Université Paris 7 - Denis Diderot

Licence Sciences et Applications

Mentions : Mathématiques, Informatique et MASS

IF1 - Introduction à l'informatique et la programmation

2008-2009

Amphi - 3-ème partie

Jean-Marie Rifflet

Les informations dans les machines

On distingue deux types de calculateurs :

- **analogiques : les données manipulées sont représentées par des phénomènes physiques (intensité, tension) proportionnelles aux données. Les circuits composant le calculateur sont tels que la valeur physique de sortie représente le résultat de l'opération réalisée sur les valeurs associées aux entrées. On travaille donc ici sur des domaines continus**

  Par exemple, la dynamo d'une bicyclette fournit plus ou moins de courant aux lampes selon qu'on pédale vite ou pas (le même phénomème s'observe avec les éoliennes).

- **digitaux ou numériques : ils manipulent essentiellement des nombres.**

  Les mécanismes électriques, magnétiques ou électroniques les composant possèdent deux états stables (typiquement deux tensions +5V et 0), représentés conventionnellement soit par 0, soit par 1 (ou Vrai/Faux).

  Il est donc nécessaire de discrétiser/numériser les phénomènes ou grandeurs considérés.
  Une donnée manipulée par une telle machine sera donc représentée par une certaine combinaisons de composants binaires.
  Par suite, une telle représentation d'une grandeur ne se fera qu'avec une précision limitée.

La mémoire : structures élémentaires

- le **bit** (*binary digit*) :
  élément d'information binaire

- l'**octet** (*byte*) :
  un groupe constitué de 8 bits

- le **mot** (*word*) :
  ensemble d'octets regroupés logiquement (mots de 2, 4, 8 ou 16 octets).

La mémoire centrale d'un ordinateur contient donc un ensemble de bits (on peut généralement adresser des octets ou des mots)

Une même séquence de bits peut évidemment être interprétée (selon le contexte et sa longueur) de différentes manières :

- **une instruction**
- **un caractère**
- **un nombre entier**
- **un nombre réel (ou plutôt décimal)**
- **une donnée codant du son, de l'image, de la video**

La représentation des caractères

Il s'agit d'associer à un jeu de caractères (par exemple ceux qu'on trouve sur un clavier) constituant en quelque sorte un alphabet un jeu de représentations comme par exemple des suites de tirets ou de points (code Morse) ou des suites de 0 et de 1 (pour une utilisation informatique)

On parle souvent de codage pour exprimer le fait qu'étant donnée une séquence elle ne peut représenter qu'une seule suite de caractères de l'alphabet

- **Le code EBCDIC :**
  Extended Binary Coded Decimal Interchange Code
  développé par IBM pour coder les caractères sur un octet à l'époque des cartes perforées, il n'est pas sorti du monde IBM

- **Le code ASCII (*American Standard Code for Information Interchange*) :**
  - standard adopté en 1960 comme codage des caractères sur un octet (seulement 7 bits utilisés)
  - il permet donc le codage de $128 = 2^7$ caractères

Représentation des caractères sur un octet (ASCII)

32	espaces	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	`	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	—	119	w
48	0	72	H	96	-	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	

Avec ce code, les caractères accentués par exemple ne sont pas codés.

D'où, la définition de nouvelles normes ISO étendant le code ASCII standard

Le code ISO 8859-15 est une évolution du ISO 8859-1 (LATIN 1) et ils sont utilisés pour la plupart des langues d'Europe occidentale

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STH	ETH	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	CD2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8	€	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	Ž	□
9	□	'	'	"	"	•	–	—	˜	™	š	›	œ	□	ž	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

- **Unicode : mis au point en 1991, pour remplacer les codes nationaux.**

  http://www.unicode.org/

  Il propose une identité, un numéro unique et une représentation sous forme de suites de bits de chaque caractère, quels que soient la langue, la machine, le logiciel.

  La version 5.1.0 couvre plus de 100.000 (cent-mille) caractères

  Trois formes de codage permettent la transmission dans un octet (8 bits), un mot de 16 bits ou un mot de 32 bits.

- **UTF-8 : utilisé par HTML pour coder chaque caractère comme une suite de 1 à 4 octets :**
  - **les 128 caractères de 0 à 127 sur un octet avec bit de poids fort nul**
    `A` **a le numéro** 65 **et est codé** `01000001`

- **les caractères de code > 127 sont codés sur plusieurs octets. Les bits de poids fort forment une suite de 1 de longueur égale au nombre d'octets utilisés pour coder le caractère. Les octets suivants ont 10 comme bits de poids forts.**
  caractère é de numéro 233 codé sur deux octets :

