Codage des nombres réels en mémoire:

1: Cas des nombres entiers (*ou partie entière d'un nombre réel*)

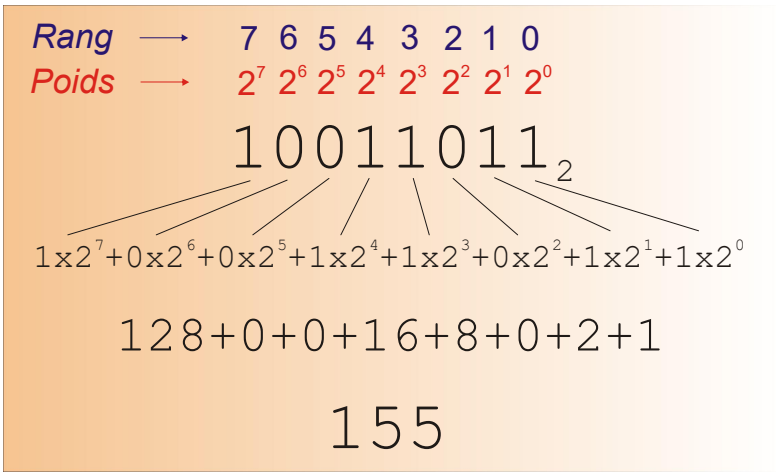
Nous avions vu dans le cours sur le codage des nombres entiers comment on transformait un nombre entier en binaire. La méthode c'est **la division successive par la base d'arrivée** soit 2 pour le binaire. On peut aussi convertir un nombre entier en binaire par une soustraction de poids!

Le poids en binaire est représenté par la base à la puissance du rang = **(la base)^{rang}**
Rappelez vous les unités en décimal sont au rang 0, les dizaines au rang 1 etc...

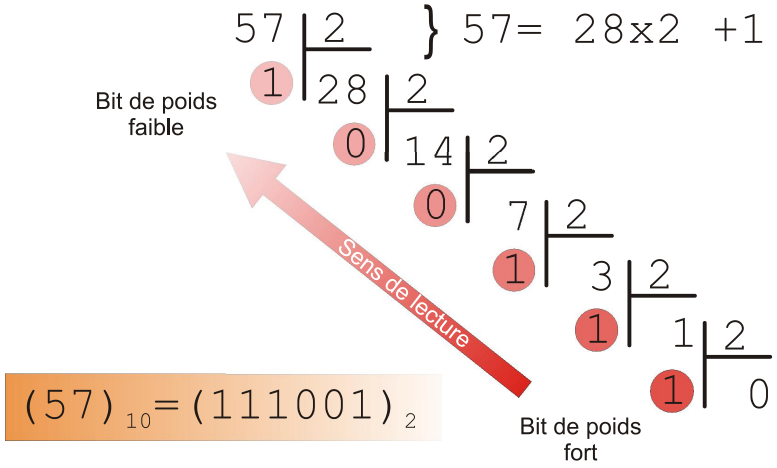
On obtient:

Rang	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Poids	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
Valeur Décimal	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536

Exemple: 155



Exemple: 57 s'écrit en binaire 32 + 16 + 8 + 1 soit $2^5 + 2^4 + 2^3 + 2^0 = 00111001_{(2)}$ valeur exprimée sur un octet (8 bits).



2 : Cas de la partie fractionnaire du nombre réel

(c'est ce qui est après la virgule ;-)

La méthode pour convertir la partie fractionnaire d'un nombre décimal en binaire est:

la multiplication successive par la base d'arrivée!

Exemple_1: Combien vaut «12,625 = ?» en binaire.
https://youtu.be/h_3JGDF8V1o
La partie entière soit ici 12 = 8 + 4 = $2^3 + 2^2$ = $00001100_{(2)}$ *(méthode somme de poids toujours sur 1 octet)*
La partie fractionnaire = 0,625 méthode de la multiplication successive par la base d'arrivée.

