que la procédure de passage d'une base puissance 2 à une autre base puissance 2, par exemple de binaire à l'hexadécimal, tient toujours même si **les nombres sont fractionnaires**. On considère alors les tranches de 4 bits du point décimal en allant vers la droite pour la partie fractionnaire du nombre considéré et les tranches de 4 bits en allant vers la gauche pour la partie entière du nombre. Il en est de même pour le passage de la base binaire à la base octale, excepté que les tranches dans ce cas sont de 3 bits au lieu de 4.

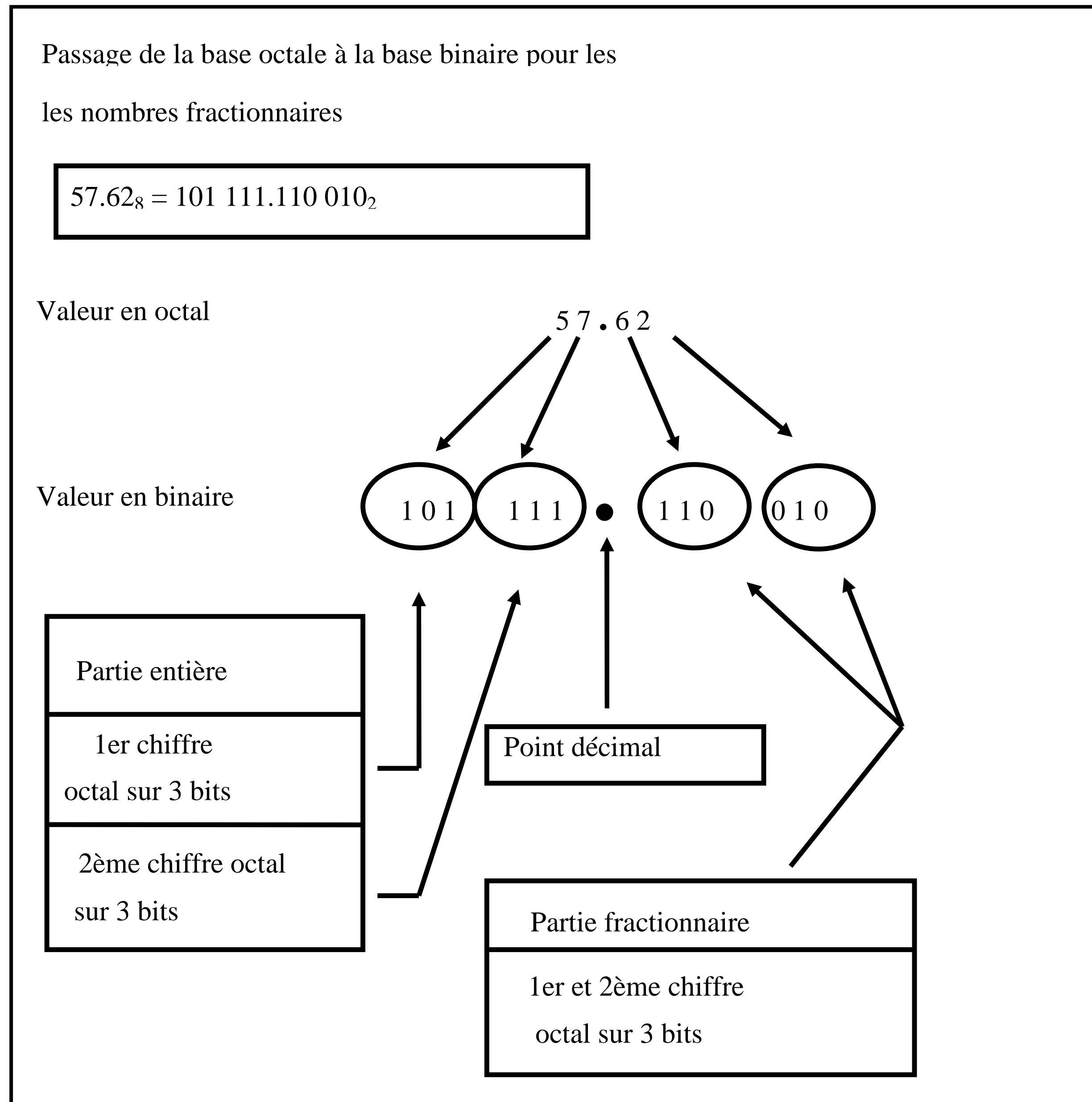
A l'inverse, on obtient le passage d'une base hexadécimale (ou octale) à la base binaire d'un nombre fractionnaire en exprimant en binaire chacun des caractères du nombre sur 4 bits (ou sur 3 bits pour la base octale) tout en respectant la position du point décimal. Voyons cela sur les exemples suivants.

Exemple 10 : Exprimer le nombre $B3F.A3_{16}$ en binaire.



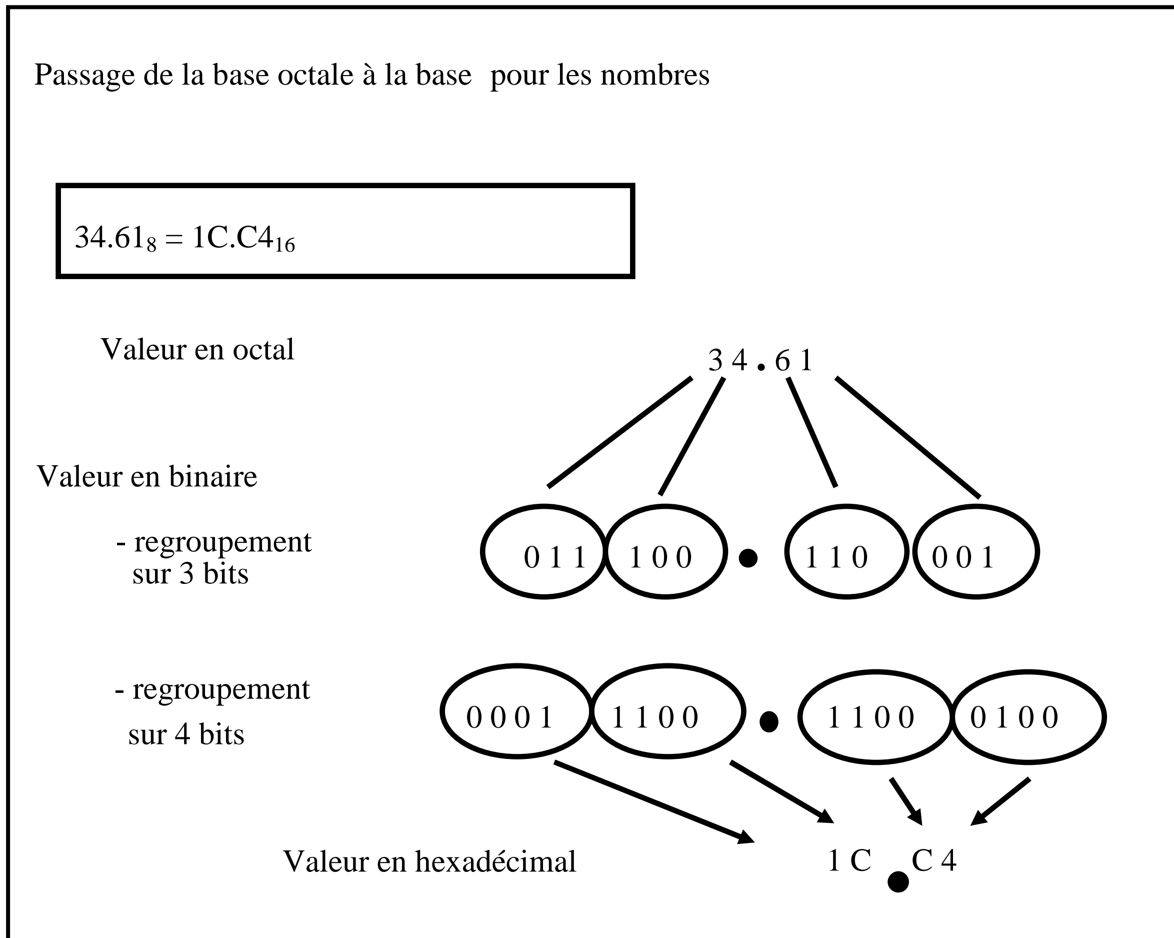
On a donc : $B3F.A3_{16} = 101100111111.10100011_2$.

Exemple 11 : Trouver l'expression de 57.62_8 en binaire.



On a donc : $57.62_8 = 101111.11101_2$

Exemple 12 : Trouver la représentation de 34.61_8 en notation hexadécimale.



On a donc $34.61_8 = 1C.C4_{16}$

1.2.5 Conclusion sur les changements de base

On peut constater qu'il est relativement facile de passer du système octal ou hexadécimal au système binaire. Cette opération est facile à automatiser.

Lors de la compilation d'un programme, les caractères sont d'abord codés en chaînes de caractères en hexadécimal, selon un code préétabli (code ASCII, EBCDIC, ...), et ensuite traduit en chaînes binaires (langage machine). L'utilisation de systèmes numériques de base, multiples de 2, différent du système décimal, permet de simplifier la codification et les calculs dans le langage des ordinateurs.

Tableau 3 : Résumé des méthodes de conversion.

RÉSUMÉ SUR LES MÉTHODES DE CONVERSION D'UN SYSTEME DE NUMÉRATION À UN AUTRE		
D	**À**	**MÉTHODE**
Base b	**décimal**	- réécrire sous forme de polynôme - calcul arithmétique des termes
décimal	**base b**	**Partie entière** - succession de divisions entières par b - regroupement dans l'ordre inverse des restes obtenus **Partie fractionnaire** - multiplications successives par b de la partie fractionnaire - regroupement des entiers obtenus
binaire	**octal**	- trouver la valeur octale de chaque tranche de 3 bits
octal	**binaire**	- trouver la valeur binaire (sur 3 de chaque symbole octal
binaire	**hexadécimal**	- trouver la valeur de chaque tranche de 4 bits
hexadécimale	**binaire**	- trouver la valeur binaire (sur 4 bits) de chaque symbole hexadécimale

Notes sur les différents systèmes de numération

Le format hexadécimal est largement utilisé en informatique car il offre une conversion facile avec le système binaire, qui est le système employé par les ordinateurs.

Il est, comme nous venons de le voir, possible de traduire du binaire vers de l'hexadécimal et réciproquement, groupe de chiffres par groupe de chiffres. Cette facilité de conversion a conduit la notation hexadécimale à être utilisée pour noter des nombres initialement quantifié ou à destination d'être quantifié en binaire, l'hexadécimal étant plus compact (quatre fois moins de chiffres) et offrant une meilleur lisibilité pour l'œil humain.

Un avantage supplémentaire de la base 16 est sa concordance avec l'octet, mot de 8 bits auquel correspond aujourd'hui fréquemment un byte, la plus petite unité de stockage adressable. Il est donc commode de noter la valeur d'un octet sur deux chiffres hexadécimaux.

La table ASCII a été construite de façon à faire commencer les suites de symboles élémentaires (chiffres, lettres minuscule, lettres majuscules) à des positions remarquables lorsqu'elles sont exprimés en binaire, octal ou hexadécimal.

Par exemple, la lettre « A » correspond ainsi au code hexadécimal 41 (40 + la position de la lettre dans l'alphabet), le chiffre « 0 » correspond au code hexadécimal 30 (30 + la valeur du chiffre).

Les conversions entre le système décimal et l'hexadécimal sont moins aisées que les conversions entre le système décimal et le binaire.

1.3. La représentation des nombres négatifs

Un autre problème de la représentation des données est celui de la représentation des nombres négatifs. En effet, certaines opérations supposent le traitement d'entiers négatifs. On peut identifier trois principales façons de représenter les nombres négatifs:

1. représentation en valeur absolue et signe
2. représentation par le complément restreint appelé complément à 1
3. représentation par le complément vrai appelé complément à 2.

1.3.1 La représentation en valeur absolue et signe

Il s'agit ici d'utiliser un bit pour représenter le signe de la valeur à représenter. Selon que le nombre est positif ou négatif, le bit d'extrême gauche prendra par convention la valeur 0 ou la valeur 1. Par exemple, sur 4 bits, 1 bit sera réservé au signe et trois bits seront utilisés pour représenter les nombres en valeur absolue:

Figure 4:

signe / valeur	valeur
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	- 0
1 0 0 1	- 1
1 0 1 0	- 2
1 0 1 1	- 3
1 1 0 0	- 4
1 1 0 1	- 5
1 1 1 0	- 6
1 1 1 1	- 7

Comme 3 bits sont utilisés pour représenter la valeur absolue, on peut donc représenter 2^3 valeurs absolues différentes, soient les valeurs comprises entre 0 et 7. Le bit de signe nous permet d'affecter à chacune de ces valeurs un signe + ou -. Autrement dit, n-1 bits servent à représenter la valeur absolue, ce qui donne une étendue possible allant de

$$\mathbf{-\ (2^{n-1} - 1)\ à\ (2^{n-1} - 1)}$$

donc de -7 à +7 dans notre exemple.

<u>Inconvénients:</u>

Cette méthode impose que le signe soit traité indépendamment de la valeur. Il faut donc des circuits différents pour l'addition et la soustraction. De plus, on obtient deux représentations différentes pour 0, soit +0 et -0.

Pour palier à ces inconvénients, il faut un mode de représentation qui permet d'utiliser le même circuit pour effectuer l’addition et la soustraction. Comme x - y = x + (-y), il faut que le

signe moins soit traité comme partie intégrante de la valeur négative. La forme complémentée utilisée pour représenter les nombres négatifs permet de réaliser cet objectif.

1.3.2 La représentation sous forme complémentée

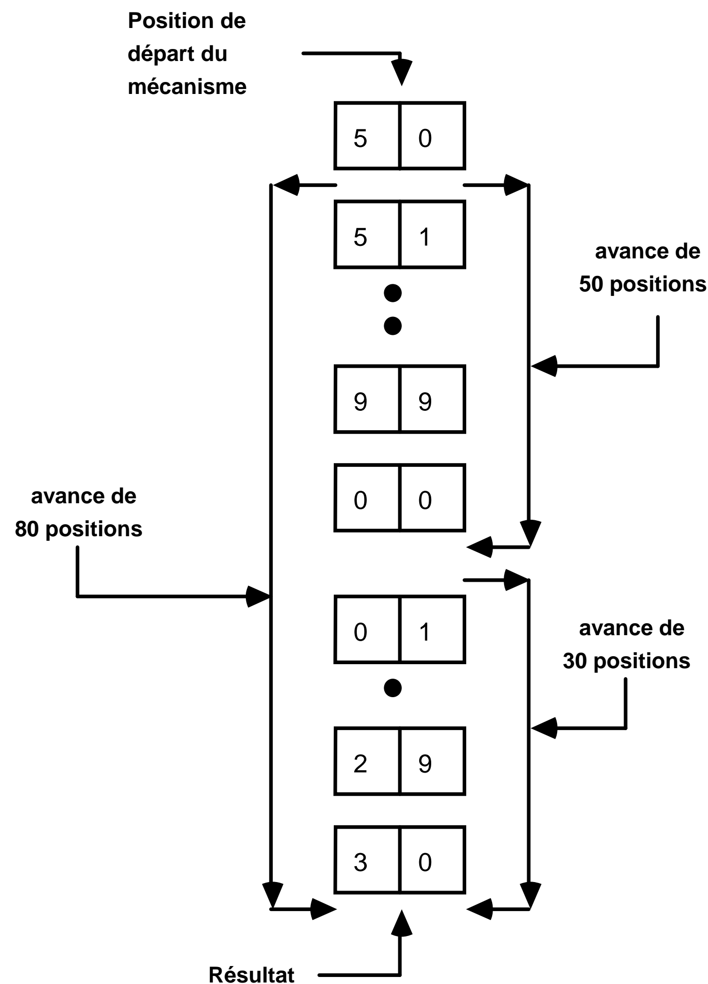
La représentation sous forme complémentée repose sur le fait que pour *n* bits de positions, le nombre de valeurs possibles et distinctes est limité à 2^n. Afin de mieux introduire cette représentation, choisissons l'exemple d'un compteur en base 10, avec disons, deux bits de position qu'on pourrait imaginer comme étant deux roues dentées ayant chacune 10 positions numérotées de 0 à 9. La plus grande valeur qu'on peut représenter avec ce mécanisme est alors 99. Pour faire une addition, par exemple 50 + 20, on se positionne à 50 et on avance de 20 positions pour atteindre 70. Jusque-là, tout cela est élémentaire.

Maintenant, si on veut effectuer l'addition d'un nombre négatif, disons 50 + (-20), une manière de faire est de se positionner au départ à 50 et de reculer de 20 positions pour trouver la position 30, le bon résultat. Cependant, cette méthode suppose que notre mécanisme de roues dentées puisse reculer et avancer au besoin, indiqué par un système de commande chargé de reconnaître le signe de l'opération.