  (110)00011 (10)101001

```
Name  LATIN SMALL LETTER E WITH ACUTE
Block  Latin-1 Supplement
Category  Letter, Lowercase [Ll]
Index entries  E WITH ACUTE, LATIN SMALL LETTER
Encodings
HTML Entity (decimal)  &#233;
HTML Entity (hex)  &#xe9;
HTML Entity (named)  &eacute;
How to type in Microsoft Windows  Alt +00E9
UTF-8 (hex)  0xC3 0xA9 (c3a9)
UTF-8 (binary)  11000011:10101001
UTF-16 (hex)  0x00E9 (00e9)
UTF-16 (decimal)  233
UTF-32 (hex)  0x000000E9 (00e9)
UTF-32 (decimal)  233
C/C++/Java source code  "\u00E9"
Python source code  u"\u00E9"
```

symbole euro de numéro 8364, codé sur 3 octets :

(1110)0010 (10)000010 (10)101100

```
Unicode      Data
Name        EURO SIGN
Block        Currency Symbols
Category     Symbol, Currency [Sc]
Comments     currency sign for the European Monetary Union
             euro, not ecu
Encodings
HTML Entity (decimal)  &#8364;
HTML Entity (hex)  &#x20ac;
HTML Entity (named)  &euro;
How to type in Microsoft Windows  Alt +20AC
UTF-8 (hex)  0xE2 0x82 0xAC (e282ac)
UTF-8 (binary)  11100010:10000010:10101100
UTF-16 (hex)  0x20AC (20ac)
UTF-16 (decimal) 8 364
UTF-32 (hex)  0x000020AC (20ac)
UTF-32 (decimal) 8 364
C/C++/Java source code "\u20AC"
Python source code u"\u20AC"
```

- **UTF-16 : utilisé pour coder chaque caractère comme une suite de 1 ou 2 mots de 16 bits**
- **UTF-32 : utilisé pour coder chaque caractère sur un mot de 32 bits**

Systèmes de numération : quelques dates :

- **5000 av. J.-C. :**
  système de numération parlé sexagésimal (base 60) des Sumériens

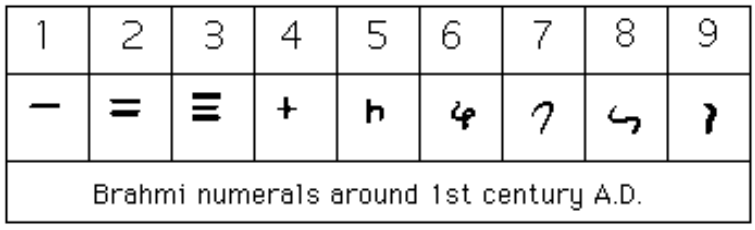
- **3200 av. J.-C. :**
  chiffres sumériens

- **3000 av. J.-C. :**
  numération hiérogliphique égyptienne. Système additionnel

- **1900-1600 av. J.-C. :**
  premier système de numération positionnel des babyloniens (base 60, pas de zéro)

- **fin du XIV^e siècle av. J.-C. :**
  apparition des chiffres chinois

- **III^e siècle av. J.-C. :**
  - ◇ **invention du zéro par les Babyloniens, pas vu comme un nombre, mais comme absence d'une certaine unité (afin d'éviter les ambiguités)**
  - ◇ **apparition des chiffres brahmi (indiens), précurseurs de nos chiffres (dits arabes) (il existait par ailleurs des symboles pour 10, 20, . . . , 100, 200, . . .**

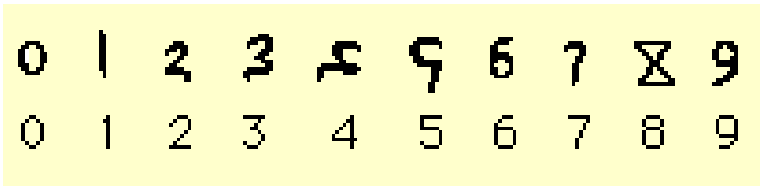


Brahmi numerals around 1st century A.D.

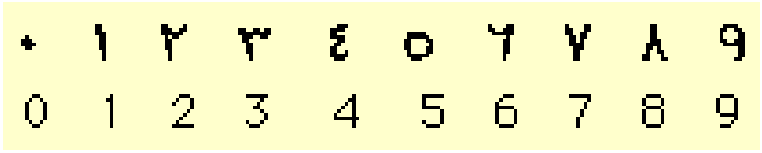
- **IV^e siècle de notre ère : naissance de la numération décimale indienne de position (ajout aux 9 chiffres d'un signe en forme de petit cercle comme chiffre zéro)**

- **458 : apparition "officielle" du nombre zéro en Inde (appelé $Sunya$ ... le vide en sanskrit traduit par $Sifr$ en arabe)**

