

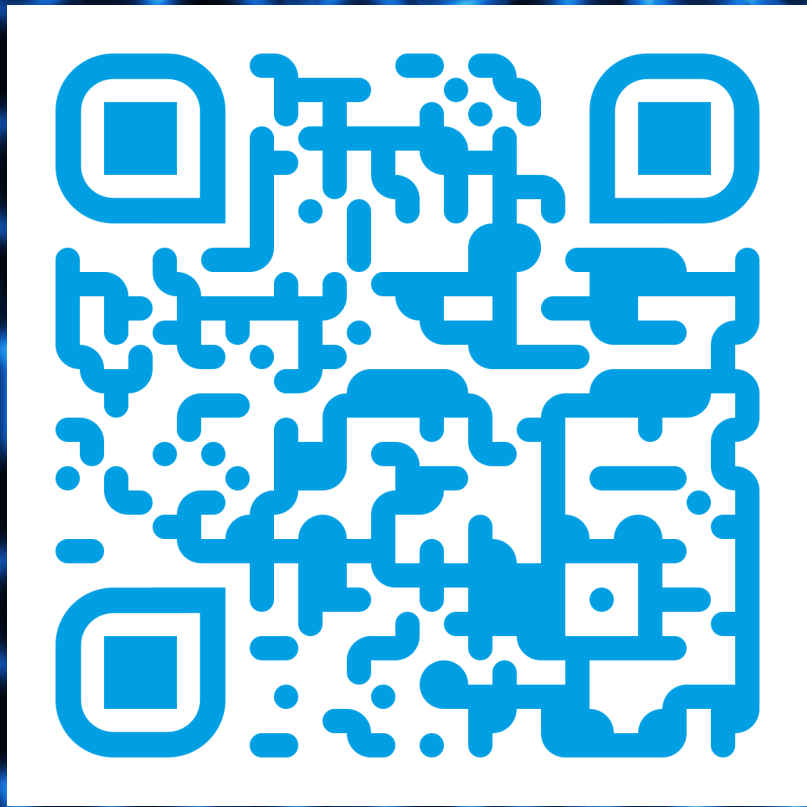
Architecture interne des ordinateurs



Pr. Dominique Ginhac
dginhac@ube.fr



<https://ginhac.com/archi/02-encodage.pdf>



Donner une base solide sur l'encodage de données en binaire afin de préparer à la programmation en assembleur.

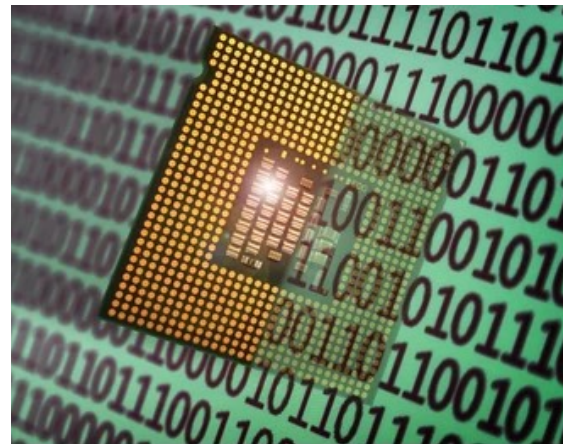
Enjoy! 😊

Une définition

L'encodage binaire est une méthode de représentation des données numériques à l'aide de 0 et 1.

C'est un système simple et efficace, particulièrement adapté aux circuits électroniques qui utilisent des niveaux de tension pour représenter ces deux états.

Encodage binaire



Dans un processeur, **tout** est stocké en binaire (base 2) sous la forme d'une succession de 0 et 1.

Entiers naturels (strictement positifs) \mathbb{N} : 42, 1337

Entiers relatifs (positifs et négatifs) \mathbb{Z} : -42, 666

Réels (avec précision limitée) \mathbb{R} : 1.618, -273.16

Caractère : 'a', 'z', 😊, 😍, 💩, ..

Chaines de caractères : "hello, world"

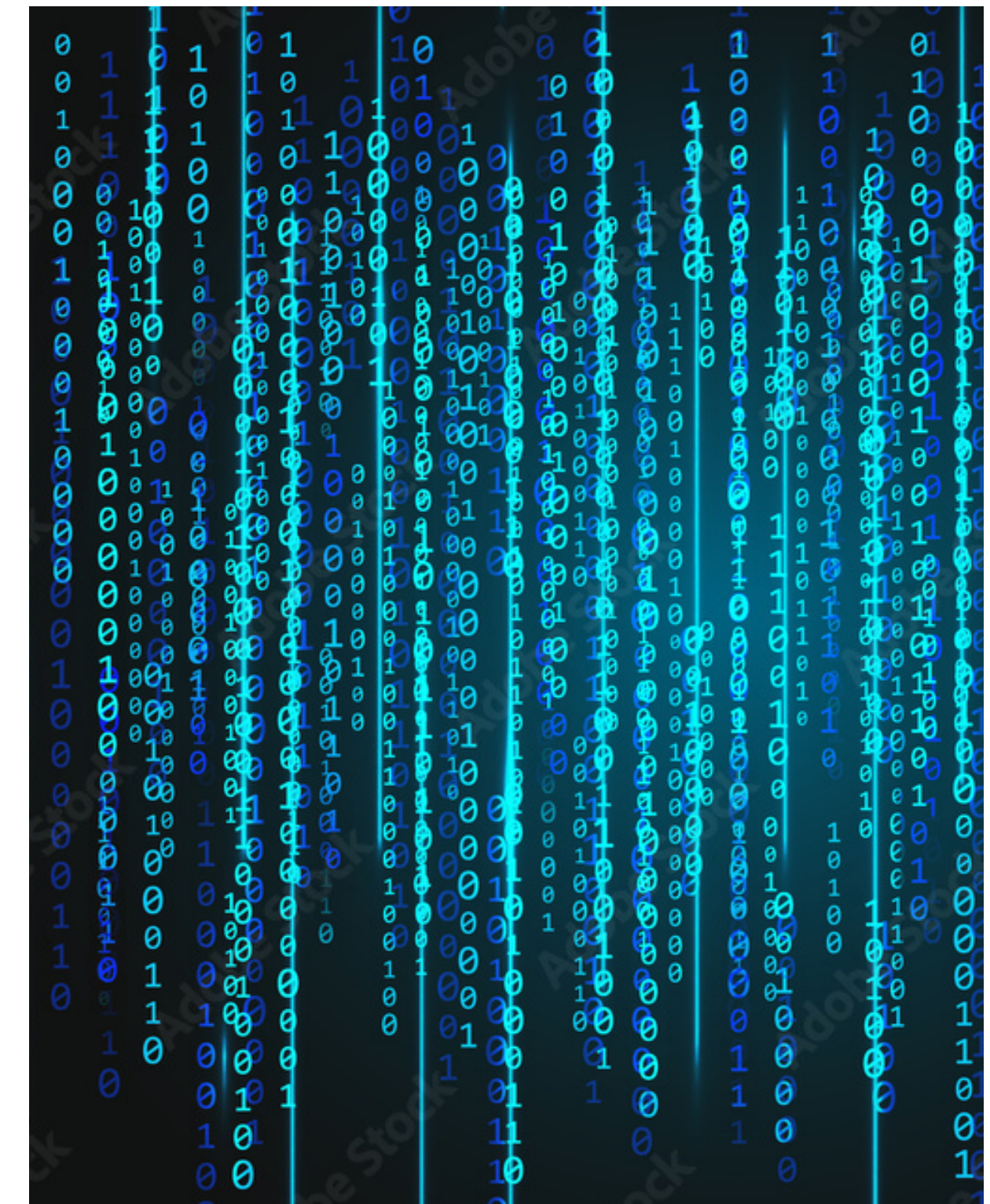
Code : "while (true) { printf("hello, world"); }"

Images,

Vidéos,

Applications,

...



