01 | Architecture des ordinateurs

I - Les composants et leurs connexions

I.1 Ordinateur

Un ordinateur est un système de traitement numérique de l'information, capable d'exécuter des programmes, c'est-à-dire une suite d'opérations enregistrées en mémoire.

I.2 Architecture interne

L'architecture de l'ordinateur se décompose en 3 parties :

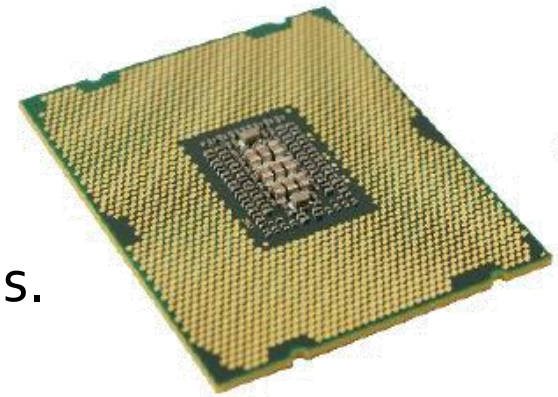
- **Le processeur**
- **La mémoire**
- **Les bus**

I - Les composants et leurs connexions

I.2 Architecture interne

Le processeur (ou unité centrale)

Composé de 2 éléments distincts : l'unité arithmétique et logique (UAL) chargée de réaliser les opérations élémentaires, et l'unité de contrôle (UC) chargée du séquençage de ces opérations.



Un registre (mémoire) est présent pour les calculs élémentaires.

Caractéristique : **fréquence d'horloge** (ou cycle) exprimée en Hertz correspondant aux nombres d'impulsions qu'il peut effectuer par seconde.

I - Les composants et leurs connexions

I.2 Architecture interne

La mémoire

La mémoire est utilisée pour le stockage des données. Il existe 4 sortes de mémoire :

- la **mémoire morte** (ROM for Read-Only Memory) est destinée à une lecture seule et contient entre autre le BIOS (Basic Input Output System)

I - Les composants et leurs connexions

I.2 Architecture interne

La mémoire

La mémoire est utilisée pour le stockage des données. Il existe 4 sortes de mémoire :

- la **mémoire morte** (ROM for Read-Only Memory) est destinée à une lecture seule et contient entre autre le BIOS (Basic Input Output System)
- la **mémoire vive** (RAM for Random Access Memory) est destinée à la mise en mémoire des données temporaires et elle est composée essentiellement de condensateurs

I - Les composants et leurs connexions

I.2 Architecture interne

La mémoire

La mémoire est utilisée pour le stockage des données. Il existe 4 sortes de mémoire :

- la **mémoire morte** (ROM for Read-Only Memory) est destinée à une lecture seule et contient entre autre le BIOS (Basic Input Output System)
- la **mémoire vive** (RAM for Random Access Memory) est destinée à la mise en mémoire des données temporaires et elle est composée essentiellement de condensateurs
- les **mémoires de masse et mémoires flash** (disques durs, clef USB, CD, DVD, …) sont destinées à conserver les données permanentes (-> temps d'accès très grand)

I - Les composants et leurs connexions

I.2 Architecture interne

Les bus

Ce sont des **câbles physiques** chargés de l'échange d'informations entre la mémoire et l'unité centrale. Ils transmettent les impulsions électriques (ou bit).

I.3 Composants et communication binaire

Les principaux composants sont : des fils de connexion, des transistors, des condensateurs ou encore des amplificateurs opérationnels.

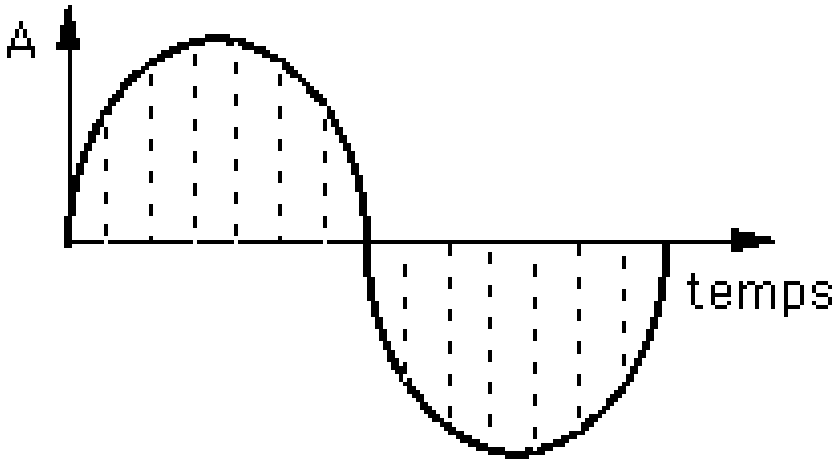
Toute communication effectuée est en réalité de nature **électrique** pouvant être traduit en **binaire** :