Supposons maintenant que l'on veuille réaliser une machine plus simple à deux roues dentées qui ne pourrait qu'avancer et qui permettrait d'effectuer les additions autant que les soustractions! Comme on va le voir, le concept de complément nous permet de le faire.

On sait qu'une fois qu'on a atteint la valeur maximale, dans ce cas-ci 99, le compteur retombe fatalement à 00. C'est précisément le fait qu'il soit impossible de tenir compte de la retenue d'extrême gauche (n'ayant pas de troisième roue) qui nous simplifie la tâche: au lieu de reculer de 20 positions, on avancera de 80 positions. On atteindra ainsi la position 30, représentant le résultat souhaité (50 - 20 = 30)

Figure 5: Compteur à 99



Le nombre 80 est donc la valeur à additionner à 50 pour obtenir 30 avec notre mécanisme qui ne peut reculer. Si on voulait effectuer l'opération 50 - 37 avec notre mécanisme

Quel serait le nombre qu'il faudrait ajouter à 50 pour que les roues représentent le bon résultat, soit 13 ? On peut facilement vérifier qu'il s'agit de 63.

On remarque que 80 est le complément qu'il faut ajouter à 20 pour atteindre 100, le nombre suivant la valeur maximale du compteur (ici 99). De même que 63 est le complément qu'il faut ajouter à 37 pour obtenir la valeur maximale plus 1, soit 100. Nous appellerons cette valeur le complément à 100. On voit que l'utilisation du complément nous permet d'effectuer une soustraction en se servant du même procédé mécanique que celui utilisé pour une addition.

Il reste cependant un problème: comment trouver le complément à 100 puisque notre compteur ne peut pas représenter la valeur 100 sur ses deux roues dentées représentant les deux positions? Ce problème est contourné en soulignant que le complément à 100 est le complément à 99 auquel on ajoute 1. Pour nos exemples, on trouve le complément à 100 de 20 (soit 80) en trouvant le complément à 99 (soit 79) et en lui ajoutant 1. De même, le complément à 100 de 37 en calculant le complément à 99 de 37, soit 62, auquel il faut ajouter 1 pour obtenir 63.

1.3.3 Représentation complémentée des nombres binaires

Lorsqu'on a affaire à des représentations en binaire, le principe de représentation des nombres négatifs sous forme complémentée est similaire.

Pour *n* bits de positions, le bit à l'extrême gauche (le bit le plus significatif ou de poids le plus élevé) servira à identifier le signe tandis que les n - 1 autres bits serviront à représenter la valeur. En fait, si le nombre à représenter est négatif, on trouve d'abord le complément restreint de la valeur absolue du nombre, c'est-à-dire ce qui manque à cette valeur pour égaler la valeur maximale possible représentable sur *n* bits. On appelle aussi le complément restreint dans le système binaire le complément à 1. Par la suite, on additionne 1 au complément restreint afin d'obtenir le complément vrai appelé aussi complément à 2 dans le système binaire.

1.3.3.1 Le complément à 1 des nombres binaires

On pourrait définir le complément à 1 comme ce qu'il faut ajouter à une valeur pour obtenir la valeur maximale représentable sur le nombre de bits disponibles. C'est le pendant binaire du complément à 99 de notre exemple avec un compteur décimal.

Par exemple, en binaire sur 4 bits, 1111 est la valeur maximale qu'on puisse représenter. Le complément à 1 de 0000 est donc 1111, le complément à 1 de 0001 est 1110 et le complément à 1 de 1010 est 0101 car:

	0 0 0 1			1 0 1 0
+		et	+	
	1 1 1 0			0 1 0 1
	1 1 1 1			1 1 1 1

On constate que le complément à 1 d'un nombre binaire se trouve simplement en remplaçant les 0 par des 1 et les 1 par des 0.

Notons que l'utilisation du complément à 1 pour représenter les nombres négatifs nous donne encore une double représentation pour le 0. Par exemple sur n bits, +0 est représenté par une chaîne de n zéros (00...0) et -0 est représenté par une chaîne de n un (11...1).

1.3.3.2 Le complément à 2

Le complément à 2 d'une valeur binaire est ce qu'il faut ajouter à cette valeur pour qu'elle atteigne une unité de plus que la valeur maximale qu'on peut représenter sur *n* bits. C'est donc le (complément à 1) + 1. Cette technique élimine le problème de la double représentation du 0 (+0 et -0) comme c'est le cas dans la représentation "signe et valeur absolue" ou celle du complément à 1.

La représentation des nombres négatifs sous forme complémentée est largement utilisée. Cela s'explique parce que le complément à 2 permet d'éliminer la double représentation de 0 tout en

gardant la facilité de reconnaître le signe par le bit d'extrême gauche. Notons que le complément à 2 du complément à 2 d'un nombre redonne le nombre.

Ainsi, sur 4 bits, avec le signe représenté sur le bit le plus significatif et 3 bits qui permettent de représenter les valeurs, on peut représenter les entiers de -8 à 7, soit un entier négatif de plus qu'un complément à 1 ou en signe plus valeur absolue:

Tableau 4 : Représentation des nombres binaire négatifs

valeur décimale	valeur binaire	complément à 1	complément à 2	signe et valeur absolue
7	0 1 1 1	0 1 1 1	0 1 1 1	0 1 1 1
6	0 1 1 0	0 1 1 0	0 1 1 0	0 1 1 0
5	0 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1
4	0 1 0 0	0 1 0 0	0 1 0 0	0 1 0 0
3	0 0 1 1	0 0 1 1	0 0 1 1	0 0 1 1
2	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
- 0	- 0 0 0 0	1 1 1 1	0 0 0 0	1 0 0 0
- 1	- 0 0 0 1	1 1 1 0	1 1 1 1	1 0 0 1
- 2	- 0 0 1 0	1 1 0 1	1 1 1 0	1 0 1 0
- 3	- 0 0 1 1	1 1 0 0	1 1 0 1	1 0 1 1
- 4	- 0 1 0 0	1 0 1 1	1 1 0 0	1 1 0 0
- 5	- 0 1 0 1	1 0 1 0	1 0 1 1	1 1 0 1
- 6	- 0 1 1 0	1 0 0 1	1 0 1 0	1 1 1 0
- 7	- 0 1 1 1	1 0 0 0	1 0 0 1	1 1 1 1
- 8	- 1 0 0 0	---	1 0 0 0	---

De façon générale, sur *n* bits, le complément à 2 des nombres négatifs permet de représenter les nombres compris entre

$$-2^{n-1} \quad \text{et} \quad 2^{n-1} - 1$$

Exemple 13 : Trouver le complément à 2 de 01010001_2.

On trouve d'abord le complément à 1 de 01010001 en changeant les 0 en 1 et vice-versa, les 1 en 0, et on lui ajoute 1 ce qui donne:

complément à 1 de 01010001

```
  1 0 1 0 1 1 1 0
+ 0 0 0 0 0 0 0 1
  ---------------
  1 0 1 0 1 1 1 1
```

indicateur du signe

Exemple 14 : Quelle est la valeur décimale de 101010001 s'il s'agit de la représentation à 2 d'un nombre entier sur 9 bits de position?

Le bit d'extrême gauche de 101010001 indique qu'il s'agit d'un nombre négatif. Par exemple, si l'ordinateur doit afficher ce nombre à l'écran, il comprendra qu'il faudra générer un signe moins avant la chaîne de caractères décimaux représentant le nombre obtenu à partir du complément à 2 de la chaîne binaire 101010001. La valeur est donc:

Valeur cherchée = - (complément à 2 de 101010001)
= - (010101110 + 1)
= - 010101111_2
= - 128 + 32 + 8 + 4 + 2 + 1
= - 175_{10}

Exemple 15 : Trouver la représentation complément à 2 sur 10 bits de position de -276_{10}

Le signe est négatif. Il indique donc qu'il s'agit du complément à 2 de la représentation binaire de 276 sur 10 bits de position. On a:

Représentation cherchée = complément à 2 de 276_{10}
= complément à 2 de 0100010100_2
= 1011101011 + 1
= 1011101100

Exemple 16 : Effectuer 5 - 3 sur 4 bits (1 bit de signe + 3 bits) en utilisant le complément à 2 pour les nombres négatifs:

-3 = complément à 2 de 0011
= complément à 1 de 0011 + 1
= 1100 + 1
= 1101

```
donc:     0 1 0 1          5
        + 1 1 0 1 →    + (-3)
          0 0 1 0          2
```

Exemple 17 : Essayons maintenant avec un nombre plus grand que 5, 5 - 7 par exemple, toujours en étant sur 4 bits.

On a:

Le complément à 2 de -7_{10}, est 0111 → 1000 + 1 = 1001.

On a donc:

```
    0 1 0 1
+   1 0 0 1
    1 1 1 0 → soit -2 en complément à 2.
```

En effet,

1 1 1 0 = -(complément à 2 de 1 1 1 0)
= - (0 0 0 1 + 1)
= - $0\,0\,1\,0_2$
= - 2_{10}.

1.3.3.3 Débordement de capacité en complément à 2.

Un débordement de capacité se produit lorsqu'on effectue une opération dont le résultat dépasse la valeur maximale représentable sur n bits.

Par exemple, on sait que sur 4 bits, si on utilise le bit le plus significatif pour le signe. Dans la représentation en complément à 2 pour les nombres négatifs, les valeurs pouvant être représentées vont de -8 à +7. Que se passe-t-il si on additionne 4 + 5?

```
        retenues:0 1
      4        0 1 0 0
  +   5    +   0 1 0 1
      9        1 0 0 1
```

Remarque : cette chaîne binaire en complément à 2 ne sera pas interprétée en décimal comme le nombre 9 mais plutôt comme

-(complément à 2 de 1001) = -7

qui n'est évidemment pas le résultat de 4 + 5 !

Le résultat obtenu n'est pas 9. La raison est que 9 ne peut être représenté sur 4 bits en complément à 2. Mais comment l'ordinateur peut-il vérifier que le nombre négatif obtenu comme résultat est faux? Cette vérification se fait en comparant les deux retenues d'extrême gauche: si les deux retenues d'extrême gauche sont différentes (comme dans notre exemple), cela signifie qu'il y a eu débordement de capacité et que le résultat est faux.

Par contre, si les retenues d'extrême gauche sont identiques, le résultat est alors correct.

Exemple 18 :

```
      retenues:1 1
       7          0 1 1 1
 +   (-4)    +    1 1 0 0
       3          0 0 1 1
```

Ici, les retenues sont identiques, ce qui nous indique qu'il n'y a pas de débordement de capacité et que par conséquent, le résultat est exact.

Exemple 19 :

```
      retenues:1 0
     (-6)         1 0 1 0
 +   (-5)    +    1 0 1 1
     -11          0 1 0 1
```

Dans ce cas ci, les retenues sont différentes, ce qu'indique un débordement de capacité; le résultat est donc faux.

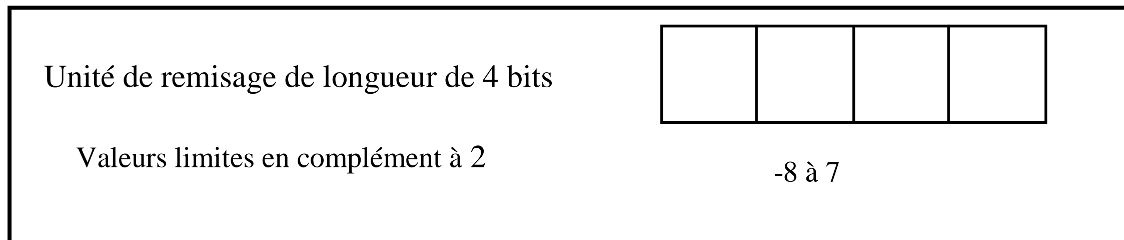
Exemple 20 : (sur 8 bits)

```
      retenues:1 0
     (-62)        1 1 0 0  0 0 1 0
 +   (-89)   +    1 0 1 0  0 1 1 1
     -151         0 1 1 0  1 0 0 1
```

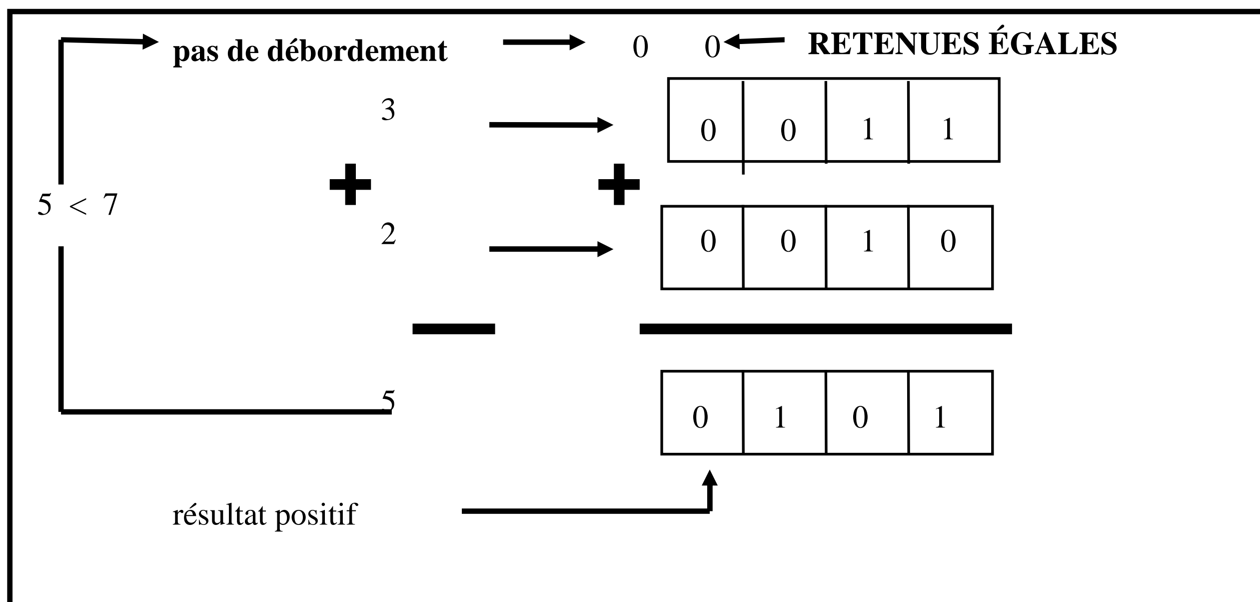
Les retenues différentes signalent un débordement de capacité. On peut d'ailleurs vérifier que -151 dépasse la capacité sur 8 bits qui est, pour les nombres négatifs, de -2^{8-1} , soit -128.

1.3.3.4 Addition et soustraction d'entiers exprimées dans la représentation complément à 2

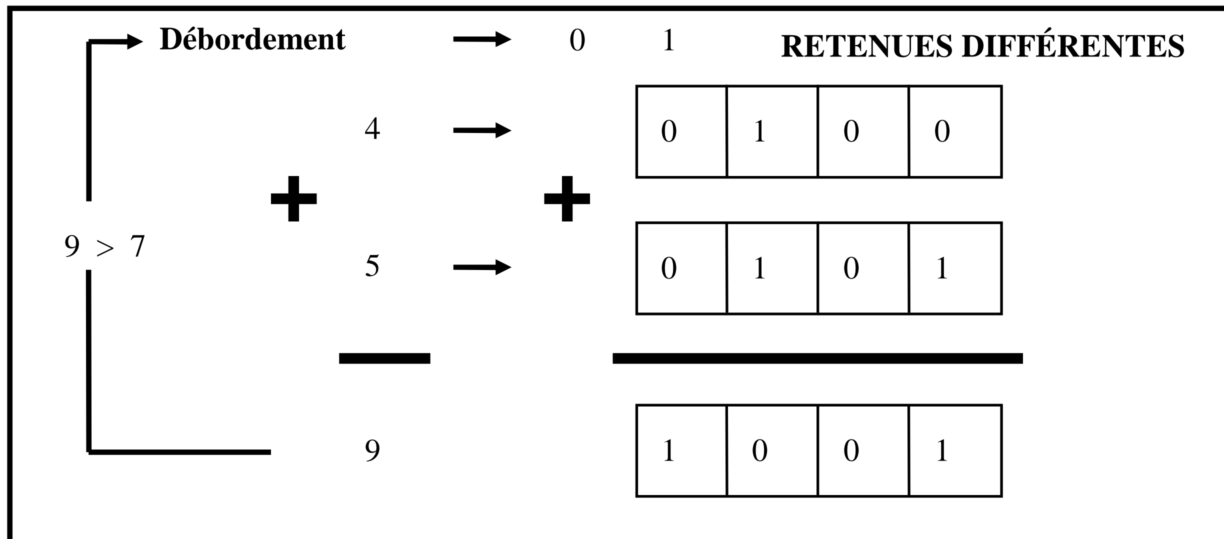
Considérez des entiers exprimés sur quatre bits.



