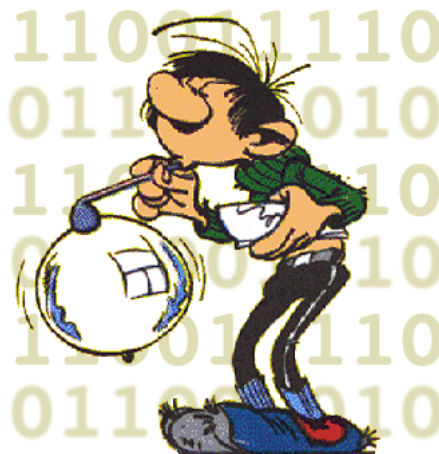


Le Binaire



Jean-Michel L ry

1. Le binaire, pourquoi ?

- 1.1. Des transistors au binaire
- 1.2. Le bit et l'octet
- 1.3. L'unité de mesure : L'octet
- 1.4. Le binaire pour comprendre l'ordinateur

2. Les différentes bases de calcul

- 2.1. La base Décimale : (base 10)
- 2.2. La base Octale : (base 8)
- 2.3. La base Hexadécimale : (base 16)
- 2.4. La base Binaire : (base 2)
- 2.5. Tableau récapitulatif des bases

3. La conversion du binaire vers le décimal

- 3.1. La formule mathématique
- 3.2. Utilisation de la base octale
 - 3.2.1. Méthode
 - 3.2.2. Première étape
 - 3.2.3. Deuxième étape
- 3.3. Utilisation de la base hexadécimale
 - 3.3.1. Première étape
 - 3.3.2. Deuxième étape
- 3.4. Conversion directe
- 3.5. Décomposition en puissance de 2

4. La conversion du décimal vers le binaire

- 4.1. Le principe
- 4.2. Utilisation de la base octale
 - 4.2.1. Méthode
 - 4.2.2. Première étape
 - 4.2.3. Deuxième étape
- 4.3. Utilisation de la base hexadécimale
 - 4.3.1. Première étape
 - 4.3.2. Deuxième étape
- 4.4. Conversion directe

5. Capacité de mémorisation de 1, 2, .. N octets

- 5.1. Capacité de mémorisation
- 5.2. Les limitations

6. Domaines d'application du binaire

- 6.1. Couleur en HTML**
- 6.2. Réseau : Adressage MAC et IP**
- 6.3. Couleur d'un écran**

7. Représentation binaire des données : le codage

- 7.1. Le codage "quantité" (entier non signé)**
- 7.2. Le codage entier**
 - 7.2.1. Le principe**
 - 7.2.2. Entier positif**
 - 7.2.3. Entier négatif**
 - 7.2.4. Les limites**
- 7.3. Le codage réel**
 - 7.3.1. La notation en virgule flottante**
 - 7.3.2. Le principe**
 - 7.3.3. Le vrai codage**
 - 7.3.4. Les limites du codage**
 - 7.3.5. Les erreurs de précision**
- 7.4. Le codage caractère**
 - 7.4.1. Le principe**
 - 7.4.2. Les tables normalisées**
 - 7.4.2.1. La table ISO**
 - 7.4.2.2. La table ASCII**
 - 7.4.3. Les tables non normalisées**
 - 7.4.3.1. La table EBCDIC**
 - 7.4.3.2. La table IBM (ASCII étendue)**
 - 7.4.3.3. La table APPLE (ASCII étendue)**
 - 7.4.4. Les pages de codes**
 - 7.4.4.1. La page 850 (IBM)**
 - 7.4.4.2. La page 8859-1 (Iso Latin-1)**
 - 7.4.4.3. La page ANSI 1252 (Windows)**
 - 7.4.5. Conclusion**
 - 7.4.5.1. Incompatibilité PC, MAC, UNIX**
 - 7.4.5.2. Compatibilité via l'ISO Latin-1**
- 7.5. Résumé des codages**

Le Binaire

1. Le binaire, pourquoi ?

1.1. Des transistors au binaire

L'ordinateur est composé essentiellement de transistors. De ce fait les potentiels électriques qui y circulent sont de 0 volt ou de 5 volts.

Même si aujourd'hui les transistors ont des potentiels bien différents (2 volts, 3 volts, 1,8 volts, voir +12 volts ou -12 volts), nous considérerons dans un souci de simplification, que les potentiels sont toujours de 0v ou 5v.

Par convention, et afin de permettre des calculs, ces deux potentiels sont identifiés par des valeurs numériques 0 et 1.

Ces deux chiffres sont ceux de la base binaire. Il n'y a dans l'ordinateur que des suites de valeurs à 0 ou 1, donc des valeurs binaires.

Voltage	Chiffre binaire
0 Volt	0
5 Volts	1

Une valeur binaire se nomme un **bit**. C'est la compression des deux termes : *binary digit*

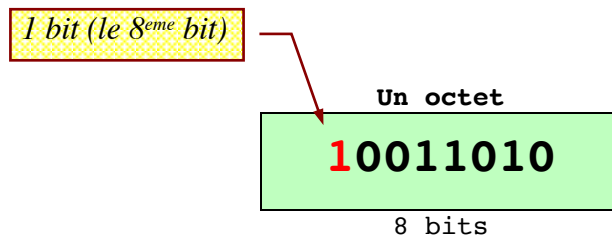
1.2. Le bit et l'octet

L'ordinateur traite les suites de chiffres binaires par paquets de 8, 16, 32, 64, 128 bits. On dira que l'ordinateur ou plus exactement le processeur est 8-bits, 16-bits, 32-bits, 64-bits ou 128-bits.

Historiquement les « premiers ordinateurs » étaient 8-bits.

Un nombre de **8 bits** se nomme un **octet**.

ATTENTION en anglais, ces termes deviennent bit (pour bit) et **byte** (pour octet). Il est alors facile de confondre un bit et un byte (octet) qui lui est composé de 8 bits !



1.3. L'unité de mesure : L'octet

Même si aujourd'hui les processeurs sont 64-bits, l'octet reste une unité de mesure importante.

En effet, la mémoire centrale d'un ordinateur était composée de « cases mémoires » pouvant contenir chacune un octet. Donner la capacité mémoire d'un ordinateur, donc son nombre de cases, revient à donner sa capacité en octets, ou plutôt en Kilo Octet (Ko) ou Méga Octets (Mo). De plus les processeurs, « cerveaux » de l'ordinateur, ont été très longtemps des 8-bits. C'est pourquoi l'octet a été utilisé aussi bien comme une unité de mesure que comme « donnée de base » pour des développements ultérieurs.

Aujourd'hui les unités de mesures « historiques » sont :

Unité	Signification	Valeur
octet	8 bits	8 bits
Ko	Kilo Octet	1024 octets
Mo	Méga Octet	1024 Ko
Go	Giga Octet	1024 Mo
To	Téra Octet	1024 Go
Po	Péta Octet	1024 To
Eo	Exa Octet	1024 Po

Nous verrons ultérieurement pourquoi le millier informatique « historique » correspond à 1024 et non à 1000.

Remarques :

Cette nomenclature est utilisée depuis le début.

Cependant, pour uniformiser les milliers informatiques avec les autres (1 Km = 1000 mètres, 1 Kg = 1000 grammes), il a été décidé que le « nouveau » Ko serait de 1000 octets.

De ce fait, le tableau précédent qui est encore largement en usage pour beaucoup d'informaticiens a été amendé, et l'ancien Ko a été nommé Kio.

Ainsi le tableau historique précédent devient officiellement

Unité	Signification	Valeur
octet	8 bits	8 bits
Kio	Kilo Octet	1024 octets
Mio	Méga Octet	1024 Ko
Gio	Giga Octet	1024 Mo
Tio	Téra Octet	1024 Go
Pio	Péta Octet	1024 To
Eio	Exa Octet	1024 Po

Et le tableau « officiel » est :

Unité	Signification	Valeur
octet	8 bits	8 bits
Ko	Kilo Octet	1000 octets
Mo	Méga Octet	1000 Ko
Go	Giga Octet	1000 Mo
To	Téra Octet	1000 Go
Po	Péta Octet	1000 To
Eo	Exa Octet	1000 Po

1.4. Le binaire pour comprendre l'ordinateur

L'ordinateur sert à faire des calculs, des traitements par milliers à notre place. Il affiche le résultat de ses traitements avec parfois des erreurs. S'il est utile de comprendre d'où vient l'erreur quand on est simple utilisateur, cela devient indispensable quand on est développeur.

La compréhension du binaire fait partie de la culture générale informatique indispensable.

Dans sa vie professionnelle, l'informaticien ne travaillera pas en permanence en binaire (heureusement pour lui !), mais **il ne pourra comprendre et résoudre certains problèmes que s'il possède une bonne connaissance de ce qui se passe « en dessous »**.

Ceci est vrai dans tous les domaines techniques, et pas seulement en informatique !

2. Les différentes bases de calcul

Dans ce chapitre, nous allons aborder différentes bases de calcul utilisées en informatique. Mais avant de passer à ces bases (8,16 et 2), il est utile de rappeler le fonctionnement de notre base 10.