- **fin du VIII^e siècle : numération décimale positionnelle et zéro dans la culture islamique dans la partie occidentale**



ou en Egypte et actuellement



- **IX^e siècle : introduction du zéro en Europe**

- **X^e siècle : introduction des chiffres arabes en Europe**

◇ **XII^e au XV^e siècle : stabilisation des chiffres arabes**

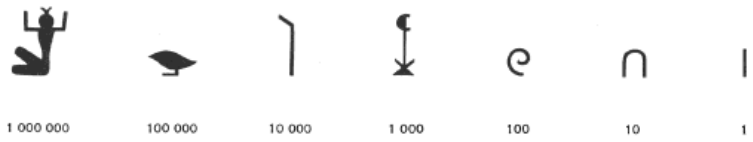
- **1489-1650 : introduction de symboles tels que +, −, =, <, >, x et de la notation à virgule**

- **1700 : promotion du système binaire par Leibniz : base deux dans laquelle le chiffre 1 représente Dieu et le chiffre 0 le vide**

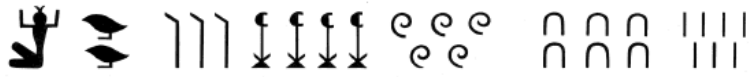
- **1795 : adoption du système métrique en France**

Systèmes de numération additionnels :

- **système égyptien: une valeur est attribuée à un pictogramme et on obtient une valeur en sommant les valeurs des pictogrammes. Ainsi à partir de**



on obtient la représentation suivante de 1234567 **:**



- **système romain :**

système romain	I	V	X	L	C	D	M
valeur décimal	1	5	10	50	100	500	1000

- Un certain nombre de règles sont utilisées pour l'écriture des nombres:

◇ **lorsqu'un symbole est placé à la droite d'un symbole plus fort que lui, sa valeur s'ajoute**

CCLXXI **correspond à** 271

◇ **on ne place jamais 4 symboles identiques à la suite.**

9 **s'écrit** IX **et non** VIIII

◇ **lorsqu'un symbole est placé à la gauche d'un symbole plus fort que lui on retranche sa valeur.**

CCXLIII **correspond à** 243

- Le plus grand nombre exprimable est MMMCMXCIX, c'est-à-dire 3999

- Système particulièrement inadapté au calcul : calculer par exemple

CCXLVII + CDLXXVIII

sans passer par l'écriture décimale.

Systèmes de numération positionnels :

Le principe est d'attribuer à un chiffre d'une suite, un poids en fonction de la position qu'il occupe dans la suite. Si on utilise b chiffres (on parle de la base b), un chiffre c dans une suite "*a comme valeur*" $c \times b^i$ si sa position est i dans la suite (comptée à partir de 0 à droite) représentant le nombre et la valeur représentée est la somme des valeurs associées aux différents chiffres en fonction de leur poids.

Dans la base b

$$c_n c_{n-1} \ldots c_1 c_0$$

représente ainsi le nombre

$$\sum_{i=0}^{n} c_i \times b^i$$

En base dix classique:

dans le nombre $\underline{35089}$**,**

- $\underline{3}$ **a comme poids** 4 **(d'où la valeur** $\underline{30000}$ **associée)**
- $\underline{5}$ **est de poids** 3 **(**$\underline{5000}$**),**
- $\underline{0}$ **est de poids** 2 **(**$\underline{0}$**),**
- $\underline{8}$ **est de poids** 1 **(**$\underline{80}$**),**
- $\underline{9}$ **est de poids** 0 **(**$\underline{9}$ **puisque** $10^0 = 1$**)**

Donc le nombre représenté est:

$30000 + 5000 + 80 + 9$**.**

Il en est ainsi pour toutes les bases

En base 8 la suite 35072 représente le nombre

$$2 \times 8^0 + 7 \times 8^1 + 0 \times 8^2 + 5 \times 8^3 + 3 \times 8^4$$

$$= 2 + 7 \times 8 + 5 \times 512 + 3 \times 4096$$

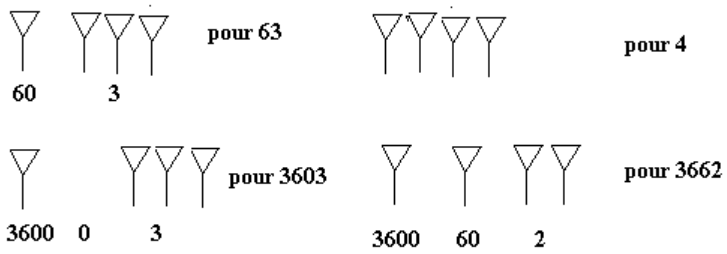
$$= 14906$$

Remarques

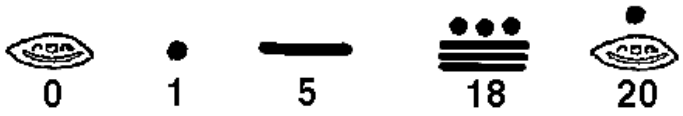
Dans toute base b

- **le nombre b s'écrit 10**
- **un nombre divisible par b^n se termine par n chiffres 0**

