heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

ARO-1

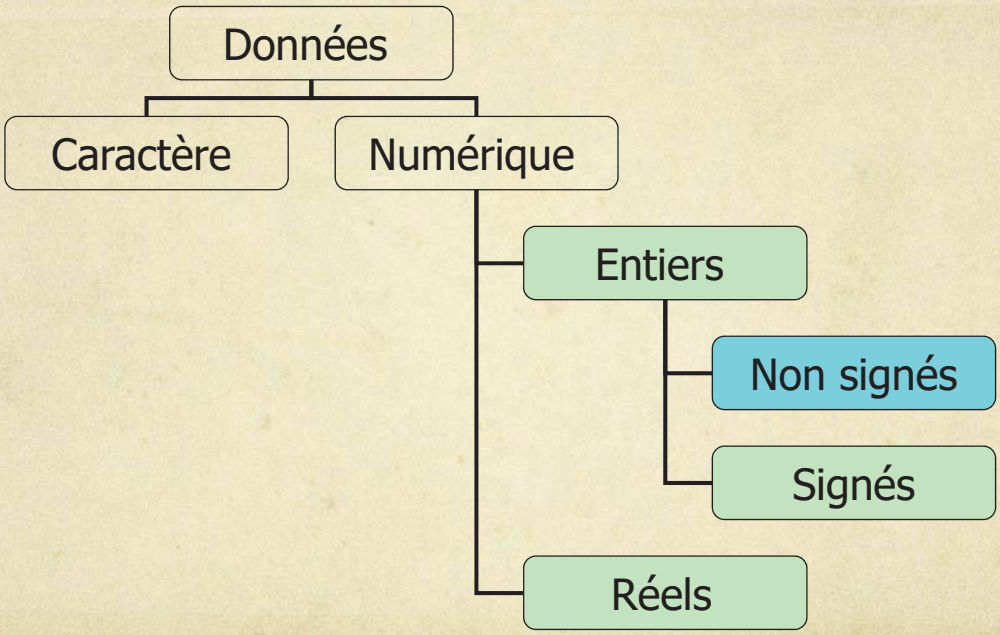
Représentation des nombres

Profs. Peña & Perez-Uribe & Mosqueron
Basé sur le cours du Prof. E. Sanchez

Polycopié : Electronique numérique

- Arithmétique binaire pages 21 à 34
  - Addition binaire
  - Nombres signés C1 et C2
  - Addition et soustraction en C2
  - Addition en BCD

Représentation des données



Représentation des nombres entiers

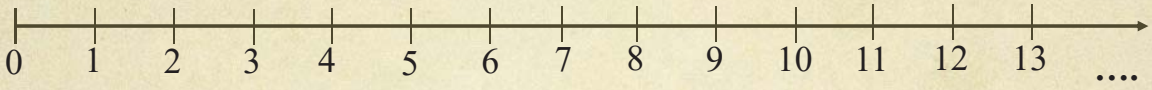
- Les instructions des ordinateurs traitent des nombres de taille fixe : 4, 8, 16, 32 ou 64 bits
- Avec un nombre composé de n bits, on ne peut représenter que 2^n valeurs entières différentes
- On souhaite disposer de valeurs positives et de valeurs négatives
- On souhaite pouvoir réaliser les 4 opérations arithmétiques (add, sub, mul, div) de la façon la plus simple possible

Représentation des nombres entiers

- Nombre entier non signé
  - Représentation binaire standard, voir chapitre précédent.
- Nombre entier signé
  - Choisir une représentation du signe :
    - A la main on utilise le signe "–" qui précède le nombre positif
    - dans le tableur Excel, la couleur Rouge est parfois utilisé
    - .....
  - En électronique numérique on a dédié un bit: le "**bit de signe**" .
    Il existe plusieurs représentations :
    - signe & valeur absolue, complément à 1, complément à 2, biaisée, ...
    - la représentation la plus utilisée est le complément à 2

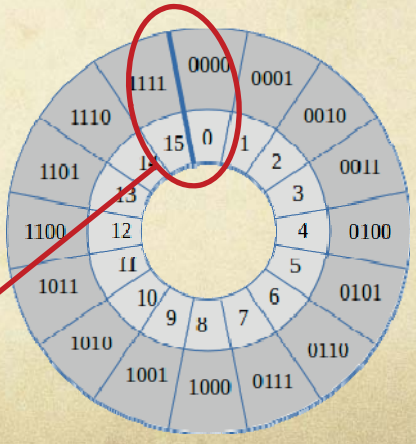
Nombres entiers non-signés

- Les nombres peuvent être représentés sur une droite:



- En informatique: nombres représentés sur N bits, plage limitée!

représentation sur un cercle
=> risque de débordement !



Débordement !

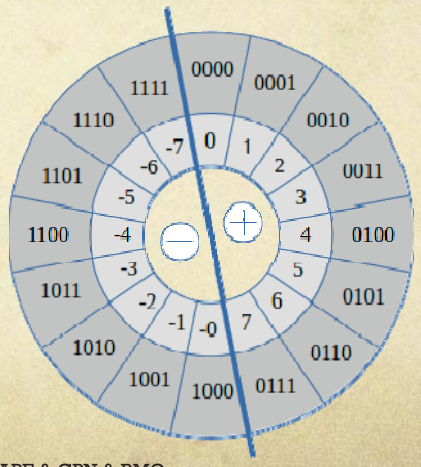
Représentation des nombres signés

- Plusieurs représentations des nombres entiers signés en binaire:
  - Signe-amplitude
  - Complément à 1
  - Complément à 2^n
  - Excédent de $2^{n-1} - 1$

Nombres négatifs: signe-amplitude

Signe-amplitude :

- Le bit de poids fort (MSB) indique le signe: 0 pour positif, 1 pour négatif, Les bits restants indiquent la valeur absolue
  - utilisée pour la mantisse des nombres en virgule flottante (notation scientifique)
  - Avec n bits on peut représenter des entiers entre: $-(2^{n-1}-1)$ et $+(2^{n-1}-1)$

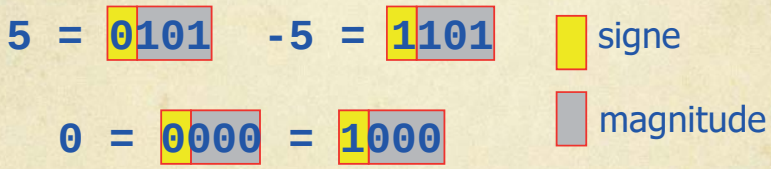


- Inconvénients :
  - 2 représentations du zéro
  - Algorithmique complexe, même pour l'addition