2.1. La base Décimale : (base 10)

La base 10 possède 10 chiffres qui sont :

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

Pour l'instant ça va !

Si on s'amuse à compter les occurrences d'un évènement, par exemples le nombre d'étudiants qui veulent s'inscrire, alors on ne pourra compter que les 9 premières personnes qui se sont présentées. Dure la sélection !

Pour la nouvelle personne qui se présente, il n'y a plus de chiffre (après 9) !

Heureusement le système de comptage est bien fait : Pour cette nouvelle personne, on recommence à compter depuis le début, soit à 0.

MAIS alors il y a ambiguïté entre ce 0 et celui du début du comptage. Pour lever cette ambiguïté, on mémorise que l'on a déjà compté les chiffres de la base une fois, ce qui fait apparaître la retenue et le premier nombre, 10.

Ce qu'il faut retenir dans cette petite « démonstration » **est que le nombre 10 n'est pas celui qui suit le chiffre 9. Si cela est vrai dans la base décimale, ça ne l'est pas du tout pour les autres bases.** En appliquant cette méthode aux autres bases, la compréhension en sera simplifiée.

2.2. La base Octale : (base 8)

La base 8 possède 8 chiffres qui sont :

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Quand on arrive au dernier chiffre de la base (7) alors pour continuer à compter on recommence à 0 avec un 1 de retenu (10).

Dans la base 8 le nombre qui suit 7 est 10 !

De manière générale, quand on « arrive à 8 » on passe à la dizaine, centaine, millier supérieur.

Utilisation de la base Octale

Base Décimale	Base Octale
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17
16	20

← Dizaine supérieure

← Dizaine supérieure

2.3. La base Hexadécimale : (base 16)

La base hexadécimale est supérieure à la base 10. Il va donc manquer des chiffres !

Comme précédemment, **le nombre 10 suit le dernier chiffre de la base 16**. En pratique cela signifie que l'on atteindra la valeur 10 dans la base hexadécimale quand on atteint la valeur 16 dans la base décimale.

Premier nombre

Il manque donc des chiffres :

0	1	2	3	4	5	6	7	8	9	?	?	?	?	?	?	?	10
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

16 chiffres

Par convention, les chiffres « manquants » sont représentés par les lettres : **A B C D E F**

La base 16 possède 16 chiffres :

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Utilisation de la base Hexadécimale

Base Décimale	Hexadécimale
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10

← Dizaine supérieure

2.4. La base Binaire : (base 2)

La base 2 possède 2 chiffres qui sont : **0 1**

Le nombre 10 suit le dernier chiffre de la base, soit 1. Cette base va « tourner » très vite !

Utilisation de la base Binaire

Base Décimale	Binaire	
0	0	
1	1	
2	10	← Dizaine
3	11	
4	100	← Centaine
5	101	
6	110	
7	111	
8	1000	← Millier
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	
16	10000	← Dix Milliers

2.5. Tableau récapitulatif des bases

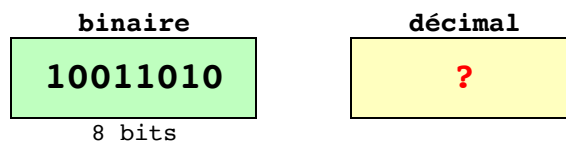
Tableau récapitulatif

Base Décimale	Octale	Hexadécimale	Binaire
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111
16	20	10	10000

3. La conversion du binaire vers le d  cimal

Pour bien comprendre « le binaire » et trouver des solutions    certains probl  mes informatiques «   nigmatiques », il faut savoir convertir un nombre binaire en un nombre d  cimal et r  ciproquement. Dans cette partie, nous traitons la conversion d'un nombre binaire en un nombre d  cimal.

Essayons de convertir l'octet suivant en d  cimal :



3.1. La formule math  matique

Une formule math  matique g  n  rale permet de convertir n'importe quel nombre exprim   dans une base **N** en un nombre d  cimal.

$$\begin{array}{c}
 (\text{ A } \text{ B } \text{ C } \text{ D })_N \\
 \swarrow \quad \searrow \quad \swarrow \quad \searrow \\
 = \\
 \swarrow \quad \searrow \quad \swarrow \quad \searrow \\
 (A \times N^3) + (B \times N^2) + (C \times N^1) + (D \times N^0)
 \end{array}$$

Par exemple le nombre octal **2607** sera converti de la mani  re suivante :

$$\begin{array}{c}
 (\text{ 2 } \text{ 6 } \text{ 0 } \text{ 7 })_8 \\
 \swarrow \quad \searrow \quad \swarrow \quad \searrow \\
 (2 \times 8^3) + (6 \times 8^2) + (0 \times 8^1) + (7 \times 8^0) \\
 (2 \times 512) + (6 \times 64) + (0 \times 8) + (7 \times 1) \\
 (1024) + (384) + (0) + (7) \\
 = \\
 (\text{ 1 } \text{ 4 } \text{ 1 } \text{ 5 })_{10}
 \end{array}$$

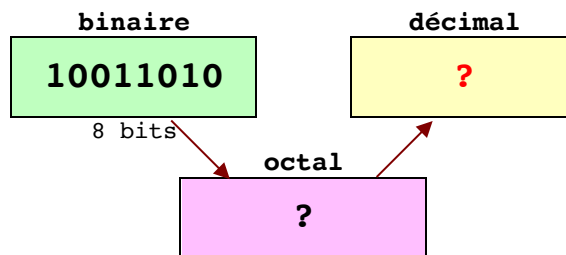
3.2. Utilisation de la base octale

3.2.1. Méthode

Pour éviter d'appliquer cette formule directement au nombre binaire, on peut utiliser une base intermédiaire : la base octale.

La conversion se fera en deux étapes :

- 1- Binaire vers octal
- 2- Octal vers décimal



L'intérêt de cette méthode est que pour la première étape aucun calcul ne sera nécessaire. Cette conversion sera « visuelle ». Pour la deuxième étape, la formule vue auparavant sera utilisée.

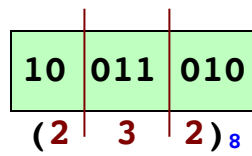
3.2.2. Première étape

Base Décimale	Octale	Binaire
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000

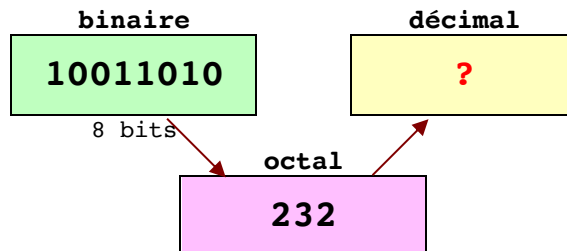
En regardant le tableau comparatif des bases, nous pouvons constater que le plus grand chiffre octal (**7**) correspond au plus grand nombre binaire sur 3 bits (**111**). En conséquence, **tout chiffre octal** (forcément plus petit que 7) **s'écrit au plus sur 3 bits**.

En conclusion : Dans un octet, **chaque groupe de 3 bits** correspond à un **chiffre octal**.

L'octet pr  c  dent ainsi d  coup   donne :



La premi  re   tape donne :

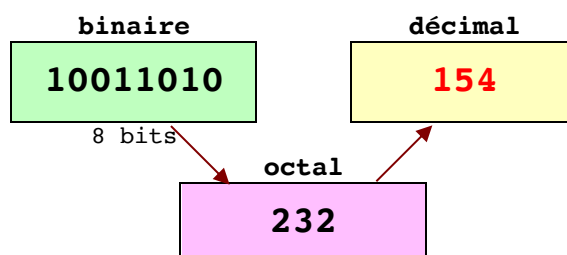


3.2.3. Deuxi  me   tape

Le nombre octal **232** sera converti de la mani  re suivante :

$$\begin{aligned}
 & (2 \ 3 \ 2)_8 \\
 & \swarrow \quad \downarrow \quad \searrow \\
 & (2 \times 8^2) + (3 \times 8^1) + (2 \times 8^0) \\
 & (2 \times 64) + (3 \times 8) + (2 \times 1) \\
 & (128) + (24) + (2) \\
 & \quad \quad \quad = \\
 & (1 \ 5 \ 4)_{10}
 \end{aligned}$$

La deuxi  me   tape donne :



3.3. Utilisation de la base hexad  cimale

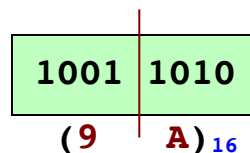
3.3.1. Premi  re   tape

Base D��cimale	Hexad��cimale	Binaire
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

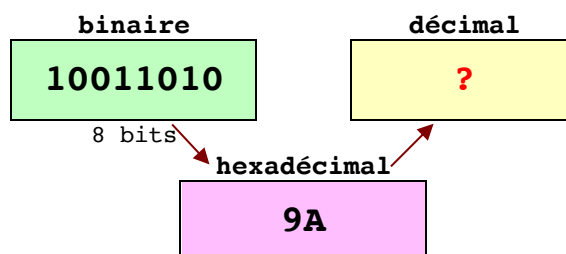
De la m  me mani  re que pr  c  demment, nous pouvons constater que le plus grand chiffre octal (**F**) correspond au plus grand nombre binaire sur 4 bits (**1111**). En cons  quence, **tout chiffre hexad  cimal** (forc  ment plus petit que F) s'  crit **au plus sur 4 bits**.

