Chapitre 3

Représentation des informations – Codage des nombres

Objectif

L'objectif de ce chapitre est de découvrir comment sont représenté les informations en mémoire et plus particulièrement les nombres.

3.1 Introduction

Les ordinateurs, cartes d'acquisitions ou commande numériques sont des systèmes de traitement de l'information.

Cette information doit être codée sous une forme conventionnelle pour être comprise.

Un microprocesseur ne traite que des « bits » : des 0 ou des 1.

Les informations à représentées sont diverses et variés. On peut représenter :

- des nombres : entier ou réel, des complexes, des booléens
- des vecteurs
- des signaux, des courbes de $\mathbb{R}$ dans $\mathbb{R}$
- des images
- des tests
- des sons, des films
- ...

Les informations numériques sont nécessairement codés sur des BIT (binary digit) : 0 ou 1. Ces états 0 et 1 peuvent être obtenus :

- électriquement 0 à 1,5V
- mémoire MOS actif ou non (clé USB)
- disque dur : orientation identique ou opposé des aimants
- CD/DVD : trou ou absence de trou
- ...

Le microprocesseur travaille essentiellement avec la mémoire vive et ses registres. La mémoire est organisé comme un grand tableau de largeur fixe : 8 bits, 16 bits, 32 bits, 64 bits, 128 bits...

À chaque ligne du tableau correspond une adresse, et le bus de données vient lire ou écrire une ligne du tableau ou paquet de lignes. On parle d'alignement des données en mémoire.

Notion d'octet : un octet est un groupe de 8 bits, en anglais on parle de Byte.

- 1 ko = 1 kB = 1024 octets = 8192 bits
- 1 Mo = 1024 ko
- 1 Go = 1024 Mo
- 1 To = 1024 Go
- ...

3.2 Numération

3.2.1 Bases de numération

Une base de numération permet de décrire tout nombre de manière unique. Usuellement nous utilisons la base 10, mais d'autres bases pourraient être utilisés.

Un nombre X réel s'écrit dans la base B sous la forme suivante :

$$X = a_n B^N + a_{n-1}B^{n-1} + ... + a_0 B^0 + a_{-1}B^{-1} + ... + a_{m-1}B^{m-1} = \sum_{i=-m}^{n} a_i B^i$$

Remarque : Pour obtenir un nombre négatif, il suffit d'ajouter le signe $-$ devant l'expression.

La notation d'un nombre dans une base quelconque sera :
$X = (a_n a_{n-1}...a_0 a_{-1}...a_{-m})_B = \overline{a_n a_{n-1}...a_0 a_{-1}...a_{-m}}^B$

La partie $(a_n a_{n-1}...a_0)$ correspond à la partie entière du nombre et la partie $(a_{-1}...a_{-m})$ correspond à la partie décimale.

Bases classiques

Base 10	$a_i \in \{0, 1, ..., 9\}$	chiffres	Ex : 123,45
Base 2	$a_i \in \{0, 1\}$	bits	Ex : $-101{,}0011$
Base 16	$a_i \in \{0, 1, ..., 9, A, B, ...F\}$		Ex : 5A5,FC2

Compléter le tableau ci-contre.

On peut remarquer qu'un élément de la base 16 permet de représenter exactement 4 éléments de la base 2. C'est pourquoi pour des raisons de concision l'affichage de grand nombre d'éléments binaires sont affichés en base 16.

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4		
5		
6		
7		
8		

3.2.2 Passage d'une base B à la base 10

Ce sens de conversion est trivial, il suffit d'appliquer la définition.
$(101{,}0011)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 5{,}1875$
Convertir en base 10, les nombres suivant : $(11001{,}101)_2$ et $(F5G{,}8F)_{16}$.

3.2.3 Passage de la base 10 à la base B

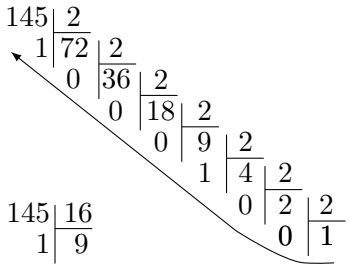
Détermination de la partie entière

Pour déterminer la partie entière dans la nouvelle base, la méthode consiste à effectuer une décomposition en réalisant des divisions euclidiennes successives par B.

