Chapitre 1

Numération binaire : opérations booléennes

Pourquoi le binaire ?

Les ordinateurs fonctionnent à l'électricité donc les informations sont représentées par l'état électrique (disons la tension) en certains points ; il y a de rares cas où la transmission se fait par rayons lumineux, mais, de toutes façons, on convertit en signaux électriques. Donc, pour transmettre une information élémentaire ou l'utiliser en vue d'un calcul, un organe de l'ordinateur devra « mesurer » la tension électrique en un certain point.

Le problème est que la tension mesurée est affectée par des parasites qui risquent d'être d'autant plus importants que les fréquences mises en jeu sont plus élevées, donc que la machine est plus rapide. En somme, $V_{mesuré}=V_{voulu}$ +P. Supposons que cette information élémentaire ait n états possibles V_1, V_2, … V_n, représentant n significations possibles.

Nous supposons les V_i en ordre croissant et P amplitude des parasites positif. Si jamais un V_i+P devenait supérieur ou égal à V_{i+1}, l'ordinateur se mettrait à calculer faux puisqu'il prendrait l'information i+1 alors qu'il devait prendre l'information i. Conclusion, il faut maintenir P aussi petit que possible, mais on n'est pas totalement maître des parasites et assurer entre les états significatifs la distance la plus grande possible.

Maintenant dans tout circuit électrique où la tension d'alimentation est V_{DD} (c'est la notation standard), on peut toujours dire que l'un des pôles de l'alimentation est à V=0 et, à ce moment, en tout point du circuit, la tension sera comprise entre 0 et V_{DD}. Donc si on veut maximiser l'écart entre les tensions des états significatifs en un point, **il faut se limiter à 2 états**, que nous noterons provisoirement A (avec V_A voisin de 0 volt) et B (avec V_B voisin de V_{DD}).

Voilà pourquoi les ordinateurs fonctionnent en binaire.

Représentation des nombres

Donc un point élémentaire ne peut représenter qu'une information très réduite comme la réponse par « oui » ou « non » à une question ou bien une assertion « vrai » ou « faux » ou encore les chiffres « 0 » ou « 1 ».

> Une telle information élémentaire s'appelle un **bit** (de l'anglais binary digit), vous le savez sans doute déjà.

Il faut donc tout un ensemble de bits pour représenter une information conséquente. Les groupes les plus utilisés sont de 8 bits (**octet**), de 16, 32 ou 64 bits. Lorsqu'on parle de **mot**, c'est souvent 32, mais il est conseillé de préciser. On a souvent besoin de « paquets » encore plus vastes.

Si on voulait trouver une information en posant une suite de questions auxquelles on ne peut répondre que par oui ou par non, il faudrait un grand nombre de questions.

Nombres entiers positifs

La théorie des bases de numération nous donne immédiatement une représentation des nombres avec seulement deux symboles : il suffit de prendre le système à base 2, avec pour chiffres 0 et 1. Vous savez qu'étant donné une base B et un ensemble de B chiffres de valeurs 0 à B-1, le nombre écrit avec p chiffres $a_{p-1}a_{p-2}\ldots a_1a_0$ a pour valeur $a_0+a_1B+a_2B^2+\ldots+a_{p-1}B^{p-1}$ soit $N=\Sigma_0^{p-1}a_iB^i$ avec $a_i<B$.

> En binaire, $a_{p-1}a_{p-2}\ldots a_1a_0$ a pour valeur $a_0+a_1.2+a_2.4+\ldots+a_{p-1}.2^{p-1}$avec a_i=0 ou 1.

Comme vous le savez, les a_i s'obtiennent à partir de la droite comme restes successifs des divisions du nombre par B. Aussi, cette représentation est unique pour une base donnée.

Applications

Combien vaut 11010 ?

0x1+1x2+0x4+1x8+1x16=26

Écrire 74 en binaire

Le 1er reste est 0. 37 divisé par 2 donne 18, reste 1. Ensuite 9 reste 0, 4 reste 1, 2 reste 0, et le dernier quotient 1 est en même temps le dernier reste d'où $74_{10}=1001010_2$.