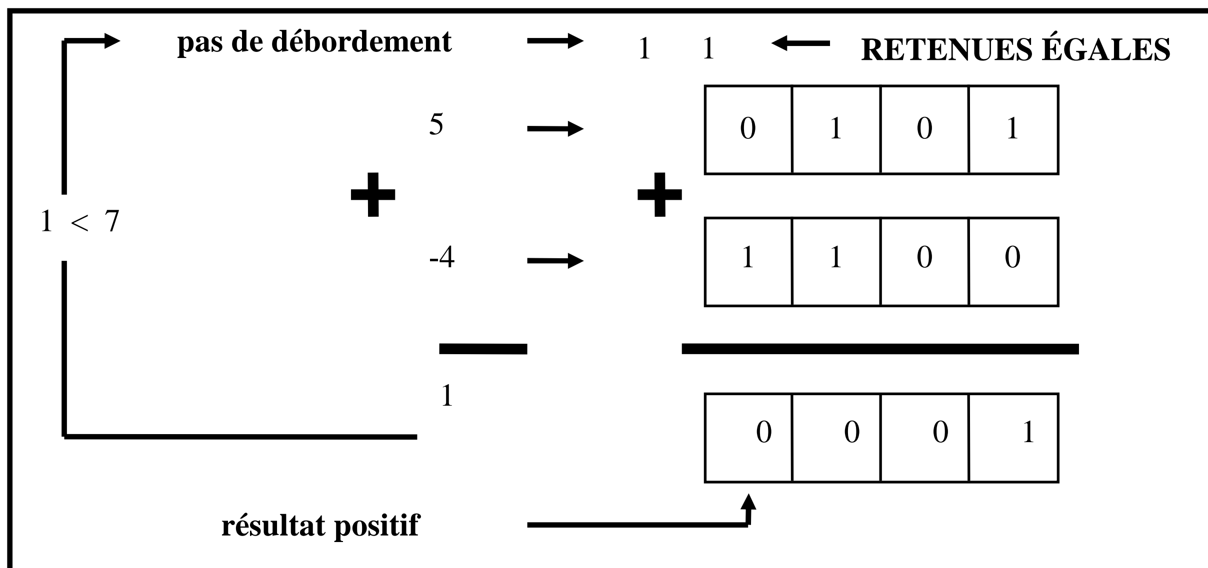
1ier cas addition sans débordement



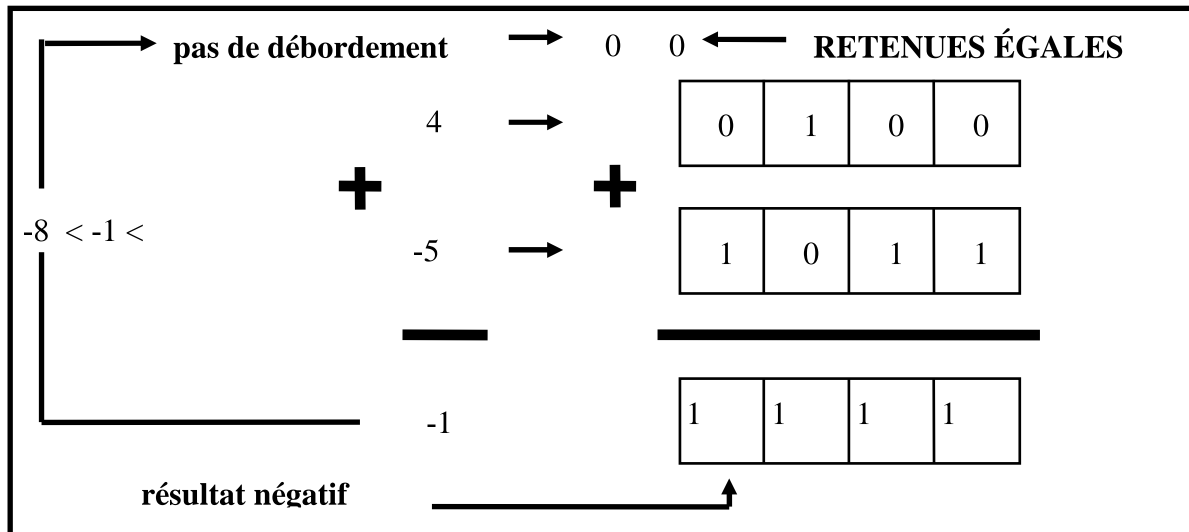
2ième cas addition avec débordement



3ème cas soustraction sans débordement et résultat positif



4ème cas soustraction sans débordement et résultat négatif


pas de débordement
0 0
RETENUES ÉGALES
4
0 1 0 0
-8 < -1 <
-5
1 0 1 1
-1
1 1 1 1
résultat négatif

5ème cas soustraction sans débordement et résultat négatif

débordement → 1 0 ← **RETENUES DIFFÉRENTES**

-4	→	1	1	0	0
+	+				
-5	→	1	0	1	1
—	—				
-9		0	1	1	1

-9 < -7

Tableau 5 : Résumé sur la représentation des nombres négatifs

RÉSUMÉ SUR LA REPRÉSENTATION DES NOMBRES NÉGATIFS

3 modes de représentation :

1. signe et valeur absolue
2. complément à 1
3. complément à 2

Règles générales pour le signe (+/- = 0/1)

- le signe est toujours représenté par le bit le plus significatif (extrême gauche)
- la reconnaissance d'un signe négatif commande la forme complémentée de la chaîne considérée

1. Signe et valeur absolue:

le bit de signe prend la valeur 0 si le nombre est positif et la valeur 1 si le nombre est négatif. Les n-1 autres bits représentent la valeur absolue du nombre.

2. Complément à 1:

la valeur négative est remplacée par son complément restreint, c'est à dire la différence entre cette valeur et la valeur maximale. En binaire, le complément à 1 se trouve en changeant les 0 par des 1 et vice-versa.

3. Complément à 2:

la valeur négative est remplacée par son complément vrai, c'est à dire le complément à 1 + 1.

Débordement de capacité:

il survient lorsque le résultat dépasse la valeur maximale que le nombre de bits permet de représenter.

Règle de reconnaissance:

deux retenues d'extrême gauche différentes signalent un débordement de capacité.

1.4. La codification des chiffres décimaux

Il existe d'autres formes de codifications permettant de représenter directement les nombres décimaux simplifiant ainsi la conversion décimal binaire. Ces codage simplifient également les entrée/sortie (les périphériques utilisant un chiffre/octet:

1.4.1 Chiffres décimaux codés en binaire (système BCD : Binary Coded Decimal)

Le code **BCD** conserve la représentation décimale d'un nombre (centaines, dizaines, unités...), mais chaque chiffre de ce nombre est reproduit en binaire.

Etant donné que chaque rang décimal (unités, dizaines, centaines...) peut contenir un chiffre de 0 à 9, chaque rang du code BCD sera représenté par quatre chiffres binaires (de 0000 à 1001), donc quatre bits.

À partir des codes EBCDIC ou ASCII, en enlevant la partie code dans chaque caractère décimal et en ne gardant que la partie numérique, on obtient pour chaque chiffre du système décimal, les représentations suivantes (système appelé BCD: Décimal Codé Binaire) : chaque chiffre d'un nombre est codé sur 4 bits

Figure 6 : Code BCD

Chiffre décimal	Code BCD
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

Par exemple, 129 se représente dans le système BCD par la chaîne binaire 0001 0010 1001 après avoir remplacé chaque chiffre par son code correspondant, soit le chiffre 1 par 0 0 0 1, le chiffre 2 par 0 0 1 0, et le chiffre 9 par 1 0 0 1.

Bien que le BCD gâche de l'espace (environ 1/6 de la mémoire disponible est perdue), il permet d'avoir une correspondance immédiate avec les codes de caractères ASCII ou EBCDIC représentant les chiffres : il suffit de réaliser un OU logique avec 00110000 (48 en base 10) pour l'ASCII, ou 11110000 pour l'EBCDIC.

1.5. Les opérations arithmétiques en codification BCD

1.5.1 L'addition

Exemple 21 :

Décimal	**BCD**
15	0 0 0 1 0 1 0 1
+ 12	+ 0 0 0 1 0 0 1 0
27	0 0 1 0 0 1 1 1

Pour ce cas, il y a correspondance entre l'addition décimale et l'addition des représentations BCD de chaque chiffre en binaire. Mais ce n'est pas toujours le cas. Considérons par exemple le nouveau cas:

Exemple 22 :

Décimal	**BCD**
15	0 0 0 1 0 1 0 1
+ 18	+ 0 0 0 1 1 0 0 0
33	0 0 1 0 1 1 0 1 → 2D

L'addition en binaire ne présente plus une correspondance entre ce qui se passe en décimal et ce qui se passe en représentation BCD binaire comme dans l'exemple précédent. Ici le retour direct aux caractères (par l'ajout de la partie code à chacun des groupes de quatre bits) produit le résultat 2D.

Pourquoi et peut-on remédier à cette situation?

La raison est simple: puisque nous disposons de quatre bits pour la représentation de chacun des chiffres, nous pouvons représenter tous les chiffres hexadécimaux (de 0 à 9 en plus de A à F qui ont pour valeur respective allant de 10 à 15). Il y a donc six chiffres de trop par rapport au système décimal, soit les chiffres hexadécimaux A, B, C, D, E, F ayant pour valeur respective 10, 11, 12, 13, 14 et 15.

Le report aux dizaines est donc retardé de six unités. Pour produire ce report à temps lorsqu'il y a dépassement à 9 dans une position décimale, il faut donc ajouter la valeur 6.Dans l'exemple précédent (15 + 18) qui donnait (1 + 1)(5 + 8) = 2D, on obtient en corrigeant: 2(D+6) = 2(13+6) = (2+1)(19-16) = 33, le résultat correspondant en décimal.

Dans le cas général, on effectue donc l'addition en motifs de 4 bits. Si le résultat dépasse 9 pour l'un ou plusieurs de ces motifs, on leur ajoute 6 pour forcer une retenue et on obtient le résultat escompté en BCD. Dans l'exemple précédent, 15 + 18, cette opération se traduit comme suit :

Exemple 22 (suite)

Décimal	**BCD**	
15	0 0 0 1 0 1 0 1	
+ 18	+ 0 0 0 1 1 0 0 0	
33	0 0 1 0 1 1 0 1	→ 2D
	+ 0 0 0 0 0 1 1 0	→ Chaîne de correction
33	0 0 1 1 0 0 1 1	

Exemple 23

Décimal	BCD	
75	0 1 1 1 0 1 0 1	
+ 68	+ 0 1 1 0 1 0 0 0	
143	1 1 0 1 1 1 0 1	→ DD
	+ 0 1 1 0 0 1 1 0	→ Chaîne de correction
143	0 0 0 1 0 1 0 0 0 0 1 1	

Exemple 24 :

Décimal	**BCD**	
175	0 0 0 1 0 1 1 1 0 1 0 1	
+ 328	+ 0 0 1 1 0 0 1 0 1 0 0 0	
503	0 1 0 0 1 0 0 1 1 1 0 1	→ 49D
	+ 0 0 0 0 0 1 1 0 0 1 1 0	→ Chaîne de correction
503	0 1 0 1 0 0 0 0 0 0 1 1	

Dans cet exemple, on remarque qu'il faut tenir compte des retenues à chaque position décimale pour décider s'il faut corriger avec 6 ou 0, comme c'est le cas pour la deuxième position (7+2) qui hérite d'une retenue de la première position (5+8 = 3 retenue 1) donnant comme résultat effectif 7+2+1 = 10 dépassant 9, par conséquent qui génère un facteur de correction 6 en deuxième position décimale. En fait, nous avons 175+328 = (1+3)(7+2)(5+8) = (4)(9)(13) comme résultat, donnant après correction de 6 de la première position (4)(9)(13+6) = (4)(9+1)(19-16) = (4)(10)(3), lequel génère une autre correction de 6 en deuxième position due au dépassement à 10 et qui donne comme résultat final (4)(10+6)(3) = (4+1)(16-16)(3) = 503.

Exemple 25 :

Décimal	**BCD**	
175	0 0 0 1 0 1 1 1 0 1 0 1	
+ 828	+ 1 0 0 0 0 0 1 0 1 0 0 0	
1003	1 0 0 1 1 0 0 1 1 1 0 1	→ 99D
	+ 0 1 1 0 0 1 1 0 0 1 1 0	→ Chaîne de correction
1003	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1	

La chaîne de correction 0 1 1 0 0 1 1 0 0 1 1 0 ou 666 en hexadécimal est justifiée par le dépassement à 9 dans chacune des positions décimales en tenant compte des retenues générées.

Remarque : Il y a dépassement à 9 à chaque fois qu'il y a retenue sur l'opération du dernier bit du motif de quatre bits.

Exemple 26:

```
     99          1001   1001
+    99        + 1001   1001
    198          0011    0010 → 2 retenues à dernière opération des motif de 4 bits
               + 0110    0110 → Chaîne de correction
               1 1001    1000
```

1.5.2 La soustraction

Elle s'effectue en additionnant le complément à 10 du nombre à soustraire sans tenir compte de la retenue à l'extrême gauche (dernière retenue).

Exemple 26 :

Décimal

```
   47
-  13
   34   est remplacé par l'addition à 47 de 87 qui est le complément à 10 de 13, on a:
```

```
Décimal            BCD
   47            0 1 0 0  0 1 1 1
+  87      +     1 0 0 0  0 1 1 1
  134            1 1 0 0  1 1 1 0 → CE
           +     0 1 1 0  0 1 1 0 → Chaîne de correction
  134    0 0 0 1  0 0 1 1  0 1 0 0
  ↑             ↑ retenue ignorée
```

qui donne le résultat escompté 34 en ignorant la dernière retenue.

Exemple 27 :

Décimal

```
   847
-  613
   234
```

puisque 387 est le complément à 10 de 613,

Décimal	**BCD**	
847	1 0 0 0 0 1 0 0 0 1 1 1	
+ 387	+ 0 0 1 1 1 0 0 0 0 1 1 1	
	1 0 1 1 1 1 0 0 1 1 1 0	→ DCE
	+ 0 1 1 0 0 1 1 0 0 1 1 0	→ Chaîne de correction
1234	0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0	
↑	↑	

retenue ignorée

qui donne le résultat escompté 234 en ignorant la dernière retenue.

Remarque : Il est clair que le complément d'un nombre doit être pris par rapport au plus grand des deux nombres.

Exemple 28 :

Décimal
99
- 1
98

Puisque 99 est le complément à 10 de 1.

Décimal	**BCD**	
99	1 0 0 1 1 0 0 1	
+ 99	+ 1 0 0 1 1 0 0 1	
198	1 0 1 1 0 0 1 0	→ une retenue au premier motif et un chiffre plus grand que 9 au deuxième motif
↑		
retenue ignorée	+0 1 1 0 0 1 1 0	→ chaînes de correction
	1 0 0 1 1 0 0 0	

L'addition en BCD signé

En fait, la routine développée ci-dessus pour l'addition de nombres BCD décompactés non signés peut également être utilisée pour additionner des nombres BCD signés en code complément à 10. Si on appelle cette routine avec par exemple les opérandes 0123 et 9995, le résultat sera 0118 qui peut être interprété comme étant égal à :

0123 - 0005 = 0118

Autres exemples :

0123 + 9880 correspond à 0123 - 0120 et renvoie 0003
0123 + 9875 correspond à 0123 - 0125 et renvoie 9998 soit -0002
0123 + 9867 correspond à 0123 - 0133 et renvoie 9990 soit -0010

La seule différence entre l'utilisation de cette routine avec des nombres BCD signés ou non signés réside dans l'interprétation des résultats. On supposera que dans notre cas cette interprétation se limite à un affichage. Notez qu'il y a lieu de mettre les nombres sur le nombre de digits qu'on dispose. Par exemple, ci-dessus, nous en avons 4 et nous travaillons sur 4 digits.

Affichage de nombres BCD signés

L'affichage de nombres BCD signés peut être réalisé conformément à l'algorithme suivant :

si le digit de poids fort est '9', alors
  afficher '-'
  complémenter à 9 les digits restants
  ajouter 1
sinon
  afficher '+'
afficher les digits restants

Dépassement de capacité en BCD signé

La détection d'un éventuel dépassement après une addition est relativement aisée

Si le digit de poids fort est '0' ou '9', alors il n'y a pas de dépassement.

Si le digit de poids fort n'est pas '0' ou '9', alors il est significatif, et le signe du résultat peut être déduit du signe des opérandes qui sont soit tous les deux positifs, soit tous les deux négatifs.

1.5.3 La multiplication

La multiplication par 10 d'un nombre représenté en BCD se réalise par un décalage de quatre bits de position vers la gauche.

