Partie I : Représentation de l'information

1. Introduction

1.1. Représentation de l'information traitée par ordinateur

Les informations traitées par un ordinateur peuvent être de différents types (texte, nombres, images, son, vidéos, etc.) mais elles sont toujours représentées et manipulées par l'ordinateur sous forme numérique (digitale). En fait, toute information sera traitée comme une suite de 0 et de 1. L'unité d'information est donc les chiffres binaires (0 et 1) que l'on appelle bit (pour **b**inary dig**it** : chiffre binaire).

On utilise la représentation binaire car elle est simple, facile à réaliser techniquement à l'aide de bistables (système à deux états réalisés à l'aide de transistors).

Le codage d'une information consiste à établir une correspondance entre la représentation externe (habituelle) de l'information (texte, image, …etc), et sa représentation interne dans la machine, qui est toujours une suite de bits.

1.2. Quantité de l'information traitée

L'unité de base de mesure de la quantité d'information en informatique est donc le **bit** tel qu'1 bit peut prendre la valeur **0** ou **1**.

Q : Combien d'états peut-on représenter avec 3 bits ? avec 4 bits ? et avec *n* bits en général ?

Chaque 8 bits constituent **1 Octet** (**Byte** en anglais) symbolisé par **Ø** (et symbolisé par **B** en anglais).

Aussi :

2^{10} bits = 1024 bits = 1 Kb (1 Kilo bits)	2^{10} Ø = 1024 Ø = 1 KØ (1 Kilo Ø)
2^{10} Kb = 1024 Kb = 1 Mb (1 Méga bits)	2^{10} KØ = 1024 KØ = 1 MØ (1 Méga Ø)
2^{10} Mb = 1024 Mb = 1 Gb (1 Giga bits)	2^{10} MØ = 1024 MØ = 1 GØ (1 Giga Ø)
2^{10} Gb = 1024 Gb = 1 Tb (1 Téra bits)	2^{10} GØ = 1024 GØ = 1 TØ (1 Téra Ø)

Q : Convertir 2GØ en bits puis en Kb?

Il est donc clair que l'information est traitée sous forme binaire dans un ordinateur, que ça soit du texte (association de codes conventionnés à chaque caractère), des images (association de codes à chaque couleur de pixel de l'image), du son (association de codes à chaque fréquence de son), …etc. Il est donc indispensable de voir de plus près la manipulation des données binaires ainsi que leur relation avec d'autres systèmes de numération.

2. Systèmes de numération

Au fil du temps plusieurs systèmes de numération sont apparus. De la mésopotamienne qui était positionnelle (la position du chiffre indique son rang, comme dans la numération arabe qu'on utilise aujourd'hui) à l'égyptienne et la romaine qui était additionnelle (le nombre représenté est égal à la somme des symboles représentés), à celle des chinois qui excellaient dans les calculs (création du boulier) et qui est également positionnelle, …etc.

Ex : numération égyptienne

1	10	100	1 000	10 000	100 000	1 000 000
						

 **= 345**

2.1. Représentation

Un nombre : $(\mathbf{XXX})_{\mathbf{b}}$ indique la représentation d'un nombre **XXX** dans la base ***b***.

Les bases usuelles qu'on connait et utilise tous les jours sont la **base 10** (système décimale) pour représenter les différentes grandeurs et les différents chiffres et nombre (monnaies, n° de tel, tailles, date, …) et la **base 60** (système sexagésimal) pour représenter le temps.

Comment un nombre est représenté dans une base *b* ?

1. Si $b \leq 10$, on utilise simplement les chiffres de 0 à b-1

Ex : *base 8 (système octal) :* n'importe quel nombre sera la combinaison de chiffres appartenant à l'ensemble {0,…, 7}

2. Si $b > 10$, on utilise simplement les chiffres de 0 à 9 ensuite des lettres dans l'ordre alphabétique.

Ex : *base 16 (système hexadécimal)* : n'importe quel nombre sera la combinaison de symboles appartenant à l'ensemble {0,…, 9,A, B, C, D, E, F} tel que : (A=10, ….., F=15)

Donc chaque système de numération utilise un ensemble de symboles (chiffres) pour représenter les différents nombres. Le nombre de ces chiffres est toujours égal à l'ordre de la base elle-même. Autrement dit : la base du système de numération est égale au cardinal de l'ensemble des symboles utilisés dans cette base.

Ex : en *binaire*, base du système binaire = 2 ; ensemble des symboles utilisés : A = {0,1}, Card (A)=2=base du système binaire.

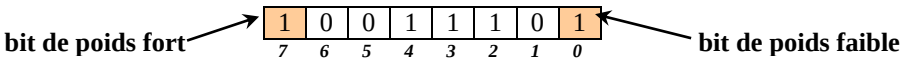
en *octal*, base du système octal = 8 ; ensemble des symboles utilisés : A ={0,1,2,3,4,5,6,7}, Card (A)=8=base du système octal.

Un nombre de n chiffres (symboles) est une suite (a_i), $0 \leq i \leq n-1$:

$$\mathbf{a_{n-1} \ldots . a_1 a_0}$$

tel que : a_0 est le terme de poids faible et a_{n-1} est le terme de poids fort.

Ex : Soit le mot binaire de 8 bits : **10011101**



Les systèmes de numération qui nous intéresse dans le domaine informatique sont : le décimal, le binaire, l'octal et l'hexadécimal.

2.2. le système décimal

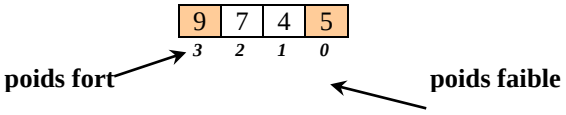
Le système décimal est celui dans lequel nous avons le plus l'habitude d'écrire. Chaque chiffre peut avoir 10 valeurs différentes : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, de ce fait, le système décimal a pour base 10. La valeur de chaque chiffre dépend de sa position, c'est-à-dire que c'est une numération positionnelle : la position la plus à droite exprime les unités, la position suivante : les dizaines, ensuite les centaines,…etc.

Par exemple, si on décompose le nombre 9745, nous aurons :

$9745 = 9 \times 1000 + 7 \times 100 + 4 \times 10 + 5 \times 1$

$9745 = 9 \times 10^3 \times 7 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$

Nous remarquons que chaque chiffre du nombre est à multiplier par une puissance de 10. Cette puissance représente le poids du chiffre.



L'exposant de cette puissance est nul pour le chiffre situé le plus à droite et s'accroît d'une unité pour chaque passage à un chiffre vers la gauche.

☝***Remarque :***
Cette façon d'écrire les nombres est appelée système de numération de position. Elle est valable pour tous les systèmes de numération que nous verrons dans ce cours (décimal, binaire, octal et hexadécimal).

2.3. le système octal

Suivant ce que nous avons cité dans la section **2.1**, le système octal utilise un système de numération ayant comme base 8 (octal : latin octo=huit) et utilise donc 8 symboles : de 0..jusqu'à..7. Ainsi, un nombre exprimé en base 8 pourra se présenter de la manière suivante par exemple : $(745)_8$

📖***Rappel** : Lorsque l'on écrit un nombre, il faudra bien préciser la base dans laquelle on l'exprime pour lever toutes ambiguïtés (745 existe aussi en base 10 par exemple). Ainsi le nombre sera mis entre parenthèses (745 dans notre exemple) et indicé d'un nombre représentant sa base (8 est mis en indice). Par convention, quand on ne précise pas la base, elle est par défaut égale à 10.*

2.4. le système binaire

Comme nous l'avons vu plus haut, dans le système binaire, chaque chiffre ne peut avoir que l'une des 2 valeurs : **0** ou **1**. De ce fait, le système a pour base 2.

Ex : Représentions des nombres de 0 à 16 en décimal et leurs équivalents en binaire et octal

Système décimal	Système octal	Système binaire
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000

2.5. le système hexadécimal

Le système hexadécimal utilise les 16 symboles suivant : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. De ce fait, le système a pour base 16.

