Mathématiques appliquées à l'informatique

Luc De Mey

Ces notes de cours sont disponibles à l'adresse : www.courstechinfo.be/Math_Info.pdf

Dernière révision : 6 mai 2013

Table des matières

1 Systèmes de numération ... 3

2 Ecriture des nombres entiers ou Numération de position ... 4

2.1 Numération de position ... 4
Exercices sur les nombres entiers en base 10 ... 5
2.1.1 Numération binaire ... 6
2.1.2 Numération hexadécimale ... 7

2.2 Calcul de la valeur d'un nombre quelle que soit la base ... 7

2.3 Transcriptions Binaires / Hexadécimale ... 8
2.3.1 Comptons en binaire et en hexadécimal ... 8
2.3.2 Conversions binaire ↔ octal ou binaire ↔ hexadécimal ... 8

2.4 Nombres de codes possibles avec N chiffres en base B ... 9

2.5 Préfixes pour représenter les puissances de 10^3 ... 10

2.6 Pour les informaticiens, 1 kilo est-ce 1000 ou 1024 ? ... 10

2.7 Calculs approximatifs de 2^n avec $n > 10$ ... 11
Exercices récapitulatifs ... 11

3 Conversion d'un nombre N entier en une base B quelconque ... 12

4 Autre méthode pour convertir d'une base B en base 10 « Méthode de Horner » ... 14

Exercices ... 14

5 Nombres binaires négatifs ... 15

5.1 Comment calculer les codes des nombres négatifs ? ... 17

5.2 La valeur du bit de signe ... 18

5.3 Conversions entre mots de différentes longueurs ... 18
Exercices ... 19

6 Opérations arithmétiques en binaires ... 20

6.1 Addition ... 20

6.2 Soustraction ... 20

6.3 Multiplication ... 21

6.4 Division ... 21

7 Opérations arithmétiques au cœur du PC ... 22

7.1 Nombre signés ou non ? ... 22

7.2 Au cœur du processeur avec DEBUG ... 22

7.3 Quelques manipulations avec DEBUG ... 22

7.4 Saisie du programme d'addition ... 23
7.4.1 Exécution du programme : ... 24
7.4.2 Vérification de la validité du résultat: ... 25

7.5 Exemples de calculs ... 25
Exercices ... 27

8 Codage des nombres réels ... 29

8.1 Utilité de la virgule flottante ... 29

8.2 Notation scientifique *30*
Exercices 30
8.3 Nombres fractionnaires binaires *30*
8.3.1 Conversion de nombre décimaux fractionnaires en binaire 31
8.3.2 Exercices 31
8.4 Nombres binaires en virgule flottante "Floating point" *31*
8.5 Représentation en machine *32*
8.6 Valeurs particulières *33*
9 Les fonctions logiques 35
9.1 La fonction ET *36*
9.2 La fonction OU *36*
9.3 La fonction NON *37*
9.4 Combinaisons de fonctions logiques *37*
9.5 Propriétés des fonctions logiques *38*
Exercices 39
10 Les portes logiques 41
10.1 Fonctions de base *41*
AND 41
OR 41
NOT 41
10.2 Combinaisons de portes logiques *42*
10.2.1 La porte NAND (Non ET) 42
10.2.2 Porte NOR (Non OU) 42
10.2.3 Porte XOR 42
11 Chronogrammes 44
12 Circuits logiques 46
12.1 Comparateur *46*
12.2 Décodeur *46*
12.3 Multiplexeur *47*
12.4 Démultiplexeur *47*
12.5 Le demi additionneur half adder → Addition de 2 bits = circuit à 2 entrées *48*
12.6 Le plein additionneur full adder *48*
12.7 Addition de deux nombres de n bits *49*
Exercices 50

1 Systèmes de numération

La numération est
une méthode pour former les nombres
une convention pour les écrire et les nommer

Pour compter, nous dénombrons une à une les unités. A partir d'une certaine quantité d'unités on crée un ensemble d'une valeur déterminée auquel on donne un nom et que l'on met sur le côté pour compter les unités suivantes jusqu'à ce qu'on puisse les regrouper dans une autre ensemble de même taille. Les regroupements d'unités sont à leur tour regroupés en nouveaux ensembles qui portent un autre nom encore.

Exemple :

100 Cents = 1 €
1000 gr =1 kg, 1000 kg = 1T
60 sec = 1 min, 60 min = 1h, 24h = 1 jour
1' = 60", 1 degré = 60', 1 tour = 360 °
1 Pouce = 2,54 cm ; 1 Pied = 12 Pouces ; 1 Yard = 3 Pieds ; 1 Mile = 1760 Yards

Dans la vie courante, on essaie de compter par dizaines, centaines, milliers… nous essayons de n'utiliser qu'une seule base: la base 10.

Les chiffres arabes sont des signes particuliers pour désigner les neufs premiers chiffres et le zéro. Dix signes nous suffisent pour écrire tous les nombres. Les unités sont autant que possible regroupées par dizaines, les dizaines par centaines etc.

Différentes bases :

Base 60	Utilisée en Mésopotamie. Il nous en reste 60 minutes, 60 secondes. Le nombre 60 a de nombreux diviseurs : 2, 3, 4, 5, 6, 10, 12, 15, 30.
Base 20	Utilisée par nos ancêtres gaulois, il nous reste le "quatre-vingts". Quatre-vingt-dix, Soixante-quinze se base sur des multiples de 20.
Base 12	Pour compter les mois, les heures et les œufs par douzaines
Base 10	Celle que nous utilisons tous les jours.
Base 2	Incontournable en informatique. Sans elle ce cours n'aurait pas lieu.
Base 16	Ressemble fort au binaire = notation plus concise pour nous « humain »
Base 8	Cette base, l'octal, était plus en vogue aux débuts de la micro-informatique

2 Ecriture des nombres entiers ou Numération de position

Considérons le nombre 1975. Ce nombre est un "mot" dont les caractères sont les chiffres. Dans ce nombre de quatre chiffres, le dernier représente les unités, l'avant-dernier les dizaines, le précédent : les centaines puis viennent les milliers.
C'est une numération en base 10.

Les romains employaient aussi cette base mais écrivait leurs nombres différemment :
Voici comment on écrit 1975 en chiffres romains : MCMLXXV
Cette écriture, plus compliquée, est encore utilisée dans certaines circonstances mais se prête difficilement aux calculs écrits. Essayez donc de faire par écrit MMV moins MCMLXXV !
Pour les romains, mille, cent, dix et un ne pouvaient que s'écrire avec des signes différents car ils ne connaissaient pas la numération de position. Ils n'avaient pas encore découvert le chiffre zéro qui leur aurait permis d'utiliser une numération de position bien plus efficace.