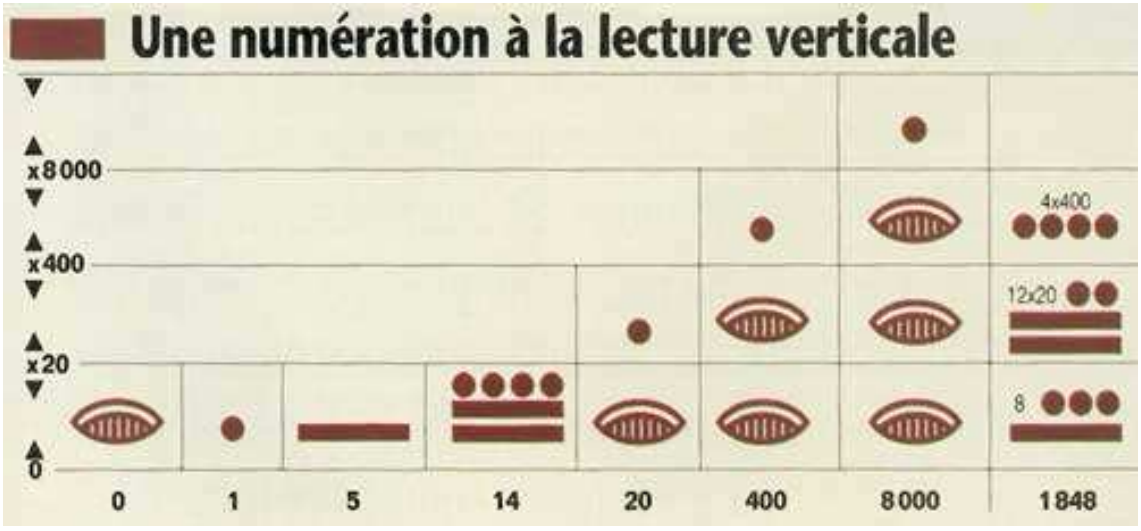
- **système sexagésimal des Babyloniens (base 60)**



- **système des mayas en base vingt (système vicédécimal) au moyen de trois symboles (une unité, cinq unités et le zéro) et une lecture verticale :**



d'où la représentation de quelques nombres:



Le codage des nombres entiers

- **dans un ordinateur, les nombres (en particulier les entiers) sont évidemment codés sous forme binaire.**

  Sur une machine à mots de n bits, il est possible de coder 2^n nombres entiers.

  Afin de pouvoir coder à la fois des nombres positifs et des nombres négatifs, un bit est réservé pour le signe (bit de gauche à 0 pour les nombres positifs et à 1 pour les nombres négatifs)

- **par exemple, la suite de 16 bits**

  0001101010111001

  représente le nombre qui s'écrit en base dix 6841_{dix} **:**

  - ◇ **les chiffres 1 correspondent aux poids 0,3,4,5,7,9, 11 et 12**
  - ◇ **les puissances de 2 correspondantes ont comme valeur décimale : 1,8,16,32,128,512,2048 et 4096**
  - ◇ **la valeur décimale du nombre est donc : 1+8+16+32+128+512+2048+4096**

- **les nombres négatifs sont représentés en complément à 2.**

  Ainsi, sur une machine à n bits, le nombre α est représenté par

  $$2^n - \alpha$$

  Ainsi, le nombre -6841 est représenté sur 16 bits par $2^{16} - 6841$, c'est-à-dire comme 65536-6841 = 58695,
  soit exactement 1110010101000111

- **sur une machine de n bits, le plus grand entier représentable est $2^{n-1} - 1$ et le plus petit -2^{n-1}.**
  Ainsi sur 16 bits l'intervalle est [-32768:32767]
  et sur 32 bits, c'est
  [-2147483648:2147483647]

- **les calculs se font modulo n (n est le nombre de bits utilisés)**

Passer d'une base à l'autre

- **de la base dix à la base deux :**
  diviser successivement par 2 jusqu'à atteindre un quotient nul.
  La représentation en base deux est obtenue en écrivant les restes successifs de droite à gauche.
  Ainsi pour le nombre 6841 (en base dix) :

6841	
1	3420
0	1710
0	855
1	427
1	213
1	106
0	53
1	26
0	13
1	6
0	3
1	1
1	0

ce qui conduit effectivement à
0001101010111001
en rajoutant ce qu'il faut de 0 au début.

