V- Manipulations de nombres en binaire

L'ordinateur est constitué de milliards de transistors qui travaillent comme des interrupteurs électriques, soit ouverts soit fermés. Soit la ligne est activée, avec une tension proche de 5 volts, soit elle est désactivée, avec une tension proche de 0. *Fermé / ouvert* s'écrit aussi bien sous forme logique *Vrai / Faux*, ou encore sous forme numérique *1 / 0*. Justement l'arithmétique binaire, ou encore arithmétique en base 2, concerne les nombres écrits exclusivement avec les deux chiffres 0 ou 1. On pourrait aussi bien parler des mots qui s'écrivent avec les deux lettres 0 ou 1. Quand ces nombres sont tous formés de *n* chiffres successifs qui sont soit 0 soit 1 (ils sont de longueur *n*), on a vu qu'ils sont au nombre de 2^n. Par exemple le nombre qui s'écrit 0010110, de longueur 7, est l'un des 128 nombres de longueur 7 (écrits avec 7 *bits*). Les nombres qui peuvent s'écrire avec *n* bits vont de 0 à $2^n - 1$. Mais quelle est la valeur des nombres en binaire ainsi obtenus ?

Exercice : On est censé connaître $2^8 = 256$, $2^{10} = 1024$, $2^{15} = 32\,768$, $2^{16} = 65\,536$. Mais combien de nombres en binaire peut-on avoir sur 64 bits, ce qui correspond aux ordinateurs d'aujourd'hui ? Ce nombre est de 2^{64}, soit

$2^{64} = 2^{10 \cdot 6 + 4} = 2^{10 \cdot 6} \cdot 2^4 = (2^{10})^6 \cdot 16 = 16 \cdot (10^3)^6 = 16 \cdot 10^{18}$ = 16 milliards de milliards.

1) Les nombres entiers positifs ou nuls, en base 2

Pour comprendre, prenons d'abord les nombres que nous avons l'habitude de manipuler, ceux qui sont formés à partir des 10 chiffres de 0 à 9, ce que l'on appelle les nombres en base 10. Prenons le nombre qui s'écrit 1789. On dit que le chiffre des unités est 9, celui des dizaines est 8, celui des centaines est 7 et celui des milliers est 1. Cela signifie que si l'on a 1789 euros, on a 1000 pièces de 1 euro, plus 700 pièces de 1 euro, plus 80 pièces de 1 euro, plus 9 pièces de 1 euro. Autrement dit, $1789 = 1 \cdot 1000 + 7 \cdot 100 + 8 \cdot 10 + 9 = 1 \cdot 10^3 + 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0$. Par analogie, un nombre en binaire s'écrit avec des puissances successives décroissantes de 2 (au lieu de 10).

Exemple : $0\,0\,1\,0\,1\,1\,0 = 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

$6\;5\;4\;3\;2\;1\;0$

$= 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$ (en base 10)

Nous venons ainsi de convertir notre nombre (écrit en base 2) en nombre en base 10. Remarquons que le nombre en binaire s'écrit de gauche à droite suivant les puissances décroissantes de 2, on dit : des bits de poids fort vers les bits de poids faibles. Les *positions* des bits correspondent aux puissances successives de 2, elles sont marquées de droite à gauche, à partir de 0 (pour 2^0) à droite.

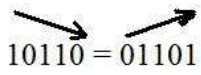
Deux remarques :

* Tous les 0 situés à gauche avant le premier 1 peuvent être supprimés :

0010110 = 10110. On peut aussi en rajouter autant qu'on veut à gauche:

10110 = 00000010110

* Au lieu d'écrire les nombres suivant les puissances décroissantes de 2, on peut, en le mettant à l'envers l'écrire suivant les puissances croissantes de 2. Il suffit de le préciser avec des flèches.



Exercice de programmation (en C)

Comment passer de l'écriture suivant les puissances décroissantes à celle suivant les puissances croissantes ? Par exemple on veut passer du tableau *ad*[*L*] contenant 10110 (avec ici *L* = 5) au tableau *ac*[*L*] contenant 01101, avec les indices des cases de 0 à *L* – 1.

```
for (i= 0 ; i < L ; i++)  ac[i]= ad[L – 1 –i] ;
```

ou encore, sans aucun risque, avec une variable *k* qui suit le mouvement de *i* :

```
k = L-1 ;
for(i=0 ; i< L ;i++)  { ac[i]= ad[k] ; k-- ; }
```

Comment faire la même chose, mais avec un seul tableau a[L], contenant au départ l'écriture suivant les puissances décroissances, et à la fin suivant les puissances croissantes ? Il s'agit de mettre à l'envers le tableau *a*[*L*] contenant par exemple 10110 pour *L* = 5.