En conclusion : Dans un octet, **chaque groupe de 4 bits** correspond    un **chiffre hexad  cimal**.

L'octet pr  c  dent ainsi d  coup   donne :



La premi  re   tape donne :

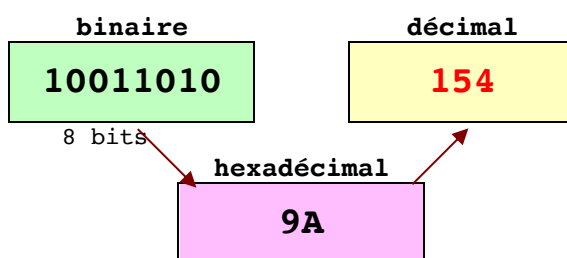


3.3.2. Deuxi  me   tape

Le nombre hexad  cimal **9A** sera converti de la mani  re suivante :

$$\begin{aligned}
 & (\text{9 A})_{16} \\
 & \swarrow \quad \searrow \\
 & (9 \times 16^1) + (10 \times 16^0) \\
 & (9 \times 16) + (10 \times 1) \\
 & (144) + (10) \\
 & = \\
 & (\text{1 5 4})_{10}
 \end{aligned}$$

La deuxi  me   tape donne :



3.4. Conversion directe

La conversion directe consiste    appliquer la formule de conversion sur l'octet :

$$\begin{aligned}
 & (\text{1 0 0 1 1 0 1 0})_2 \\
 & \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \\
 & (1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)
 \end{aligned}$$

Avant de faire cette op  ration, simplifions le calcul :

0x2^x donnera toujours **0** . Et le **0** est neutre dans l'addition.

1x2^x donnera toujours **2^x** .

Ce qui donne :

$$\begin{aligned}
 & (1 \times 2^7) + (\cancel{0 \times 2^6}) + (\cancel{0 \times 2^5}) + (1 \times 2^4) + (1 \times 2^3) + (\cancel{0 \times 2^2}) + (1 \times 2^1) + (\cancel{0 \times 2^0}) \\
 & \quad \quad \quad 2^7 \qquad \qquad \qquad + \quad 2^4 \quad + \quad 2^3 \qquad \qquad \qquad + \quad 2^1
 \end{aligned}$$

En conclusion :

Dans l'octet, seuls les **1** sont pertinents. Ils correspondent à une puissance de **2**. Exactement à **2^y**, avec y leur position (de 0 à 7).

$$\begin{array}{cccccccc}
 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
 (& 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 &)_2 \\
 \downarrow & & & \downarrow & \downarrow & \downarrow & & \downarrow & & \\
 2^7 & + & & 2^4 & + & 2^3 & + & 2^1 & &
 \end{array}$$

Il suffit de connaître les puissances de 2 pour savoir interpréter directement un octet.

Rappel des premières puissances de 2.

$2^0 = 1$	$2^6 = 64$
$2^1 = 2$	$2^7 = 128$
$2^2 = 4$	$2^8 = 256$
$2^3 = 8$	$2^9 = 512$
$2^4 = 16$	$2^{10} = 1024$ <u>le kilo informatique</u>
$2^5 = 32$	

3.5. Décomposition en puissance de 2

En réalité, la règle sous-jacente à la formule est que tout nombre peut être décomposé en une somme de puissances de 2.

$$1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 = 2^7 + 2^4 + 2^3 + 2^1$$

se décompose en

$$\begin{array}{rcl}
 + & 1 & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 = 2^7 \text{ (7 zéros) } = 128 \\
 + & & 1 \ 0 \ 0 \ 0 \ 0 \ 0 = 2^4 \text{ (4 zéros) } = 16 \\
 + & & 1 \ 0 \ 0 \ 0 = 2^3 \text{ (3 zéros) } = 8 \\
 + & & 1 \ 0 = 2^1 \text{ (1 zéro) } = 2
 \end{array}$$

$$1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 = 2^7 + 2^4 + 2^3 + 2^1 = 154$$

On peut en déduire que :

Tout nombre binaire commençant par un 1 suivi par des 0 est une puissance de 2.

C'est exactement : **2^{nombre_de_zéros}**

4. La conversion du d cimal vers le binaire

4.1. Le principe


Nous abordons maintenant la conversion inverse, **de la base d cimale vers la base binaire**. Nous allons utiliser les m mes proc d s, c'est- -dire l'utilisation de bases interm diaires, puis la conversion directe.

Le principe de base est de consid rer que dans le nombre d cimal de d part, **chaque paquet de N  l ments (N = 8 pour octal) constitue une « dizaine » dans la base N**. En effet pour la base octale, le nombre 10 correspond au chiffre 8 en d cimal.

Prenons l'exemple de **20** allumettes dispos es sur une table, et exprimons ce nombre dans la base octale :


 nombre d'allumettes : (20)₁₀

Si on les regroupe par paquet de 8, cela donne deux paquets (**2x8=16**) et il en reste 4 :


 nombre d'allumettes : ((2x8)+4)₁₀

Chaque paquet de 8 allumettes est en fait une dizaine en octal (8₁₀ = 10₈) .

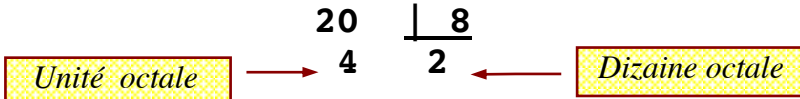
Nous avons donc :

En d cimal : 8₁₀ + 8₁₀ + 4₁₀ = 20₁₀ ((2x8)+4)₁₀

En octal : 10₈ + 10₈ + 4₈ = 24₈ ((2x10)+4)₁₀

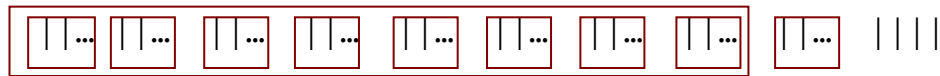
Dans cet exemple 20₁₀ = 24₈ . Il y a donc 24 allumettes exprim es en octal.

Compter le nombre de fois que l'on trouve un paquet de N (8) allumettes dans un nombre d cimal revient   faire une **division enti re du nombre par N**. Le quotient de la division est le nombre de paquets (les dizaines dans la base N), le reste les unit s :



De la m  me mani  re, s'il y a 8 paquets (ou plus) alors on peut regrouper les paquets eux-m  mes en groupe de 8, ce qui donnera alors des « dizaines de dizaines », donc des centaines octales. C'est l'exemple suivant :

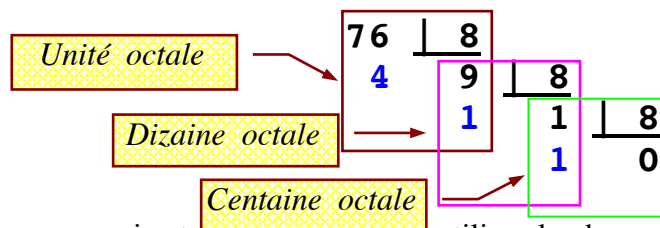
D  c: $8_{10} + 8_{10} + 8_{10} + 8_{10} + 8_{10} + 8_{10} + 8_{10} + 8_{10} + 8_{10} + 4_{10} = 76_{10} \text{ (} 9 \times 8 \text{)} + 4$



Oct: $10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 4_8$

Octal : $100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 100_8 + 4_8 = 114_8$

Ceci revient    faire des divisions enti  res "en cascades". Quand le quotient est   gal    **0** on ne peut plus diviser. Le reste de chaque division donne l'unit  , la dizaine, la centaine, ...



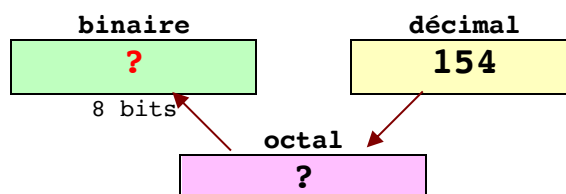
Dans un premier temps, nous allons utiliser les bases octale et hexad  cimale comme bases interm  diaires.

4.2. Utilisation de la base octale

4.2.1. M  thode

La conversion se fera en deux   tapes :

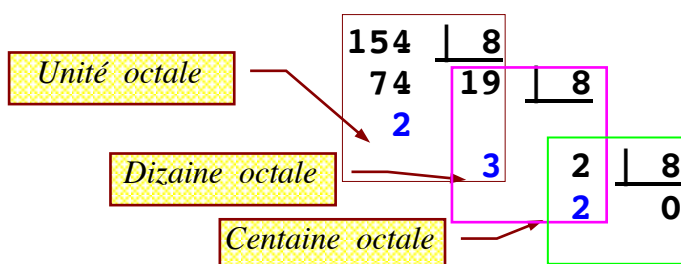
- 1- D  cimal vers octal
- 2- Octal vers binaire



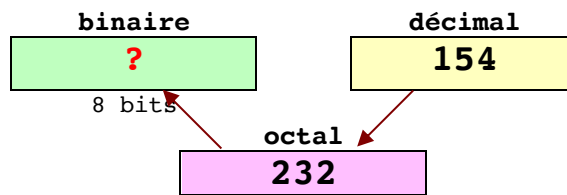
L'int  r  t de cette m  thode est que pour la deuxi  me   tape aucun calcul ne sera n  cessaire. Cette conversion sera « visuelle ». Pour la premi  re   tape, la m  thode vue auparavant sera utilis  e.