- **de la base deux à la base dix:**
  - ◇ **ou bien en lisant le nombre de droite à gauche on calcule les puissances successives de 2 et on somme celles pour lesquelles on a un chiffre 1**
  - ◇ **ou bien en lisant le nombre de gauche à droite, on applique la méthode dite de Horner qui correspond, sur notre exemple, à calculer l'expression suivante :**

```
((((((((((((1)* 2
   + 1)* 2
      + 0)* 2
        + 1)* 2
          + 0)* 2
            + 1)* 2
              + 0)* 2
                + 1)* 2
                  + 1)* 2
                    + 1)* 2
                      + 0 )* 2
                        + 0)* 2
                          + 1
```

Représentation de l'opposé d'un nombre

- Pour déterminer rapidement la représentation de $-n$, on peut

- **rechercher dans la représentation de n le premier 1 en partant de la droite**
- **basculer tous les chiffres à sa gauche : les 1 deviennent des 0 et les 0 des 1**

Ainsi pour le nombre écrit 6841 en base dix

0001101010111001 [6841]
1110010101000111 [-6841]

Autre exemple:

0110010101001100 [25932]
1001101010110100 [-25932]

- Une autre manière de faire est d'inverser tous les bits puis d'ajouter 1

Les base huit et seize

L'écriture de nombres en base deux est fastidieuse et peu compacte.

Il est courant et pratique d'utiliser :

- **la base huit**
  - ◇ **représentation octale permettant de représenter des séquences de 3 bits en un seul caractère**
  - ◇ **chiffres de la base sont 0,1,2,3,4,5,6 et 7**
  - ◇ **correspondance avec la base deux :**

octal	0	1	2	3	4	5	6	7
binaire	000	001	010	011	100	101	110	111

  - ◇ **Exemple : le nombre de 16 bits**

    0001101010111001

    se traduit en 015271**,**

    ce qui est la représentation octale du nombre écrit 6841 **en base dix**

- **la base seize qui donne une écritue plus compacte.**
  - ◇ **regroupement de séquences de 4 bits en un seul caractère (en commenant par la droite)**
  - ◇ **chiffres: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E et F**
  - ◇ **correspondance avec la base deux :**

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

  - ◇ **Exemple : le nombre de 16 bits**
    0001101010111001
    se traduit en 1AB9,
    ce qui est la représentation hexadécimale du nombre décimal 6841

Sens de lecture des octets

Dans tout ce que nous avons fait, nous avons supposé que l'octet de poids fort était à gauche, et donc que les nombres se lisaient comme nous en avons l'habitude.

Ce n'est pas toujours le cas !

On distingue:

- **la représentation "par le grand bout" (*big endian*) dans laquelle l'octet de poids fort est stocké avant l'octet de poids faible (chez Motorola par exemple). C'est la manière usuelle de lire les nombres**

- **la représentation "par le petit bout" (*little endian*) dans laquelle l'octet de poids faible est stocké avant l'octet de poids fort (c'est le cas chez Intel)**

Arithmétique en base deux

- **les opérations sur les entiers s'appuient sur des tables d'addition et de multiplication:**

addition :

+	0	1
0	0	1
1	1	10

multiplication :

*	0	1
0	0	0
1	0	1

- **additions**
  - ◇ **deux nombres de même signe**

    dont la somme est représentable avec le nombre de bits dont on dispose)

```
  --111111111----1    retenues
+ 0001101010111001    6841
  0000011101010001    1873
  ------------------
  0010001000001010    8714
```

On a donc réalisé : 6841 + 1873

```
+ 1110010101000111     -6841
  1111100010101111     -1873
  ------------------
 1110111010111110110   -8714
```

Le bit de gauche est perdu et le complément à 2 est

0010001000001010

On a donc réalisé : -6841 + -1873

◇ **somme de deux entiers positifs de somme "trop grande"**

```
    1111---1--------    retenues
    0001101010111001     6841
+   0110110011000000    27840
--------------------
    1000011101111001
```

On a en fait réaliser une opération qui sur 16 bits a un résultat négatif. qui est l'opposé de 0111100010000111 **(**30855**), et donc sur 16 bits,**

6841 + 27840 = -30885

◇ **positif + négatif avec résultat positif**

Ici on calcule -6841 + 27840

```
  11-11--11-------    retenues
  1110010101000111     -6841
+ 0110110011000000     27840
  ------------------
 1010100100000111
```

sur 16 bits, un 17-ème chiffre apparaît, on l'oublie purement et simplement. Les 16 bits de poids faible 0101001000000111 **correspondent au résultat** 20999

◇ positif + négatif avec résultat négatif

--1--1----------	*retenues*
0001101010111001	*6841*
+ 1001001101000000	*-27840*

1010110111111001	

Le résultat est le complément de

0101001000000111

c'est-à-dire -20999

- **multiplications**
  - ◇ **un exemple simple : calcul de 29*11 en ne gardant que les chiffres significatifs :**

```
          11101
        * 1011
--------------
          11101
    +    11101.
    +  11101...
---------------
     100111111
```

  100111111 est la représentation de 319 = 29*11

  - ◇ **deux nombres positifs plus compliqués**

  On imagine que si les nombres sont suffisamment grands, le résultat ne pourra pas tenir sur le nombre de bits disponibles.