Décimal	**BCD**
47 X 10	Décalage de 4 bits à gauche de 0 1 0 0 0 1 1 1
470	0 1 0 0 0 1 1 1 0 0 0 0

La multiplication en général se fera par une suite d'additions et de multiplications par 10 dépendant du chiffre considéré dans le multiplicateur suivie de l'addition des résultats intermédiaires obtenus à chacune des positions décimales

Exemple 28 :

47 X 32 = 47 X (30 + 2) = [(3 X 47) X 10] + (2 X 47)
= [(47 + 47 + 47) X 10] + (47 + 47)

```
Décimal                    BCD
   47                  0 1 0 0  0 1 1 1
+  47             +    0 1 0 0  0 1 1 1
                       1 0 0 0  1 1 1 0 → 8F
                  +    0 0 0 0  0 1 1 0 → Chaîne de correction
   94                  1 0 0 1  0 1 0 0
+  47             +    0 1 0 0  0 1 1 1
                       1 1 0 1  1 0 1 1 → DB
                  +    0 1 1 0  0 1 1 0 → Chaîne de correction
  141         0 0 0 1  0 1 0 0  0 0 0 1
X  10                                    → Décalage de 4 bits vers la gauche
 1410  0 0 0 1  0 1 0 0  0 0 0 1  0 0 0 0 → = [(47 +47 +47) X 10]

   47                  0 1 0 0  0 1 1 1
+  47             +    0 1 0 0  0 1 1 1
                       1 0 0 0  1 1 1 0 → 8F
                  +    0 0 0 0  0 1 1 0 → Chaîne de correction
   94                  1 0 0 1  0 1 0 0 → = (47 + 47)

 1410       0 0 0 1  1 0 0 0  1 0 0 0  0 0 0 0
+  94       +                 1 0 0 1  0 1 0 0
            0 0 0 1  0 1 0 0  1 0 1 0  0 1 0 0 → 14A4
        +   0 0 0 0  0 0 0 0  0 1 1 0  0 0 0 0 → Chaîne de correction
 1504       0 0 0 1  0 1 0 1  0 0 0 0  0 1 0 0
```

1.5.4 **La division**

La division par 10 d'un nombre représenté en BCD se réalise par un décalage de quatre bits de position vers la droite. Par exemple, on a:

Décimal	BCD
47 / 10	Décalage de 4 bits à droite de 0 1 0 0 0 1 1 1 donnant
4.7	0 1 0 0 . 0 1 1 1

On indique ce décalage par le point décimal.

La division entière par 10 d'un nombre représenté en BCD se réalise en éliminant les 4 bits à l'extrême gauche. Par exemple, on a:

Décimal	**BCD**
47 DIV 10	Élimination des 4 bits de gauche de 0 1 0 0 0 1 1 1 donnant
4	0 1 0 0

La division en général se fera par une suite successive de soustractions au dividende du diviseur, et lorsque le résultat devient négatif, de restitution du reste et de multiplication par 10 pour générer la décimale qui suit en répétant l'opération.

Exemple 29 :

47 / 36 donnera:

47 - 36 = 11 → donnant 1 reste 11
11 – 36 = -25 → reste négatif
-25 + 36 = 11 → restitution du reste
11 X 10 = 110 → décalage vers la gauche et
génération de la première décimale après le point,
le résultat étant → 1.0 + ..
110 – 36 = 74 → donnant **1.1** +..
74 – 36 = 38 → donnant **1.2** + ..
38 – 36 = 2 → donnant **1.3** + ..
2 – 26 = - 34 → reste négatif
-34 + 36 = 2 → restitution du reste
2 X 10 = 20 → décalage vers la gauche et
génération de la deuxième décimale après le point
le résultat étant → 1.30 + ..
20 – 36 = - 16 → reste négatif
-16 + 36 = 20 → restitution du reste
20 * 10 = 200 → décalage vers la gauche et
génération de la troisième décimale après le point
le résultat étant → 1.300 + ..
200 – 36 = 174 → donnant **1.301** + ..
174 – 36 = 138 → donnant **1.302** + ..
138 – 36 = 102 → donnant **1.303** + ..
102 – 36 = 66 → donnant **1.304** + ..
66 – 36 = 30 → donnant **1.305** + ..
30 – 36 = - 6 → reste négatif
-6 + 36 = 30 → restitution du reste
30 X 10 = 360 → décalage vers la gauche et
génération de la quatrième décimale après le point
le résultat étant → 1.3050 + ..
→ et ainsi de suite.

Autres types de Codifications

1.5.5 Chiffres décimaux codés en excédent-3

On obtient cette codification en ajoutant 3 (ou 0 0 1 1) au code BCD de chaque caractère numérique.

Chiffre décimal	Code BCD	Code excédent-3
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

L'avantage majeur de cette représentation est de simplifier les opérations impliquant le complément à 9 des nombres comme dans la soustraction. En effet, le code excédent-3 du complément à 9 d'un chiffre est obtenu directement du code excédent-3 du chiffre en remplaçant les 0 par 1 et les 1 par 0, opération très rapide pour un ordinateur. On a:

Chiffre décimal	Code excédent-3	Code excédent-3 du complément à 9
0	0 0 1 1	1 1 0 0
1	0 1 0 0	1 0 1 1
2	0 1 0 1	1 0 1 0
3	0 1 1 0	1 0 0 1
4	0 1 1 1	1 0 0 0
5	1 0 0 0	0 1 1 1
6	1 0 0 1	0 1 1 0
7	1 0 1 0	0 1 0 1
8	1 0 1 1	0 1 0 0
9	1 1 0 0	0 0 1 1

Comme exemple, considérons le nombre 356. Sa représentation en excédent-3 sera:

0 1 1 0 1 0 0 0 1 0 0 1 .

La représentation excédent-3 du complément à 9 de 356 sera obtenue en remplaçant les 0 par des 1 et les 1 par des 0 dans la chaîne représentant le nombre 356. On obtient:

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

qui s'interprète en décimal comme le nombre 643. On a bel et bien 356 + 643 = 999.

1.5.6 Chiffres décimaux codés 2 dans 5

On obtient cette codification sur 5 bits en partant de 0 0 0 1 1 pour le code de 0 et en déplaçant les 1 dans le code vers la gauche, un à la fois pour la suite des chiffres décimaux. On a:

Chiffre décimal	Code BCD	Code 2 dans 5
0	0 0 0 0	0 0 0 1 1
1	0 0 0 1	0 0 1 0 1
2	0 0 1 0	0 0 1 1 0
3	0 0 1 1	0 1 0 0 1
4	0 1 0 0	0 1 0 1 0
5	0 1 0 1	0 1 1 0 0
6	0 1 1 0	1 0 0 0 1
7	0 1 1 1	1 0 0 1 0
8	1 0 0 0	1 0 1 0 0
9	1 0 0 1	1 1 0 0 0

L'avantage que présente cette codification est de permettre de mieux détecter les erreurs dans la transmission de nombres, le code de 5 bits de chaque chiffre ne devant avoir que deux bits à 1.

1.5.7 Chiffres décimaux codés biquinaires

Cette codification est sur 7 bits divisée en deux régions. La première région est formée des 2 premiers bits de gauche et la deuxième région est formée des 5 autres. Les régions partent avec 0 1 et 0 0 0 0 1 comme valeur. On obtient le code des autres chiffres décimaux en déplaçant les 1 dans chacune des régions, une après l'autre, vers la gauche, d'une position à la fois. On a:

Chiffre décimal	Code BCD	Code biquinaire
0	0 0 0 0	0 1 0 0 0 0 1
1	0 0 0 1	0 1 0 0 0 1 0
2	0 0 1 0	0 1 0 0 1 0 0
3	0 0 1 1	0 1 0 1 0 0 0
4	0 1 0 0	0 1 1 0 0 0 0
5	0 1 0 1	1 0 0 0 0 0 1
6	0 1 1 0	1 0 0 0 0 1 0
7	0 1 1 1	1 0 0 0 1 0 0
8	1 0 0 0	1 0 0 1 0 0 0
9	1 0 0 1	1 0 1 0 0 0 0

L'avantage que présente cette codification est aussi de permettre de mieux détecter les erreurs dans la transmission de nombres, le code de 7 bits de chaque chiffre ne devant avoir que deux bits à 1, et un seul par région.

1.6. Représentation des caractères alphanumériques

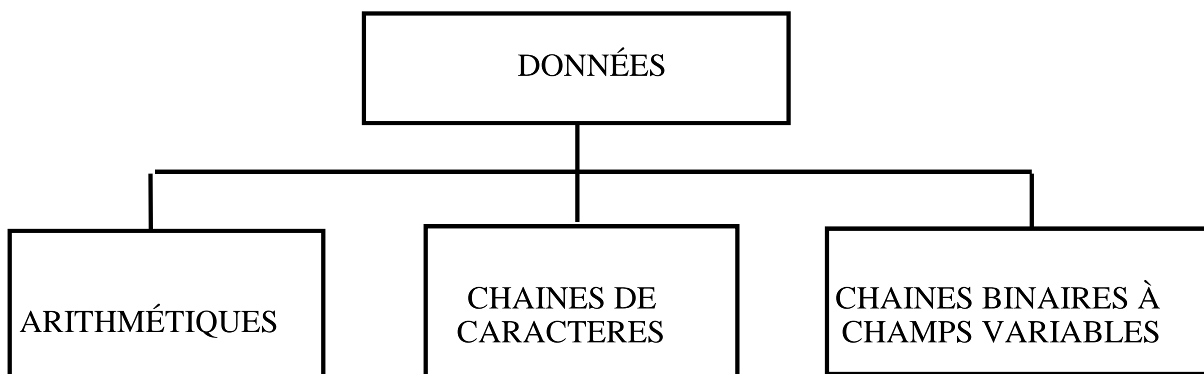
Les caractères numériques et alphabétiques des données externes, c'est-à-dire les données provenant d'unités périphériques ou destinées à être transférées vers ces mêmes unités, sont représentés à l'aide d'une chaîne de bits selon un code préétabli. Les ordinateurs n'utilisent pas tous les mêmes jeux de caractères. Les codes les plus utilisés sont l'**ASCII** de 7 bits (American Standard Code for Information Interchange) permettant la codification de 128 caractères (2^7), l'ASCII étendu à 8 bits augmentant le nombre de caractères à 256 (2^8) et l'**EBCDIC** (Extended Binary Coded Decimal Interchange Code) également à 8 bits, un code basé sur le code BCDIC de 7 bits. Il existe également le jeu de caractères scientifiques **CDC** qui est un code à 6 bits. Sur l'ensemble des possibilités de codage d'un jeu de caractères, un certain nombre sont réservées pour les fonctions de contrôle tels le retour de chariot et le saut de ligne. Les autres possibilités représentent les caractères comme les lettres majuscules et minuscules, les chiffres et les caractères graphiques. On pourra trouver à la fin de ce chapitre la table ascii.

1.7. Un exemple de représentation des données : Le VAX 11/780

1.7.1 Les données

On retrouve plusieurs façons de représenter et de ranger les données de base dans la mémoire des ordinateurs, chacune d'elles ayant des conséquences sur les différents langages de programmation utilisés et ce, à tous les niveaux. Les concepteurs du VAX 11/780 ont partagé les données en trois catégories dépendant de la nature même de ces données et de leur traitement. Il n'y a en fait que des chaînes de 0 et 1 à l'interne trouvant leur équivalence physique, ce ne sont que les algorithmes de reconnaissance de ces chaînes qui changent et qui leur donnent différentes interprétations.

A titre d'exemple, la chaîne binaire : 0 1 0 0 0 0 1 0 représente le caractère B (dans le code ASCII) pour l'interpréteur de caractère, mais cette même chaîne représentera la valeur 66 (en décimal) pour l'additionneur. Ces trois groupes de données se retrouvent exprimées dans le schéma suivant:

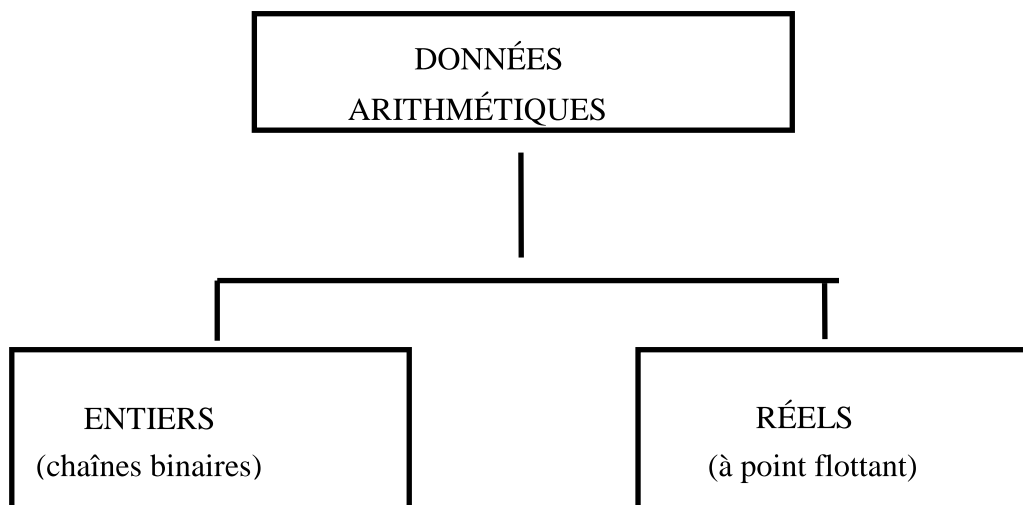


Pour chacune de ces catégories, il y a dans l'ordinateur un circuit électronique spécial permettant au processeur de traiter et d'interpréter les différentes chaînes binaires reçues de la mémoire.

L'aiguillage se fait à l'aide de signaux et de codes générés par l'unité de contrôle au moment opportun. Le résultat des traitements du processeur n'est ni plus ni moins que d'autres chaînes binaires retournées à la mémoire qui sont à leur tour codées dans l'une ou l'autre de ces trois catégories.

1.7.2 Les données de nature arithmétique

Les chaînes numériques en vue d'un traitement de nature arithmétique se subdivisent en deux groupes: les entiers et les réels, correspondant à deux codifications distinctes. Essentiellement, les entiers sont transformés en base 2 et la chaîne binaire obtenue est rangée tel quel en mémoire centrale (ou du moins à quelques modifications près). Les réels sont transformés selon la représentation dite " à point flottant".



Exemples 33

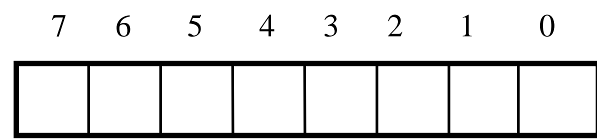
ENTIERS	**REELS**
5	3.14159
-122	0.0071
0	-9.8
8691	0.000000123
-273	782000000000000.

Essentiellement, un réel est identifié par la présence d'un point décimal dans la chaîne de caractères numériques contrairement à l'entier qui n'en a pas.

1.7.2.1 Les entiers

Les entiers transformés en chaînes binaires base 2 peuvent être représentés à l'interne de quatre façons sur le VAX, dépendant de la grandeur du nombre. L'unité fondamentale pour la représentation est l'octet (ou caractère) formé d'un regroupement de 8 bits consécutifs numérotés de 0 à 7 comme montré dans la figure suivante:

OCTET



On appelle:

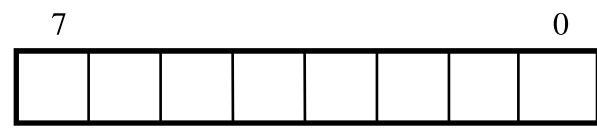
MOT un regroupement de deux octets ou de 16 bits consécutifs numérotés de 0 à 15,

DOUBLE-MOT un regroupement de deux mots ou de 4 octets ou de 32 bits consécutifs numérotés de 0 à 31,

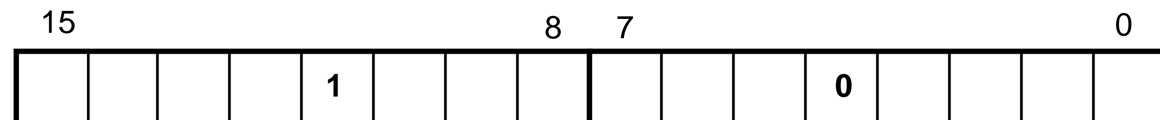
QUADRI-MOT un regroupement de deux double-mots ou de 4 mots ou de 8 octets ou de 64 bits consécutifs numérotés de 0 à 63.

