Représentation numérique de l'information

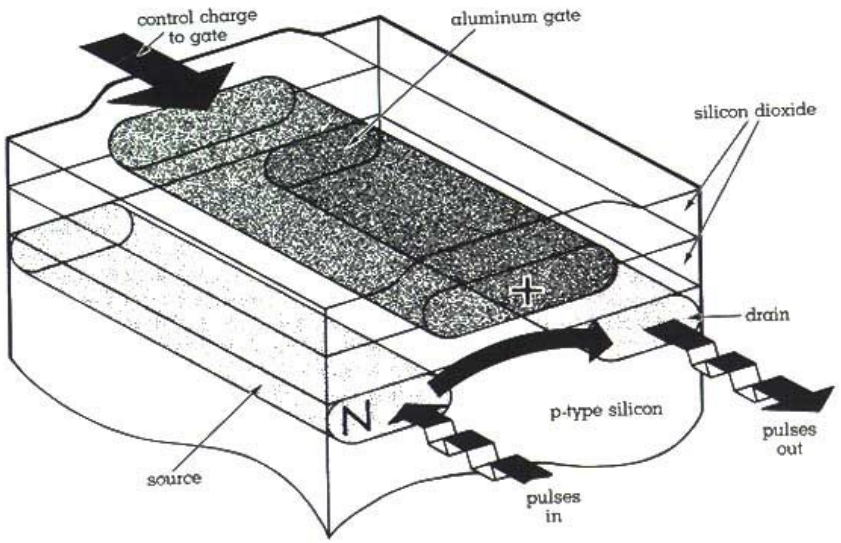
Ce que manipule un ordinateur

- Un ordinateur est assemblage de circuits électroniques.
- Un circuit électronique numérique manipule différentes grandeurs physiques (par exemple tension électrique) pour représenter l'information.
- **Le codage** des informations est lié aux différents états de ces grandeurs physiques.

Le bit

- L'unité élémentaire utilisée en informatique pour coder l'information est appelé **un bit**.
- Le mot « bit » étant la contraction de ***bi****nary* *digi****t*** (chiffre binaire), un bit peut prendre deux valeurs : **0** ou **1**
- Pourquoi le bit est l'unité de base du codage de information dans un système informatique ?
- Les éléments de base manipulant les informations sur les circuits numériques sont les transistors à effet de champs (FET).

Vue simplifiée d'un transistor



- De façon simplifiée, suivant la grandeur de la tension appliquée sur la grille (gate en anglais) du transistor, le courant passe, ou non, de la source au drain. Ces deux états physiques (tensions) possibles représentent le 0 et le 1.

Source :
http://eurserveur.insa-lyon.fr/LesCours/physique/AppPhysique/approphys/5telecom/GSM/memoire_interne.html

Le bit et codage de l'information

- Avec seulement deux états, comment coder des informations plus complexes que 0 et 1 ?
- Simplement en utilisant des séquences de bits.
- Par exemple, si on associe respectivement les états 0 et 1 aux chiffres 0 et 1 du système binaire, on peut coder des informations plus complexes tels que des nombres entiers.

Notation

- Dans la suite nous noterons $(x_n x_{n-1} ... x_i ... x_0)_B$ la représentation en notation positionnelle du nombre $x_n x_{n-1} ... x_i ... x_0$ en base B.
- $(123)_{10}$ soit 123 en base 10, soit encore 123
- $(101)_2$ soit 101 en base 2, soit encore 5 en décimal
- $(21)_3$ soit 21 en base 3, soit encore 7

Conversion décimal vers base B

- Pour convertir $(X)_{10}$ dans la base B, il suffit d'effectuer une suite de divisions entière par cette base :

1) La division de X par B donne un quotient Q_0 et un reste R_0

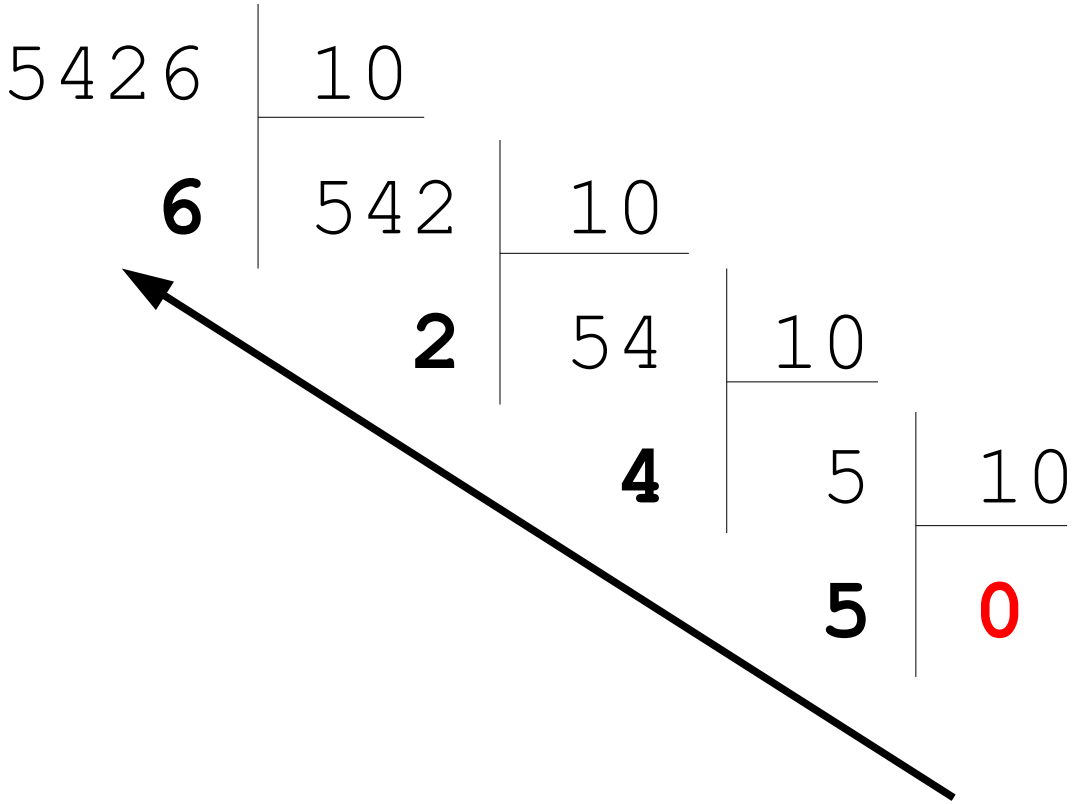
2) Si $Q_0 \neq 0$ alors on divise Q_0 par B pour obtenir Q_1 et R_1

3) On procède ainsi jusqu'à ce que Q_n soit égal à 0.

