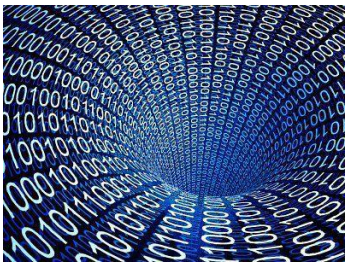
Représentation binaire



Les ordinateurs et les programmes mémorisent, transmettent et transforment des données aussi variées que des nombres, des textes, des images, des sons, etc. Pourtant, quand on les observe à une plus petite échelle, les ordinateurs ne manipulent que des objets beaucoup plus simples : des 0 et 1. Voyons comment cela suffit à représenter la plupart des nombres.

I Décrire des entiers naturels.

Le tableau suivant donne les 10 premiers nombres en base 10 dans la colonne du milieu et leur représentation binaire dans la colonne de droite :

0 x *8* + **0** x *4* + **0** x *2* + **0** x *1* =	0	**0000**
0 x *8* + **0** x *4* + **0** x *2* + **1** x *1* =	1	**0001**
0 x *8* + **0** x *4* + **1** x *2* + **0** x *1* =	2	**0010**
0 x *8* + **0** x *4* + **1** x *2* + **1** x *1* =	3	**0011**
0 x *8* + **1** x *4* + **0** x *2* + **0** x *1* =	4	**0100**
0 x *8* + **1** x *4* + **0** x *2* + **1** x *1* =	5	**0101**
0 x *8* + **1** x *4* + **1** x *2* + **0** x *1* =	6	**0110**
0 x *8* + **1** x *4* + **1** x *2* + **1** x *1* =	7	**0111**
1 x *8* + **0** x *4* + **0** x *2* + **0** x *1* =	8	**1000**
1 x *8* + **0** x *4* + **0** x *2* + **1** x *1* =	9	**1001**

Si vous observez bien ce codage, vous constaterez quelque chose de plus : il est le résultat d'un calcul, montré dans la colonne de gauche. Le codage binaire d'un nombre (que nous, humains, écrivons quasiment toujours en base 10, en utilisant 10 chiffres), *est ce même nombre, écrit en base 2*.

Du codage binaire au nombre entier positif

Voici le calcul qui permet de retrouver un nombre à partir de son code binaire : additionner les bits, chacun multiplié par la puissance de deux qui lui correspond, comme ci-dessous, donne le nombre en base 10.

$$\begin{array}{l|l} \mathbf{1} \times 8 + \mathbf{0} \times 4 + \mathbf{0} \times 2 + \mathbf{1} \times 1 = & \\ \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 = & 9 \\ ((\mathbf{1} \times 2 + \mathbf{0}) \times 2 + \mathbf{0}) \times 2 + \mathbf{1} = & \end{array}$$

- À la première ligne, nous avons multiplié respectivement par 1, 2, 4, 8 de droite à gauche les bits qui représentent 9.
- À la seconde ligne, nous nous sommes aperçus que 1, 2, 4, 8 sont en fait les puissances de 2 : 2^0, 2^1, 2^2...

- À la troisième ligne, nous factorisons 2 autant que possible ; cette écriture avec ses parenthèses bien placées nous montre qu'il suffit de multiplier par deux puis d'additionner les bits de gauche à droite pour trouver le nombre en base 10.

Voilà donc un calcul, une méthode mécanique que l'on appelle **algorithme**, qui permet de passer du codage binaire au nombre dans sa représentation usuelle.

Du nombre entier positif à son codage binaire

Réciproquement, il existe un calcul pour obtenir le code binaire de n'importe quel nombre entier. Si nous notons 9 % 2 = 1 le reste de la division par 2 de 9 ou 4 % 2 = 0 le reste de la division de 4 par 2, etc., alors le calcul suivant, où nous calculons successivement le reste de la division par deux puis divisons par deux nous donne... le codage binaire du nombre 9 :

$$9 \,\%\, 2 = \mathbf{1}$$
$$(9 \;/\; 2 = 4) \,\%\, 2 = \mathbf{0}$$
$$(4 \;/\; 2 = 2) \,\%\, 2 = \mathbf{0}$$
$$(2 \;/\; 2 = 1) \,\%\, 2 = \mathbf{1}$$

Représenter des entiers positifs de toutes tailles

Bien sûr, cela marche avec tous les nombres entiers positifs, et nous pouvons facilement calculer combien de bits sont nécessaires pour les coder :

Entiers	0 à 1	0 à 3	0 à 7	0 à 15	0 à 255	0 à 1023	0 à environ 4 milliards	0 à plus de 1.8×10^{20}
Codés sur	1 bit	2 bits	3 bits	4 bits	8 bits	10 bits	32 bits	64 bits

Nos ordinateurs utilisent aujourd'hui souvent 32 ou 64 bits pour coder les nombres entiers, ils peuvent donc coder directement des nombres très grands et, si besoin était, en utilisant plus de bits... des nombres vertigineusement grands !