On peut schématiser les quatre unités d'information de la façon suivante:

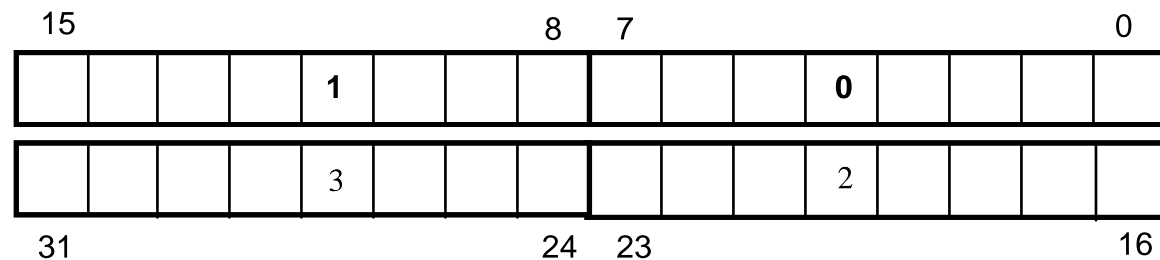
OCTET



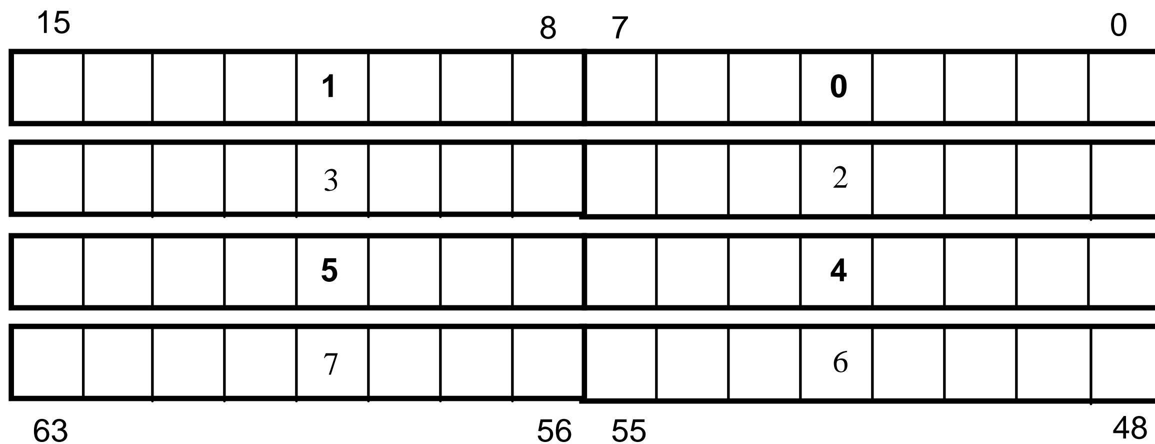
MOT



DOUBLE-MOT



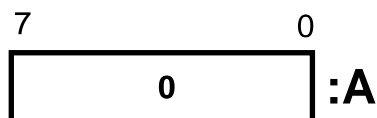
QUADRI-MOT



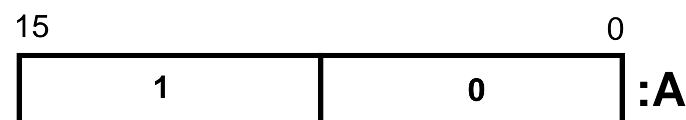
Tout nombre entier est représenté à l'interne dans l'un de ces types qui est précisé et choisi par le programmeur ou par le système d'exploitation.

L'unité de rangement de base est l'octet (caractère) et chaque octet est adressable. Selon la déclaration du type, l'ordinateur a accès à l'unité d'information prescrit (octet, mot, double-mot, quadri-mot) par son adresse symbolisée dans le schéma suivant par A.

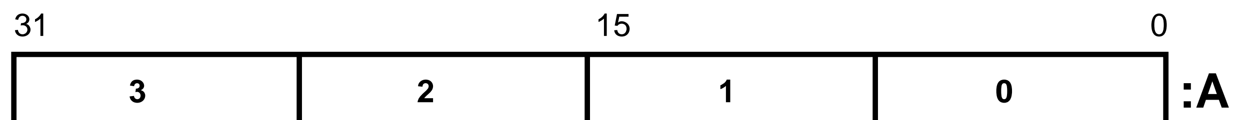
OCTET



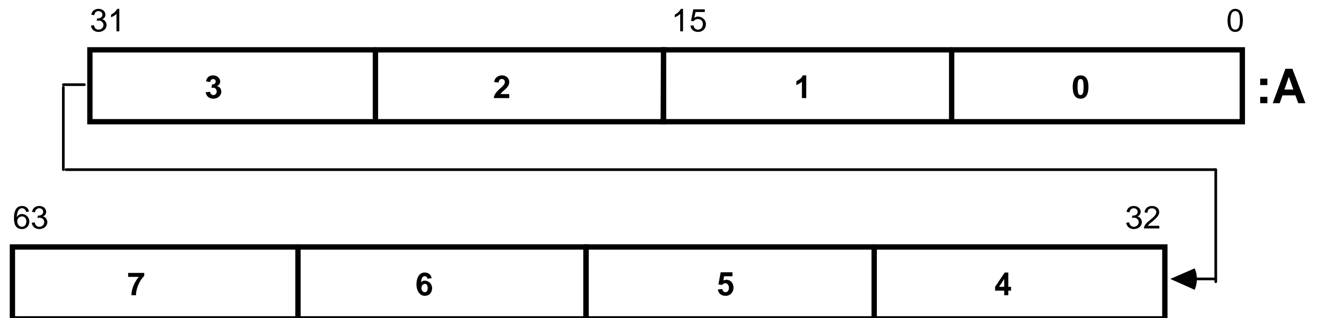
MOT



DOUBLE-MOT

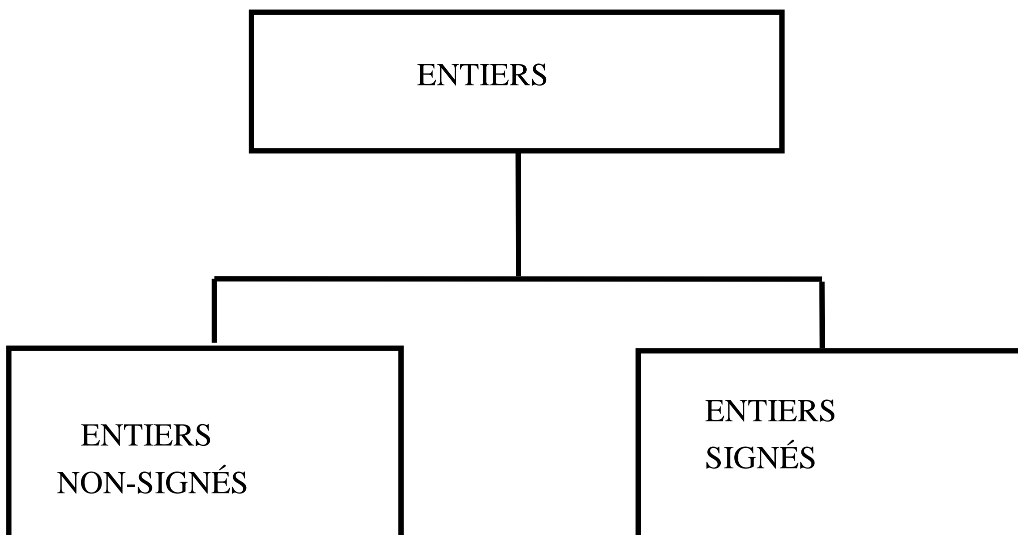


QUADRI-MOT



Par l`adresse A, l'ordinateur peut reproduire la chaîne binaire contenue dans l`unité de rangement correspondant à cette adresse pour fin de traitement (lecture en mémoire) ou encore y ranger une chaîne binaire résultant d'un traitement (écriture en mémoire). Notons qu'une lecture en mémoire préserve l'information lue, contrairement à l'écriture qui détruit à jamais l'ancien contenu pour faire place au nouveau.

L`ordinateur reconnaît deux sortes d'entiers: les entiers signés et les non-signés. Autrement dit, il fait la distinction entre deux nombres comme 4 et +4, le second étant signé contrairement au premier. On a donc la situation suivante:

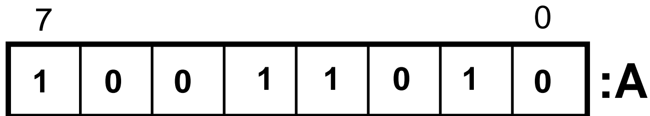


1.7.2.1.1 Les entiers non-signés

L'entier non-signé, nécessairement positif, est représenté par une chaîne binaire numérique de même valeur décimale que le nombre.

Exemple 34

On a : $154_{10} = 9A_{16} = 1001\ 1010_2$ logé en mémoire dans un octet va donner



L'intervalle des valeurs possibles dans une unité de rangement dépend du type de représentation déclarée, plus elle est longue en bits plus cet intervalle est grand. Pour le :

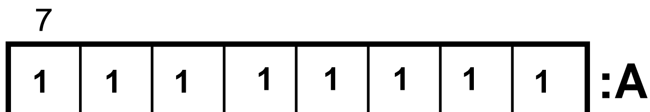
CARACTÈRE→ 0 à 255 ou (2^8- 1);

MOT → 0 à 65535 ou (2^{16}- 1);

DOUBLE-MOT → 0 à 4294967295 ou (2^{32}- 1) et

QUADRI-MOT → 0 à 18446744073709551615 ou (2^{64}- 1).

On peut facilement calculer ces valeurs limites d'intervalle. Par exemple pour le caractère, la plus grande valeur binaire qui puisse être contenue dans l'unité octet est formée uniquement d'états 1 pour tous les bits de position



donnant comme valeur décimale 255.

On peut comprendre que toute autre valeur supérieure à 255 ne pourra jamais être logée dans un caractère et une telle valeur provoquerait un débordement de capacité dans la codification caractère. Quant à la borne inférieure, elle est obtenue par des états 0 pour toutes les positions binaires.

Il en est de même pour les calculs des bornes des autres unités de rangement d'entiers tels que le mot, le double-mot et le quadri-mot.

1.7.2.1.2 Les entiers signés

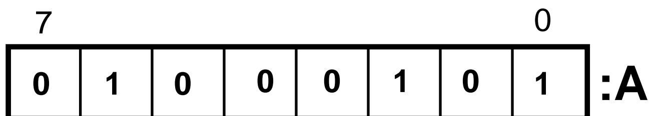
Pour l'entier signé, c'est le bit de plus fort poids correspondant à celui de plus haut numéro qui a la responsabilité du signe. S'il contient 0, alors le nombre est positif. La présence d'un 1 indique que le nombre est négatif. De plus, la représentation des nombres est celle du complément à 2. On sait que cette dernière fournit une chaîne binaire de même valeur décimale pour les entiers positifs avec la présence d'un 0 au bit de plus fort poids.

Pour les négatifs, les 0 sont changés en 1 et inversement donnant un 1 dans le bit de plus fort poids permettant d'identifier le signe négatif, c'est le complément à 1.

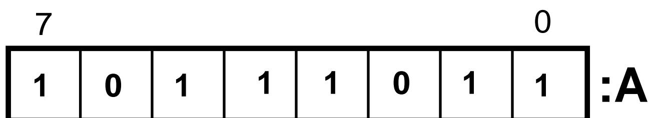
Le complément à 2 est obtenu par l'addition de 1 au complément à 1, ce qui permet d'avoir une représentation unique pour la valeur décimale nulle 0 et de gagner une position du côté négatif. Ainsi, le bit de plus fort poids est porteur du signe du nombre entier.

Exemple 35 :

Dans un caractère, +69 sera représenté en mémoire sous la forme suivante:



Tandis que -69 par



L'étendue des valeurs possibles pour les différents types est évidemment changée pour les nombres signés. Les longueurs en bits des unités de rangement étant les mêmes, ce qui est donné au négatifs est perdu pour les positifs.

Les situations sont présentées dans le tableau suivant pour les différentes codifications.

ÉTENDUE DES VALEURS ENTIERES

TYPE	SIGNÉ	NON-SIGNÉ
Caractère	128 à 127	0 à 255
Mot	-32768 à 32767	0 à 65535
Double-mot	-2^{31} à $2^{31} - 1$	0 à $2^{32} -1$

Ou -2 147 483 648 à 2 147 483 647 pour le type signé
Et 0 à 4 294 967 295 pour le type non-signé

Quadri-mot	-2^{63} à $2^{63}- 1$	0 à $2^{64} - 1$

Ou -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 pour le type signé
Et 0 à 18 446 744 073 709 551 615 pour le type non-signé

1.7.2.2 Les réels

Dans un nombre réel, par exemple -97.63, on retrouve le signe (+ ou -), la partie entière et la partie fractionnaire, de même que la base 10 qui est là par défaut. On a donc quatre caractéristiques fondamentales.

La représentation dite " à point flottant " d'un réel utilise aussi quatre paramètres qui pourraient à la rigueur être exprimés en termes des précédents. Ces quatre paramètres servant à décrire un réel sont liés sous la forme suivante:

$$N = s\,M \times B^{e}$$

où s désigne le signe (le + est souvent par défaut),
M représente la mantisse,
B donne la base
e désigne l'exposant.

Exemple 36 :

$-245.0 = -2.45 \times 10^{2}$

$0.0083 = 8.3 \times 10^{-3}$

$791000000000000 = 7.91 \times 10^{14}$.

Notons qu'un nombre donné peut se représenter sous plusieurs formes, toutes équivalentes.

Exemple 37 :

$$\begin{aligned} -0.0083 &= -8.3 \times 10^{-3} \\ &= -0.83 \times 10^{-2} \\ &= -83. \times 10^{-4} \end{aligned}$$

d'où l'expression " à point flottant ".

Elle est particulièrement utile pour l'écriture de très grands nombres comme celle de très petits.

Exemple 38 :

$0.00000000000000000000000123 = 1.23 \times 10^{-23}$

$123000000000000000000000000 = 1.23 \times 10^{23}$

Le format habituel en base 10 est d'écrire la valeur de la mantisse entre 1 et 10.

1.7.3 Représentation à point flottant

La représentation "à point flottant" de l'ordinateur utilise la base 2 naturellement. Pour tout nombre rationnel **r,** on a la forme suivante:

$$r = (+/-)\ m \times 2^{e}$$

où la mantisse m → $0,5 \leq m < 1$
et l'exposant e est représenté en binaire.

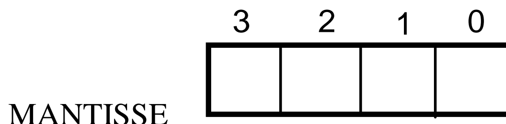
Une mantisse contenue dans ces limites est dite normalisée.

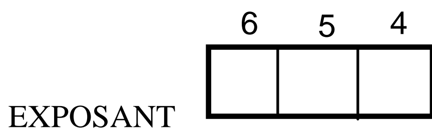
La mantisse **m** étant toujours supérieure à 0,5, elle débute donc toujours par un 1 dans sa représentation binaire. Les compagnies utilisent ce fait pour introduire ce qu'ils appellent le "bit caché", c'est-à-dire que le premier 1 dans la mantisse ne trouve pas de représentation physique et explicite à l'interne mais plutôt une représentation dite ¨logicielle¨. L'ordinateur élimine automatiquement ce 1 à l'écriture de la mantisse à l'interne et le re-génère par un circuit électronique approprié à la lecture. On gagne ainsi une position binaire pour la mantisse donc de la précision car c'est elle qui en est responsable comme on le verra plus loin.

Dépendant des compagnies, l'exposant peut être exprimé soit sous la forme du complément à 2, soit sous la forme binaire relative. Cette dernière est obtenue en ajoutant systématiquement à l'écriture de l'exposant (qui peut être négatif) une constante égale à la moitié de la valeur maximum de l'unité de rangement réservé à l'exposant pour obtenir une valeur écrite nécessairement positive. À la lecture, cette même constante est soustraite automatiquement à la valeur lue par un circuit électronique approprié. On dit que l'opération est réalisée de façon ¨câblée¨.

Quant au signe, une position binaire suffit en suivant une convention. Celle utilisée est toujours 0 pour indiquer le + et 1 pour marquer le - .

Exemple 39 : Supposons que l'on dispose d'une unité de rangement de quatre bits pour la mantisse, d'une de trois bits pour l'exposant.





et que l`on veuille représenter dans ces unités de rangement le nombre réel décimal -0.4306640625 . Peut-on le faire et si oui, avec quelle précision?

En premier lieu, il nous faut écrire le nombre en binaire. On trouve sans trop de difficulté l'égalité

$-0.4306640625_{10} = -0.0110111001_2$

Il faut maintenant normaliser la mantisse. On trouve

$-0.0110111001 = -0.110111001 \times 2^{-1}$.

La mantisse normalisée sera analysée de la façon suivante, s'il y a l'utilisation du concept de bit caché:

-0. 1   10111001
    ↑   ↑
bit caché   partie mémorisée

De plus la partie mémorisée sera tronquée ou arrondie, dépendant du fabricant, à quatre bits étant donnée la longueur de l'unité de rangement de la mantisse.

Supposons que l'on arrondit comme le font la plupart des compagnies d'ordinateurs. Les quatre bits retenus pour mémorisation seront

0.1011 1001, comme 1001 est plus grand que 0,5 on ajoute 1 aux 4 bits que nous gardons qui deviennent 1100, donc

MANTISSE

3	2	1	0
1	1	0	0

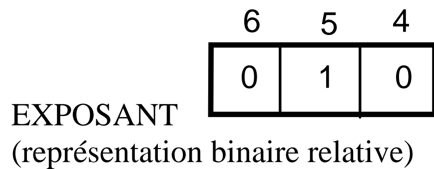
Quant à l'exposant, il y a deux possibilités de traitement. Pour celle du complément à 2 sur trois bits de position, -1 devient 1 1 1 .