4) $(X)_{10}$ se code alors $(R_n...R_1R_0)_B$

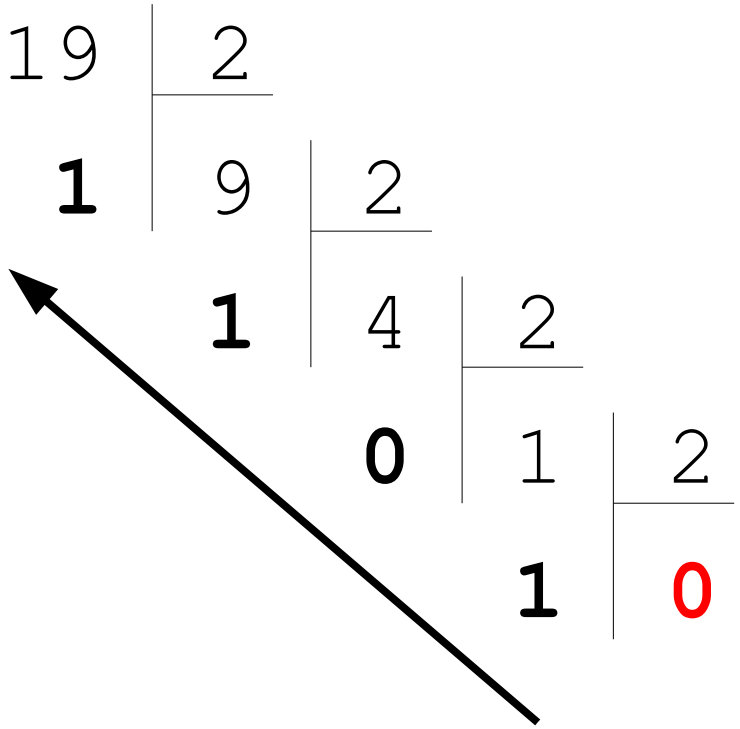
Exemples

- Convertissons $(5426)_{10}$ dans la base 10 pour vérifier que notre méthode fonctionne.



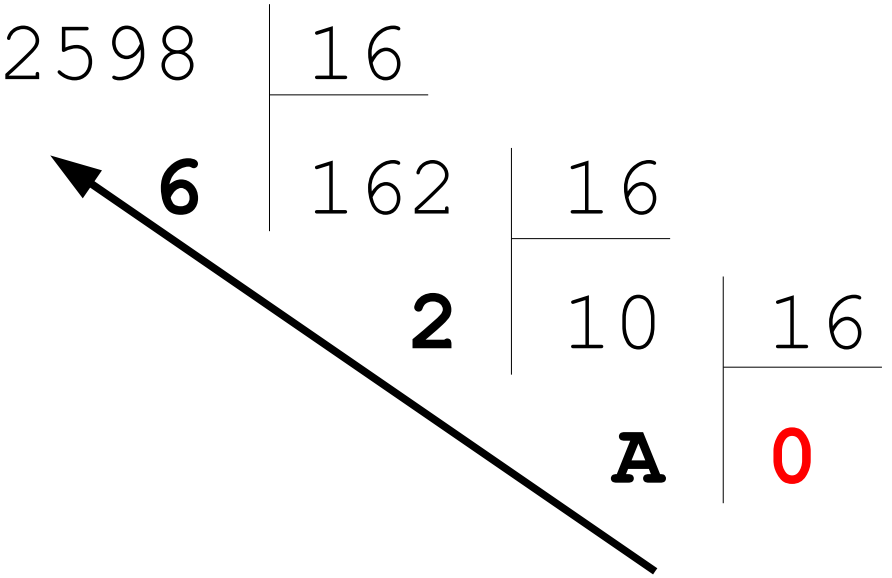
Exemples

- $(19)_{10}$ en base 2



- $(19)_{10}$ se note $(1011)_2$

- $(2598)_{10}$ en base 16



Note : le chiffre A de la base 16 vaut 10 dans le système décimal (voir plus loin)

- $(2598)_{10} = (A26)_{16}$

Le système binaire (1/2)

- La séquence « 00 » représente la valeur binaire $(00)_2$ qui peut s'écrire aussi $(0)_2$ et qui s'écrit $(0)_{10}$ (c'est à dire 0 dans le système décimal)
- La séquence « 01 » représente la valeur binaire $(01)_2$ qui peut s'écrire aussi $(1)_2$ et qui s'écrit $(1)_{10}$
- La séquence « 10 » représente la valeur binaire $(10)_2$ et qui s'écrit $(2)_{10}$
- La séquence « 11 » représente la valeur binaire $(11)_2$ et qui s'écrit $(3)_{10}$

Le système binaire (2/2)

- Ainsi la séquence « 011010 » représente la valeur $(011010)_2$ qui peut s'écrire $(26)_{10}$
  - En effet $(011010)_2$ doit se lire comme $0*2^0+1*2^1+0*2^2+1*2^3+1*2^4+0*2^5$ soit 0+2+0+8+16 soit encore 26 en système décimal.
  - Rappel : pour convertir un nombre Z d'une base B quelconque vers un nombre dans la base 10 (décimal) :

    $Z = (z_n z_{n-1} ... z_i ... z_0)_B$ avec quelque soit i, z_i € $[(0)_{10},(B-1)_{10}]$

    il suffit de sommer tous les z_i*B^i

Un peu de détente ...

- *« Il n'y a que 10 sortes de personnes, celles qui comprennent le binaire et celles qui ne le comprennent pas. »*

- *« Il n'y a que $(10)_2$ sortes de personnes, celles qui comprennent le binaire et celles qui ne le comprennent pas. »*

Représentation binaire

- Pour l'instant, la notation positionnelle de base 2 que nous avons utilisée ne permet que de représenter des entiers naturels
- Pour représenter des entiers relatifs, il faudra utiliser d'autres codages, comme **l'utilisation d'un bit de signe** ou le **complément à deux** (mais nous aborderons ce point plus tard).

Comparaison (1/2)

- Comme nous le verrons, il est parfois utile de comparer des nombres. Ci-après le principe général permettant de dire si un nombre X est plus grand, plus petit ou égal à un nombre Y, X et Y étant représentés tous les deux dans une même base B.
- $X=(x_n...x_i...x_0)_B$ et $Y=(y_n...y_i...y_0)_B$
  - Remarque : ici X et Y sont représentés sur le même nombre de chiffres. Si ce n'est pas le cas, il faut « étendre le nombre ayant le moins de chiffres » en lui rajoutant le chiffre zéro du coté du poids fort (c'est à dire à gauche) autant de fois que nécessaire. Cela n'affecte pas sa valeur.

Comparaison (2/2)

- On parcourt le X et Y de la gauche vers la droite (des poids forts vers les poids faibles) et si l'on trouve un $x_i \neq y_i$ alors
  - si $x_i < y_i$ alors X est inférieur à Y sinon X est supérieur à Y.
- Et si l'on a pas trouvé de $x_i \neq y_i$ alors c'est que les deux nombres X et Y sont égaux.

Comparaison binaire

- Ainsi si l'on veut comparer $(11010)_2$ et $(1111)_2$, il faut étendre le second nombre en $(01111)_2$ et en appliquant le principe précédent, on en déduit que $(11010)_2 > (01111)_2$. Ce qui est exact puisque $(26)_{10} > (15)_{10}$
- Si on compare $(00011010)_2$ et $(00101111)_2$, on voit facilement que $(00011010)_2 < (00101111)_2$,
- Si on compare $(0000\ 0001)_2$ et $(1)_2$ on constatera que les deux nombres sont égaux.