  C'est ce qui se passe par exemple avec 6841*315 sur une machine 16 bits:

```
                1101010111001
              * 100111011
    ------------------------------
                1101010111001
               1101010111001.
              1101010111001...
             1101010111001....
            1101010111001.....
           1101010111001........
    ------------------------------
      10000011100000110100011
```

Les 16 chiffres de poids faible

111000011010011

correspondent au complément à 2 de

0001111001011101

doit le nombre écrit 7773 en base dix

Le résultat obtenu est donc -7773

Sur une machine 32 bits, on obtient effectivement 10000011100000110100011, c'est-à-dire 2154915 (6841*315).

Les nombres non entiers

- les nombres rationnels (fractions):
  - ◇ apparus avec la notion de partage (les quantièmes 1/2, 1/3 , 1/5, . . .) que les Egyptiens savaient additionner
  - ◇ les Grecs ont considéré les rapports de grandeurs entières et faisaient les 4 opérations (éléments d'Euclide, 3-ème siècle avant J.C)
- nombres irrationnels
  - ◇ découverte par les Grecs de la non-rationnalité de $\sqrt{2}$ (qualifié d'irrationnel ou non énonçable)
  - ◇ vers l'an 1000, Arabes et Persans considéraient comme nombre tout rapport de longueur, d'où l'idée de nombres irrationnels.
    Irrationnalité de π (Lambert, 1761)

- les nombres décimaux : une partie entière et une partie décimale

Représentation des nombre non entiers

- **virgule fixe comme sur les caisses enregistreuses.**
  Par exemple :
  $r_1 = 7467,56$ **ou** $r_2 = 0,0000842$
- **fractionnaire comme couple d'entiers**
- **virgule flottante**
  - ◇ **éventuellement un signe + ou –**
  - ◇ **une** $mantisse\ m$ **(en virgule fixe)**
  - ◇ **un** $exposant\ e$
  - ◇ **une** $base\ b$

  soit donc: $\pm m \times b^e$

Par exemple quelques-unes des écritures pour $r_1 = 7467,56$ sont:

- ☐ $74,6756 \times 10^2$
- ☐ $746756000000 \times 10^{-8}$

ou pour $r_2 = 0,0000842$

- ☐ 842×10^{-7}
- ☐ $0,00000000842 \times 10^4$

Existence d' une *infinité* de représentations d'un nombre sous cette forme, d'où le choix d'une représentation normalisée:

> **dans la base b, la mantisse m est prise dans l'intervalle $[1 : b[$ (avec une représentation particulière pour zéro)**

pour r_1 : $7,46756 \times 10^3$
pour r_2 : $8,42 \times 10^{-5}$

Changement de base en virgule fixe

On se limitera ici au passage de la représentation en virgule fixe de la base $d$ix à la base $d$eux.

- **les chiffres de la partie entière d'un nombre écrit en notation positionnelle en base b correspondent successivement, de droite à gauche avant la virgule aux puissances positives de la base (chiffre des unités correspondant à la puissance 0 de la base, chiffre des "$basaines$" multiples de b^1, puis les multiples de b^2, . . .)**

- **les chiffres de la partie décimale (c'est-à-dire à la droite de la virgule) d'un nombre écrit en notation positionnelle en base b correspondent successivement, de gauche à droite après la virgule, aux puissances négatives de la base (les "$basièmes$" fractions de b^{-1}, puis celles de b^{-2}, . . .).**

Passage de la base dix à la base deux

- **la partie entière se calcule par divisions successives par 2 (il s'agit d'un nombre entier)**
- **la partie décimale se calcule par multiplications successives par 2 de la partie décimale et on garde à chaque étape le chiffre qui sort avant la virgule. La suite de ces chiffres 0 ou 1 donne la partie décimale en base deux.**

Appliquons cette transformation au nombre qui s'écrit `0,3` en base dix.

$$0,3 \times 2 = \underline{0}.6$$
$$0,6 \times 2 = \underline{1}.2$$
$$0,2 \times 2 = \underline{0}.4$$
$$0,4 \times 2 = \underline{0}.8$$
$$0,8 \times 2 = \underline{1}.6$$
$$0,6 \times 2 = \underline{1}.2$$
$$\ldots$$

On voit apparaître une répétition infinie de la séquence 1001 pour la représentation du nombre qui est donc

0,01001100110011001100 1...

Ainsi, un nombre finiment représenté en base dix ne l'est pas nécessairement en base deux.

