NUMERATION

Rappel

Base 10

Base 10	->	Base 2	Division par 2
Base 10	->	Base 8	Divisions successives par 8
Base 10	->	Base 16	Divisions successives par 16

Base 2

Base 2	->	Base 8	Groupement sur bits 3
Base 2	->	Base 10	Tableau
Base 2	->	Base 16	Groupement sur bits 4

Base 8

Base 8	->	Base 2	Eclatement sur 3 bits
Base 8	->	Base 10	Tableau
Base 8	->	Base 16	Passer par le binaire

Base 16

Base 16	->	Base 2	Eclatement sur 4 bits
Base 16	->	Base 8	Passer par le binaire
Base 16	->	Base 10	Tableau

NUMERATION

1 La numération

La nécessité de quantifier (attribuer une grandeur mesurable), notamment pour les échanges commerciaux, s'est faite dés la structuration de la vie sociale. Les tentatives de représentation symbolique de quantités furent nombreuses; bâtons, chiffres romains, etc.. Avant que ne s'impose la numération arabe, universellement adoptée étant donné sa bonne capacité à traiter les calculs courants.

L'emploi quotidien de ce système nous fait oublier la structure et les règles qui régissent l'écriture des nombres, notamment la notion de base acquise en cours primaire.

1.1 Principe de la numération

La numération traditionnelle représente un nombre par la juxtaposition de symboles, appelés chiffres, pris parmi un ensemble. Par exemple, dans le système décimal (10), cet ensemble contient dix symboles différents.

{0,1,2,3,4,5,6,7,8,9}

On peut très bien en utiliser d'autres!

{□,●,❍,♦,❖,☒,⍓,⌘,❀,❁}

Cette dernière symbolique n'étant pas pratique nous la laisserons de coté pour reprendre nos symboles habituels.

1.2 La Base 10 (numération décimale)

Un nombre est dit de Base 10 si son type de représentation s'effectue avec 10 symboles.
{0,1,2,3,4,5,6,7,8,9}

Lorsqu'un nombre est écrit, la position respective des chiffres détermine leur poids :
..... Milliers, centaines, dizaines, unités, dixièmes, centièmes..
0,1,2,3,4,5,6,7,8,9,10,11,12,13...

Par exemple 1994

	Milliers	Centaines	Dizaines	Unités
Poids	1000	100	10	1
Nombre	1	9	9	4

...01994,000...= ... (0 * 10000) + (**1** * 1000) + (**9** * 100) + (**9** * 10) + (**4** * 1) + (0 * 0.1)

ou 3.14

3,14= **3***1+**1***0.1+**4***0.01

Pour se repérer, la virgule sépare les unités des dixièmes.

1.2.1 Rappel sur les puissances de 10

10^6	10^5	10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}
1000000	100000	10000	1000	100	10	0	0.1	0.01	0.001
Méga			Kilo			Unité			Milli

Par exemple on peut noter π (pi) $314*10^{-2}$
ce qui revient à déterminer $\pi=314*0.01=3.14$

Ce type de notation est souvent utilisé pour les NOMBRES très petit ou très grands.
Par exemple la terre est distante de la lune de $3844*10^5$ mètres.
Sans cette notation on devrait écrire: 384 400 000 mètres.

ou la masse au repos d'un électron M=0.910953*10-30 kg
M=0.000000000000000000000000000910953 kg

1.3 La Base 2

1.3.1 Présentation du binaire

En logique il est très simple de différencier 2 états, ouvert fermé, éteint allumé…
Donc 2 états, donc 2 symboles.. {0,1} .
On appel aussi la base 2 le Binaire, car il n'y a que 2 élément {1,0}

Passage en base 2.

BASE 10	BASE 2
0000	0000
0001	0001
0002	0010
0003	0011
0004	0100
0005	0101
0006	0110

Le principe reste le même qu'en base 10.

La position du chiffre détermine le poids du nombre.

Poids	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	16	8	4	2	1	1/2	1/4	1/8	1/16	Base 10
$N1_{(2)}$	0	0	1	0	0	0	0	0	0	=4
$N2_{(2)}$	1	0	1	0	0	1	0	0	0	=20,5
$N3_{(2)}$	0	0	0	0	1	0	1	0	0	=1,25

```
N1=0*16 + 0*8 + 1*4 + 0*2 + 0*1 + 0*0.5 + 0*0.25 + 0*0.125 + 0*0.0625 = 4
N2=1*16 + 0*8 + 1*4 + 0*2 + 0*1 + 1*0.5 + 0*0.25 + 0*0.125 + 0*0.0625 = 20,5
N3=0*16 + 0*8 + 0*4 + 0*2 + 1*1 + 0*0.5 + 1*0.25 + 0*0.125 + 0*0.0625 = 1,25
```

(Exercices de conversion)

Le passage de la base 10 à la base 2 se fait par divisions successives par 2 pour la partie entière, et par la multiplication par 2 pour la partie fractionnaire

Soit le nombre 20,375 en base 10

20 |_ 2
0 10 |_2
0 5 |_2
1 2 |_ 1
0 **1**

Le résultat est lu à "l'envers" : 10100

Pour la partie fractionnaire, on effectue des multiplications par 2
Ainsi, pour 0,375

0,375 * 2 = **0**,75
0,75 * 2 = **1**,50
0,50 * 2 = **1**

Le résultat est "lu à l'endroit" : 0,**011**

$20{,}375_{(10)} = 10100{,}011_{(2)}$

(exercices "système Binaire")

1.3.2 Les opérations simples en base 2

Toutes les opérations arithmétiques de base connues en base 10 sont applicables en base 2.