0,625 * 2 = **1**,25 on garde la partie entière du résultat de la multiplication (**1**) et on remet le reste:
0,25 * 2 = **0**,5 on garde la partie entière du résultat de la multiplication (**0**) et on remet le reste:
0,5 * 2 = **1**,0 il reste 0 c'est fini, il n'y a plus de reste!

On obtient donc le résultat final:

12,625 = $00001100,1010_{(2)}$ = $0C,A_{(H)}$

Exemple_2: Combien vaut «12,7 = ?» en binaire.
https://youtu.be/1fBpkvVOW90
La partie entière (on vient de la faire) = $00001100_{(2)}$
La partie fractionnaire = 0,7

0,7 * 2 = **1**,4 soit **1** (valeur du rang -1)
0,4 * 2 = **0**,8 soit **0** (valeur du rang -2)
0,8 * 2 = **1**,6 soit **1** (valeur du rang -3)
0,6 * 2 = **1**,2 soit **1** (valeur du rang -4)
0,2 * 2 = **0**,4 soit **0** (valeur du rang -5)
0,4 * 2 = **0**,8 soit **0** (mais je l'ai déjà fait à la deuxième ligne, ça ne s'arrêtera donc jamais ..!)

On obtient donc le résultat suivant:

12,7 = 00001100,10110011001100110011001100110011001100110011001100...(2)

Faut surtout pas oublier les trois petits points à la fin puisque cela ne s'arrête jamais!

Comment faire pour mettre toute la valeur de 12,7 en binaire dans la mémoire d'un ordinateur ?
C'est impossible ... à nombre infini correspond mémoire infinie ... On a donc dû réaliser une structure de représentation des nombres réels en mémoire.
Les nombres en mémoire sont codés par leurs types représentatif de la valeur du nombre à stocker.

En langage C :

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant double	8	$1.7*10^{-308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$3.4*10^{-4932}$ à $3.4*10^{4932}$

Comment mettre ce nombre infini (12,7) en mémoire ?

Ce nombre peut-être défini suivant trois types:

Float codé sur **4 octets** (32 bits)
Double Float codé sur **8 octets** (64 bits)
Long Double Float codé sur **10 octets** (80 bits)

C'est différents bits sont répartis en 3 champs:

Le signe «**S**» 1 bit (**0** si positif et **1** si négatif)
L'exposant «**E**» (nombre de bit dépend du type choisi)
La mantisse «**M**» (nombre de bit dépend du type choisi)

Les nombres de type **float** sont codés sur **32 bits** dont :

- **M** = 23 bits pour la **M**antisse
- **E** = 8 bits pour l'**E**xposant
- **S** = 1 bit pour le **S**igne

Les nombres de type **double float** sont codés sur **64 bits** dont :

- **M** = 52 bits pour la mantisse
- **E** = 11 bits pour l'exposant
- **S** = 1 bit pour le signe

Les nombres de type **long double float** sont codés sur **80 bits** dont :

- **M** = 64 bits pour la mantisse
- **E** = 15 bits pour l'exposant
- **S** = 1 bit pour le signe

La précision des nombres réels est approchée. Elle dépend du type choisi pour sa représentation :

- de 6 chiffres (après la virgule) pour le type **float**
- de 15 chiffres pour le type **double**
- de 17 chiffres pour le type **long double**

Reprenons notre exemple pour un type float (32 bits):

$$12,7 = 00001100,10110011001100110011001100110...{}_{(2)}$$

https://youtu.be/9fuf_z_8wNY

1: Le nombre est positif donc le bit de signe est égale à «**0**» donc **S = 0**

2: Pour obtenir la valeur de l'exposant il faut:

Déplacer la virgule, c'est à dire obtenir une représentation du nombre sous la forme: **1,...**
On ramène la virgule derrière le premier «**1**» de la représentation binaire du nombre)

Deux cas sont possibles:

1: **Si le nombre est supérieur à 1** (cas du nombre 12,7) le déplacement de la virgule se fait vers la gauche est ***le nombre de décalage de cette virgule est positif.***

Soit 3 décalages à gauche pour le nombre:

12,7 Codé en binaire	0000**1**100,1001100...$_{(2)}$
Devient après **+3** décalages à gauche	**1**,10010011001100...$_{(2)}$

La valeur de l'**E**xposant s'obtient en ajoutant la valeur 127 au nombre de décalage.
Soit l'**E**xposant = 3 + 127 = 130 = 128 +2 = **E = 10000010$_{(2)}$** valeur binaire de l'exposant!

2: **Si le nombre est inférieur à 1** le décalage de la virgule effectue vers la droite et:

la valeur du nombre de décalage est négatif!

Le nombre codé en binaire	0,0000001101010...(2)
Devient après **7** décalages à droite	1,101010...(2)

La valeur de l'exposant s'obtient donc en effectuant le même calcul:

E = -7 + 127 = 120 = 64 + 32 + 16 + 8 = $2^6 + 2^5 + 2^4 + 2^3$ = 01111000(2)

3: Pour obtenir la valeur de la **M**antisse il suffit de prendre les 23 bits qui se situe après la virgule.

Exemple pour 12,7:

12,7 = 00001100,10110011001100110011001100110...(2)

Après décalage de la virgule on obtient en rouge la mantisse codée sur 23 bits pour un type float:

00001,10010110011001100110011001100110011...(2)

En finalité le nombre 12,7 sera représenté en mémoire sous la forme (**S**igne,**E**xposant,**M**antisse):

S	**Exposant 8 bits**								**Mantisse 23 bits type Float**																			
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
4				1					4			B				3				3				3				3			

Résultat on obtient pour la valeur **12,7 = 414B3333$_H$** pour un type Float codé sur 32 bits

Le codage du nombre 12,7 est stocké en mémoire sur 4 octets = 41_H $4B_H$ 33_H 33_H

Dans cette représentation une infinité de bits on été ignoré, tous les bits (en verre) qui arrivent après le format imposé par le type choisi pour représenter cette valeur en mémoire. Ces bits ignorés représente l'**erreur de troncature**.

Facultatif ***(pour ceux qui veulent aller plus loin)***

On peut calculer cette erreur de troncature, il faut reprendre la représentation du nombre :

12,7 = 00001100,10110011001100110011001100110011001100...(2)

En vert tous les bits qui ont été ignoré dans la représentation du nombre en mémoire.

On a exprimé le rand des bits en fonction de leurs positions dans le nombre.

Après la virgule le rang devient négatif ! Soit rang -1, -2,-3, ... etc

Le premier «**1**» qui a été ignoré se situe au rang = **-23,** le suivant au rang **-24** puis le rang **-27** ... etc

Le poids de ces bits qui ont été tronqué correspond à $2^{-23} + 2^{-24} + 2^{-27} + 2^{-28} + 2^{-31} + ...$

Merci les Mathématiques qui vont nous permettre de calculer cette erreur de troncature en calculant la somme de tout ces bits qui ont été ignoré!

C'est une progression de type géométrique ;-) dont nous savons calculer la somme!

Erreur = $2^{-23} + 2^{-24} + 2^{-27} + 2^{-28} + 2^{-31} + ...$

Si on regroupe les puissances paire et impaire ensemble on obtient:

Erreur = $(2^{-23} + 2^{-27} + 2^{-31} + ...) + (2^{-24} + 2^{-28} + 2^{-32} + ...)$

Erreur = $(2^{-23} + 2^{-27} + 2^{-31} + ...) + 1/2\ (2^{-23} + 2^{-27} + 2^{-31} + ...)$

Si on pose **A = $(2^{-23} + 2^{-27} + 2^{-31} + ...)$**

On obtient: **Erreur = A + 1/2 (A) = 3/2 (A)**

Reste plus qu'à calculer **A = ?** C'est relativement facile si on démontre que l'on est en présence d'une progression de type géométrique. Dans ce cas la valeur de la progression est égale à:

$\sum_{1}^{n} U_n = U_1\left(\frac{1-q^n}{1-q}\right)$ avec U_1 le premier terme de la progression est q la raison tel que :

$U_{n+1} = U_n * q$ (on peut calculer le terme suivant de la progression à partir du précédent en le multipliant par la raison **q**). Si cela est vrai quelque soit le terme de la progression on est bien en présence d'une progression géométrique!