4.2.2. Premi  re   tape

Ceci revient    faire des divisions enti  res "en cascades". Quand le quotient est   gal    **0** on ne peut plus diviser. Le reste de chaque division donne l'unit  , la dizaine, la centaine, ...



$$154_{10} = 232_8$$

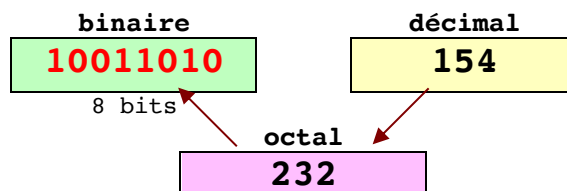


4.2.3. Deuxi  me   tape

La conversion (octal -> binaire) consiste    faire l'op  ration inverse sur un ensemble de 3 bits.

(2	3	2)	8
10	011	010	

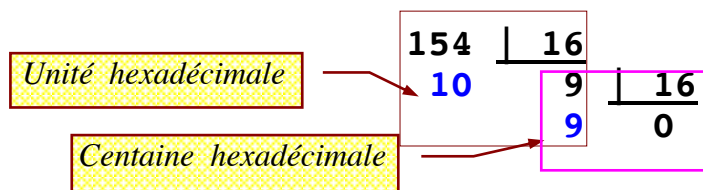
Le r  sultat de la deuxi  me   tape est :



4.3. Utilisation de la base hexad  cimale

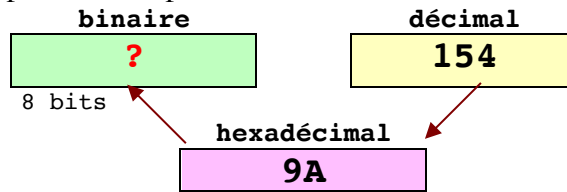
4.3.1. Premi  re   tape

Le raisonnement est identique, mais avec des divisons enti  res par 16 .



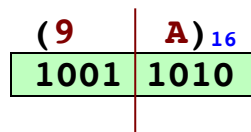
$$154_{10} = 9A_{16}$$

Le résultat de la première étape est :

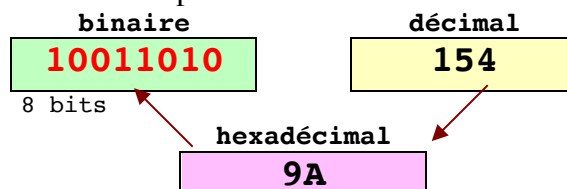


4.3.2. Deuxième étape

La conversion (hexadécimal -> binaire) consiste à faire l'opération inverse sur 4 bits.



Le résultat de la deuxième étape est :



4.4. Conversion directe

La conversion directe d'un nombre décimal en un nombre binaire consiste à trouver sa décomposition en puissance de 2.

Pour obtenir cette décomposition, on cherche la plus grande puissance de 2 contenu dans le nombre, et on la soustrait. On recommence avec le reste et ainsi de suite.

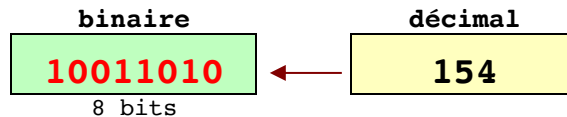
Avec le nombre 154, cela donne :

$$\begin{array}{r}
 154 \\
 - 128 \\
 \hline
 26
 \end{array}
 \quad
 \begin{array}{r}
 26 \\
 - 16 \\
 \hline
 10
 \end{array}
 \quad
 \begin{array}{r}
 10 \\
 - 8 \\
 \hline
 2
 \end{array}
 \quad
 \begin{array}{r}
 2 \\
 - 2 \\
 \hline
 0
 \end{array}$$

Par conséquent :

$$\begin{aligned}
 154 &= 128 + 16 + 8 + 2 \\
 &= 2^7 + 2^4 + 2^3 + 2^1 \\
 &= \begin{array}{l} + 2^7 \text{ (7 zéros)} \\ + 2^4 \text{ (4 zéros)} \\ + 2^3 \text{ (3 zéros)} \\ + 2^1 \text{ (1 zéro)} \end{array} = \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} = 128 \\
 &= \begin{array}{cccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} = 16 \\
 &= \begin{array}{cccccccc} & & 1 & 0 & 0 & 0 & 0 & 0 \end{array} = 8 \\
 &= \begin{array}{cccccccc} & & & 1 & 0 & & & 0 \end{array} = 2 \\
 \hline
 2^7 + 2^4 + 2^3 + 2^1 &= \begin{array}{cccccccc} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array} = 154
 \end{aligned}$$

La conversion donne :



5. Capacit  de m morisation de 1, 2, ... N octets

Nous avons montr  les diff rentes m thodes pour convertir les nombres binaires en nombres d cimaux ou inversement. Nous allons  tudier dans ce chapitre la capacit  de m morisation des octets.

5.1. Capacit  de m morisation

La valeur num rique que l'on peut  crire sur un octet d pend du nombre maximal binaire sur 8 bits .

	binaire	d�cimal
Plus petit nombre	00000000 8 bits	0
Plus grand nombre	11111111 8 bits	?

Pour conna tre la valeur d cimale de **11111111** il suffit de faire la somme de toutes les puissances de 2 :

$$\begin{aligned}
 \mathbf{11111111} &= \mathbf{2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0} \\
 &= \mathbf{128 + 64 + 32 + 16 + 8 + 4 + 2 + 1} \\
 &= \mathbf{255}
 \end{aligned}$$

Il y a cependant une m thode plus rapide :

On peut constater que **11111111** est le nombre binaire qui pr c de **100000000** .

$$\begin{array}{r}
 \\
 \\
 + \\
 \hline
 1
 \end{array}$$

or $\mathbf{100000000} = \mathbf{2^8} = \mathbf{256}$
donc $\mathbf{11111111} = \mathbf{2^8 - 1} = \mathbf{255}$

	binaire	d�cimal
Plus grand nombre	11111111 8 bits	255

Plus g n ralement la plus grande valeur d cimale cod e sur **N** bits est : $\mathbf{2^N - 1}$

	binaire	décimal
Plus grand nombre	1111..1111 N bits	$2^N - 1$

Remarque : Pour les puissances **N** supérieures à 10, la formule mathématique suivante facilite le calcul :

$$2^N = 2^{10} \times 2^{N-10}$$

Exemple :

$$\begin{aligned}
 2^{15} &= 2^{10} \times 2^5 \\
 &= 1024 \times 32 \\
 &= 32\,768
 \end{aligned}$$

5.2. Les limitations

Nous pouvons en déduire les valeurs **décimales maximales** qu'il est possible de mémoriser sur 1,2,3,4 octets. **La valeur minimale sera toujours 0** .

1 octet	binaire 11111111 8 bits	décimal $(2^8 - 1)$ 255
2 octets	binaire 111111..1111 16 bits	décimal $(2^{16} - 1)$ 65 535
3 octets	binaire 111111..111111 24 bits	décimal $(2^{24} - 1)$ 16 777 215
4 octets	binaire 1111111..111111 32 bits	décimal $(2^{32} - 1)$ 4 294 967 295

6. Domaines d'application du binaire

Le binaire est partout en informatique. Aussi bien dans les ordinateurs, les réseaux, les fichiers textes, **partout !** Mais on le voit rarement.

Pour vous convaincre de son omniprésence, voici quelques exemples où il est relativement visible.

6.1. Couleur en HTML

Un informaticien qui écrit ces propres « pages web », doit savoir coder la couleur du fond. Ce codage se fait en indiquant la quantité de **Rouge Vert Bleu**, les trois couleurs primaires (**RGB** en anglais), qu'il faut mélanger.

Dans la page "web" chaque couleur est codée sur **1 octet**. Chaque couleur pourra varier de la valeur **0** à **255** !

La syntaxe dans le langage HTML est :

BGCOLOR=#RRVVB

Ou **RR** est l'intensité (0 à 255) du **Rouge** exprimé en **hexadécimal**

VV est l'intensité (0 à 255) du **Vert** exprimé en **hexadécimal**

BB est l'intensité (0 à 255) du **Bleu** exprimé en **hexadécimal**

Ainsi, pour un fond de page « web » en vert, il faut coder les 3 octets de la manière suivante :

	rouge	vert	bleu
3 octets	00000000	11111111	00000000
Hexadécimal	00	FF	00

La syntaxe dans le langage HTML sera donc :

BGCOLOR=#00FF00

Pour une autre couleur plus complexe, il suffit de la choisir de manière interactive sous Photoshop (par exemple), et de convertir les valeurs décimales des trois couleurs en hexadécimal.