2.1 Numération de position

Revenons au nombre 1975 (écrit de manière habituelle cette fois)

La valeur que l'on attribue à chaque chiffre dépend du chiffre en lui-même et de sa position.

- Le chiffre 5 vaut 5 x 1.
- Le chiffre 7 représente des dizaines, il vaut 7 x 10.
- Le chiffre 9 qui suit représente des centaines, il vaut 9 x 100.
- Le chiffre 1 vaut 1 x 1000.

Nous formons donc les nombres à l'aide d'une notation où la position est très importante.
Le chiffre le plus à droite représente des unités, celui directement à gauche, les dizaines, etc.
La position que le chiffre occupe dans le nombre est donc à considérer à partir de la droite. Nous numéroterons donc ces positions en allant de droite à gauche. Ainsi le chiffre de droite aura toujours le même numéro quelle que soit la taille du nombre.
Cette numérotation commencera par le numéro 0 pour le premier chiffre (à droite donc)

3	2	1	0
1	9	7	5

La règle qui permet de déterminer le poids d'un chiffre est la suivante :

$$\text{Poids d'un chiffre} = \text{base}^{\text{position}}$$

Voici ce que cela donne dans notre exemple :
Le poids du chiffre 5 est 10^0 , sa valeur est 5 x 1 (car $10^0 = 1$)
Le poids du chiffre 7 est 10^1 , sa valeur est 7 x 10 = 70
Le poids du chiffre 9 est 10^2 , sa valeur est $9 \times 10^2 = 9 \times 100 = 900$
Le poids du chiffre 1 est 10^3 , sa valeur est $1 \times 10^3 = 1 \times 1000 = 1000$

Positions	3	2	1	0
Chiffres décimaux	**1**	**9**	**7**	**5**
Valeurs de chaque chiffre	1×10^3 1000	9×10^2 900	7×10^1 70	5×10^0 5

$$1975 = 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$$

En termes plus mathématiques, on peut dire que la valeur d'un nombre **N** représenté par ***n*** chiffres en base **B** est la valeur numérique d'un polynôme du n-1ième degré où **B** est la base et dont les coefficients sont entiers et inférieurs à ***B***

$$N = c_{n-1} B^{n-1} + \dots + c_i B^i + \dots + c_2 B^2 + c_1 B + c_0$$

$$= \sum_{i=0}^{i=n-1} c_i 2^i$$

Ici en base 10, $x = 10$ et les coefficients $c_{n-1}, c_{n-2}, \dots c_i, \dots, c_1, c_0$ ont tous une valeur inférieure à 10. La suite de ces coefficients $c_{n-1}\ c_{n-2} \dots c_1\ c_0$ n'est autre que la suite des chiffres qui forment le nombre.

Nous utilisons désormais uniquement des numérations de position quelle que soit la base de numération.

EXERCICES SUR LES NOMBRES ENTIERS EN BASE 10

1. *Lorsqu'on écrit un zéro à droite d'un nombre entier, de combien de fois sa valeur augmente-t-elle ?*
2. *Combien y a-t-il de nombres entiers de deux, trois, quatre … chiffres ?*
    a. *Si on écrit les zéros non significatifs*
    b. *Si on n'écrit pas les zéros non significatifs*
3. *Quel est le plus grand nombre entier que l'on puisse écrire avec quatre chiffres ?*
4. *Quel est le plus petit nombre entier que l'on puisse écrire avec par quatre chiffres significatifs ?*
5. *De combien le plus petit nombre entier de trois chiffres dépasse-t-il le plus grand nombre de deux chiffres ?*
6. *a) Quel est le plus petit nombre entier écrit avec cinq chiffres significatifs différents ?*
   b) Quel est le plus grand nombre entier écrits avec cinq chiffres significatifs différents ?
7. *Avec les chiffres 1, 2 et 3 former le maximum de nombres différents où chaque chiffre n'apparaît qu'une seule fois. Classer ces nombres par ordre croissant.*
8. *Même question avec les chiffres 1, 2, 3 et 4*
9. *Un livre possède 1000 pages, combien de fois a-t-on employé le caractère 0 pour numéroter ces pages ?*
10. *On écrit la suite naturelle des nombres, quel est le 33^{ieme} chiffre écrit*

2.1.1 Numération binaire

En binaire la base est 2. Nous n'utilisons que deux chiffres : 0 et 1.
Remarquez qu'en base 2, le chiffre 2 n'existe pas ; tout comme le chiffre 10 n'existe pas en base 10.
Il s'agit toujours d'une numération de position.
De droite à gauche nous avons donc les unités et ce que nous pourrions appeler les "*deuzaines*", les "*quatraines*", les huitaines, les seizaines, les "*trente-deuzaines*" etc. Et tant pis si ce n'est pas français !

Exemple : que vaut le nombre binaire 10110 ?
Le poids d'un chiffre dépend de sa position et de la base
Poids = base ^{position} ici en binaire le poids = 2 ^{position}

Positions	4	3	2	1	0
Chiffres binaires	**1**	**0**	**1**	**1**	**0**
Valeurs de chaque chiffre	1×2^4 16	0×2^3 0	1×2^2 4	1×2^1 2	0×2^0 0

On a donc ici une seizaine, une "*quatraine*" et une "*deuzaine*" soit 16 + 4 + 2 = 22

Un peu de vocabulaire : Bit, Byte, Octet, ... et autres Mots

Les codes binaires sont incontournables en informatique car l'information la plus élémentaire y est le bit (*Binary digit* – chiffre binaire)

Les mots de 8 ou de 16 bits écrits en binaire sont plus lisibles si on les inscrit en laissant un espace entre les groupes de quatre bits comme ceci : 0100 0001

Un groupe de 4 bits est parfois appelé "*Quartet*" ou "*nibble*" mais ces termes sont peu utilisés.

On a avantage à représenter les zéros non significatifs pour montrer la taille des codes transcrits. Remarquez que ces 0 à gauche ne sont d'ailleurs pas toujours "non significatifs". En effet, les codes binaires ne représentent pas toujours des valeurs numériques. Ce sont parfois simplement des codes qui ne représentent pas des quantités et qui n'ont pas non plus de valeur ordinale. Inutile donc de faire de l'arithmétique avec ces codes. Dans ce cas cela n'a aucun sens de vouloir les convertir en décimal et ce serait une erreur d'omettre l'écriture des zéros à gauche.

