Chapitre III

Révisions 3 : Représentation des nombres en machine

Sommaire

1 Représentation des entiers naturels

1.1 Pour l'homme

a - La base 10

La base de représentation décimale **ou base 10** des nombres fut probablement adoptée dans le cours de la préhistoire par les peuples *indo-européens*. Elle découle très naturellement du nombre total de doigts présents sur les mains humaines, les hommes s'en servant pour faire du dénombrement élémentaire.

Le principe de décomposition d'un entier en base 10 est d'écrire le nombre en une somme d'autant de puissances de 10 que nécessaire. Par exemple :

$$6753 = 3.10^0 + 5.10^1 + 7.10^2 + 6.10^3$$

En généralisant à tout nombre entier naturel n, cela donne :

$$n = \left(\underset{\substack{\uparrow \\ \text{poids le} \\ \text{plus fort}}}{d_{n_d}} \quad \ldots d_1 \quad \underset{\substack{\uparrow \\ \text{poids} \\ \text{le plus} \\ \text{faible}}}{d_0} \right)_{10} = \sum_{k=0}^{k=n_d} d_k \cdot 10^k \qquad \text{avec } d_k \in \{0, 1, 2, \ldots, 9\}$$

EXERCICE N°1: *Interpréter le fonctionnement du script python suivant :*

Listing III.1 – Que fais–je ?

```
def Chiffre (n) :
    """n est un entier naturel donné par son écriture décimale"""
    c=[]
    while n!=0 :
        c.append(n%10)
        n//=10
    c.reverse() # inverse l'ordre des éléments d'une liste
    return c
print Chiffre (4587)
```

b - Les autres bases

On peut représenter un nombre entier dans n'importe quelle base selon ce même principe. Très généralement, un nombre en base k peut s'écrire :

$$(d_{n_k} \ldots, d_1, d_0)_k = d_0 \cdot k^0 + d_1 \cdot k^1 + \ldots + d_{n_k} \cdot k^{n_k}$$

REMARQUE : la conversion en base décimale est immédiate en réalisant à la main la somme décomposée précédente. Voyons cela sur un exemple, le nombre suivant en base 5

$$(204003)_5 = 3 \times 5^0 + 0 \times 5^1 + 0 \times 5^2 + 4 \times 5^3 + 0 \times 5^4 + 2 \times 5^5 = 6753$$

NB : les TD seront l'occasion de revenir sur les algorithmes simples de transformation des nombres entre bases.

1.2 ...et pour la machine : la base 2 obligatoire ! ! !

a - Principe

Nous avons déjà évoqué dans le cours Révisions 1 la limitation des ordinateurs à la manipulation de **deux états de potentiels** haut et bas. Par exemple, les potentiels des "pattes" d'une mémoires DDR peuvent prendre uniquement des valeurs $\{0\ V\ ;\ 1,5\ V\}$. S'agissant d'une base à deux états, nous choisissons d'écrire ces derniers **0** et **1**. Ce sont les chiffres de la base **2** ou base binaire.

Les puces mémoires/processeurs sont constituées de pattes, chacune pouvant être au chiffre **0** ou **1**. On appelle cela un **bit** pour l'anglais **BI**nary digi**T**. Combien d'états et donc de valeurs peut recevoir une puce constituée de n pattes ?

