Conversions des Flottants
IEEE 754

Ce document a pour objectif de vous familiariser avec les conversions entre plusieurs bases pour les flottants en respectant la norme IEEE 754.

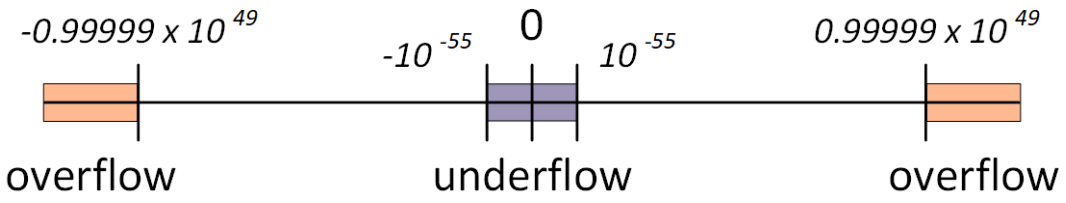
La plupart des conversions que nous effectuerons seront entre les bases 2, 10, et 16 pour les flottants IEEE 754.
Pour rappel, un nombre en indice peut indiquer la base (10_2 indique du binaire, 10_{10} indique du décimal, ...) tout comme un symbole en préfixe (% indique du binaire, et $ indique de l'hexadécimal). Sans symbole particulier, on considère qu'il s'agit de la base 10 usuelle.

1 Représentation des flottants

Les flottants concernent les nombres à virgules. Représenter et manipuler des nombres à virgules n'est pas aisé dans le sens où plusieurs questions se posent : combien de nombres après la virgule faut-il gérer ? Quel est le plus petit pas/décalage supporté par un ordinateur ? ... Ces questions touchent au principal problème rencontré par l'informatique dans le traitement des données scientifiques : la « précision » .

La précision décrit combien de bits seront utilisés pour représenter les nombres flottants. Dans la norme IEEE 754, il existe plusieurs formats pour représenter les flottants, dont la *simple précision* (sur 32 bits), la *double précision* (sur 64 bits), et la *double précision étendue* (sur 80 bits).

Parmi les contraintes que vous aviez déjà vus, un nombre entier codé sur 8 bits ne pourra pas aller au delà de la valeur 256 : si on a %1111 1111 et qu'on lui ajoute 1, alors de la valeur 256, on repassera à 0. Ce phénomène s'appelle un *integer overflow* (dépassement d'entier), qu'il ne faut pas confondre avec les autres types d'*overflows* (stack overflow, buffer overflow). Dans le cas des flottants, ce type de dépassement est également possible, mais, il existe également le cas inverse : l'*underflow* où l'on cherche à représenter une valeur trop petite pour la précision choisie.



(extrait de « *Englander : The Architecture of Computer Hardware and Systems Software* »)

Ainsi, non seulement il existe une limite haute/basse sur la partie entière représentable d'un nombre, mais également sur la quantité de chiffres après la virgule. Les nombres flottants sont donc parfois des approximations lorsqu'ils sont manipulés. En développement, on ne teste *jamais* l'égalité entre deux variables représentées par des nombres flottants, mais uniquement un écart entre elles (si cet écart est suffisamment négligeable, alors elles peuvent être considérées comme égales).

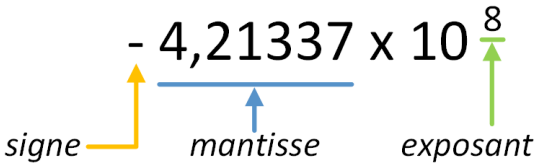
Lorsque l'on travaille sur les nombres à virgules, vous connaissez déjà une forme de notation dite scientifique. On y met un unique entier entre 1 et 9 (inclus), et on l'accompagne d'une virgule suivi d'un nombre, puis on le multiplie par une puissance de 10.

$$-2541,3945 = -2,5413945 \times 10^3$$

Cette notation est parfois simplifiée en utilisant un e pour *exposant*.

$$0,0001337 = 1,337 \times 10^{-4} = 1,337\text{e} - 4$$

Ce format $\pm a \times 10^n$ est également utilisé pour représenter les nombres dits flottants (car la virgule « flotte » selon la puissance de 10 utilisée). Cette notation se compose de trois éléments :