> En termes plus mathématiques, on pourrait dire que les bits nécessaires pour écrire la valeur N proviennent de la série des coefficients du polynôme suivant :
>
> $$N = b_{n-1}\, 2^{n-1} + \ldots + b_i\, 2^i + \ldots + b_2\, 2^2 + b_1\, 2 + b_0$$
>
> $$= \sum_{i=0}^{i=n-1} b_i\, 2^i$$
>
> Les coefficients b_{n-1} ... b_i,... b_2 , b_1 et b_0 valent chacun 0 ou 1.
>
> C'est la série de bits pour écrire N en binaire.

2.1.2 Numération hexadécimale

La base est 16. Il nous faut donc 16 chiffres, nous avons déjà les chiffres 0 à 9, ajoutons-y les caractères A, B, C, E et F pour représenter les "chiffres" allant de 10 à 15.

Remarquez qu'en base 16, le chiffre 16 n'existe pas ; tout comme le chiffre 10 n'existe pas en décimal ni le chiffre 2 en binaire.

Nous appliquons toujours les mêmes principes de la numération de position.

Le poids d'un chiffre dépend de sa position et de la base

Poids = base ^{position} ici en hexadécimal le poids = 16 ^{position}

De droite à gauche nous avons donc les unités puis les "seizaines", les "256^{zaines}" etc.

Exemple : que vaut le nombre hexadécimal 1A2F ?

Positions	3	2	1	0
Chiffres hexadécimaux	**1**	**A**	**2**	**F**
Valeurs de chaque chiffre	1 x 16^3 4096	10 x 16^2 2560	2 x 16^1 32	F x 16^0 15

L'addition des valeurs de ces 4 chiffres donne : 4096 + 2560 + 32 + 15 = 6703

$$N = c_n\,16^n + \ldots + c_i\,16^i + \ldots + c_2\,16^2 + c_1\,16 + c_0$$

$$= \sum_{i=0}^{i=n-1} c_i\,16^i$$

2.2 Calcul de la valeur d'un nombre quelle que soit la base

Appliquez le principe de la numération de position. Chaque chiffe dans le nombre à évaluer à une valeur qui dépend de sa position : la valeur propre du chiffre multipliée par la base exposant la position. Additionner les valeurs obtenues pour chaque chiffre.

Exemples :

$1011011_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$= 64 + 0 + 16 + 8 + 0 + 2 + 1$
$= 91$

$175_8 = 1 \times 8^2 + 7 \times 8^1 + 5 \times 8^0$
$= 1 \times 64 + 7 \times 8 + 5 \times 1$
$= 125$

$7D_{16} = 7 \times 16^1 + 13 \times 16^0$
$= 7 \times 16 + 13 \times 1$
$= 112 + 13$
$= 125$

2.3 Transcriptions Binaires / Hexadécimale

Les codes hexadécimaux sont bien pratiques en informatique. Ils représentent les codes binaires de manière compacte et nous évitent de devoir lire de longues enfilades de 0 et de 1 qui conviennent mieux aux ordinateurs qu'aux humains.

Un groupe de quatre bits permet de former 16 combinaisons différentes. On peut faire correspondre un chiffre hexadécimal à chacune de ces combinaisons. L'hexadécimal est en quelque sorte du binaire condensé.

Le code hexadécimal 1A2F est bien plus lisible que 0001 1010 0010 1111 en binaire

2.3.1 Comptons en binaire et en hexadécimal

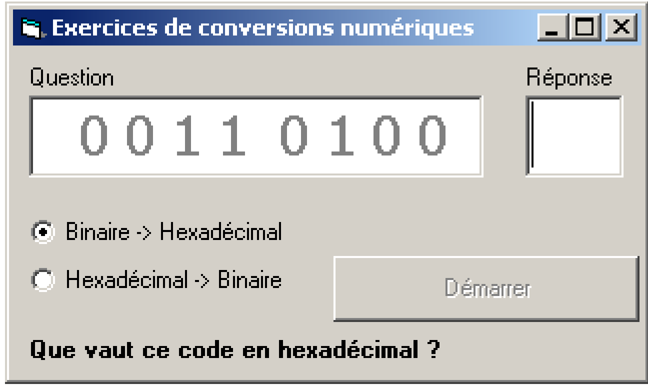
Sachez compter jusqu'à 16 en binaire et en hexadécimal et vous pourrez transcrire rapidement n'importe quel nombre en passant d'une base à l'autre.

Apprenez pour cela à reproduire le tableau ci-contre :

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

Exercez-vous avec la petite application représentées ci-dessous et que vous pouvez télécharger à l'adresse suivante :

http://courstechinfo.be/Bases/SysInfo.html#Hexa



2.3.2 Conversions binaire ↔ octal ou binaire ↔ hexadécimal

Octal ≈ binaire où l'on regroupe les bits par groupe de 3 (en commençant par la droite)

Hexa ≈ binaire quand on regroupe les bits par 4.

(Transition vers paragraphe suivant : Calculer l'épaisseur d'une feuille de papier à cigarette pliée 42 fois.)

2.4 Nombres de codes possibles avec N chiffres en base B

Partons de l'exemple des nombres décimaux

1 chiffre ⇒ 10 codes différents
2 chiffres ⇒ 100 codes car pour chaque dizaine on peut associer 10 codes pour les unités
3 chiffres ⇒ 1000 codes (de 000 à 999)
…
n chiffres ⇒ 10^n codes

En hexadécimal

1 chiffre ⇒ 16 codes (de 0 à F)
2 chiffres ⇒ 16 x 16 codes = 256 (00 à FF)
…
n chiffres ⇒ 16^n codes possibles

En binaire

1 chiffre binaire = 1 **bi**nary dig**it** = 1 **bit**

1 chiffre ⇒ 2 valeurs possibles 0 et1
2 chiffres ⇒ 4 combinaisons possibles
…
n bits ⇒ 2^n codes possibles

Tailles des nombres entiers :

1 byte ou un octet = 8 bits ⇒ 256 codes possibles
Mot de 2 bytes = 16 bits ⇒ 2^{16} = 65536 codes possibles
parfois appelé *word*, *short* ou *integer*
Mot de 4 bytes = 32 bits ⇒ 2^{32} = 4 milliards de codes
souvent appelé *long*
= taille d'un registre à l'heure des Pentiums