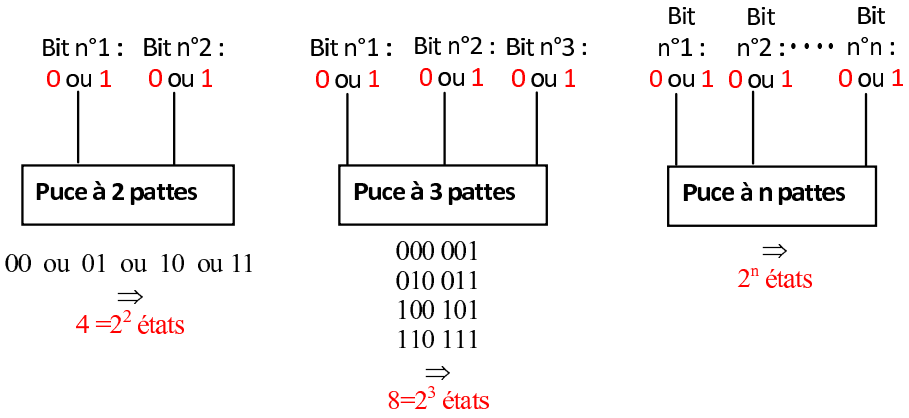

FIGURE III.1 – Etats possibles d'une puce à n pattes

Ainsi donc, il nous faudra coder les nombres en base 2 à l'aide des deux chiffres $\{0,1\}$. Selon le même principe que précédemment, un entier naturel n s'écrit donc :

$$n = (d_{n_b} \dots d_1 d_0)_2 = \sum_{k=0}^{k=n_b} d_k \cdot 2^k \qquad \text{avec } d_k \in \{0,1\}$$

REMARQUE - (1.2) - 1: IMPORTANT : Les circuits informatiques telles que les puces mémoires ou les processeurs regroupent souvent les nombres binaires par lots de 8 bits, on appelle cela **un octet**. Les nombres informatiques sont ainsi exprimés sur 1, 2, 4, ou encore 8 octets, soit respectivement 8 bits, 16 bits, 32 bits ou 64 bits. Dans la structure des mémoires actuelles des ordinateurs courants, la plus unité adressable que l'on appelle **Bytes** en anglais, est justement codée sur 8 bits soit 1 octet (ce n'était par exemple pas le cas dans les années 70 ou le byte pouvait correspondre à 6,7,8 ou 9 bits).

Ces différentes "longueurs" de "mots" offrent les possibilités de codage d'entiers suivantes :

Profondeur en octets	intervalle de codage entiers base 2			intervalle de codage entiers base 10		
1 (8 *bits*)	00000000	$\longrightarrow$	11111111	0	$\longrightarrow$	255
2 (16 *bits*)	00000000 00000000	$\longrightarrow$	11111111 11111111	0	$\longrightarrow$	65535
4 (32 *bits*)	00000000 00000000 00000000 00000000	$\longrightarrow$	11111111 11111111 11111111 11111111	0	$\longrightarrow$	4294967295

A RETENIR : Une profondeur d'écriture binaire de N *bits* permet de représenter les nombres entiers en base 10 de l'intervalle $\left[0..2^N-1\right]$

b - Conversion

QUESTION : comment représenter en base 2 format 16 bits un nombre entier naturel donné en base 10 ?

RÉPONSE : il existe essentiellement 2 méthodes : la méthode dite «par soustraction» et la méthode dite «par division» (entière). On donne ici le principe de la seconde :

On souhaite décomposer le nombre n_{10} en base 2. Pour cela :

- On réalise la division entière de n_{10} par 2. Si le quotient n'est pas nul, la valeur du reste (0 ou 1) correspond au 1^{er} bit : 0 ou 1.
- On réalise ensuite la division entière du précédent quotient par 2. De même si le quotient n'est pas nul, la valeur du reste (0 ou 1) correspond au 2^{sd} bit.
- Ainsi de suite jusqu'au bit de poids le plus fort.

EXEMPLE SIMPLE : cherchons à décomposer 24_{10} en base 2 par division entières successives :

$$\begin{aligned}
24 &= (12\times 2)+0\\
&= ((6\times 2+0)\times 2)+0\\
&= (((3\times 2+0)\times 2+0)\times 2)+0\\
&= ((((1\times 2+1)\times 2+0)\times 2+0)\times 2)+0\\
&= (((((0\times 2+1)\times 2+1)\times 2+0)\times 2+0)\times 2)+0\\
&= \mathbf{0}\times 2^5+\mathbf{1}\times 2^4+\mathbf{1}\times 2^3+\mathbf{0}\times 2^2+\mathbf{0}\times 2^1+\mathbf{0}\times 10^0
\end{aligned}$$

Ainsi la représentation binaire de $(24)_{10}$ est donc :

$$(1100)_2$$

Par exemple, on souhaite coder en binaire le nombre entier 1515 (Date de la bataille de Marignan [1] !)