Le fait que la représentation soit infinie correspond à ce que 0,3 ne s'exprime pas comme une somme finie de puissances négatives du nombre deux.

Comme on peut l'imaginer, cela est source de nombreuses difficultés et résultats surprenants aux conséquences parfois catastrophiques si on n'y prend pas garde.

Voir les exemples un peu plus loin.

Représentation dans un ordinateur

- la base b est bien évidemment 2

- représentation normalisée
  (norme IEEE754)
  - ◇ nombres codés sur 32 bits en simple précision et 64 bits en double précision
  - ◇ nombre de bits des composants de la représentation:

	signe	exposant	mantisse	excès exposant
simple précision	1 bit	8 bits	23 bits	127
double précision	1 bit	11 bits	52 bits	1023

  - ◇ nombre positif si bit de signe $= 0$
  - ◇ mantisse dans l'intervalle $[1,0 : 10,0[$
    (on parle ici en base deux !)
    Le seul chiffre (binaire) à gauche de la virgule est un 1 et n'est donc pas représenté

- ◇ **dans la partie exposant, ce qui est stocké, c'est la valeur vraie de l'exposant (qui désigne évidemment une puissance de deux) augmentée de la valeur donnée dans le tableau.**

Ainsi, en simple précision

- **la valeur décimale**234 **désigne un exposant vrai égal à** 107 **(**$234 - 127$**)**
- **la valeur décimale** 34 **désigne un exposant vrai égal à** -93 **(**$34 - 127$**)**
- **les valeurs** 0 **(tous les bits à** 0**) et** 255 **(tous les bits à** 1**) sont réservées et ne représentent pas les valeurs vraies d'exposant -127 et +128.**

Intervalles des valeurs

- **la plus grande valeur correspond à un exposant de valeur maximale (127) et tous les bits de la mantisse à 1, c'est-à-dire $(2 - 2^{-23})^{127}$ en simple précision et $(2 - 2^{-52})^{1023}$ en double**

- **la plus petite valeur correspond à la même configuration avec un bit de signe égal à 1, donc $-(2 - 2^{-23})^{127}$ en simple précision et $-(2 - 2^{-52})^{1023}$ en double**

Entiers sous forme flottante

Exemple : l'entier 2093787663_{dix} :

- **sa représentation binaire comme entier est**

  $01111100110011001010101000001111$

  (on a distingué les paquets de 8 bits successifs pour faciliter la lecture)

- **sa forme normalisée comme nombre réel en base deux (sans se préoccuper de sa mémorisation effective) est:**

  $1.11110011001100101010100001111$

  avec un exposant égal à 30_{dix}, c'est-à-dire 11110_{deux};

- **dans sa représentation sur 32 bits:**
  - ◇ **le nombre étant positif, le bit de signe est 0**
  - ◇ **on ajoute à l'exposant l'excès 127_{dix}, c'est-à-dire 1111111_{deux}, ce qui donne la valeur 10011101**

- **la mémorisation de ce nombre en mémoire ne permet de conserver que 23 bits (24 en fait puisque le premier 1 est *gratuit* et n'est pas enregistré), ce qui donne les 23 bits:**

  11110011001100101010100

- **finalement la représentation du nombre est obtenue par la mise bout à bout des 3 suites de bits précédentes, ce qui donne (on a distingué les paquets de 4 bits successifs pour faciliter le passage ultérieur à une expression symbolique en héxadécimal):**

  01001110111110011001100101010100

  Cette suite de 32 bits vue comme la représentation d'un nombre entier s'exprime sous forme hexadécimale de la manière suivante

  4EF99954

◇ **vérification expérimentale**

Le programme C suivant :

```
#include <stdio.h>
main( ) {
   int n = 2093787663;
   union { float f; int n;
         } val;
   val.f = 2093787663;
   printf(" n = %d\n val.f = %f\n",
         n, val.f);
   printf(" val.f = %e\n val.n = %x\n",
         val.f, val.n);
}
```

confirme ces résultats :

```
 n = 2093787663
 val.f = 2093787648.000000 // approchee
 val.f = 2.093788e+09
 val.n = 4ef99954   // hexadecimal
```

En Java, on écrirait par exemple :

```
class EntierReel {
 public static void
    main(String [ ] a){
      int n = 2093787663;
      float f = 2093787663;
      Deug.println("n = " + n +
                          "    f = " + f);
  }
}
```

dont l'exécution donne :

```
n = 2093787663    f = 2.09378765E9
```

Représentation des nombres décimaux

Si on reprend le nombre 0,3, sa forme normalisée en base deux est infinie et a la forme suivante où tous les nombres sont exprimés en base deux:

$$10^{-10} \times 1,00110011001100110011\ldots$$

Il sera donc représenté en machine avec

- **le bit de signe 0**
- **les 8 bits de la partie exposant**

  `1111111 + (-10) = 01111101`
- **23 bits extraits du développement infini de la partie après la virgule:**

  normalement `00110011001100110011001`

  mais en fait `00110011001100110011010`

  (arrondi à cause du 1 qui suit)
- **la représentation complète sur 32 bits est:** `00111110100110011001100110011010`

  soit en notation hexa `3E99999A`

ce que confirme les résultats du programme C suivant:

```
main( ) {
   union {
   float f;
   int n; } val;
   val.f = 0.3;
   printf(" val.f = %f\n val.f = %e\n
          val.n = %p\n", val.f,
          val.f, val.n);
}
```