Conclusion

Nombre de codes possibles avec N chiffres en base B = $\mathbf{B^N}$

2.5 Préfixes pour représenter les puissances de 10^3

Nous sommes amenés en informatique à devoir chiffrer des grandeurs très grandes et d'autres très petites. La pratique du système métrique nous a habitués à exprimer ces nombres à l'aide de multiples de 10 et même souvent de 1000. Cela correspond à notre habitude de regrouper les chiffres par trois comme dans 1 000 ou 1 000 000 = 10^3 et 10^6

Pour les grands nombres, les puissances successives de 10^3 portent ces noms :

Kilo	$1\ k = 10^{3}$
Méga	$1\ M = 10^{6}$
Giga	$1\ G = 10^{9}$
Tera	$1\ T = 10^{12}$
Peta	$1\ P = 10^{15}$
Exa	$1\ E = 10^{18}$

Les petits nombres s'expriment au moyen des puissances de 10^{-3} :

milli	$1\ m = 10^{-3}$
micro	$1\ \mu = 10^{-6}$
nano	$1\ n = 10^{-9}$
pico	$1\ p = 10^{-12}$
femto	$1\ f = 10^{-15}$

Exercices :

1° Combien il y a-t-il de µs dans une ms ?

2° Les physiciens utilisent l'Angström comme unité pour mesurer les très petites dimensions. C'est le cas par exemple pour les dimensions des atomes. Un Angstrom est un dix-milliardième de mètre Comment peut-on écrire cette distance en ne se référant qu'aux unités vues ci-dessus ?

2.6 Pour les informaticiens, 1 kilo est-ce 1000 ou 1024 ?

Nous savons que $2^{10} = 1024$.

Ce nombre proche de 1000 est souvent désigné par le préfixe "kilo".

- Quand il s'agit de dimension de mémoires, on parle de KB (kilo bytes) ou de Ko (kilo octet) pour dénombrer des multiples de 1024 bytes. De même 1 MB ou 1 Mo = 1024x1024 bytes quand on parle de tailles de mémoire car le nombre de cellules mémoire dans n composant est toujours une puissance de 2 et donc un multiple de 2^{10} ou 2^{20}.
- Dans les autres cas, quand les kilos, les mégas et autres gigas ne concernent pas la mémoire tous ces préfixes représentent des multiples de 1000. 1 kHz = 1000 Hz
  20 Go sur un disque = 20 milliards d'octets et non pas 20x1024x1024x1024.

Notez que le préfixe kilo s'écrit toujours avec un k minuscule quand sa valeur est 1000.

Ex. 1 kHz. 1k = 1000

Les informaticiens écrivent souvent ce préfixe avec une lettre majuscule pour désigner la valeur 1024.

Ex. 1Ko. 1K = 1024

2.7 Calculs approximatifs de 2^n avec n > 10

$$\boxed{2^{10} \approx 10^3} \qquad \text{car} \qquad 1024 \approx 1000$$

Il est facile de calculer approximativement et mentalement ce que vaut 2^n dès que n > 10.

Exemple que vaut 2^{24} ?

$$2^{24} = 2^4 \times 2^{10} \times 2^{10} = 16 \times 1024 \times 1024 = 16\ \text{M} \approx 16.000.000$$

Conclusions : Puisque on a directement

$2^{10} \approx 10^3$
$2^{20} \approx 10^6$
$2^{30} \approx 10^9$
$2^{40} \approx 10^{12}$
etc.

EXERCICES

1° Calculer

2^{12}, 2^{32}, 2^{16}, 2^{27}, 2^{36}

2° Les premiers PC, les PC/XT, avaient un bus d'adressage formé de 20 lignes d'adresse. Ces lignes ne pouvant véhiculer que des codes binaires 0 ou 1 pour former les adresses des cellules mémoire. Quelle était la taille maximum de la mémoire pour ces PC ?

3° On aura bientôt utilisé tous les codes de numéro d'immatriculation composés de 3 lettres suivies de 3 chiffres pour les plaques belges. Combien de nouveaux codes d'immatriculation pourra-t-on faire en plaçant cette fois d'abord 3 chiffres puis 3 lettres ?

EXERCICES RÉCAPITULATIFS

$C0_{16}$ =

$1010\ 0010_{2}$ =

Complétez le tableau :

Binaire	Hexadécimal	Décimal
1 0000 0000		
	50	
		10
		20
		40
		80

Octal	Binaire	Hexa
12		
5655		
244371		

Quelques fractions de secondes :

1 s = ... ms — 0,018 µs = ... ns — 0,01 ns = ... ps

3600 ps = ... ns — 1 µs = ... ms — 0,05 s = ... ms

Base 10 → Base B

3 Conversion d'un nombre N entier en une base B quelconque

RAPPEL :

Nous avons vu au paragraphe 2.2 comment convertir un nombre de base quelconque en base 10. Il suffit pour ce faire d'avoir compris le principe de la numération de position. Chaque chiffre à une valeur qui dépend du chiffre lui-même et de sa position. On obtient la valeur d'un nombre en additionnant les valeurs des chiffres qui le composent.

Une autre manière d'exprimer la même chose est de dire qu'en lisant un nombre de droite à gauche on rencontre les puissances successives de la base :

- unités, deuzaines, quatraines, huitaines, seizaines etc. pour le binaire (base 2)
- unités, huitaines, soixante quatraines etc. pour l'octal (base 8)
- unités, dizaines, centaines, milliers, etc. pour le décimal (base 10)
- unités, seizaines, deux cent cinquante seizaines, etc. pour l'hexadécimal (base 16)

Voyons à présent comment coder dans une base B quelconque un nombre N dont on connaît la valeur, c'est à dire son écriture en base 10. Il faut pour cela dénombrer les puissances successives de la base :

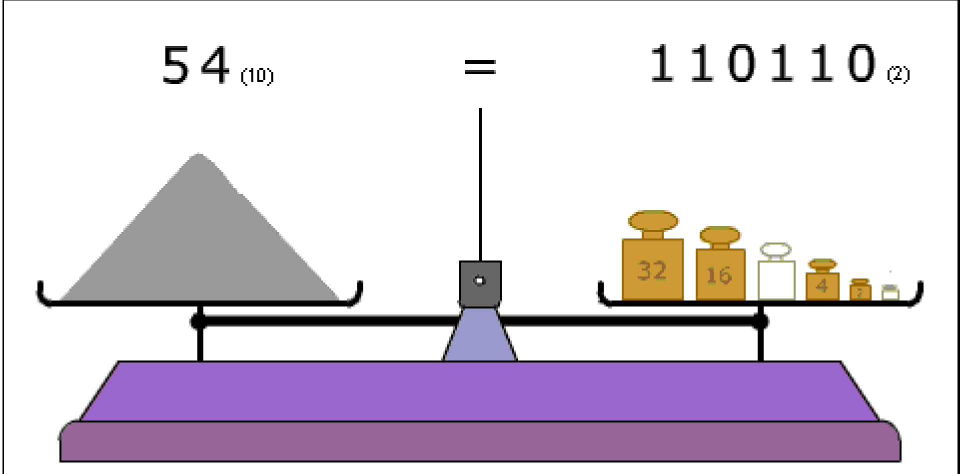
- le nombre d'unités, de deuzaines, de quatraines etc. pour convertir en binaire
- le nombre d'unités, de seizaines, etc. pour convertir en base 16