L'addition:

Réaliser la somme de N1 avec N2

$N1=0110_{(2)}$ ==> $N1= 6_{(10)}$
$N2=1101_{(2)}$ ==> $N2= 13_{(10)}$

Donc le résultat est 19 (d) ==> R=10011 (b)

retenue		1	1			
nombre 1	N1=		0	1	1	0
nombre 2	N2=	+	1	1	0	1
résultat	=	1	0	0	1	1

Comme dans le système décimal (base 2), lorsqu'il y a un dépassement (plus de symboles au dessus) on rajoute une colonne à l'extrême gauche de poids plus fort.

$1_{(2)}+1_{(2)}=10_{(2)}$ (on lira 'un zéro' et non pas dix!)

Réaliser la somme de N1 + N2 + N3 + N4= 41

$N1=0110_{(2)}$ ==> $N1= 6_{(10)}$
$N2=1101_{(2)}$ ==> $N2= 13_{(10)}$
$N3=0111_{(2)}$ ==> $N3= 7_{(10)}$
$N4=1111_{(2)}$ ==> $N3= 15_{(10)}$

	1						
	1	*0*					
		1	*0*	*1*			
		0	1	1	0		
		1	1	0	1		
		0	1	1	1		
	+	1	1	1	1		
1	0	1	0	0	1	=	$41_{(10)}$

(Exercices Addition)

La soustraction:

Réaliser la différence de N1 avec N2, où N1>N2

N1=$1101_{(2)}$ ==> N1= $13_{(10)}$
N2=$0110_{(2)}$ ==> N2= $6_{(10)}$

Donc le résultat est $7_{(10)}$ ==> R=$111_{(2)}$

nombre 1	N1=		1	$_1$1	$_1$0	1		
nombre 2	N2=	-	$_{1+}$0	$_{1+}$1	1	0		
résultat			0	1	1	1	=	$7_{(10)}$

calculez $3_{(10)}$ -$4_{(10)}$ = -1 sur 4 bits (ici, N1<N2 !)

		10	10	1	1		
	-1	1+0	1	0	0	=	? infini !
…	1	1	1	1	1		

(Exercices soustraction)

La multiplication :

Réaliser la multiplication de N1 avec N2

N1=$1101_{(2)}$ ==> N1= $13_{(10)}$
N2=$0110_{(2)}$ ==> N2= $6_{(10)}$

nombre 1	N1=		1	1	0	1		
nombre 2	N2=	*	0	1	1	0		
résultat			0	0	0	0		
1	1	1	1	0	1	.		
	1	1	0	1	.	.		
0	0	0	0	.	.	.		
1	0	0	1	1	1	0	=	$78_{(10)}$

(Cf exercices Multiplication)

La division :

Le principe est le même qu'en base 10, mais peut être moins naturel !
(La division entière ou division euclidienne est une opération qui à deux entiers naturels appelés **dividende** et **diviseur** associe deux autres appelés **quotient** et **reste**)

```
10110  |_ 101
101      100
   01
     10
      10                donc 1010 = 101*100+10
```

1.4 La base 8 (octal)

Cette base est plutôt utilisée par les informaticiens. Les symboles sont {0,1,2,3,4,5,6,7}.

Poids	8^4	8^3	8^2	8^1	8^0	8^{-1}	8^{-2}	8^{-3}	8^{-4}	
	4096	512	64	8	1	1/8	1/64	1/512	1/4096	Base 10
N1(B8)	7	7	7	7	7	0	0	0	0	=32767
N2 (B8)	1	2	1	0	0	1	0	0	0	=5184,125
N3 (B8)	0	0	0	0	1	0	1	0	0	=1,015625

Base 2 -> 8
Le passage de la base 2 à la base 8 se fait de façon immédiate en groupant les chiffres par 3, ainsi :

$1011101,01101_{(2)} = 1 \mid 011 \mid 101 , 011 \mid 010 = 1\ 3\ 5 , 3\ 2\ _{(8)}$

Base 10 -> 8
Le passage de la base 10 à la base 8 se fait par divisions successives par 8

```
450 |_8          56 |_ 8        7|_8
      56                 7             0
 50               0             7
  2
```

Soit $450_{(8)} = 702_{(10)}$

Base 8 -> 2
Le passage de la base 8 à la base 2 se fait en "éclatant" chaque chiffre octal en un nombre binaire codé sur 3 bits : exemple $743\ _{(8)}$

```
7      4      3
111    101    011
```

Soit $743_{(8)} = 111101011_{(2)}$

(Cf exercices Octal)
(PB:3h: exercices de rappel)

1.5 La base 16 (hexadécimal)

La base 16 est apparue avec la logique micro programmée et les microprocesseurs. L'ensemble des symboles contient 16 éléments. Comme il n'est pas possible traditionnellement d'écrire, avec un seul caractère, un chiffre dont la valeur est supérieur à 9, l'ensemble comporte des lettres.

Par convention, A est équivalent à 10, B à 11 et ainsi de suite. L'ensemble des symboles de la base 16 est donc: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

La base 16 est une forme contractée de la base 2.