Par exemple un *marron clair* aura les couleurs suivantes (trouvées sous Photoshop) :

Décimal

Rouge : 252

Vert : 236

Bleu : 187

La conversion en hexadécimal donnera :

	Décimal	Binaire	Hexadécimal
Rouge :	252	1111 1100	FC
Vert :	236	1110 1100	EC
Bleu :	187	1011 1011	BB

La syntaxe dans le langage HTML sera donc :

BGCOLOR=#FCECBB

Remarque :

Un fond	blanc	sera codé	BGCOLOR=#FFFFFF
Un fond	noir	sera codé	BGCOLOR=#000000

6.2. Réseau : Adressage MAC et IP

Les réseaux n'échappent pas au binaire. C'est le cas des adresses des machines.

La première adresse est l'adresse « physique » de la carte réseau, l'**adresse MAC**. Elle a la forme suivante :

00.0A.27.32.AF.F3

C'est donc une suite de **6 octets** exprimés en hexadécimal, qui permettent d'identifier le numéro de la carte réseau de la machine.

La seconde adresse « plus visible » est l'adresse « internet » ou **adresse IP**. Elle a la forme suivante :

134.157.15.2

C'est donc une suite de **4 octets** exprimés en décimal, qui permettent d'identifier l'adresse internet de la machine.

On donne généralement un nom plus clair à cette adresse numérique sous la forme :

ibm2.cicrp.jussieu.fr

1 3 4 . 1 5 7 . 1 5 . 2

- Le domaine principal est **fr**. Il peut donc y avoir **256** grands domaines dans le monde (**fr, us, org, com, edu, gov, gouv, net...**) en théorie.
- Le sous-domaine de **fr** est **jussieu**. En France, il peut y avoir **256** sous-domaines (**jussieu, inra, inria, ...**) en théorie.

- A **jussieu**, il peut y avoir **256** sous-domaines, dont le **cicrp** en théorie.
- Et au **cicrp** il peut y avoir **256** machines, en théorie.

Le nombre total théorique de machines dans le monde est donc de :

$$256 \times 256 \times 256 \times 256 = 256^4 = 2^{32} = \boxed{4\ 294\ 967\ 296}$$

Ce nombre est théorique car certaines adresses sont réservées pour la gestion du réseau, ou pour des réseaux privés. Le nombre réel de machines est donc inférieur.

Cette limitation du nombre de machines dans le monde ou dans un sous-réseau (**256** au maximum) a engendré l'émergence de nouveaux protocoles tels que **DHCP** ou **IPV6**

Le protocole **DHCP** permet par exemple de gérer 400 machines avec seulement 256 adresses, par une gestion dynamique des adresses. C'est donc la pénurie d'adresses IP qui a provoqué cette gestion particulière.

6.3. Couleur d'un écran

Chaque point de l'écran peut prendre une couleur différente. Le nombre de couleurs possibles pour un point (pixel) dépend de la valeur binaire que l'on peut écrire sur un, deux, N octets.

La couleur de chaque point est numérotée. Si chaque pixel est codé sur un octet alors la couleur d'un pixel varie de **0** à **255**. Ce qui correspond à **256** couleurs.

Si l'écran possède une résolution de 1000 pixels x 1000 pixels, soit 1 million de pixels, alors il faut 1Mo de RAM graphique pour gérer l'affichage.

Si on veut **65536** couleurs par pixel, il faut alors 2 octets par pixel (**65536** correspond au nombre de codages sur 16 bits). 2 Mo de RAM graphique sont nécessaires. Et ainsi de suite.

Les grandes valeurs pour le nombre de couleurs sont :

Nombre de couleurs	Nombre de bits
16	4 bits
256	8 bits
32768	15 bits
65536	16 bits
16,7 Millions	24 bits
4 Milliards	32 bits

7. Représentation binaire des données : le codage

Nous avons vu comment coder un nombre décimal en binaire et comment décoder un nombre binaire en décimal. Cependant il ne faut pas oublier que l'ordinateur est « à notre service », et que son rôle est de traiter nos données, et de restituer le résultat.

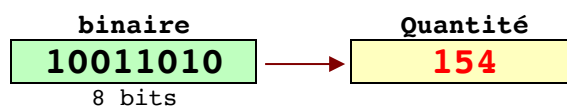
Si la nature des données informatique est unique, le binaire, nos données sont multiples.

En effet aujourd'hui on utilise l'ordinateur aussi bien pour taper du texte, faire des calculs avec des chiffres entiers ou avec des virgules, ou traiter des images.

Le problème sous-jacent est de savoir **comment coder nos différentes informations, en un seul type d'information informatique, le binaire**. Et surtout **comment décoder correctement cette information si on ne sait comment elle a été produite** (texte, entier,...).

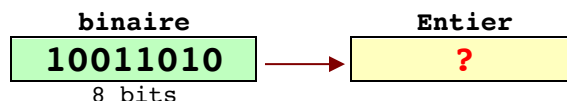
7.1. Le codage "quantité" (entier non signé)

Il s'agit du codage que nous avons vu précédemment.



7.2. Le codage entier

Si on considère que cet octet contient un entier avec un signe + ou -, alors l'interprétation sera différente.



7.2.1. Le principe

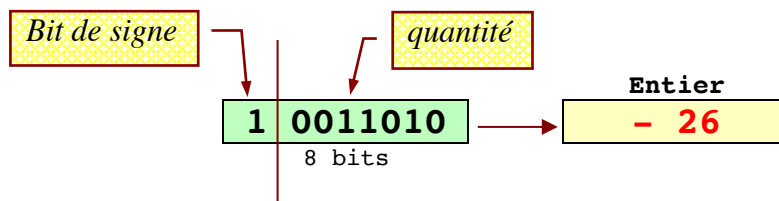
La différence entre un entier et une "quantité" (entier non signé) est le signe. Ce signe peut être un "+" ou un "-". Or il suffit d'une seule valeur binaire pour coder une information pouvant prendre deux états.

On va donc considérer que l'un des bits est le « **bit de signe** ». Ce sera le 8^{ème},

La convention est la suivante :

0 -> **+**
1 -> **-**

Ainsi cet octet serait interpr  t   de la mani  re suivante :



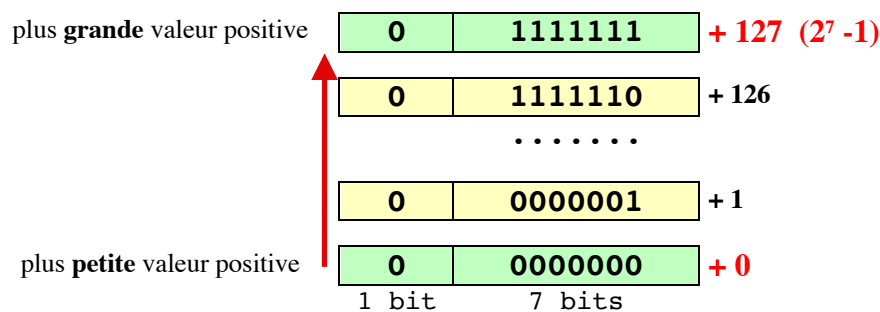
Remarque :

Interpr  ter le huiti  me bit comme un bit de signe revient dans cet exemple    retirer la valeur 128    la quantit   154.

Nous pouvons donc   crire des valeurs comprises entre +127 et -127 d'apr  s ce codage. Cette interpr  tation est juste pour les entiers positifs, mais fausse pour les entiers n  gatifs.

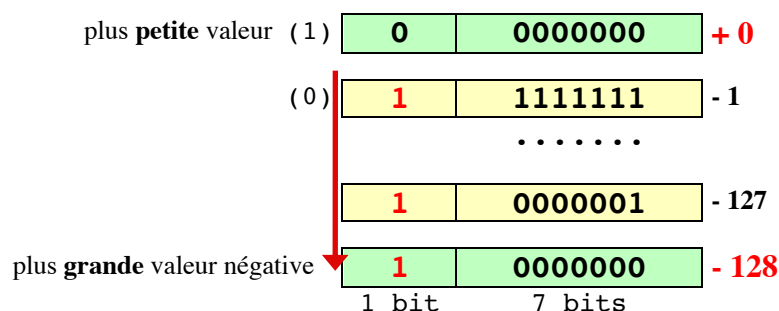
7.2.2. Entier positif

En partant de la valeur centrale 0, on obtient les valeurs positives successives en ajoutant la valeur binaire 1    chaque fois. La plus grande valeur enti  re positive sur 8 bits est donc **+127**.