— *signe* : positif ou négatif (ici « - »)
— *exposant* : la puissance de 10 (ici 8)
— *mantisse* (ou significande) : le nombre décimal (ici $4,21337$)

Dans la notation IEEE 754, on utilise ces mêmes concepts, mais appliqués à la base 2 et avec une taille précise pour représenter chacun d'entre eux. Ainsi, la mantisse a une valeur minimale et une valeur maximale, tout comme l'exposant. De plus, le vocabulaire varie légèrement du fait que la norme impose quelques contraintes. On parlera donc dans le vocabulaire formel de :

— *signe* : positif ou négatif
— *exposant biaisé* : la puissance de 2, à laquelle il faut ajouter une valeur (le biais)
— *mantisse* : la partie décimale après la virgule (en ignorant le 1 de la partie entière)

représentation des flottants IEEE 754 simple précision (32 bits)

représentation des flottants IEEE 754 double précision (64 bits)

De plus, étant donné que les représentations numériques ont des limites, on distingue plusieurs cas dans la taille des flottants : la *simple précision* (ou *single precision* en anglais) sur 32 bits, la *double précision* sur 64 bits, et d'autres formats que nous n'étudierons pas. La précision fait varier la taille générale des flottants, ce qui implique que les flottants double précision couvreront plus de grandes valeurs, mais également plus de petites valeurs proches de 0, que les flottants simple précision.

— *signe* : 1 bit
— *exposant* : 8 bits (simple précision), 11 bits (double précision), 15 bits (quadruple precision)
— *mantisse* : 23 bits (simple précision), 52 bits (double précision), 112 bits (quadruple precision)

Enfin, pour convertir les décimaux en flottants IEEE 754, il existe une convention pour les *nombres normalisés* (valeur absolue supérieure à 1), et les *nombre dénormalisés* (valeur absolue inférieure à 1).

2 Valeurs réservées

La norme prévoit également une réservation de certaines valeurs clés. Vous devez les connaître pour pouvoir les reconnaître.

Type	Valeur hexadécimale	Signe	Exposant	Mantisse
+ Zéro	$ 0000 0000	0	%0000 0000	% 000 0000 0000 0000 0000 0000
- Zéro	$ 8000 0000	1	%0000 0000	% 000 0000 0000 0000 0000 0000
+ ∞	$ 7F80 0000	0	%1111 1111	% 000 0000 0000 0000 0000 0000
- ∞	$ FF80 0000	1	%1111 1111	% 000 0000 0000 0000 0000 0000
NaN (borne B)	$ _F80 0001	X	%1111 1111	% 000 0000 0000 0000 0000 0001
NaN (borne H)	$ _FFF FFFF	X	%1111 1111	% 111 1111 1111 1111 1111 1111

3 Nombres normalisés : base 10 vers IEEE 754

Les nombres normalisés dans le format IEEE 754 concernent les nombres dont la valeur absolue est supérieure à 1 (mais restent inférieurs à la borne maximale gérée par la précision choisie). L'objectif est de représenter les nombres Voici les étapes pour convertir les nombres normalisés :

1. Récupérer le signe du nombre (positif = 0, négatif = 1)
2. Séparer la partie entière de la partie décimale
3. Convertir la partie entière en binaire (sans gérer le signe)
4. Convertir la partie décimale en binaire (sans gérer le signe)
5. Fusion des parties entière et décimale en binaire tout en gardant la virgule
6. Réécriture en notation scientifique base 2
7. Calculer l'exposant pour le format IEEE 754 en l'ajoutant au biais de la précision choisie
8. Convertir l'exposant biaisé en binaire
9. Reporter la mantisse selon la précision choisie

Nous traiterons ici comme exemple la valeur $-42,15625$.

3.1 Récupérer le signe du nombre

On place dans le bit de poids fort servant à coder le signe un 0 si le nombre est positif, ou un 1 si le nombre est négatif. Dans l'ensemble des étapes suivantes, on peut ignorer le signe du nombre.

On retient donc 1 que l'on place au bit 31 en simple précision, et au bit 63 en double précision.
Pour les étapes suivantes, on travaillera donc sur $42,15625$.

3.2 Séparer la partie entière de la partie décimale

On sépare la partie entière de la partie décimale afin de les convertir en binaire chacune de leur façon.

$42,15625$ devient donc 42 et $0,15625$.

3.3 Convertir la partie entière en binaire

On convertit la partie entière en binaire avec l'algorithme classique de division par 2.