On divise le nombre par B jusqu'à obtenir un résultat non divisible (dans les entiers) par B.

passage de 145 en base 2 et base 16.

D'après la figure ci-contre : $145 = (1001\,0001)_2 = (91)_{16}$.



Remarque : Il faut lire les résultats en remontant du bas vers le haut et en réécrivant de gauche à droite.

Remarque : En regroupant les termes en base binaire par groupe de 4 (en partant de la droite), on peut passer à la base 16 directement en prenant la correspondance du quartet en héxadécimal : $145 = (\underbrace{1001}_{(9)_{16}}\ \underbrace{0001}_{(1)_{16}})_2 = (91)_{16}$.

Passer le nombre 2013 en base 2 puis 16.

Détermination de la partie décimale

L'expression de la partie décimale dans une nouvelle base est obtenu par multiplication successive par B de la partie décimale du résultat précédent, l'unité obtenu correspond à un chiffre de la décomposition.

passage de 0,145 en base 2 et base 16.

$$\begin{aligned}
0{,}145 \times 2 &= \mathbf{0}{,}29 \\
0{,}29 \times 2 &= \mathbf{0}{,}58 \\
0{,}58 \times 2 &= \mathbf{1}{,}16 \\
0{,}16 \times 2 &= \mathbf{0}{,}32 \\
0{,}32 \times 2 &= \mathbf{0}{,}64 \\
0{,}64 \times 2 &= \mathbf{1}{,}28 \\
&\vdots \\
0{,}145 &\approx (0{,}0010\,01)_2
\end{aligned}$$

$$\begin{aligned}
0{,}145 \times 16 &= \mathbf{2}{,}32 \\
0{,}32 \times 16 &= \mathbf{5}{,}12 \\
0{,}12 \times 16 &= \mathbf{1}{,}92 \\
0{,}92 \times 16 &= \mathbf{14}{,}72 \\
0{,}72 \times 16 &= \mathbf{11}{,}52 \\
0{,}52 \times 16 &= \mathbf{8}{,}32 \\
&\vdots \\
0{,}145 &\approx (0{,}251EB8)_{16}
\end{aligned}$$

Remarque : La décomposition de la partie décimale peut conduire à une suite infinie de termes.

Déterminer la partie décimale en base 2 et 16 de 0,6875.

3.2.4 Opérations

Les opérations mathématiques $+$, $-$, $\times$ et $/$ fonctionnent exactement de la même manière dans les bases B qu'en base 10.

On peut utiliser les mêmes techniques de calcul.

Additionnons puis multiplier en binaire les nombres $a = (10)_{10} = (1010)_2$ avec $b = (7)_{10} = (111)_2$

$$\begin{array}{rccccc}
 & 1 & 0 & 1 & 0 & \\
+ & & 1 & 1 & 1 & \\
\hline
= & 1 & 0 & 0 & 0 & 1
\end{array}$$

On retrouve bien un résultat qui vaut $(10001)_2 = (17)_{10}$.

$$\begin{array}{rccccccc}
 & & & 1 & 0 & 1 & 0 & \\
\times & & & & 1 & 1 & 1 & \\
\hline
 & & & 1 & 0 & 1 & 0 & \\
+ & & 1 & 0 & 1 & 0 & & \\
+ & 1 & 0 & 1 & 0 & & & \\
\hline
= & 1 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

On retrouve bien un résultat qui vaut $(1000110)_2 = (70)_{10}$.

3.3 Représentation des nombres entiers en machine.

Le stockage des données dans la machine est nécessairement sous forme de 0 et de 1. Dans une représentation binaire d'un mot de plusieurs bits, rien n'indique comme il doit être lu. Il est nécessaire de déclarer un type au mot binaire pour l'interpréter correctement.

Un mot binaire est stocké sur un ou plusieurs octets (bytes en anglais), donc un multiple de 8 bits.