Poids	**16^3**	**16^2**	**16^1**	**16^0**	**16^{-1}**	**16^{-2}**	**16^{-3}**	
	4096	256	16	1	1/16	1/256	1/4096	Base 10
N1	0	1	F	F	0	0	0	=511
N2	0	1	2	A	1	0	0	=298,0625
N3	0	0	0	1	0	0	0	=1,00

Table de conversion:

Décimale	Binaire	Octal	Hexadécimale
0	00000	0	0
1	00001	1	1
2	00010	2	2
3	00011	3	3
4	00100	4	4
5	00101	5	5
6	00110	6	6
7	00111	7	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10
17	10001	21	11

Base 10 -> 16
Le passage de la base 2 à la base 16 se fait par divisions successives par 16
249 |_ 16 15|_16
89 15 **15** 0 soit $249_{(10)} = F9_{(16)}$

9

Base 2 -> 16
Le passage de la base 2 à la base 16 se fait de façon immédiate en groupant les chiffres par 4, ainsi :
$1011101,01101_{(2)}$ = 101 | 1101 , 0110 | 10 = 5 D , 68 $_{(16)}$

Base 16 -> 2
Le passage de la base 16 à la base 2 se fait en "éclatant" chaque chiffre octal en un nombre binaire codé sur 4 bits
A B C $_{(16)}$
1010 1011 1100
Soit $ABC_{(16)} = 101010111100_{(2)}$

Base 16 -> 8
Le plus simple est de passer par la base 2

(exercices hexa)

2 La représentation des nombres

Ces représentations font appel à des règles qui ont été adoptées pour des raisons de facilité de traitement matériel et logiciel.

2.1 Représentation des nombres entiers non signés

La technique consiste uniquement à transformer les nombres décimaux en binaires et de stocker chaque chiffre binaire sur un bit

Exemple : $74_{(10)} = 01001010_{(2)}$

2.2 Représentation des nombres entiers signés : signe et valeur absolue

La solution la plus simple. Un élément binaire est ajouté au module pour la représentation du signe. Habituellement, il est utilisé la correspondance suivante:

0 ==> +
1 ==> -

Et le signe est placé à gauche du module (représentation sur 8 bits):

Signe	b6	b5	b4	b3	b2	b1	b0

Exemple : $-23_{(10)}$ = $10010111_{(2)}$

Sur 8 bits, on peut coder 2^7 nombre positifs (de +0 à 127) et 2^7 nombre négatifs (-0 à -127), on va donc de -127 à +127, avec 2 zéros !

Seulement avec cette représentation, il y a un problème, on peut représenter 0 de deux façons : 00000000 et 10000000 sont respectivement égaux à 0 et -0.

Exercices :

Si on effectue une opération arithmétique entre des nombres négatifs et positifs, on obtient un résultat erroné. Exemple 3-4 = 3+(-4) = -1

Dans la représentation "signe et valeur absolue“, $-4_{(10)} = 10000100_{(2)}$

Hors 3 + (-4) = (-7) au lieu de (-1)

	0	0	0	0	0	0	1	1	
+	1	0	0	0	0	1	0	0	
	1	0	0	0	0	1	1	1	= **-7 (10) !!!**

(Exercices "signe et valeur absolue")

2.2.1 Représentation des nombres en complément à 1

Le complément à un est l'opération qui inverse la valeur de chacun des bits d'un nombre binaire. Il est la première étape du complément à deux.

Pour obtenir un nombre à complément à 1 il suffit de permuter les « 0 » en « 1 » et les « 1 » en « 0 ».

Par exemple 10010101 donne en complément à « 1 » 01101010

On notera le N en complément à 1 : $\overline{N}$

2.2.2 Représentation des nombres en complément à 2

Cette représentation évite les inconvénients de la représentation classique, où apparaissait deux zéros. Ce procédé nous permettra de créer des nombres négatifs.

$(-N)=\overline{N}+1$

Exemple :
Pour obtenir la représentation sur 16 bits de l'entier $-10_{(10)}$ en complément à deux, on part de la représentation de $10_{(10)}$ en binaire, soit 0000 0000 0000 1010

On complémente :

1111 1111 1111 0101

Et on ajoute 1

0000 0000 0000 0001

1111 1111 1111 0110 soit $FFF6_{(16)}$ ou $177768_{(8)}$

Exercices : calculez 3-4

- 3-4 = 3+(-4)
- On va coder (-4)

On prend le nombre 4 :	0	0	0	0	0	1	0	0
						1	*1*	
On inverse les bits :	1	1	1	1	1	0	1	1
On ajoute 1 : 11111100	0	0	0	0	0	0	0	1
	1	1	1	1	1	1	0	0 = $-4_{(10)}$

- Le bit de signe (bit de poids fort) est automatiquement mis à 1 par l'opération d'inversion.

Calculons maintenant 3+(-4) :

	0	0	0	0	0	0	1	1	$= 3_{(10)}$
+	1	1	1	1	1	1	0	0	$= -4_{(10)}$
	1	1	1	1	1	1	1	1	

Conclusion : $3+(-4)=11111111_{(2)}$

Si l'on obtient un résultat négatif (bit de poids fort à 1), il faut **refaire le complément à deux** pour obtenir l'équivalent décimal positif

Le complément à deux de 11111111 est 00000001 soit 1 en décimal,
donc 11111111 = (-1) en décimal.

La représentation en complément à 2 permets de coder, sur 8 bits, les n ombres de -128 à +127, avec seul 1 zéro !

Exemple sur 4 bits :

	Non signé	Représentation classique	Représentation en 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 complément à 2)

2.3 La représentation BCD (ou DCB "décimal codé binaire")

Dans certains cas, on peut préférer aux représentations binaires décrites ci-dessus un codage qui permette de passer plus facilement du code décimal d'un nombre à sa représentation en machine. Le codage couramment utilisé porte le nom de BCD (Binary Coded Decimal).