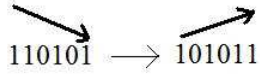
```
gauche = 0 ; droite = L – 1 ;
while (gauche < droite)  { aux = a[g] ; a[g] = a[d] ; a[d] = aux ;   /* échange des contenus des cases */
                           gauche ++ ; droite -- ;
                         }
```

1-1) Conversion d'un nombre (en binaire) en nombre (en base 10)

On sait déjà faire cela manuellement, par exemple ce nombre écrit suivant ses puissances décroissantes :

$110101 = 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$

On peut faire de même par programme. On commence par placer le nombre binaire dans un tableau *a*[*L*] avec *L* donné. Par souci de simplicité (mais ce n'est pas obligatoire) réécrivons le nombre suivant ses puissances croissantes, en le mettant à l'envers, toujours dans le même tableau, par exemple :



Enfin on parcourt le tableau *a*[*L*] en additionnant les termes successifs, tous de la forme $a[i] * 2^i$, avec *i* de 0 à *L* – 1.

```
gauche = 0 ; droite = L – 1 ;    /* mise à l'envers du tableau a[L]  */
while (gauche < droite)  { aux = a[g] ; a[g] = a[d] ; a[d] = aux ;   /* échange des contenus des cases */
                           gauche ++ ; droite -- ;
                         }
cumul=0 ;     /* à la fin, cumul contiendra le nombre en base 10 */
for(i = 0 ; i < L ; i++)  cumul += a[i] * pow(2, i) ;     /* pow est la fonction puissance */
printf(« %d », cumul) ;
```

On peut éviter la lourdeur de la fonction puissance *pow*, en constatant qu'à chaque étape la puissance est multipliée par deux. D'où la modification des dernières lignes du programme précédent, avec l'introduction de la variable *puissance* :

```
cumul=0 ;   puissance = 1 ;
```

```
for(i = 0 ; i < L ; i++)  { cumul += a[i] * puissance ;   puissance = 2*puissance ;}
printf(« %d », cumul) ;
```

Meilleure méthode de programmation

Plaçons le nombre binaire de longueur L (suivant ses puissances décroissantes) dans un tableau $a[L]$. Par exemple pour $L = 6$, le nombre 110101 est tel que $a[0]=1$, $a[1]=1$, $a[2]=0$, $a[3]=1$, $a[4]=0$, $a[5]=1$. On utilise alors une méthode dite à la Horner :

$$
\begin{aligned}
110101 &= a[0] \cdot 2^5 + a[1] \cdot 2^4 + a[2] \cdot 2^3 + a[3] \cdot 2^2 + a[4] \cdot 2^1 + a[5] \cdot 2^0 \\
&= 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 \\
&= 1 + 2\,(0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4) \\
&= 1 + 2\,(0 + 2\,(1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3)) \\
&= 1 + 2\,(0 + 2\,(1 + 2\,(0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2))) \\
&= 1 + 2\,(0 + 2\,(1 + 2\,(0 + 2\,(1 \cdot 2^0 + 1 \cdot 2^1)))) \\
&= 1 + 2\,(0 + 2\,(1 + 2\,(0 + 2\,(1 + 2 \cdot 1)))) \\
\text{soit} \quad &= a[5] + 2\,(a[4] + 2\,(a[3] + 2\,(a[2] + 2\,(a[1] + 2\,a[0]))))
\end{aligned}
$$

Pour le programme itératif, on va de droite à gauche pour le calcul. On commence par placer le $a[0]$ final dans une variable *cumul*, puis on double et on ajoute $a[1]$ pour avoir le nouveau *cumul*, puis on double et on ajoute $a[2]$, et ainsi de suite.

```
Mettre le nombre de longueur L donnée dans la tableau a[L]
cumul= a[0] ;
for(i=1 ; i < L ; i++)  cumul = 2*cumul + a[i] ;
printf(« %d », cumul) ;
```

1-2) Passage d'un nombre en base 10 au nombre en binaire

Il existe essentiellement deux méthodes.