Nombres négatifs: signe-amplitude

- Avec n bits on peut représenter des entiers entre:
  $-(2^{n-1}-1)$ et $+(2^{n-1}-1)$

- Exemple avec n=4:



Nombres négatifs: signe-amplitude

- Exemples d'opérations arithmétiques (avec n=4):

```
  5   0101
 +3   0011
 ---------
  8   1000
```

résultat faux (0):
dépassement de capacité

la soustraction devrait pouvoir être traitée comme une addition:
5 - 3 = 5 + (-3)

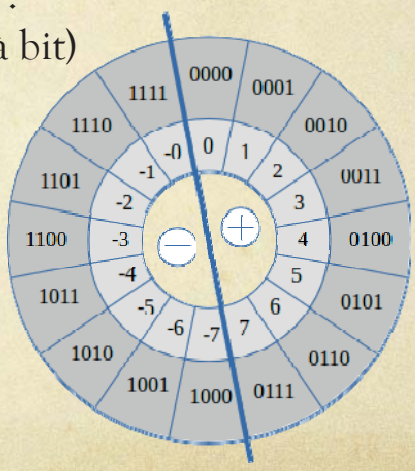
```
  5   0101
 -3   1011
 ---------
  2   0000
```

résultat faux (0)

Nombres négatifs: complément à 1

Complément à 1 : en fait, c'est le complément à $2^n - 1$

- Avantage : facile à calculer (inverser tous les bits)
- Formule pour calculer le complément à 1 :
  C1(A) = $2^n - 1 - A$ = not(A) (inversion bit à bit)



- Inconvénients :
  - 2 représentations du zéro
    => pas utilisé

Nombres négatifs : complément à 2^n

- Complément à 2^n => représentation naturelle
  3 - 4 = -1, soit 0011 - 0100 = 1111 !
- Un compteur-décompteur n bits en binaire pur
  - compte en boucle : 0, 1, ... $2^n - 1$, 0, 1, ...
  - sur 4 bits:
    - si compte + 1 depuis 0 ("0000"):
      "0000" → "0001" → "0010" → "0011" → ...
      0 +1 +2 +3 ...
    - si décompte - 1 depuis 0 ("0000"):
      "0000" → "1111" → "1110" → "1101" → ...
      0 -1 -2 -3 ...

Nombres négatifs : complément à 2^n

- Sur un compteur n bits, la valeur –1 est naturellement représentée par 2n – 1, qui est la valeur obtenue en décomptant 1 fois depuis 0
- Avec cette représentation, en additionnant –1 et +1 on obtient 2^n :
  - -1 est le complément à 2^n de +1
  - -2 est le complément à 2^n de +2, etc
- D'où : «représentation en complément à 2^n»
- Autre terminologie souvent utilisée :

  ce nombre est *(écrit)* «en *(notation)* complément à 2»

Nombres négatifs : complément à 2^n

- La représentation en complément à 2^n d'un nombre négatif –A, s'obtient en calculant le complément à 2^n de +A, soit :

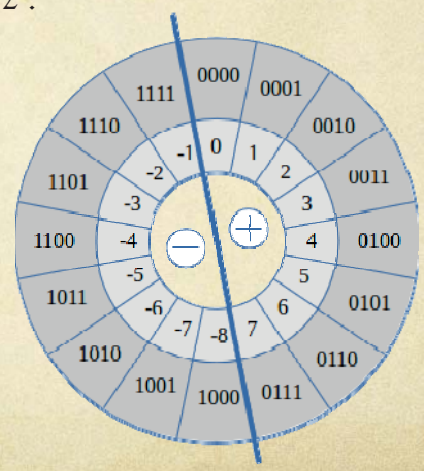
  $C_2(A) = 2^n - A$
- Le complément à 2^n d'un nombre s'obtient en inversant chacun de ses n bits, puis en ajoutant 1 au résultat
- Inversion d'un bit : (not) — mettre 0 à la place d'un 1 et mettre 1 à la place d'un 0

Nombres négatifs: complément à 2

Complément à 2 : c'est le complément à 2^n (C_2)

- Avantage : même circuit d'addition pour non-signé et signé
- Formule pour calculer le complément à 2 :
  C2(A) = 2^n – A
- Représentation:

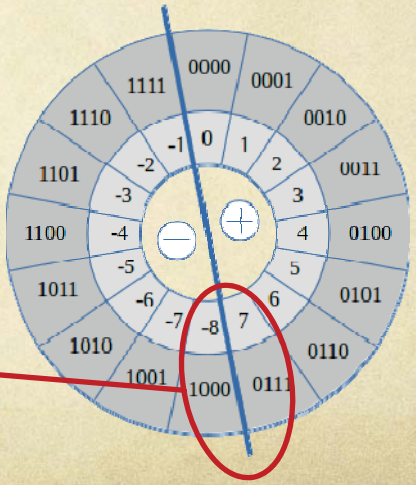
$$x_{n-1}x_{n-2}\cdots x_0 = -x_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$



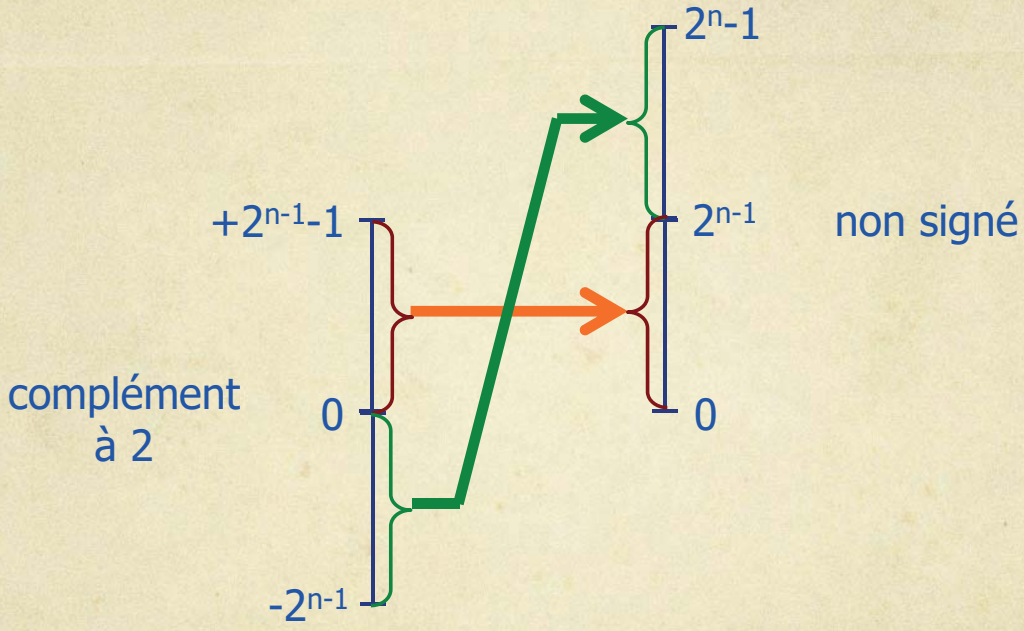
- Présenté à la suite ...
  - Généralement utilisé pour les nombres signés en informatique