Il y a pour ce faire deux méthodes : de "gauche à droite" et de "droite à gauche"

Méthode intuitive : de gauche à droite

- Quelle est la plus grande puissance *p* de la base B que l'on puisse retrouver dans N et combien de fois retrouve-t-on B^p dans N? Cela donne le premier chiffre à gauche, en position *p*.
  Exemple : Soit à convertir $420_{(10)}$ en base 16.
  $420_{(10)} > 16^2$, $16^2=256$ va <u>une fois</u> dans 420 ⇒ premier chiffre = 1.en position *2*
  Reste à représenter $420 - 1x16^2 = 164$ unités
- On répète la même question tant que le reste est supérieur à la base.
  $164_{(10)} > 16^1$, 16^1 va 10 fois dans 164 ⇒ chiffre suivant = $A_{(16)}$ en position *1*
  Reste 164 – 10x16 = 4 unités
- Le nombre d'unité qui reste inférieur à B est le chiffre le plus à droite en position *0*.

Vous trouverez à l'adresse http://www.courstechinfo.be/MathInfo/PoidsDesBits.html un exercice pour vous habituer aux "poids des bits" pour les représentations en binaire.



Méthode systématique : de droite à gauche

Le nombre d'unités est le reste de la division de N par la base. Ce qui donne le chiffre en position 0 et dont le poids est B^0.
En divisant à nouveau le quotient de la division précédente par la base on obtient le chiffre de position 1 dont le poids est B^1
Des divisions répétées par la base donnent successivement les chiffres de poids B^0, B^1 B^2, B^3, B^4 etc. ce qui nous permet d'écrire le nombre de droite à gauche.

Exemples : Convertir **1830_{10}** en binaire ⇒ divisions successives par 2

1830 : 2	= 915	reste **0**	$0 * 2^0$ = zéro unité
915 : 2	= 457	reste **1**	1×2^1
457 : 2	= 228	reste **1**	1×2^2
228 : 2	= 114	reste **0**	0×2^3
114 : 2	= 57	reste **0**	0×2^4
57 : 2	= 28	reste **1**	1×2^5
28 : 2	= 14	reste **0**	0×2^6
14 : 2	= 7	reste **0**	0×2^7
7 : 2	= 3	reste **1**	1×2^8
3 : 2	= 1	reste **1**	1×2^9
1 : 2	= 0	reste **1**	1×2^{10}

0 C'est fini, il ne reste plus rien à diviser

Le résultat est : **$111\ 0010\ 0110_2$**

Convertir **1830** en hexadécimal ⇒ divisions successives par 16

1830 : 16	= 114	reste **6**	$0 * 16^0$ = zéro unité
114 : 16	= 7	reste **2**	1×16^1
7 : 16	= 0	reste **7**	1×16^2

0 C'est fini, il ne reste plus rien à diviser

Résultat : **726_{16}** (ce qui concorde bien avec le code $111\ 0010\ 0110_2$ trouvé en binaire)

Conclusions :

Ce procédé fonctionne pour toutes les bases mais en informatiques seuls nous concernent le binaire et l'hexadécimal, parfois mais plus rarement l'octal (base 8). La conversion en binaire est la plus facile, le reste vaut 0 pour les nombres pairs et 1 pour les nombres impairs.
On a donc avantage à convertir d'abord en binaire. Le passage en hexadécimal comme nous l'avons vu au début du cours n'est plus alors qu'un jeu d'enfant.

4 Autre méthode pour convertir d'une base B en base 10 « Méthode de Horner »

Nous avons vu au paragraphe 2.2 comment calculer la valeur d'un nombre quelle que soit la base utilisée pour le représenter. Nous additionnions les valeurs obtenues en calculant les valeurs de chaque chiffre compte tenu de leurs positions dans le nombre.

La méthode qui suit donne le même résultat, ceux qui aiment suivre des "recettes de cuisines" la trouveront plus systématique.

Montrons comment cela marche pour le binaire mais la méthode est valable quelle que soit la base.

Voici l'algorithme :

```
Lire la valeur du chiffre à gauche
Répéter tant qu'il reste des chiffres à droite
{
        Multiplier par la base
        Ajouter le chiffre suivant
}
```

Exemples :

1101_2 = ((((**1** x 2) + **1**) x 2 + **0**) x 2 +**1**

1	**1**	**0**	**1**
	2	6	12
	3	6	13

123_8 = (((**1** x 8) + **2**) x 8) + **3**

1	2	3
	8	80
	10	83

$20C_{16}$ = (((**2** x 16) + **0**) x 16) + **12**

2	**0**	**12**
	32	64
	32	76

<u>EXERCICES</u>

Recherchercher par la méthode de Hornner :

203_8 = ... 101010_2 = ... $20A_{16}$ = ...

Méthode au choix :

166_{10} = ... $_{16}$

100_{10} = ... $_{2}$

100_{10} = ... $_{16}$

1023_{10} = ... $_{16}$

1023_{10} = ... $_{2}$

$COCA_{16}$ = ... $_{10}$

1011011_2 = ... $_{10}$

236_8 = ... $_{10}$

FFF_{16} = ... $_{10}$

5 Nombres binaires négatifs

Nous avons jusqu'à présent parlé de nombres entiers positifs ou nuls.

En décimal,

1, 2, 3 etc. sont des nombres positifs, ils sont supérieurs à 0 (>0)
-1, -2, -3 etc. sont des nombres négatifs, ils sont inférieurs à 0 (< 0)

De même en binaire,

1, 10, 11, 100, 101 etc. sont des nombres binaires positifs,
-1, -10, -11, -100, -101 etc. sont des nombres binaires négatifs.