1-2-1) Recherche répétée de la plus forte puissance de 2 inférieure ou égale au nombre

Prenons par exemple le nombre 13. La plus forte puissance de 2 qui est dans 13 est $8 = 2^3$.
$13 = 2^3 + 5$. Puis on recommence avec 5 :
$5 = 2^2 + 1$. On recommence avec 1
$1 = 2^0$. C'est fini.
Finalement $13 = 2^3 + 2^2 + 2^0 = 1101$ en binaire décroissant

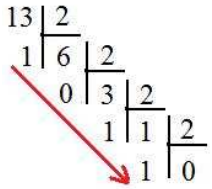
Programme (exhaustif, avec les déclarations)

```
#include <stdio.h>
#include <stdlib.h>
#define N 53
int a[10];

int main()
{ int nombre, puissance, exposant,nb1,exposantmax,i;

  nombre=N; nb1=0;
  while (nombre>0)
```

```
    { nb1++;
       puissance = 1; exposant = 0;
      while ( 2*puissance <= nombre)
        { puissance = 2* puissance;
          exposant ++;
        }
      if (nb1==1) exposantmax=exposant;
      a[exposant]=1;
      nombre -= puissance;
    }
  printf("Ecriture suivant les puissances croissantes\n");
  for(i=0;i<=exposantmax; i++) printf("%d ",a[i]);
  printf("\n\nEcriture suivant les puissances décroissantes\n");
  for(i=exposantmax; i>=0; i--) printf("%d ",a[i]);
  getchar();return 0;
}
```

1-2-2) Deuxième méthode, par divisions successives par 2

Reprenons le nombre 13 qui s'écrit 1101 en binaire décroissant. Comment passer de 13 à 1101 ? On commence par diviser 13 par 2, ce qui donne comme quotient 6 et comme reste 1. Ce reste est le dernier chiffre de l'écriture en binaire. Puis on recommence avec 6 :



On s'arrête lorsque le quotient est 0. La lecture des restes successifs donne 1011, il s'agit de l'écriture en binaire mais suivant les puissances croissantes de 2, ce qui s'écrit 1011. En le lisant à l'envers, on a l'écriture 1101 suivant les puissances décroissantes. Chaque division est de la forme

avec le nouveau quotient *q* obtenu à partir de l'ancien quotient *q*, sauf au départ, où l'on part du nombre 13. Pour des besoins d'unification on posera en conditions initiales *q* = 13 (en général *q* = *N*). On en déduit le programme, où l'on place les restes successifs dans un tableau de façon à pouvoir le lire à partir de la fin (suivant les puissances décroissantes) :

```
q = N; i=0;
  while (q>0)
   { r[i]= q%2;
     q = q/2;
     i++;
   }
  longueur=i;
  printf("Ecriture suivant les puissances croissantes :\n");
  for(j=0; j<longueur; j++) printf("%d ",r[j]);
  printf("\n\nEcriture suivant les puissances decroissantes :\n");
  for(j=longueur-1;j>=0; j--) printf("%d ",r[j]);
```

Cette deuxième méthode, moins évidente à trouver que la première, est nettement plus simple. Il convient de bien la connaître.

1-2-3) Tous les nombres en binaire de longueur *L* donnée

Une variante du problème précédent consiste à écrire tous les nombres ayant *L* chiffres en binaire, même s'il y a des 0 superflus. Maintenant, pour *L* = 5, 13 va s'écrire 01101 avec un zéro supplémentaire du côté des grandes puissances de 2. Il suffit pour cela de faire *L* divisions par 2. Les nombres concernés vont de 0 à $2^L - 1$ en décimal. Il n'y a plus qu'à faire les *L* divisions successives pour chacun de ces nombres.

Par exemple, pour *L* = 4, on a 16 nombres en binaire, associés aux nombres de 0 à 15 en base 10 :

0 : 0000, 1 : 0001, 2 : 0010, 3 : 0011, 4 : 0100, 5 : 0101, 6 : 0110, 7 : 0111, 8 : 1000 ; 9 : 1001, 10 : 1010, 11 : 1011, 12 : 1100, 13 : 1101, 14 : 1110, 15 : 1111

On en déduit le programme.

```
On se donne L
for (nombre = 0 ; nombre < pow(2,L) ; nombre++)
  { q=nombre ;  for(i=0 ; i<L ;i++)  { r[i]=q%2 ; q=q/2 ;}
    for(i=L – 1 ; i>=0 ; i--) printf(« %d »,r[i]) ;  /* affichage selon les puissances décroissantes */
    printf(« \n ») ;   /* aller à la ligne */
      }
```

1-2-4) Multiplications et divisions par 2, grâce à des décalages

Prenons un nombre, par exemple 13, qui s'écrit 1101 en binaire (décroissant). Si on le décale d'un cran vers la droite, on perd son dernier chiffre 1, et l'on obtient 110, soit 6. Ce nombre 6 justement le quotient de 13 divisé par 2. Le reste de la division, qui est 1, est le nombre perdu lorsque l'on passe de 1101 à 110. Il s'agit du dernier chiffre. Cela permet de comprendre pourquoi, dans les divisions successives par 2, les restes donnent le nombre binaire suivant ses puissances croissantes, et non pas décroissantes.