Pour celle dite " binaire relative ", à -1 on ajoute la constante 3 donnant 2 en décimal ou 0 1 0 en binaire.

La constante vaut 3 car sur trois bits, la valeur maximale d'une telle unité de rangement est 1 1 1 en binaire ou 7 en décimal. La moitié de cette valeur maximale est donc 3 (il s'agit d'une division entière). On a donc les représentations suivantes comme possibilités:

EXPOSANT
(représentation complément à 2)

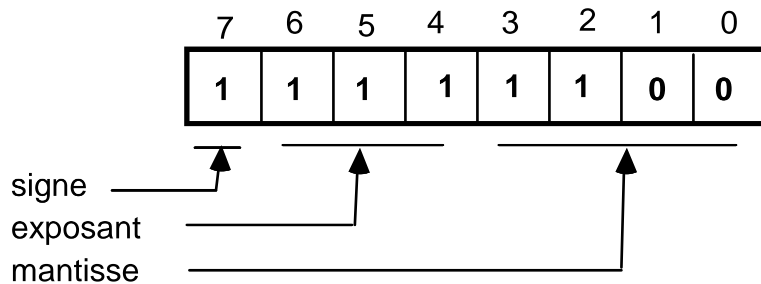
6	5	4
1	1	1



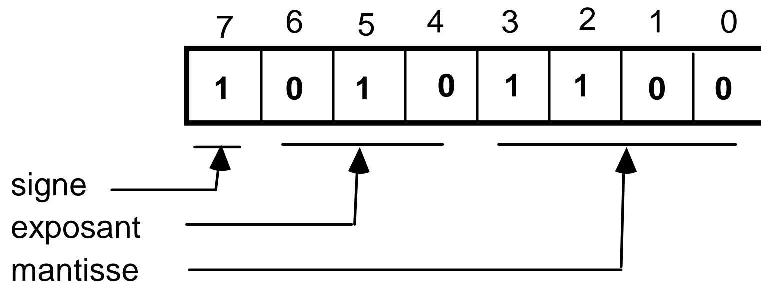
Pour le signe, il est négatif donc le bit de position porteur de cette information sera à l'état logique 1 selon la convention.

On obtient donc, en regroupant les morceaux, les deux représentations suivantes pour le réel -0.4306640625 :

La première utilisant le bit caché pour la mantisse et le complément à 2 pour l'exposant, ce qui donne

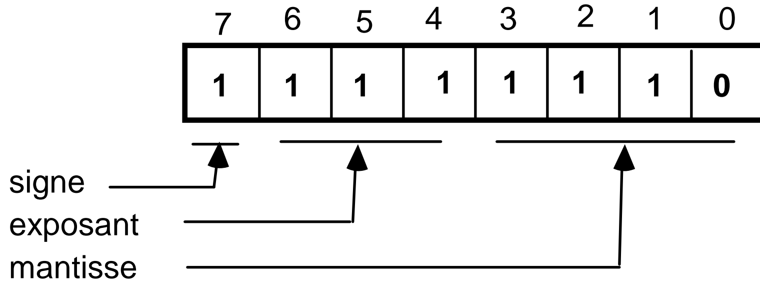


La deuxième utilisant le bit caché pour la mantisse et le binaire relatif pour l'exposant, on obtient

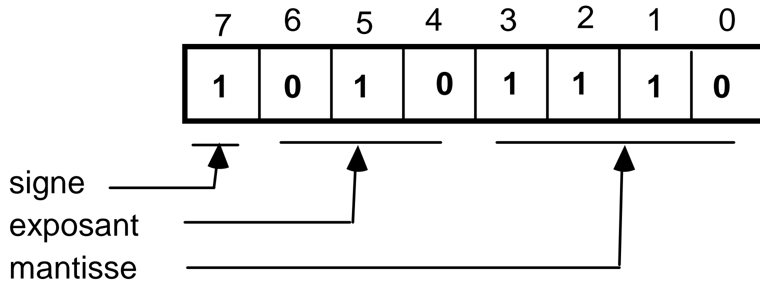


La non utilisation du bit caché pour la mantisse tout en utilisant l'arrondie nous fournit deux autres versions pour le même réel, soit:

La troisième n'utilisant pas le bit caché pour la mantisse et le complément à 2 pour l'exposant qui donne



La quatrième n'utilisant pas le bit caché pour la mantisse et le binaire relatif pour l'exposant qui conduit à



Notons que ne pas utiliser l'arrondie nous donnerait quatre autres représentations d'où la multiplicité des représentations possibles et des variantes d'un ordinateur à l'autre.

Notons également qu'à la lecture en mémoire de ce nombre, l'ordinateur déduira comme réel la valeur -0.4375 dans les quatre cas, commettant une erreur absolue de 0.0068359375 .

Cette erreur est due à la petite dimension de la mantisse.

La représentation à l`interne, comme le montre l'exemple précédent, exige de subdiviser l'unité de rangement déclaré (octet, mot, double-mot, quadri-mot) en trois zones:

une première pour le signe,

une seconde pour l'exposant et

une troisième pour la mantisse.

La longueur en bits de ces zones dépend du fabricant et peut varier d'un ordinateur à l'autre, de même que leur position relative. Le signe ne demande qu'un seul bit et c'est généralement le bit de plus fort poids qui en a la responsabilité. La convention habituelle est d'y loger un 0 pour les réels positifs ou un 1 pour les négatifs.

Quant à la zone réservée pour l'exposant, plus elle est longue en unités de bits, plus l'étendue des valeurs potentielles des réels sera grande. La longueur en bits par défaut de cette zone varie d'un ordinateur à l'autre. Elle est de 8 pour la plupart.

La zone responsable de la précision est celle de la mantisse. Plus elle peut loger d'unités binaires d'information, plus le nombre de chiffres significatifs dans les calculs sera élevé.

Généralement, la longueur totale de l'unité d'information pour l'ordinateur est fixe. On parle alors d'un micro-ordinateur 16 bits ou d'un mini-ordinateur 32 bits, etc. Par conséquent, la somme des trois longueurs des zones est fixe. Ainsi, l'étendue des valeurs possibles des réels et la précision sont en opposition.

Plus on accorde du pouvoir à l'un, plus l'autre s'affaiblit. Autrement dit, les bits donnés à l'un sont retirés à l'autre. Une plus grande étendue offre une moins bonne précision pour une unité d'information donnée et inversement.

Le VAX offre à ses programmeurs quatre formats comme unités d'information, ce sont:

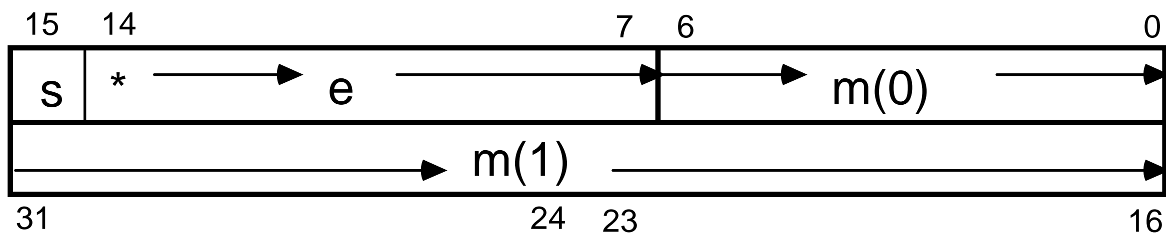
le format F (F-flotting) ou simple précision,

le format D (D-flotting) ou double précision,

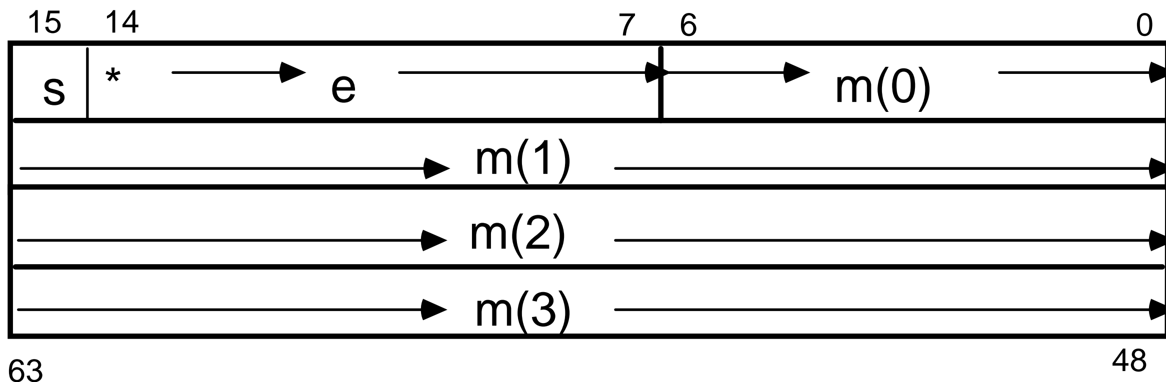
les formats G et H ou précision dite "précisions étendues"

On schématise ces quatre formats de la manière suivante où l'on indique pour chacun les trois zones
(s) signe - (e) exposant - (m) mantisse.

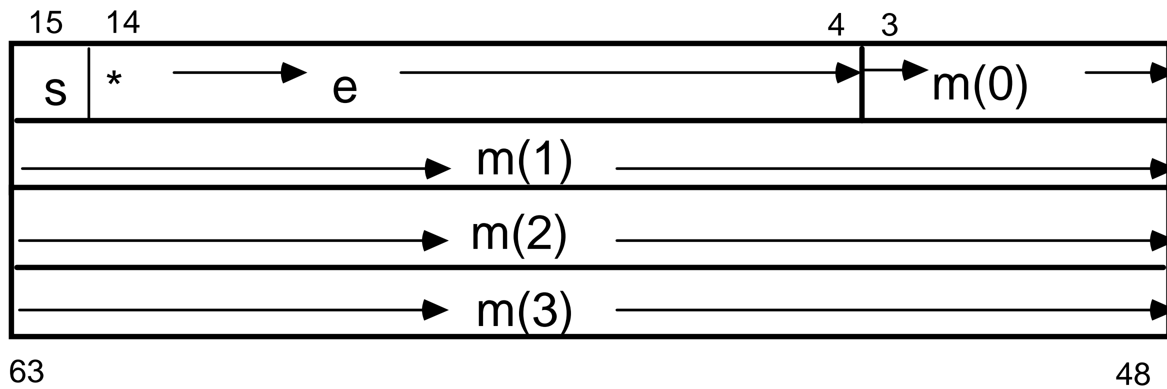
Le format F:



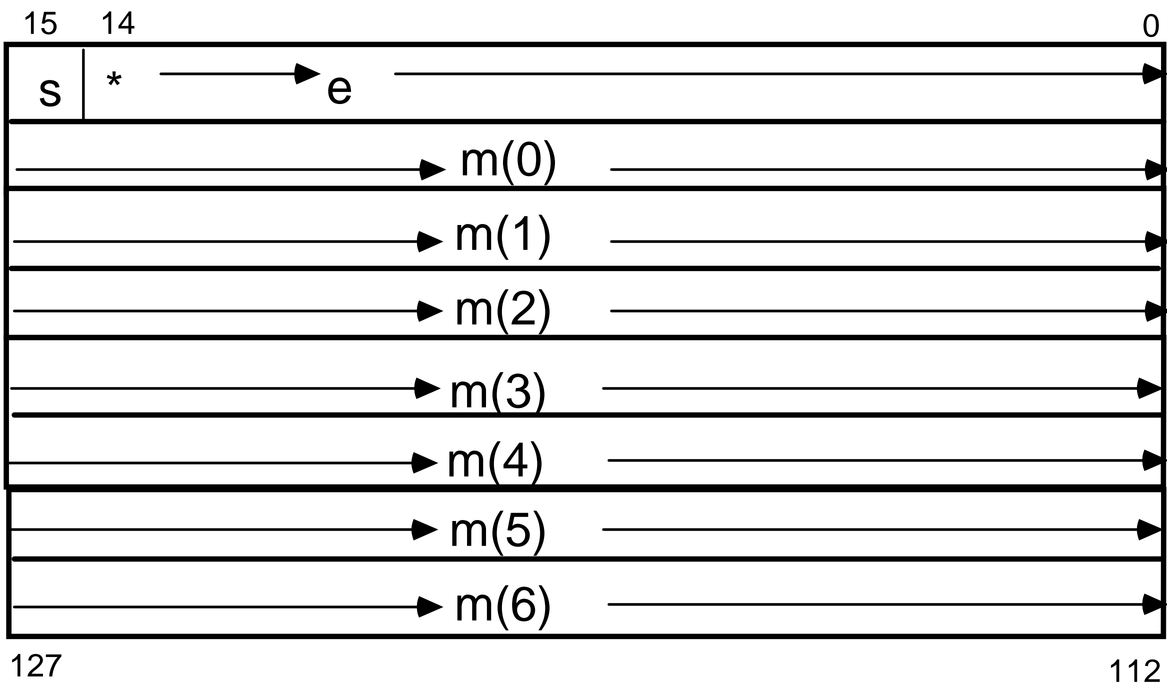
Le format D:



Le format G:



Le format H:



Les flèches montrent le sens de l'écriture de la mantisse débutant par l'étoile *, et dans l'ordre croissant des indices de *m*.

Valeurs spéciales

Dans la norme IEEE 754, on se réserve quelques valeurs pour l'exposent (les valeurs 000…00 et les 111…11) pour signifier que nous avons affaire à des valeurs spéciales ne pouvant pas être représentées par le format précédent utilisant le bit caché.

Zéro : En effet, parce que le bit cache suppose une mantisse commençant par 1, la valeur 0 ne peut être représentée comme ci-dessus. On réserve donc l'exposant composé de bits à 0 et la mantisse composée de bits à 0 pour représenter le nombre. Cela conduit à deux représentations pour 0 sur 32 bits:

0 | 0000 0000 | 000 0000 0000 0000 0000 0000 (le 0 « positif »)
1 | 0000 0000 | 000 0000 0000 0000 0000 0000 (le 0 « négatif »)

Les deux représentations de 0 doivent donner lieu à une égalité quand elles sont comparées.

Valeur dénormalisée: Si les bits de l'exposant sont tous des 0, mais la mantisse est différente de zéro, alors la valeur est *dénormalisée*, ce qui veut dire qu'il n'est plus supposé que la mantisse possède un bit caché. La valeur représente donc un nombre $(-1)^s \times 0.f \times 2^{-126}$. Pour une double précision, les nombres dénormalisés sont de la forme $(-1)^s \times 0.f \times 2^{-1022}$. Notons que de ce point de vue, le 0 peut être interprété comme un type spécial de nombres dénormalisés.

Infini: Les valeurs +infini et –infini sont représentées avec un exposant ayant des bits tous à 1 et une mantisse de bits tous égaux à 0. Le signe fait la distinction entre les +infini et –infini Ayant la possibilité de noter l'infini comme une valeur spécifique est utile car cela permet aux opérations de continuer même en cas de situations de débordement de capacité.

Situation d'erreur (NaN - *Not a Number*) est utilisée pour représenter une valeur qui ne représente pas un nombre réel. Les nombre NaN sont repésentés par le format ayant les bits de l'exposant égaux tous à et une mantisse différente de 0. Il existe deux catégories de NaN: QNaN (*Quiet NaN*) and SNaN (*Signalling NaN*).

1. Un QNaN est NaN ayant le bit le plus significatif égal à 1. Cette représentation est utile pour signifier que le résultat n'est pas défini mathématiquement.
2. Un SNaN est un NaN ayant le bit le plus significatif égal à 0. Cette situation est utile assigner une valeur non initialisée et en signaler un usage prématuré.

D'un point de vue sémantique, les QNaN's représentent les opérations indéterminées, tandis que les SNaN's représentent celles qui sont invalides.

Ci-dessous un résumé des valeurs spéciales

Signe	Exposant (*e*)	Mantisse (*m*)	Valeur
0	00..00	00..00	+0
0	00..00	00..01 : 11..11	Réels positifs dénormalisés $0.m \times 2^{(-b+1)}$
0	00..01 : 11..10	XX..XX	Réels positifs normalisés $1.m\times 2^{(e-b)}$
0	11..11	00..00	+infini
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Réels négatifs dénormalisés $-0.m\times 2^{(-b+1)}$
1	00..01 : 11..10	XX..XX	Réels négatifs normalisés $-1.m\times 2^{(e-b)}$
1	11..11	00..00	-infini
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11.11	QNaN

Opérations sur les valeurs spéciales

Les opérations arithmétiques sur les valeurs spéciales sont bien définies par la norme IEEE. Le tableau ci-dessous en fait résumé .