Dans les traitements suivants, le 1 de tête sera considéré comme implicite, donc il sera omis lors de l'écriture finale de la mantisse. Pour le traitement suivant concernant la conversion binaire de la partie décimale, il est parfois utile de connaître le nombre de bits utilisés pour représenter cette partie (moins le 1 de tête). Néanmoins, pour connaître le nombre de décalages nécessaires, il est nécessaire de conserver le nombre binaire tel quel.

$42_{10} = 101010_2$

3.4 Convertir la partie décimale en binaire

On convertit la partie décimale en binaire. Pour cela, on effectue des multiplications successives par 2 en faisant l'extraction de la partie entière. En effet, les bits représentent toujours des puissances de 2, mais des puissances négatives (donc 2^{-1}, 2^{-2}, 2^{-3}, ... soit $0,5$, $0,25$, $0,125$, ...).

La condition d'arrêt est soit d'atteindre « $1,0$ », soit que la taille de la partie entière convertie en binaire + le nombre de multiplication tienne sur la taille de la mantisse exploitée. Dans notre cas, 42 est converti en 101010_2, donc en omettant le 1 de tête, 01010_2, celui-ci prendra 5 bits dans la mantisse. En simple précision, la mantisse étant de 23 bits, on pourra donc placer 18 bits $(23 - 5)$ de nombres après la virgule.
S'il est impossible de représenter le nombre sur 23 bits (exemple : $\frac{1}{3}$), on arrondit la dernière multiplication réalisable à 0 ou 1.

multiplication		résultat	entier
$0,15625 \times 2$	=	$0,31250$	0
$0,3125 \times 2$	=	$0,6250$	0
$0,625 \times 2$	=	$1,250$	1
$0,25 \times 2$	=	$0,50$	0
$0,5 \times 2$	=	$1,0$	1

On lit cette fois les multiplications dans l'ordre où elles ont été effectuées.

$0,15625_{10} = 0,00101_2$

3.5 Fusion des parties entière et décimale en binaire tout en gardant la virgule

On fusionne les parties entières et décimales converties en binaire, tout en conservant le placement de la virgule.

$42_{10} = 101010_2$

$0,15625_{10} = 0,00101_2$

$42,15625_{10} = 101010,00101_2$

3.6 Réécriture en notation scientifique base 2

On déplace la virgule pour atteindre le 1 le plus à gauche de la partie entière, et on garde ce décalage comme exposant.

$101010,00101_2 = 10\ 1010,0010\ 1_2 \times 2^0 = 1,0101\ 0001\ 01_2 \times 2^5$

3.7 Calculer l'exposant en ajoutant le biais

On ajoute le biais (lié à la précision choisie) à l'exposant précédemment obtenu. En simple précision, le biais est de 127. En double précision il est de 1023. En quadruple précision, il est de 16383.

$1,0101000101_2 \times 2^5 \ \Rightarrow \ 2^{\textcircled{5}}$

Simple précision : $5 + 127 = 132$

Double précision : $5 + 1023 = 1028$

3.8 Convertir l'exposant biaisé en binaire

On convertit en binaire l'exposant biaisé, et on le reporte dans le champ prévu.

Simple précision (8 bits) : $132_{10} = \ 1000\ 0100_2$

Double précision (11 bits) : $1028_{10} = \ 100\ 0000\ 0100_2$

3.9 Reporter la mantisse selon la précision choisie

On reporte la mantisse dans le champ prévu en omettant le 1 en tête : en effet, celui-ci est implicite par l'écriture scientifique en base 2.
De plus, on ajoute autant de 0 que nécessaire à droite du nombre.

$1,0101000101_2 \times 2^5 \ \Rightarrow \ 0101000101_2$

Simple précision (23 bits) :
$01010001010000000000000_2$ $\qquad$ $010\ 1000\ 1010\ 0000\ 0000\ 0000_2$

Double précision (52 bits) :
$0101000101000000000000000000000000000000000000000000_2$
$0101\ 0001\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

3.10 Résultat

Enfin, on peut fusionner l'ensemble des trois champs pour représenter un nombre normalisé au format IEEE 754 : signe (1), exposant (10000100_2 ou 10000000100_2), mantisse

Simple précision (32 bits) :
$11000010001010001010000000000000_2$ $\qquad$ $1\underline{100\ 0010\ 0}010\ 1000\ 1010\ 0000\ 0000\ 0000_2$