En langage *C*, l'opération de décalage vers la droite est >> , et vers la gauche c'est <<.

Par exemple 13>>1 signifie le décalage d'un cran, et donne 6, quotient de la division de 13 par 2, avec passage de 1101 à 110 en binaire.

13>>2 , décalage de deux crans, donne 3, correspondant au quotient de la division de 13 par 4, soit le passage en binaire de 1101 à 11.

13>>3 donne 1, correspondant à la division de 13 par 8, soit le passage en binaire de 1101 à 1.

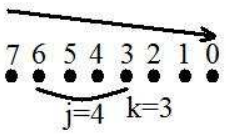
Inversement 13<<1, fait passer de 1101 à 11010, soit 26, le double de 13.

13<<2 multiplie le nombre 13 par 4, ce qui donne 52, ou 110100 en binaire.

Exercice de programmation

Un nombre entier positif *x* étant donné, dégager les *j* bits de son écriture en binaire à partir du bit numéro *k*. Précisons que *k* est l'exposant de la puissance concernée, cet indice se lit de droite à gauche dans l'écriture du nombre selon les puissances décroissantes, et les *j* bits sont les bits *k*, *k*+1, *k*+2, etc.

Par exemple, pour un nombre de 8 bits écrits suivant les puissances décroissantes de 2, *k* = 3 correspond à la puissance 2^3, et *j* = 4 correspond aux 4 bits qui se succèdent à partir de *k* = 3 vers les puissances croissantes.



On commence par décaler le nombre *x* vers la droite de *k* crans, soit *x* >>*k*, de façon que les *j* bits se trouvent à l'extrémité droite du nombre obtenu. Puis on fabrique un masque où les *j* bits de droite sont des 1 et tous les autres qui sont à gauche sont des 0, comme …00000001111. Comment faire pour avoir ce masque ? On commence par prendre le nombre binaire 0, puis on prend son complément ~0, soit …11111111. On le décale vers la gauche de *j* crans en faisant ~0<<j, ce qui donne …11110000. Enfin on prend son complément ~(~0<<j), d'où …00001111. Il reste à faire un *et* bit à bit (&) entre le nombre *x*>>*k* et le masque, ce qui respecte les *j* bits de droite de *x*>>*k* et met à 0 tous ceux de gauche. On a obtenu le nombre correspondant aux *j* bits cherchés. On en déduit la fonction *bits*(*x*, *k*, *j*) qui ramène ce nombre, que l'on peut écrire en binaire pour avoir les *j* bits cherchés.

```
int bits(int x, int k, int j)
{ return (x >> k) & ~(~0 << j); }
```

Exemple : *x* =238 soit 11101110 en binaire. On a pris *k* = 2, et *j* de 1 à 8

j

```
1:   1     1
2:   3     1 1
3:   3     0 1 1
4:  11     1 0 1 1
5:  27     1 1 0 1 1
6:  59     1 1 1 0 1 1
7:  59     0 1 1 1 0 1 1
8:  59     0 0 1 1 1 0 1 1
```

1-2-5) Addition et multiplication en binaire

On fait comme dans le système décimal, en posant l'opération à faire, avec un calcul final fait de droite à gauche [1], et en appliquant les règles d'addition ou de multiplication bit à bit suivantes :

0	0	1	1
+0	+1	+0	+1
0	1	1	10

avec dans le cas 1+1 le résultat 0 et la présence d'une retenue égale à 1.

0 . 0 = 0, 0 . 1 = 0, 1 . 0 = 0, 1 . 1 = 1

* Un exemple d'addition :

```
  1 0 1 1 0 1
+ 1 1 0 1 1 0
 1 1 1          retenues
---------------
1 1 0 0 0 1 1        cela correspond en décimal à 45 + 54 = 99
```

Si l'on a à additionner plus que deux nombres, on fera d'abord l'addition des deux premiers, puis on additionnera le troisième, etc. On se ramène toujours à l'addition de deux nombres, afin d'éviter que la retenue ne puisse dépasser 1. Le pire cas est alors 1+1+1 = 11, avec le résultat 1 et la retenue 1. Cela s'applique aussi à la multiplication de deux nombres, puisqu'elle se ramène à des additions avec décalage.

[1] De droite à gauche : encore un effet de l'influence arabe, lorsque ce genre de calcul a traversé la Méditerranée du sud au nord, dès les années 1200, notamment grâce à Léonard de Pise, surnommé Fibonacci.