3.3.1 Représentation des entiers naturels $\mathbb{N}$ – type entier non signé (unsigned int)

La représentation des entiers peut être est réalisée sur 1, 2, 4 ou 8 octets, soit 8, 16, 32 ou 64 bits, voir plus.

Pour représenter les entiers positifs, il suffit d'utiliser le codage en binaire vu précédemment.

Si on note N ne nombre de bits de la représentation, il sera possible d'écrire 2^N valeurs.

Ainsi sur :

- 8 bits, il est possible de compter de 0 jusqu'à $2^8 - 1 = 255$ (la première valeur étant 0)
- 16 bits, il est possible de compter de 0 jusqu'à $2^{16} - 1 = 65\,535$
- 32 bits, il est possible de compter de 0 jusqu'à $2^{32} - 1 = 4\,294\,967\,295$
- ...

3.3.2 Représentation des entiers relatifs $\mathbb{Z}$ – type entier signé (int)

Le problème pour représenter les nombres négatifs est qu'il faut faire comprendre à la machine le symbole $-$.

Une solution élémentaire consiste à prendre un bit (généralement le bit de poids fort, le plus à gauche) pour déclarer le signe : 0 le nombre est positif, 1 le nombre est négatifs et le reste des bits permet de représenter la valeur absolue du nombre.

Cette solution présente deux inconvénients : le 0 est défini deux fois et les opérations entre deux nombres ne sont plus élémentaires.

On utilise le complément à 2 pour représenter $a \in \mathbb{Z}$:

- 1 bit détermine le signe, le bit de poids fort à gauche
- les autres bits (7, 15, 31 et 63 en pratique) détermine la valeur :
  - pour un nombre positif, on utilise le codage binaire naturel
  - pour un nombre négatif, on utilise le complément à 2, on code en binaire naturel le nombre $2^N - |a|$

Calcul du complément à 2 d'un nombre : en pratique le calcul pour un nombre négatif est obtenu en prenant le complément du la représentation du nombre positif et en lui ajoutant 1 (le dépassement éventuelle de capacité du à la retenue de l'addition est ignoré).

Exemple : On a $110 = (01101110)_2$

Donc $-110 = (10010001)_2 + 1 = (10010010)_2$

Vérifions que $110 - 110 = 0$

$$\begin{array}{r} 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0 \\ +\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0 \\ \hline =\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

Il faut noter que $1+1$ en binaire fait 0 et on retient 1. La dernière retenue du calcul est « oubliée ».

Un autre méthode pour déterminer le complètement à 2, consiste à garder tout les 0 à partir de la droite en conservant le premier 1 rencontré. Puis tout les autres bits sont inversés :

- on a $110 = (01101110)_2$
- on conserve la partie droite jusqu'au premier 1 : (01101**10**)
- on prend le complètement des autres bits : (**10010**10)
- d'où la représentation de -110 en complément à 2 : $(10010010)_2$

Déterminer la représentation de -145 en complément à 2.

Avec cette représentation, sur :

- 8 bits, il est possible de compter de -128 jusqu'à 127 (la première valeur étant 0)
- 16 bits, il est possible de compter de $-32\,768$ jusqu'à $32\,767$
- 32 bits, il est possible de compter de $2\,147\,483\,647$ jusqu'à $2\,147\,483\,648$
- ...

3.3.3 Particularité en Python

On vient de voir que suivant le nombre d'octet utiliser pour représenter un entier, on a une limitation du nombre de valeur représentable. Cela signifie que si on ne fait pas attention le résultat d'une opération peut dépasser la capacité de représentation et une erreur de type *overflow* apparaitra.

Il faut noter que le type `int` de Python est un type à représentation infinie. Il n'y a aucune limitation, Python modifiera le nombre de bits nécessaire au stockage à la volée.

```
>>> entier = 45
>>> type(entier)
<class 'int'>
>>> entier.bit_length()
6
>>> entier=1268748613210745746414 12
>>> entier.bit_length()
77
```

La fonction `type` renvoie le type de la variable.

La méthode `bit_length()` renvoie le nombre de bit utilisé pour représenter l'entier.