Le problème est que les circuits électroniques digitaux ne peuvent enregistrer que des 0 ou des 1 mais pas de signes + ou -. Le seul moyen est alors de convenir que si un nombre est susceptible d'être négatif on lui réserver un bit pour indiquer le signe. Reste à déterminer le bit qui dans un nombre binaire conviendrait le mieux pour symboliser le signe et quel caractère (0 ou 1) conviendrait le mieux pour représenter le signe "plus" ou le signe "moins".

Sur papier, les nombres ont des dimensions variables : en additionnant deux nombres de deux chiffres on obtient un nombre de deux ou trois chiffres, en multipliant deux nombres de deux chiffres ont obtient des nombres de 3 ou 4 chiffres.

En machine par contre, les nombres ne sont pas extensibles. Ils ont des dimensions fixes. C'est exactement ce que nous avons sur le compteur kilométrique d'une voiture. S'il est de six chiffres, le kilométrage maximum qu'il pourra indiquer est 999.999 km. C'est généralement suffisant. De même, dans les ordinateurs les nombres (binaires) ont aussi des dimensions fixes de 1, 2, 4 ou 8 octets.

Revenons à l'exemple de la voiture et imaginez un compteur kilométrique qui compte les km en marche avant et qui les décompte en marche arrière. Que pourrait-on lire sur un compteur d'une voiture neuve (compteur initialement à 000.000) si elle parcourt 1 km en marche arrière ? Le compteur décompte 1 km et affiche donc … 999.999 km ! Ce code correspond parfaitement à la valeur –1 puisqu'on obtient 0 si on lui ajoute à nouveau 1.

$x + 1 = 0 \quad \Rightarrow \quad x = -1 \quad \Rightarrow$ dans ce cas ci 999.999 équivaut à -1

Cette caractéristique étrange est due au fait que ce nombre à une dimension finie (6 chiffres décimaux)

De même, quel serait le code d'un nombre de 8 bits pour représenter la valeur –1 ?
Le code $1111\ 1111_2$ (= FF_{16}) convient puisque, si on ajoute 1 à ce nombre, on obtient 00000000_2 (ou 00_{16}), le bit de report déborde à gauche de l'espace qui est réservé au nombre, ce bit est ignoré.

Le bit le plus à gauche du mot binaire est à 1, c'est ce bit qui va représenter le signe.
Le tableau de la page suivante montre ce que cela donne avec des nombres de 8 bits

Si on admet que le nombre peut représenter des valeurs négatives, on parle de nombres "signés".

Comme pour les nombres "non signés", on peut représenter $2^8 = 256$ codes avec 8 bits
mais ici le bit de gauche est le signe
1 = signe moins
0 = signe plus

Il y a donc moyen de représenter

→ 128 codes avec le bit de signe à 1
ce sont 128 nombres négatifs
(de –1 à – 128)

→ 128 codes avec le bit de signe à 0
le nombre 0 et 127 nombres positifs
(de 1 à +127)

Nombres de 8 bits		Valeur en base 10
Lu en hexadécimal	Lu en binaire	
7F	0111 1111	127
7E	0111 1110	126
…	…	…
10	0001 0000	16
0F	0000 1111	15
OE	0000 1110	14
OD	0000 1101	13
OC	0000 1100	12
0B	0000 1011	11
0A	0000 1010	10
09	0000 1001	9
08	0000 1000	8
…		
03	0000 0011	3
02	0000 0010	2
01	0000 0001	1
00	0000 0000	0
FF	1111 1111	-1
FE	1111 1110	-2
FD	1111 1101	-3
FC	1111 1100	-4
FB	1111 1011	-5
FA	1111 1010	-6
F9	1111 1001	-7
…	…	…
86	1000 0110	-122
85	1000 0101	-123
84	1000 0100	-124
83	1000 0011	-125
82	1000 0010	-126
81	1000 0001	-127
80	1000 0000	-128

5.1 Comment calculer les codes des nombres négatifs ?

Le calcul se fait en deux étapes :

1° Calcul du complément à 1 = Remplacer tous les 0 par des 1 et tous les 1 par des 0.

2° Calcul du complément à 2 = Ajouter 1 au complément à 1

Exemple : comment écrire –4 en binaire ou en hexadécimal ?

```
+4                 =     0000 0100
complément à 1     =     1111 1011 ↘
                                     +1
complément à 2     =     1111 1100 ↙     =  FC  =  - 4
```

Cas particuliers :

- Le complément à 2 de 0 est encore 0
- Le complément à 2 de 80H est aussi 80H ! Les nombres négatifs et positifs ne sont pas répartis symétriquement. Avec un byte la valeur minimum est –128 contre +127 pour la valeur positive.

NB. Le complément à 1 est aussi appelé "*complément logique*" ou "*complément restreint*" De même, certains désignent le complément à 2 par l'expression "*complément arithmétique*".

Analogie en décimal

```
 25
-17
---
 08
```

Existe-t-il un complément arithmétique de 17 tel que 25 + Complément de 17 = 8 ?

Oui, à condition de décréter que comme dans une machine les nombres ont une taille fixe au-delà de laquelle les reports sont ignorés.

Puisque deux chiffres suffisent pour écrire 25, 17 et 08 nous limitons la taille de ces nombres à 2 caractères.
La question devient : Quel nombre faut-il ajouter à 25 pour que la réponse se terminer par les chiffres 08 ?
Ce nombre est 83. En effet **25 + 83 = 108** mais on ignore le 1 à gauche puisque nous avons décidé de donner une taille fixe de deux chiffres pour les nombres de cet exemple.

83 est donc dans ce cas le complément arithmétique de 17.