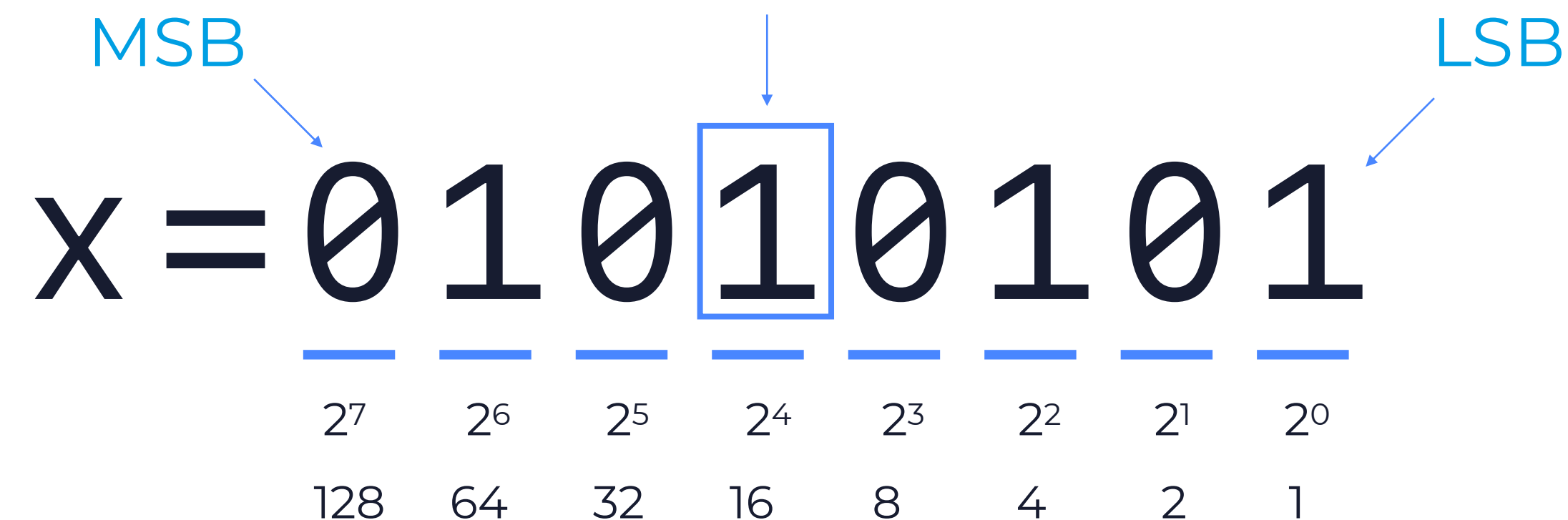
Encodage **bin**aire

Dans un processeur, on stocke **les données** sur un **nombre fini de bits** en mémoire.

Bit le plus significatif

Bit

Bit le moins significatif



Chaque position correspond à une puissance de deux.

Si $x \in \mathbb{N}$, $x = 64+16+4+1 = 85$ codé sur **8 bits**

Sur **n bits**, on peut stocker **2^n valeurs** différentes. Sur 8 bits, $2^8=256$ valeurs comprises entre 0000 0000 et 1111 1111.

Le nombre de bits est déterminé par le développeur (`int valeur=42;`) ou imposé par une norme (IEEE754).

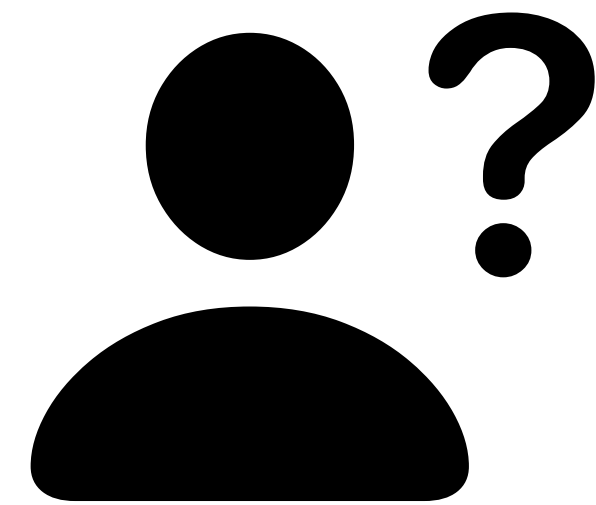
Nombre de bits

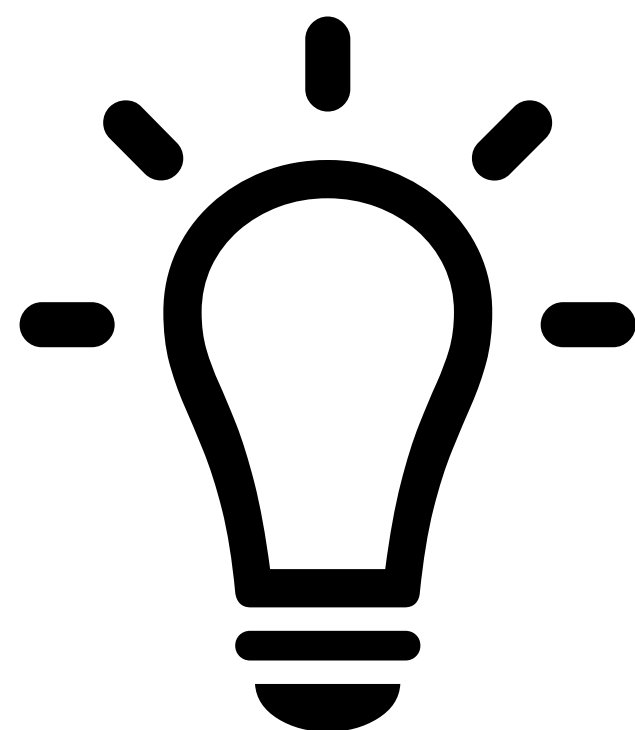
En théorie, on peut utiliser n'importe quel nombre de bits (3, 7, 13, ...)

En pratique, on se limite à des **profondeurs en puissances de 2** car cela permet une gestion efficace des données au niveau matériel.

1 bit	⇒ 2 valeurs	⇒ <code>bool</code> en C/C++
8 bits (1 octet)	⇒ 256 valeurs	⇒ <code>char</code> en C/C++
16 bits	⇒ 65 536 valeurs	⇒ <code>short</code> en C/C++
32 bits	⇒ 4 294 967 295 valeurs	⇒ <code>int</code> ou <code>float</code> en C/C++
64 bits	⇒ 18 446 744 073 709 551 616 valeurs ($1.8 * 10^{19}$)	⇒ <code>long</code> ou <code>double</code> en C/C++
128 bits	⇒ 340 282 366 920 938 463 463 374 607 431 768 211 456 valeurs ($= 3.4 * 10^{38}$)	⇒ <code>long double</code> en C/C++ (quadruple précision)

1 kilo-octet (1 ko)
1000 ou **1024** octets





1 kilo-octet = 1 ko = 10^3 octets = 1000 octets

1 kibi-octet = 1 Kio = 2^{10} octets = 1024 octets

Représentation des entiers positifs N

8 bits

$0b10010101 = 149$

Convention

<u>1</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128			16		4		1

$203 = 128 + 64 + 8 + 2 + 1$

$0b11001011$ 8 bits

<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>1</u>
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64			8		2	1

Additions binaires sur N

Cas très simples
à 1 bit

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

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

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

$$\begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$

Retenue

Dépassement de capacité

Cas à plusieurs
bit

$$\begin{array}{r} 110 \\ + 001 \\ \hline 111 \end{array}$$

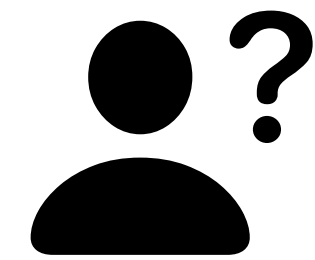
$$\begin{array}{r} 1010 \\ + 0011 \\ \hline 1101 \end{array}$$

$$\begin{array}{r} 1010001 \\ + 1010110 \\ \hline 10100111 \end{array}$$

Retenue

Dépassement de capacité

Représentation des entiers relatifs \mathbb{Z}



Comment représenter un nombre négatif sur N bits ?