> Quand on manipule simultanément des bases différentes notamment pour des conversions, on indique la base en indice.

Autre manière de créer la représentation binaire

Nous avons parlé des suites de questions auxquelles on ne peut répondre que par oui ou non. Supposons que nous ayons à deviner un nombre compris entre 0 et 255 par une telle suite de questions. Voici la suite de questions que nous posons et un exemple de réponses obtenues :

1 - Le nombre est-il supérieur ou égal à 128 ? Non

2 - Le nombre est-il supérieur ou égal à 64 ? Oui

3 - Le nombre est-il supérieur ou égal à 96 ? Non

4 - Le nombre est-il supérieur ou égal à 80 ? Non

5 - Le nombre est-il supérieur ou égal à 72 ? Oui

6 - Le nombre est-il supérieur ou égal à 76 ? Non

7 - Le nombre est-il supérieur ou égal à 74 ? Oui

8 - Le nombre est-il supérieur ou égal à 75 ? Non

Donc, nous en concluons que le nombre est 74 (le même que celui de la 2^{e} application ci-dessus).

Si nous écrivons la suite des réponses en remplaçant oui par 1 et non par 0, nous obtenons 01001010, donc, à part le 0 en tête qui ne change pas la valeur, la même chose que par la théorie des bases de numération.

Mais la 2^{e} méthode va nous permettre d'autres découvertes.

Optimalité

Cette représentation qui est **unique** (d'après la théorie des bases) est **optimale**. En effet, les questions que nous avons posées ont pour effet à chaque fois de diviser par 2 l'intervalle de recherche : au départ, il a la largeur 256 ; la réponse à la 1re question nous montre que le nombre est dans la moitié gauche de 0 à 127 ; nous posons la 2^{e} question par rapport au milieu de la 1re moitié, soit 64 et les questions suivantes prennent toujours le milieu de l'intervalle restant jusqu'à un intervalle de largeur 1.

Cette méthode s'appelle une **recherche dichotomique** ; nous la retrouverons en algorithmique. En quoi est-elle optimale ?

Si nous divisons l'intervalle en deux parties inégales, on y gagnera s'il est dans la partie la plus petite, mais on y perdra s'il est dans la plus grande. Comme nous ne savons rien sur le nombre cherché, sa probabilité est uniforme sur l'intervalle, il y a plus de chances qu'il soit dans la partie la plus grande, donc la probabilité est plus grande pour qu'on y perde si on ne divise pas en deux moitiés égales.

Capacité de la représentation

On voit que pour pouvoir trouver un nombre de 0 à 255, donc parmi 256 valeurs possibles, il faut un minimum de 8 questions, donc 8 bits. La théorie des bases de numération le montre aussi : avec P chiffres en base B, les valeurs possibles vont de 0 à B^{P-1} et il y a B^{P} valeurs possibles. Ainsi, en base 10, vous savez qu'avec 4 chiffres, on va jusqu'à 9999, soit 10^{4}=10000 valeurs possibles.

> En binaire, avec P bits, on va jusqu'à 2^{P}-1 soit 2^{P} valeurs possibles ou 2^{P} motifs binaires différents possibles. Inversement, pour pouvoir représenter le nombre N, il faut au moins $\log_2(N+1)$ bits, soit $3{,}3\log_{10}(N+1)$.

En effet $N \leq 2^{P}-1$, donc $2^{P} \geq N+1$, soit $P \geq \log_2(N+1)$.

Valeurs à connaître

Bits	Nombre de valeurs	0 à …
4 (3)	16 (8)	15 (7)
8 (7)	256 (128)	255 (127)

16 (15)	65536 (32768)	65535 (32767)
32 (31)	4,2… (2,1…) milliards	4,2… (2,1…) milliards

Nous avons mis entre ( ) les valeurs pour P-1 bits car elles vont nous servir tout de suite.