* Un exemple de multiplication, avec des additions de deux en deux en binaire :

```
      101101
    × 1101
  ---------
      101101
           0
    101101
  ---------
    11100001
  101101
  ---------
  1001001001
```

Cela correspond en décimal à 45 . 13 = 585

Notons qu'en binaire, si l'on multiplie *a* et *b*, il suffit de réécrire *a* chaque fois que l'on tombe sur un 1 dans *b* (en lisant de droite à gauche), et de mettre 0 sinon, tout en pratiquant un décalage d'un cran vers la gauche à chaque étage. La seule différence avec la multiplication dans le système décimal est que l'on s'impose de faire des additions de deux en deux, pour éviter des retenues trop grandes.

1-3) Conversions dans d'autres bases

Ce que nous avons fait en base 2, par analogie avec le système décimal (base 10) peut se généraliser à n'importe quelle base b. C'est encore plus simple pour des bases qui sont des puissances de 2, notamment le système octal (base 8) et le système hexadécimal (base 16). Dans ce cas on passe d'abord en binaire, puis :

* Si l'on veut une écriture en octal, on pratique un regroupement par blocs de longueur 3 à partir de la droite.

Par exemple : 26 (décimal) → 11010 (binaire) ou 011 010 → 32 en octal (soit 3 . 8 + 2)

* Si l'on veut une écriture en hexadécimal dont les chiffres sont par convention 0 1 2 3 4 5 6 7 8 9 A B C D E F, on regroupe les termes par blocs de longueur 4 à partir de la droite.

Par exemple : 26 (décimal) → 11010 (binaire) ou 0001 1010 → 1A en hexadécimal (soit 1 . 16 + 10)

1-4) Arithmétique en précision finie

Lorsqu'il travaille sur des nombres entiers, l'ordinateur ne fait aucune erreur (« Il est monstrueusement inhumain car il ne fait pas d'erreurs » disait une pub). En fait cela n'exact que si l'on ne dépasse pas les limites finies de la place mémoire dans lesquelles sont bloqués les nombres entiers. La somme de deux nombres entiers qui rentrent dans ces limites peut, elle, dépasser la limite, ce qui aboutit à un résultat faux. De même la propriété d'associativité peut être mise en défaut. Avec *a* et $b - c$ qui restent dans les limites, ainsi que $a + (b - c)$, il peut arriver que $a + b$ soit hors limites, donnant un résultat faux, et l'on n'aura plus $a + (b - c) = (a + b) - c$. Heureusement, dans les récents langages C, les entiers sont assimilés aux entiers longs, et l'on peut aller jusqu'à des nombres dépassant le milliard sans aucun risque.

2) Les nombres entiers relatifs, positifs ou négatifs, en base 2

On dit qu'il s'agit de nombres « signés » (ils ont le signe + ou le signe –). Voici comment l'on procède en binaire, sans utiliser les symboles + ou – , mais en remplaçant ce signe par un chiffre 0 ou 1. Tous les nombres en binaire de longueur *L*, correspondant aux nombres de 0 à $2^L - 1$ en décimal, au nombre de 2^L, vont maintenant représenter les nombres de -2^{L-1} à $+2^{L-1} - 1$. Par exemple pour $L = 3$:

0	000	devient +0
1	001	+1
2	010	+2
3	011	+3
4	100	- 4
5	101	- 3
6	110	- 2
7	111	- 1

Une telle correspondance est facile à comprendre : quand le bit le plus à gauche est 0, on retrouve les entiers positifs (ou nuls). Lorsque le bit le plus à gauche est 1, il s'agit d'un nombre négatif, et pour savoir qui est ce nombre, il suffit de l'écrire avec ses puissances de 2 en addition, sauf celle la plus à gauche mise en soustraction. Par exemple 110 devient $-2^2 + 2^1 = -4 + 2 = -2$.

Autre exemple : 1101001 donne $-2^6 + 2^5 + 2^3 + 2^0 = -64 + 32 + 8 + 1 = -64 + 41 = -23$.

Comment passer d'un nombre à son opposé ? On commence par prendre le complément du nombre bit à bit (en remplaçant les 0 par des 1 et vice versa) et en ajoutant 1. On dit que l'on prend le complément à deux, mais qu'importe ! [2] L'avantage de cette écriture de l'opposé, c'est que grâce à lui on peut faire des additions comme d'habitude, suivant la règle que l'on connaît déjà pour les nombres habituels : soustraire, c'est ajouter l'opposé. [3] Autrement dit, l'additionneur qui se trouve à l'intérieur de l'ordinateur sera aussi capable de faire des soustractions.

Exemples d'opposés :

1101101 (correspondant à – 64 + 45 = – 19) est d'abord complémenté, soit 0010010 puis on lui ajoute 1, ce qui donne 0010011, qui est justement 16 + 2 + 1 = 19, l'opposé de – 19.