①

MSB = signe
1 = nombre négatif
0 = nombre positif

Approche triviale

$$(-1)^{\text{MSB}} * x$$

②

Magnitude stockée sur N-1 bits en binaire naturel

Exemple sur 4 bits : $x = 5 = 1 * (4+1) = 0b0101$ $-5 = -1 * (4 + 1) = 0b1101$

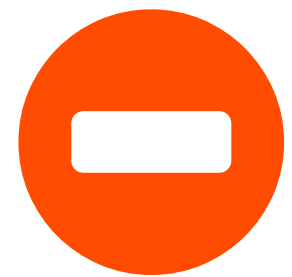
Représentation des entiers relatifs \mathbb{Z}

Approche triviale : **Signe** sur 1 bit et **Magnitude** codée sur n-1 bits



$$3 + 4 = 0b0011 + 0b0100 = 0b0111 = 7$$

OK



$$3 + 6 = 0b0011 + 0b0110 = 0b1001 = -1$$

Changement de signe (signe=1)

$$-3 + -4 = 0b1011 + 0b1100 = 0b10111 = 23 \text{ ou } 7$$

Dépassement de capacité, changement de signe (signe=0, retenue=1)

$$-3 + 4 = 0b1011 + 0b0100 = 0b1111 = -7$$

Pas de dépassement, toujours <0 (signe=1)

$$3 + -4 = 0b0011 + 0b1100 = 0b1111 = -7$$

Pas de dépassement, toujours <0 (signe=1)

$$5 + -4 = 0b0101 + 0b1100 = 0b10001 = 17 \text{ ou } 1$$

Dépassement de capacité (signe=0, retenue=1)

$$-5 + 4 = 0b1101 + 0b0100 = 0b10001 = 17 \text{ ou } 1$$

Dépassement de capacité (signe=0, retenue=1)

$$-3 + 3 = 0b1011 + 0b0011 = 0b1110 = -6$$

Pas de dépassement, toujours <0 (signe=1)

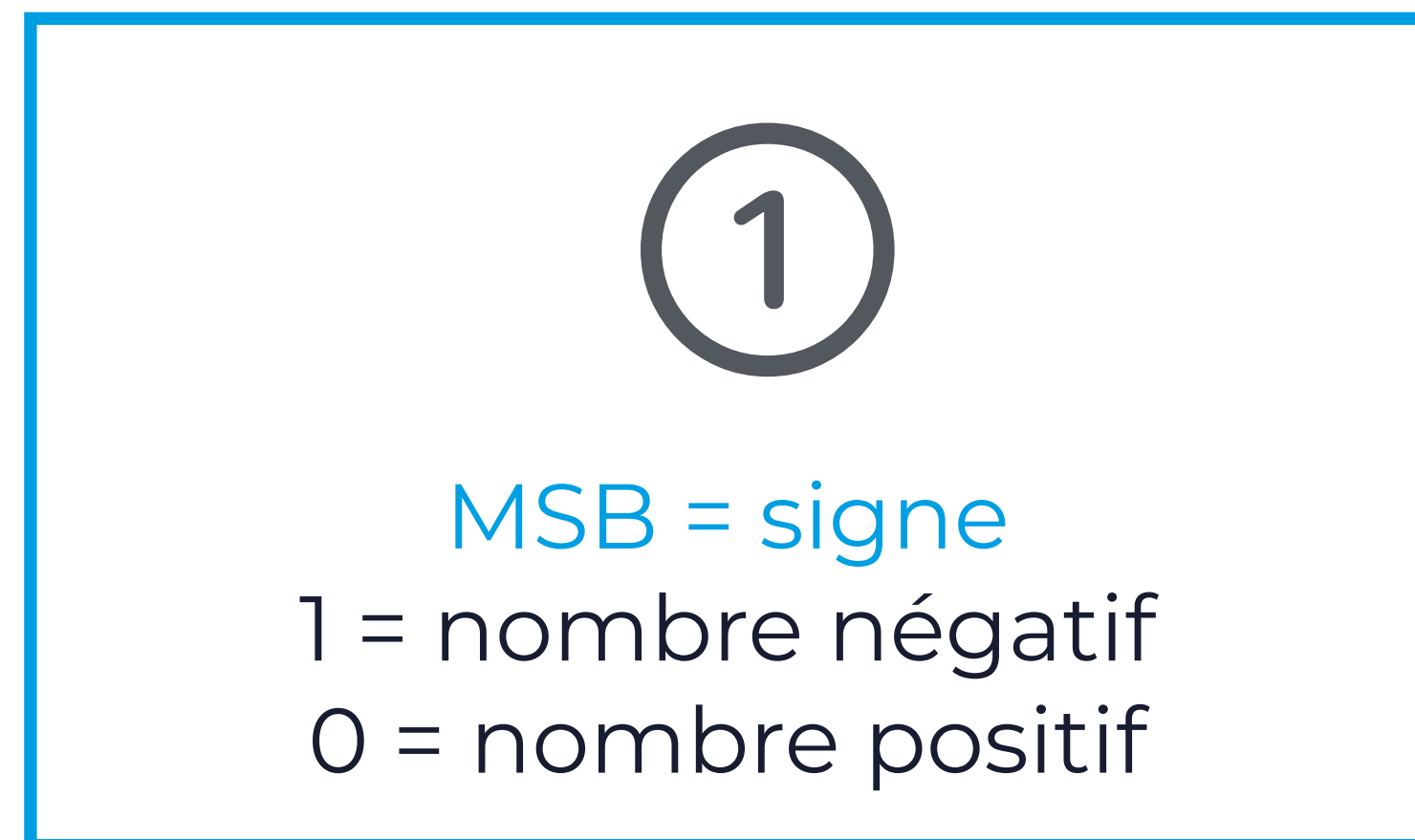
$$-4 + 4 = 0b1100 + 0b0100 = 0b10000 = 16 \text{ ou } 0$$

Dépassement de capacité, toujours >0 (signe=0, retenue=1)

2 représentations pour **0** = 0b**0000** et 0b**1000**

Représentation des entiers relatifs \mathbb{Z}

Le **complément à 1** offre une **symétrie binaire** entre nombre positifs et négatifs autour de 0, ce qui crée les conditions pour utiliser les opérations arithmétiques classiques.



Approche C1
 $2^N - 1 - x$
↓
Inverser tous
les bits



Exemple sur 4 bits : $x = 5 = 0b0101 \rightarrow C1 = 0b1010 \rightarrow -5 = 0b1010$ ($2^4 - 1 - 5$ en non signé)

$$-5 = 0b1010 = -2^{4-1} + 1 + 0*2^2 + 1*2^1 + 0*2^0 = -8 + 1 + 2$$

Représentation des entiers relatifs \mathbb{Z}

Approche C1 : **Signe** sur 1 bit et **Magnitude** codée sur n-1 bits en C1

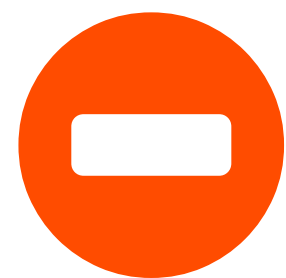


$$3 + 4 = 0b0011 + 0b0100 = 0b0111 = 7 \quad \text{OK}$$

$$-4 + 3 = 0b1011 + 0b0011 = 0b1110 = -1 \quad \text{OK}$$

$$-5 + 4 = 0b1010 + 0b0100 = 0b1110 = -1 \quad \text{OK}$$

$$-3 + 3 = 0b1100 + 0b0011 = 0b1111 = 0 \quad \text{OK}$$



$$3 + 6 = 0b0011 + 0b0110 = 0b1001 = -1$$

Changement de signe (signe=1)

$$-3 + -6 = 0b1100 + 0b1001 = 0b10101 = 21 \text{ ou } 5$$

Dépassement de capacité et changement de signe (signe=0, retenue=0)

$$4 + -3 = 0b0100 + 0b1100 = 0b10000 = 16 \text{ ou } 0$$

Dépassement de capacité (signe=0, retenue=1)

$$-3 + -4 = 0b1100 + 0b1011 = 0b10111 = 23 \text{ ou } 7$$

Dépassement de capacité (signe=0, retenue=1)

$$-2 + -4 = 0b1101 + 0b1011 = 0b11000 = 24 \text{ ou } -7$$

Dépassement de capacité (signe=1, retenue=1)

$$5 + -4 = 0b0101 + 0b1011 = 0b10000 = 16 \text{ ou } 0$$

Dépassement de capacité (signe=0, retenue=1)

2 représentations pour 0 = 0b0000 et 0b1111

Représentation des entiers relatifs \mathbb{Z}

Approche C1 : **Signe** sur 1 bit et **Magnitude** codée sur n-1 bits en C1

En complément à 1, lorsqu'un dépassement se produit après une addition, il faut **négliger ce bit de dépassement et l'ajouter** au résultat pour obtenir la valeur correcte.