Nombres négatifs : complément à 2

- Représentation sur un cercle des nombres signés en complément à 2
  - Débordement:
    il a lieu entre +7 et –8 !
  - Il est nommé: overflow
- Avec n bits on peut représenter des entiers entre:
  $-(2^{n-1})$ et $+(2^{n-1}-1)$



Nombres négatifs : complément à 2



Nombres négatifs : complément à 2

- Exemples d'opérations arithmétiques (avec n=4):

```
  5   0101
 +3   0011
 ---------
  8   1000
```

résultat faux (-8):
dépassement de capacité

la soustraction peut être traitée comme une addition:
5 - 3 = 5 + (-3)

```
  5   0101
 -3   1101
 ---------
  2   0010
```

résultat correct

Soustraction avec complément à 2

- Exemple: calculer 25 - 18 en binaire

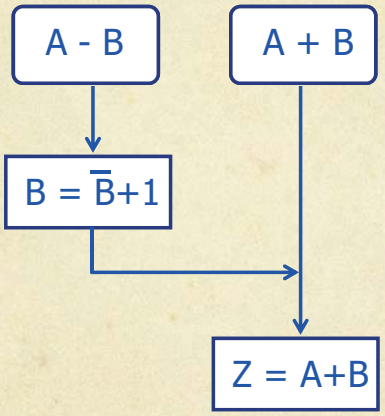
Il revient à calculer le [25 + C_2(18)] ➔ 25 – 18 = + 7

exposant de 2:	5	4	3	2	1	0	
+ 18 :	0	1	0	0	1	0	
C_1(18)	1	0	1	1	0	1	☺ simple inversion des bits
+ 1 :						1	
C_2(18) :	1	0	1	1	1	0	☺ 1ère étape C_2(18)

Retenue :	1	1	1				
+ 25 :	0	1	1	0	0	1	☺
+ C_2(18) :	1	0	1	1	1	0	☺ 2ème étape 25 + C_2(18)
+ 7 :	0	0	0	1	1	1	☺ Résultat signé !!!

Opérations avec complément à 2

- Les opérations d'addition et de soustraction sont simplifiées en complément à deux:



Relation entre C_1 et C_2

- Complément à 1 :
  - $C_1(A) = 2^n - 1 - A = not(A)$

- Complément à 2 :
  - $C_2(A) = 2^n - A = 2^n - 1 + 1 - A = C_1(A) + 1$

  d'où:

  $C_2(A) = not(A) + 1$ (utilisé très fréquemment)

Signe-Magnitude Vs C_2

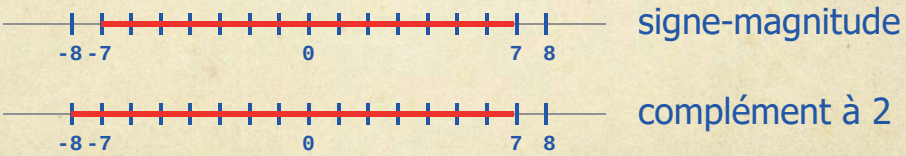
- Exemple avec n=4:

```
 5 = 0101        -5 = 1011
 3 = 0011        -3 = 1101
 8 = impossible  -8 = 1000

 0 = 0000
```

signe

- Si n=4:



Signe-Magnitude Vs C_2

	non signé	signe-magnitude	complément à 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	0	-8
1001	9	-1	-7
1010	10	-2	-6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

Exercices série II

1. En binaire, sur 8 bits, écrivez les nombres suivants
   - sans signe : $+128_{10}$
   - en notation «complément à 2» : -128_{10}
   - en notation «complément à 2» : $+128_{10}$
   - le complément à 2 de : $+128_{10}$
   - en notation «signe-amplitude» : -127_{10}
   - en notation «signe-amplitude» : $+128_{10}$
   - en notation «excédent de 127» : $+128_{10}$
   - en notation «excédent de 127» : -128_{10}

Exercices série II

2. Comment se justifie la recette de cuisine pour calculer le complément à 2 d'un nombre «inverser tous les bits puis ajouter 1» ?
   En examinant les chiffres 1 à 1 depuis la droite, trouvez une autre recette.

3. Extension de nombres signés : comment étendre sur 2n bits un nombre de n bits, signé, en notation «complément à 2»?

Etendre un nombre en complément à 2

- Pour étendre, par exemple de 8 à 16 bits, un nombre **sans signe**, il suffit de le compléter avec des 0 sur sa gauche
  - Exemple : le nombre 8bits sans signe **1001 1100**
    étendu sur 16 bits devient **0000 0000 1001 1100**
- Qu'en est-il pour un nombre signé, en notation «complément à 2»?
  - Si le nombre est positif, on le complète avec des 0, comme un nombre sans signe

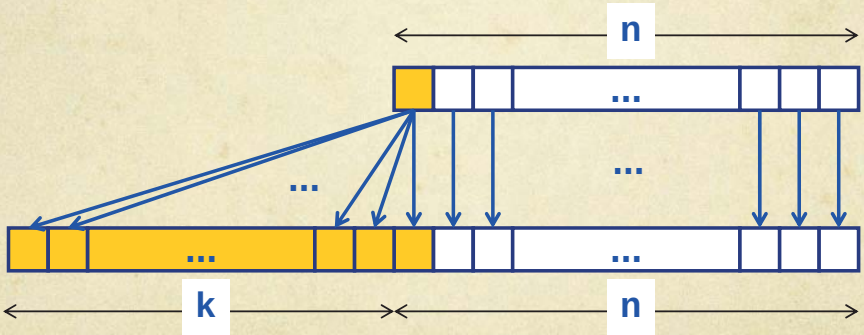
Etendre un nombre en complément à 2

- Si le **nombre est négatif**, en passant de 8 à 16 bits on passe de «complément à 2^8» à «complément à 2^{16}» !
- Il faudra lui ajouter $2^{16} - 2^8 = 1111111100000000$
- Pour étendre de k bits un nombre signé, en notation «complément à 2», il faut le compléter sur sa gauche avec k copies du bit de signe