Manipulation des informations

- Historiquement, les ordinateurs ont été tout d'abord utilisés pour effectuer des calculs.
- Ainsi, parmi les fonctionnalités de base du microprocesseur, le cœur de l'ordinateur, les opérations arithmétiques sont fondamentales.
- Étudions l'addition, la soustraction et la multiplication de nombres représentés en binaire (puisque c'est la forme sous laquelle ils sont manipulés par l'ordinateur).

Addition de nombres binaires

- Le principe général est le même que pour l'addition de nombres dans n'importe quelle base B.
- À savoir si on veut additionner deux nombres $X=(x_n x_{n-1}...x_i...x_0)_B$ et $Y=(y_n y_{n-1}...y_i...y_0)_B$ le résultat est $R=(r_{n+1} r_n r_{n-1}...r_i...r_0)_B$ avec
  - si $x_i+y_i+\text{retenue} < B$, $r_i=x_i+y_i+\text{retenue}$ (cas 1)
  - sinon $r_i=x_i+y_i+\text{retenue}-B$ (cas 2)
  - Dans le cas 2, la retenue à additionner au rang i+1 aura la valeur 1 et dans le cas 1, la valeur 0.
  - si i=0 alors la retenue vaut bien évidemment 0.

Addition de nombres binaires

- $(0)_2+(0)_2=(0)_2$

  0+0 faisant bien 0

- $(1)_2+(0)_2=(1)_2$ et $(0)_2+(1)_2=(1)_2$

  1+0 et 0+1 faisant bien 1

- $(1)_2+(1)_2=(10)_2$ il y a eu une retenue de 1.

  1+1 faisant bien 2=$(10)_2$

- $(10)_2+(1)_2=(11)_2$ et $(1)_2+(10)_2=(11)_2$

  2+1 et 1+2 faisant bien 3=$(11)_2$

- $(11)_2+(1)_2=(100)_2$ et $(1)_2+(11)_2=(100)_2$ et $(10)_2+(10)_2=(100)_2$

Soustraction de nombres binaires

- Pour le moment nous ne savons que représenter des nombres positifs et il n'est donc pas possible de faire la soustraction de nombres binaires donnant un résultat négatif.
- Comme nous savons comparer des nombres, nous pouvons déterminer si l'opération de soustraction est possible ou pas.
- De façon générale pour réaliser l'opération de soustraction de Y sur X (c'est à dire X-Y) il faut que $X \geq Y$.

Soustraction de nombres binaires

- À savoir si on veut soustraire deux nombres $X=(x_n x_{n-1}...x_i...x_0)_B$ et $Y=(y_n y_{n-1}...y_i...y_0)_B$ représentés dans une base B et en accord avec ce que nous avons évoqué précédemment, le résultat est $R=(r_n r_{n-1}...r_i...r_0)_B$ avec
  - si $x_i \geq y_i$+retenue, $r_i=x_i-y_i$-retenue (cas 1)
  - sinon $r_i=B+x_i-y_i$-retenue (cas 2)
  - Dans le cas 2, la retenue à soustraire au rang i+1 aura la valeur 1 et dans le cas 1, la valeur 0.
  - si i=0 alors la retenue vaut bien évidemment 0.

Soustraction de nombres binaires

- $(0)_2$-$(0)_2$=$(0)_2$ et $(1)_2$-$(1)_2$=$(0)_2$ et $(10)_2$-$(10)_2$=$(0)_2$

  0-0 et 1-1 et 2-2, etc. faisant bien 0

- $(1)_2$-$(0)_2$=$(1)_2$

  1-0 et 0+1 faisant bien 1

- $(10)_2$-$(1)_2$=$(01)_2$=$(1)_2$ et $(101)_2$-$(100)_2$=$(1)_2$ et $(100)_2$-$(11)_2$=$(1)_2$ (une retenue pour les deux premières opérations et deux pour la seconde)

  2-1 et 5-4 et 4-3 faisant bien 1=$(1)_2$

Multiplication de nombres binaires

- Si on veut additionner deux nombres binaires $X=(x_n x_{n-1}...x_i...x_0)_2$ et $Y=(y_m y_{m-1}...y_j...y_0)_2$ le résultat se calcule de la façon suivante :

- $(x_n*y_0\ x_{n-1}*y_0\ ...\ x_i*y_0\ ...\ x_0*y_0)_2+$
  $(x_n*y_1\ x_{n-1}*y_1\ ...\ x_i*y_1\ ...\ x_0*y_1\ \{0\}^1)_2+...+$
  $(x_n*y_j\ x_{n-1}*y_j\ ...\ x_i*y_j\ ...\ x_0*y_j\ \{0\}^j)_2+...+$
  $(x_n*y_m\ x_{n-1}*y_m\ ...\ x_i*y_m\ ...\ x_0*y_m\ \{0\}^m)_2$

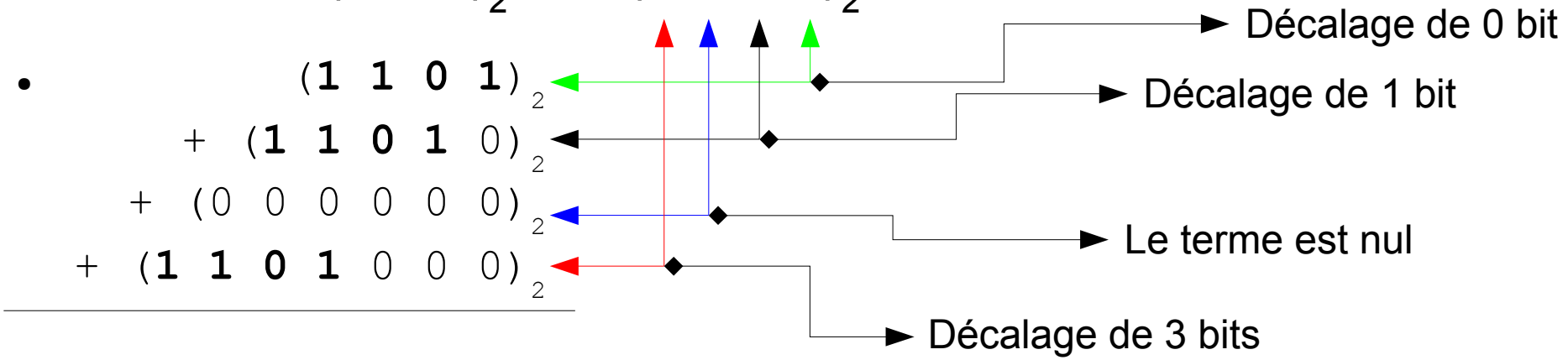
- où $\{0\}^L$ représente L 0
- Il suffit donc de sommer ces différents nombres binaires. Évidemment, les nombres multipliés par les bits de Y valant 0 peuvent être retirés du calcul (puisqu'ils sont nuls).