$$4 + -3 = 0b0100 + 0b1100 = 0b10000 = 16 \text{ ou } 0$$

$$-3 + -4 = 0b1100 + 0b1011 = 0b10111 = 23 \text{ ou } 7$$

$$-2 + -4 = 0b1101 + 0b1011 = 0b11000 = 24 \text{ ou } -7$$

$$5 + -4 = 0b0101 + 0b1011 = 0b10000 = 16 \text{ ou } 0$$



$$\begin{aligned} 0b\boxed{1}0000 &\rightarrow 0b\mathbf{0}000 + \boxed{1} = 0b\mathbf{0}001 = \mathbf{1} \\ 0b\boxed{1}0111 &\rightarrow 0b\mathbf{0}111 + \boxed{1} = 0b\mathbf{1}000 = \mathbf{-7} \\ 0b\boxed{1}1000 &\rightarrow 0b\mathbf{1}000 + \boxed{1} = 0b\mathbf{1}001 = \mathbf{-6} \\ 0b\boxed{1}0000 &\rightarrow 0b\mathbf{0}000 + \boxed{1} = 0b\mathbf{0}001 = \mathbf{1} \end{aligned}$$

En complément à 1, les nombres sont représentés sur un cercle où les nombres positifs et négatifs coexistent. Ajouter le bit de dépassement au résultat revient à **reconstituer la "boucle" cyclique** de la représentation des nombres relatifs.

Cela complique les opérations arithmétiques (et les circuits), car cela nécessite une étape supplémentaire pour gérer le bit de dépassement.

Représentation des entiers relatifs \mathbb{Z}

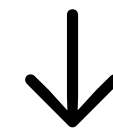
Le **complément à 2** est une amélioration directe du complément à 1 en offrant une **symétrie parfaite autour de 0**, ce qui permet d'utiliser sans modification les opérations arithmétiques classiques.

①

MSB = signe
1 = nombre négatif
0 = nombre positif

Approche C2

$$2^N - x$$



Inverser tous
les bits et
ajouter 1

②

Magnitude stockée sur
N-1 bits en
Complément à 2

Exemple sur 4 bits : $x = 5 = 0b0101 \rightarrow C1 = 0b1010 \rightarrow C2 = C1 + 1 = 0b1011$

$-5 = 0b1011$ (égal à $2^4 - 5$ en non signé - rappel $C1 = 2^4 - 1 - 5$)

$$-5 = 0b1011 = -2^{4-1} + 0*2^2 + 1*2^1 + 1*2^0 = -8 + 2 + 1$$

Représentation des entiers relatifs \mathbb{Z}

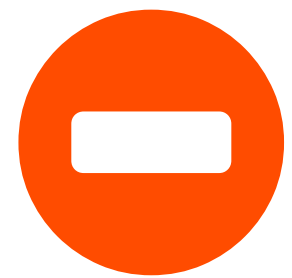
Approche C2 : **Signe** sur 1 bit et **Magnitude** codée sur n-1 bits en C2



$$3 + 4 = 0b0011 + 0b0100 = 0b0111 = 7 \quad \text{OK}$$

$$-4 + 3 = 0b1100 + 0b0011 = 0b1111 = -1 \quad \text{OK}$$

$$-5 + 4 = 0b1011 + 0b0100 = 0b1111 = -1 \quad \text{OK}$$



$$3 + 6 = 0b0011 + 0b0110 = 0b1001 = -1$$

Changement de signe

$$-3 + 3 = 0b1101 + 0b0011 = 0b10000 = 16 \text{ ou } 0$$

Dépassement de capacité (retenue=1)

$$4 + -3 = 0b0100 + 0b1101 = 0b10001 = 17 \text{ ou } 1$$

Dépassement de capacité (retenue=1)

$$-3 + -4 = 0b1101 + 0b1100 = 0b11001 = 25 \text{ ou } -7$$

Dépassement de capacité (retenue=1)

$$-2 + -4 = 0b1110 + 0b1100 = 0b11010 = 26 \text{ ou } -6$$

Dépassement de capacité (retenue=1)

$$5 + -4 = 0b0101 + 0b1100 = 0b10001 = 17 \text{ ou } 1$$

Dépassement de capacité (retenue=1)

2 représentations pour 0 = 0b0000 et 0b10000

Dépassement de capacité (retenue=1)



Si on **néglige le dépassement de capacité** (retenue), les calculs sont justes et il existe une seule représentation de 0.

Représentation des entiers relatifs \mathbb{Z}

Complément à 1

$$-4 + 3 = 0b1011 + 0b0011 = 0b1110 = -1$$

$$4 + -3 = 0b0100 + 0b1100 = 0b10000$$
$$0b10000 \rightarrow 0b0000 + 1 = 0b0001 = 1$$

$$-3 + -4 = 0b1100 + 0b1011 = 0b10111$$
$$0b10111 \rightarrow 0b0111 + 1 = 0b1000 = -7$$

Complément à 2

$$-4 + 3 = 0b1100 + 0b0011 = 0b1111 = -1$$

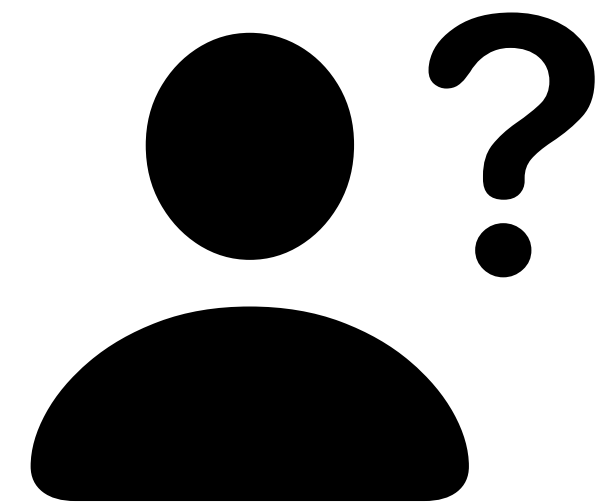
$$4 + -3 = 0b0100 + 0b1101 = 0b10001 \rightarrow 0b0001 = 1$$

$$-3 + -4 = 0b1101 + 0b1100 = 0b11001 \rightarrow 0b1001 = -7$$

Le complément à 2 **intègre naturellement les nombres négatifs** dans les règles habituelles de l'addition et de la soustraction binaire et offre une **représentation unique pour 0**.

En ignorant le bit de report, les additionneurs binaires en complément à 2 **sont plus simples à implémenter matériellement**, car ils traitent les nombres positifs et négatifs de la même manière.

Quelle est la **valeur décimale**
de $x = 0b1100\ 1011$?



Valeur de 0b11001011 ?

0b11001011 = 203

En entiers non signés (\mathbb{N})
 $0 \leq x \leq 255$ sur 8 bits (256 valeurs)

0b11001011 = -53

En entiers signés (\mathbb{Z})
 $-128 \leq x \leq 127$ sur 8 bits (256 valeurs)
-128 = 0b1000 0000 127 = 0b0111 1111

La seule différence est le MSB : 2^{N-1} en non signé et -2^{N-1} en signé

Nous ne pouvons pas savoir a priori ce qu'un nombre binaire signifie. Cela dépend du format choisi par l'utilisateur (int ou unsigned int en C/C++ par exemple).

Encodage hexadécimal

L'hexadécimal est une base comportant 16 symboles : 0 ... 9 et A ... F

Un symbole hexadécimal représente 4 bits, ce qui compacte l'écriture et facilite la lecture d'un nombre binaire.