1. Tout le monde connait cette célèbre date de bataille, mais peu de gens savent qu'elle opposa François 1^{er} et les vénitiens aux mercenaires Suisses à **Marignan**, aujourd'hui Melegnano en Italie.

1515	2									
1	757	2								
	1	378	2							
		0	189	2						
			1	94	2					
				0	47	2				
					1	23	2			
						1	11	2		
							1	5	2	
								1	2	2
									0	1	2
										1	0

La représentation binaire de $(1515)_{10}$ est donc :

$$(10111101011)_2$$

2 Représentation des entiers relatifs

2.1 Notation en complément à deux

Un première idée d'écriture des nombres entiers relatifs est de représenter un nombre entier naturel sur une certaine profondeur,*par exemple 7 bits ou 15 bits*, et de réserver un bit pour le signe, pour obtenir dans notre exemple *8 bits ou 16 bits*.

Pour un codage sur N bits, le bit réservé pour le signe S est d_{N-1} avec :

$$S = (-1)^{d_{N-1}} \Longrightarrow \begin{cases} S = +1 \ (n \text{ positif}) \text{ si } d_{N-1} = 0 \\ S = -1 \ (n \text{ négatif}) \text{ si } d_{N-1} = 1 \end{cases}$$

La représentation binaire de n est donc :

$$\boxed{n = (-1)^{d_{N-1}} \sum_{k=0}^{N-2} d_{n_b} 2^k}$$

Afin d'être distingué d'un entier naturel, un entier relatif est représenté avec le bit de signe est **souligné** :

$$(\underline{1}0010011)_2 = (-19)_{10}$$

Cette convention de représentation présente cependant deux inconvénients majeurs (par exemple en 8 bits) :
- présence de deux zéros : $(\underline{0}0000000)_2 = (0)_{10}$ et $(\underline{1}0000000)_2 = (-0)_{10}$
- la somme en binaire ne fonctionne plus, avec par exemple :

$$\underbrace{(\underline{0}0000011)_2}_{=(+3)_{10}} + \underbrace{(\underline{1}0000010)_2}_{=(-2)_{10}} = \underbrace{(\underline{1}0000101)_2}_{=(-5)_{10} \neq 1}$$

Conclusion : l'objectif d'une machine étant essentiellement de faire des calculs non limités au entiers naturels, et ce rapidement, cette convention de représentation ne peut donc convenir.

Idée : On garde la même notation pour les entiers positifs, et on adopte la notation dite en **complément à deux** pour les entiers négatifs. Le principe est le suivant, on veut coder un entier relatif négatif n :

- On garde le N$^{\text{ième}}$ bit d_{N-1} pour le signe et on prend les $N-1$ bits restants qui représentent la valeur absolue du nombre
- On inverse les $N-1$ bits de la valeur absolue du nombre (opération booléenne *NON*). Cette opération s'appelle le complément à 1.
- On ajoute le $N^{\text{ième}}$ bit de signe
- On ajoute 1 à ce résultat. c'est le complément à 2

Exemple :

Codons par exemple $n = (-7)_{10}$ avec cette méthode sur 8 bits :

- Valeur absolue $\longrightarrow |n| = 7 = (\_0000111)_2$
- Complément à 1 $\longrightarrow \overline{|n|} = (\_1111000)_2$
- Bit de signe (ici 1 à la place de l'*underscore*) $\longrightarrow (11111000)_2$
- Complément à 2 $\longrightarrow (11111001)_2$

Remarque : les défauts de la précédente méthode de codage sont éliminés :

- codage de zéro (toujours sur 8 bits) $\begin{cases} (+0)_{10} = (00000000)_2 \overset{\text{inv 7 bits}}{\longrightarrow} (01111111)_2 \overset{+1}{\longrightarrow} 10000000_2 = (-0)_{10} \\ (-0)_{10} = (10000000)_2 \overset{\text{inv 7 bits}}{\longrightarrow} (11111111)_2 \overset{+1}{\longrightarrow} 00000000_2 = (+0)_{10} \end{cases}$
- $(+3)_{10} + (-2)_{10} = (1)_{10}$ soit en binaire avec $(+3)_{10} = (00000011)_2$ et $(-2)_{10} = (11111110)_2$

$$\begin{array}{rl} & (00000011)_2 \\ + & (11111110)_2 \\ = & \underbrace{(00000001)_2}_{=(+1)_{10}} \end{array}$$