Ex : si nous reprenons le tableau précédent mais en valeurs décimales et leurs équivalents binaires et hexadécimales, nous aurons :

Système décimal	Système hexadécimal	Système binaire

0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

3. Conversion et changement de base

3.1. Conversion d'un nombre de base *b* quelconque en nombre décimal

Tout nombre entier naturel peut se coder comme la somme pondérée des puissances de sa base ***b***, quel que soit cette base :

Soit $a_n a_{n-1} \ldots\ldots a_2 a_1 a_0$ exprimé en base ***b*** noté $(a_n a_{n-1} \ldots\ldots a_2 a_1 a_0)_b$. La valeur de ce nombre en décimal est égale à :

$$a_n \times b^n + a_{n-1} \times b^{n-1} + \ldots\ldots + a_2 \times b^2 + a_1 \times b^1 + a_0 \times b^0$$

Ex : Convertissons les nombres suivants en décimal : $(1011)_2$, $(16257)_8$ et $(F53)_{16}$

$(\mathbf{1011})_2 = (\mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0)_{10} = (\mathbf{1} \times 8 + \mathbf{0} \times 4 + \mathbf{1} \times 2 + \mathbf{1} \times 1)_{10} = (\mathbf{11})_{10}$

$(\mathbf{16257})_8 = \mathbf{1} \times 8^4 + \mathbf{6} \times 8^3 + \mathbf{2} \times 8^2 + \mathbf{5} \times 8^1 + \mathbf{7} \times 8^0 = \mathbf{1} \times 4096 + \mathbf{6} \times 512 + \mathbf{2} \times 64 + \mathbf{5} \times 8 + \mathbf{7}$
$= 4096 + 3072 + 128 + 40 + 7$
$= \mathbf{7343}$

$(\mathbf{F53})_{16} = \mathbf{15}\, x16^2 + \mathbf{5} \times 16^1 + \mathbf{3} \times 16^0 = \mathbf{15}\, x16^2 + \mathbf{5} \times 16^1 + \mathbf{3} \times 16^0 = \mathbf{15}\, x256 + \mathbf{5} \times 16 + \mathbf{3} = 3840 + 80 + 3 = \mathbf{3923}$

☝***Remarque :***
Dans le cas où il y a une partie fractionnaire ***$a_1 a_2 \ldots\ldots a_n$*** *(les nombres fractionnaires sont ceux qui comportent des chiffres après la virgule), sa valeur sera égale en décimal à la somme suivante :* $a_1 \times b^{-1} + a_2 x\, b^{-2} + \ldots\ldots + a_n \times b^{-n}$

Ex : Convertissons les nombres fractionnaires suivants en décimal : $(1110.101)_2$, $(642.21)_8$ et $(A3F.C)_{16}$

$(\mathbf{1110.101})_2 = (\mathbf{1}\times 2^3 + \mathbf{1}\times 2^2 + \mathbf{1}\times 2^1 + \mathbf{0}\times 2^0 + \mathbf{1}\times 2^{-1} + \mathbf{0}\times 2^{-2} + \mathbf{1}\times 2^{-3})_{10} = (8+4+2+0+1/2+1/4+1/8)_{10} = (\mathbf{14.625})_{10}$

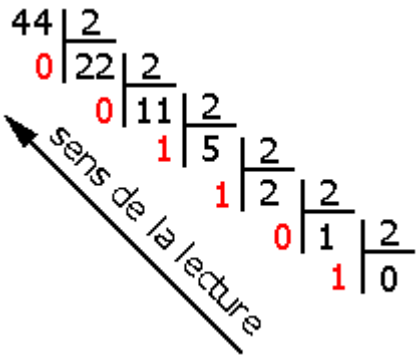
$(\mathbf{642.21})_8 = (\mathbf{6}\times 8^2 + \mathbf{4}\times 8^1 + \mathbf{2}\times 8^0 + \mathbf{2}\times 8^{-1} + \mathbf{1}\times 8^{-2})_{10} = (\mathbf{6}\times 64 + \mathbf{4}\times 8 + \mathbf{2}\times 1 + \mathbf{2}\times 0.125 + \mathbf{1}\times 0.015625)_{10}$
$= (384 + 32 + 2 + 0.25 + 0.015625)_{10} = (\mathbf{418.265\ 625})_{10}$

$(\mathbf{A3F.C})_{16} = (\mathbf{10}\times 16^2 + \mathbf{3}\times 16^1 + \mathbf{15}\times 16^0 + \mathbf{12}\times 16^{-1})_{10} = (\mathbf{10}\times 256 + \mathbf{3}\times 16 + \mathbf{15}\times 1 + \mathbf{12}\times 0.0625)_{10} = (\mathbf{2623.75})_{10}$

3.2. Conversion d'un nombre décimal en nombre binaire

Pour obtenir l'expression binaire d'un nombre exprimé en décimal, il suffit de diviser successivement ce nombre par **2** jusqu'à ce que le quotient obtenu soit égal à **0**. Les restes de ces divisions lus de bas en haut représentent le nombre binaire.

Ex : Convertissons le nombre décimal 44 en binaire :



$(\mathbf{44})_{10} = (\mathbf{101100})_2$

3.2.1.Conversion partie décimal -fractionnaire→binaire

La partie entière est converti du décimal en binaire en utilisant la méthode de la division successive comme on vient de le voir, quand à a partie fractionnaire du nombre décimal, elle sera converti en partie fractionnaire binaire en utilisant une méthode de multiplication successives. La partie entière du résultat de chaque multiplication est considérée comme constituant de la partie fractionnaire binaire et la partie fractionnaire du résultat de la multiplication est repris pour être lui-même multiplié par 2, et ainsi de suite jusqu'à ce que le produit obtenu est 1.00, le processus de conversion est alors achevé.

Ex : Convertissons les nombres fractionnaires décimaux 0.375 et 0.84375 en binaire :

0.375 × 2 = **0**.75

0.75 × 2 = **1**.50

0.50 × 2 = **1**.00

$(0.375)_{10}=(0.\mathbf{0\ 1\ 1})_2$

0.84375 × 2 = **1**.6875

0.6875 × 2 = **1**.375

0.375 × 2 = **0**.75

0.75× 2 = **1**.50

0.50× 2 = **1**.00

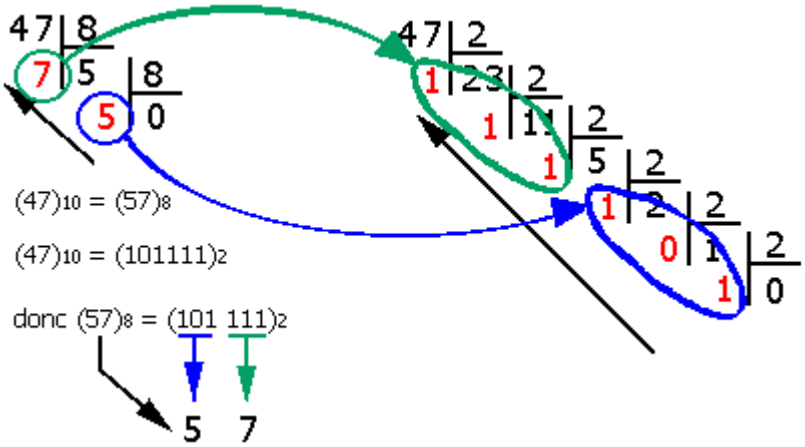
$(0.84375)_{10}=(0.\mathbf{11011})_2$

3.3. Relation entre les nombres binaires et les nombres octaux

D'abord, si nous souhaitons obtenir l'expression octale d'un nombre exprimé en décimal, il suffit de suivre la méthode de la division successive par **8** (comme nous l'avons fait pour

convertir vers la base **2**) jusqu'à ce que le quotient obtenu soit égal à **0**. Les restes de ces divisions lus de bas en haut représentent le nombre octal.