Multiplication de nombres binaires

- Multiplions $(\mathbf{1101})_2$ par $(1011)_2$.
- $(\mathbf{1}*1\ \mathbf{1}*1\ \mathbf{0}*1\ \mathbf{1}*1)_2 +$
  $(\mathbf{1}*1\ \mathbf{1}*1\ \mathbf{0}*1\ \mathbf{1}*1\ 0)_2 +$
  $(\mathbf{1}*0\ \mathbf{1}*0\ \mathbf{0}*0\ \mathbf{1}*0\ 0\ 0)_2 +$
  $(\mathbf{1}*1\ \mathbf{1}*1\ \mathbf{0}*1\ \mathbf{1}*1\ 0\ 0\ 0)_2$
- $(\mathbf{1\ 1\ 0\ 1})_2 +$
  $(\mathbf{1\ 1\ 0\ 1}\ 0)_2 +$
  $(0\ 0\ 0\ 0\ 0\ 0)_2 +$
  $(\mathbf{1\ 1\ 0\ 1}\ 0\ 0\ 0)_2$
- Soit un total de $(10001111)_2$ soit 128+8+4+2+1, $(143)_{10}$. Et effet, $(1101)_2$ par $(1011)_2$, soit $(13)_{10}$ par $(11)_{10}$ donne $(143)_{10}$.

Multiplication de nombres binaires

- Pour résumer, la multiplication de deux nombres binaires consiste à décaler le multiplicande vers la gauche d'autant de bits que la position du bit à multiplier dans le multiplicateur à chaque bit que ce bit est à 1. Si le bit est à 0, le terme sera nul et il est inutile de le prendre en compte.
- Illustrons $(\mathbf{1101})_2$ par $(1\ 0\ 1\ 1)_2$.
-



Quelques définitions importantes

- Mot binaire (parfois appelé Mot) est une quantité déterminée de bits traitée comme entité unique par l'ordinateur.
  - De façon concrète, le mot est le nombre de bits qu'un microprocesseur peut manipuler en même temps (et donc qui circulent sur les bus).
  - Aujourd'hui la plupart des microprocesseurs utilisent des mots de 32 ou 64 bits (suivant que leur architecture est 32 ou 64 bits). À fréquence équivalente, dans un même cycle de temps, un microprocesseur utilisant des mots de 64 bits manipule deux fois plus d'information qu'un microprocesseur 32 bits.

Quelques définitions importantes

- Une machine 64 bits pourra donc effectuer des opérations complexes plus rapidement qu'une machine 32 bits (*à fréquence de fonctionnement équivalente*)
- Les premiers ordinateurs utilisaient des mots de 8 bits.
- **Un groupe de 8 bits** s'appelle **un octet** (byte en anglais – à ne pas confondre avec bit).
- **Un groupe de 4 bits** s'appelle **un quartet** (nibble en anglais).

Octet

- Un octet étant un groupe de 8 bits, il peut représenter 256 (2^8) valeurs différentes.
- Les valeurs vont de $(0)_{10}$ à $(255)_{10}$
  - $(0)_{10}=(0000\ 0000)_2$
  - $(52)_{10}=(0011\ 0100)_2$
  - $(127)_{10}=(0111\ 1111)_2$
  - $(137)_{10}=(1000\ 1001)_2$
  - $(253)_{10}=(1111\ 1101)_2$
  - $(255)_{10}=(1111\ 1111)_2$

Pour des raisons de lisibilité, nous présenterons souvent les 8 bits par 2 quartets.

Les unités informatiques

Nom	**Nombre de bits**
Octet	8
doublet	16
quadlet	32
octlet	64

Multiples d'octets tels que définis par IEC 60027-2						
Préfixe Système International				***Préfixe binaire***		
Nom	**Symbole**	**Valeur**		**Nom**	**Symbole**	**Valeur**
kilooctet	ko	10^3		kibioctet	kio	2^{10}
mégaoctet	Mo	10^6		mébioctet	Mio	2^{20}
gigaoctet	Go	10^9		gibioctet	Gio	2^{30}
téraoctet	To	10^{12}		tébioctet	Tio	2^{40}
pétaoctet	Po	10^{15}		pébioctet	Pio	2^{50}
exaoctet	Eo	10^{18}		exbioctet	Eio	2^{60}
zettaoctet	Zo	10^{21}		zébioctet	Zio	2^{70}
yottaoctet	Yo	10^{24}		yobioctet	Yio	2^{80}

Opérations arithmétiques binaires simples

- Parfois, il est nécessaire d'appliquer des opérations binaires simples (par exemple appliquer un masque de sous réseau à l'adresse IP du destinataire pour savoir si la machine destination est dans le sous réseau)
- Les opérateurs simples les plus courants sont l'opérateur unaire NON (qui consiste à inverser tous les bits du nombre passé en argument) et les opérateurs binaires (binaire au sens où ils prennent deux arguments) OU, ET et XOR qui consiste à appliquer bits à bits les tables de vérité des opérateurs ad hoc.

Opérations arithmétiques binaires simples

- Tables de vérité

ET	0	1
0	0	0
1	0	1

OU	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NON	0	1
	1	0

- Quelques propriétés simples

NON (A ET B) = (NON A) OU (NON B)

NON (A OU B) = (NON A) ET (NON B)

NON (NON A) = A

A ET $(111...111)_2$ = A

A ET $(000...000)_2$ = $(000...000)_2$

A OU $(111...111)_2$ = $(111...111)_2$

A OU $(000...000)_2$ = A

Opérations arithmétiques binaires simples

- Soit $X=(x_n...x_i...x_0)_2$ et $Y=(y_n...y_i...y_0)_2$
  - NON X = $(non(x_n)...non(x_i)...non(x_0))_2$
  - NON $(0011\ 0100)_2 = (1100\ 1011)_2$
  - X OU Y = $(x_n$ ou y_n ... x_i ou y_i ... x_0 ou $y_0)_2$
  - $(0011\ 0100)_2$ OU $(1000\ 0101)_2 = (1011\ 0101)_2$
  - X ET Y = $(x_n$ et y_n ... x_i et y_i ... x_0 et $y_0)_2$
  - $(0011\ 0100)_2$ ET $(1000\ 0101)_2 = (0000\ 0100)_2$

Application au réseau

- Soit une machine S possédant la configuration réseau :
  - Adresse IP : 164.81.1.12
  - Masque de sous réseau : 255.255.0.0
  - Passerelle (adresse IP du « routeur » à qui la machine enverra les données si le destinataire avec qui elle veut communiquer n'est pas présent sur son sous réseau) : 164.81.1.254
- Définir l'adresse du sous réseau sur lequel se trouve S sachant qu'il suffit d'appliquer un ET binaire entre l'adresse de la machine et le masque pour trouver cette adresse de sous réseau
  - 164.81.1.12 ET 255.255.0.0 = **164.81.0.0**

Les entiers relatifs

- Pour pouvoir représenter des entiers relatifs, on peut utiliser un bit de signe.
- Ce codage utilise le bit de poids fort (bit le plus à gauche) pour représenter le signe (la valeur 0 étant utilisée pour un nombre positif et la valeur 1 pour un nombre négatif) et le reste des bits pour représenter la valeur absolu de l'entier.
- Ainsi sur 8 bits (1 octet), on peut coder $2*2^7-1$ valeurs (de -127 à +127)

Les entiers relatifs

- Quelques entiers
  - $(\mathbf{0}111\ 1111)_2 = (\mathbf{+}127)_{10}$
  - $(\mathbf{1}111\ 1111)_2 = (\mathbf{-}127)_{10}$
  - $(\mathbf{0}000\ 0001)_2 = (\mathbf{+}1)_{10}$
  - $(\mathbf{1}000\ 0001)_2 = (\mathbf{-}1)_{10}$
  - $(\mathbf{0}000\ 0000)_2 = (\mathbf{+}0)_{10} = (0)_{10}$
  - $(\mathbf{1}000\ 0000)_2 = (\mathbf{-}0)_{10} = (0)_{10}$
- Inconvénient (mineur) : Il y a donc deux façons de coder $(0)_{10}$

Le bit de signe est le bit le plus à gauche (ici en gras).