Exercice :

a) Calculer la valeur décimale des octets :1010 1101 , 0111 0010 et 1011 0011
b) Faire la conversion décimal / binaire de 33, 127 , 139 , 211
c) Comment réaliser l'addition de nombres codés en binaire ?

II Représenter des nombres entiers négatifs et positifs.

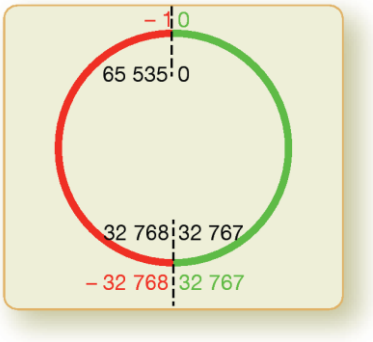
Pour représenter les entiers relatifs en notation binaire, on doit étendre la représentation aux nombres négatifs. Une solution est de réserver un bit pour le signe de l'entier à représenter et d'utiliser les autres pour représenter sa valeur absolue. Ainsi, avec des mots de 16 bits, en utilisant 1 bit pour le signe et 15 bits pour la valeur absolue, on pourrait représenter les entiers relatifs de $-111111111111111_2 = -(2^{15}-1)_{10} = -32767_{10}$ à $111111111111111_2 = (2^{15}-1)_{10} = 32767_{10}$.
Cependant, cette méthode a plusieurs inconvénients, l'un d'eux étant qu'il y a deux zéros, l'un positif et l'autre négatif.

On a donc préféré une autre méthode, qui consiste à représenter un entier relatif par un entier naturel. Si on utilise des mots de 16 bits, on peut représenter les entiers relatifs compris entre -32 768 et 32 767 : on représente :

- un entier relatif x positif ou nul comme l'entier naturel x
- et un entier relatif x strictement négatif comme l'entier naturel $x+2^{16}=x+65536$, qui est compris entre 32 768 et 65535.

Ainsi, les entiers naturels de 0 à 32 767 servent à représenter les entiers relatifs positifs ou nuls, à droite sur la figure, et les entiers naturels de 32 768 à 65 535 décrivent les entiers relatifs strictement négatifs, à gauche sur la figure.

L'entier relatif -1 est représenté comme l'entier naturel $-1+2^{16}=65535$, c'est à dire par le mot 1111111111111111. Cette manière de représenter les entiers relatifs s'appelle la notation en **complément à deux**.



Avec des mots de seize bits, on écrit les entiers relatifs compris entre $-2^{15}=-32768$ et $2^{15}-1=32767$.

Plus généralement, avec des mots de n bits, on écrit les entiers relatifs compris entre -2^{n-1} et $2^{n-1}-1$:

- un entier relatif x positif ou nul compris entre 0 et $2^{n-1}-1$ est représenté par l'entier naturel x compris entre 0 et $2^{n-1}-1$;
- un entier relatif x strictement négatif compris entre -2^{n-1} et – 1 est représenté par l'entier naturel $x+2^n$ compris entre 2^{n-1} et 2^n-1 .

Exercice : Quels entiers relatifs peut-on représenter avec des mots de 8 bits ? Combien sont-ils ?
Même questions avec des mots de 16 bits, 32 bits et 64 bits.

Exercice : Trouver la représentation binaire sur huit bits des entiers relatifs 0 et -128.

Exercice : Trouver la représentation décimale des entiers relatifs dont la représentation binaire sur huit bits est 0111 1111 et 1000 0001.

Exercice : Calculer la représentation binaire sur huit bits de l'entier relatif 4, puis celle de son opposé.

Algorithme de conversion d'un nombre en binaire complément à 2 sur un octet :

Trouver l'opposé d'un nombre en complément à 2 :

Soit $n = 0000\ 0101_2 = 5_{10}$

Son opposé est -5 obtenu en inversant tous les 1 en 0 et tous les 0 en 1 et en ajoutant 1:

0000 0101 --> 1111 1010 + 1 = 1111 1011 = -5

Vérification : 5 + (-5) = 0 et $0000\ 0101_2 + 1111\ 1011_2 = 00000000_2$

Exercice : Coder -15 en binaire sur un octet, puis sur 16 bits.

III Représenter les nombres réels.

Comme la notation décimale, la notation binaire permet aussi de représenter des nombres à virgule. En notation décimale, les chiffres à gauche de la virgule représentent des unités, des dizaines, des centaines, etc. et ceux à droite de la virgule, des dixièmes, des centièmes, des millièmes, etc. De même, en binaire, les chiffres à droite de la virgule représentent des demis, des quarts, des huitièmes, des seizièmes, etc. On peut ainsi représenter, le nombre un et un quart : 1.01. Toutefois, cette manière de faire ne permet pas de représenter des nombres très grands ou très petits comme le nombre d'Avogadro ou la constante de Planck. On utilise donc une autre représentation similaire à la notation scientifique des calculatrices, sauf qu'elle est en base deux et non en base dix. L'**IEEE 754** est un standard pour la représentation des nombres à virgule flottante en binaire.