Ex : Convertissons $(47)_{10}$ dans le système octal et le système binaire. Nous obtenons :



Nous pouvons remarquer qu'après 3 divisions en binaire nous avons le même quotient qu'après une seule en octal. De plus le premier reste en octal obtenu peut être mis en relation directe avec les trois premiers restes en binaire :

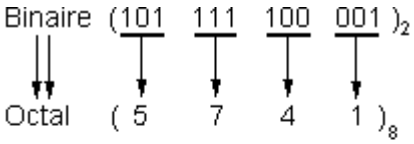
$(111)_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1 \times 4 + 1 \times 2 + 1 \times 1 = (7)_8$

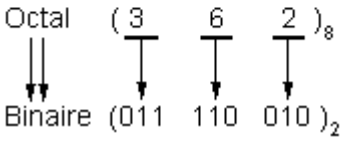
et il en est de même pour le caractère octal suivant :

$(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 4 + 0 \times 2 + 1 \times 1 = (5)_8$

Cette propriété d'équivalence entre chaque chiffre octal et chaque groupe de 3 chiffres binaires vient du fait que 8 est une puissance de 2 : **$8=2^3$**. Elle nous permet de passer facilement d'un système à base 8 à un système à base 2 et vice versa.

Exemple de conversion binaire octal et octal binaire :

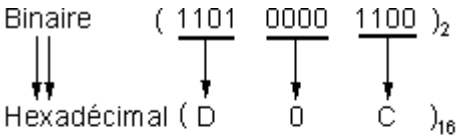


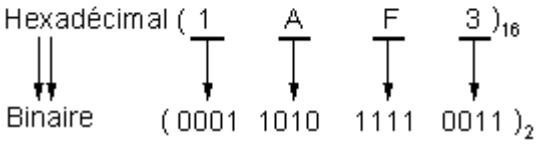


3.4. Relation entre les nombres binaires et les nombres hexadécimaux

Si nous avons besoin d'obtenir l'expression hexadécimal d'un nombre exprimé en décimal, il faut toujours suivre la méthode de la division successive, cette fois-ci par **16** (comme nous l'avons fait pour convertir vers les bases **2** et **8**) jusqu'à ce que le quotient obtenu soit égal à **0**. Les restes de ces divisions lus de bas en haut représentent le nombre hexadécimal (tout en prenant compte que les restes de **10** à **15** sont codés : de **A** à **F**).

La propriété d'équivalence que nous venons de voir, dans **3.3**, entre le binaire et l'octal existe entre l'hexadécimal et le binaire du moment que 16 est aussi une puissance de 2 : **$16=2^4$**. Donc la règle est la même mais nous travaillerons par groupe de **4** chiffres binaires maintenant au lieu de 3.





Remarques :

1 : *Pour la conversion de tout nombre entier de la base 10 vers une base quelconque, on procède toujours par divisions successives. On divise le nombre à convertir par la base dans laquelle nous voulons le convertir, puis le quotient obtenu par la base, et ainsi de suite jusqu'a obtention d'un quotient nul. La suite des restes obtenus correspond aux chiffres dans la base visée.*

2 : *Pour la conversion de la partie fractionnaire décimal vers son équivalent octal (ou hexadécimal), on procède toujours par multiplications successives comme on a fait pour la conversion décimal-fractionnaire vers le binaire dans la section **3.2.1**. On multiplie la partie fractionnaire par 8 (ou 16), la partie entière du résultat entre dans la constitution de la partie fractionnaire octale (hexadécimal) ensuite, la partie fractionnaire du résultat de la multiplication est lui-même multiplié à nouveau par 8 (par 16) et ainsi de suite jusqu'à obtention d'un résultat égal à 0.00.*

Ex : Convertissons le nombre décimal 418.265 625 en octal :

418 ÷ 8 = **52**	reste	**2**	0. 265 625 × 8 = **2**.125
52 ÷ 8 = **6**	reste	**4**	0.125 × 8 = **1**.00
6 ÷ 8 = **0**	reste	**6**	0.00 × 8 = **0**.00

Donc :

$(418.265\ 625)_{10}=(\mathbf{642.21})_8$

(De la même façon, on procède pour la conversion fractionnaire décimale vers son équivalent hexadécimal)

4. Les opérations de base en binaire

4.1. Addition et multiplication binaire

Le principe du calcul numérique est le même dans les systèmes de numération de position. Donc nous raisonnerons de la même façon que pour réaliser des opérations dans le système décimal où on a l'habitude d'effectuer nos opérations arithmétique quotidiennement.

Addition binaire

Révisons d'abord la familière addition décimale. L'addition de 2 nombres décimaux s'effectue selon un algorithme à 3 étapes :

Etape 1 : ajouter les chiffres les plus à droite (première colonne)

Etape 2 : noter le chiffre d'unité de cette somme à la même colonne toujours et si cette somme dépasse **9**, on reporte à la colonne suivante la retenue (chiffre de deuxième position de la somme obtenue)

Etape 3 : s'il y a d'autres colonnes, répéter les 2 étapes précédentes sans oublier de rajouter la retenue jusqu'à ce qu'il n'y ait plus de colonnes.

En binaire, l'algorithme est le même sauf que la retenue sera en cas où la somme dépasse **2** (et non pas **9**), tel que :

0 + 0 = 0 **0 + 1 = 1** **1 + 0 = 1** **1 + 1 = 0 avec retenue de 1** **1 + 1+ 1 = 1 avec retenue de 1**

Ex : évaluons la somme binaire : 111+101

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 111 \\ +\ 101 \\ \hline 1100 \end{array}$$

Multiplication binaire

Se résume comme en décimal en multiplication de nombres par des chiffres suivi d'additions décalée. En fait, en binaire c'est encore plus simple du moment que la multiplication par 0 ou 1 donne 0 ou le nombre lui-même (pas de tableaux de multiplication à apprendre comme en décimal !)

Ex : évaluons le produit binaire : 1101011 × 10110

$$\begin{array}{r} 1101011 \\ \times\ \ 10110 \\ \hline 0000000 \\ 1101011 \\ 1101011 \\ 0000000 \\ 1101011 \end{array}$$

ça revient à faire ceci si on procède à l'addition des termes un à un et éliminons les lignes à 0 sans oublier de prendre le décalage en considération :

$$\begin{array}{r} 1101011 \\ \times\ \ 10110 \\ \hline 11010110 \\ +\ \ 1101011 \\ \hline 1010000010 \\ +1101011 \\ \hline 100100110010 \end{array}$$

Donc : 1101011 × 10110 = 100100110010

☝***Remarque :***

pour les multiplications des nombres fractionnaires, la règle est la même qu'en décimal.

Ex : évaluons le produit binaire : 11.01 × 101.1

```
    11.01
×   101.1
-------
    1101
   1101
-------
  100111
 1101
-------
10001.111
-------
```

4.2. Soustraction binaire

La soustraction s'effectue suivant le principe de retenue comme en décimal

0 - 0 = 0 1 - 1 = 0 1 - 0 = 1 0 - 1 = 1 avec retenue sur la colonne suivante

Ex : évaluons les soustractions suivantes :

```
    0          011          00 0 1 00
  11101        11000        1101.00110
- 1011       - 10011      - 110.11011
-------      -------      -----------
  10010        101          110.01011
```

☝***Remarque :***

1 : *Dans le cas de fractions, il faut d'abord aligner verticalement les virgules avant de commencer l'opération de soustraction.*

2 : Quand dans une colonne, apparait la différence 0-1, nous opéron une retenue sur la 1ère colonne non nulle et tous les 0 juste avant deviennent des 1.