Les entiers relatifs

- Inconvénient (majeur) : il n'est pas possible d'utiliser l'addition usuelle si un des nombres est négatif
  - $(\mathbf{1}000\ 0001)_2 + (\mathbf{0}000\ 0011)_2 = (\mathbf{1}000\ 0100)_2$
  - ce qui signifierait que $(\mathbf{-}1)_{10} + (3)_{10} = (\mathbf{-}4)_{10}$ au lieu de $(2)_{10}$
- Pour remédier à ces inconvénients la notation la plus utilisée pour les entiers relatifs est le **complément à deux**.

Le complément à deux

- Comme précédemment le bit de poids fort est utilisé pour le signe.
- En revanche, cette fois, le bit de signe est pondéré par -2^n si on travaille sur une représentation sur *n* bits.
- Les entiers codables sur *n* bits seront dans l'intervalle $[-2^{n-1}, 2^{n-1}-1]$.
- Les nombres positifs sont représentés comme précédemment sur *n-1* bits.
- Afin de bien faire la différence avec un nombre dans une représentation binaire classique nous indicerons par A2 les nombres binaires en complément à deux.

Le complément à deux

- Les nombres négatifs sont obtenus de la manière suivante :
  - On inverse les bits de l'écriture binaire de sa valeur absolue (opération binaire NON : opération aussi appelée **complément à un**)
  - On ajoute $(1)_2$ au résultat (les dépassements sont ignorés – c'est à dire que si une retenue apparaît lors de la somme sur le bit de poids fort, on ne la propage pas ; autrement dit on reste sur le nombre de bits initial).
    - Cette opération correspond au calcul de $2^n - |x|$, où n est la longueur de la représentation et $|x|$ la valeur absolue du nombre à coder.
    - Ainsi pour des nombres sur 8 bits $(-1)_{10}$ s'écrit comme $(256)_{10}-(1)_{10}=(255)_{10}=(1111\ 1111)_{A2}$

Le complément à deux

- Illustrons sur un octet
  - Codons $(-1)_{10}$
    - La valeur absolue en binaire se représente ainsi : $(0000\ 0001)_2$
    - Faisons son complément à un : $(1111\ 1110)_2$
    - Ajoutons $(-1)_{10} = (0000\ 0001)_2$ c'est à dire réalisons l'addition de $(1111\ 1110)_2$ et $(0000\ 0001)_2$

      Nous obtenons bien $(1111\ 1111)_{A2}$

Le complément à deux

- $(0)_{10}=(0000\ 0000)_{A2}$
- $(-2)_{10}=(1111\ 1110)_{A2}$
- $(-128)_{10}=(1000\ 0000)_{A2}$
- $(127)_{10}=(0111\ 1111)_{A2}$
- $(1)_{10}=(0000\ 0001)_{A2}$
- $(2)_{10}=(0000\ 0010)_{A2}$

Le complément à deux

- En complément à deux, les deux inconvénients mentionnés précédemment n'existent plus.
  - $(-0)_{10}=(+0)_{10}=(0)_{10}=(0000\ 0000)_{A2}$
  - $(1111\ 1111)_{A2} + (0000\ 0011)_{A2} = (0000\ 0010)_{A2}$
    ce qui signifie que $(-1)_{10} + (3)_{10} = (2)_{10}$ et non $(-4)_{10}$ comme précédemment.
    - Le dépassement de capacité (la retenue qui se propage au delà du bit de poids fort) ne doit pas être pris en compte.
- Le résultat de l'addition usuelle de nombres représentés en complément à deux est le codage en complément à deux du résultat de l'addition des nombres.

Le complément à deux

- Et la soustraction ?
  - C'est simplement l'addition de l'opposé du nombre à soustraire !
  - Grâce à ce codage ingénieux, un même circuit peut réaliser les deux opérations.
  - Attention, toutefois à un effet de bord du complément à deux. Lorsque l'addition ou la soustraction de 2 entiers de même signe (codés en complément à deux) donne un résultat de signe contraire, alors celui-ci est erroné. Il s'agit d'un dépassement de capacité (le nombre ne peut pas être codé sur le nombre de bits donnés).

Le système hexadécimal

- Système de numération positionnelle en base 16
- Il utilise donc 16 symboles : les chiffres arabes pour les dix premiers chiffres (0 à 9) et les lettres A à F pour les six suivants
- Il est pratique à utiliser en informatique.
  - Chaque chiffre hexadécimal correspond à 4 bits (4 chiffres binaires – c'est à dire un quartet).
    - La manipulation est donc plus facile
    - La conversion est simple
    - L'écriture est plus compacte

Le système hexadécimal

- Dans cette représentation, un octet se présentera sous la forme de 2 chiffres hexa (il est fréquent d'abréger).
- Ainsi, $(1000\ 1010)_2$ soit $(138)_{10}$ se notera $(8A)_{16}$ ou parfois aussi 0x8A.
  Le **0x** indiquant qu'il s'agit d'un nombre en notation hexadécimale.
- $0x8A=(8*16^1+10*16^0)_{10}$
  $= (138)_{10}$

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

Les réels

- Il est fréquent qu'une application ait besoin de manipuler des données numériques autres que entières.
- Il a donc fallu trouver un codage pour les nombres réels.
- Il y en a plusieurs et nous en aborderons deux :
  - Le codage partie entière et partie fractionnaire
  - Le codage en virgule flottante

Codons les réels
Parties entière et fractionnaire

- Il suffit d'ajouter une partie fractionnaire après la virgule
- Soit $(x_n...x_i...x_0,y_1...y_j...y_m)_B$ dont la valeur décimal sera $x_n*B^n + ... + x_i*B^i + ... + x_0*B^0 + y_1*B^{-1} + ... + y_j*B^{-j} + ... + y_m*B^{-m}$
- $(101,011)_2 = 5,375$
- $(E2,5A7)_{16} = 14*16^1+2*16^0+5*16^{-1}+10*16^{-2}+7*16^{-3}$
  $= 226,353271484375$
- $(67,123)_{10} = 67,123$

Conversion décimale vers base B

- Pour convertir un nombre réel décimal dans une base B, on procède comme suit :
  - On convertit **la partie entière** en utilisant la méthode vue précédemment
  - Pour convertir la partie fractionnaire :
    1) On multiplie la partie fractionnaire initiale F par la base B pour obtenir les parties entière E_1 et fractionnaire F_1
    2) Si $F_1 \neq 0$ alors on garde la partie E_1 et on multiplie par B la partie fractionnaire, F_1, pour obtenir E_2 et F_2
    3) Et on répète ces opérations jusqu'à ce que la partie fractionnaire F_n soit égale à zéro **ou** jusqu'à ce que la précision souhaitée soit atteinte