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

8 bits $0b$ $\boxed{01011101}$ = $0x5D$
 2^3 2^2 2^1 2^0 2^3 2^2 2^1 2^0
 93 en décimal

16 bits $0x$ $\underline{5}$ \underline{A} \underline{C} $\underline{3}$ = 23235
 $5 \cdot 16^3$ $10 \cdot 16^2$ $12 \cdot 16^1$ $3 \cdot 16^0$
 $0b0101101011000011$

$0x$ \underline{C} A F E = 51966 ou -13570
 1100 en non signé en signé

Représentation des réels \mathbb{R}

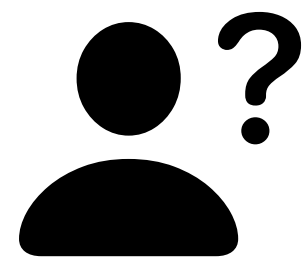
1 2 3 4 . 5 6 7 8 9

$$= 1000 + 200 + 30 + 4 + 0.5 + 0.06 + 0.007 + 0.0008 + 0.00009$$

1 2 3 4 . 5 6 7 8 9
 10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} 10^{-4} 10^{-5}

Partie entière

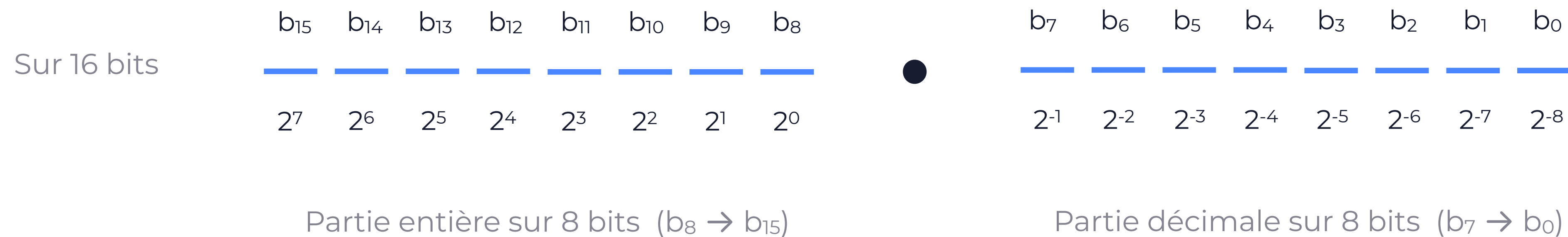
Partie décimale



Comment représenter ce **nombre réel** sur N bits ?

Représentation des réels \mathbb{R}

Première idée : Prendre la **moitié des bits** pour la partie entière et l'autre moitié pour la partie décimale (principe de la virgule fixe).

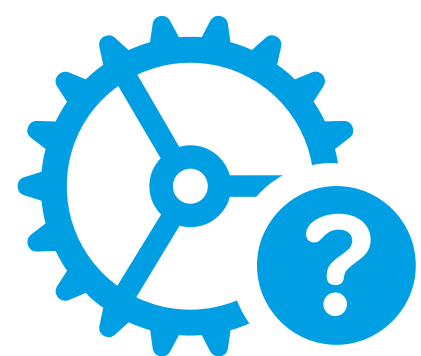


0 → 255

1/256 → 255/256

0,00390625

0,99609375



Approche très limitée :

- Sur 16 bits : Maximum = 255.996 et Précision = 1/256
- Sur 32 bits : Maximum = 65535.999984 et Précision = 1/65536

Sur 32 bits : 1234 = 0b0000 0100 1101 0010 et .56789 = 0b 1001 0001 0110 0001 = 0,5678863525

Représentation des réels \mathbb{R}

Deuxième idée : S'inspirer de la notation scientifique

1234.56789



(**signe**) caractéristique. mantisse x base^{exposant}

Norme IEEE 754

Norme sur l'arithmétique des **nombre réels à virgule flottante** (version initiale 1985 pour le 32/64 bits, version révisée 2008 pour le 128 bits).

Inspirée de la notation scientifique : (signe) caractéristique. mantisse x base^{exposant}



L'exposant donne la position (flottante) de la virgule.

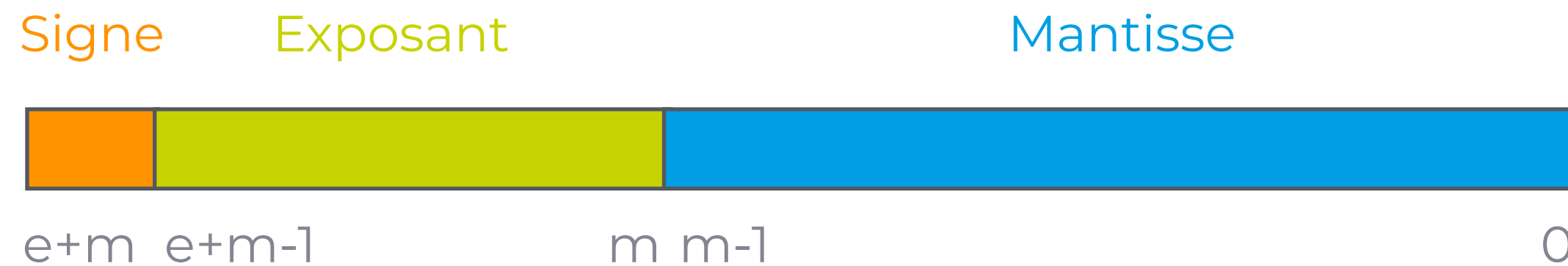
L'exposant du nombre est positif ou négatif.

Utilisation d'un biais de $2^{e-1} - 1$ pour décaler les valeurs négatives vers des valeurs positives.

$$exposant_{biaise} = exposant_{reel} + 2^{e-1} - 1$$

Exposant biaisé stocké sur e bits.