Un nombre est représenté sous la forme $\boxed{s\ \ (1+m)\times 2^n}$ où :

- s est le **signe** du nombre : 0 si +, 1 si -
- n son **exposant**
- m sa **mantisse** : c'est un nombre à virgule, compris entre 0 inclus et 1 exclu.

L'exposant peut être positif ou négatif. Cependant, la représentation habituelle des nombres signés (complément à 2) rendrait la comparaison entre les nombres flottants un peu plus difficile. Pour régler ce problème, l'exposant est décalé, afin de le stocker sous forme d'un nombre non signé. Ce décalage est de $2^{e-1}-1$ (*e* représente le nombre de bits de l'exposant) ; il s'agit donc d'une valeur constante une fois que le nombre de bits *e* est fixé.

Si un nombre réel x s'écrit :

$$x = \pm\ 2^{e_R} \times (1+m)$$

on posera $e_C = e_R + (2^{n-1}-1)$ si l'exposant est codé sur n bits.

Par exemple :

- **Quand on utilise 32 bits** : pour représenter un nombre à virgule on utilise 1 bit de signe, 8 bits pour l'exposant et 23 pour la mantisse. L'exposant e_R est donc décalé de $2^{8-1}-1=127$.

$$e_C = e_R + 217$$

L'exposant d'un nombre normalisé va donc de -126 à +127. L'exposant -127 (qui est décalé vers la valeur 0) est réservé pour zéro, tandis que l'exposant 128 (décalé vers 255) est réservé pour coder les infinis.

-**quand on utilise 64 bits :** 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse. Le signe + est représenté par 0 et le signe - par 1.
L'exposant e_R est un entier relatif représenté comme l'entier naturel
$e_C = e_R + (10^{11-1}-1) = e_R + 1023$ qui est compris entre 1 et 2 046.
La mantisse m est un nombre binaire à virgule compris entre 1 inclus et 2 exclu, comprenant 52 chiffres après la virgule.

$$e_C = e_R + 1023$$

Exemple : Trouver le nombre à virgule représenté par le mot de 64 bits
1100010001101001001111000011100000000000000000000000000000000000

Exemple : trouver la représentation sur 32 bits du réel 278.

IV Les unités de base.

Pour des raisons pratiques, les bits sont regroupés par paquets adjacents pour représenter de l'information : Un **octet** (byte) est constitué de 8 bits (il peut représenter 256 valeurs). Historiquement cela correspondait au codage d'un caractère. L'octet est choisi comme unité pour représenter les capacités mémoire.

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

Si l'on numérote les bits de 0 à 7, le bit numéro 0 est le bit de poids faible, et le bit numéro 7 est le bit de poids fort.

Si l'on considère un octet comme un nombre écrit en base 2, sa valeur numérique est $\sum_{i=0}^{7} b_i \times 2^i$

11010110_2

Ici, on a $= 1\times 2^7 + 1\times 2^6 + 0\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0$

$= 214_{10}$

Standard SI		Unités binaires	
Unité	Valeur	Unité	Valeur
kilobit (kb)	10^3	kibibit (Kibit)	2^{10}
mégabit (Mb)	10^6	mébibit (Mibit)	2^{20}
gigabit (Gb)	10^9	gibibit (Gibit)	2^{30}
térabit (Tb)	10^{12}	tébibit (Tibit)	2^{40}
pétabit (Pb)	10^{15}	pébibit (Pibit)	2^{50}
kilooctet (ko)	10^3	kibioctet (Kio)	2^{10}
mégaoctet (Mo)	10^6	mébioctet (Mio)	2^{20}
gigaoctet (Go)	10^9	gibioctet (Gio)	2^{30}
téraoctet (To)	10^{12}	tébioctet (Tio)	2^{40}
pétaoctet (Po)	10^{15}	pébioctet (Pio)	2^{50}
exaoctet (Eo)	10^{18}	exbioctet (Eio)	2^{60}

Usage traditionnel $\neq$ notations officielles :

$1ko = 2^{10} octets = 1024 octets$

$1Mo = 2^{20} \quad octets = 2^{10} ko = 1048576 \quad octets$

$1To = 2^{40} \quad octets = 1024 Go = 1099511627776 \quad octets$

Un **mot** est la quantité de bits standard manipulée par un microprocesseur (CPU). La taille du mot s'exprime en bits ou en octets. Un microprocesseur est d'autant plus performant que ses mots sont longs, car les données qu'il traite à chaque cycle sont plus nombreuses.
C'est pourquoi on classe les microprocesseurs par la taille de leur mot (16, 32, 64 bits : 2, 4 ou 8 octets).