- Exemple : le nombre 8 bits signé 1001 1100
  étendu sur 16 bits devient 1111 1111 1001 1100

Extension du signe en C_2

- Si l'on veut passer un entier signé **x** d'un format **n** bits vers un format **n+k** bits, en gardant la même valeur, il suffit de faire une extension de signe: le bit de signe est répété sur les nouveaux **k** bits de poids fort



- Traitement du dépassement de capacité pour une addition:
  - si les deux opérandes sont du même signe:
    dépassement si le résultat est du signe opposé
  - si les deux opérandes sont de signe opposé:
    il n'y a jamais de dépassement de capacité
- Plus formellement, pour des nombres n bits, signés en complément à 2:

$$\text{overflow} = c^n \oplus c^{n-1}$$

- Exemple:

```
      0111   carry (retenue)
 5     0101
+3    +0011
 8     1000
```

ov = $c^n \oplus c^{n-1} = 0 \oplus 1 = 1$

Entiers signés

- Les entiers signés (integer) sont codés sur 32 bits et en utilisant le complément à deux.
- Un entier signé peut donc prendre les valeurs -2,147,483,648 à +2,147,483,647
- Google a dû modifier le type de variable utilisée pour compter le nombre de « vues » des vidéos sur YouTube en 2014 lorsque certains on dépassé plus de 2 milliards

Exercice série II

4. Ecrivez en binaire sur 8 bits en C2 (nombre signé) les nombres suivants:
   - -37
   - +53

+37 = 0010 0101, -37 = C2(+37)= 1101 1011

+53 = 0011 0101

5. Ecrivez en binaire sur 12 bits en C2 les mêmes nombres que ci-dessus, soit -37 et +53

-37 = 1111 1101 1011

+53 = 0000 0011 0101

6. Pour les deux représentations effectuez le calcul :
   - 53 – 37 Que constatez-vous ? Même résultat

53 – 37 = 53 + C2(37) =00 00 0011 0101 + 1111 1101 1011 = 0000 0001 0000

Exercices série III

- Quelles multiplications sont les plus faciles à faire en décimal?
- Et en binaire?
- Peut-on multiplier par dix en binaire à l'aide d'une simple addition?

Nombres réels

Représentation des nombres réels

- Un nombre réel est représenté en décimal sous la forme:
  $d_m d_{m-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots d_{-n}$
  où la valeur du nombre est:

$$d = \sum_{i=-n}^{m} 10^i \times d_i$$

- Par exemple, 12.34_{10} représente le nombre:

$$1x10^1+2x10^0+3x10^{-1}+4x10^{-2} = 12\ 34/100$$

- En conséquence, en décimal on ne peut représenter exactement que des nombres fractionnaires de la forme $X/10^k$

Représentation des nombres réels

- En binaire, nous avons:
  $b_m b_{m-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-n}$
  où la valeur du nombre est:

$$b = \sum_{i=-n}^{m} 2^i \times b_i$$

- Par exemple, 101.11_2 représente le nombre:

$$1x2^2+0x2^1+1x2^0+1x2^{-1}+1x2^{-2} = 5\ 3/4$$

- En conséquence, en binaire on ne peut représenter exactement que des nombres fractionnaires de la forme $X/2^k$
  [35]Exemples:
  - $1/3 = 0.0101010101[01]_2$
  - $1/10 = 0.0001100110011[0011]_2$

Représentation des nombres réels

- Passage à binaire d'un nombre réel en base 10:

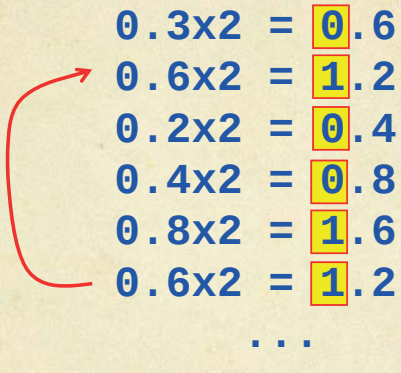
0.375 = ?

0.375x2 = 0.75
0.75x2 = 1.5
0.5x2 = 1.0

0.375 = 0.011

Représentation des nombres réels

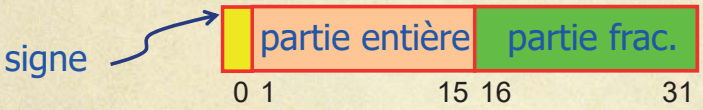
0.3 = ?



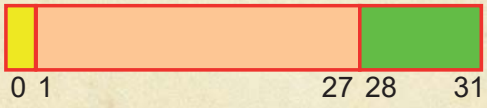
0.3 = 0.01001[1001]= 0.0[1001]

Codage binaire des nombres réels

- Comment coder un nombre réel en utilisant un nombre fixe de bits ? Par exemple, avec 32 bits ?
- Idée: on suppose que la virgule est vers le milieu



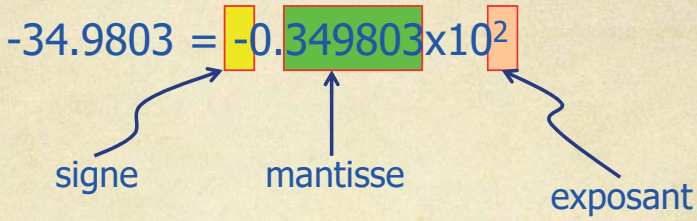
- Max value = ~65536 ; plus petite valeur = 2^{-16} = 0.00001525878
- Autre idée:



Max value = ~268'435'456; plus petite valeur = 2^{-4} = 0.0625

- Autre idée ?

Codage binaire des nombres réels



Séquence binaire:

William Kahan

- Père du « floating point »
- Il est récompensé avec le ACM Turing Award (le prix Nobel de l'informatique) en 1989 pour son travail