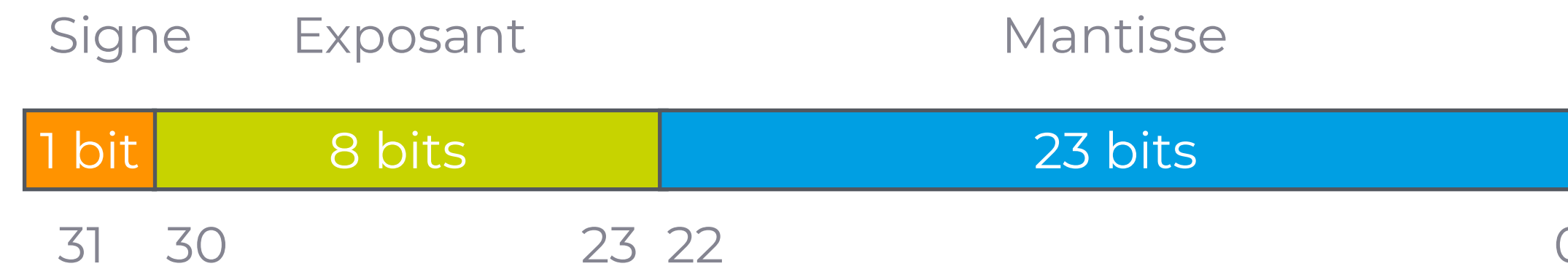
Nombre stocké sur e+m+1 bits



Norme IEEE 754

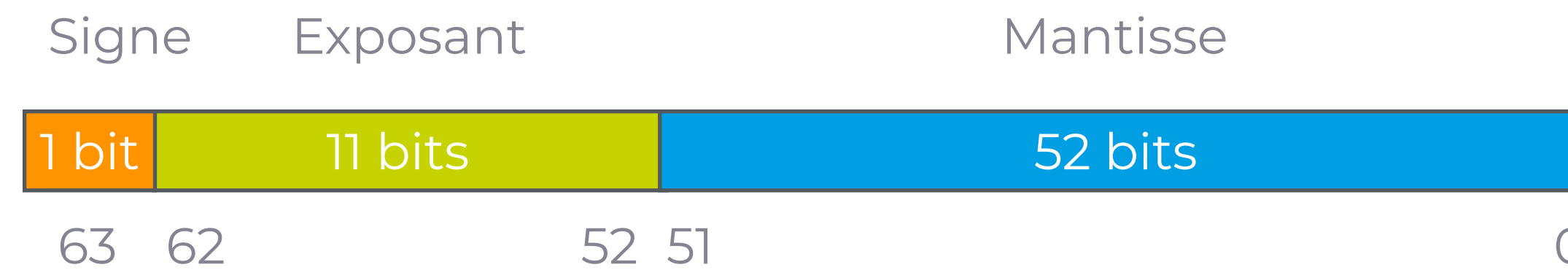
(signe) 1. mantisse x 2^{exposant}

32 bits
Simple précision
(float en C/C++)
m=23, e=8



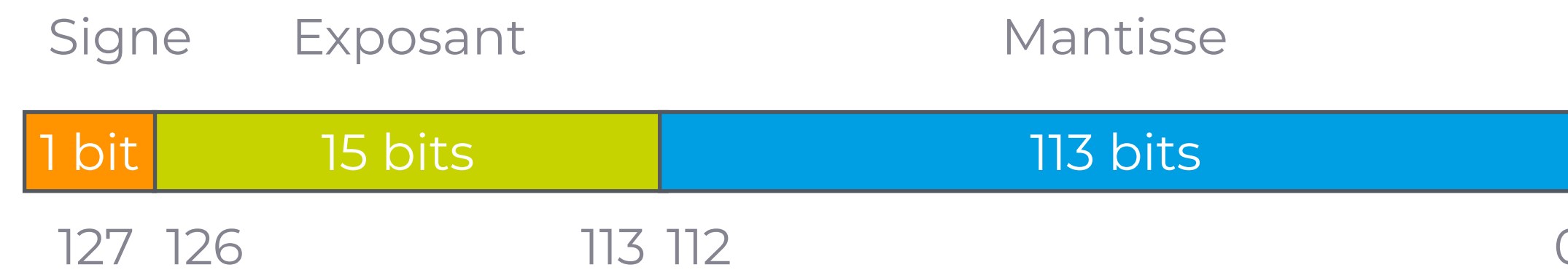
Exposant décalé de
 $2^{e-1} - 1 = 2^7 - 1 = 127$
 $-127 \leq \text{exposant}_{reel} \leq 128$
 $0 \leq \text{exposant}_{biaise} \leq 255$

64 bits
Double précision
(double en C/C++)
m=52, e=11



Exposant décalé de
 $2^{e-1} - 1 = 2^{10} - 1 = 1023$
 $-1023 \leq \text{exposant}_{reel} \leq 1024$
 $0 \leq \text{exposant}_{biaise} \leq 2047$

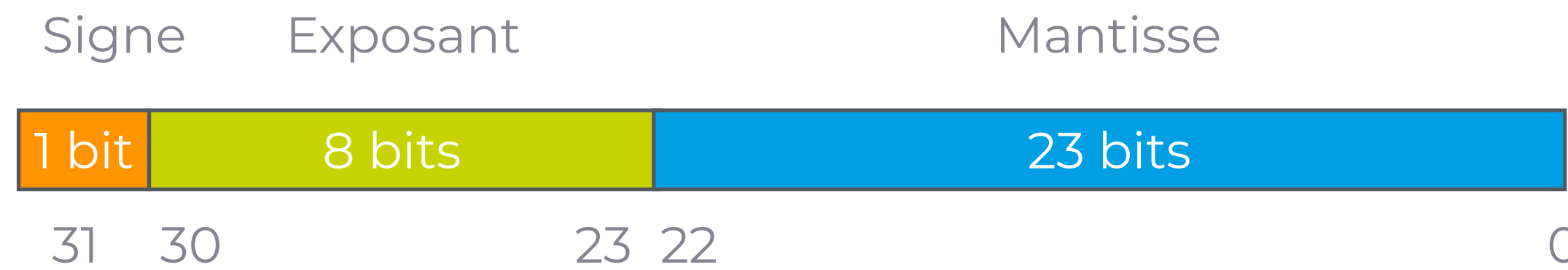
128 bits
Quadruple précision
(long double en C/C++)
m=113, e=15



Exposant décalé de
 $2^{e-1} - 1 = 2^{14} - 1 = 16383$
 $-16383 \leq \text{exposant}_{reel} \leq 16384$
 $0 \leq \text{exposant}_{biaise} \leq 32767$

Cas particuliers de la norme IEEE 754

32 bits
Simple précision
(float en C/C++)
m=24, e=8



Exposant décalé de
 $2^{e-1} - 1 = 2^7 - 1 = 127$
 $-127 \leq \text{exposant}_{reel} \leq 128$
 $0 \leq \text{exposant}_{biaise} \leq 255$

+0 = 0000 0000 0000 0000 0000 0000 0000 0000 = 0x0000 0000

-0 = 1000 0000 0000 0000 0000 0000 0000 0000 = 0x8000 0000

$+\infty$ = 0111 1111 1000 0000 0000 0000 0000 0000 = 0x7F80 0000

$-\infty$ = 1111 1111 1000 0000 0000 0000 0000 0000 = 0xFF80 0000

NaN = s111 1111 1 mantisse avec mantisse!≠ 0 et s indifférent

$+\infty$ et $-\infty$ sont utilisés pour représenter les résultats d'opérations dépassant les limites numériques (1/0) et pour gérer les limites sans générer des erreurs critiques dans les calculs.

NaN est une représentation essentielle pour traiter les résultats indéfinis ou non valides (0/0, sqrt(-1), ...) , tout en maintenant la continuité des calculs sans générer d'erreurs fatales

Valeur de 0x411A 0000 ?

32 bits
Simple précision
m=24, e=8

4 1 1 A 0 0 0 0
0100 0001 0001 1010 0000 0000 0000 0000

Signe = 0 → Nombre positif

Exposant = 0b1000 0010 = $2^7 + 2^1 = 128 + 2 = 130 \rightarrow e = 130 - 127 = 3$

Mantisse = 0b0011 0100 0000 ... 0000 = $2^{-3} + 2^{-4} + 2^{-6}$
= $0.125 + 0.0625 + 0.015625 = 0.203125$

$$(+)\ 1.203125 \times 2^3 = 9.625$$

Représentation de 5.75 et -5.75

Méthode 1

Partie entière : $5 = 2^2 + 2^0 = 0b101$

Partie décimale : $0.75 = 2^{-1} + 2^{-2} = 0b11$

$5.75 = 0b101.11 = 0b1.0111 * 2^2$

Exposant = $127 + 2 = 129 = 0b1000\ 0001$

Méthode 2

$5.75 = 1.4375 * 2^2$

Exposant = $127 + 2 = 129 = 0b1000\ 0001$

Partie décimale : $0.4375 = 2^{-2} + 2^{-3} + 2^{-4} = 0b0111$

Exposant = $127 + 2 = 129 = 0b1000\ 0001$

Mantisse = $0b0111\ 0000\ \dots\ 0000$

5.75 0100 0000 1011 1000 0000 0000 0000 0000

4 0 B 8 0 0 0 0

-5.75 1100 0000 1011 1000 0000 0000 0000 0000

C 0 B 8 0 0 0 0

Représentation de $\pi \approx 3.1415926$

Partie entière : $3 = 2^1 + 2^0 = 0b11$

Partie décimale : $0.1415926 \approx 0b0010\ 0100\ 0011\ 1111\ 0110\ 100$

$\pi = 0b11.0010\ 0100\ 0011\ 1111\ 0110\ 1 = 0b1.1001\ 0010\ 0001\ 1111\ 1011\ 01 * 2$

Exposant = $127 + 1 = 128 = 0b1000\ 0000$

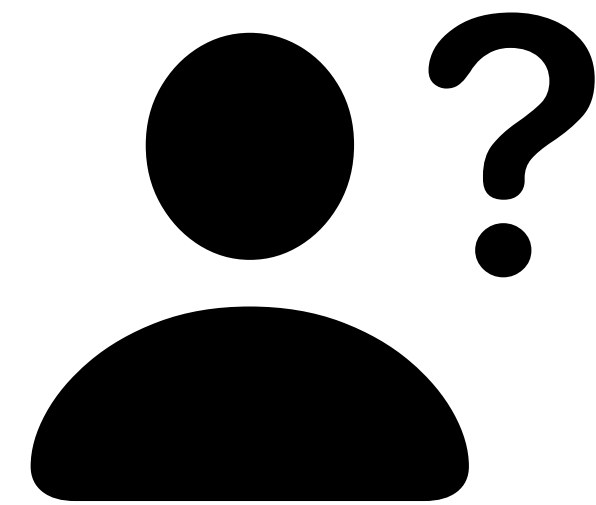
Mantisse = $0b1001\ 0010\ 0001\ 1111\ 1011\ 010$

π	0100	0000	0100	1001	0000	1111	1101	1010
	4	0	4	9	0	F	D	A

Valeur stockée : 3.141592502593994140625

Erreur de conversion : 0.0000000097406005859375

Comment faire une **addition/soustraction** de **nombre réels codés** en **IEEE754**



Comment faire une **addition/soustraction** de **nombre réels codés** en **IEEE754**



1. Extraire le signe S , la mantisse M et l'exposant E de chaque nombre
2. Aligner les exposants en décalant les mantisses
3. Additionner ou soustraire les mantisses selon le signe
4. Normaliser la mantisse et l'exposant
5. Encoder le résultat en IEEE754