Conclusion : ainsi donc pour un nombre binaire codés sur N bits, on peut coder :

$$\begin{array}{lll} 2^{N-1} \text{ nombres positifs} & \longrightarrow & n \in [0, 2^{N-1}-1] \\ 2^{N-1} \text{ nombres négatifs} & \longrightarrow & n \in [-2^{N-1}, -1] \end{array}$$

soit un intervalle totale de représentation : $\boxed{n \in [-2^{N-1}; 2^{N-1}-1]}$

et un total de $\boxed{2^{N-1} + \underbrace{1}_{\text{pour 0}} + 2^{N-1} - 1 = 2^N \text{ valeurs}}$

Donc par exemple pour un codage sur 8 bits, on peut représenter l'intervalle d'entiers relatifs : $[-128; +127]$ soit 256 valeurs comme attendu. Plus généralement :

Profondeur (octets	intervalle de codage entiers base 2			intervalle de codage entiers base 10		
1 (8 *bits*)	10000000	$\longrightarrow$	01111111	-128	$\longrightarrow$	127
2 (16 *bits*)	10000000 00000000	$\longrightarrow$	01111111 11111111	-32768	$\longrightarrow$	32767
4 (32 *bits*)	10000000 00000000 00000000 00000000	$\longrightarrow$	01111111 11111111 11111111 11111111	-2147483648	$\longrightarrow$	2147483647

2.2 Le problème du dépassement de capacité - cas particulier de Python

QUESTION (LÉGITIME !!!) : que se passe-t-il lorsque l'on demande à une machine (via un langage) de traiter des nombres plus grand que le dimensionnement prévu pour les variables ?

RÉPONSE :
Certains langages nécessitent une déclaration préalable du type de chacune des variables employées en préambule de tout programme. C'est par exemple le cas du Pascal ou du VisualBasic, ou encore Fortran ; on appelle cela le **typage statique**. En particulier, la profondeur des entiers doit être précisée. Toute opération de dépassement d'un entier sera signalée par une erreur de capacité de type *overflow* (Pascal), ou conduira à une erreur de calcul, parfois sans aucun signalement ("écrasement" mémoire sans précaution !!!) (Fortran 77).

EXERCICE N°2: *Tester la "profondeur" mémoire de votre calculatrice en insérant au clavier le plus grand nombre binaire entier positif*[2]

Dans le cas du langage Python, aucune limite de profondeur n'est fixée !!! Lorsque l'on tente de manipuler un nombre plus grand que la profondeur maximale prévu (32 ou 64 bits suivant les machines et les systèmes d'exploitation), Python découpe l'entier en lots de 15 bits et place les fragments dans un tableau d'entier chacun de largeur 16 bits. Ce tableau contient donc le codage du nombre en base 2^15 ; on ajoute également dans ce tableau un entier codé sur 16 bits indiquant le nombre de fragments et dont le signe est celui du nombre codé.

Lorsque Python passe dans ce type "propriétaire" de représentation des nombres, il le signale par l'écriture d'un suffixe "L" pour *Large Integer* à la suite du nombre. Par exemple en se plaçant à la limite de codage des entiers relatifs en 32 bits, soit :

$$(01111111111111111111111111111111)_2 = (2147483647)_{10}$$

et en ajoutant 1 :

```
>>> 2147483647+1
2147483648L
>>>
```

2. sur les modèles TI nspire CX/CX CAS, il suffit par exemple d'inscrire "0b" comme préfixe à toute écriture de nombre binaire. C'est le cas sur bon nombre de calculatrices.

REMARQUE - (2.2) - 2: La notation des entiers négatifs en complément à 2 **nécessite de connaître la profondeur en bits disponible sur la machine**, puisque l'on devra inscrire obligatoirement 1 sur le bit de signe qui se trouve **complètement à gauche**. Pour éviter ce souci, les calculatrices permettent de placer un signe − devant la valeur absolue d'une entier relatif binaire pour coder le nombre entier négatif correspondant.