Comment trouver ce complément arithmétique en base 10 ?
La méthode ressemble fort au calcul du complément à 1 comme en binaire suivi de l'addition d'une unité. Ici, en décimal, le complément restreint sera un complément à 9.
Complément à 9 :

```
 99
-17
---
 82
```

Complément arithmétique : 82 +1 = 83

99 – 17 + 1 = 100 – 17 = 83

5.2 La valeur du bit de signe

Le bit de signe est le bit le plus significatif du code (MSB *Most Significant Bit*), celui qui est le plus à gauche. Dans le cas d'un nombre de n bits numérotés de 0 à n-1, c'est le bit n-1 . Bien souvent on se contente de constater que ce bit est à 1 pour en conclure que le nombre considéré est négatif. La valeur absolue de ce nombre est alors déterminée en calculant le complément arithmétique de son code.
Une autre manière d'envisager la chose serait de considérer que le bit n-1 a, contrairement aux autres bits, une valeur négative : -2^{n-1}

Exemple :

Si un byte est considéré comme un code signé le bit 7 quand il est à 1 vaut -128.
Si le byte est considéré comme non signé, le poids du bit 7 est simplement $2^7 = 128$.
Ainsi -123 = -128 + 5 = 80H + 5 = 85H

Plus généralement :

- Pour les nombres non signés nous calculions la valeur du nombre comme suit :

$$N = b_{n-1}\,2^{n-1} + \ldots + b_i\,2^i + \ldots + b_2\,2^2 + b_1\,2 + b_0$$

$$= \sum_{i=0}^{i=n-1} b_i\,2^i$$

- Dans le cas des nombres signés la valeur sera

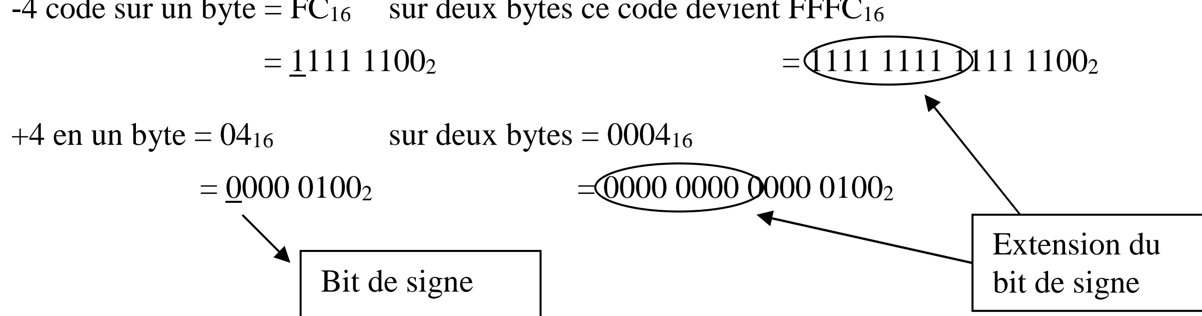
$$N = -\,b_{n-1}\,2^{n-1} + b_{n-2}\,2^{n-2} + \ldots + b_i\,2^i + \ldots + b_2\,2^2 + b_1\,2 + b_0$$

$$= -2^{n-1} + \sum_{i=0}^{i=n-2} b_i\,2^i$$

5.3 Conversions entre mots de différentes longueurs

Pour étendre la taille d'un nombre non signé, on ajoute des 0 à sa gauche.
Pour étendre la taille d'un nombre signé, ajoute sur la gauche des bits identiques au bit de signe.

Exemples :



EXERCICES

1. Déterminez les valeurs des compléments logiques et arithmétiques des codes binaires suivants :

	Complément à 1	Complément à 2
1100 1001		
0000 1111		
0111 0011 0001 0000		

2. Calculer les compléments à 1 et à 2 pour les nombres suivants exprimés sous forme hexadécimale. Faites le calcul en binaire puis notez la réponse en hexa.

   $AA_{(16)}$

   $FF_{(16)}$

   $1248_{(16)}$

3. Que vaut le code $C0_{(16)}$

   a) s'il s'agit d'un nombre non signé ?

   b) s'il s'agit d'un nombre signé ?

4. Les codes suivants ont une taille de 16 bits, ils sont signés et donnés en hexadécimal. Calculez leurs valeurs et donnez la réponse en décimal.

   FFFF

   8000

   7FFF

   00FF

5. Quelles sont les valeurs minimum et maximum que peut prendre un nombre entier signé codé sur 4 octets ?

6. Comment écrire -512 en binaire ? Combien faut-il de bytes au minimum pour encoder cette valeur ?

7. Quel est le plus petit nombre entier négatif qui puisse être traité dans les registres d'un Pentium 64 bits ?

8. La valeur -192 peut-elle être codée sur un byte ? justifiez votre réponse.

9. Comment écrire -150 en binaire et en hexadécimal ?

10. Que vaut $8001_{(16)}$ selon que ce code de 2 octets est signé ou non signé ?

6 Opérations arithmétiques en binaires

Nous nous limitons dans ce chapitre au cas des nombres entiers.

Aussi étonnant que cela puisse paraître, il faut bien l'avouer, ce n'est qu'à de très rares occasions qu'un informaticien est amené à faire par écrit des calculs en binaire ou en hexadécimal. On dispose bien souvent d'une machine, une calculatrice ou un ordinateur, pour réaliser de telles opérations. Il est cependant important de connaître les mécanismes de ces opérations pour saisir comment ces machines fonctionnent, tout comme nous nous sommes attachés à comprendre les méthodes des changements de bases ou du codage des nombres signés.

Les calculs en binaire (ou en hexadécimal) peuvent toujours se faire exactement de la même manière que ceux que nous faisions à l'école primaire en base 10.

6.1 Addition

« Un plus un » fait deux, c'est un fait indépendant du mode de représentation des nombres.

En binaire, deux s'écrit 10 $1_{(2)} + 1_{(2)} = 10_{(2)}$ ⇒ « 1 + 1 = 2, j'écrit 0 et je reporte 1 »
De même (1+1+1=3) $1_{(2)} + 1_{(2)} + 1_{(2)} = 11_{(2)}$ ⇒ « 1 + 1 + 1 = 3 ; j'écrit 1 et je reporte 1 »

Pour le reste $0 + 0 = 0$, $0 + 1 = 1$ et $1 + 0 = 1$
Ces trois derniers calculs n'engendrent aucun report.

Vous en savez assez maintenant pour additionner par écrit deux nombres de *n* bits.
Il suffit d'aligner convenablement ces deux nombres, l'un au dessus de l'autre, une colonne à droite pour les unités puis successivement vers la gauche les colonnes des deuzaines, des quatraines etc.
Additionnez ensuite les bits en commençant par la droite sans oublier de noter les reports.

Exemple :

```
                                 1   1 1 1
  10110101                      10110101
+   100111          →         +   100111
__________                    __________
                                11011100
```

Remarquez que l'opération convient aussi bien pour les nombres non signés que pour les nombres signés. Une règle est cependant essentielle : convenir pour chaque variable du mode signé ou non signé une fois pour toute et ne jamais plus changer de convention.

6.2 Soustraction

Si vous y tenez, vous pouvez appliquer à nouveau la même méthode qu'à l'école primaire.
Aligner le nombre à soustraire sous le premier nombre puis on effectue la soustraction en commençant par les chiffres à droite. Si le chiffre du dessous est trop important, il faut enregistrer une "retenue" qu'on retranche dans le calcul de la colonne suivante.