Codage binaire des nombres réels

- Un même nombre réel peut être écrit de différentes façons. Par exemple:

$$0.110x2^5 = 110x2^2 = 0.0110x2^6$$

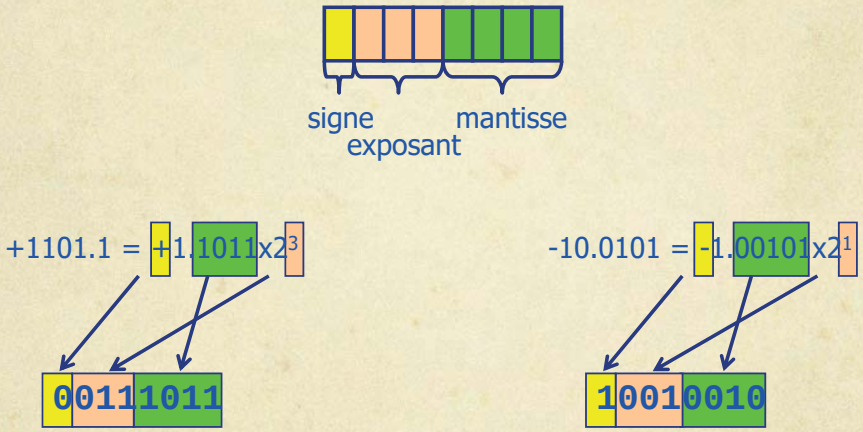
- Pour éviter des représentations différentes du même nombre, la mantisse est normalisée. Dans la convention la plus courante, un nombre binaire normalisé différent de zéro a la forme:

$$\pm 1.bbb...bx2^{\pm e}$$

- Comme, sous cette forme, le bit le plus signifiant est toujours égal à 1, il n'est pas nécessaire de le coder: il est **implicite**

Codage binaire des nombres réels

- Exemple:
  supposons la représentation suivante:



Codage binaire des nombres réels

- Problème: sous cette forme, il est impossible de coder le nombre zéro
- Solution: le nombre zéro est représenté avec tous les bits à 0. Avec la représentation de l'exemple précédent, nous avons: **0.0 = 0 000 0000**
- Par extension, tous les nombres avec exposant égal à 0 sont dits non normalisés: le bit à gauche du point décimal est égal à 0 et non pas à 1, comme c'est le cas pour les autres nombres, normalisés

Codage binaire des nombres réels

- Problème: comment représenter le nombre 1.0?
- Problème: comment représenter les exposants négatifs?

- Solution: Comme un exposant est un nombre entier signé, une solution serait de le représenter en complément à 2.

NON

- Ce n'est pas la solution choisie

Codage binaire des nombres réels

- En général, l'exposant est représenté de façon biaisée: une constante, le biais, doit être soustrait de la valeur dans le champ pour obtenir la vraie valeur de l'exposant:

  champ exposant = exposant + biais

- Typiquement, la valeur du biais est $2^{k-1}-1$, où k est le nombre de bits du champ de l'exposant

- Toutefois, les deux valeurs extrêmes du champ exposant sont réservées pour des cas particuliers:
  - 00...00: pour les nombres non normalisés ($0 \leq X < 1$)
  - 11..11: pour infini (positif et négatif) et NaN (*not a number*)

Codage binaire des nombres réels

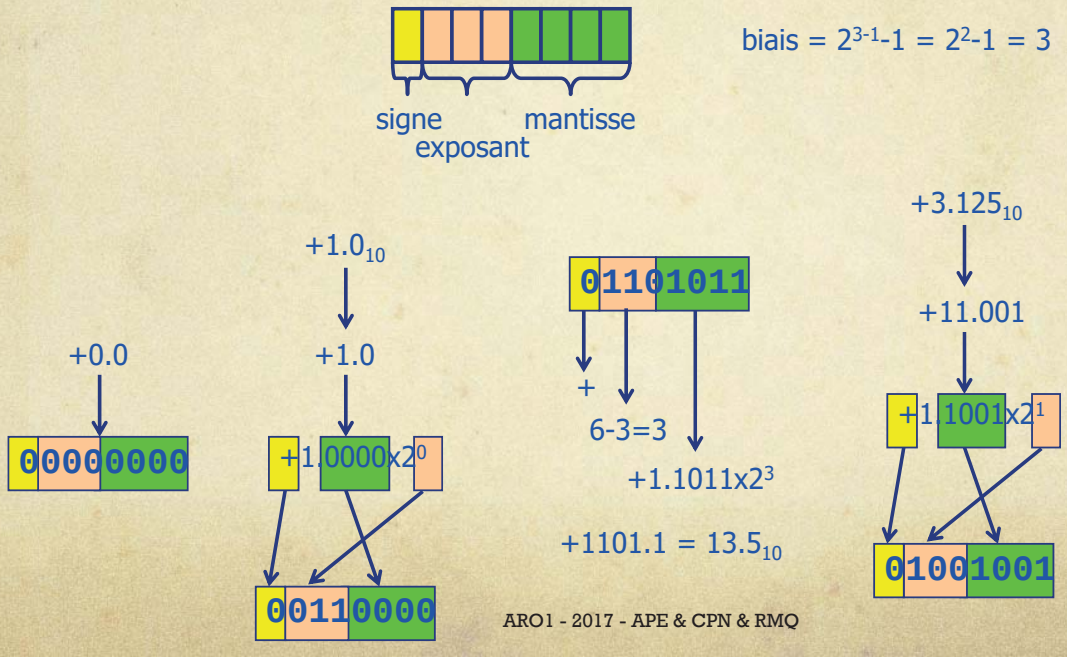
- Exemple:
  si k=4, la valeur du biais sera 7, l'exposant pourra avoir une valeur entre -2^3+1 et 2^3, et la valeur représentée dans le champ exposant sera exposant+biais

champ exposant	exposant	champ exposant	exposant
0000	non normalisé	1000	1
0001	-6	1001	2
0010	-5	1010	3
0011	-4	1011	4
0100	-3	1100	5
0101	-2	1101	6
0110	-1	1110	7
0111	0	1111	infini

- où les valeurs -7 et 8 de l'exposant sont réservées pour les cas spéciaux (nombres non normalisés et infini, respectivement)
- Cette représentation facilite la comparaison entre deux nombres et permet la représentation des nombres 0.0 et 1.0

Codage binaire des nombres réels

- Exemple:
  supposons la représentation suivante:



Nombres non normalisés

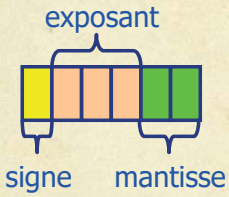
- Avec la représentation normalisée des nombres réels, le bit à gauche du point décimal est toujours égal à 1, implicitement. Il est donc impossible de représenter la valeur 0.0 de façon normalisée
- Pour résoudre ce problème, certains nombres sont non normalisés: dans ces cas, le bit à gauche du point décimal est égal à 0
- Les nombres non normalisés sont ceux dont le champ de l'exposant est égal à 0

Nombres non normalisés

- Il existe donc deux représentations pour 0.0:
  - +0.0 = 0 00...00
  - -0.0 = 1 00...00
- Les nombres non normalisés différents de 0.0 sont utilisés pour représenter de très petites valeurs, proches de 0.0
- Normalement, la valeur de l'exposant d'un nombre non normalisé devrait être -biais. Toutefois, pour assurer une meilleure transition entre les nombres normalisés et les non normalisés, l'exposant est calculé dans ces cas comme 1-biais