Double précision (64 bits) :
$1100000001000101000101000000000000000000000000000000000000000000_2$
$1\underline{100\ 0000\ 0100}\ 0101\ 0001\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

Simple précision (32 bits) : $-42.15625_{10} = \text{C228A000}_{16}$ $\qquad$ C228 A000_{16}

Double précision (64 bits) : $-42.15625_{10} = \text{C045140000000000}_{16}$ $\qquad$ $\text{C045 1400 0000 0000}_{16}$

4 Nombres normalisés : IEEE 754 vers base 10

Pour interpréter une valeur flottante IEEE 754, il est nécessaire de s'assurer tout d'abord qu'il ne s'agisse pas d'une des valeurs réservées, particulièrement les NaN (*Not a Number* en anglais) indiquant que la valeur n'est pas un nombre, et les valeurs infinies. Pour cela, il faut effectuer les mêmes étapes en sens inverse que pour la conversion, et s'intéresser à la valeur de l'exposant en priorité : un exposant à la valeur maximale (c'est-à-dire qu'en binaire il n'y aura que des 1) indiquera une valeur réservée (pour rappel : l'infini ou NaN). Un exposant et une mantisse à 0 indiqueront qu'il s'agit de la valeur zéro.

Voici les étapes pour convertir les nombres normalisés :

1. Séparer les champs
2. Retrouver le signe
3. Convertir l'exposant biaisé en base 10
4. Convertir la mantisse en base 10
5. Calculer la valeur décimale

4.1 Séparer les champs

La première étape consiste simplement à séparer les champs selon la précision du flottant étudié.

	simple précision (32 bits)	double précision (64 bits)	quadruple précision (128 bits)
signe	1 bit	1 bit	1 bit
exposant	8 bits	11 bits	15 bits
mantisse	23 bits	52 bits	112 bits

$\text{BA13 C700}_{16} = 1\underline{011\ 1010\ 0}001\ 0011\ 1100\ 0111\ 0000\ 0000_2$ simple précision (32 bits)

Signe (1 bit) :	1_2	
Exposant (8 bits) :	$0111\ 0100_2$	Exposant classique (ni nul ni au max)
Mantisse (23 bits) :	$001\ 0011\ 1100\ 0111\ 0000\ 0000_2$	

Au delà des valeurs réservées, la formule générale à appliquer est la suivante :

$$(-1)^{\text{signe}} \times (1 + \text{mantisse}) \times 2^{\text{exposant - biais}}$$

4.2 Retrouver le signe

On observe simplement si le bit de signe est à 1 (nombre négatif) ou à 0 (nombre positif).

Signe : 1 (nombre négatif)

4.3 Convertir l'exposant biaisé en base 10

On convertit l'exposant biaisé binaire en décimal afin de pouvoir soustraire le biais.
$0111\ 0100_2 = 116_{10}$

4.4 Convertir la mantisse en base 10

La mantisse est précédée d'un « 0, » afin de retrouver les puissances de 2 associées.

$001\ 0011\ 1100\ 0111\ 0000\ 0000_2 \Rightarrow 0,0010\ 0111\ 1000\ 111_2$

$$\begin{aligned} 0,0010\ 0111\ 1000\ 111_2 &= 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + ... + 1 \times 2^{-15} \\ &= 2^{-3} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-9} + 2^{-13} + 2^{-14} + 2^{-15} \\ &= 0,125 + 0,015625 + 0,0078125 + 0,00390625 + 0,001953125 \\ &\quad + 0,0001220703125 + 0,00006103515625 + 0,000030517578125 \\ &= 0,154510498046875 \end{aligned}$$

4.5 Calculer la valeur décimale

On applique enfin la formule suivante avec les valeurs trouvées et le biais associé à la précision :

$$(-1)^{\text{signe}} \times (1 + \text{mantisse}) \times 2^{\text{exposant - biais}}$$

Précision	Biais
Simple (32 bits)	127
Double (64 bits)	1023
Quadruple (128 bits)	16383

$$(-1)^1 \times (1 + 0,154510498046875) \times 2^{116-127}$$
$$=$$
$$(-1) \times (1,154510498046875) \times 2^{-11}$$
$$=$$
$$-1,154510498046875 \times 2^{-11}$$
$$=$$
$$-0,00056372582912445068$$
$$=$$
$$\textit{(notation scientifique)}\ -5,6372582912445068 \times 10^{-4}$$

Il ne faut surtout pas oublier qu'il s'agit d'une approximation : la valeur initiale qui était manipulée par l'ordinateur a été transformée en format IEEE 754 qui a provoqué une limitation dans la quantité de chiffres après la virgule. Les opérations elles-mêmes de conversion peuvent également souffrir d'un manque de précision.