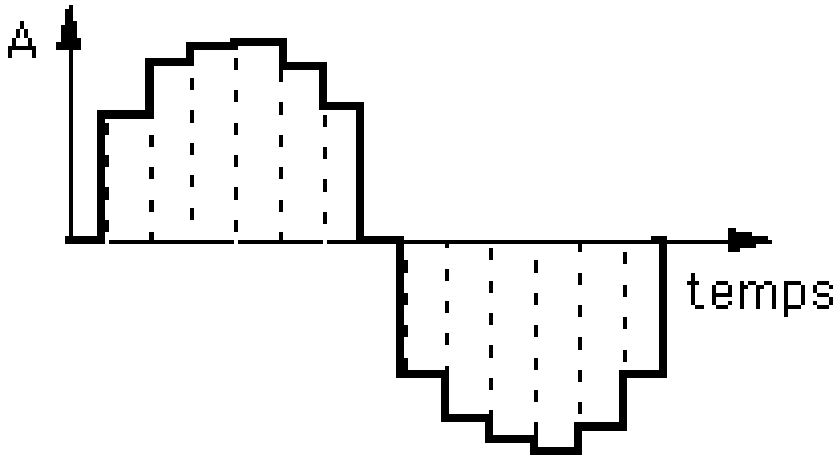
- absence de signal électrique (0)
- présence de signal électrique (1)

II - Le codage des données en mémoire

II.1 Du matériel à l'information

L'information doit être codée sous une forme conventionnelle et doit s'adapter à la structure matérielle de l'ordinateur.
Ainsi, la **représentation binaire** de l'information est dite **représentation numérique** et s'oppose à la **représentation analogique** utilisant des évolutions continues de grandeurs physiques.





Exemple : La TNT (Télévision Numérique Terrestre)

II - Le codage des données en mémoire

II.1 Du matériel à l'information

Une information élémentaire binaire est appelée en informatique un **bit**, diminutif de Binary digIT.

Un **octet** est un mot binaire de 8 bits (donc 2^8=256 mots possibles).

Exemple : 10011101

Système international			Système binaire (pour les octets)	
1 kilo-octet	10^3 octets		1 kibi-octet (kio)	2^{10} octets
1 méga-octet	10^6 octets		*1* mébi-octet (Mio)	2^{20} octets
1 giga-octet	10^9 octets		1 gibi-octet (Gio)	2^{30} octets
1 tera-octet	10^{12} octets		*1* tébi-octet	2^{40} octets

II - Le codage des données en mémoire

II.2 Différents types de données standard

Dans bon nombre de langages de programmation, les variables doivent être déclarées avant de les utiliser (incluant son type), de façon à ce qu'un espace mémoire dans la RAM leur soit alloué.

En Python, les variables n'ont pas à être déclarées !

Nous traiterons ici les quatre types standards suivants :

- **Les booléens (bool)**
- **Les nombres entiers (int)**
- **Les nombres décimaux (float)**
- **Les chaînes de caractères (string)**

II - Le codage des données en mémoire

II.2.1 Les booléens

Une variable booléenne est une variable valant **vrai** ou **faux**, souvent notés **0** ou **1**. Voici nos premières lignes de code :

```
2+2==4      #Le symbole == traduit une égalité (vraie ou fausse). Ce code renvoie True
x = 4<3     #Définit la variable x comme le résultat de 4<3 (donc x vaut False).
type(x)     #renvoie le type de la variable x, donc bool
```

II.2.2 Les nombres entiers naturels

Les entiers sont codés en mémoire en utilisant une base (binaire, décimale, hexadécimale…)