---

[2] En fait il s'agit du complément à 2^n, *n* étant la longueur du nombre. Plus précisément, si *x* est le nombre, son « opposé » est $2^n - |x|$.

[3] Lorsque l'on travaille avec des nombres entiers en base 10, on sait bien que la soustraction est en fait une addition, puisque soustraire c'est additionner l'opposé, par exemple 10 – 7 = 10 + (– 7). Mais cela n'empêche pas que l'on doive savoir pratiquer des soustractions, et qu'elles sont différentes des additions. Voici comment on traite une soustraction, en en faisant terme à terme avec des retenues éventuelles :

```
  2389
- 1792
  597
```

Pour éviter d'imposer à l'ordinateur de savoir faire des soustractions par une méthode autre que celle des additions, on fabrique l'opposé par « complément à 2 », puis on fait une addition classique, et cela marche.

011011 (nombre positif 27 à cause de son 0 à gauche) devient par complément 100100 puis en ajoutant 1 : 100101, soit – 32 + 5 = –27.

Remarque : pour un nombre non signé on pouvait ajouter à sa gauche autant de 0 que l'on veut : 1101 = 001101. Pour les nombres signés, cela n'est pas possible de la même façon, puisque le bit le plus à gauche est le signe, par exemple 1101 est – 8 + 4 + 1 = – 3, tandis que 001101 est + 8 + 4 +1 = + 13. Mais on peut quand même écrire un nombre positif ou négatif avec la longueur que l'on désire. Le nombre +3 s'écrit 011 avec son zéro à gauche pour indiquer le +, mais il peut aussi s'écrire 0011 ou 00011, …. tandis que le nombre -3 s'écrit 101 (= -4 + 1) ou aussi 1101, 11101, 111101, etc.

Exercice : écrire – 5 sur 16 bits.

– 5 peut s'écrire au minimum sur quatre bits (il est entre – 8 et 7), soit 1101. Pour l'avoir sur 16 bits, il suffit de rajouter des 1 devant, et l'on obtient 1111111111111011, qui est aussi, comme on peut le vérifier : $- 2^{15} + 32763 = - 32768 + 32763 = -5$.

On est maintenant en mesure de faire des additions avec des nombres éventuellement négatifs, c'est-à-dire aussi des soustractions. Il convient que les deux nombres aient la même longueur, pour que les bits de signe puissent être l'un au-dessous de l'autre. Normalement cela se fait sur une longueur de 32 ou 64 bits, mais ici nous allons prendre des nombres plus courts. Faisons par exemple des additions avec des nombres signés sur quatre bits : ils sont compris entre – 8 et +7. Le résultat devra aussi être bloqué sur quatre bits. Si le résultat sort de la zone entre – 8 et +7, il sera faux et l'on dit alors qu'il y a dépassement de capacité (*overflow*). Pour tester cet *overflow*, sachant que la machine n'est pas capable de savoir si le résultat est bien compris entre – 8 et +7, il conviendra de trouver un autre moyen de le savoir, compréhensible par l'additionneur.

Il pourra aussi arriver que le calcul provoque une retenue égale à 1 sur un cinquième bit. Il faudra l'enlever, comme si elle n'existait pas, puisque l'on ne peut pas sortir des quatre bits. Mais alors deux choses sont possibles. Soit la suppression de ce bit de trop éventuel laisse le résultat exact, et c'est ce qui va arriver si le résultat reste dans la zone comprise entre – 8 et +7. Soit la suppression de ce bit provoque une erreur de calcul, et cela aura lieu parce que le résultat sort de la zone entre – 8 et + 7 (*overflow*). Vérifions tout cela sur des exemples.

* cas de deux nombres positifs

0011 3
0100 +4
0111 7 c'est juste

0011 3
0110 +6
1001 c'est faux, au lieu de +9 on trouve – 7. On est dans un cas de dépassement de capacité. Remarquons que la retenue au rang 3 (en allant de droite à gauche) est 1 et qu'elle est 0 au rang 4.

* cas de deux nombres négatifs

1101 –3
1100 –4
1 | 1001 –7 C'est juste à condition de supprimer le bit 1 de trop, en position 4

1101 –3
1001 –7
1 | 0110 C’est faux puisque l’on ne doit conserver que 4 bits dans le résultat, et que l’on trouve 6 au lieu de –10 (si l’on conservait le 5è bit, ce serait juste, mais on ne doit pas le garder !). Il y a dans ce cas dépassement de capacité. Remarquons que cela s’est produit lorsque la retenue au rang 3 est 0 et celle au rang 4 est 1

* cas avec un nombre positif et un nombre négatif. Avec ces deux nombres dans la zone permise, le résultat sera aussi dans cette zone. Il n’y a jamais de dépassement de capacité dans ce cas.