Explications par l'exemple

- Supposons la représentation suivante:



- Le biais est égal à 3
- Il y a un total de 64 nombres représentés:

Quelques exemples

0 000 00 = +0.0
0 000 01 = $+0.01x2^{-2}$ = +0.0001
0 000 10 = $+0.10x2^{-2}$ = +0.001
0 000 11 = $+0.11x2^{-2}$ = +0.0011
0 001 00 = $+1.00x2^{-2}$ = +0.01
0 001 01 = $+1.01x2^{-2}$ = +0.0101
0 001 10 = $+1.10x2^{-2}$ = +0.011
0 001 11 = $+1.11x2^{-2}$ = +0.0111
0 010 00 = $+1.00x2^{-1}$ = +0.1
0 010 01 = $+1.01x2^{-1}$ = +0.101
0 010 10 = $+1.10x2^{-1}$ = +0.11
0 010 11 = $+1.11x2^{-1}$ = +0.111
0 011 00 = $+1.00x2^{0}$ = +1.0
0 011 01 = $+1.01x2^{0}$ = +1.01
0 011 10 = $+1.10x2^{0}$ = +1.1
0 011 11 = $+1.11x2^{0}$ = +1.11

0 100 00 = $+1.00x2^{1}$ = +10.0
0 100 01 = $+1.01x2^{1}$ = +10.1
0 100 10 = $+1.10x2^{1}$ = +11.0
0 100 11 = $+1.11x2^{1}$ = +11.1
0 101 00 = $+1.00x2^{2}$ = +100.0
0 101 01 = $+1.01x2^{2}$ = +101.0
0 101 10 = $+1.10x2^{2}$ = +110.0
0 101 11 = $+1.11x2^{2}$ = +111.0
0 110 00 = $+1.00x2^{3}$ = +1000.0
0 110 01 = $+1.01x2^{3}$ = +1010.0
0 110 10 = $+1.10x2^{3}$ = +1100.0
0 110 11 = $+1.11x2^{3}$ = +1110.0
0 111 00 = $+\infty$
0 111 01 = NaN
0 111 10 = NaN
0 111 11 = NaN

non normalisé valeurs spéciales

Quelques exemples

1 000 00 = -0.0
1 000 01 = -0.01x2-2 = -0.0001
1 000 10 = -0.10x2-2 = -0.001
1 000 11 = -0.11x2-2 = -0.0011
1 001 00 = $-1.00 \times 2^{-2} = -0.01$
1 001 01 = $-1.01 \times 2^{-2} = -0.0101$
1 001 10 = $-1.10 \times 2^{-2} = -0.011$
1 001 11 = $-1.11 \times 2^{-2} = -0.0111$
1 010 00 = $-1.00 \times 2^{-1} = -0.1$
1 010 01 = $-1.01 \times 2^{-1} = -0.101$
1 010 10 = $-1.10 \times 2^{-1} = -0.11$
1 010 11 = $-1.11 \times 2^{-1} = -0.111$
1 011 00 = $-1.00 \times 2^{0} = -1.0$
1 011 01 = $-1.01 \times 2^{0} = -1.01$
1 011 10 = $-1.10 \times 2^{0} = -1.1$
1 011 11 = $-1.11 \times 2^{0} = -1.11$

1 100 00 = $-1.00 \times 2^{1} = -10.0$
1 100 01 = $-1.01 \times 2^{1} = -10.1$
1 100 10 = $-1.10 \times 2^{1} = -11.0$
1 100 11 = $-1.11 \times 2^{1} = -11.1$
1 101 00 = $-1.00 \times 2^{2} = -100.0$
1 101 01 = $-1.01 \times 2^{2} = -101.0$
1 101 10 = $-1.10 \times 2^{2} = -110.0$
1 101 11 = $-1.11 \times 2^{2} = -111.0$
1 110 00 = $-1.00 \times 2^{3} = -1000.0$
1 110 01 = $-1.01 \times 2^{3} = -1010.0$
1 110 10 = $-1.10 \times 2^{3} = -1100.0$
1 110 11 = $-1.11 \times 2^{3} = -1110.0$
1 111 00 = $-\infty$
1 111 01 = NaN
1 111 10 = NaN
1 111 11 = NaN

non normalisé valeurs spéciales

Standard IEEE 754-2008

- **Précision simple** ou **Binary32**:
  - nombre de bits: 32
  - mantisse sur 23 bits
  - exposant sur 8 bits (biais = $2^{8-1} - 1 = 127$)
- La valeur d'un nombre est donnée par l'expression:

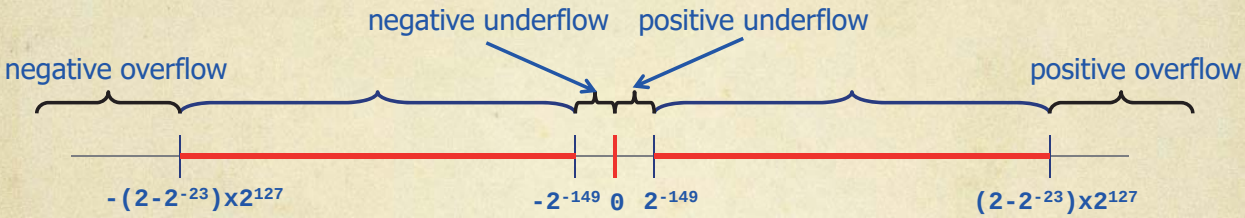
$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{e_b - bias}$$

- Exemple:

A = 1 10000010 00110000000000000000000
s = 1 (négatif)
e = ($2^7 + 2^1$) - 127 = 130 - 127 = 3
m = $2^{-3} + 2^{-4}$ = 0.125 + 0.0625 = 0.1875
A = -1.1875×2^3 = -9.5

Standard IEEE 754-2008

- Le rang des nombres couverts en précision simple est:



- Il y a toujours un compromis entre le rang et la précision: si on augmente le nombre de bits de l'exposant, on étend le rang de nombres couverts. Mais, comme il y a toujours un nombre fixe de nombres que l'on peut représenter, la précision diminue. La seule façon d'augmenter le rang et la précision est d'augmenter le nombre total de bits du format

Standard IEEE 754-2008

- Valeurs particulières:
  - deux valeurs pour 0: les deux signes, avec exposant=mantisse=0
  - infini positif: signe positif, exposant=1...1, mantisse=0
  - infini négatif: signe négatif, exposant=1...1, mantisse=0
  - NaN (*not a number*): signe indifférent, exposant=1...1, mantisse≠0