4.3. Division binaire

Il s'agit de multiplications et de soustractions successives comme en décimal. En cas de nombres fractionnaires, on déplace d'abord la virgule, ensuite on effectue l'opération de division

Ex : effectuons les divisions : 1010001 ÷ 11 et 111.00001 ÷ 1.01, nous avons :

```
1010001 | 11                111100.001 | 101
  100   | 11011               100      | 101.101
    10                        1000
     1                          110
     10                           10
      11                          101
       0
```

☝***Remarque :***

Nous avons étudié ces opérations du côté purement arithmétiques, mais du point de vue 'structure machine' il peut y avoir quelques problèmes qui peuvent être détectés pour ne pas fournir un résultat faux. Par exemple, si on travaille sur 6 bits, l'addition suivante : ***111001 + 010010*** *fournit un résultat sur 7 bit :1001011, donc le 1 le plus à gauche sera perdu ! on parle alors de dépassement de capacité (overflow). Il faut donc qu'il y ait un indicateur de dépassement et l'erreur doit être signalée.*

5. Les entiers négatifs

Les entiers négatifs peuvent être codés selon 3 méthodes : signe et valeur absolue, complément à 1 (complément logique) et complément à 2 (ou arithmétique).

5.1. Signe et valeur absolue

Les entiers sont codés de la façon suivante : ± *valeur absolue*. On sacrifie un bit pour représenter le signe. On représentera le signe + par un 0 et le signe – par un 1. On peut ainsi avec un mot de *k* bits, coder les entiers positifs ou négatifs N, tel que N est dans l'intervalle :

$$-(2^{k-1}-1) \leq \mathbf{N} \leq +(2^{k-1}-1)$$

L'inconvénient de cette méthode est que le 0 a deux représentation distinctes : 000..0 et 10..0, soit +0 et -0. Aussi, les opérations arithmétiques sont compliquées à cause du bit de signe qui doit être traité à part.

5.2. Les compléments à 1 et à 2

On calcule le complément logique (complément à 1) en remplaçant pour les valeurs négatives, chaque bit à 0 par 1 et vice-versa dans la valeur absolue. Le complément à 2 (arithmétique) est obtenu par addition du (*complément à 1)* +1.

En complément à 1 aussi et pour k bits, on a l'intervalle suivant : $-(2^{k-1}-1) \leq \mathbf{N} \leq +(2^{k-1}-1)$, mais en complément à 2 l'intervalle est : $-2^{k-1} \leq \mathbf{N} \leq +(2^{k-1}-1)$ (càd : on a une valeur de plus car elle évite la double représentation du zéro)

Ex : représentation de (-6) sur 4 bits :
En signe et valeur absolue : **1110**
En complément à 1 : **1001**
En complément à 2 : 1001 + 1 = **1010**

Q : quelle est l'intervalle des nombres signés qu'on peut représenter sur 4 bits pour les 3 méthodes ?

On remarque que le bit le plus à gauche (bit de signe), dans les trois méthodes, est toujours à 1 si le nombre est négatif et est à 0 si le nombre est positif.

En complément à 1 et à 2, les opérations arithmétiques sont avantageuses car la soustraction d'un nombre se réduit à l'addition de son complément. On n'utilisera que des circuits réalisant l'addition et il n'y a pas de traitement particulier pour le bit de signe.

Dans une addition en complément à 1, une retenue générée par le bit de signe doit être ajoutée au résultat obtenu. Par contre, en complément à 2, on ignore tout simplement cette retenue.

Ex : soustraction de nombres sur 4 bits :

Décimal	**signe+val.abs**	**compl à 1**	**compl à 2**
+7	0111	0111	0111
-6	+ 1110	+ 1001	+ 1010
+1	**?101**	**10000**	**10001**
		1	**0001**
		0001	

Remarque :

En complément à 2, un dépassement de capacité ne se produit que si les retenues générées juste avant le bit de signe et par le bit de signe lui-même sont différentes.

***Ex :** addition en complément à 2 sur 3 bits :*

(-4)	100
+ (-1)	111
-5	**~~1~~011 ≠ -5 , donc le résultat est faux !**

6. Les nombres fractionnaires

6.1. Représentation en virgule flottante

Nous savons qu'il est nécessaire de stocker des données dans les machines. Ainsi le nombre **9,750** se trouvera mémorisé sous la forme suivante : **1001,11**. Toutefois cette expression binaire ne suffit pas à définir totalement notre donnée car il n'y a aucune indication sur la valeur du poids binaire affecté aux différents bits, d'où la notion de virgule.
En utilisant cette notion de virgule, notre nombre peut s'écrire de la manière suivante :
$N = 1001,11 \times 2^0$
$N = 100,111 \times 2^1$
$N = 10,0111 \times 2^2$
$N = 1,00111 \times 2^3$
$N = 0,100111 \times 2^4$
Cette dernière expression présente l'avantage de représenter la grandeur par un nombre inférieur à 1 multiplié par une puissance de 2. L'exposant 4 (100 en binaire) est bien entendu représentatif de la position de la virgule. Donc pour définir totalement notre information (9,750) il faudra dans ce système de représentation deux termes : le terme **100111** appelé ***Mantisse*** et le terme **100** appelé ***Exposant.***

Donc, la représentation en virgule flottante consiste à représenter les nombres **N** sous la forme suivante : $\mathbf{N=M \times B^E}$ avec : B : base (dans notre cas, on étudie B=2)
M : Mantisse
E : exposant
L'exposant est un entier, la mantisse un nombre purement fractionnaire (n'ayant pas de chiffres significatifs à gauche de la virgule). Celle-ci est normalisée, c'est-à-dire qu'elle comporte le maximum de chiffres significatifs : le premier bit à droite de la virgule et à 1 (ex : 0.101110). A l'exception de la valeur 0 (qui est en général représentée par le mot 00…0), on a donc toujours :

$$0.1_2 \leq |M| < 1_2 \text{ soit } 0.5 \leq |M| < 1_{10}$$

Exposant et mantisse doivent pouvoir représenter des nombres positifs ou négatifs, donc pourraient être codés sous la forme signe+val.abs, complément à 1 ou complément à 2. Souvent, la mantisse est de la forme signe+val.abs et l'exposant est sans signe, mais **biaisé** (ou **décalé**).
Ex :

SM	E	M

Où SM est le signe de la mantisse, E l'exposant biaisé et M la mantisse.
Avec 4 bits, par exemple, on peut représenter 2^4= 16 valeurs de E, qui vont de 0 à 15. On peut faire correspondre les 8 premières valeurs (de 0 à 7) à un exposant <0 et les 8 suivants (de 8 à 15) à un exposant ≥0. Un exposant nul est ainsi représenté par la valeur 8, un exposant égal à +1 par la valeur 9, un exposant égal à -1 par la valeur 7. On dit que le biais est égal à 8. C'est la valeur qu'il faut soustraire à l'exposant biaisé (de 0 à 15) pour obtenir l'exposant effectif (de -8 à +7).

Ex : Représentation d'entiers signés sur 3 bits (exposant codé sur 3 bits →8 valeurs possibles entre 0 et 7)

Décimal	signe et val.abs	C à 2	représentations biaisées
+3	011	011	111
+2	010	010	110
+1	001	001	101
0	000/100	000	100
-1	101	111	011
-2	110	110	010
-3	111	101	001
-4	-----	100	000

On peut remarquer que la représentation biaisée est identique au complément à 2, à l'exception du bit de signe, qui est inversé (représentation biaisée : bit de signe à 1 → valeur ≥0, bit à 0 →valeur <0).
L'*exposant* détermine l'**intervalle** des nombres représentables et la taille de la *mantisse* détermine la **précision** de ces nombres.

6.1.1. Les opérations arithmétiques en virgule flottante

Pour la **multiplication**, il suffit d'additionner les exposants, de multiplier les mantisses et de renormaliser le résultat si nécessaire.
Ex : $(0.2 \times 10^{-3}) \times (0.3 \times 10^{7}) = ?$
-addition des exposants : $-3 + 7 = 4$
- multiplication des mantisses : $0.2 \times 0.3 = 0.06$
- résultat avant normalisation : 0.06×10^{4}
- résultat normalisé : 0.6×10^{3}

Pour la **division,** il suffit de soustraire les exposants et diviser les mantisses et de renormaliser le résultat si nécessaire.