En conclusion, le nombre retrouvé en fin de conversion n'est pas nécessairement la valeur exacte initialement manipulée.

Si un programme implique de manipuler et modifier plusieurs fois de suite un nombre flottant, il vaut mieux trouver une autre représentation, et ne convertir en flottant que la valeur finale (par exemple en créant un type abstrait « *fraction* » qui sera utilisé dans le programme, et uniquement lorsque les traitements auront été terminés, la valeur sera convertie en flottant).

5 Nombres dénormalisés

Les nombres dénormalisés (*subnormal numbers*, *denormalized numbers*, ou *denormals* en anglais) concernent les nombres dont la valeur absolue est inférieure strictement à 1 (c'est-à-dire les nombres tels que : $-1 < n < 1$). Plus précisément, ils servent à représenter des valeurs proches de 0, aux extrêmités de la représentation normalisée.

Dans le format binaire, on reconnait un nombre comme dénormalisé si son exposant est nul (il ne contient que des 0) et sa mantisse est non nulle. En effet, si la mantisse et l'exposant sont nuls, alors cela représente la valeur réservée 0.

Type	Valeur hexadécimale	Signe	Exposant	Mantisse
± Zéro	\$ _000 0000	X	%0000 0000	% 000 0000 0000 0000 0000 0000
Dénormalisé (B)	\$ _000 0001	X	%0000 0000	% 000 0000 0000 0000 0000 0001
Dénormalisé (H)	\$ _07F FFFF	X	%0000 0000	% 111 1111 1111 1111 1111 1111
Normalisé (B)	\$ _080 0000	X	%0000 0001	% 000 0000 0000 0000 0000 0000
Normalisé (H)	\$ _F7F FFFF	X	%1111 1110	% 111 1111 1111 1111 1111 1111
± ∞	\$ _F80 0000	X	%1111 1111	% 000 0000 0000 0000 0000 0000
NaN (borne B)	\$ _F80 0001	X	%1111 1111	% 000 0000 0000 0000 0000 0001
NaN (borne H)	\$ _FFF FFFF	X	%1111 1111	% 111 1111 1111 1111 1111 1111

Ces nombres impliquent deux changements majeurs dans leur conversion :

— la mantisse ne contient pas de 1 implicite (celui ajouté/retiré avant la virgule lors de la conversion de la mantisse)

— on ajoute 1 à la soustraction entre l'exposant (toujours à 0) et le biais

Pour rappel, les nombres normalisés sont représentés par cette formule :

$$(-1)^{\text{signe}} \times (1 + \text{mantisse}) \times 2^{\text{exposant - biais}}$$

Mais pour les dénormalisés, on utilise cette formule :

$$(-1)^{\text{signe}} \times (0 + \text{mantisse}) \times 2^{\text{exposant - biais + 1}}$$
$$=$$
$$(-1)^{\text{signe}} \times \text{mantisse} \times 2^{\text{1 - biais}}$$

Si on gardait le format normalisé, en multipliant par « *1,mantisse* » on obtiendrait uniquement des valeurs proches de $2^{1-biais}$. Alors qu'en appliquant l'usage du « *0,mantisse* » on se rend compte que le résultat de la multiplication par $2^{1-biais}$ peut offrir des valeurs beaucoup plus petites que seule la mantisse contrôle.

Vous pouvez déduire de ces explications que la représentation de 0 est en réalité associée aux dénormalisés, car si on utilisait la représentation normalisée, alors la mantisse à 0 contiendrait tout de même le 1 implicite (ce qui ne ferait pas une valeur nulle).

5.1 Bornes dénormalisées

Essayons de trouver le plus petit nombre positif non-nul dénormalisé et le plus grand.