1101 –3
0001 +1
1110 –2 C’est juste

1101 –3
0101 +5
1 | 0010 +2 C’est juste à condition de supprimer le bit 1 de trop, en position 4.

Finalement, l’addition de deux nombres signés de longueur *n* est juste, en supprimant le bit 1 éventuel qui arrive en position *n* (en dehors de la zone de longueur *n*), à condition qu’il n’y ait pas de dépassement de capacité. Comment reconnaître s’il y a dépassement de capacité ? D’après ce qui précède, cela arrive dans deux cas exactement : les retenues en position *n* –1 et *n* sont 0 1 ou 1 0. Le test d’*overflow* se fait sur ces deux possibilités.

Exercices de calcul sur des nombres signés (ici de longueur 6), en comparant le résultat avec celui du calcul en base 10

110011
011001
001100 en supprimant le bit 1 de trop à gauche. C’est l’addition d’un nombre positif et d’un nombre négatif. Elle est forcément juste, cela correspond à – 13 + 25 = 12.

100101
010110
111011 C’est forcément juste, correspondant à – 27 + 22 = – 5

100110
101011
010001 On a supprimé le bit 1 de trop à gauche. Le résultat est faux : avec – 26 – 21 on trouve ici + 17 au lieu de – 47. C’était prévisible car on est dans un cas de dépassement de capacité, avec les retenues en position 6 et 5 qui sont 1 0.

Par contre si l’on avait travaillé avec des nombres de longueur 7, le résultat aurait été juste : 1100110 + 1101011 = 1010001 (on a supprimé le bit de trop), cela correspond bien à : – 26 – 21 = – 47. Il n’y a pas dépassement de capacité dans ce cas, les deux retenues de rang 7 et 6 sont 1 1.

001110

011001

100111 Ce résultat est faux, car les retenues en position 6 et 5 sont 0 1, ce qui signifie dépassement de capacité (14 + 25 = 39, et dans le cas présent nous avons trouvé – 25). Mais si l'on travaillait sur 7 bits, ce serait juste : 0001110 + 0011001 = 0100111. Il n'y a pas dépassement de capacité dans ce cas puisque les retenues aux rangs 7 et 6 sont 0 0.

3) La représentation des nombres à virgule

Nous allons maintenant travailler avec des nombres à virgule (que les anglo-saxons notent avec un point). Dans le système décimal, écrire un nombre comme 13,48 cela signifie :

$1 . 10^1 + 3 . 10^0 + 4 . 10^{-1} + 8 . 10^{-2}$, la seule nouveauté étant la présence de puissances négatives de 10 pour les nombres situés derrière la virgule. Il en est de même en binaire avec des puissances de 2. Par exemple 110,101 signifie $1 . 2^2 + 1 . 2^1 + 0 . 2^0 + 1 . 2^{-1} + 0 . 2^{-2} + 1 . 2^{-3}$. Il nous reste à apprendre la conversion base 10 ↔ base 2.

* Passage de la base 2 à la base 10, par exemple pour le nombre 110,101. Sa partie entière 110 donne 4 + 2 = 6, sa partie derrière la virgule est 1/2 + 1/8 = 0,5 + 0,125 = 0,625. Le nombre en base 10 s'écrit 6,625.

* Passage de la basse 10 à la base 2 :

Prenons l'exemple de 0,35. On veut l'écrire sous la forme 0,35 = a 1/2 + b 1/4 + c 1/8 + … avec a, b, c, …, égaux à 0 ou 1. Pour avoir a, on fait 0,35 . 2 = 0,7 = a + b 1/2 + c 1/4 + … qui est de la forme a + 0, …, ce qui donne a = 0. On en est à 0,7 = b 1/2 + c 1/4 + …, faisons comme précédemment 0,7 . 2 = 1,4 = b + c 1/2 + …, d'où b = 1, et il reste 0,4 = c 1/2 + … On remultiplie par 2 pour avoir c, et ainsi de suite.

Cela revient à faire la suite $x_{n+1} = 2\ x_n$ ramenée modulo 1, avec x_0 = 0,35, en dégageant à chaque fois la partie entière. On obtient ainsi la succession :

0,35 . 2 = 0,7 0
0,7 . 2 = 1,4 1
0,4 . 2 = 0, 8 0
0,8 . 2 = 1,6 1
0,6 . 2 = 1,2 1
0,2 . 2 = 0,4 0
0,4 . 2 = 0,8 0
… d'où 0,35 = 0, 0101100…

On remarque que la succession des chiffres derrière la virgule est finalement périodique avec derrière 01 le bloc 0110 qui se répète indéfiniment. Cela entraîne qu'avec un nombre toujours limité de bits, on n'aura jamais exactement 0,35 en binaire.