Chaque chiffre d'un nombre représenté en base 10 est codé sur quatre bits (un quartet)
Avec n bits (n multiple de 4), il est possible de représenter les nombres entre 0 et $10^{n/4}-1$

Ainsi, l'entier 1992 sera représenté par : 0001 1001 1001 0010 $_{(2)}$

0	0000
1	0001
3	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Les quartets 1010 à 1111 sont des combinaisons inutilisées donc illégales.

Le codage du signe peut être fait séparément en lui réservant un demi-octet (*nibble* ou *quartet*) auquel on affectera une valeur en dehors de l'intervalle [0,9].

Par convention,
- le signe "-" sera codé 1101, càd $D_{(16)}$
- le signe "+" sera codé 1100, càd $C_{(16)}$

alors -1992 sera codé $D1992_{(16)}$

Remarques
6+5 = 0110 + 0101 = 1011 ce qui est interdit en BCD !
Il faut donc ajouter 6 (les 6 codes non significatifs) ce qui donne 1011+0110 = 1 0001 ce qui donne bien 11 en décimal ! (0001 0001)

Exercice :
Calculez la valeur décimale de : **0001100110010100 $_{(2)}$**

0001		1001		1001		0100 (2)
1*1000	+	9*100	+	9*10	+	4*1
1994						

Calculez la valeur décimale de : **1010 $_{(2)}$**
1010 ne fait pas 10 $_{(10)}$ car 1010 fait parti des valeurs interdites
10 = 0001 0000 !

Calculez 12+6:

	00010010	(12)
+	00000110	(6)
	00011000	(=18)

Calculez 22+8

	00100010	(22)
+	00001000	(8)
	0010**1010**	Valeur interdite ! on rajoute 6 !
+	00000110	(6)
	00110000	(=30)

(Exercices "BCD")

Il existe des variantes du codage BCD:

Le code Excess 3 (XS 3)

Le codage binaire excédent 3 qui consiste à représenter le chiffre à coder + 3.
Par exemple, pour coder 48 :

	4	8
	+3	+3
	7	11
soit	0111	1011

En Excess 3, les codes non valides sont : 0000 (zéro),0001 (un),0010 (deux),1101 (treize), 1110 (quatorze) et 1111 (quinze)

Le code Aïken

Le code Aiken où 0, 1, 2, 3, 4 sont codés comme en BCD et 5, 6, 7, 8, 9 sont codés de 1011 à 1111. Il permet d'obtenir le complément à 9 en permutant les 1 et les 0.

2.4 Code à distance unité (code Gray ou code Réflechi)

Ce codage permet de ne faire changer qu'un seul bit à la fois quand un nombre est augmenté d'une unité. Le code distance à unité le plus fréquemment utilisé est le code Gray ou code réfléchi.

Le code Gray est un code non pondéré (aucun poids est affecté à la position d'un bit. On convient simplement d'un tableau de correspondances entre les objets à coder et une représentation)

Exemple le code Gray : Pour passer d'une ligne à la suivante, on inverse le bit le plus à droite possible conduisant à un nombre nouveau

Base 10	Code Gray
0	0 0 0 0
1	0 0 0 1
2	0 0 1 1
3	0 0 1 0
4	0 1 1 0
5	0 1 1 1
6	0 1 0 1
7	0 1 0 0
8	1 1 0 0
9	1 1 0 1
10	1 1 1 1

Ce code à été imaginé pour éviter les problèmes de transition car lorsque l'on passe de ***n*** à ***n+1*** dans un code binaire on peut lire plusieurs états parasites, exemple pour passer de 3 à 4

3 =====> 0011
0010 états parasite de transition
0000 états parasite de transition
4 =====> **0100**

On peut donc lire 2 puis 0 et enfin 4!!

Alors que en code Gray le problème disparaît.

3 =====> 0010
4 =====> **0110**

Ce code gray est surtout utilisé pour des capteurs de positions, en effet, un seul bit change à chaque fois, ce qui évite toute ambiguïté de lecture.

Le code Gray sert également dans les tables de Karnaugh utilisées lors de la conception de circuits logiques.

(Exercices Code Gray)

2.5 Les nombres réels

Deux méthode permettent de représenter les nombres réels : les représentations en virgule fixe et virgule flottante

2.5.1 Représentation en Virgule fixe

Cette technique fixe arbitrairement la position de la virgule "entre" deux chiffres de la représentation binaire

Exemple : $12,75_{(10)}$ à représenter en virgule fixe sur un octet.
On suppose que la virgule si situe entre le bit 3 et 4
$12_{(10)} = 1100_{(2)}$
$0,75_{(10)} = 0,11_{(2)}$
1100,1100

2.5.2 Représentation en Virgule flottante

Les nombres à virgule flottante sont les nombres les plus souvent utilisés dans un ordinateur. Ce sont des approximations rationnelles de nombres réels.

Les nombres à virgule flottante possèdent

- un signe **s** (dans {-1, 1}),
- une mantisse entière **m** (parfois appelée significande)
- un exposant ***e***.

Un tel triplet représente un réel $s*m*b^e$ où b est la base de représentation (parfois 2, mais aussi 16 pour des raisons de rapidité de calculs, ou éventuellement toute autre valeur).

En faisant varier **e**, on fait « flotter » la virgule décimale. Généralement, ***m*** est d'une taille fixée.