Addition de 3.5 et 1.25

Etape 1 : Extraire le signe, mantisse, exposant

$$3.5 = 1.75 * 2^1$$

$$\text{Signe} = 0$$

$$\text{Exposant} = 127 + 1 = 128 = 0b1000\ 0000$$

$$\text{Mantisse} : 0.75 = 2^{-1} + 2^{-2} = 0b11$$

$$3.5 = 0b0100\ 0000\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000 = 0x40600000$$

$$1.25 = 1.25 * 2^0$$

$$\text{Signe} = 0$$

$$\text{Exposant} = 127 + 0 = 127 = 0b0111\ 1111$$

$$\text{Mantisse} : 0.25 = 2^{-2} = 0b01$$

$$1.25 = 0b0011\ 1111\ 1010\ 0000\ 0000\ 0000\ 0000\ 0000 = 0x3FA00000$$

Addition de 3.5 et 1.25

Etape 2 : Aligner les exposants en décalant les mantisses

$$3.5 = 1.75 * 2^1$$

Signe = 0

$$\text{Exposant} = 127 + 1 = 128 = 0b1000\ 0000$$

$$\text{Mantisse} : 0.75 = 2^{-1} + 2^{-2} = 0b11$$

$$\rightarrow 3.5 = 0b1.11 * 2^1$$

$$1.25 = 1.25 * 2^0$$

Signe = 0

$$\text{Exposant} = 127 + 0 = 127 = 0b0111\ 1111$$

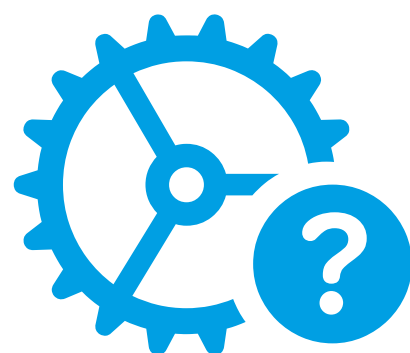
$$\text{Mantisse} : 0.25 = 2^{-2} = 0b01$$

$$\rightarrow 1.25 = 0b1.01 * 2^0$$

Exposant_{3.5} = 128 et Exposant_{1.25} = 127 non alignés → Décalage 1 bit à gauche (*2) → Exposant_{1.25} = 128

Décalage 1 bit à droite (/2) → Mantisse_{1.25} = 0b0.101 = 2⁻¹ + 2⁻³ = 0.625 car 1.25 = 0.625 * 2¹

$$\rightarrow 1.25 = 0b0.101 * 2^1$$



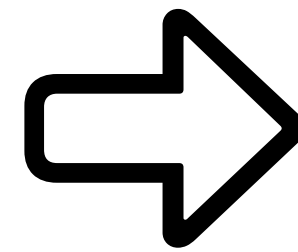
On aligne toujours l'exposant le plus petit en faisant un décalage vers la droite de la mantisse pour ne pas perdre de précision (perte du MSB de 0b11 → 0b1 en décalant à gauche)

Addition de 3.5 et 1.25

Etape 3 : Additionner les nombres alignés

$$3.5 = 0b1.11 * 2^1$$

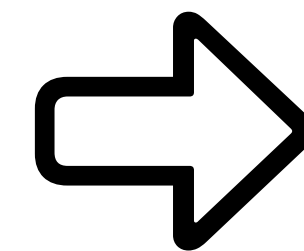
$$1.25 = 0b0.101 * 2^1$$



$$\begin{array}{r} 1.110 * 2^1 \\ + 0.101 * 2^1 \\ \hline 10.011 * 2^1 \end{array}$$

Etape 4 : Normaliser la mantisse et l'exposant

$$0b10.011 * 2^1 = 0b1.0011 * 2^2$$



Mantisse = 0b0011

Exposant = 127 + 2 = 129

Etape 5 : Encoder le résultat en IEEE754

0100 0000 1001 1000 0000 0000 0000 0000
4 0 9 8 0 0 0 0

Vérification

$$3.5 + 1.25 = 4.75$$

$$4.75 = 1.1875 * 2^2$$

$$\text{Exposant} = 127 + 2 = 129 = 0b1000\ 0001$$

$$\text{Partie décimale} : 0.1875 = 2^{-3} + 2^{-4} = 0b0011$$

0100 0000 1001 1000 0000 0000 0000 0000

4 0 9 8 0 0 0 0

Soustraction de 5.5 et 2.75

Etape 1 : Extraire le signe, mantisse, exposant

$$5.5 = 1.375 * 2^2$$

Signe = 0

Exposant = $127 + 2 = 129 = 0b1000\ 0001$

Mantisse : $0.375 = 2^{-2} + 2^{-3} = 0b011$

$$2.75 = 1.375 * 2^1$$

Signe = 0

Exposant = $127 + 1 = 128 = 0b1000\ 0000$

Mantisse : $0.375 = 2^{-2} + 2^{-3} = 0b011$

Etape 2 : Aligner les exposants en décalant les mantisses

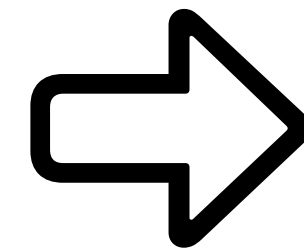
$$5.5 = 1.375 * 2^2 \rightarrow 0b1.011 * 2^2$$

$$2.75 = 1.375 * 2^1 \rightarrow 0b1.011 * 2^1 \rightarrow 0b0.1011 * 2^2$$

Soustraction de 5.5 et 2.75

Etape 3 : Soustraire les nombres alignés

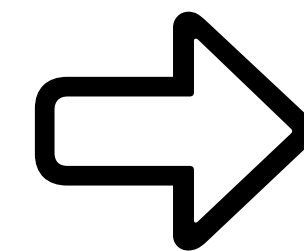
$$5.5 = 0b1.011 * 2^2$$
$$2.75 = 0b0.1011 * 2^2$$



$$\begin{array}{r} 1.0110 * 2^2 \\ - 0.1011 * 2^2 \\ \hline 0.1011 * 2^2 \end{array}$$

Etape 4 : Normaliser la mantisse et l'exposant

$$0b0.1011 * 2^2 = 0b1.011 * 2^1$$



Mantisse = 0b011

Exposant = 127 + 1 = 128

Etape 5 : Encoder le résultat en IEEE754

S=0 car >0

0100 0000 0011 0000 0000 0000 0000 0000

4 0 3 0 0 0 0 0

Soustraction de 5.5 et 2.75

Etape 3 : Additionner avec le complément à 2

$$5.5 - 2.75 = 5.5 + -2.75$$

$$5.5 = 0b1.011 * 2^2$$

$$2.75 = 0b0.1011 * 2^2 \rightarrow -2.75 = 0b1.0101 * 2^2$$

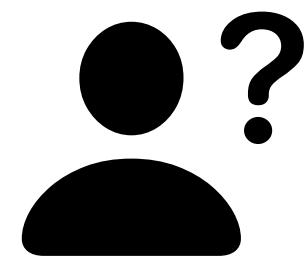
$$\begin{array}{r} 1.0110 * 2^2 \\ + 1.0101 * 2^2 \\ \hline \end{array}$$

$$(1)0.1011 * 2^2$$

On néglige le dépassement et on obtient le même résultat que précédemment.

Représentation des caractères

Hello, world



Comment représenter cet ensemble de caractères sur N bits ?

Table ASCII

American Standard Code for Information Interchange

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Table reliant un caractère à une valeur entière comprise entre 0x00 et 0x7F sur 7 bits à l'origine.