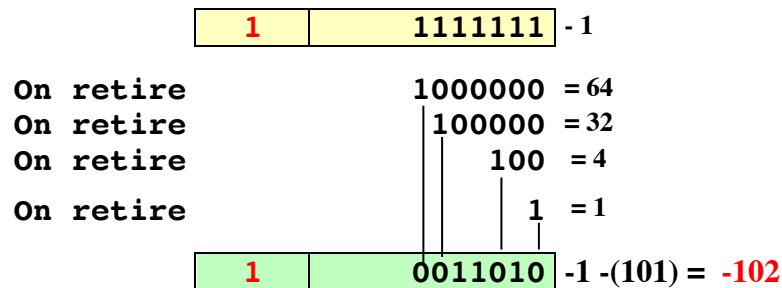
7.2.3. Entier n  gatif

En partant de la valeur centrale 0, on obtient les valeurs n  gatives successives en retirant la valeur binaire 1    chaque fois. Pour pouvoir retirer 1    0 on suppose que 0 (sur 8 bits) est en r  alit   un nombre binaire sur 9 bits, donc le 9  me bit    1    disparu (**1 00000000**). La plus grande valeur enti  re n  gative sur 8 bits est donc **-128**.



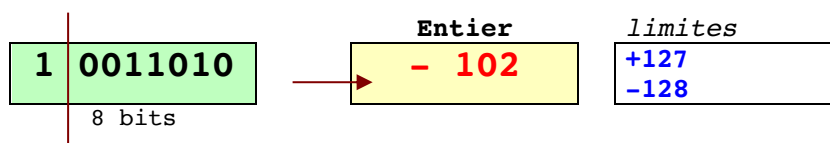
Pour obtenir la vraie valeur négative de l'octet **10011010** on peut appliquer la méthode suivante :

On part de la valeur de -1 et l'on note les valeurs binaires manquantes. Chaque valeur binaire manquante correspond à la soustraction d'un nombre décimal. En ayant la somme de ces nombres on peut en déduire ce qu'il faut retirer à -1 pour obtenir la véritable valeur négative.



7.2.4. Les limites

Sur un octet, la valeur entière varie de + 127 à -128



Nombre de bits	Portée
8 bits signés	+127 .. -128
16 bits signés	+32 767 .. -32 768
32 bits signés	+2 147 483 647 .. -2 147 483 648

Un entier sera généralement codé sur **32** bits

7.3. Le codage réel

7.3.1. La notation en virgule flottante

Un nombre réel peut être écrit suivant 2 notations.

- La notation en **virgule fixe**.
- La notation en **virgule flottante**.

En prenant l'exemple du nombre **-2.3** cela donne :

Virgule Fixe : **-2.3**

Virgule Flottante : **-2.3 x 10⁰**
 ou **-23 x 10⁻¹**
 ou **-0.023 x 10⁺²**
 ou **-0.23 x 10⁺¹**

7.3.2. Le principe

Pour comprendre la méthode de codage, considérons que nous travaillons avec les puissances de 10. La démonstration en sera simplifiée.

En réalité le vrai codage utilise des puissances de 2 (binaire oblige). Le codage présenté sera faux, mais le principe sera juste.

La notation en virgule flottante est celle qui a été retenue en informatique.

Le terme anglais est : *floating point*

Puisque la formulation du nombre peut varier en faisant « flotter » la virgule grâce à la puissance, la convention retenue sera d'exprimer le nombre sous la forme :

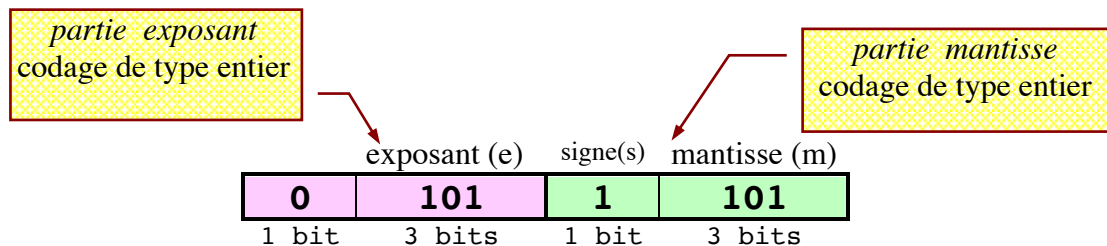
$$\pm 0.m \times 10^e$$

par exemple : **- 0.23 x 10⁺¹**

Il devient inutile de coder "0." et "10" car ces parties ne changent pas !

Seuls la mantisse (**m**) et l'exposant (**e**) seront codés sous forme d'entiers.

L'octet est partagé en deux parties. La première contient l'exposant, la deuxième la mantisse.



Avec ce codage, la plus grande valeur sera :

+	e = 7	+	m = 7
<div style="border: 1px solid black; background-color: #FFDAB9; display: inline-block; padding: 2px 10px;">0</div>	<div style="border: 1px solid black; background-color: #FFDAB9; display: inline-block; padding: 2px 10px;">111</div>	<div style="border: 1px solid black; background-color: #D9F7D9; display: inline-block; padding: 2px 10px;">0</div>	<div style="border: 1px solid black; background-color: #D9F7D9; display: inline-block; padding: 2px 10px;">111</div>
1 bit	3 bits	1 bit	3 bits

$$+ 0.7 \times 10^7$$

$$+ 7\,000\,000$$

Remarque :

En fait, c'est un nombre entier qui est codé sous format réel. Grâce à cette astuce, la capacité de mémorisation d'un entier passe de +127 à +7000000 !!

Si la capacité de mémorisation est augmentée, on ne peut cependant pas coder tous les entiers entre 0 et 7000000. En effet, la mantisse ne pourra prendre que les valeurs suivantes : 0,1,2,3,4,5,6,7. **Les données ne seront pas d'une grande précision !**

Pour résoudre ce problème il suffit de découper l'octet en deux parties inégales, avec plus de bits pour la mantisse.

+	e = 3	+	m = 15
<div style="border: 1px solid black; background-color: #FFDAB9; display: inline-block; padding: 2px 10px;">0</div>	<div style="border: 1px solid black; background-color: #FFDAB9; display: inline-block; padding: 2px 10px;">11</div>	<div style="border: 1px solid black; background-color: #D9F7D9; display: inline-block; padding: 2px 10px;">0</div>	<div style="border: 1px solid black; background-color: #D9F7D9; display: inline-block; padding: 2px 10px;">1111</div>
1 bit	2 bits	1 bit	4 bits

Ceci augmente la précision puisque la mantisse peut varier de 0 à 15 maintenant, mais diminue la puissance. C'est ce type de découpage asymétrique qui est utilisé.

Malgré cela notre valeur $-0.23 \times 10^{+1}$ ne peut toujours pas être codée car la précision s'arrête à 15 et non à 23. Il faudra donc utiliser plusieurs octets pour le codage d'un réel.

Un réel sera généralement codé sur 32 ou 64 bits

De plus **la précision dans les calculs scientifiques est très importante**. En voici une petite démonstration :

Faisons le calcul suivant en mathématique :

$$2 \rightarrow \sqrt{2} \rightarrow (\sqrt{2})^2 \rightarrow 2 \quad \text{Le résultat sera toujours 2 !}$$

En informatique, cela donnera :

$$2 \rightarrow \sqrt{2} \rightarrow (1,4)^2 \rightarrow 1,96 \quad \text{Le résultat est faux !}$$

La valeur de $\sqrt{2}$ n'est pas assez précise.

Si on prend une valeur plus précise, l'erreur diminue :

$$2 \rightarrow \sqrt{2} \rightarrow (1,4142)^2 \rightarrow 1,99996 \quad \text{Le résultat est presque juste !}$$

Quand un résultat erroné est utilisé dans un autre calcul, **l'erreur se propage et grandit**.

Il est important d'avoir la meilleure précision pour les calculs scientifiques.

7.3.3. Le vrai codage

Si le principe précédent est juste, le calcul basé sur des puissances de 10 est erroné.

Rappel de la règle générale : Le nombre réel est réécrit sous une forme $\pm 0.m \times 10^{\pm e}$ où **m** est la *mantisse* et **e** l'*exposant*. Le nombre -22.625 serait exprimé comme $-0.22625 \times 10^{+2}$ en base 10.

Cette règle générale est appliquée en base 2, et non en base 10.

Le codage binaire est de la forme $\pm 1.m \times 2^{\pm e}$. Ainsi -22.625 s'écrit -1.0110101×2^4 .

Pour obtenir ce résultat, le nombre décimal -22.625 est d'abord traduit sous la forme binaire -10110.101 , de la manière suivante :

- 22 en base 10 est une somme de puissance de 2 soit : $16 + 4 + 2 = 2^4 + 2^2 + 2^1 = 10110$ en binaire.
- 0.625 en base 10 est une somme de puissance négative de 2 soit : $0.5 + 0.125 = 2^{-1} + 2^{-3} = 0.101$ en binaire.

Le nombre -10110.101 est ensuite traduit en -1.0110101×2^4 et finalement écrit sur 32 ou 64 bits.

Représentation avec mantisse et exposant

La valeur -1.0110101×2^4 est présentée en une suite d'octets selon la norme IEEE 754.