Nombres entiers signés

Comme il n'y a que deux signes, la première manière qui vient à l'idée est d'utiliser la représentation ci-dessus pour la valeur absolue, précédée d'un bit qui représentera le signe. C'est la représentation « signe, magnitude » ; elle a été effectivement utilisée avec bit de signe 0 pour +, 1 pour – ; ainsi pour les nombres positifs signés, il n'y a pas de différence avec les nombres non signés.

Par exemple, si on veut 8 bits au total, la valeur absolue sera sur 7 bits, donc vaudra 127 au maximum.

Exemples : 3 : 00000011 ; -3 : 10000011 ; 127 : 01111111 ; -127 : 11111111.

Cette représentation a un inconvénient : on ne procède pas de la même manière pour ajouter un nombre positif ou un nombre négatif, ce qui complique les organes de calcul.

Atelier

Raisonnons sur 3 bits. Les nombres sont :

0	000		
+1	001	-1	101
+2	010	-2	110
+3	011	-3	111

Essayons l'addition -2 + +2 : -2 110

\+ +2 010

1000 donc 0 car le 1 en 4e position sort de la représentation. Mais -1 101

\+ +1 001

110 soit -2 ; cela ne va pas car nous voudrions pouvoir procéder de la même façon quels que soient les signes et les valeurs.

> Les calculs se font exactement comme en décimal, sauf que la retenue se produit dès qu'on arrive à 2 au lieu de 10.

Complémentation

Suite de l'atelier

Pour passer d'un nombre positif à son opposé, une autre manière d'inverser le bit de signe est d'inverser tous les bits. Essayons sur les exemples précédents :

+2	010		+1	001
+ -2	101		+ -1	110
	111			111

On obtient le complément de 0, c'est encourageant. Essayons alors d'ajouter 1 après inversion : cela constitue le complément à 2.

> Pour représenter l'opposé d'un nombre par son complément à 2, on inverse les bits et on ajoute 1. Dans toutes les opérations, on néglige toute retenue qui sort à gauche du nombre.

Voici les nombres dans notre exemple à 3 bits :

0	000				
+1	001	-1	111	-(-1)	001
+2	010	-2	110	-(-2)	010
+3	011	-3	101	-(-3)	011

Si on refait l'opération, on retombe sur le nombre positif ; c'est ce qu'il fallait assurer. Appliquée à 000, l'opération redonne 000 : -0 = 0.

Voici quelques additions :

+2	010	+1	001	-2	110	-1	111	+3	011
+ -2	110	+ -1	111	+ +1	001	+ +2	010	+ -1	111
1	000	1	000	-1	111	+1	1	001	+2	1	010

On a encadré les retenues négligées, puisqu'elles sont hors représentation.

Sur 3 bits, il y a un 8^{e} motif binaire : 100 ; que représente-t-il ? On voit que +3 (011) + -1 (111) → 1 100 donc on peut dire que 100 représente -4.

Dans cette représentation, on a un nombre négatif en plus que du côté des positifs : c'est pour compenser la place du 0.

Si on effectue +3 (011) + +1 (001), on trouve -4 (100). Il est venu une retenue sur le bit de signe, ce qui n'a pas de sens puisque le bit de signe ne contribue pas à la valeur : le motif binaire 100 vaudrait 4 si la représentation avait plus que 3 bits. Une telle situation, due à la limitation de la largeur de la représentation, où il arrive une retenue sur le bit de signe s'appelle un **débordement**. Si on ajoute 1 au plus grand nombre positif, on obtient le plus petit négatif : donc les nombres se replient sur eux-mêmes, on peut donc les figurer sur un cercle.

Les positifs et les négatifs se raccordent de part et d'autre la frontière des débordements -4|+3 en 3 bits. Plus la représentation est vaste, plus le cercle est large. En 8 bits, les nombres vont de -128 à +127 ; en 16 bits, ils vont de -32768 à +32767.