3 Représentation des réels

3.1 La notation scientifique décimale

a - Nécessité des "flottants" en machine

Dans les problèmes de sciences en général, nous sommes souvent amenés à exploiter **des nombres à virgule**. Par exemple, lors de la mesure d'une épaisseur de lame de verre à l'aide d'un Palmer[3], une valeur peut-être $e = 1,12\ mm$.

QUESTION : **Si nous souhaitons exploiter cette mesure avec un ordinateur, doit-il absolument comprendre ce qu'est un nombre à virgule ?**

RÉPONSE : Non !!! En choisissant comme unité d'écriture le μm, le résultat devient $h = 1120\ \mu m$, c'est à dire un entier que l'ordinateur comprend !!!

Cependant, nous savons que l'intervalle des nombres entiers représentables en machine est limité par la profondeur, c'est à dire le nombre de bits choisis. Ainsi, lorsque nous effectuons un calcul en choisissant de "translater" la virgule complètement à droite pour touts les opérandes afin d'obtenir des entiers (en changeant l'unité), nous obtenons des nombres plus grands susceptibles d'entrainer un "*overflow*".

Imaginons par exemple que nous codions les entiers sur 32 bits dans une machine et qu'à l'aide de celle-ci nous souhaitions calculer le volume de la lame de verre ; un objectif plutôt raisonnable pour les ordinateurs modernes et pourtant !!! Cela donne :

$$\begin{bmatrix} \text{epaisseur} & e = 1120\ \mu m \\ \text{largeur} & l = 2,5\ cm = 25000\ mum \\ \text{longueur} & L = 6\ cm = 60000\ mum \end{bmatrix} \implies V(\mu m^3) = 1120{*}25000{*}60000 = 1680000000000 \mu m^3 > \underbrace{2147483647}_{\text{limite 32 bits}}$$

Ainsi, ce simple calcul provoque un dépassement !!!

SOLUTION : on doit pouvoir étendre la fourchette des nombres représentables. Ceci est possible en définissant un type de nombres dits **à virgule flottante** ou type *float* en Python. Le principe est de pouvoir déplacer à loisir la position de la virgule en indiquant un exposant de puissance de 10 ; c'est la classique notation scientifique décimale dans laquelle on distingue d'une part les chiffres significatifs, et d'autre part les ordres de grandeurs à travers la puissance de 10 :
Par exemple : $c = 299792458\ m.s^{-1} = 3,0.10^8\ m.s^{-1}$ et $\mu_0 = 4\pi.10^{-7}\ H.m^{-1}$

Le calcul précédent devient possible si nous souhaitons toujours garder le résultat en μm^3 :

$$\begin{bmatrix} \text{epaisseur} & e = 1120\ \mu m = 112.10^1 \mu m \\ \text{largeur} & l = 2,5\ cm = 25000\ mum = 25.10^3 \mu m \\ \text{longueur} & L = 6\ cm = 60000\ mum = 6.10^4 \mu m \end{bmatrix} \implies V(\mu m^3) = 112 * 25 * 6 = \underbrace{16800}_{<2147483647}\ .10^8 \mu m^3$$

On évite ainsi les dépassements.

3. cf TP

b - Décomposition en notation scientifique décimale

En physique par exemple, il est fréquent d'employer la notation scientifique décimale. Par exemple, elle est bien commode pour écrire certaines constantes :

$$\begin{cases} c = 2,99792458.10^{8}\ m.s^{-1} \\ k = 1,3806488.10^{-23}\ J.K^{-1} \\ \mathcal{N}_a = 6,022142.10^{23} mol^{-1} \end{cases}$$

Un réel x, dont le nombre de décimales est fini ou non, peut s'écrire sous la forme suivante :

$$\boxed{x = (-1)^S \times 10^E \times \left(d_0 + d_1.10^{-1} + d_2.10^{-2} + \ldots + d_i.10^{-i} + \ldots\right)} \text{ avec } \begin{cases} S = 0,1 \text{ qui fixe le signe de x} \\ E \text{ entier relatif (exposant)} \\ d_i \in \{0,1,2,\ldots,9\} \textbf{ et on fixe } d_0 \neq 0 \end{cases}$$