Les deux formats de cette norme sont :

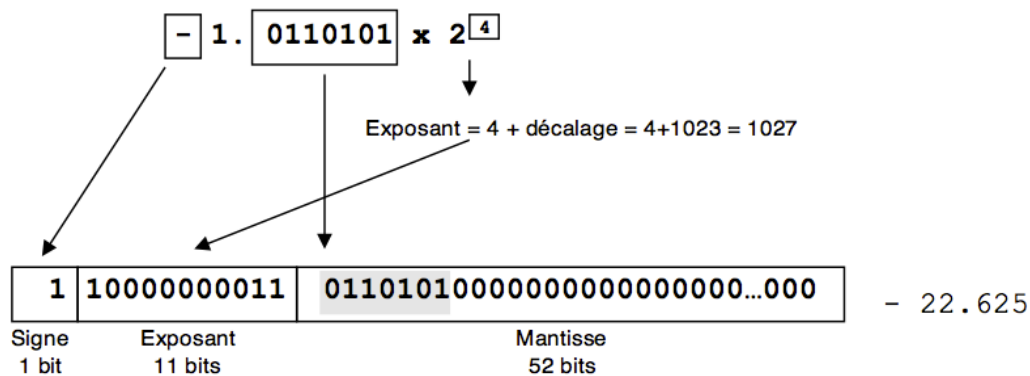
- Sur **32 bits** : **1 bit de signe** de la mantisse, **8 bits d'exposant** (valeurs -126 à 127 avec un décalage), **23 bits de mantisse**. C'est le type *float* sur 32 bits ;
- Sur **64 bits** : **1 bit de signe** de la mantisse, **11 bits d'exposant** (valeurs -1022 à 1023 avec un décalage), **52 bits de mantisse**. C'est le type *float* sur 64 bits ou *double*.

La formule utilisée par la norme est : **valeur = (signe) \times 1.m \times 2^{e - décalage}**

- $\text{décalage} = 2^{n-1} - 1$ (n = nombre de bits pour l'exposant, soit 8 sur 32 bits ou 11 sur 64 bits), soit 127 ($2^7 - 1$) sur 32 bits, ou 1023 ($2^{10} - 1$) sur 64 bits.
- $p = e - \text{décalage}$. Le calcul de l'exposant binaire est le suivant : **e = p + 127** sur 32 bits, ou **e = p + 1023** sur 64 bits.

La figure suivante présente le codage sur **64 bits** de -22.625 soit $-1.\boxed{0110101} \times 2^4$ en binaire.

La formule, (signe) \times 1.m \times 2^{e - décalage}, se traduit par : **m=0110101** et **e = 4+1023 = 1027=10000000011**



Conséquence de cette représentation sur l'exactitude des calculs

Cette représentation binaire d'un réel ne code pas parfaitement tous les nombres. Certains réels *exacts* en base 10 deviennent *infinis* en binaire. C'est le cas du nombre 2.2, dont la mantisse binaire est infinie :

0.2×2	=	0.4	→	0	→	0.0
0.4×2	=	0.8	→	0	→	0.00
0.8×2	=	1.6	→	1	→	0.001
0.6×2	=	1.2	→	1	→	0.0011
0.2×2	=	0.4	→	1	→	0.00110
0.4×2	=	0.8	→	0	→	0.001100

...

Le calcul continue indéfiniment ! La représentation de **2.2** sur vingt-trois décimales est : 10.00110011001100110011001, ce qui correspond à **2.199999928**.

Le calcul 2×2.2 dans un programme donnera le résultat approché 4.399999856 à la place du résultat exact 4.4. Cette erreur grandira au fur et à mesure des calculs.

7.3.4. Les limites du codage

Valeurs minimales

Sur 32 bits le plus petit nombre positif différent de 0, et le plus grand nombre négatif différent de 0 correspondent à : $\pm 1.17549435 \times 10^{-38}$

Sur 64 bits le plus petit nombre positif différent de 0, et le plus grand nombre négatif différent de 0 correspondent à : $\pm 2.2250738585072014 \times 10^{-308}$

Valeurs maximales

Sur 32 bits le plus grand nombre positif fini, et le plus petit nombre négatif fini sont : $\pm 3.40282326 \times 10^{38}$

Sur 64 bits le plus grand nombre positif fini, et le plus petit nombre négatif fini sont : $\pm 1.7976931348623157 \times 10^{308}$

Format	Décimales	Portée
32 bits, signés	24 bits	1.17 E-38 .. 3.4 E+38
64 bits, signés	53 bits	2.22 E-308..1.79 E+308
128 bits, signés	106 bits	2.22 E-308..1.79 E+308

7.3.5. Les erreurs de pr  cision

Pour les nombres d  cimaux dont la valeur apr  s la virgule est infinie comme $\pi=3.1415926535898$, et les nombres dont le codage binaire apr  s la virgule est une suite infinie de 0 et de 1, comme 2.2, le fait d'  crire leur valeur dans une variable dont la taille m  moire est par d  finition finie, implique que la valeur est fautive : la partie d  cimale qui ne « rentre » pas dans la m  moire est perdue.

Les valeurs r  elles infinies sont toujours fautes !

Prenons l'exemple des valeurs 0.1 et 0.7 qui n'ont pas de repr  sentation exacte en binaire !

La somme de 0.1 et de 0.7 doit donner 0.8, or le r  sultat informatique est *0.79999999999999991118*, soit presque 0.8    la 16  me d  cimale pr  s.

Si on prend ce r  sultat et qu'on le multiplie par 10 on pense obtenir 8.0 soit l'entier 8. Or on obtient *7.99999999999999991118*, la conversion en entier donnera ... 7 !

7.4. Le codage caract  re

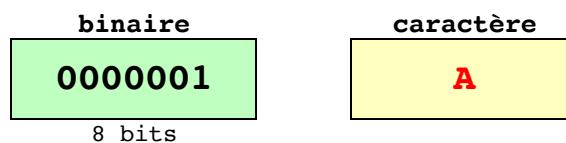
Un caract  re est une information non num  rique. Nous avons vu jusque-l   des conversions de donn  es num  riques (quantit  , entier, r  el) en binaire, mais comment coder une information non num  rique ?

7.4.1. Le principe

Pour coder un caract  re, on va simplement num  roter l'ensemble des caract  res, puis coder le num  ro de ce caract  re. Ceci pourrait donner le codage suivant :

A N   1 00000001
B N   2 00000010 ...

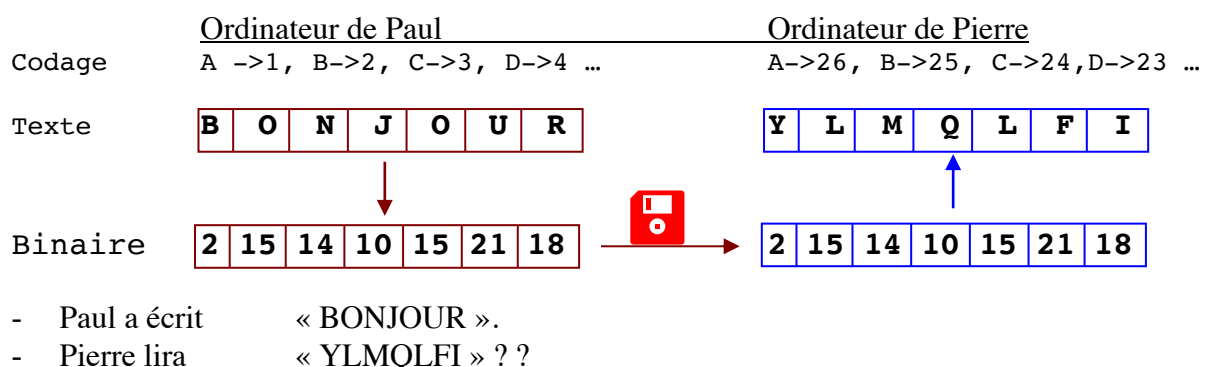
Avec cette num  rotation, le codage binaire du « A » serait identique    celui de la quantit   1. Le codage d'un caract  re se fera sur 8 bits. Ceci permet de coder au plus 256 caract  res (de 0    255).



Cependant cette convention arbitraire posera des probl  mes d'  change d'information entre des ordinateurs utilisant des conventions diff  rentes, comme le montre l'exemple suivant :

- L'ordinateur de Paul utilise une num  rotation des caract  res est dans l'ordre alphab  tique.
- L'ordinateur de Pierre utilise une num  rotation des caract  res est dans l'ordre inverse.

Voil   ce qui se passera si Paul   crit un texte « BONJOUR, ... », et qu'il le transmet    pierre par une disquette.

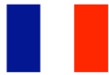


Le code binaire est bien le m  me, mais son interpr  tation diff  rente sur chaque machine donne un texte diff  rent. Cette d  monstration « grossi  re » montre le fond du probl  me !

Il est imp  ratif de normaliser une table de codage des caract  res !

7.4.2. Les tables normalisées

Chaque pays possède un organisme de normalisation :



AFNOR Association Française pour la **NOR**malisation



ANSI American National Standard Institute



BSI British Standard Institute

Il existe aussi un organisme international ISO



ISO International Standard Organisation

7.4.2.1. La table ISO

L'organisme ISO a proposé une table de codage de 256 caractères, structurée en deux parties :

- De 0 à 127 les caractères informatiques et les caractères alphabétiques.
- De 128 à 255 les accents sans déplacement (pas les caractères accentués). Cette table permet d'avoir tous les accents pour l'ensemble des langues nationales.