Operation	Résultat
n ÷ ±infini	0
±infinity × ±infinity	±infinity
±nonzero ÷ 0	±infini
infini + infini	infini
±0 ÷ ±0	*NaN*
infini - infini	*NaN*
±infini ÷ ±infini	*NaN*
±infini × 0	*NaN*

Pour plus de détails, consulter les adresses suivantes:

http://steve.hollasch.net/cgindex/coding/ieeefloat.html

http://www-etud.iro.umontreal.ca/~tsikhana/IFT1215-H10/files/ReprEnVirgFlot.pdf

1.7.3.1 Addition et soustraction en virgule flottante.

La première chose qu'il faut regarder avant de procéder à l' addition en virgule flottante, c'est de vérifier si les exposants des deux opérandes (chiffres qu'il faut additionner) ont la même valeur. Procédons à l'aide d'exemples.

Exemple : Supposons que l'on veuille faire l'addition des nombres $0.100110 * 2^4$ et $0.110101 * 2^1$. Il faut en premier lieu rendre les exposants égaux. On réalise cela en décalant de 3 positions binaires le point décimal du deuxième nombre pour obtenir ainsi $0.000110101 * 2^4$. Décaler d'une position vers la droite le point décimal revient à diviser la mantisse par 2. Pour conserver la valeur du nombre, à chaque décalage d'une position, on ajoute 1 à l'exposant. Aussi, 3 décalages vers la droite sont compensés par l'addition de 3 à l'exposant. Cependant, la mantisse étant étendue sur 6 bits vers la droite, les bits dépassant la 6ème position seront ignorés. Le deuxième nombre que la machine va considérer pour l'addition deviendra $0.000110 * 2^4$ si on tronque ou $0.000111 * 2^4$ si on arrondit, cela dépendant du fabricant de la machine. Par la suite, l'addition des deux mantisses se fera en donnant comme résultat la mantisse ainsi tronquée:

$$
\begin{array}{rl}
 & 0.100110\ (*2^4) \\
+ & \underline{0.000110}\ (*2^4) \\
 & 0.101100\ (*2^4)
\end{array}
$$

Ou bien avec la mantisse arrondie:

$$
\begin{array}{rl}
 & 0.100110\ (*2^4) \\
+ & \underline{0.000111}\ (*2^4) \\
 & 0.101101\ (*2^4)
\end{array}
$$

On réalise ici que le résultat d'une addition peut être très imprécis en virgule flottante. En termes clairs, l'addition demandée en décimale était :

35/2 + 117/64 qui devrait donner comme résultat 1237/64 ou 19+21/64 19.32815.

Les résultats obtenus sont:

avec la mantisse tronquée: 35/2 + 3/2 = 19
avec la mantisse arrondie: 35/2 + 7/4 = 77/4 = 19.25.

L'erreur commise est dans le premier cas de 0.32815 et dans le second de 0.07815 . Cette erreur est d'autant plus importante que la différence entre les exposants est grande. Notons que l'erreur commise est moins grande pour l'arrondi que pour la troncature.

En analyse numérique, on démontre que pour la somme de n nombres, la précision est plus grande si on ordonne d'abord les nombres en ordre croissant pour les additionner l'un après l'autre en partant du plus petit au plus grand. Cet algorithme réduit au minimum la différence entre les exposants à chaque addition successive de deux nombres.

Donnons un autre exemple encore plus frappant démontrant que l'addition en virgule flottante n'est pas très précise dans les cas où les exposants sont très différents.

Effectuons l'addition des deux nombres suivants exprimés sur cinq bits:

$0.10110 * 2^3 + 0.10011 * 2^{-2}$. On ramène les exposants égaux, c'est-à-dire le plus petit vers le plus grand, ce qui donne :

$0.10110 * 2^3 + 0.0000010011 * 2^3$. Sur cinq bits, la machine considèrera les deux nombres

$0.10110 * 2^3 + 0.00000 * 2^3 = 0.10110 * 2^3$. La machine dans ce cas est incapable d'additionner le deuxième nombre correctement.

Pour contrer cette lacune, on prévoit un registre ayant une capacité plus grande que cinq bits ou, en général, celle des registres contenant les opérandes. Généralement, le registre résultat est double par rapport à celui des opérandes.

Une autre difficulté peut survenir dans l'addition en virgule flottante. En effet, qu'arrive-t-il s'il y a une retenue sur le bit d'extrême gauche lors de l'addition des mantisses normalisées? N'oublions pas que ce bit de position est réservé dans le registre au signe du nombre.

On solutionne ce problème en normalisant la mantisse du résultat obtenu en prévoyant pour ce cas un décalage d'une position vers la gauche du point décimal tout en ajoutant 1 à l'exposant.

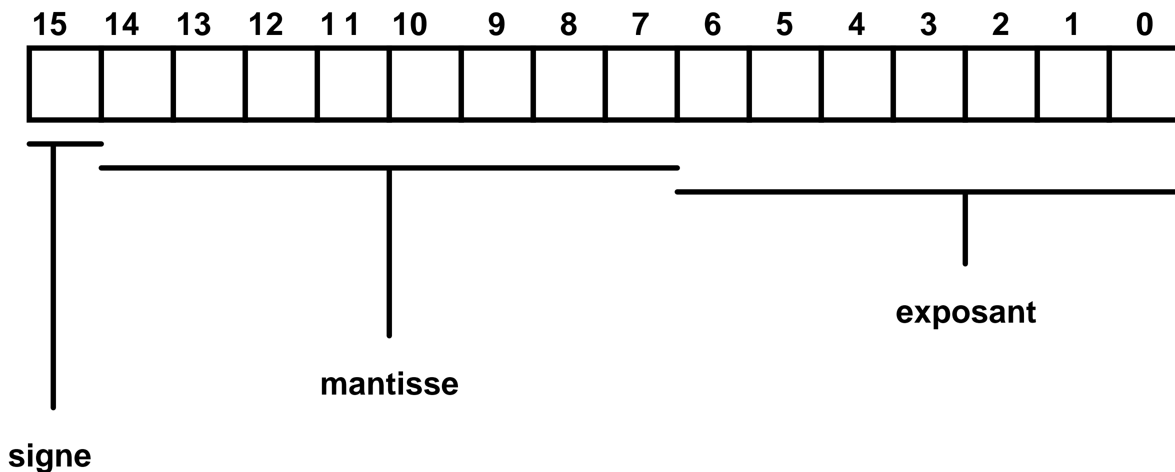
Exemple :

Considérons l'addition des deux nombres $0.110111 * 2^3$ et $0.1011011 * 2^2$. En ramenant les exposants égaux, on trouve $0.110111 * 2^3$ et $0.01011011 * 2^3$ et l'addition des mantisses donne $1.0011011 * 2^3$, d'où la retenue sur le bit du signe. Une normalisation du résultat donnera $0.10011011 * 2^4$ ou sur six bits avec arrondi $0.100111 * 2^4$.

Tant qu'à la soustraction, la même procédure s'applique. Notons que la méthode illustrée plus haut ne s'applique que pour le cas où les mantisses sont normalisées. L'algorithme n'est pas le même si ces mantisses ne le sont pas.

Voyons un dernier exemple où l'on prévoit un double registre pour le résultat, l'exposant exprimé en complément à 2 et l'arrondi de la mantisse résultante.

Exemple : Considérons la configuration suivante pour notre arithmétique en virgule flottante.



Supposons que l'on veuille additionner 27.825 et -4.625. On a:

27.825_{10} $= 11011.110100110011001..._2$
$= 0.11011110100110011001.. * 2^5$ et
$- 4.625_{10}$ $= - 100.101_2$
$= - 0.100101 * 2^3$.

On a donc pour les deux opérandes où les mantisses sont normalisées et arrondies (il y a perte de précision dans le cas du premier opérande), les exposants sont en complément à 2 sur 7 bits:

Opérande 1: Mantisse = 0.11011111
Exposant = 0000101
Signe = 0

Opérande 2: Mantisse = 0.10010100
Exposant = 0000011
Signe = 1.

Les registres ont donc comme contenus:

Opérande 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	0	0	0	0	1	0	1

Opérande 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1

L'ajustement des exposants est par la suite effectué, ce qui donne comme contenus des registres:

Opérande 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	0	0	0	0	1	0	1

Opérande 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1

L'addition des mantisses est par la suite faite dans un double registre, ce qui donne:

```
   11011111
-  00100100
   10111011
```

Le double registre du résultat aura comme contenu:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0

Par la suite, la mantisse du résultat sera livrée sur 8 bits en l'arrondissant, ce qui donne: 0.10111011 et le registre contenant le résultat final aura comme contenu:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	1	0	0	0	0	0	1	0	1

étant interprété comme le nombre suivant:

$0.10111011 * 2^5 = 10111.011_2 = 23.3125_{10}$.

Notons l'erreur commise de 0.1125_{10} car la réponse exacte est 23.2_{10} .

1.7.3.2 Multiplication et division en virgule flottante.

Considérons deux nombres N et M écrits dans la représentation virgule flottante, représentation équivalente à la forme exponentielle

$N = C_1 * B^{e1}$ et $M = C_2 * B^{e2}$.

Le produit donnera comme résultat:

$M * N = C_1 * C_2 * B^{(e1+ e2)}$

et le quotient donnera:

$\mathbf{M / N = C_1 / C_2 * B^{(e1- e2)}}$.

On peut donc résumer la multiplication par la multiplication des mantisses normalisées et par l'addition des exposants. Nous n'avons pas besoin de rendre les exposants égaux comme dans l'addition et la soustraction. Remarquons également qu'étant donné que les mantisses normalisées sont comprises entre 1 et 1/2, leur produit donnera toujours un nombre entre 1 et 1/4; par conséquent la multiplication nécessite tout au plus un seul décalage vers la droite du point décimal pour rendre la mantisse du résultat normalisée.

Tant qu'à la division, elle se résume à la division des mantisses et à la soustraction des exposants. La mantisse résultante sera obligatoirement comprise entre 2 et 1/2, ce qui entraînera tout au plus un décalage vers la gauche du point décimal.

Les algorithmes de multiplication et de division des nombres binaires ont déjà été vus. On peut constater que la multiplication et la division sont plus efficaces pour les nombres exprimés en virgule flottante en comparaison des opérations addition et soustraction du point de vue vitesse d'opération et précision.

1.7.4 Les chaînes de caractères

Nous savons que toute information fournie ou reçue au terminal se présente sous forme de caractères ou chaîne de caractères. Par exemple,

*** +) ; =**
" ceci est une chaîne de caractères "
401, rue Principale.

sont des chaînes de caractères. L'ordinateur utilise pour chacun d'eux, y compris le blanc, un code binaire. Le code ASCII est nettement le plus répandu que l'on retrouve dans les tables fournies par le fabricant.

Exemples 40

On a pour les caractères suivants la codification correspondante en ASCII:

A	**7**	**$**	**?**	→ Caractère
65	55	36	63	→ Décimal
41	37	24	3F	→ Hexadécimal
0100 0001	0011 0111	0010 0100	0011 1111	→ Binaire

Le VAX subdivise les chaînes en deux groupes:

les chaînes numériques,

les chaînes non-numériques.

Exemple 41 :

252 rue De L'Espérance → est une chaîne non-numérique,

tandis que

+814 → est une chaîne numérique.

1.7.4.1 Les chaînes non-numériques

Le paramètre important d'une chaîne non numérique est le nombre de caractères de la chaîne. A l'interne, elle se retrouve dans une pile où chaque caractère occupe une unité de rangement de 8 bits (octet) codé en ASCII référée par une adresse au caractère de tête.

Exemple 42

La chaîne G7H2B1 donne la représentation suivante en mémoire codée en ASCII

Représentation hexadécimale

	7 6 5 4	3 2 1 0
G	4	7
7	3	7
H	4	8
2	3	2
B	4	2
1	3	1

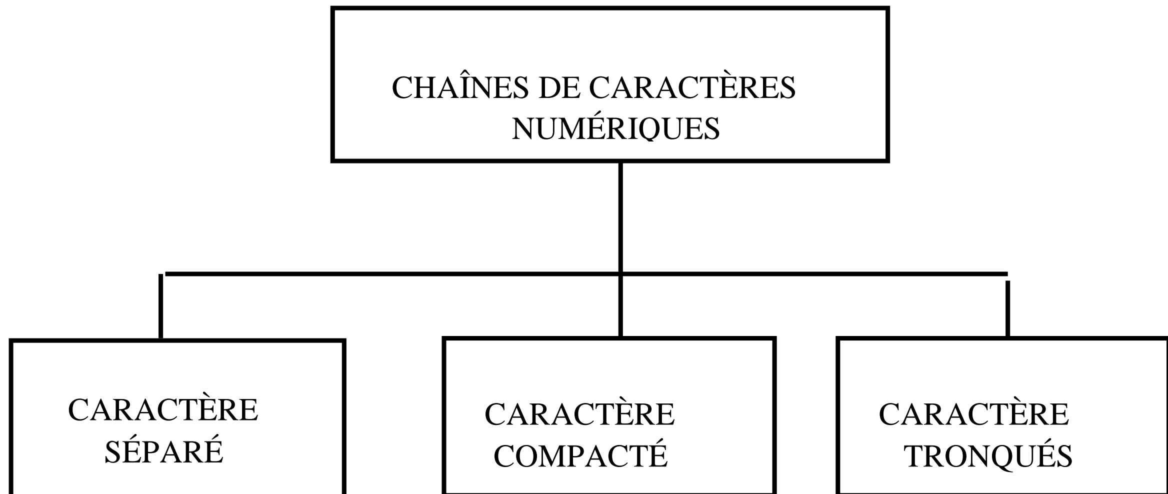
Représentation binaire

	7	6	5	4	3	2	1	0
G	0	1	0	0	0	1	1	1
7	0	0	1	1	0	1	1	1
H	0	1	0	0	1	0	0	0
2	0	0	1	1	0	0	1	0
B	0	1	0	0	0	0	1	0
1	0	0	1	1	0	0	0	1

1.7.5 Les chaînes numériques

Il y a sur le VAX, trois façons de les coder:

1. codification à caractères séparés,
2. codification à caractères compactés et
3. codification à caractères tronqués.



Voyons ces différentes codifications. Nous verrons que la dernière, celle à caractères tronqués, permet une économie substantielle en mémoire interne.

1.7.5.1.1 La codification à caractères séparés.

Cette codification est similaire en tout point à la codification des chaînes de caractères non numériques. C'est-à-dire que les caractères sont rangés dans une pile, un caractère par octet, un après l'autre, et codé en ASCII.

Exemple 43 : La chaîne numérique +/-814 donnera à l'interne :

+	-	8	1	4
2B	2D	38	31	34

+ / -	2	B/D	: A
8	3	8	N = 4
1	3	1	
4	3	4	

+ / -	0010	1011/1101	: A
8	0011	1000	N = 4
1	0011	0001	
4	0011	0100	

Le nombre d'unités de rangement de 8 bits est égal au nombre de caractères de la chaîne numérique.

1.7.5.1.2 La codification à caractères compactés

La façon de ranger l'information est de mémoriser les caractères codés en ASCII un par octet et en pile à l'exception du dernier de la chaîne qui absorbe le signe. On y trouve un 7 (ou 0111 en binaire) si le nombre à mémoriser est négatif, ou un 3 (ou 0011) si s'agit d'un nombre positif. On sauve ainsi un emplacement de 8 bits par nombre à mémoriser. Par exemple, la chaîne +/-814 donne à l'interne:

Exemple 44 :

8	3	8	: A
1	3	1	N = 3
+ / - 4	3/7	4	

8	0011	1000	: A
1	0011	0001	N = 3
+ / - 4	0011/0111	0100	

1.7.5.1.3 La codification à caractères tronqués

Dans le code ASCII, les caractères numériques sont tels que la première partie du code, c'est-à-dire les quatre premiers bits, est toujours 3 (ou 0011 en binaire) tandis que la deuxième partie transcrit la valeur décimale du caractère en binaire. Notons que l'on indique le code ASCII d'un caractère le plus souvent dans la notation hexadécimale, ou encore dans la notation décimale correspondant à l'ordre du caractère dans la liste de l'ensemble des codes ASCII. Pour les caractères numériques, on a explicitement:

Aussi, les fabricants ont exploité ce fait en créant des circuits électroniques permettant d'éliminer automatiquement la première partie du code des caractères numériques et de ne ranger en mémoire que la seconde partie du code. On a ainsi une importante économie d'espace car cette méthode permet de ranger deux caractères numériques par unité de rangement de 8 bits. Tant qu'au signe, c'est la deuxième partie du dernier octet qui en a la responsabilité où on y trouve un C (ou 1100 en binaire) ou un D (ou 1101) dépendant qu'il s'agit d'un nombre positif ou négatif.