       On ne peut pas toujours convertir en un nombre de chiffres fini la partie fractionnaire
    4) Le codage de la partie fractionnaire consiste $(E_1 E_2 ... E_n)_B$

Exemples

- Convertissons $(10,8125)_{10}$ en base 2
  - La partie entière $(10)_{10} = (1010)_2$
  - La partie fractionnaire $(0,8125)_{10}$
    - `0,8125 * 2 = 1,625 = 1 + 0,625`
      `0,625 * 2 = 1,250 = 1 + 0,250`
      `0,250 * 2 = 0,5 = 0 + 0,5`
      `0,5 * 2 = 1,0 = 1 + 0`

- $(10,8125)_{10} = (1010,1101)_2$

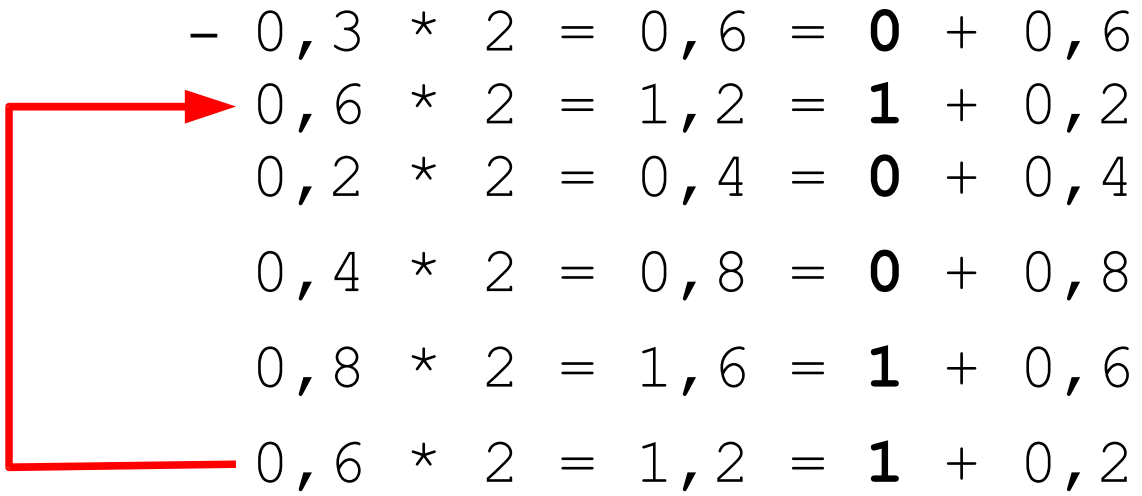
Exemples

- Convertissons $(128,70703125)_{10}$ en base 16
  - La partie entière $(128)_{10} = (80)_{16}$
  - La partie fractionnaire $(0,70703125)_{10}$
    - `0,70703125 * 16 = 11,3125 = 11 + 0,3125`
      `0,3125 * 16 = 5,0 = 5 + 0`

- $(128,70703125)_{10} = (80,B5)_{2}$

Réels : les erreurs de représentation

- Essayons de coder $(0,3)_{10}$ en base 2



  - On voit que le codage est infini
    - $(0,3)_{10}=(0,01001[1001])_2$ avec [1001] se répétant à l'infini

Codons les réels
Virgule flottante

- Comme son nom l'indique, le principe de ce codage consiste à utiliser une virgule flottante.
- La précision dans ce codage sera limitée mais suffisante car il ne prendra en compte que les chiffres significatifs.
- Les nombres seront représentés dans la forme normalisée suivante :
  - +/- 0,M * B^E
    +/- : codage du signe
    M : La mantisse (aussi appelée significande) sera un nombre de x chiffres dans la base B
    E : L'exposant sera un nombre de y chiffres dans la base B

Exemples

- $(4562,05)_{10}$ se normalise en virgule flottante en base 10 par $0,456205*10^{4}$
  - Mantisse : 456205
  - Exposant : 4
  - Signe : +
- $(0,00145)_{10}$ se normalise en virgule flottante en base 10 par $0,145*10^{-2}$
  - Mantisse : 145
  - Exposant : -2
  - Signe : +

Exemples

- $(11,01)_2=(3,25)_{10}$ se normalise en virgule flottante en base 2 par
  - Mantisse : **1101**
  - Exposant : **2**
  - Signe : +
  - En effet $(0,\mathbf{1101})_2=(0,8125)_{10}$ et $2^{\mathbf{2}} = 4$, ce qui donne bien $0,8125*4 = (3,25)_{10}$

Standard IEEE 754

- Spécifie des formats de codage **binaire** de réels en virgule flottante et les opérations associées
- Les deux formats fixés par la norme IEEE 754 sont sur 32 bits (« simple précision ») et 64 bits (« double précision »).
  - La répartition des bits est la suivante, où $1 \leq M < 2$. En effet, puisqu'il s'agit d'un codage binaire et que premier bit de la mantisse d'un nombre normalisé étant toujours 1, il n'est représenté dans aucun des deux formats : on parle de bit implicite. Pour ces deux formats, les précisions sont donc respectivement de 24 et de 53 bits.

Standard IEEE 754

- On note que dans ce codage l'exposant est toujours positif

	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$	24 bits	7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$	53 bits	16

- La norme IEEE 754 prévoit aussi des tailles minimales pour ces types étendus. Dans la pratique seul le type « double précision étendue » sur 80 bits est utilisé.

Dans ces représentations, on notera l'introduction d'un biais de $2^{n-1}-1$ au niveau de l'exposant de taille n bits.

	Signe	Exposant	Mantisse
Simple précision étendue	1 bit	11 bits ou plus	32 bits ou plus
Double précision étendue	1 bit	15 bits ou plus	64 bits ou plus

Représenter des caractères

- L'information manipulée en informatique n'est pas uniquement « numérique » (au sens premier du terme) mais elle est parfois textuelle.
- Ainsi, il a fallu inventer un codage pour représenter les caractères à l'aide de bits.
- Le codage le plus connu et le plus utilisé est l'ASCII (American Standard Code for Information Interchange « Code américain normalisé pour l'échange d'information »).
  - ASCII contient les caractères nécessaires pour écrire en anglais.