Cette table utilise tous les codages binaires possibles de l'octet (256 codages). On dit que c'est une [table de codage sur 8-bits](#).

7.4.2.2. La table ASCII

Les Américains n'utilisant pas d'accent, ont choisi de ne pas utiliser cette table internationale normalisée ! Ils ont défini leur propre table reprenant **la première partie de la table ISO** (128 caractères).

Ceci a eu une grande incidence pour l'informatique mondiale. A cette époque, tout ce qui se faisait en informatique venait des Etats-Unis. Ainsi tous les informaticiens tapaient du texte sans aucun accent. Ceci ne posait pas de réels problèmes puisqu'ils écrivaient avant tout des programmes.

Cette table se nomme la table **ASCII** (American Standard Code for Information and Interchange)

Cette table n'utilise pas tous les codages binaires possibles de l'octet (128 codages). Seuls 7 bits de l'octet sont utilisés.

On dit que c'est une **table de codage sur 7-bits**.

Le 8^{ème} bit reste toujours à 0 dans ce codage.

Table **ASCII** (Normalisée)

Décimal	ASCII	Décimal	ASCII	Décimal	ASCII
0	NUL	43	+	86	V
1	SOH	44	,	87	W
2	STX	45	-	88	X
3	ETX	46	.	89	Y
4	EOT	47	/	90	Z
5	ENQ	48	0	91	[
6	ACK	49	1	92	\
7	BEL	50	2	93]
8	BS	51	3	94	^
9	HT	52	4	95	_
10	LF	53	5	96	`
11	VT	54	6	97	a
12	FF	55	7	98	b
13	CR	56	8	99	c
14	SO	57	9	100	d
15	SI	58	:	101	e
16	DLE	59	;	102	f
17	DC1	60	<	103	g
18	DC2	61	=	104	h
19	DC3	62	>	105	i
20	DC4	63	?	106	j
21	NAK	64	@	107	k
22	SYN	65	A	108	l
23	ETB	66	B	109	m
24	CAN	67	C	110	n
25	EM	68	D	111	o
26	SUB	69	E	112	p
27	ESC	70	F	113	q
28	FS	71	G	114	r
29	GS	72	H	115	s
30	RS	73	I	116	t
31	US	74	J	117	u
32	Space	75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	\$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	
39	'	82	R	125	}
40	(83	S	126	~
41)	84	T	127	DEL
42	*	85	U		

7.4.3. Les tables non normalisées

7.4.3.1. La table EBCDIC

Parmi les grandes sociétés américaines, certaines n'utilisaient pas la table ASCII mais leur propre table de codage. C'est le cas d'**IBM** qui utilisait une table propriétaire **EBCDIC** pour ses serveurs. Il fallait alors des programmes de traduction pour convertir un texte ASCII en un texte EBCDIC ! Désormais IBM utilise la table ASCII.

7.4.3.2. La table IBM (ASCII étendue)

Le problème du codage des caractères nationaux (avec des accents) s'est de nouveau posé à IBM lorsque la société décida de vendre un micro-ordinateur dans le monde entier et de conquérir le marché de l'informatique personnelle.

IBM décida d'inventer une nouvelle table de codage au lieu de prendre la table ISO normalisée ! Pourquoi faire simple quand on peut faire compliqué !

Cette table utilisant la table ASCII comme première partie fut nommée par IBM : **ASCII étendue**. Cet abus de dénomination laisse entendre que cette table est normalisée ce qui est faux !

De plus cette table **ne contenait pas tous les caractères accentués**. C'était le cas de certains caractères en portugais.

Pour résoudre ce dernier problème on introduisit la notion de **Page de Codes** .

Une page de code **est la deuxième partie de la table de codage** (caractères dont le code varie de 128 à 255). Elle peut être **changée dynamiquement en cours d'une session** dans l'environnement d'un logiciel. Les différentes pages de codes correspondent aux caractères spécifiques des langues nationales.

Voici deux pages de codes :

Page de code 850	Multilingue
Page de code 437	Française

La page 850 (Multilingue) est la page standard dans le monde des PC. Elle contient entre autres tous les caractères accentués de la langue française. Nous n'avons pas besoin de la page 437 sauf peut être pour certains signes monétaires.

7.4.3.3. La table APPLE (ASCII étendue)

Pour la commercialisation de ses micro-ordinateurs, APPLE inventa aussi une table de codages de 256 caractères (ASCCI + caractères accentués). Pour corser le tout, les caractères accentués ne sont pas codés avec les mêmes numéros que ceux des PC, et le nom de cette table est ASCII étendue ! La table APPLE est donnée en annexe.

7.4.4. Les pages de codes

7.4.4.1. La page 850 (IBM)

La page de code 850 est utilisée par un PC sous DOS par exemple. Elle est donnée en annexe.

7.4.4.2. La page 8859-1 (Iso Latin-1)

ISO a ensuite normalisé plusieurs pages de codes dont l'**ISO Latin-1**.
La page de code **8859-1** est donnée en annexe.

7.4.4.3. La page ANSI 1252 (Windows)

Un PC sous Windows utilise généralement la page de code Normalisé ANSI. Cette table reprend le codage ISO Latin-1 avec quelques caractères en plus. Elle est donnée en annexe avec la table ISO Latin-1.

7.4.5. Conclusion

7.4.5.1. Incompatibilité PC, MAC, UNIX

Le but initial était de définir un codage « binaire » des caractères normalisé et universel. Ceci afin de faciliter les échanges d'informations. **Les enjeux commerciaux ont abouti à la solution inverse** : chacun sa table de codage.

Les machines UNIX utilisent généralement la table ASCII, les PC la table IBM (page de code 850 ou ANSI 1252 sous Windows) et les Macintosh la table APPLE.

Ainsi le caractère « é » tapé sur un PC utilisant la page de code 850 aura le code 130.

Sur un Macintosh ou une machine UNIX, ce caractère aura la forme suivante :

Code décimal	PC	Macintosh	Unix
130	é	ç	inexistant

Le mot « était » tapé sur un PC, sera affiché avec la forme :

PC	Macintosh	Unix
était	çtait	taït

La solution est de taper des textes sans accents ! C'est malheureusement le seul moyen sûr.

7.4.5.2. Compatibilité via l'ISO Latin-1

La page de code normalisé ISO Latin-1 propose une solution pour **l'échange de courrier**. Dans le cas du courrier électronique les logiciels tels Eudora, Netscape, ou d'autres, permettent de choisir la « police de caractères » ISO Latin-1. Dans ce cas, le texte que vous tapez avec ce logiciel est codé en page de code 8859-1 pour les caractères accentués. Si le destinataire utilise un logiciel possédant les mêmes qualités, il pourra paramétrer de la même manière son « lecteur de courrier » et ainsi voir les caractères accentués que vous avez effectivement tapés.

ATTENTION !

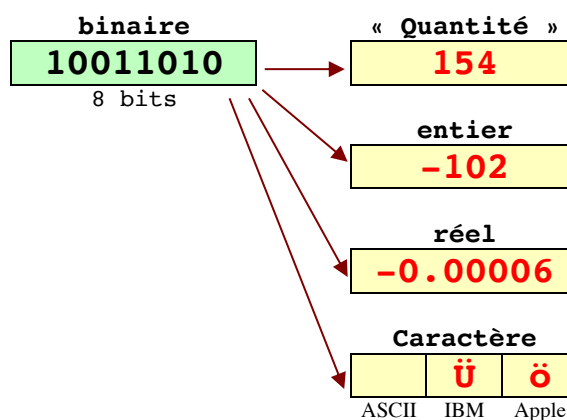
Si vous décidez de sauvegarder votre courrier sur disque afin de lire le texte avec un autre logiciel, vous sortez de votre environnement « ISO Latin-1 » pour revenir dans l'environnement par défaut de votre Micro (PC, MAC) donc avec une autre page de code.

Dans ce cas, votre fichier sera mal interprété et affiché avec les mauvais accents par un autre logiciel comme un traitement de texte.

De plus lorsque le système possède une **interface graphique** (Windows, UNIX avec X11, CDE,KDE) il est possible de paramétrer la police ISO Latin-1 Par défaut. Dans ce cas le texte devient « échangeable » car il est codé de la même manière au moment de son écriture.

7.5. Résumé des codages

Un même octet peut être interprété de différentes manières. Si nous considérons que les entiers et les réels sont également codés sur un octet, alors notre octet de départ (**10011010**) peut être interprété de plusieurs manières.



Pour interpréter correctement un ou plusieurs octets il faut savoir comment il a été codé !

Les logiciels résolvent ce problème en mettant sous forme binaire leur « **signature** » dans leur fichier. Ainsi avant de lire un fichier, le logiciel vérifie qu'il sait le décoder en cherchant sa signature. S'il ne la trouve pas alors il arrête l'interprétation qui serait fausse.

Ce n'est pas le cas de la commande MSDOS « type » qui interprète tous les octets en codage caractère. Appliquer cette commande sur un fichier binaire permet de vérifier cette interprétation erronée :

```
type cv.doc
type word.exe
```