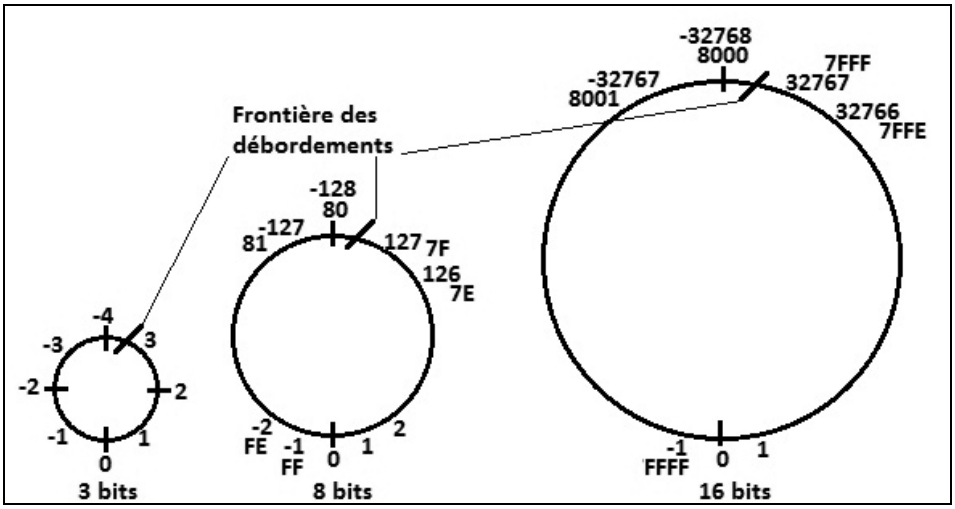

Figure 1 : Cercles des nombres entiers

La notation hexadécimale

Pour une valeur donnée, il faut beaucoup de chiffres en binaire. Si vous essayez de transmettre un nombre binaire au téléphone, vous verrez qu'il est impossible qu'il n'y ait pas d'erreurs. On a donc besoin d'une notation plus concise que le binaire telle que le passage entre elle et le binaire soit très facile. La solution est de faire appel à une base qui soit une puissance de 2. On a employé l'octal (base $8=2^3$) ; mais maintenant on emploie universellement l'hexadécimal (base $16=2^4$).

L'hexadécimal emploie 16 chiffres : les 10 premiers sont leurs correspondants du système décimal : 0…9. Ensuite on utilise les lettres de A (10) à F (15).

Pour convertir de binaire à hexadécimal, il suffit de grouper les bits 4 par 4 (en ajoutant des 0 à gauche si nécessaire) et de remplacer chaque groupe par le chiffre hexadécimal correspondant. D'hexadécimal à binaire, on remplace chaque chiffre par le motif binaire correspondant, selon le tableau :

Hexa	Binaire	Valeur	Hexa	Binaire	Valeur
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

Dans la figure 1, on a indiqué les valeurs hexadécimales correspondantes.

Comme les groupes de bits les plus utilisés actuellement sont l'octet (8 bits) et ses multiples 16, 32 et 64 bits, donc tous multiples de 4, l'hexadécimal est spécialement pratique. Dans tous les cas, -1 est représenté par plein 1, le maximum positif est 0 puis rien que des 1, le minimum négatif par 1 puis des 0,

donc en 8 bits : 2 chiffres, -1 → FF, 127 → 7F, -128 → 80 ;

en 16 bits : 4 chiffres, -1 → FFFF, 32767 → 7FFF, -32768 → 8000 ;

en 32 bits : 8 chiffres, -1 → FFFFFFFF.

Étant donné un nombre positif en 8 bits, par exemple 7F, le même nombre en 16 bits s'écrit 007F ; pour un négatif, on place FF à gauche : FF devient FFFF ; on dit qu'on étend le signe.

Exercice

Nous avons pris l'exemple du nombre 74_{10}. Écrivez-le en hexadécimal.

On a vu qu'il s'écrivait 01001010 en binaire, donc $4A_{16}$.