Table ASCII basique

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

ASCII étendu

- Comme vous le constatez sur la table précédente, les caractères sont codés sur 7 bits (ou 8 mais avec un bit de poids fort toujours à zéro)
  - Ce bit a été utilisé comme bit de parité pour détecter les erreurs lors de transmissions numériques (voir le cours sur les codes détecteurs et correcteurs d'erreurs)
- Vous constatez la présence de différents caractères « étranges » qui servent au contrôle dans un texte (exemple : retour à la ligne, fin de fichier)

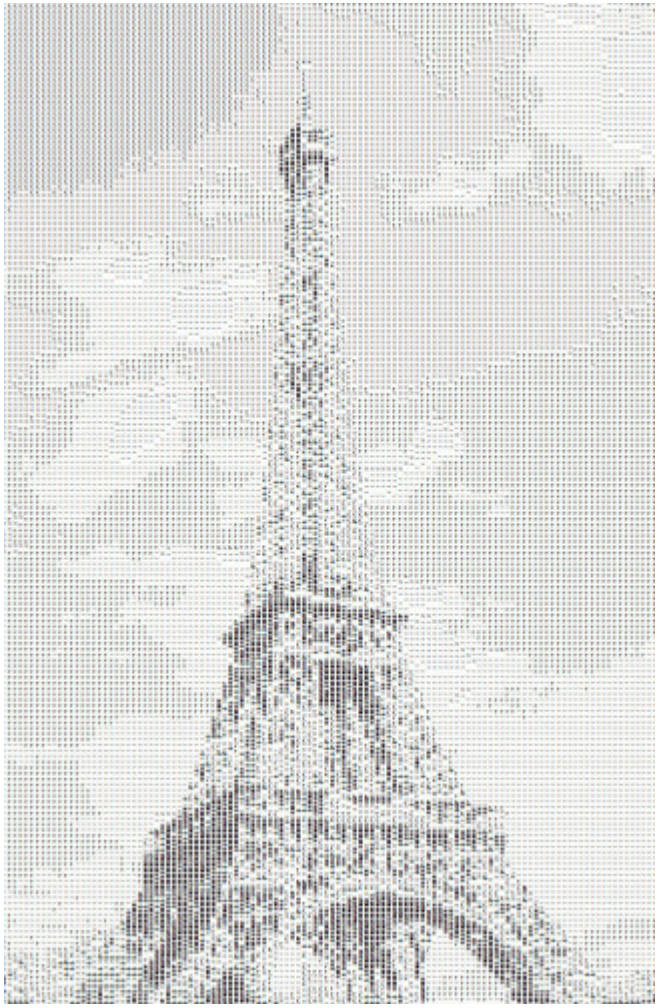
ASCII étendu

- Vous constatez aussi surement que les caractères accentués nécessaires au français par exemple ne sont pas présents. Ainsi un version étendue a été normalisée dans sa dernière version en 1986.
- Dans cette version les 8 bits sont totalement utilisés pour coder 256 caractères différents.
- D'autres codages pour les caractères existent : ISO/CEI 8859-1 (Latin 1), Unicode (UTF-8, UTF-16, ...), etc.

ASCII étendu

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

L'art ASCII



Chaine de caractères

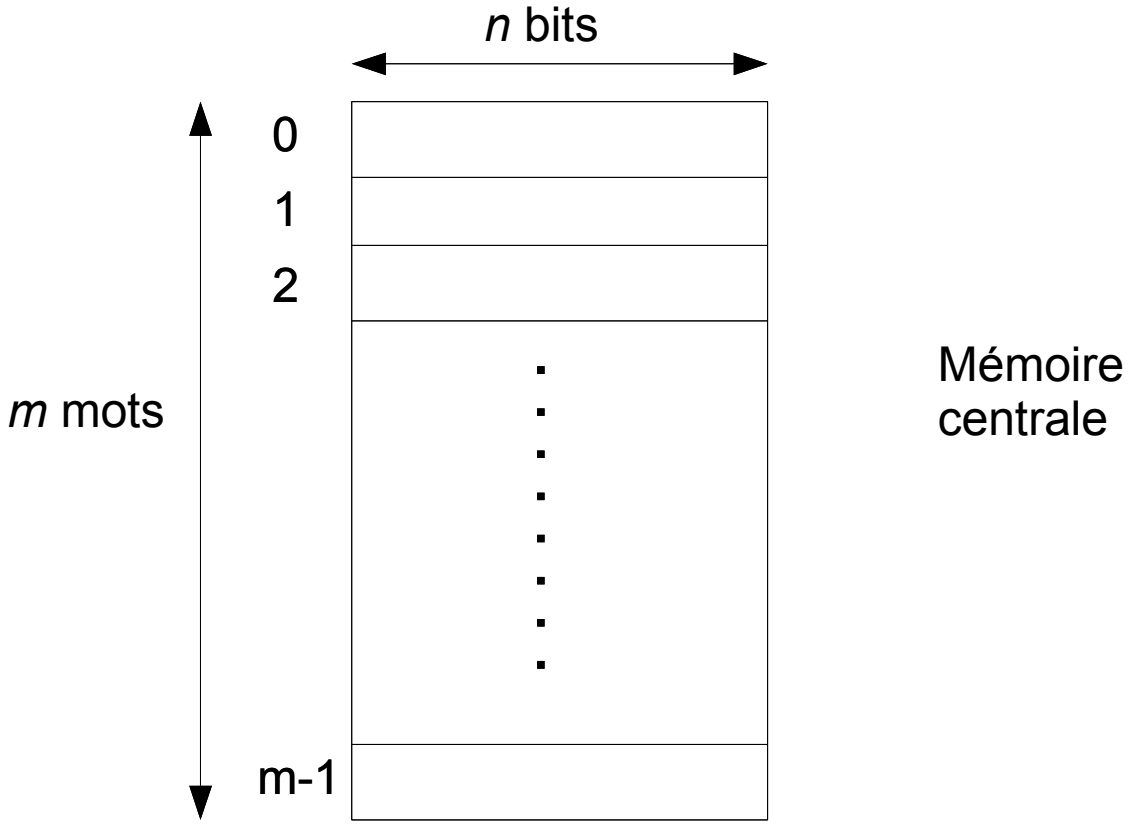
- Il est rare de n'avoir qu'à coder un simple caractère.
- L'information textuelle se rencontre le plus souvent sous la forme d'une chaine de caractères.
- Une chaine de caractère consiste comme son nom l'indique en une suite de caractères qui se termine dans la mémoire par un caractère spécial appelé Null et codé 0x00 dans la table ASCII.

Booléen

- Même si un booléen ne peut prendre que 2 valeurs (0 ou 1, respectivement VRAI ou FAUX), ces données sont en général codées sur au moins 8 bits.
- En effet le degré de granularité de la mémoire est souvent de 8 bits (1 octet) suivant l'architecture mémoire (voir les diapos ci-après) .

Mot mémoire

- Un mot mémoire est une unité d'information (un mot binaire – une séquence de bits) adressable en mémoire. Suivant l'architecture de la mémoire, le mot pourra être de 8, 16, 32 ou 64 bits.



Mot mémoire et adresses (Théorie)

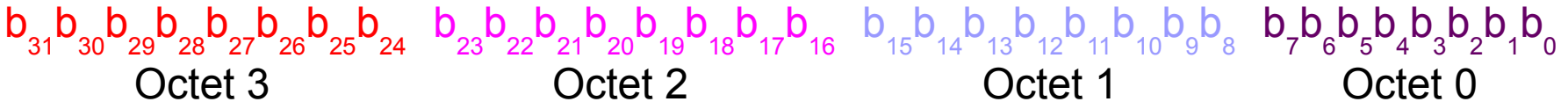
- Ainsi pour une mémoire de 4Go et des mots de 32 bits, il y a 10^9 adresses mémoires.
- Chaque mot (emplacement mémoire) est désigné par une **adresse**.
- Heureusement avec l'exemple ci-dessus, si la technologie utilisée est 32 bits, il est possible d'adresser 2^{32} valeurs, donc de désigner 2^{32} emplacements mémoire. Or avec 4 Go nous avons $10^9 < 2^{32}$
  - Il est en revanche impossible d'adresser plus de 2^{34} octets ≈ 17 Go (2^{32} emplacements * 32 bits pour la taille des emplacements / 8 bits pour avoir la taille en octets).