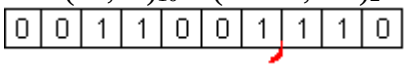
Pour l'**addition,** il faut que les exposants aient la même valeur ; on est donc obligé de dénormaliser la plus petite valeur pour amener son exposant à la même valeur que celui du plus grand nombre. Après avoir additionné les mantisses, une normalisation peut être nécessaire.
Ex : $(0.300 \times 10^{4}) + (0.998 \times 10^{6}) = ?$
-dénormalisation : $0.300 \times 10^{4} \rightarrow 0.003 \times 10^{6}$
- addition des mantisses : $0.003 + 0.998 = 1.001$
- normalisation du résultat : $1.001 \times 10^{6} \rightarrow 0.1001 \times 10^{7}$

La **soustraction** s'effectue de la même façon que l'addition, sauf que l'on doit effectuer la soustraction et non plus l'addition des mantisses.

6.2. Représentation en virgule fixe

La représentation de nombre en virgule flottante n'est pas la seule imaginable. Il existe la représentation de nombres en virgule fixe. La différence de base avec la représentation en virgule flottante est, comme son nom l'indique, le nombre de chiffres après la virgule (le plus à droite) est toujours le même, donc la précision des nombres fractionnaires représentés par virgule fixe est toujours la même.

Ex :
Soit $(25,75)_{10} = (11001,110)_2$



Dans cette configuration par exemple, la position de la virgule est **fixe** (entre le 3ème et le 4ème

bit), mais comme la virgule n'est pas réellement visualisée ou représentée, à la base la case la plus à droite représente le poids 2^0 : ce qui est évidemment faux dans notre cas. Cette représentation suppose la multiplication implicite de ce nombre par 2^{-3} pour obtenir la valeur exacte. Le terme $^{-3}$ est représentatif du positionnement fixe de la virgule. Il devra impérativement être mémorisé dans la machine.

7. Les différents codages

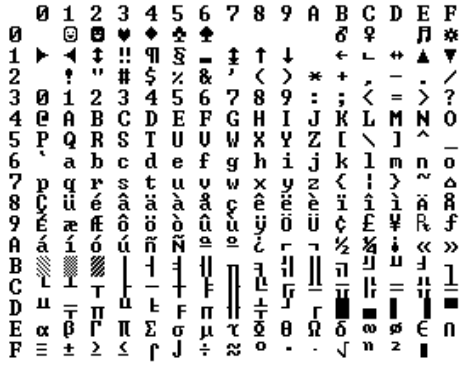
Le code BCD : Abréviation de *Binary Coded Decimal* en anglais et **DCB** (*Décimal Codé Binaire*). Ce code cherche à concilier les avantages du système décimal et du code binaire. Il est surtout utilisé pour l'affichage de données décimales (calculatrices par exemple). A chaque chiffre du système décimal, on fait correspondre un mot binaire (de quatre bits en général).
Pour coder un nombre décimal en BCD, on va coder séparément chaque chiffre du nombre de base dix en Binaire.
Ex : (BCD sur 4 bits) : 1985 = 0001 1001 1000 $0101_{(BCD)}$

☝***Remarque :***
. Le nombre codé en BCD ne correspond pas au nombre décimal converti en binaire naturel.
. Le codage décimal BCD est simple, mais il n'est pas possible de faire des opérations mathématiques directement dessus.
. Il existe plusieurs types de codes DCB, mais le plus connu est celui présenté dans cette section.

Le code EBCDIC (*Extended Binary Coded Decimal Interchange)* : ce code est utilisé principalement par IBM. Il est représenté sur 8 bits et est utilisé dans le codage de caractère, c'est-à-dire, pour chaque caractère est associé son code EBCDIC.
Ex : code du caractère **A** (en majuscule) en **EBCDIC** = **11000001**
code du caractère **0** en **EBCDIC** = **11110000**

Le code ASCII : Ce code propose des extensions différentes selon le " code de page ". Le code de page 850 est un jeu de caractères " multilingue " alors que le code de page 864 définit le jeu de caractères arabes, le code de page 437 définit le jeu de caractères français...etc

La table des codes ASCII



Remarque : La table des codes ASCII ci-dessus, affiche les caractères imprimables et non les codes de contrôle. En effet les caractères dont les codes sont 10, 13 et 27 en décimal, représentent respectivement Line Feed (Aller à la ligne), Carriage Return (Retour Chariot) et Scape (Escape), le tableau ci-dessous en donne quelques exemples.

Partie II : Algèbre de Boole

1. Introduction

C'est une algèbre binaire mise en œuvre par le mathématicien anglais **George BOOLE** (1815-1864) pour étudier la logique. Les variables, dites booléennes, ne peuvent prendre que deux valeurs : **VRAI** ou **FAUX** ou bien encore 1 ou 0. On peut alors définir des opérateurs sur ces variables et consigner le résultat dans une **TABLE DE VERITE**. Ces opérateurs peuvent être réalisés par des circuits électroniques : ils sont alors appelés **PORTES LOGIQUES.**

2. Définition et propriétés de l'algèbre de Boole

Soit *B* un ensemble sur lequel sont définies deux opérations binaires + et ×, et une opération unitaire, notée ' ; soient 0 et 1 deux éléments distincts de B. Dans ces conditions, le sextuplet : (*B*,+,×, ', 0, 1) est une algèbre de Boole si les postulats suivants, relatifs à des éléments quelconques a, b et c de *B* sont satisfaits :

Commutativité	$a + b = b + a$	$a \times b = b \times a$
Distributivité	$a + (b \times c) = (a + b) \times (a + c)$	$a \times (b + c) = (a \times b) + (a \times c)$
Identité	$a + 0 = a$	$a \times 1 = a$
Complémentarité	$a + a' = 1$	$a \times a' = 0$

0 est dit *l'élément nul*, 1 est *l'élément unité* et a' (ou $\bar{a}$) est le complément de *a.* Les opérations + et × sont appelées respectivement : somme et produit. Souvent le nous écrivons le symbole × sous la forme **.** (un simple point) ou bien nous l'ignorons tout simplement. C'est-à-dire : $a \times (b + c) = a \,.\, (b + c) = a\,(b + c)$

2.1. Priorité des opérateurs

La première priorité est aux parenthèses, ensuite la priorité est à la négation ′ (ou $\bar{\ }$) ensuite à l'opérateur × et enfin à l'opérateur +.

Ex :

$a + b \times c$ signifie $a + (b \times c)$ et non pas $(a + b) \times c$

$a \times b'$ signifie $a \times (b')$ et non pas $(a \times b)'$

☝***Remarque :***

Attention ! Bien que leur nom et leur symbole se ressemblent, ***ne confondons pas la somme et le produit logiques avec la somme et le produit arithmétiques, tels que nous les connaissons.*** *Les premiers s'exercent sur des valeurs logiques, les seconds sur des nombres.*

Convention :

Puisque notre utilisation de l'algèbre de Boole sera exclusivement pour réaliser des circuits logiques à partir de portes logiques, nous appellerons dès maintenant l'opérateur × par ET logique (AND) et l'opérateur + par OU logique(OR). Quand au complément de a : $\bar{a}$, il représente la négation logique de a, c'est-à-dire NON a (NOT a).

Ex :

Soit B l'ensemble {0,1} sur lequel sont définies les opérateurs + et ×. Supposons que les compléments soient définis par $\bar{1} = 0$ et $\bar{0} = 1$, alors B est une algèbre de Boole.

+	**1**	**0**
1	1	1
0	1	0

×	**1**	**0**
1	1	0
0	0	0

2.2. Principe de Dualité

Le dual d'un énoncé quelconque d'une algèbre de Boole *B* est l'énoncé obtenu par inversion des opérateurs + et × ainsi que des éléments 0 et 1 dans l'expression originale. Le dual d'une expression vérifiée vraie est également vrai. C'est-à-dire que si l'expression d'origine est vraie, son dual est vrai également.