Par exemple on peut écrire la constante de Boltzmann k selon cette décomposition :

$$k = (-1)^0 \times 10^{-38} \left(1 + 3.10^{-1} + 8.10^{-2} + 0.10^{-3} + 6.10^{-4} + 4.10^{-5} + 8.10^{-6} + 8.10^{-7}\right)\ J.K^{-1}$$

3.2 Décomposition en base binaire

a - Principe

On peut exploiter l'idée générale de la décomposition précédente pour écrire un réel x mais cette fois en base binaire avec :

$$\boxed{x = (-1)^S \times 2^E \times \left(b_0 + b_1.10^{-1} + b_2.10^{-2} + \ldots + b_i.10^{-i} + \ldots\right)} \text{ avec } \begin{cases} S = 0,1 \text{ qui fixe le signe de x} \\ E \text{ entier relatif (exposant)} \\ b_i \in \{0,1,2,\ldots,9\} \textbf{ et on fixe } b_0 = 1 \end{cases} \quad \text{(III.1)}$$

Conclusion : Pour connaître le réel x il faut connaître S, E et les b_i.

b - Ecriture normalisée

La profondeur de codage des nombres en machine étant limitée (en python, c'est la taille mémoire qui est le facteur limitant), on doit fatalement limiter E et la donnée des b_i.

On normalise l'écriture des réels flottants de la manière suivante :

▶ Définition de la **mantisse** M
On choisit de ne garder que les $m+1$ premiers termes b_i ce qui permet de définir **la mantisse** avec $b_0 = 1$:

$$M = 1 + b_1 \times 2^{-1} + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \ldots + b_m \times 2^{-m}$$

avec $M \overset{b_0=1}{=} \sum_{k=0}^{m} \frac{b_k}{2^k} < \sum_{k=0}^{m} 2^{-k} < \sum_{k=0}^{\infty} 2^{-k} = \frac{1}{1-2^{-1}} = 2$ ainsi $\boxed{1 \leq M < 2}$

$\Longrightarrow$ **la mantisse est connue lorsque l'on connait les m premiers b_i** $\longrightarrow$ **le codage de M occupe m bits dans la mémoire**

► Définition de l'exposant E

On réserve e bits pour coder l'exposant.

On écrira en fait en mémoire *l'exposant décalé* E' tel que :

$$E' = E + 2^{e-1} - 1 \qquad \text{en imposant } E' \in [0, 2^e - 1] \text{ donc } E \in \mathbb{N}$$

Ainsi, on peut déterminer l'intervalle des valeurs possibles de l'exposant E :

$$\begin{aligned} & 0 \leq E' \leq 2^e - 1 \\ \Leftrightarrow\ & 0 + 1 - 2^{e-1} \leq E' + 1 - 2^{e-1} \leq 2^e - 1 + 1 - 2^{e-1} \\ \Leftrightarrow\ & 1 - 2^{e-1} \leq E \leq 2^{e-1} \end{aligned}$$

$\longrightarrow \quad E_{norm} \in [1 - 2^{e-1}, 2^{e-1}]$

Remarque - (3.2) - 3:

Attention : cet intervalle est différent de celui employé pour coder un entier relatif sur e bits. La convention adoptée ici ne vaut que pour l'exposant.

Certaines valeurs d'exposant sont réservées pour le codage de cas particulier :
- ► $E' = 2^e - 1$ soit $E = 2^{e-1}$ est employé pour coder l'infini ou un calcul conduisant à coder un "NaN" i.e. **N**ot **A** **N**umber, comme par exemple la division par 0.
- ► $E' = 0$ soit $E = 1 - 2^{e-1}$ est employé pour coder les nombres dénormalisés pour lesquels $M < 1$.