II - Le codage des données en mémoire

II.2.2 Les nombres entiers naturels

Pour tout nombre N entier naturel, il existe une décomposition unique dans une base $b \geq 1$ telle que :

$$N = \sum_{k=0}^{\infty} \alpha_k b^k \text{ avec } 0 \leq \alpha_k \leq b - 1 \text{ tous nuls à partir d'un certain rang } k.$$

$$(392)_{10} = 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

poids croissant

Ou encore en considérant le dernier rang noté n (**méthode de Hörner**) :

$$N = \sum_{k=0}^{n} \alpha_k b^k = \Big(\big(((\alpha_n b + \alpha_{n-1}) b + \alpha_{n-2}) b + \cdots \big) + \alpha_1 \Big) b + \alpha_0$$

II - Le codage des données en mémoire

II.2.2 Les nombres entiers naturels

Base 10	Base 2	Base 16		Base 10	Base 2	Base 16
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

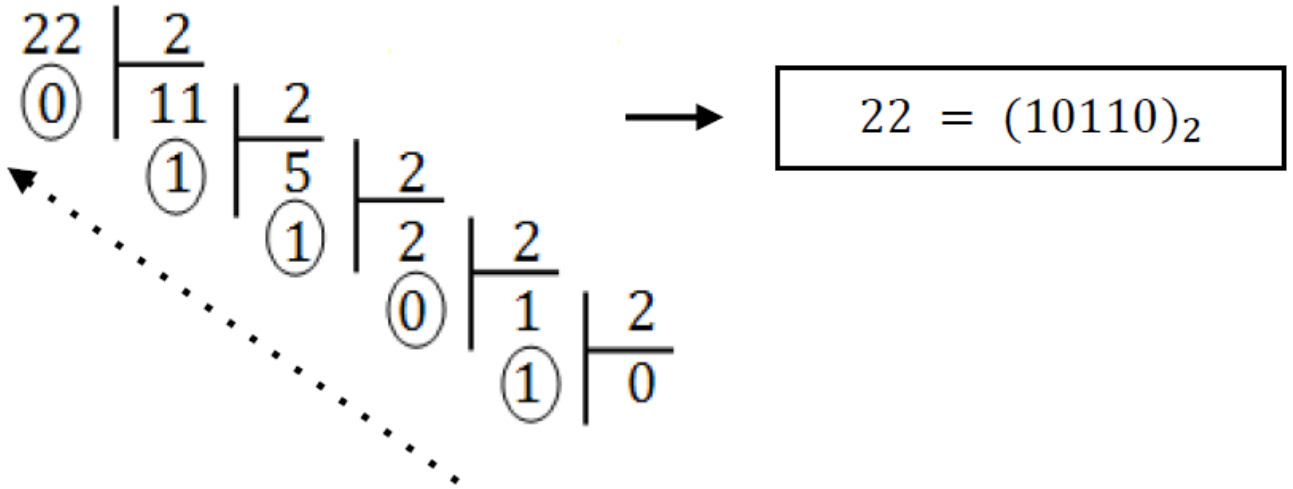
II - Le codage des données en mémoire

II.2.2 Les nombres entiers naturels

Notations Python :

- Aucun préfixe pour les nombres en base 10 : 255
- Préfixe ob pour les nombres binaires : ob11111111
- Préfixe ox pour les nombres hexadécimaux : oxFF
- Préfixe oo pour les nombres en base octale : oo377

Exemple :
Conversion de 22 en base binaire.



$$22 = 2 \times 11 = 2 \times (2 \times 5 + 1) = 2 \times (2 \times (2 \times 2 + 1) + 1) = 2^4 + 2^2 + 2$$

II - Le codage des données en mémoire

II.2.2 Les nombres entiers naturels

Astuces de conversions :

- Binaire -> Hexadécimale

  1011110 -> | 0101 | 1110 | -> | 5 | 14 | -> 5E

- Hexadécimale -> Binaire

  A1 -> | 10 | 1 | -> | 1010 | 0001| -> 10100001

- Idem pour l'octale (avec 3 bits)

II - Le codage des données en mémoire

II.2.2 Les nombres entiers naturels

Astuces de conversions :

- Binaire -> Hexadécimale

  1011110 -> | 0101 | 1110 | -> | 5 | 14 | -> 5E

- Hexadécimale -> Binaire

  A1 -> | 10 | 1 | -> | 1010 | 0001| -> 10100001

- Idem pour l'octale (avec 3 bits)

II - Le codage des données en mémoire

II.2.2 Les nombres entiers relatifs

Situation : On dispose de n bits et on souhaite donner la représentation de entiers allant de -2^{n-1} à $2^{n-1}-1$

Codage par complément à deux. On inverse tous les bits (complément à un), puis on ajoute 1.