Ex :
Le dual de l'expression booléenne $(1 + a) \times (b + 0) = b$ *est* $(0 \times a) + (b \times 1) = b$

2.3. Théorèmes fondamentaux

Soient *a, b* et *c* des éléments d'une algèbre de Boole :

***1. Loi d'idempotence* :** $a + a = a$ $\quad$ $a \times a = a$
$a + 1 = 1$ $\quad$ $a \times 0 = 0$

***2. Loi d'absorption* :** $a + (a \times b) = a$ $\quad$ $a \times (a + b) = a$

***3. Loi d'associativité* :** $(a + b) + c = a + (b + c)$ $\quad$ $(a \times b) \times c = a \times (b \times c)$

***4. Unicité du complément* :** si $a + x = 1$ et $a \times x = 0$ → $x = \bar{a}$

C.à.d : $a + \bar{a} = 1$ et $a \times \bar{a} = 0$

5. Loi d'involution : $\bar{\bar{a}} = a$

$\bar{0} = 1$ et $\bar{1} = 0$

6. Loi de Morgan : $\overline{(a+b)} = \bar{a} \times \bar{b}$ $\quad$ $\overline{(a \times b)} = \bar{a} + \bar{b}$

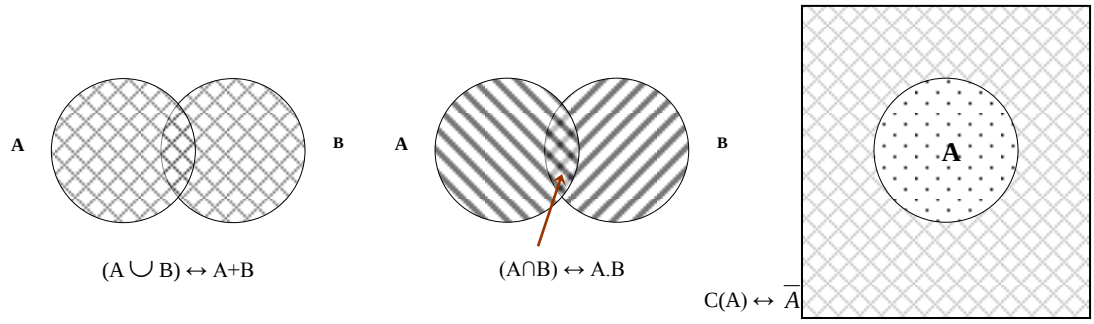
Généralisation de la loi de Morgan :

$\overline{(a+b+c+\cdots)} = \bar{a} \times \bar{b} \times \bar{c} \times \cdots$ $\quad$ $\overline{(a \times b \times c \times \cdots)} = \bar{a} + \bar{b} + \bar{c} + \cdots$

Remarques (Diagramme de Venn):

Il existe un moyen très visuel de traduire ces concepts un peu abstraits: il consiste à considérer les classes de Boole comme des ensembles. Dès lors, les opérations fondamentales de Boole prennent la forme d'opérateurs sur les ensembles et on peut alors utiliser des diagrammes de Venn pour les représenter:

- La somme logique de deux classes (A + B) se traduit par l'union ($A \cup B$) entre les deux ensembles correspondants
- Le produit logique (A . B) par l'intersection ($A \cap B$),
- Le complément $\bar{A}$ de A par le complément d'un ensemble A.



$(A \cup B) \leftrightarrow A+B$ $(A \cap B) \leftrightarrow A.B$ $C(A) \leftrightarrow \overline{A}$

Ex : Simplifions les expressions suivantes autant que possible en utilisant les différentes lois vues au cours :

$E_1 = a + (\overline{a}b)$ et $E_2 = (a+b)\cdot(a+\overline{b})$

$E_1 = a + (\overline{a}b) = (a+\overline{a}).(a+b) = 1.(a+b) = a+b$

$E_2 = (a+b)\cdot(a+\overline{b}) =$

$a.(a+\overline{b}) + b.(a+\overline{b}) = aa + a\overline{b} + b\overline{b} = a + a\overline{b} + ab = a + a(\overline{b}+b) = a + a.1 = a + a = a$

2.4. Fonction Booléenne

$F(A,B,C) = AB\overline{C} + A\overline{C}B + \overline{A}BC$

C'est une fonction qui relie N variables logiques avec un ensemble d'opérateurs logiques. Sachant qu'une variable logique (booléenne) est une variable qui peut prendre soit la valeur 0 ou 1. La valeur d'une fonction logique est égale à 1 ou 0 également, selon les valeurs des variables logiques. Si une fonction logique possède N variables logiques, ça implique qu'il peut y avoir 2^n combinaisons de ses variables, donc cette même fonction peut avoir 2^n valeurs. Les 2^n combinaisons sont représentées dans une table appelée table de vérité (T.V).

Ex :

Soit la fonction logique F (A,B,C)= $\overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + ABC$

A	B	C		F
0	0	0		0
0	0	1		1
0	1	0		0
0	1	1		1
1	0	0		0
1	0	1		1
1	1	0		0
1	1	1		1

2.4.1. Forme canonique d'une fonction logique

On appel forme canonique d'une fonction la forme ou chaque terme de la fonction comportent toutes les variables.

Ex :

La fonction $F(A,B,C) = AB\overline{C} + A\overline{C}B + \overline{A}BC$ est sous forme canonique.

2.4.1.1. **Première forme canonique** (forme disjonctive)

C'est la forme exprimée en somme de produits (ou somme des **mintermes**). On dit aussi que c'est une disjonction de conjonctions. Cette forme est la forme la plus utilisée.

Ex :
La fonction suivante est sous la première forme canonique :

$$F(A,B,C) = \overline{A}.B.C + A.\overline{B}.C + A.B.\overline{C} + A.B.C$$

2.4.1.2. Deuxième forme canonique (forme conjonctive)

C'est la forme exprimée en produit de sommes (ou produit de **maxtermes**). On dit aussi que c'est une conjonction de disjonctions. Les deux formes (1ère et 2ème) canoniques sont équivalentes.

Ex :
La fonction suivante est sous la deuxième forme canonique :

$$F(A,B,C) = (A+B+C)\ (A+B+\overline{C})(A+\overline{B}+C)\ (\overline{A}+B+C)$$

2.4.2. Extraction de la fonction logique à partir de la T.V

Une fonction peut être exprimée sous sa première forme canonique ou sous sa deuxième forme canonique à partir de sa propre table de vérité. La meilleure façon d'illustrer ça, sera par un exemple :

A	B	C		F		
0	0	0		0	$\longrightarrow$	$A+B+C$
0	0	1		0	$\longrightarrow$	$A+B+\overline{C}$ ***maxterme***
0	1	0		0	$\longrightarrow$	$A+\overline{B}+C$
0	1	1		1	$\longrightarrow$	$\overline{A}.B.C$ ***minterme***
1	0	0		0	$\longrightarrow$	$\overline{A}+B+C$
1	0	1		1	$\longrightarrow$	$A.\overline{B}.C$
1	1	0		1	$\longrightarrow$	$A.B.\overline{C}$
1	1	1		1	$\longrightarrow$	$A.B.C$

On peut exprimer la fonction F sous forme de produits de maxtermes (2ème FC) à partir des lignes contenant 0 :

$$F(A,B,C) = (A+B+C)\ (A+B+\overline{C})(A+\overline{B}+C)\ (\overline{A}+B+C)$$

ou sous forme de sommes de mintermes (1ère FC) à partir des lignes contenant 1 dans la table de vérité de F.

$$F(A,B,C) = \overline{A}.B.C + A.\overline{B}.C + A.B.\overline{C} + A.B.C$$

Remarque : *'Comment Retrouver une forme canonique à partir d'une équation simplifiée ?'*

On peut toujours ramener n'importe qu'elle fonction logique à l'une des formes canoniques. Cela revient à rajouter les variables manquants dans les termes qui ne contiennent pas toutes les variables (les termes non canoniques). Cela est possible en utilisant les règles de l'algèbre de Boole :