à savoir :

```
 val.f = 0.300000
 val.f = 3.000000e-01
 val.n = 3e99999a
```

Problèmes numériques

Exemple introductif

Considérons par exemple la séquence de code Java suivante:

```
if ((0.3 - 0.2) == (0.2 - 0.1))
    Deug.println("oui");
else
    Deug.println("non");
```

Son exécution affiche `non`

L'explication du résultat est dans la représentation des 3 nombres impliqués:

Forme décimale	Forme hexadécimale	représentation binaire sur 32 bits
0.3	3E99999A	00111110 10011001 10011001 10011010
0.2	3E4CCCCD	00111110 01001100 11001100 11001101
0.1	3DCCCCCD	00111101 11001100 11001100 11001101

La représentation de `0.3 - 0.2` **est**

`3dccccce`

alors que celle de `0.2 - 0.1` **est**

`3dccccd`

L'exemple précédent met en lumière les problèmes d'imprécision introduite par la nécessité de représenter finiment des valeurs représentées infiniment.

Autres exemples :

- **dans l'exemple suivant écrit avec une boucle simple en Java:**

```
class Ajouter {
 public static void main(String[] a){
  double somme = 0;
    while (somme != 20)
       somme = somme + 0.1;
    Deug.println("Apres la boucle");
 }
}
```

l'exécution conduit à une boucle infinie (le programme ne se termine pas!): la valeur de la variable n'est jamais (exactement) 20.

- **le test de nullité d'une variable de type float ou double comme dans la séquence C ou Java suivante:**

```
if (val == 0)
    .......
```

"n'a pas de sens"

Un tel test doit être remplacé par "quelque chose" (ici en Java) de la forme

```
if (Math.abs(val) < eps)
    .......
```

Disons simplement que la valeur epsilon dépend tout à la fois de la machine, et de l'application.

- **addition de nombres d'ordres de grandeur très différents**

  Pour additionner deux nombres, il faut ajuster leurs exposants. Cet ajustement se fait sur le plus grand des deux en alignant la mantisse et en effectuant la somme. Si les ordres de grandeur sont trop éloignés, le plus petit des deux nombres sera ignoré.

  Le programme

```
class SommeReels {
 public static void main(String [ ] a){
   float reel1, reel2, somme;
   reel1 = (float) 1.2E3;
   reel2 = (float) 1.3E-2;
   somme = reel1 + reel2;
   Deug.println(somme);
   reel1 = (float) 1.2E4;
   reel2 = (float) 1.3E-4;
   somme = reel1 + reel2;
   Deug.println(somme);
 }
}
```

produit les affichages suivants:

`1200.013` **vision de 1200 + 0,013**
`12000.0` **vision de 12000 + 0,00013**

La première opération se fait sous la forme:

$(1,2 \times 10^3) + (0,000013 \times 10^3)$

et la seconde sous la forme

$(1,2 \times 10^4) + (0,0000000013 \times 10^4)$

On conoît aisément que la limitation du nombre de chiffres entraîne la perte des informations relatives au second nombre (tout cela devant être codé en binaire).

Des calculs au format double améliore les choses sur ces données.

On obtient alors l'affichage

`1200.013`
`12000.00013`

Quelques définitions utiles

Les valeurs suivantes correspondent à des situations où un calcul conduit à une valeur non représentable, c'est-à-dire de valeur absolue trop grande (*overflow*) ou une perte de précision (*underflow*) approximée par zéro:

- **negative overflow: nombres inférieurs à $-(2-2^{-23})^{127}$ (en simple précision)**
- **positive overflow: nombres positifs supérieurs à $(2-2^{-23})^{127}$ (en simple précision)**

Valeurs spéciales

- le zéro: il est représenté par la valeur nulle des champs exposants et mantisse (il existe donc +0 et un -0)

- infini positif: tous les bits du champ exposant à 1 et bits du champ mantisse à 0, bit de signe 0

- infini négatif :tous les bits du champ exposant à 1 et bits du champ matisse à 0, bit de signe 1

- faux nombres : tous les bits du champ exposant à 1 et champ mantisse non nul

- forme dénormalisée: champ exposant nul et mantisse non nulle, sans premier bit 1 implicite