5.1.1 Borne inférieure

Le plus petit nombre positif non-nul dénormalisé en simple précision (32 bits) est :

$0000\ 0001_{16} = 0\underline{000\ 0000\ 0}000\ 0000\ 0000\ 0000\ 0001_2$

Signe (1 bit) :	0_2	
Exposant (8 bits) :	$0000\ 0000_2$	Exposant nul (dénormalisé)
Mantisse (23 bits) :	$000\ 0000\ 0000\ 0000\ 0000\ 0001_2$	

$$(-1)^{\text{signe}} \times \text{mantisse} \times 2^{1\text{ - biais}}$$
$$=$$
$$(-1)^0 \times 2^{-23} \times 2^{1-127}$$
$$=$$
$$2^{-23} \times 2^{-126}$$
$$=$$
$$2^{-149}$$

Le plus petit nombre positif non-nul dénormalisé en simple précision est donc 2^{-149} (soit environ $1,4 \times 10^{-45}$)

5.1.2 Borne supérieure

Le plus grand nombre positif non-nul dénormalisé en simple précision (32 bits) est :

$007F\ FFFF_{16} = 0\underline{000\ 0000\ 0}111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

Signe (1 bit) :	0_2	
Exposant (8 bits) :	$0000\ 0000_2$	Exposant nul (dénormalisé)
Mantisse (23 bits) :	$111\ 1111\ 1111\ 1111\ 1111\ 1111_2$	

$$(-1)^{\text{signe}} \times \text{mantisse} \times 2^{1\text{ - biais}}$$
$$=$$
$$(-1)^0 \times (\sum_{n=1}^{23} 2^{-n}) \times 2^{1-127}$$
$$=$$
$$(2^{-1} + ... + 2^{-23}) \times 2^{-126}$$
$$\approx$$
$$0,999999988 \times 2^{-126}$$

Le plus grand nombre positif non-nul dénormalisé en simple précision est proche de $0,999999988 \times 2^{-126}$ (soit environ $1,17549421 \times 10^{-45}$), ce qui se rapproche du plus petit nombre normalisé ($1,0 \times 2^{-126}$).

6 Limites de la représentation IEEE 754 (et des flottants en général)

6.1 Problèmes de représentation

Beaucoup de nombres ne sont pas représentables avec la représentation IEEE 754, y compris des nombres parfois très simples. Essayons de représenter $0,3$:

multiplication		résultat	entier
$0,3 \times 2$	=	$0,6$	0
$0,6 \times 2$	=	$1,2$	1
$0,2 \times 2$	=	$0,4$	0
$0,4 \times 2$	=	$0,8$	0
$0,8 \times 2$	=	$1,6$	1
$0,6 \times 2$	=	$1,2$	1
$0,2 \times 2$	=	$0,4$	0
...	=	...	...

Représenter exactement $0,3$ est en réalité impossible, on va devoir représenter une valeur proche, mais pas égale. Essayons néanmoins de représenter $0,1$ et $0,2$ (dont la somme est *normalement* égale à $0,3$) :

multiplication		résultat	entier	multiplication		résultat	entier
$0,1 \times 2$	=	$0,2$	0	$0,2 \times 2$	=	$0,4$	0
$0,2 \times 2$	=	$0,4$	0	$0,4 \times 2$	=	$0,8$	0
$0,4 \times 2$	=	$0,8$	0	$0,8 \times 2$	=	$1,6$	1
...	=	...	...	...	=	...	...

Exactement comme pour leur somme : il est déjà impossible de représenter exactement $0,1$ et $0,2$. On comprend ainsi que la partie après la virgule doit se terminer par un 0 ou un 5 si l'on souhaite espérer représenter exactement le nombre (mais ceci n'est pas la seule condition : il faut que l'ensemble de la partie décimale ne provoque aucun bouclage lors de sa multiplication par 2).

Ainsi, *il est extrêmement déconseillé de comparer directement deux flottants* (c'est même une très mauvaise pratique pouvant entraîner de nombreux problèmes). Il est nécessaire de définir un seuil d'acceptation afin de considérer si le nombre l'atteint. Par exemple, si on l'on souhaite vérifier si le résultat d'une opération produit le nombre $14,3$, on va plutôt chercher à vérifier si le nombre résultant est supérieur à $14,2$ et inférieur à $14,4$:

```
float res;
float cmp1, cmp2;

/* ... calculs ... */
res = 28.6 / 2;
/* ... calculs ... */

cmp1 = 14,2;
cmp2 = 14,4;

if ((res > cmp1) && (res < cmp2))
  return (true);
else
  return (false);
```