- Multiplier un terme avec une expression qui vaut 1

- Additionner à un terme avec une expression qui vaut 0
- Par la suite faire la distribution
C'est-à-dire : il suffit de compléter la ou les variables manquantes dans chaque terme sans modifier l'état de la fonction. Par exemple, pour la fonction logique, H1, de trois variables (a, b, c) et dont l'équation logique "simplifiée" est :

$$H_1 = a.\bar{b} + a.b.\bar{c}$$

la première forme canonique peut être obtenue de la manière suivante :

$$H_1 = a.\bar{b} + a.b.\bar{c} = a.\bar{b}.(c + \bar{c}) + a.b.\bar{c} = a.\bar{b}.c + a.\bar{b}.\bar{c} + a.b.\bar{c}$$

On remarque que le terme ajouté ne modifie pas la fonction, car ce terme vaut 1

Exemples :

1. $F(A, B) = A + B$
$= A\,(B + \bar{B}) + B(\,A + \bar{A})$
$= AB + A\bar{B} + AB + \bar{A}B$
$= AB + A\bar{B} + \bar{A}B$

2. $F(A, B, C) = AB + C$
$= AB(C + \bar{C}) + C(\,A + \bar{A})$
$= ABC + AB\bar{C} + AC + \bar{A}C$
$= ABC + AB\bar{C} + AC(B + \bar{B}) + \bar{A}C\,(B + \bar{B})$
$= ABC + AB\bar{C} + ABC + A\bar{B}C + \bar{A}BC + \bar{A}\bar{B}C$
$= ABC + AB\bar{C} + A\bar{B}C + \bar{A}\,B\,C + \bar{A}\,\bar{B}\,C$

2.4.3. Complément d'une fonction

On obtient le complément d'une fonction (négation d'une fonction) par application de la loi de Morgan si elle est sous l'une de ses formes canoniques ou par inversion de sa table de vérité. En général, on déduit le complément d'une fonction sous l'une des deux formes canoniques en remplaçant les OU par des ET, les ET par des OU et chaque terme par son complément.

Ex1 :

$F = A + \bar{A}B + (C + \bar{D})$ $\qquad$ $\bar{F} = \overline{A + \bar{A}B + (C + \bar{D})} = \bar{A}.(\bar{\bar{A}} + \bar{B}).\bar{C}.\bar{\bar{D}} = \bar{A}.(A + \bar{B}).\bar{C}.D$

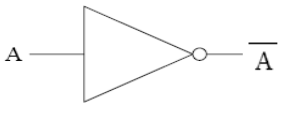
Ex2 :

Calculez le complément de la fonction présentée dans 2.4.2

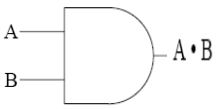
2.5. Portes logiques

Ces portes électroniques sont construites à partir de plusieurs transistors reliés entre eux.
Les circuits combinatoires les plus simples sont appelés portes logiques. Ils sont la base de la logique mathématique qui effectue les opérations à l'intérieur du processeur, et plus particulièrement à l'intérieur de l'UAL. C'est donc la base de tous les calculs internes du processeur. Nous allons décrire le fonctionnement des portes logiques les plus simples. Les portes logiques sont à l'origine de tous les calculs effectués dans le transistor. Leur

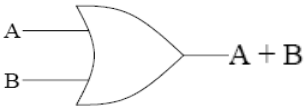
fonctionnement étant basé sur le passage éventuel du courant, elles ne peuvent que traiter des informations en langage binaire. Enfin l'association de portes logiques permet de traiter une instruction du microprocesseur (opérations simples par exemple).



NOT A



A **AND** B



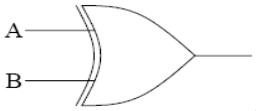
A **OR** B

2.5.1. Autres opérateurs logiques

2.5.1.1. OU exclusif (XOR)

L'opérateur XOR donne 1 si les deux variables sont de valeurs différentes, sinon donne 0. Sa table de vérité est donc la suivante :

A	B	**A⊕B**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**



A⊕B

Propriétés du XOR

$A \oplus B = \overline{A}B + A\overline{B}$ $\qquad \overline{A \oplus B} = AB + \overline{A}\overline{B}$

$A \oplus 0 = A$ $\qquad A \oplus A = 0$

$A \oplus 1 = \overline{A}$ $\qquad A \oplus \overline{A} = 1$

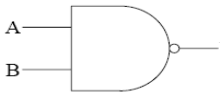
$A \oplus B = B \oplus A$ $\qquad (A \oplus B) \oplus C = A \oplus (B \oplus C)$

$A \oplus B = \overline{\overline{A}B + A\overline{B}} = (A + B)(\overline{A} + \overline{B}) = (A + B)\overline{AB}$

2.5.1.2.NON ET (NAND)

C'est le complément de la fonction ET (AND)

A	B	$A \uparrow B = \overline{A \cdot B}$
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

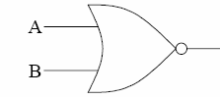


A NAND B

2.5.1.3.NON OU (NOR)

C'est le complément de la fonction OU (OR)

A	B	$A \downarrow B = \overline{A + B}$
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**



A NOR B

Propriétés des opérateurs NAND et NOR

A NAND 0 = 1
A NAND 1 = $\overline{A}$
A NAND B = B NAND A

A NOR 0 = $\overline{A}$
A NOR 1 = 0
A NOR B = B NAND A

Remarque :
Les opérateurs NOR et NAND ne sont pas associatifs

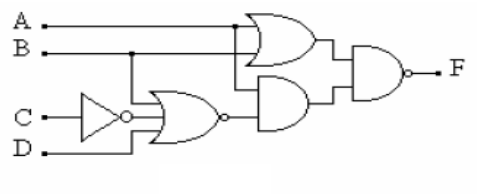
2.5.2 Schéma d'un circuit logique (Logigramme)

C'est un circuit qui combine les différentes portes logiques nécessaires de façon à représenter une fonction logique.

Exemple :

Le circuit logique représentant la fonction $F = \overline{(A + B).(\overline{B + \overline{C} + D}).A}$

est le suivant :



2.6. Simplification des fonctions logiques

2.6.1. Simplification algébrique

L'objectif de l'étude des fonctions Booléennes est principalement de matérialiser ces dernières dans le but de réaliser l'automatisme (côté électronique) après l'avoir représenté sous forme d'un logigramme (portes logiques). On peut se baser sur les différentes lois existantes, sur la mise en facteur successive, …etc. pour simplifier algébriquement une fonction logique, mais il n'est pas toujours envisageable de passer par ces méthodes pour la simplification, car ça peut être fastidieux et le résultat peut ne pas être optimal.

Exemples :

$f = xyz + \overline{x}yz + xy\overline{z} + x\overline{y}z = (\overline{x}yz + xyz) + (xy\overline{z} + xyz) + (x\overline{y}z + xyz)$

d'où $f = yz(x + \overline{x}) + xz(y + \overline{y}) + xy(z + \overline{z})$
d'où $f = yz(1) + xz(1) + xy(1)$
d'où $f = xy + yz + zx$

$S = (ab + c)(a\bar{b} + c)(\bar{a}b + c)(ab + \bar{c})$

$$S = \underbrace{ab\bar{a}\bar{b}ab}_{1} + \underbrace{ab\bar{a}\bar{b}\bar{c}}_{2} + \underbrace{ab\bar{a}\bar{b}cab}_{3} + \underbrace{ab\bar{a}\bar{b}c\bar{c}}_{4} + \underbrace{abc\bar{a}bab}_{5} + \underbrace{abc\bar{a}\bar{b}\bar{c}}_{6} + \underbrace{abccab}_{7} + \underbrace{abccc}_{8}$$
$$+ \underbrace{ca\bar{b}\bar{a}bab}_{9} + \underbrace{ca\bar{b}\bar{a}\bar{b}\bar{c}}_{10} + \underbrace{ca\bar{b}cab}_{11} + \underbrace{ca\bar{b}c\bar{c}}_{12} + \underbrace{cc\bar{a}bab}_{13} + \underbrace{cc\bar{a}\bar{b}\bar{c}}_{14} + \underbrace{cccab}_{15} + \underbrace{ccc\bar{c}}_{16}$$