3.3.4 Opérations

Comme on l'a vu dans la partie précédente, les opérations mathématiques sont réalisés comme on le ferait à la main :

- addition : on somme les bits un par un en prenant en compte les retenus
- soustraction : on réalise un complément à 2 du nombre à soustraire pour obtenir la représentation de son opposé, puis on fait une addition
- multiplication : même méthode qu'à la main. En pratique, on utilise les décalages des bits...

3.4 Représentation des nombres réels en machine.

Nous avons vu comment écrire un réel dans n'importe quelle base et notamment la base 2. Le problème est que cette écriture ne peut être utilisé dans un ordinateur en raison de la présence la virgule.

La notation scientifique pour les nombres est adoptée depuis longtemps. Le principe du codage des réels en mémoire est basé sur cette représentation.

Tout nombre x sera représenté sous la forme $\varepsilon m \times B^e$ avec

- m la mantisse
- B la valeur de la base, ici $B = 2$
- e l'exposant
- ε le signe

3.4.1 Représentation des réels NORME IEEE 754

L'IEEE 754 est une norme de représentation des réels et plus précisément une norme de stockage du signe, de la mantisse et de l'exposant. Elle définit également le 0, l'infini et les NaN (Not a Number !).

Forme générale

Le mot binaire représentant le réel comportera :

- 1 bit, celui de poids fort pour représenter le signe
- n_1 bits pour représenter l'exposant
- n_2 bits pour représenter la mantisse

La première remarque à faire est que l'exposant pouvant être négatif, il faut prévoir le signe. Le choix aurait pu être fait d'utiliser une notation de type complément à 2 mais c'est un décalage qui est réalisé. La valeur du décalage est de $decalage = 2^{n_1-1} - 1$.

L'interprétation d'un nombre est donc : $\boxed{x = \text{signe} \times \text{mantisse} \times 2^{\text{exposant}-\text{decalage}}}$

Particularités :

- le nombre est positif si le bit de signe est nul et négatif sinon,
- si tous les bits de l'exposant sont à 1, alors le nombre est :
  - l'infini si la mantisse est nulle,
  - Not a Number sinon.
- si tous les bits de l'exposant sont à 0, alors le nombre est :
  - nul si la mantisse est nul,
  - non normalisé sinon (nous ne détaillerons pas ce cas).

Format simple précision sur 32 bits

Dans ce format, 1 bit est utilisé pour le signe, 8 bits pour l'exposant et 23+1 bits pour la mantisse.

Le +1 bit est un bit masqué. En effet, la mantisse est un nombre décimal compris entre 1 et 2 (< 2), donc le premier chiffre est nécessairement 1. Il n'est donc pas représenté.

On ne code que la mantisse réduite m' telle que $m = 1 + m'$.

Le décalage est égale à $2^{8-1} - 1 = 127$.

Représentation de 0,145

1. écriture en base binaire du nombre : nous avons vu que $0{,}145 \approx (0{,}001001)_2$,
2. mise sous forme scientifique : $0{,}145 = 1.001 \times 2^{-3}$
3. on code le bit signe : 0
4. on code l'exposant décalé sur 8 bits : $127 - 3 = 124 = (01111100)_2$
5. la mantisse réduite est directement la partie décimale de l'écriture en base 2, complété par des 0 pour obtenir les 23 bits : $(00100000000000000000000)_2$

On en déduit que la représentation de $0{,}145 \approx (00111110\,00010000\,00000000\,00000000)_2$.

Remarque : On peut observer que la limitation se situe à 23 chiffres significatifs sur la mantisse réduite. Si la décomposition exacte nécessite plus de chiffres alors le nombre représenté en mémoire sera approché.

Le plus petit nombre représentable est $\pm 2^{-126} = \pm 1{,}175 \times 10^{-38}$ et le plus grand nombre représentable est $\pm 1.11111111111111111111111 \times 2^{127} \approx \pm 2^{128} = \pm 3{,}4 \times 10^{38}$.

Donner la représentation en 32 bits du nombre 0,6875.