Exemple 45 : La chaîne +/-814 ne prendra que deux emplacements de 8 bits comme illustré dans la figure suivante:

8	1
4	C / D

: A
N = 2

1000	0001
0100	1100/1101

: A
N = 2

Notons que l'ordinateur est obligé de générer un zéro au début d'une chaîne numérique qui contient un nombre impair de caractères afin d'assurer la représentation du signe à l'endroit prescrit plus haut, ce qu'il fait automatiquement à l'aide d'un circuit électronique approprié.

Exemple 46 : + 1372146

1	3
7	2
1	4
6	C

: A
N = 4

0001	0011
0111	0010
0001	0100
0110	1100

: A
N = 4

Exemple 47 : + 137214

0	1
3	7
2	1
4	C

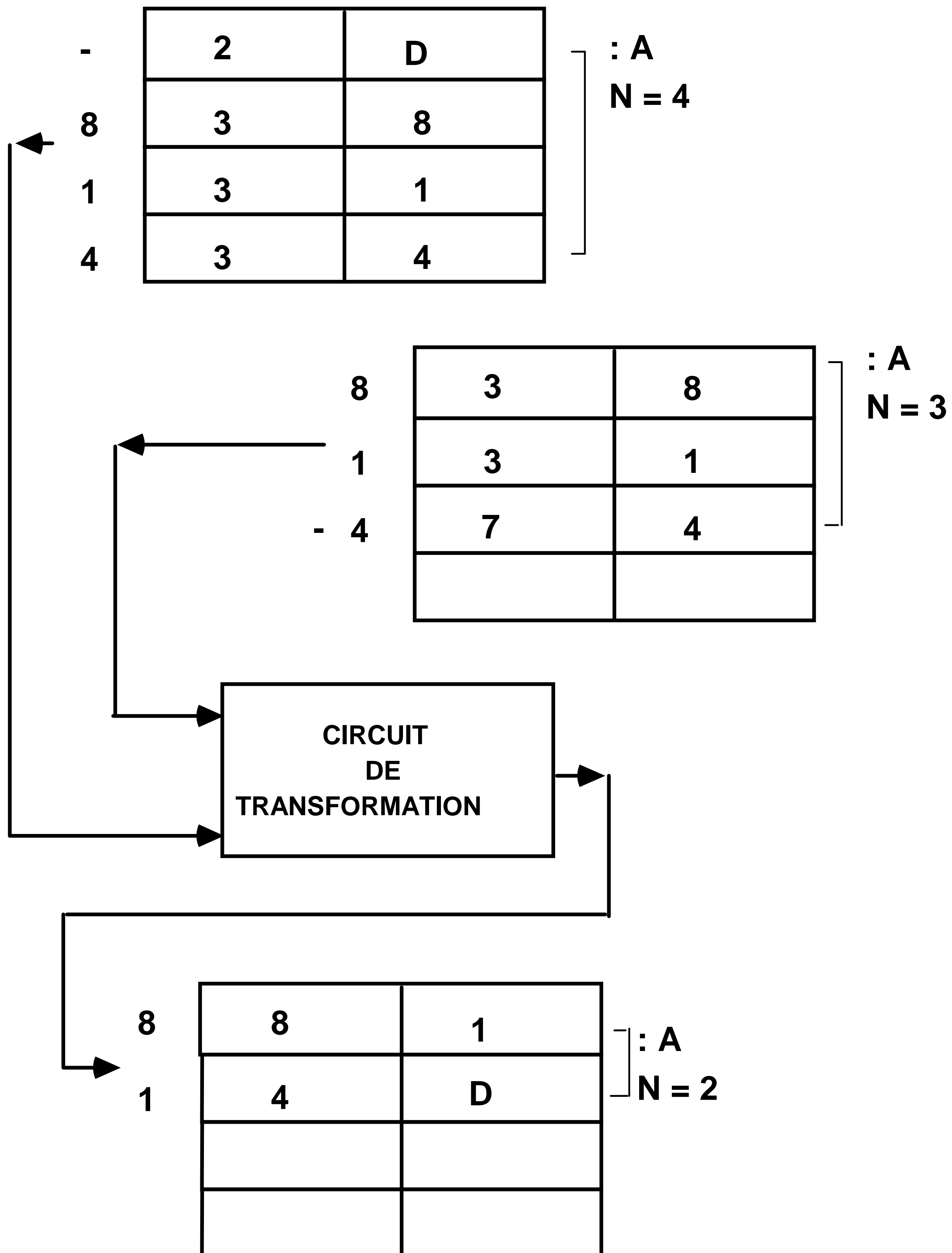
: A
N = 4

0000	0001
0011	0111
0010	0001
0100	1100

: A
N = 4

Dans le second exemple, l'ordinateur considère la chaîne numérique +0137214 au lieu de +137214 pour rendre le nombre de caractères pair. Ceci a pour effet d'assurer qu'un chiffre soit retenu dans la première partie du dernier octet et que l'indication du signe soit logée dans la deuxième partie du même octet. Le zéro se trouve logé dans la première partie du premier octet.

Cette technique de rangement, vu son efficacité, est utilisée constamment par l'ordinateur transformant à l'aide d'un circuit électronique approprié les chaînes purement numériques aussitôt identifiées.


-
2
D
: A
N = 4
8
3
8
1
3
1
4
3
4
8
3
8
: A
N = 3
1
3
1
- 4
7
4
CIRCUIT
DE
TRANSFORMATION
8
8
1
: A
1
4
D
N = 2

1.7.6 Les chaînes binaires à champs variables

C'est une technique de rangement des chaînes binaires qui permet de la loger à n'importe quel endroit dans la mémoire. De plus la longueur de la chaîne en bits peut être comprise entre 0 et 32. Une telle souplesse de localisation exige trois paramètres:

1. une adresse de référence A qui permettra de localiser par rapport à elle le début de l'unité de rangement,
2. une distance P en bits précisant l'endroit du début de l'unité de rangement par rapport à l'octet d'adresse A et
3. une longueur L en bits de l'unité de rangement précisant qu'il s'agit des L bits suivant l'endroit indiqué par le paramètre P.

Le schéma suivant montre la position des valeurs positives et négatives du paramètre P par rapport à l'adresse A.

	-25							-32	
M	-17	-18	-19	-20	-21	-22	-23	-24	
É	-9	-10	-11	-12	-13	-14	-15	-16	
M	-1	-2	-3	-4	-5	-6	-7	-8	
O	7	6	5	4	3	2	1	0	: A
I	15	14	13	12	11	10	9	8	
R	23	22	21	20	19	18	17	16	
E	31	30	29	28	27	26	25	24	
	39	38	37	36	35	34	33	32	

Le schéma suivant montre l'utilité de ces trois paramètres A, P et L dans un cas particulier.

MÉMOIRE

P = 36

L = 9

4 3 2 1

9 8 7 6 5

L'étendue des valeurs possibles pour le paramètre P est de

$$-2^{31} \text{ à } 2^{31} - 1 .$$

ou

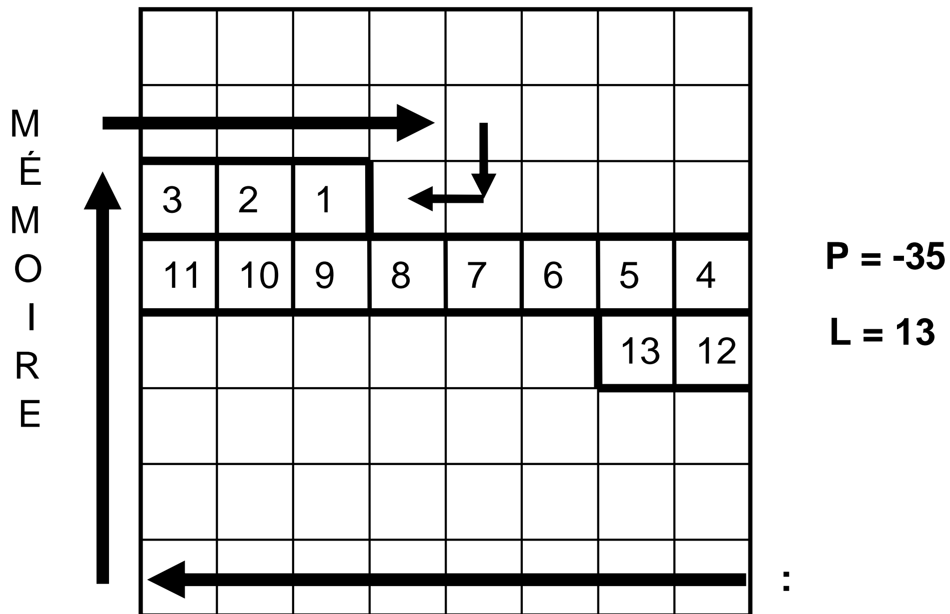
$$-2\,147\,483\,648 \text{ à } 2\,147\,483\,647$$

Celle de L va de 0 à 32 bits.

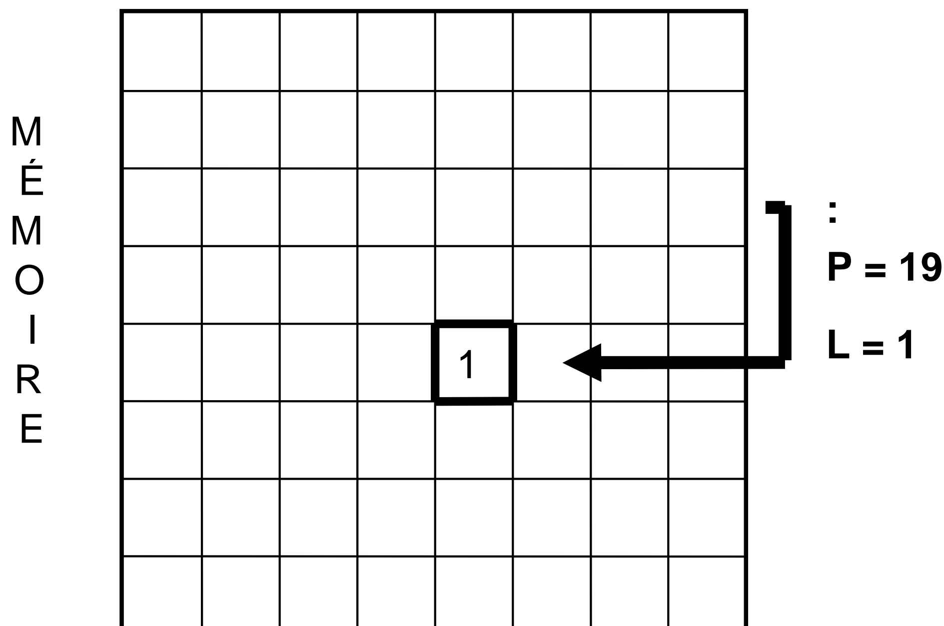
Exemple 48

Les exemples suivants montrent le cas où P est négatif et le cas où L est égal à 1 . On peut donc aller lire ou écrire dans un bit isolé, l'accès étant possible par cette technique.

Le cas où P < 0:



Le cas où L = 1 :



Cette technique de localisation est largement utilisée par le système d'exploitation.

1.8. Les jeux de caractères

Tous les ordinateurs n'utilisent pas le même jeu de caractères, mais ASCII (American Standard Code for Information Interchange) est le plus courant. Ce jeu de caractères est un code à 7 bits. Sur les 128 possibilités, 33 sont réservées pour des fonctions de “contrôle”, tel le retour de chariot et le saut de ligne. Malheureusement, peu de ces caractères suivants ont conservé leur signification originale.

Les 95 caractères restants sont utilisés pour représenter les lettres majuscules et minuscules, les chiffres et les caractères graphiques. Une valeur numérique destinée à être utilisée à l'intérieur de l'ordinateur est assignée à chacun de ces caractères. Entre autres, c'est cette valeur que l'on obtient quand on utilise la fonction Pascal standard ORD. Les avantages de l'ASCII sont le fait qu'il soit standard, qu'il soit largement répandu et que les lettres, tant majuscules que minuscules, soient contiguës.

L'EBCDIC (Extended Binary Coded Decimal Interchange Code) est un autre jeu de caractères. Il s'agit d'un code à 8 bits basé sur le code à 7 bits BCDIC. Il est utilisé sur les gros ordinateurs IBM. Un des désavantages de l'EBCDIC est que certains caractères peu fréquemment utilisés viennent s'intercaler au beau milieu des lettres majuscules et des lettres minuscules.

Il existe encore d'autres jeux de caractères. Mentionnons seulement le jeu de caractères scientifiques de CDC qui est un code à 6 bits.

Table ASCII STANDARD

Elle comporte 128 codes. Les 32 premiers codes ASCII (000 à 031) sont des **codes de commande** et forment un ensemble particulier de caractères non imprimables. Le second bloc de 32 codes ASCII (032 à 063) comportent de divers symboles de ponctuation, des caractères spéciaux, et les symboles numériques. Le troisième groupe de 32 codes ASCII (064 à 095)est affecté aux lettres alphabétiques majuscules et à six symboles spéciaux. Le quatrième bloc de 32 codes (096 à 127) est réservé aux symboles alphabétiques minuscules, à cinq symboles spéciaux supplémentaires et à un code de commande (DEL / effacer).

Table ascii standard

HEX	DEC	CHR	CTRL		HEX	DEC	CHR		HEX	DEC	CHR		HEX	DEC	CHR
00	0	NUL	^@		20	32	SP		40	64	@		60	96	`
01	1	SOH	^A		21	33	!		41	65	A		61	97	a
02	2	STX	^B		22	34	"		42	66	B		62	98	b
03	3	ETX	^C		23	35	#		43	67	C		63	99	c
04	4	EOT	^D		24	36	$		44	68	D		64	100	d
05	5	ENQ	^E		25	37	%		45	69	E		65	101	e
06	6	ACK	^F		26	38	&		46	70	F		66	102	f
07	7	BEL	^G		27	39	'		47	71	G		67	103	g
08	8	BS	^H		28	40	(		48	72	H		68	104	h
09	9	HT	^I		29	41	)		49	73	I		69	105	i
0A	10	LF	^J		2A	42	*		4A	74	J		6A	106	j
0B	11	VT	^K		2B	43	+		4B	75	K		6B	107	k
0C	12	FF	^L		2C	44	,		4C	76	L		6C	108	l
0D	13	CR	^M		2D	45	–		4D	77	M		6D	109	m
0E	14	SO	^N		2E	46	.		4E	78	N		6E	100	n
0F	15	SI	^O		2F	47	/		4F	79	O		6F	111	o
10	16	DLE	^P		30	48	0		50	80	P		70	112	p
11	17	DC1	^Q		31	49	1		51	81	Q		71	113	q
12	18	DC2	^R		32	50	2		52	82	R		72	114	r
13	19	DC3	^S		33	51	3		53	83	S		73	115	s
14	20	DC4	^T		34	52	4		54	84	T		74	116	t
15	21	NAK	^U		35	53	5		55	85	U		75	117	u
16	22	SYN	^V		36	54	6		56	86	V		76	118	v
17	23	ETB	^W		37	55	7		57	87	W		77	119	w
18	24	CAN	^X		38	56	8		58	88	X		78	120	x
19	25	EM	^Y		39	57	9		59	89	Y		79	121	y
1A	26	SUB	^Z		3A	58	:		5A	90	Z		7A	122	z
1B	27	ESC			3B	59	;		5B	91	[		7B	123	{
1C	28	FS			3C	60	<		5C	92	\		7C	124	|
1D	29	GS			3D	61	=		5D	93	]		7D	125	}
1E	30	RS			3E	62	>		5E	94	^		7E	126	~
1F	31	US			3F	63	?		5F	95	_		7		

Signification des codes de contrôle

0	:	NUL	:	Nul
1	:	SOH	:	Début d'en-tête
2	:	STX	:	Début de texte
3	:	ETX	:	Fin de texte
4	:	EOT	:	Fin de transmission
5	:	ENQ	:	Demande
6	:	ACK	:	Accusé de réception
7	:	BEL	:	Sonnerie
8	:	BS	:	Retour arrière
9	:	HT	:	Tabulation horizontale
10	:	LF	:	Interligne
11	:	VT	:	Tabulation verticale
12	:	FF	:	Avance papier
13	:	CR	:	Retour de chariot
14	:	SO	:	Hors-code
15	:	SI	:	En-code
16	:	DLE	:	Échappement transmission
17	:	DC1	:	Commande d'appareil auxiliaire 1
18	:	DC2	:	Commande d'appareil auxiliaire 2
19	:	DC3	:	Commande d'appareil auxiliaire 3
20	:	DC4	:	Commande d'appareil auxiliaire 4
21	:	NAK	:	Accusé de réception négatif
22	:	SYN	:	Synchronisation au repos
23	:	EOB	:	Fin de bloc de transmission
24	:	CAN	:	Annulation
25	:	EM	:	Fin de support
26	:	SUB	:	Substitution
27	:	ESC	:	Échappement
28	:	IFS	:	Séparateur de fichier
29	:	IGS	:	Séparateur de groupe
30	:	IRS	:	Séparateur d'article
31	:	IUS	:	Séparateur de sous-article
127	:	DEL	:	Oblitération