Nombres réels

En décimal, 31,54 = $3x10^1+1x10^0+5x10^{-1}+4x10^{-2}$.Donc en base 2, $a_3a_2a_1a_0,a_{-1}a_{-2}$ vaudra $a_3x8+a_2x4+a_1x2+a_0+a_{-1}x1/2+a_{-2}x1/4$ (les a sont 0 ou 1).

Tout le problème est de savoir où on met la virgule.

Virgule fixe

La virgule est toujours à la même place, par exemple, sur deux octets, le 1^{er} sera la partie entière, le 2^e la partie fractionnaire. En nombres positifs, on peut représenter de 0,00000001 à 11111111,11111111.

La limitation est que la précision relative possible varie avec l'ordre de grandeur du nombre.

Virgule flottante

On utilise une représentation **structurée** où les différents bits jouent des rôles différents (les entiers signés en sont déjà un exemple puisque le bit de signe joue un rôle spécial).

Le motif binaire sera la juxtaposition de deux parties : caractéristique | mantisse

Et le nombre vaudra $0,mantisse x 2^{caractéristique}$.

Par exemple, sur 16 bits, disons 4 pour la caractéristique, 12 pour la mantisse,

0100 001000000000 vaudra $0,001x2^4$, 0011 010000000000 vaudra $0,01x2^3$, c'est-à-dire la même chose : 2.

Donc la notation n'est pas univoque : il existe des motifs binaires synonymes. Parmi ceux-ci, celui dont le 1^{er} chiffre de la mantisse est 1 fournit la forme **normalisée**. Pour notre exemple, c'est 0010 100000000000 = $0,1 \times 2^2$.

Dans cette représentation, la précision relative est toujours la même : $1/2^{nombre\ de\ bits\ de\ la\ mantisse}$.

Les signes

Il y a le signe général du nombre et celui de la caractéristique. Le bit de signe général est mis à la place du 1er bit de la mantisse puisque, en forme normalisée, celui-ci n'a pas à être écrit vu que c'est toujours 1 ; mais dans les calculs, il devra être conservé à part car les résultats intermédiaires ne sont pas forcément normalisés ; c'est le problème de l'unité arithmétique de l'ordinateur, nous ne nous en préoccuperons pas. Ce procédé fait gagner 1 bit.

Pour la caractéristique, donc la puissance de 2, la 1re idée est de prendre la représentation des nombres signés que nous avons vue. Donc si la caractéristique est sur un octet, les exposants iront de -128 à +127. Mais, le plus souvent, on utilise la forme « exposant biaisé » (*biassed exponent*) où on ajoute 128 à tous les exposants (en éliminant les retenues) :

Exposant	Exposant Hexa	Caractéristique Hexa
+127	7F	FF
1	01	81
0	00	80
-1	FF	7F
-128	80	00

Performances

Soient p le nombre de bits de la mantisse et q celui de la caractéristique.

p détermine la précision relative puisque le dernier bit vaut 2^{-p} et c'est en valeur relative si on fait abstraction de la multiplication par $2^{exposant}$. Maintenant $2^{-p}=10^{-0,3.p}$.

> Pour que $2^x=10^y$, prenons le log à base 10 des deux membres ($\log_{10}10=1$) : $y=x.\log_{10}2$ d'où $y=0,3.x$.

Donc avec une mantisse de 24 bits, la précision sera $10^{-0,3.24}=10^{-7}$.

q détermine la gamme des nombres que l'on peut traiter. Avec une caractéristique sur un octet, on va de 2^{-128} à 2^{127}. Donc la plus grande valeur absolue envisageable est $10^{0,3.127}=10^{38}$. La plus petite valeur absolue qu'on peut distinguer de 0 est 10^{-38}.

> Nous avons terminé les représentations de données purement numériques. Les données que nous aborderons ensuite seront représentées par des motifs binaires ; c'est le traitement qu'on effectuera qui fera que ces motifs binaires représenteront les données voulues ; sinon, ils pourraient être pris pour des représentations de nombres ; c'est pourquoi on parle de **représentation numérique** des données.