Terme 1 : $ab\bar{a}\bar{b}ab = ab\,(\bar{a}\bar{b}ab)$ qui comme $\bar{a}\bar{b}ab = 0$ disparaît
Terme 2 : $ab\bar{a}\bar{b}\bar{c} = ab\bar{c}\,(\bar{a}\bar{b} \,.\, ab) = ab\bar{c}\,(a\bar{a})\,(b\bar{b})$ disparaît
Terme 3 : $ab\bar{a}\bar{b}cab = abc\,(b\bar{b})$ disparaît
Terme 4 : $ab\bar{a}\bar{b}c\bar{c} = a\,(bc \,.\, \bar{b}\bar{c}) = a\,(b\bar{b})\,(c\bar{c})$ disparaît
Terme 5 : $abc\bar{a}bab = bc\,(a\bar{a})$ disparaît
Terme 6 : $abc\bar{a}\bar{b}\bar{c} = b\,(a\bar{a} \,.\, c\bar{c})$ disparaît
Terme 7 : $abccab = abcabc = abc$ •
Terme 8 : $abcc\bar{c} = abc\bar{c} = ab\,(c\bar{c})$ disparaît
Terme 9 : $ca\bar{b}\bar{a}bab = ca\bar{b}\bar{a}b = c\,(\bar{a}b \,.\, a\bar{b}) = c\,(a \,.\, \bar{a})\,(b \,.\, \bar{b})$ disparaît
Terme 10 : $ca\bar{b}\bar{a}\bar{b}\bar{c} = ca\bar{b} \,.\, c\bar{a}b = (c \,.\, \bar{c})\,(a \,.\, \bar{a})\,(b \,.\, \bar{b})$ disparaît
Terme 11 : $ca\bar{b}cab = ca\,(b \,.\, \bar{b})$ disparaît
Terme 12 : $ca\bar{b}c\bar{c} = a\bar{b}\,(c \,.\, \bar{c}) = 0$ disparaît
Terme 13 : $cc\bar{a}bab = cb\,(a \,.\, \bar{a}) = 0$ disparaît
Terme 14 : $cc\bar{a}\bar{b}\bar{c} = \bar{a}b\,(c \,.\, \bar{c}) = 0$ disparaît
Terme 15 : $cccab = cab = abc$ •
Terme 16 : $ccc\bar{c} = c\bar{c} = 0$ disparaît

d'où $S = \underbrace{abc}_{7} + \underbrace{abc}_{15} = abc$ **S = abc**

Ce que nous constatons d'après ces exemples, c'est que la simplification est une étape fondamentale avant de passer à la réalisation des circuits, car elle permet de réaliser des économies en coût, en temps et en espace qu'occuperont les circuits intégrés.
Le souci avec la simplification algébrique c'est qu'elle est exposée à l'erreur et qu'elle ne nous fournira pas forcément la forme la plus simple de la fonction concernée, c'est pourquoi la méthode graphique des tableaux de Karnaugh est plus appropriée car elle ne présente pas les inconvénients de la simplification algébrique. Les tableaux de Karnaugh permettent de traiter des fonctions logiques à 2, 3, 4, voire 5 variables.

2.6.2. Simplification graphique

2.6.2.1. Méthode de Karnaugh

Cette méthode graphique permet la simplification de fonctions logique, en se basant sur la T.V de cette dernière. On pourra donc démarrer de la table de vérité elle-même ou de l'une des formes canoniques de la fonction à simplifier. Comme dans la méthode de Karnaugh, ce qui nous intéresse sont les '1' de la TV de la fonction, il est évident que c'est la 1ère FC qui nous facilitera la tâche, car elle nous indique directement où placer les '1' de cette fonction. Si par contre, on n'a pas une forme canonique de la fonction à simplifier, il faut la transformer d'abord algébriquement sous forme canonique.

2.6.2.2. Construction de la table de Karnaugh

Il s'agit d'un tableau à double entrées de $\mathbf{2^n}$ cases, tel que **n** est le nombre de variables dans la fonction à simplifier. Chaque case va correspondre à une ligne de la TV de la fonction. Les lignes et les colonnes du tableau sont numérotées de telle façon qu'en passant d'une case à une autre, une seule variable change de valeur (ex : dans la table de 3 variables : Les variables AB sont notées horizontalement dans l'ordre: **00 01 11** et enfin **10** et non pas : **00 01 10 11** pour que cette règle soit respectée).

A \ B	0	1
0		
1		

AB \ C	00	01	11	10
0				
1				

AB \ CD	00	01	11	10
00				
01				
11				
10				

Remarque : *Les tables de Karnaugh sont pratiques pour simplifier des fonctions logiques à 2, 3, 4, 5 ou 6 variables (au-delà de 6 variables, elle n'est plus pratique). Pour le cas de 5 variables par ex, il s'agit de dresser une table des 4 variables A B C et D supposant que la cinquième variable E=0 ensuite dresser une deuxième table des 4 variables A B C et D pour le cas E=1.*

2.6.2.3. Remplir la table de Karnaugh

Il s'agit de placer les « 1 » de la fonction logique à simplifier dans la table de vérité, ceci se fait à partir de la TV de la fonction à simplifier ou à partir de sa première forme canonique (les mintermes). Ça peut se faire aussi à partir de sa 2ème FC puisque cette dernière donne les « 0 » donc les cases vides sont des cases à « 1 ». Donc, ce qui nous intéresse dans cette méthode ce sont les cases à « 1 » qui vont être étudiées visuellement pour être groupées suivant des règles bien précises.

Ex : F(A,B)=A' B +A B'+ A B

A \ B	0	1
0		1
1	1	1

2.6.2.4. Méthode de travail

Il s'agit de constituer des groupes de « 1 » dans la table, tel que chaque groupe constituera un terme de la fonction simplifiée. Donc, il est déjà évident qu'on aura à former le minimum de groupes possibles. Dans chaque groupe il faut regrouper le maximum de « 1 » adjacents possibles, tel que deux cases sont adjacentes si en passant de l'une à l'autre, une et une seule variable change. Cependant quelques règles, à part ces deux-là, sont indispensables :

- La 1ère et la dernière ligne d'une table de Karnaugh sont considérées adjacentes.
- La 1ère et la dernière colonne d'une table de Karnaugh sont considérées adjacentes.

- Les 4 coins d'une table de Karnaugh sont considérées adjacents aussi.

AB / CD	00	01	11	10
00	■	■	■	■
01				
11				
10	■	■	■	■

AB / CD	00	01	11	10
00	■			■
01	■			■
11	■			■
10	■			■

AB / CD	00	01	11	10
00	■			■
01		x		
11			x	
10	■			■

- Les cases notées « x » ne sont pas adjacentes (adjacences en diagonal pas valide car ne respecte pas la règle d'adjacence précitée.
- Il faut former que le nombre de « 1 » dans chaque groupe soit une puissance de 2 (2, 4, 8, …).
- Les groupes de « 1 » peuvent se recouvrir (intersections possibles, mais sans redondances)
- Ne former un nouveau groupe que s'il permet de regrouper un ou des « 1 » qui n'ont pas encore été groupés.
- S'il reste à la fin un « 1 » isolé, il formera un groupe à lui seul.

2.6.2.5. Extraction de la fonction simplifiée

- De chaque groupe, éliminer la ou les variables qui apparaissent avec leurs compléments (car A+A'=1). Càd. On élimine de chaque groupe les variables qui changent de valeurs et on garde celles qui gardent la même valeur dans le même groupe.
- Chaque groupe donne naissance à un terme sous forme de produit de variables (qui gardent la même valeur au sein de leur groupe).
- Additionner logiquement les groupes ainsi obtenus pour obtenir à la fin la fonction simplifiée.

- **Bibliographie :**

‘**Techniques numériques**, cours et problèmes’, Série Schaum, Roger.Tokheim’
‘**Mathématiques pour Informaticiens**, cours et problèmes’, Série Seymour Lipschutz’
‘**Architecture et Technologie des ordinateurs**, édition Dunod, P.Zanella, Y.Ligier’