Ceci s'oppose à la représentation dite en virgule fixe, où l'exposant **e** est fixé.

Rappel :
π (pi) 3,14
On peut noter $\pi = 314*10^{-2}$
$\pi = 0,314*10^{1}$
$\pi = 0,00314*10^{3}$
Mantisse exposant

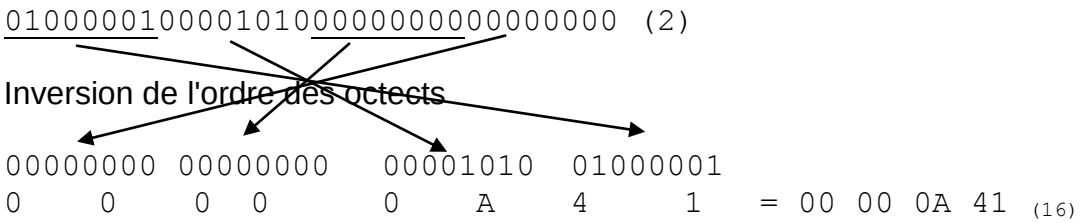
Les différences de représentation interne des nombres flottants d'un ordinateur à un autre obligeaient à reprendre finement les programmes de calcul scientifique pour les porter d'une machine à une autre jusqu'à ce qu'un format normalisé soit proposé par l'IEEE.

Big et Little Endian

Description d'une façon (parmi d'autres) dont on stocke les nombres dans plusieurs octets : l'octet de poids fort est stocké avant l'octet de poids faible. Utilisé sur la famille des 680x0 de Motorola, c'est la façon habituelle de voir un nombre. Opposé à little endian.

La représentation en Big Endian ou Little Endian consiste donc à déterminer l'arrangement des octets.

Exemple :



Il n'y a pas de solution miracle au problème de l'arrangement des octets (Endian). Chacun doit s'entendre sur le format d'emmagasinage des données. Un des processeurs aura à traduire (changer l'ordre) des données provenant de l'autre processeur.

Certains nouveaux processeurs (tel que le PowerPC) peuvent être Bi-Endian (ils peuvent utiliser un format ou l'autre) -- mais habituellement le système d'exploitation qui les utilise est dépendant d'un certain type «d'Endian». Alors ils sont fixés sur une méthode et rarement vont-ils utiliser l'autre.

Quelques faits : les processeurs Intel (x86 et Pentium) sont Little Endian et les processeurs Motorola (la série 680x0) sont Big Endian. Le MacOS est Big Endian et Windows est Little Endian.

2.5.3 Exemple de représentations flottantes

Norme IEEE 754 (Institue of Electrical and Electronics Engineers)

La norme IEEE 754 (reprise par la norme internationale CEI 60559) spécifie deux formats de nombres en virgule flottante et les opérations associées. La quasi-totalité des architectures d'ordinateurs actuelles, y compris PowerPC, et AMD64, incluent une implémentation matérielle des calculs sur flottants IEEE, directement dans le microprocesseur, garantissant une exécution rapide.

Les flottants IEEE peuvent être codés sur 32 bits (« simple précision ») ou 64 bits (« double précision »). Il est aujourd'hui très rare que des programmes utilisent la simple précision, en tout cas sur station de travail. La répartition des bits est la suivante :

	Signe	**Exposant**	**Mantisse**
Simple précision	1 bit	8 bits	23 bits
Double précision	1 bit	11 bits	52 bits

Simple précision : **s**eeeeeeee**mmmmmmmmmmmmmmmmmmmmmmm**

La valeur d'un nombre ainsi codé est donc :

$(-1)^S * 2^{(E-127)} * (1,M)$ **pour les nombres codés en simple précision**

$(-1)^S * 2^{(E-1023)} * (1,M)$ **pour les nombres codés en double précision**

Exemple :

On désire coder $2,5_{(10)}$ en flottant de type *short real dans la norme IEEE-754*

- Convertir 2,5 en binaire = $10,1_{(2)}$
- Normaliser -> $1,01*2^1$ (mantisse 01, exposant 1)
- Calculer la représentation de l'exposant d'après la formule : **$(-1)^S * 2^{(E-127)} * (1,M)$**

  e-127 =1
  e =127+1
  e =128

- Convertir l'exposant en binaire $128_{(10)} = 10000000_{(2)}$
- Représenter la mantisse en binaire sur 23 bits= 01000000000000000000000
- Calculer le bit de signe, ici s=0
- La représentation est donc :
  01000000001000000000000000000000
  seeeeeeee**mmmmmmmmmmmmmmmmmmmmmmm**

On désire coder $1_{(10)}$ en flottant de type *short real dans la norme IEEE-754*

- Convertir 1 en binaire = $1,0_{(2)}$
- Normaliser -> $1,0*2^0$ (mantisse 0, exposant 0)

- Calculer la représentation de l'exposant d'après la formule : $\mathbf{(-1)^S * 2^{(E-127)} * (1,M)}$

  e-127=0
  e=127

- Convertir l'exposant en binaire $127_{(10)} = 01111111_{(2)}$

- Calculer le bit de signe, ici s=0

- La représentation est donc 0 | 01111111 | 000 0000 0000 0000 0000 0000

- 0011 1111 1000 0000 0000 0000 0000 0000
  3F 80 00 00 $_{(16)}$

Précautions d'emploi

Certains nombres ne peuvent pas être représentés :

- Les nombres positifs supérieures à $1,11111… * 2^{127}$
  127 est la plus grande caractéristique codable