Mot mémoire / adressage (Pratique)

- Attention ! Certains lecteurs s'interrogeront peut être sur les données précédentes. En effet, en général, on ne considère qu'on ne peut adresser que 4 Go sur une architecture 32 bits !
- Mais pourquoi est ce que nous trouvons un peu plus de 4 fois plus ?
  - Nous avons considéré une architecture mémoire matérielle appelée « **adressable par mot** » (*word-addressable*) et non une architecture mémoire « **adressable par octet** » (*byte-addressable*). C'est pourtant ce dernier cas qui est le plus fréquent dans la vie courante
  - Ainsi $2^{32} \approx$ 4Go (les arrondis expliquent le facteur « *un peu plus de 4 plus* »)

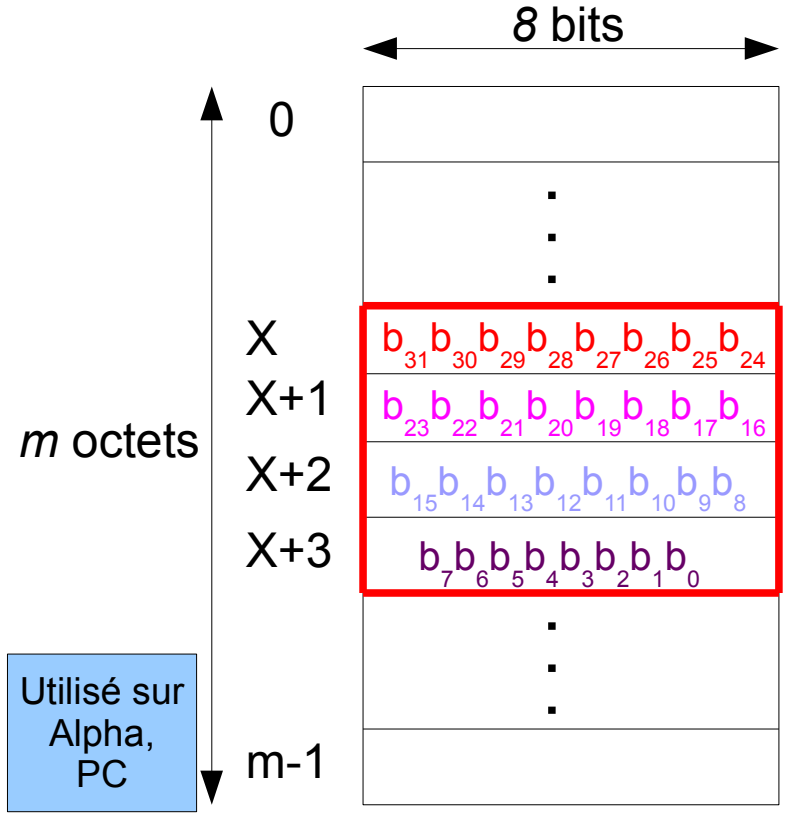
Big endian / Little endian

- Comme nous venons de le voir la quasi-totalité des cas un mot mémoire (l'unité d'adressage) correspond à un octet.
- Certains codages utilisant plus de bits qu'il n'y en a dans un octet, il faut décider comment les organiser dans la mémoire.
- Prenons l'exemple d'un entier devant être codé sur 32 bits (soit 4 octets).

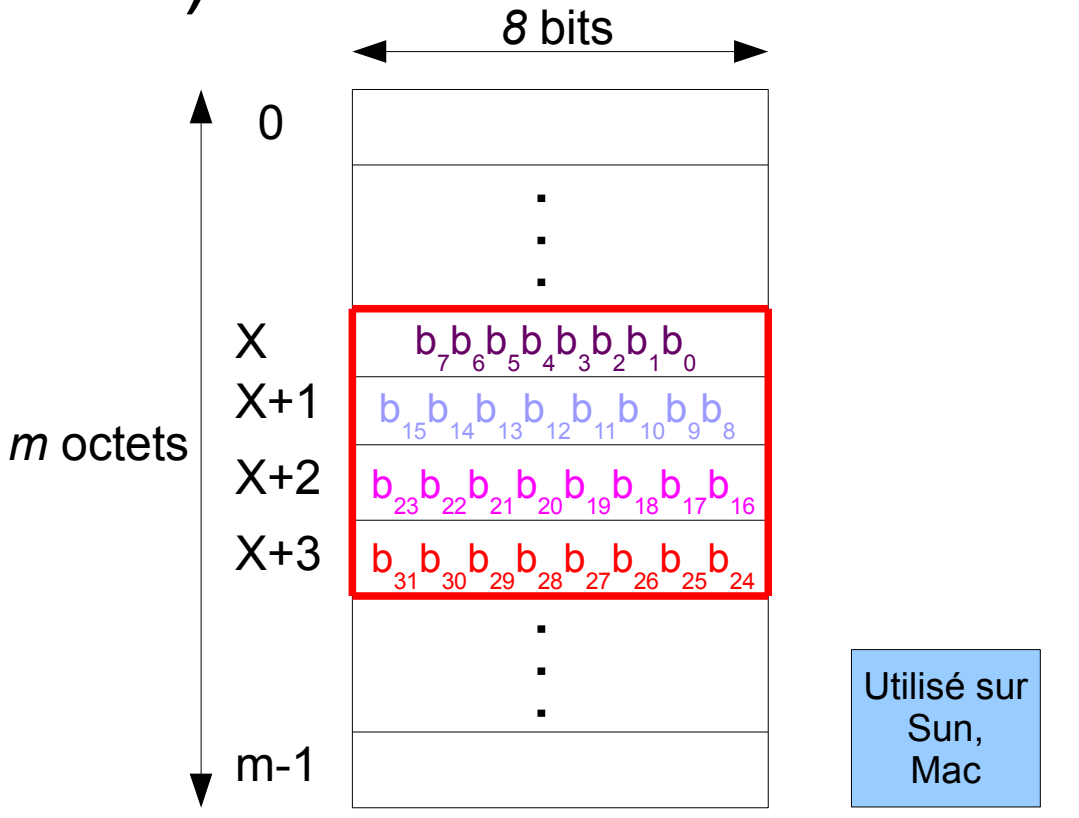


Big endian / Little endian

- Il y a deux conventions pour organiser cet entier en mémoire (elles varient suivant les constructeurs/architectures).



Little endian (petit-boutiste)



Big endian (gros-boutiste)

La notion de type

- Nous venons de voir comment coder de façon numérique des informations de nature différente: numérique et textuel
- Toutefois, dans les codages que nous avons présentés, la plupart n'avaient pas une représentation bien définie et il existait d'ailleurs pour certains catégories d'information différentes façons de les coder .
- Un type consistera en une représentation de données d'une certaine nature et un codage bien défini. Cette notion sera approfondie dans la suite.

Liens

- Cours sur l'arithmétique flottante

  http://hal.inria.fr/inria-00071477
- Codage des nombres, Eric Cariou
- Calculatrice IEEE754 : http://babbage.cs.qc.edu/IEEE-754/