Finalement l'intervalle normalisé de valeurs possibles pour l'exposant est :

$$E_{norm.poss} \in [2 - 2^{e-1}, 2^{e-1} - 1]$$

A retenir :

Conclusion : un nombre binaire codé sur $m + e + 1$ *bits* s'écrit :

$$\underbrace{S}_{\text{signe}}\ \underbrace{c_{e-1} \ldots c_1 c_0}_{\text{exposant E'}}\ \underbrace{b_1 b_2 \ldots b_m}_{\text{mantisse}}$$

et sa valeur décimale est donnée par la relation III.1 une fois normalisée, soit :

$$x = (-1)^s \times 2^E \times M$$

Remarque - (3.2) - 4: Comme pour les entiers, l'intervalle des réels représentables en mémoire est limité. Ajoutons que s'il est possible de représenter tous les entiers dans l'intervalle donné, ce n'est pas le cas avec les réels puisqu'il y en a une infinité dans tout intervalle. Le nombre de bits consacrés à la mantisse $b1 \ldots b_m$ étant limité à m cela signifie qu'il existe un intervalle minimal "incompressible" entre deux réels successifs représentables. Ainsi, la plupart des réels décimaux ne sont pas représentables exactement en mémoire avec ce format et seront inscrits de manière approchée avec une précision dépendant de la profondeur de bits.

3.3 Norme IEEE754 - limitation des nombres en virgule flottante

Depuis 1985, la norme L'Institute of Electrical and Electronics Engineers (ou IEEE) a défini le principe de représentation en machine des nombres à virgule flottante ; il s'agit de la norme IEEE754. Elle fixe le nombre de bits consacrés à M,E et s en fonction de la profondeur de représentation choisie :

Profondeur	Signe S	Exposant décalé E'	Mantisse M
32 *bits*	1 *bits*	8 *bits*	23 *bits*
64 *bits*	1 *bits*	11 *bits*	52 *bits*

EXERCICE N°3: *Quelques précisions sur cette norme*

❶ *Quel est l'intervalle des exposants représentables en norme IEEE754 pour une profondeur de 64 bits ?*

❷ *Déterminer l'expression du plus grand réel positif représentable x_M, et donner sa valeur numérique sous la forme $r \times 10^n$ avec n entier et $|r| < 10$, toujours en 64 bits.*

❸ *Déterminer l'expression du plus petit réel positif normalisé représentable x_m , et donner sa valeur numérique sous la forme $x \times 10^n$ encore une fois en 64 bits.*

RÉPONSE :

❶ $E \in [-2^{10-1} + 1, 2^10] = [-1023 + 1, 1024]$

❷ $x_M = (-1)^0 \times 2^{1023} \times \left[1 + \sum_{i=1}^{52} \frac{1}{2^i}\right] = 1,79769.10^{308}$

❸ $x_m = (-1)^0 \times 2^{-1022} \times \underbrace{\left[1 + \sum_{i=1}^{52} \frac{0}{2^i}\right]}_{=1} = 2,22507.10^{-308}$

Dans la mesure où le nombre de bits consacré à M est limité, il existe une précision absolue de représentation des nombres réels en virgule flottante. Cette précision absolue est définie par la différence entre deux réels flottants consécutifs représentables :

$$\delta x = |x' - x|$$

EXERCICE N°4: *Précision de représentation*

❶ *Déterminer la plus petite variation de mantisse possible.*

❷ *En déduire la precision absolue δx pour les plus petits nombres représentables en 64 bits et pour les plus grands.*

REMARQUE - (3.3) - 5: Il est possible de sortir de la norme IEEE754 pour étendre l'intervalle des flottants représentables ou bien augmenter la précision sur un flottant. Il suffit pour cela de modifier le nombre de bits consacrés à e et m avec la contrainte $e + m = cste = 7,\ 15,\ 31,\ ou\ 63\ bits$ (le bit manquant étant celui réservé au signe du flottant). On a un compromis à trouver entre portée (intervalle des réels représentés) et précision. Pour augmenter la précision, il faut augmenter m, mais alors e diminue et on peut représenter moins d'entiers. Pour augmenter la portée, il faut augmenter e, mais alors m diminue et la précision également.