- Les nombres négatifs inférieurs à -3,4 * 1038

- Les nombres trop proches de 0
  La plus petite caractéristique codable est 2^{-126}, la plus petite mantisse est 0,000000…1 (2^{-23})
  Le plus petit nombre codable est donc $2^{-149} = 1,5*10^{-45}$

La représentation en virgule flottante est donc source d'imprécision

Rappels

a-b=c ↔ a=c+b
a+b=c ↔ a=c-b

a-b=-c ↔ a=-c+b
a+b=-c↔ a=-c-b

3 La représentation des caractères

Les caractères (lettres, chiffres, symboles de ponctuation) sont généralement codés sur un octet. Les codes les plus fréquemment utilisés sont le code ASCII (*American Standard Code for Information Interchange*) et l'EBCDIC (*Extended Binary Coded Decialm Interchange Code*)

3.1 ASCII (*American Standard Code for Information Interchange*)

Le code ASCII représente chaque caractère sur 7 bits (on parle parfois de code ASCII étendu sur 8 bits). On peut donc coder 2^7 = 128 caractères.

- Les caractères de 0 à 31 ainsi que le 127 ne sont pas affichables, et correspondent à des directives de terminal ou des fonctions de commandes
- Le caractère 32 est l'espace blanc.
- Les autres correspondent aux chiffres, aux lettres majuscules et minuscules et à quelques symboles de ponctuation.

	000	**001**	**010**	**011**	**100**	**101**	**110**	**110**
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Table ASCII 7 bits

3.2 ASCII Etendu

Comme ce standard n'utilise que 7 bits (au lieu de 8), il reste 128 caractères disponibles pour les langues nationales. Par exemple, l'ISO 8859-1, appelée aussi Latin-1, étend l'ASCII avec les caractères accentués utiles aux langues d'Europe occidentale comme le français. (pages de code)

- ISO 8859-1 (latin-1 ou européen occidental) — probablement la partie la plus largement utilisée de ISO 8859, couvrant la plupart des langues européennes occidentales : l'allemand, l'anglais, le basque, le catalan, le danois, l'écossais, l'espagnol, le féringien, le finnois (partiellement[2]), le français (partiellement[2]), l'islandais, l'irlandais, l'italien, le néerlandais (partiellement[1]), le norvégien, le portugais, le rhéto-roman et le suédois, certaines langues européennes sud-orientales (l'albanais), ainsi que des langues africaines.

  Le symbole de l'euro est dans la version révisée ISO 8859-15 (latin-9).

- ISO 8859-2 (latin-2 or européen central) — supporte les langues d'Europe centrale ou de l'Est basées sur un alphabet romain, y compris le polonais, le tchèque, le slovaque, le slovène et le hongrois. Le symbole de l'euro manquant est présent dans la version ISO 8859-16.

- ISO 8859-3 (latin-3 or européen du Sud) — le turc, le maltais, et l'espéranto ; largement supplanté par ISO 8859-9 pour le turc, et par Unicode pour l'espéranto.

- ISO 8859-4 (latin-4 or européen du Nord) — l'estonien, le letton, le lituanien, le groenlandais, et le sami.

- ISO 8859-5 (cyrillique) — Couvre la plupart des langues slaves utilisant un alphabet cyrillique, y compris le biélorusse, le bulgare, le macédonien, le russe, le serbe et l'ukrainien.

- ISO 8859-6 (arabe) — Couvre les glyphes arabes les plus communs, mais pas tous.

- ISO 8859-7 (grec) — Couvre la langue grecque moderne (orthographe monotonique). Peut être utilisé aussi pour le grec ancien écrit sans les accents ou dans l'orthographe monotonique, mais manque les signes diacritiques pour l'orthographe polytonique. Ceux-ci ont été introduits avec Unicode.

- ISO 8859-8 (hébreu) — Couvre l'alphabet hébraïque moderne tel qu'il est utilisé en Israël. En pratique, deux codes différents existent : logique et visuel.

- ISO 8859-15 (latin-9 ou parfois de façon impropre latin-0) — une révision de 8859-1 qui abandonne quelques symboles peu utilisés, les remplaçant avec le symbole d'Euro € et les lettres Š, š, Ž, ž, Œ, œ, et Ÿ, qui complète la couverture du français et du finlandais.

- ISO 8859-16 (latin-10 or européen du Sud-Est) — pour l'albanais, le croate, le hongrois, l'italien, le polonais, le roumain et le slovène, mais aussi le finlandais, le français, l'allemand et le gaélique irlandais (en nouvelle orthographe). Cette police mise plus sur les lettres que les symboles. Le signe de monnaie est remplacé avec le symbole d'Euro. Pour autant, le format n'est pas totalement universel :

3.3 Fichiers ASCII

L'ACSII n'est pas un format totalement universel :
- sous Mac OS (Apple Macintosh), la fin de ligne est indiquée par un retour de chariot (CR)
- sous Linux, la fin de ligne est indiquée par un saut de ligne (LF)
- sous Microsoft Windows, la fin de ligne est indiquée par un retour chariot suivi d'un saut de ligne (CRLF).

Ainsi, lorsque l'on ouvre un fichier ASCII créé par un système sur un autre système, il faut en général faire de la mise en forme (c'est-à-dire refaire les fin de ligne) afin de pouvoir l'afficher et le lire de manière confortable. Toutefois, cela ne perturbe en général pas les programmes utilisant les fichiers ASCII.