Codage d'un nombre négatif par complément à deux : Soit x entre -2^{n-1} et 0. On le code en binaire par le complément à deux de $|x|$.

		entier négatif
1 0 0	-4	
1 0 1	-3	
1 1 0	-2	
1 1 1	-1	
0 0 0	0	0
0 0 1	1	1
0 1 0	2	2
0 1 1	3	3
1 0 0		
1 0 1		
1 1 0		
1 1 1		

entier positif

II - Le codage des données en mémoire

II.2.2 Les nombres entiers relatifs

Représentation des entiers relatifs négatifs, notation en complément (illustré avec l'exemple -5, sur 4 bits) :
Méthode 1 (par décalage) :

- On cherche la représentation en binaire de $-5+2^4$ (décalage) soit $-5+16=11$
- $(11)_{10} \rightarrow (1011)_2$

Méthode 2 (par inversion des bits) :

- On cherche la représentation binaire de $|-5|$ c'est-à-dire 5, ce qui donne 0101 (sur 4 bits)
- On inverse tous les bits obtenus, ce qui donne : 1010 (complément à un)
- On ajoute 1 soit finalement $(1011)_2$

II - Le codage des données en mémoire

II.2.3 Les nombres décimaux

Il existe deux conventions standard pour coder les nombres décimaux :

- la convention à virgule fixe (usuellement obsolète)
- La convention à **virgule flottante**

Un nombre à virgule flottante est exprimé sous la forme $\boldsymbol{sm2^E}$ où s est le signe du nombre (0: positif ou 1 : négatif), E son exposant (entier relatif tenant compte du décalage) et m sa mantisse (nombre à virgule compris entre 1 inclus et 2 exclu). Un flottant peut être codé dans un format simple, double, voire quadruple précision.

Représentation sous la forme :

Bit de signe	Bits d'exposant	Bits de mantisse (dont 1 caché)

II - Le codage des données en mémoire

II.2.3 Les nombres décimaux

La valeur du nombre avec une précision 32 bits est donnée par :

$N = (-1)^s \times (1 + \sum_{i=1}^{23} m_i 2^{-i}) \times 2^{e-E_{max}}$ avec $e = e_1 e_2 \dots e_8$ les bits d'exposant

et $E_{max} = 127$ (décalage)

Par exemple : N=*1 10000010 00110000000000000000000*

- s=1 donc N est de signe négatif
- e=2^7+2^1 (somme des bits d'exposant) donc $e-Emax$=130−127=3
- m=1+2^{-3}+2^{-4} (bit caché + somme des bits de mantisse) =1.1875

Finalement, on a :

$$N = (-1)^1 \times 1.1875 \times 2^3 = -9.5$$

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

La représentation des nombres a des conséquences directes sur le calcul numérique, en particulier **l'overflow** (dépassement d'entier) et **la représentation limitée en mémoire**.

- Erreurs d'arrondis :

```
>>> 0.3 == 0.2 + 0.1      # l'instruction == réalise une comparaison de terme de gauche à celui de droite
False

>>> 0.3-0.2-0.1
-2.7755575615628914e-17
```

Ne jamais utiliser de comparaisons == entre des flottants !

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

- Erreurs d'arrondis :

```
>>> 1.2000000000000001 > 1.2
True            #(double précision : 16 chiffres derrière la virgule, les 16 chiffres sont pris en compte)

>>> 1.20000000000000001 > 1.2
False           #(17 chiffres derrière la virgule, le 17ème chiffre est ignoré)
```

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

- Erreurs d'arrondis :

```
>>> 0.3-(0.2+0.1)
-5.551115123125783e-17

>>> (0.3-0.2)-0.1
-2.775557561562891e-17

>>> (2.55*2.06)*1.55 == 2.55*(2.06*1.55)
False
```

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

- Erreurs d'absorption :

Ce problème peut survenir dès que l'on somme 2 flottants $\boldsymbol{a}$ et $\boldsymbol{b}$ ayant des ordres de grandeur très différents.

```
>>> 2**30 + 1 – 2**30
1                          #(en double et simple précision)

>>> 2**30 + 1.0 – 2**30
0.0                        #(en simple précision)

>>> 2**30 + 1.0 – 2**30
1.0                        #(en double précision)



>>> 2**53 + 1.0 – 2**53
0.0                        #(en simple et double précision)
```

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

- Erreurs de cancellation (ou annulation) :