Afin de conserver un maximum de précisions, il est courant qu'en interne d'un processeur celui-ci manipule des valeurs flottantes beaucoup plus grandes que la simple ou la double précision. Par exemple, chez Intel, vous pouvez constater que les processeurs 32 bits (donc manipulant des entiers sur 32 bits) disposent de registres à virgule flottante sur 80 bits et 48 bits, tandis que les processeurs 64 bits disposent de registres à virgule flottante sur 80 bits et 64 bits (section 3.2 *Overview of the basic execution environment* du Volume 1 *Basic Architecture*). Historiquement les calculs flottants étaient réalisés sur un processeur externe qu'il fallait relier au processeur principal : la famille de FPU x87 (*Floating Point Unit*). Aujourd'hui, les circuits et instructions des x87 sont intégrés au processeur principal

La représentation IEEE 754 des nombres à virgule flottante n'est pas la seule existante. Toujours chez Intel, plusieurs formats cohabitent (IEEE 754, Bfloat16, ...), mais d'autres fabricants ou concepteurs de processeurs disposent de leurs propres formats, la représentation IEEE 754 ne devenant ainsi qu'un simple standard de représentation en cas d'échange de flottants entre des machines. Parmi les nombreux formats de représentation des nombres à virgule flottante, vous pouvez trouver : IEEE 754, Bfloat16, TensorFloat-32, FP8, HFP, ...

6.2 Transtypage

Lorsqu'un programme manipule un entier, puis, que cet entier est transformé en un nombre à virgule flottante, une opération de *transtypage* est réalisée. Elle est invisible aux yeux du développeur, mais sa réalisation n'est pas anodine pour le processeur : il s'agit d'une opération partant de la représentation entière en binaire pour obtenir un nombre au format scientifique en base 2.

$$42_{10} \quad \Rightarrow \quad \%101010 \quad \Rightarrow \quad \%1,01010 \times 2^5$$

$$\text{Signe : } 0 \qquad \text{Exposant : } 5 \qquad \text{Mantisse : } 01010$$

De même, récupérer la partie entière d'un flottant implique d'effectuer le décalage de la virgule... et de choisir si l'on applique ou non l'arrondi. Ces opérations sont relativement triviales, mais elles doivent être exécutées, et impliquent quelques cycles d'exécutions donc du temps.

6.3 Affichage des flottants avec printf(3)

Si on teste dans un code simple les opérations précédentes, on ne verra probablement pas les détails, car le compilateur effectue lui aussi des arrondis pour permettre aux utilisateurs de facilement lire les résultats :

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float flottant = 28.6;
  int entier;

  flottant = flottant / 2;
  entier = (int) flottant;
  printf("Integer : %d -- Float : %f\n", entier, flottant);

  return (0);
}

/* Integer : 14 -- Float : 14.300000 */
```

Pour observer finement les valeurs, on doit utiliser des options précises de la fonction *printf(3)* afin d'afficher avec beaucoup plus de précision les nombres.

```c
#include <stdio.h>
#include <float.h>

void FloatPrinter(float f)
{
  int Digs = DECIMAL_DIG;
  printf("  %.*e  \n", Digs, f);
}

int main(int argc, char **argv)
{
  float f1 = 0.1;
  float f2 = 0.2;
  float flottant = 28.6;
  int entier;

  /* Affichage de 28.6 en arrondi puis en precis */
  printf("Integer : %d -- Float : %f\n", entier, flottant);
  FloatPrinter(flottant);
  printf("\n");

  /* Calcul et resultat arrondi */
  flottant = f1 + f2;
  printf("res : %f -- (f1 : %f + %f : f2)\n", flottant, f1, f2);

  /* Precision augmentee sur le resultat */
  printf("res : ");
  FloatPrinter(flottant);
  printf("f1  : ");
  FloatPrinter(f1);
  printf("f2  : ");
  FloatPrinter(f2);

  return (0);
}

/*
  Integer : 55280 -- Float : 28.600000
  2.86000003814697265625e+01

  res : 0.300000 -- (f1 : 0.100000 + 0.200000 : f2)
  res :   3.00000011920928955078e-01
  f1  :   1.00000001490116119384e-01
  f2  :   2.00000002980232238769e-01
*/
```

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en novembre 2022 (dernière mise à jour décembre 2024)