3.4 Exemples

Exemple : la chaîne de caractère "Ordinateur" sera codée :

O	r	d	i	n	a	t	e	u	r
4F	72	64	69	6E	61	74	65	75	$72_{(16)}$

Exemple : la séquence binaire 10000011001100100011001001111
Représente le message ALLO

Quelques exemples de représentations internes sur une machine, où les entiers et réels sont codés sur 32bits, en complément à 2 pour les entiers et en format IEEE-754 pour les réprésentations flottantes. Les caractères sont codés en ASCII

Information à coder	Nombre de bits occupés	Représentation interne HEXA
Chiffre 1 en entier	32	00000001
Chiffre -1 en entier	32	FFFFFFF
Chiffre 1 en flottant	32	Exp 7F, mantisse 0, signe 0
Chiffre -1 en flottant	32	Exp 7F, mantisse 0, signe 1
Caractère "1"	8	31
Caractère "+1"	16	2B31
Caractère "-1"	16	2D31
Caractère "1.0"	24	312E30
Caractère "-1.0"	32	2D312E30

3.5 EBCDIC (Extended Binary Coded Decimal Interchange Code)

L'EBCDIC est un mode de codage des caractères sur 8 bits créé par IBM à l'époque des cartes perforées. Il existe 6 versions différentes, incompatibles entre elles. Ce mode de codage a été critiqué pour cette raison, mais aussi parce que certains caractères de ponctuation ne sont pas disponibles dans certaines versions.

3.6 UNICODE

Unicode est un standard informatique qui vise à donner à tout caractère de n'importe quel système d'écriture de langue un identifiant numérique. Unicode a été développé dans le but de remplacer l'utilisation de pages de code nationales. Dans la pratique, tous les systèmes d'écriture ne sont pas encore présents, car un travail de recherche documentaire auprès de spécialistes peut encore s'avérer nécessaire pour des caractères rares ou des systèmes peu connus.

Unicode accepte trois formes de présentation pour représenter un caractère (au sens de symbole) : l'UTF-8, l'UTF-16 et l'UTF-32. Le chiffre après UTF représente le nombre de bits sur lequel le caractère est codé

4 La représentation des Images

Toute image affichée sur un écran ou une imprimante est constituée de pixels. Le pixel ou point est l'unité de base d'une image numérique. Son nom provient de l'expression anglaise "picture element", c'est-à-dire, « élément d'image » ou « point élémentaire ».

4.1 BITMAP : Images 24 bits (ou « couleurs vraies »)

Chaque point de l'image est mémorisé. Ces images sont très gourmandes en mémoire compte tenu que chaque pixel est codé sur un bit (noir & blanc) ou sur 3 octets pour les images couleur.

Couleur
Chaque octet correspond à l'une des 3 couleur primaires (Rouge, Vert, Bleu). Les 256 couleurs codables sur un octet correspondent au poids de chacune des couleurs primaires qui compose le point coloré.

Ce système donne un total de 16,5 millions de pixels codables (256*256*256), ce qui est largement suffisant car l'œil humain est loin de pouvoir en discerner autant.

R	V	B	Couleur
0	0	0	noir
0	0	1	nuance de noir
255	0	0	rouge
0	255	0	vert
0	0	255	bleu
128	128	128	gris
255	255	255	blanc

Calculez la place mémoire nécessaire pour réprésenter les images suivantes :

- La résolution du VGA est de 640 x 480, soit 307 200 pixels (900Ko);
- La résolution du Super-VGA est de 800 x 600, soit 480 000 pixels (1406 Ko);
- La résolution du XGA est de 1 024 x 768, soit 786 432 pixels(2,25 Mo)

4.2 BITMAP : Images à palettes, images en 256 couleurs (8 bits)

On utilise souvent un autre codage des images BITMAP, le codage à l'aide de la "palette de couleur": le logiciel détermine 256 couleurs importantes dans l'image, ces couleurs sont codées sur 3 octets (mais une seule fois), ensuite chaque pixel est codé sur 1 octet qui correspond à une entrée dans la palette de couleurs

De cette façon, on divise à peu près par 3 l'encombrement mémoire d'une image, mais on perd en nuance et contraste. C'est la technique utilisée pour les images BMP.

4.3 BITMAP : Images avec gestion de la translucidité (canal alpha)

On peut attribuer à une image un canal supplémentaire, appelé canal alpha, qui définit le degré de transparence de l'image. Il s'agit d'un canal similaire aux canaux traditionnels définissant les composantes de couleur, codé sur un nombre fixe de bits par pixel (en général 8 ou 16). On échelonne ainsi linéairement la translucidité d'un pixel, de l'opacité complète à la transparence.

4.4 Les images vectorielles

Les images sont représentées à l'aide d'équations mathématiques, portions de droite et de courbes. Un cercle sera déterminée par les coordonnées du centre et la valeur du rayon, avec éventuellement la couleur et l'épaisseur du trait.

L'intérêt de cette méthode est la possibilité de modifier la taille du dessin dans altérer la définition et les proportions. (ex: WMF)

L'usage de prédilection de ce type d'images concerne les schémas qu'il est possible de générer avec certains logiciels de CAO (Conception Assistée par Ordinateur) comme AutoCAD ou CATIA. Ce type d'images est aussi utilisé pour les animations Flash, utilisées sur Internet pour la création de bannières publicitaires, l'introduction de sites web, voire des sites web complets.