Standard IEEE 754-2008

- **Binary16 (half precision):**
  - nombre de bits: 16
  - mantisse sur 10 bits
  - exposant sur 5 bits (biais = $2^{5-1} - 1 = 15$)
- **Binary64 (Précision double):**
  - nombre de bits: 64
  - mantisse sur 52 bits
  - exposant sur 11 bits (biais = $2^{11-1} - 1 = 1023$)
- **Binary128 ou précision Quad:**
  - nombre de bits: 128
  - mantisse sur 112 bits
  - exposant sur 15 bits (biais = $2^{15-1} - 1 = 16383$)

Calcul de la valeur d'un Binary32

s	$e_8e_7e_6e_5e_4e_3e_2e_1$	$m_{23}\ m_{22}\ m_{21}\ ....\ m_3\ m_2\ m_1$	valeur
0/1	0 0 0 0 0 0 0 0	0 0 0 0 ..... 0 0 0 0	+/- 0.0
0/1	1 1 1 1 1 1 1 1	0 0 0 0 ..... 0 0 0 0	+/- ∞
0/1	1 1 1 1 1 1 1 1	!= 0 0 0 0 ..... 0 0 0 0	NaN
0/1	!= 1 .... 1 , != 0 0		$(-1)^s \times (1 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{e_b - 127}$
0/1	0 0 0 0 0 0 0 0	!= 0 0 0 0 ..... 0 0 0 0	$(-1)^s \times (0 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{-126}$

Virgule flottante : Standard IEEE 754

- Précision simple:
  - nombre de bits: 32
  - mantisse sur 23 bits
  - exposant sur 8 bits (biais = $2^{8-1} - 1 = 127$)
- La valeur d'un nombre est donnée par l'expression:

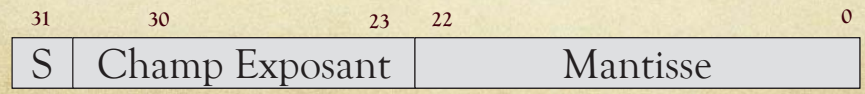
$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{e - E_{\max}}$$

Virgule flottante : Standard IEEE 754

- Précision simple normalisée:
  - 1) déterminer le signe : 0 positif, 1 négatif
  - 2) Exposant: ici le biais est de 127 donc

    Exp= champ_Exposant – 127
  - 3) Mantisse: pour déterminer le nombre il faut

    1.Mantisse le 1 étant implicite dans la représentation
  - 4) donc la valeur réelle sera
    - 1.Mantisse * $\mathbf{2}^{exposant}$
  - 5) décalage à gauche ou à droite de la virgule suivant l'exposant
  - 6) Conversion binaire décimal avec le signe

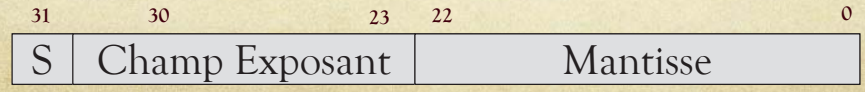


Virgule flottante : Standard IEEE 754

- Précision simple non-normalisée:
  - 1) déterminer le signe : 0 positif, 1 négatif
  - 2) Exposant: ici le biais est de 126 donc

    champs exposant = 0000 0000
  - 3) Mantisse: pour déterminer le nombre il faut

    0.Mantisse le 0 étant implicite dans la représentation
  - 4) donc la valeur réelle sera
    - 0.Mantisse * $\mathbf{2^{-126}}$
  - 5) décalage à gauche ou à droite de la virgule pour obtenir 1.xxxx * $\mathbf{2^{-z}}$ avec z=-126 - nombre de décalage à gauche
  - 6) Conversion binaire décimal avec le signe ou en puissance de 2



Virgule flottante : Standard IEEE 754

Type	Exposant	Mantisse	Valeur approchée	Écart / préc
Zéro	0000 0000	000 0000 0000 0000 0000 0000	0,0	
Plus petit **nombre dénormalisé**	0000 0000	000 0000 0000 0000 0000 0001	$1,4\times10^{-45}$	$1,4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0010	$2,8\times10^{-45}$	$1,4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0011	$4,2\times10^{-45}$	$1,4\times10^{-45}$
Autre nombre dénormalisé	0000 0000	100 0000 0000 0000 0000 0000	$5,9\times10^{-39}$	
Plus grand nombre dénormalisé	0000 0000	111 1111 1111 1111 1111 1111	$1,17549421\times10^{-38}$	
Plus petit **nombre normalisé**	0000 0001	000 0000 0000 0000 0000 0000	$1,17549435\times10^{-38}$	$1,4\times10^{-45}$
Nombre normalisé suivant	0000 0001	000 0000 0000 0000 0000 0001	$1,17549449\times10^{-38}$	$1,4\times10^{-45}$
Presque le double	0000 0001	111 1111 1111 1111 1111 1111	$2,35098856\times10^{-38}$	$1,4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0000	$2,35098870\times10^{-38}$	$1,4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0001	$2,35098898\times10^{-38}$	$2,8\times10^{-45}$
Presque 1	0111 1110	111 1111 1111 1111 1111 1111	0,99999994	$0,6\times10^{-7}$

1	0111 1111	000 0000 0000 0000 0000 0000	1,00000000	
Nombre suivant 1	0111 1111	000 0000 0000 0000 0000 0001	1,00000012	$1,2\times10^{-7}$
Presque le plus grand nombre	1111 1110	111 1111 1111 1111 1111 1110	$3,40282326\times10^{38}$	
Plus grand nombre normalisé	1111 1110	111 1111 1111 1111 1111 1111	$3,40282346\times10^{38}$	2×10^{31}
Infini	1111 1111	000 0000 0000 0000 0000 0000	Infini	
Première valeur (dénormalisée) de ***NaN*** avertisseur	1111 1111	000 0000 0000 0000 0000 0001	*NaN*	
NaN normalisé (avertisseur)	1111 1111	010 0000 0000 0000 0000 0000	*NaN*	
Dernière valeur (dénormalisée) de ***NaN*** avertisseur	1111 1111	011 1111 1111 1111 1111 1111	*NaN*	
Première valeur (dénormalisée) de ***NaN*** silencieux	1111 1111	100 0000 0000 0000 0000 0000	*NaN*	
Dernière valeur (dénormalisée) de ***NaN*** silencieux	1111 1111	111 1111 1111 1111 1111 1111	*NaN*	

Standard IEEE-754: hard et soft

- Les processeurs intégrant des FPU's (floating-point processing units) utilisent le standard et codent les nombres « réels » en virgule flotant, même si de fois ils ne respectent pas le standard à 100%.
- Au niveau logiciel, par exemple, dans le langage C, le compilateur gcc pour les architectures compatible Intel 32 bits utilise le format *simple précision* pour les variables de type **float**, *précision double* pour les variables de type **double**, et la *précision quad* pour les variables de type **long double**.