3.4 Exemples de représentation en mémoire et conversions (partie optionnelle !)

QUESTIONS : comment représenter en base 10 un nombre donné en binaire format 64 bits IEEE754 ?

RÉPONSE : par simple "lecture" !!!!

Sur un exemple : $(x)_2 = 0000010001101001001111000011100000000000000000000000000000000000$
soit :

- bit de signe $= 0 \Longrightarrow$ nombre positif
- bits d'exposant décalé $E' = 00001000110 = 70$ soit $E = E' - 2^{e-1} + 1 = 70 - 2^{10} + 1 = 70 - 1024 + 1 = -953$
- bits de mantisse :

$$m = 1+2^{-1}+2^{-4}+2^{-7}+2^{-8}+2^{-9}+2^{-10}+2^{-15}+2^{-16}+2^{-17} = \frac{(2^{17}+2^{16}+2^{13}+2^{10}+2^9+2^8+2^7+2^2+2+1)}{2^{17}}$$

soit $m = \dfrac{206727}{131072}$

Finalement ce nombre flottant est :

$$x_{10} = +2^{-953} \times \frac{206727}{131072}$$

QUESTIONS : comment représenter en base 2 format 64 bits IEEE754 un nombre donné en décimal ?

RÉPONSE : les choses se compliquent un peu ici !!! Il existe plusieurs méthodes ; nous proposons ici la méthode de décomposition par division déjà exploitée pour la conversion des entiers :
On rappelle qu'un réel flottant normalisé s'écrit de la manière suivante :

$$x = (-1)^S \times 2^E \times \underbrace{\left[1 + \sum_{i=0}^{52} b_i \cdot 2^{-i}\right]}_{1 \leq M < 2}$$

On veut coder le décimal $x = 0,4$

- BIT DE SIGNE :

$$x > 0 \quad \Longrightarrow \quad \text{le bit de signe est 0}$$

- BITS D'EXPOSANT :
  - $x < 1 \quad \Longrightarrow \quad E < 0$
  - $x < 0.5 = 2^{-1} \quad \Longrightarrow \quad E \neq -1$
  - $x > \dfrac{1}{2^2} = 0,25 \quad \Longrightarrow \quad E = -2 \quad \Longrightarrow \quad E' = E + 1023 = 1021$
  - Ainsi : $E' = (01111111101)_2$

- BITS DE MANTISSE : Il reste à coder la somme apparaissant dans la mantisse soit :

$$\sum_{i=0}^{52} b_i \cdot 2^{-i} = x \times 2^{-E} - 1 = 0.4 \times 4 - 1 = 0,6$$

On va exploiter comme pour les entiers la méthode par division afin de décomposer 0,6 en puissance de 2^{-i}, soit finalement une méthode par multiplication car diviser par 2^{-i} revient à multiplier par 2^i :

$$
\begin{array}{ll}
0,6 \times 2 = 1,2 & \text{le coefficient } b_1 \text{ de } 2^{-1} \text{ est } \quad 1 \\
0,2 \times 2 = 0,4 & \text{le coefficient } b_2 \text{ de } 2^{-2} \text{ est } \quad 0 \\
0,4 \times 2 = 0,8 & \text{le coefficient } b_3 \text{ de } 2^{-3} \text{ est } \quad 0 \\
0,8 \times 2 = 1,6 & \text{le coefficient } b_4 \text{ de } 2^{-4} \text{ est } \quad 1 \\
0,6 \times 2 = 1,2 & \text{le coefficient } b_5 \text{ de } 2^{-5} \text{ est } \quad 1 \\
\multicolumn{2}{c}{\ldots}
\end{array}
$$

$\Longrightarrow$ on constate que ce codage est périodique, et qu'il ne sera limité que par la profondeur de représentation choisie, or ici nous codons la somme de la mantisse sur 52 bits. Ainsi $0,4$ va s'écrire en norme IEEE754 :

$$
\underbrace{0}_{=S} \; \underbrace{01111111101}_{=E'} \; \underbrace{1001100110011001100110011001100110011001100110011001}_{\text{coeff. } b_1, b_2, \ldots, b_{52}}
$$