4.5 Quelques Format d'images

4.5.1 GIF (Graphics Interchange Format)

Le Graphics Interchange Format (littéralement « format d'échange de graphiques »), plus connu sous l'acronyme GIF, est un format d'image numérique couramment utilisé sur le Web.

GIF a été mis au point par CompuServe en 1987 pour permettre le téléchargement d'images en couleur. Ce format utilise l'algorithme de compression LZW, nettement plus efficace que l'algorithme RLE utilisé par la plupart des formats alors disponibles (PCX, ILBM puis BMP).

LZW
LZW (Cet algorithme est appelé « à dictionnaire », car il se base sur la répétition de chaînes de caractères dans un même flux)

RLE
On dispose d'un compteur, en général sur un octet, indiquant combien de points blancs ou noirs se suivent. Exemple : WWWBWWWWWWWWWWBBBBBWBBB
3W1B10W5B1W3B

4.5.2 JPEG

Bien que plus intéressant pour des photographies ou des images lourdes, la compression JPEG provoque une perte d'information (algorithme de compression destructif) pouvant aboutir à une perte de qualité visible si l'utilisateur privilégie un taux de compression élevé, particulièrement lorsque l'image contient des changements nets de couleur ou peu de couleurs (par exemple des logos, captures d'écran, diagrammes, ...). Le format JPEG ne gère ni les animations ni la transparence

4.5.3 TIFF

TIFF est un format extrêmement flexible.
Il est notoirement connu pour permettre l'enregistrement des données multi-octets au format big endian ou little endian.
Il permet d'utiliser de nombreux types de compression, avec ou sans perte de données : (brut, PackBits, LZW, CCITT Fax 3 et 4, JPEG.)
Cette considérable flexibilité fait que TIFF est utilisé dans des applications très diverses, des scanners industriels aux appareils photo numériques en passant par les imprimantes. En revanche cela fait également qu'il n'existe pas de logiciel capable d'afficher n'importe quelle image TIFF. Un fichier TIFF commence par les deux caractères ASCII MM pour big endian ou II pour little endian. Les deux octets suivants représentent 42, en big endian ou little endian

4.5.4 PNG (Portable Network Graphics)

Le PNG (Portable Network Graphics) est un format d'images numériques libre de droit, qui a été créé pour remplacer le format propriétaire GIF, dont la compression était soumise à un brevet. Le PNG est un format non destructeur spécialement adapté pour publier des images simples comprenant des aplats de couleurs (surface de couleur uniforme). Compression Deflate (Deflate est un algorithme de compression de données sans pertes qui couple l'algorithme LZ77 et le codage de Huffman (compression de type statistique))

Lorsque l'image **PNG** utilise une palette de 256 couleurs maximum, il n'est alors possible d'utiliser qu'un seul niveau de transparence (totalement transparent ou totalement opaque).

RAPPEL

	Type (bitmap/ vectoriel)	Compression des données	Nombre de couleurs supportées	Affichage progressif	Animation	Transparence
JPEG	Bitmap	Oui, réglable (avec perte)	16 millions	Oui	Non	Non
JPEG2000	Bitmap	Oui, avec ou sans perte	16 millions	Oui	Non	Non
GIF	Bitmap	Oui, Sans perte	256 maxi (palette)	Oui	Oui	Oui
PNG	Bitmap	Oui, sans perte	Palettisé (256 couleurs ou moins) ou 16 millions	Oui	Non	Oui (couche Alpha)
TIFF	Bitmap	Compression ou pas avec ou sans pertes	de monochrome à 16 millions	Non	Non	Oui (couche Alpha)
SVG	vectoriel	compression possible	16 millions	* ne s'applique pas	Oui	Oui (par nature)

4.6 Protection des Droits d'Auteurs

Pour tenter de faire respecter le droit d'auteur (en France) et le copyright (dans presque tous les autres pays), il existe des techniques de marquage numérique d'une image

4.6.1 Protection par signature visible

Cette technique consiste à intégrer une indication sur l'image, par exemple l'organisme ou l'auteur à qui appartient l'image, afin de dissuader les pirates de s'en servir. L'inconvénient

de cette méthode est qu'il est très facile d'éliminer ce type de tatouage avec un outil de traitement d'images, puisque le tatouage est visible.

4.6.2 Protection par signature cryptée

Cette technique consiste à cacher le tatouage dans les données de l'image. Cette approche a l'avantage de ne pas gêner la lecture de l'image par le simple spectateur tout en permettant une facile identification.

L'auteur en tire un avantage complémentaire : l'éventuel pirate inattentif ne sera pas tenté de retirer ou modifier la signature ; le pirate plus volontaire verra son activité illégale rendue un peu plus difficile ou facilement prouvable (par la seule présence du tatouage).

Sténographie :

Usage des bits de poids faible d'une image
L'idée est de prendre un message et de le modifier de manière aussi discrète que possible afin d'y dissimuler l'information à transmettre. Le message original est le plus souvent une image.

La technique de base, dite LSB pour Least Significant Bit, consiste à modifier le bit de poids faible des pixels codant l'image : une image numérique est une suite de points, que l'on appelle pixel, et dont on code la couleur a l'aide d'un triplet d'octets par exemple pour une couleur RGB sur 24 bits. Chaque octet indique l'intensité de la couleur correspondante (rouge, vert ou bleu) par un niveau parmi 256.

Passer d'un niveau n au niveau immédiatement supérieur (n+1) ou inférieur (n-1) ne modifie que peu la teinte du pixel, or c'est ce que l'on fait en modifiant le bit de poids faible de l'octet.