Format double précision sur 64 bits

Dans ce format, 1 bit est utilisé pour le signe, 11 bits pour l'exposant et 52+1 bits pour la mantisse.

Le décalage est égale à $2^{11-1} - 1 = 1023$.

Le plus petit nombre représentable est $\pm 2^{-1022} = \pm 2{,}2 \times 10^{-308}$ et le plus grand nombre représentable est $\pm 1.11111111111111111111111... \times 2^{1023} \approx \pm 2^{1024} = \pm 1{,}8 \times 10^{308}$.

Remarque : En fonction de la précision du calcul souhaité, on choisira entre la simple precision ou la double precision. La conséquence de ce choix est un temps de calcul plus long en double précision (car plus de bits à manipuler) et un espace de stockage également plus important.

Représentation en Python

Python utilise par défaut une représentation double precision nommé « float ».

```
>>> reel = 1.25
>>> type(reel)
<class 'float'>
```

Il existe un type `decimal` à précision infinie en utilisant le module `Decimal`.

3.4.2 Opérations

Pour réaliser les opérations sur la représentation des nombres réelles, il faut :
- pour une multiplication/division :
  - réaliser l'opération sur les mantisses et définir la nouvelle mantisse car le résultat peut être supérieur à 2 ou inférieur à 1.
  - additionner pour la multiplication ou soustraire pour la division les exposants
- pour une addition/soustraction :
  - effectuer l'opération sur les mantisses en pensant à décaler les représentations du nombre nécessaire de bits pour prendre en compte les exposants différents
  - modifier le résultat pour être sous la bonne forme
- vérifier le non dépassement de capacité

Les opérations directes sur les nombres réels sont très difficiles à illustrer.

3.5 Représentation d'autres informations

3.5.1 Caractères

Ils ne peuvent pas être décrit sur une base de numération. Il faut donc définir un code binaire associé à chaque caractère.

Il existe plusieurs codages :
- ASCII (1961) qui définit 128 caractères et est codé sur 7 bits (voir FIGURE 3.5.1). Les ordinateurs travaillant sur des octets, des versions étendus ont été proposées.
- ISO 8859 (1986) définit 15 jeux de 256 caractères codé sur 1 octet. Ce sont des extensions du code ASCII pour prendre en compte les caractères spéciaux des différentes langues
- Unicode (1991) et UTF8 (1993) définissent environ 245 000 mais peuvent en contenir plus d'un million. Il n'y a plus de relation directe entre le codage d'un caractère et son espace mémoire. Suivant la version de la norme choisie le nombre de bits nécessaire à la représentation peut varier fortement. Notons par exemple que pour l'UTF8 il faut au plus 4 octets pour représenter les caractères.

[Code ASCII – 7 bits] [Enregistrement voix humaine : 200 points toutes les 5 ms environ]

FIGURE 3.1: Représentation des caractères et des signaux temporels

3.5.2 Signaux temporels

Les signaux temporels ne peuvent être représentés que de façon discrète, c'est-à-dire une suite de point de coordonnées (x, y) suffisamment proche pour bien représenter le signal (voir FIGURE 3.5.1).

Chaque point peut être représenté sous la forme d'un entier ou d'un flottant et stocké dans un tableau/une liste.

3.5.3 Images matricielles

Les images matricielles sont stockées et affichées sous forme d'un tableau de pixel (« picture element ») colorés, on parle de carte de point (« bitmap »).

Le codage de chaque pixel peut être réalisé en :

– noir et blanc : 1 bit par pixel
– niveau de gris : 4, 8 ou 16 bits par pixel
– couleur RGB : on décompose la couleur en sa composante rouge, verte et bleue et chaque composante est stocké sur 4, 8 ou 16 bits.
– une palette interne et chaque pixel est associé au numéro de la couleur dans la palette

L'image est ensuite généralement compressée et donne les différents de fichiers connus :
– sans compression : bmp, raw, tiff
– compression sans perte : png, tiff
– compression avec perte : jpg, tiff

[Couleur (évidemment sur papier !)] [Niveau de gris] [Noir et blanc]

FIGURE 3.2: Différentes palettes de couleurs pour une même image.