Ce problème peut survenir dès que l'on soustrait 2 flottants $\boldsymbol{a}$ et $\boldsymbol{b}$ ayant des ordres de grandeur proches.

```
def f(x):
        return (x+1)**2 - x**2 - 2*x - 1

>>> f(10000000000)        # On remplace x par 10000000000 dans la fonction f définie
0                         # Avec un entier, le problème ne se pose pas.

>>> f(1E10)
-2049.0                   # Constater l'ampleur des dégâts avec un flottant...
```

II - Le codage des données en mémoire

II.2.4 Problèmes liés à la représentation des nombres

- Propagation des erreurs :

```
>>> import math
>>> math.sqrt(100)
10.0                                (résultat à priori fiable)
```

```
>>> math.sqrt(exp(log(100)))        # « log » désigne  sous Python le logarithme népérien
« ln(x) »
10.000000000000002                  (17 chiffres significatifs)

>>> math.sqrt(exp(log(1000)-log(10)))
9.999999999999998                   (16 chiffres significatifs)
```

II - Le codage des données en mémoire

II.2.5 Les chaînes de caractères

Le codage des caractères en mémoire relève d'une simple association entre un code binaire et un symbole.

Le standard ASCII :

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	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	{	|	}	~	

II - Le codage des données en mémoire

II.2.5 Les chaînes de caractères

Exemple : Ecriture du mot « Informatique » en code hexadécimal

49 6E 66 6F 72 6D 61 74 69 71 75 65

D'autres standards existent pour intégrer plus de caractères :

- Unicode
- UTF-8

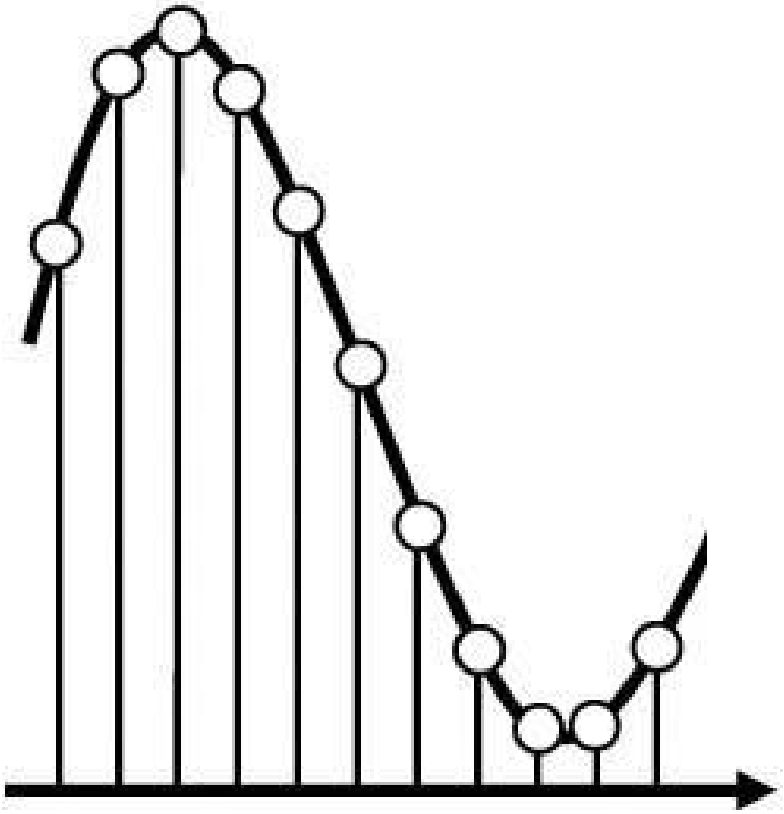
II - Le codage des données en mémoire

II.3 Codage des données complexes

II.3.1 Codage des signaux et courbes

Il est nécessaire de numériser les signaux analogiques, a l'aide de :

- **L'échantillonnage** : évaluation de la courbe sur un nombre fini de points en abscisse
- **La quantification** : approche de la valeur en ordonnée dans un ensemble fini de valeur



II - Le codage des données en mémoire

II.3.2 Codage du son

Le son est donc codé comme un signal, donc comme une liste de points régulièrement répartis au cours de temps.

II.3.3 Codage des images

Les images sont discrétisées spatialement en **pixels**. La couleur de chaque pixel est mémorisée sous la forme de 3 niveaux de couleurs rouge, vert, bleu (RVB).