Dans notre exemple: **$A = (2^{-23} + 2^{-27} + 2^{-31} + 2^{-35} + 2^{-39} + 2^{-43} + ...)$**

Donc: $U_1 = 2^{-23}$, $U_2 = 2^{-27}$, $U_3 = 2^{-31}$, ...

Pour démontrer qu'on est bien en présence d'une progression géométrique il faut vérifier que q est bien une constante quelque soit le terme de la progression.

Soit : $U_{n+1} = U_n * q$ donne dans notre exemple

$U_2 = U_1 * q$ soit **q =** $U_2 / U_1 = 2^{-27} / 2^{-23} = 2^{-4} =$ **1/16**

$U_3 = U_2 * q$ soit **q =** $U_3 / U_2 = 2^{-31} / 2^{-27} = 2^{-4} =$ **1/16**

$U_4 = U_3 * q$ ...

On est bien en présence d'une progression géométrique on peut donc calculer la valeur finale de cette progression en calculant:

$$\mathbf{A} = \sum_{1}^{n} U_n = U_1 + U_2 + U_3 + ... + U_n$$

$$\mathbf{A} = \sum_{1}^{n} U_n = U_1\left(\frac{1-q^n}{1-q}\right)$$

Soit : **A =** $$\sum_{1}^{n} Un = U1\left(\frac{1-q^n}{1-q}\right) = 2^{-23}\left(\frac{1-\left(\frac{1}{16}\right)^n}{1-\frac{1}{16}}\right) = \frac{1}{2^{23}}\left(\frac{1}{1-\frac{1}{16}}\right) = \frac{1}{2^{23}}\left(\frac{2^4}{15}\right) = \frac{1}{15*2^{19}}$$

A = 3,81469726562 E-7

Erreur = 3/2 A = 5,72204589843 E-7

On retrouve bien la précision d'une donnée codée en type float : 6 chiffres après la virgule !

Soit 12,7 '=' 414B3333$_H$ est égale réellement à : **12,6999994278** dans sa représentation en mémoire

Fin de la partie facultative ;-)

Si on vient de voir comment coder un nombre réel en mémoire, il est maintenant possible d'effectuer l'opération inverse. Connaissant la valeur qui est stocker en mémoire on peut calculer sa valeur décimale.

Exemple: valeur = 4647B924$_H$

S	**Exposant 8 bits**								**Mantisse 23 bits type Float**																							
0	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
4			6				4		7			B			9			2			4										

Signe : Positif (**S = 0**)

Exposant = $8C_H = 10001100_{(2)} = 128 + 8 + 4 =$ **140**

Nombre de décalage = 140 - 127 = 13 décalages (ou soustrait les 127 qui ont été ajouté lors du codage)

Normalisation de la **M**antisse sous la forme **1,M**

On obtient: 1,10001111011100100100100

On applique ensuite le nombre de décalage soit (13 décalages vers la droite, inverse du codage):

Valeur = **$11000111101110,0100100100_{(2)} = 2^{13} + 2^{12} + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 2^1 + 2^{-2} + 2^{-5} + 2^{-8}$**

Valeur = **8192 + 4096 + 256 + 128 + 64 + 32 + 8 + 4 + 2 + 1/4 + 1/32 + 1/256**

Valeur = +12782,2851562 La valeur de départ était 12782,3 (erreur = 0,0148438)