Problèmes de précision

- La représentation de nombres réels en base 2 pose quelques problèmes. Par exemple, la valeur 0.1 ne peut être codé qu'a l'aide d'une séquence infinie des bits.

$$0.1 = 0.0[0011]_2$$

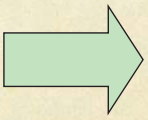
Voici quelques exemples illustrant des problèmes de précision:

```
#include <stdio.h>

void main()
{
  float x=0.0;
  int i;

 for(i=0;i<100;i++)
  x = x + 0.1;

 printf("%f\n",x);
}
```



10.000002

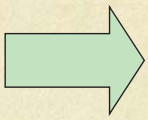
Problèmes de précision

```
#include <stdio.h>

void main()
{
  double x=1000000000.0;
  int i;

 for(i=0;i<100;i++)
  x = x + 0.1;

 printf("%lf\n",x);
}
```



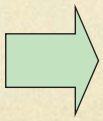
1000000010.000002

```
#include <stdio.h>

void main()
{
  long double x=1000000000.0;
  int i;

 for(i=0;i<100000;i++)
  x = x + 0.1;

 printf("%Lf\n",x);
}
```



1000009999.999998

Un exemple financier

- Calculer un impôt de 5% sur un appel téléphonique de 0.70 CHFs
- 1.05 x 0.70 = 0.735
- En Binary64:

  1.05 x 0.70 = 0.73499999999999998667732370449812151491641998291015625

  Arrondi au centième: 0.73

- Le calcul des opérations arithmétiques en base 10, sont donc généralement réalisés en logiciel: 100x ou 1000x plus lent qu'en matériel.

Du flottant décimal

- Les problèmes liés à la représentation des nombres couramment utilisés dans des domaines, tels que celui de la finance, où le résultat d'un calcul est finalement examiné ou pris en compte par un humain, ont motivé une révision du standard IEEE754-1985.
- La version 2008 du standard IEEE754 définit des formats pour coder des nombres en virgule flottant décimale. Ces formats s'appellent: **decimal32**, **decimal64** et **decimal128**.

	Decimal32	**Decimal64**	**Decimal128**
Valeur **min**	1.0×10^{-101}	1.0×10^{-398}	1.0×10^{-6176}
Valeur **max**	$9.99...\times 10^{96}$	$9.99...\times 10^{384}$	$9.99...\times 10^{6144}$

Représentation d'un nombre en decimal64

Code une mantisse de 16 digits décimales ($0..10^{16} - 1$) et un exposant entre -383 et +384

$$(-1)^{sign\_bit} \times 10^{exponent\_bits-398} \times true\_significand_2$$

Normalisés:

```
s 00eeeeeeee (0)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
s 01eeeeeeee (0)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
s 10eeeeeeee (0)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
```

Non normalisés:

```
s 1100eeeeee (100)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
s 1101eeeeee (100)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
s 1110eeeeee (100)TTTttttttttttttttttttttttttttttttttttttttttttttttttttt
```

Infini et NaN:

```
s 11110xxxxx x ... x       Infini
s 11111xxxxx x ... x       NaN
```

Encore quelques problèmes

- Les nombres dans l'ordinateur sont représentés par des séquences de 4, 8, 16, 32, 64, 128 bits, etc.
- Il est possible d'excéder la capacité de représentation d'un ordinateur (*overflow*), lors d'une simple addition de deux nombres entiers 32-bits, par exemple

10 + 9 = 19

```
 1010
+1001
------
 10011
```

bit perdu à cause du dépassement de capacité

2'147'483'648 + 2'147'483'650 = 2 ?

```
 10000000 … 0000
+10000000 … 0010
----------------
 00000000 … 0010
```

bit perdu à cause du dépassement de capacité

Encore quelques problèmes

- Il est possible d'excéder la capacité de représentation d'un ordinateur (overflow), lors d'une addition par exemple

10 + 9 = 19

```
 1010
+1001
------
 10011
```

bit perdu à cause du dépassement de capacité

Encore quelques problèmes

- Les **`int`** ne sont pas des entiers et les **`float`** ne sont pas des réels
- Est-ce que **$x^2 \geq 0$** est toujours vrai?
  - pour les **`float`**, oui
  - pour les **`int`**, pas toujours:
    - **`40000 * 40000`** → **`1600000000`**
    - **`50000 * 50000`** → ??
- Est-ce que **`(x+y)+z = x+(y+z)`** est toujours vrai?
  - pour les **`int`**, oui
  - pour les **`float`**, pas toujours:
    - **`(1e20 + -1e20) + 3.14`** → **`3.14`**
    - **`1e20 + (-1e20 + 3.14)`** → ??

Conversion de type à effets catastrophiques

- Tout le logiciel de la fusée **Ariane 5** (et de l'Ariane 4) est programmé en Ada, langage très sûr.

  Et pourtant ...

  l'échec du premier tir d'Ariane 5 a été causé par une erreur du système informatique de guidage. Une erreur qui a coûté $370 millions.
- Cette erreur est survenue lors d'une conversion de type qui a causé un dépassement de capacité d'une variable.

```
X: Long_float;          -- 64-bit float
Y: Integer;             -- 16-bit integer
Y := Integer(X);        -- conversion de type
```

Conversion de type à effets catastrophiques (2)

- Pendant la guerre du golfe au début des années 1990, les américains utilisaient des missiles Patriot pour intercepter les missiles Scud des iraquiens.
- Les missiles Patriot calculaient la trajectoire des missiles Scud en utilisant un horloge interne. Cet horloge était un simple compteur de dixièmes de seconde, donc un entier. Pour calculer le temps depuis le boot-up du missile, on multiplié ce compteur par la valeur 0.1 décimale codée sur 24 bits.
- Or, la valeur 0.1 décimale ne peut pas être représentée de manière exacte en binaire: pour un codage utilisant 24 bits, l'erreur est de 0.00000000000000000000011001100...ou env. 0.000000095 décimale.

Conversion de type à effets catastrophiques (2)

- Selon le rapport de l'armée, après 100 heures de boot up, un missile Patriot aurait cumulé une erreur de 0.34 sec dans son horloge interne.
- A une vitesse de 1676 m/s, les missiles auraient donc cumulé une erreur de plus d'un demi kilomètre.
- Le 25 février 1991, un missile Patriot a raté un missile Scud et par une erreur de logiciel, il a tué 28 soldats et bléssé d'autres 100 personnes en Irak!