'A' = 0x41 = 65

'a' = 0x61 = 97

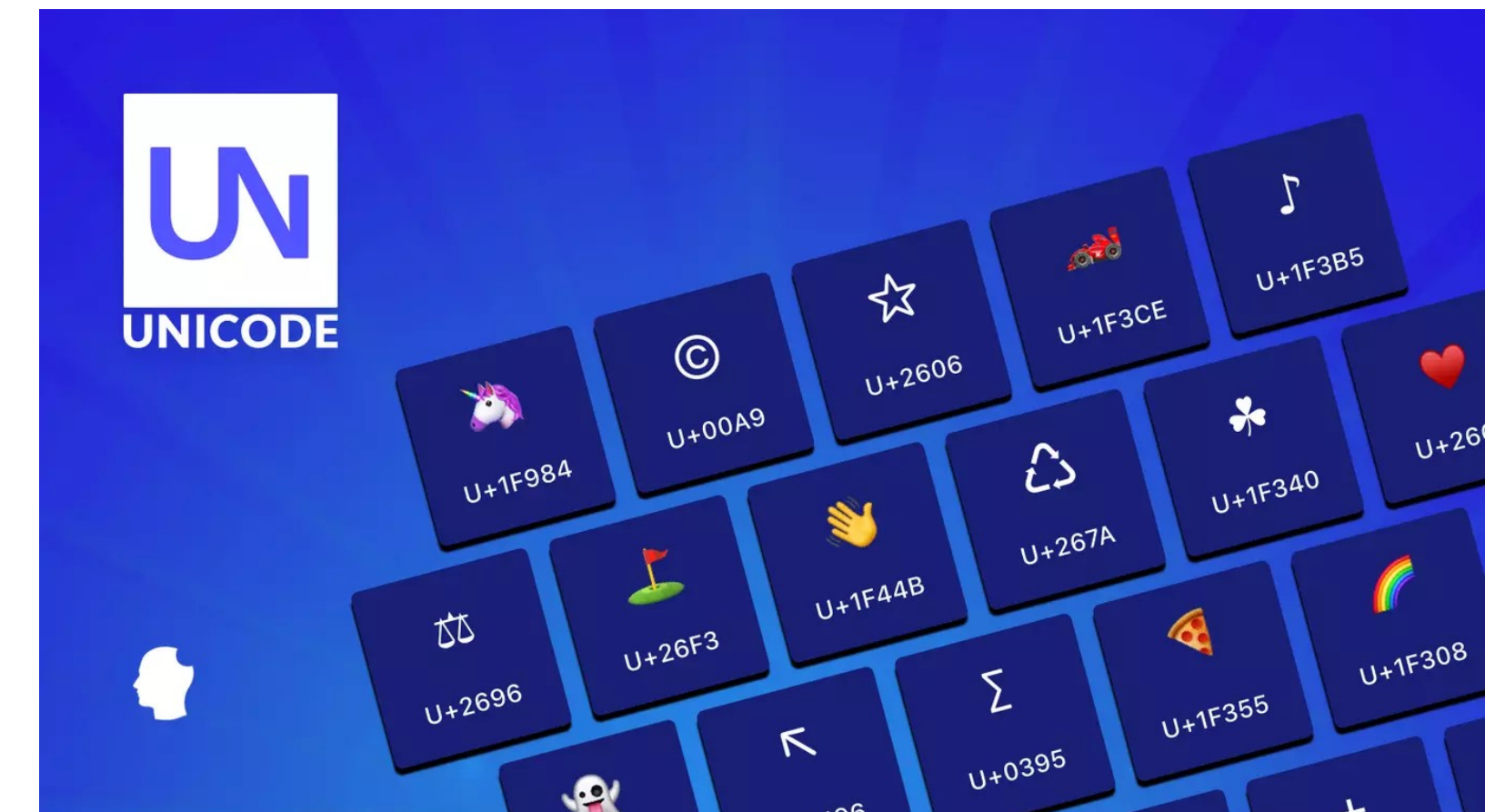
'4' = 0x34

"hello, world" =
0x68656C6C6F2C20776F
726C64

UTF

Universal Character Set Transformation Format

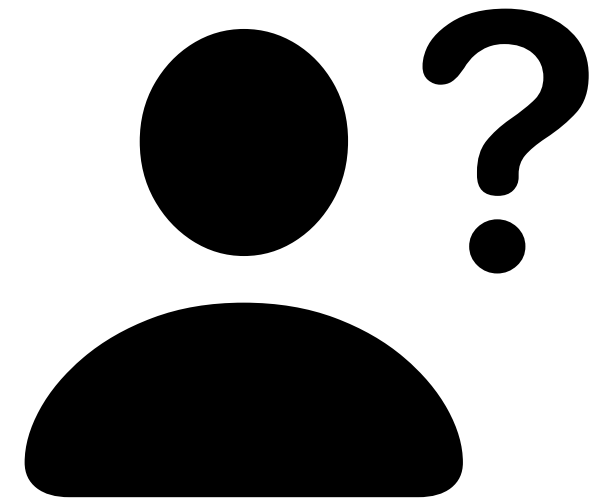
	H	e	l	l	o
Unicode	U+0048	U+0065	U+006C	U+006C	U+006F
UTF-8	48	65	6C	6C	6F
UTF-16	00 48	00 65	00 6C	00 6C	00 6F
ASCII	48	65	6C	6C	6F



Extension de l'ASCII pour encoder l'ensemble des caractères du « répertoire universel de caractères codés » (Unicode)

Utilise de 1 à 4 octets (UTF-8 → UTF-32)

Quelle est la **valeur**
de $x = 0x416C\ 6C6F$ sur 32 bits ?



Valeur de `0x416C 6C6F` ?

= `1097624687` en entiers non signés (\mathbb{N}) ou signés (\mathbb{Z})

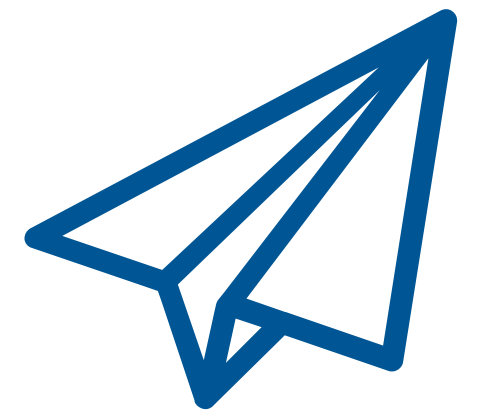
= `14.7764` en réel (\mathbb{R})

= `allo` en chaîne de caractères

Nous ne pouvons pas savoir a priori ce qu'un nombre binaire signifie. Cela dépend du **format choisi** par l'utilisateur (`int`, `float` ou `string` en C++ par exemple).

#2

TAKE HOME MESSAGE



Tout est binaire

Dans un ordinateur, tout, absolument tout, est stocké en format binaire.



Nombre de bits prédéterminé

On utilise un nombre fini et pré-déterminé de bits pour représenter de l'information.



Encodage/décodage spécifique selon les formats

C'est l'encodage/décodage des données qui définit le type de données stockées ou lues sur N bits

Questions





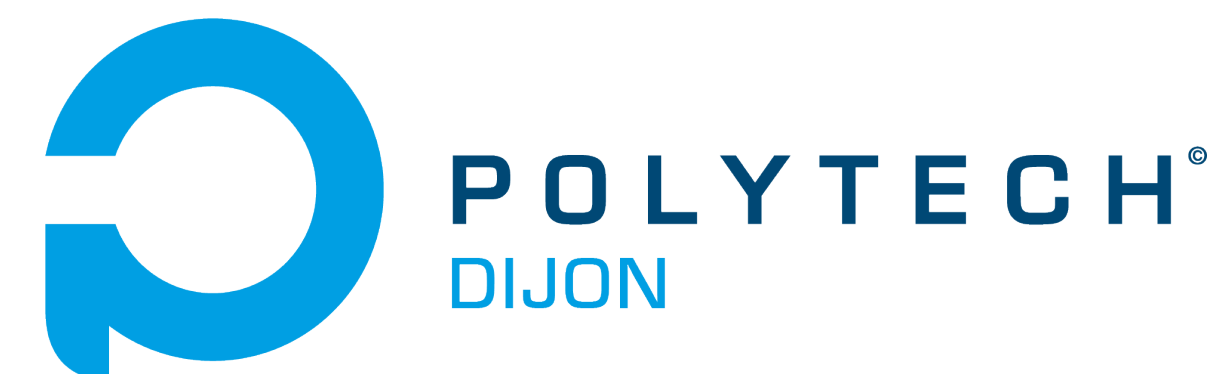
Contact

Pr. Dominique Ginhac

dginhac@ube.fr

Retrouvez toutes les infos sur :

<https://github.com/dginhac/polytech-dijon-archi>



This work is **licensed** under a
Creative Commons Attribution-NonCommercial 4.0 International License.

<https://creativecommons.org/licenses/by-nc/4.0/>