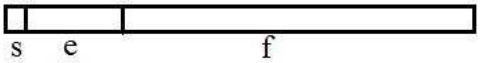
A son tour un nombre comme 17,35 (= 17 + 0,35) s'écrira 10001,0101100… Cela s'appelle une écriture en virgule fixe, par opposition à l'écriture des nombres en virgule flottante (*float*) utilisés par l'ordinateur. Virgule flottante signifie que l'on peut déplacer la virgule selon notre gré, comme par exemple dans le système décimal : $17,35 = 1735 . 10^{-2} = 1,735 . 10^1$, etc. à condition de faire intervenir des puissances de 10 en multiplication. C'est particulièrement intéressant pour les très grands nombres ou ceux très proches de 0, pour lesquels on choisit l'écriture scientifique, où l'on a des nombres de quelques unités multipliés par des puissances de 10, comme par exemple $1,72 . 10^{15}$ ou $2,34 . 10^{-12}$. Parmi toutes les écritures possibles, on préfère celle où la partie entière

du nombre (à gauche de la virgule) est à un chiffre, celui-ci étant compris entre 1 et 9. On fait de même en binaire.

Un nombre flottant en binaire est codé sur 32 bits, en simple précision, ou sur 64 bits en double précision. Il est fabriqué à partir de trois variables : *s* pour son signe, *e* pour son exposant et *f* pour la partie derrière la virgule. Voici la valeur du nombre lorsqu'on l'écrit sur 32 bits :

$(-1)^s \cdot 2^{e-127} \cdot (1 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \ldots + f_{23} \cdot 2^{-23})$

avec *s*, *e* et *f* qui occupent chacun une partie des 32 bits.



* *s* occupe un bit, valant 0 ou 1. Il va donner le signe positif ou négatif du nombre, puisqu'il intervient par $(-1)^s$, ce qui donne +1 ou – 1.
* *e* occupe 8 bits, soit 256 valeurs (de 0 à 255). Il intervient par 2^{e-127}, allant de 2^{-127} à 2^{128}.
* la partie derrière la virgule *f* est un mot de 23 bits, de la forme binaire $f_1\ f_2\ f_3\ \ldots\ f_{23}$. A partir de là on fabrique le nombre : $1 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \ldots + f_{23} \cdot 2^{-23}$. Ce nombre de partie entière 1 est de la forme 1, *f* et il est toujours situé dans l'intervalle [1 2[.

Exemple : le mot de 32 bits est 1 10000010 0011000….0 avec *s* = 1, *e* = 128 + 2 = 130, soit *e* – 127 = 3, et $f = 2^{-3} + 2^{-4} = 0{,}125 + 0{,}0625 = 0{,}1875$, d'où le nombre :
$-2^3 \cdot 1{,}1875 = -8 \cdot 1{,}1875 = -9{,}5$.

Quel est l'intérêt de cette écriture en virgule flottante ? Rappelons que pour les nombres entiers écrits avec *n* bits nous obtenions 2^n nombres s'étalant sur un intervalle de longueur 2^n. Cet intervalle est bien trop restreint pour les nombres à virgule, où grâce à l'écriture en *float* on arrive à une gamme de nombres s'étalant entre 2^{-127} et 2^{128} au signe près.

En effet,
* Si *e* = 0, *e* – 127 = – 127. En mettant des 0 derrière la virgule, on obtient le nombre $(-1)^s\ 2^{-127}$ que l'on assimile à 0.
* Si *e* = 255 , *e* – 127 = 128, et avec des 0 derrière la virgule le nombre est $(-1)^s\ 2^{128}$, proche de $-\infty$ ou $+\infty$.

Comme on l'a vu, le caractère fini de l'écriture des nombres peut conduire à des pertes minimes pour les nombres décimaux, et plus largement pour les nombres réels quelconques qui ont une infinité de termes derrière la virgule et dont on ne peut avoir qu'une approximation. Au-delà de sa définition initiale, pas très claire, évoquant le déplacement de la virgule, le mot *float* prend un sens bien plus évocateur : les nombres qui *flottent* sont des nombres qui peuvent ne pas être parfaitement exacts.

Exercice : convertir 3,14 en flottant 32 bits.
En virgule fixe, 3,14 devient 11,0010001111…, ou encore $2^1 \cdot 1{,}10010001111\ldots$
Sa représentation en flottant est 1 10000000 10010001111….