Exemple :

```
 15
 35
-17
 1
___
 18
```

On peut appliquer exactement le même procédé en binaire :

```
  1010 10
  100101
-  11011
  1 1  1
________
  001010
```

Sachez toutefois que même les machines ne se donne jamais ce mal.
Plutôt que de faire une soustraction elles additionnent le complément du terme à soustraire.

Plutôt que de soustraire un nombre nous allons ajouter son complément. La méthode n'a de sens que pour des nombres ayant une taille finie. Reprenons l'exemple ci-dessus et fixons la taille des nombres à 8 bits.

Nous additionnerons donc le complément de 00011011
Complément à 1 → 11100100
Complément à 2 → 00011011

```
 1 1 1    1  1
  00100101
+ 11100101
----------
  00001010
```

6.3 Multiplication

Les multiplications écrites se font de la même manière en binaire que en décimal. Il suffit de connaître la table de multiplication par 0 et par 1.

	0	1
0	0	0
1	0	1

```
     11010110
   ×      101
   ----------
     11010110
    00000000
   11010110
  -----------
  10000101110
```

Comme pour toutes les bases, pour multiplier par la base il suffit d'ajouter un zéro à droite du nombre. En binaire cela revient à multiplier le nombre par 2.

6.4 Division

Ici aussi, nous pouvons utiliser la même méthode que lors des calculs écrits en décimal.

En binaire, l'écriture des multiples de 2 se termine par le chiffre 0. Pour diviser par 2, il suffit donc d'enlever le zéro à droite du nombre.

Ex.10 / 2 = 5 $1010_{(2)} / 10_{(2)} = 101_{(2)}$

```
10000101110 | 101
 101        |---------
 ---        | 11010110
  110
  101
  ---
    110
    101
    ---
      111
      101
      ---
       101
       101
       ---
        00
```

Conclusion de ce chapitre :

Mis à part les additions, il est fort rare de faire des opérations arithmétiques en binaire "à la main" comme il est tout aussi rare de faire de l'informatique sans machine. Le but du chapitre était de comprendre comment se font les opérations afin de pouvoir imaginer ce qui se passe dans les machines, et d'être capable de comprendre et d'interpréter les résultats, ce à quoi nous nous attacherons dans le chapitre suivante

7 Opérations arithmétiques au cœur du PC

7.1 Nombre signés ou non ?

Dans les chapitres précédents nous avons appris comment les nombres entiers sont codés en binaire. Nous avons aussi vu comment le processeur traite ces codes pour faire des opérations arithmétiques.
La question qui est souvent posée est : « Comment le processeur sait-il si les nombres binaires sont à considérer comme signés ou comme non signé ? »
Réponse : l'ordinateur n'en sait rien. Peu importe pour lui que les nombres soient signés ou non, de toutes manières cela ne change en rien sa manière de faire les calculs.
Le programmeur par contre doit savoir si les codes qu'il assigne à chacune de ses aux variables entières sont des nombres signés ou non, en fonction de quoi il pourra interpréter le code FF comme -1 ou 255.

7.2 Au cœur du processeur avec DEBUG

DEBUG est un utilitaire de mise au point qui fonctionne en mode invite de commande. Il nous fait voir l'ordinateur comme si nous étions à la place du microprocesseur. DEBUG nous montre et nous laisse modifier le contenu des registres du CPU et celui de la mémoire vive. Les modifications se font en hexadécimal mais il est même possible de lire et d'écrire des instructions en langage assembleur.
Pour plus de renseignement sur DEBUG lisez la demi-douzaine de pages qui se trouve à l'adresse http://courstechinfo.be/OS/Debug.pdf

7.3 Quelques manipulations avec DEBUG

Lancer DEBUG :

```
C:\>DEBUG ↵
-
```

Vous êtes en mode invite de commande, tapez DEBUG puis la touche [Enter]. Vous verrez apparaître un tiret. Ce signe est l'invite par laquelle le programme DEBUG vous signale qu'il est prêt à recevoir une commande. Toutes les commandes se font ensuite en tapant une seule touche suivie d'éventuels paramètres.

Première commande : le point d'interrogation '?'

```
-? ↵
```

DEBUG affiche une aide succincte. On y lit par exemple que pour quitter il faudra utiliser la commande 'q'. NB. DEBUG ignore la casse.

```
-D 100 ↵
```

=> *DUMP de 128 bytes à partir de l'adresse 100*

```
154F:0100  3D 00 80 90 90 00 00 00-00 00 00 00 00 00 00 00   =...............
154F:0110  00 00 00 00 00 00 00 00-00 00 00 00 34 00 3E 15   ............4.>.
154F:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Les adresses sont données en deux parties : une adresse de *segment* et un déplacement par rapport à cette référence appelé *offset*.
L'adresse du premier byte affiché est donc du genre 154F:0100
L'adresse de base est dans ce cas 154F, elle est déterminée par le système d'exploitation en fonction des autres applications déjà chargées en mémoire. Le segment que Windows vous accorde pour vos expérimentations est ici le segment 154F.
L'offset est 0100, c'est bien ce que vous aviez demandé par la commande "D 100"
Chaque ligne affiche 16 bytes en hexadécimal avec sur la droite des caractères qui correspondent aux codes ASCII de ces bytes quand ils sont imprimables.

-F 100 L 80 00 ↵

=> *Remplir à partir de l'adresse 100, 80 bytes avec le code 00*
F pour "*Fill*" = Remplir

Vérifiez ce qui a changé avec la commande Dump : **D 100 ↵**

-E 100 "ABCDEFabcdef1234567890" ↵

=> *Entrer à partir de l'adresse 100 les codes ASCII de la suite de caractères « ABCDEFabcdef1234567890 »*

Vérifiez : **D 100 ↵**

-E 100 40 20 ↵

=> *Cette fois les codes ont été tapés en hexadécimal, octet par octet.*

Vérifier ce que cela donne.

-R ↵

=> *Demander l'affichage des registres*

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=154F  ES=154F  SS=154F  CS=154F  IP=0100   NV UP EI PL NZ NA PO NC
154F:0100 3D0080        CMP     AX,8000
```

Ne considérons que les registres principaux :

DS, ES, SS et CS — sont des registres de segments.
Ils contiennent ici l'adresse du segment que l'OS nous a alloué.

AX, BX, CX et DX — sont des registres "généraux" ils sont destinés à contenir des données.

IP — est l' "*Instruction Register*" parfois aussi appelé "*Program Counter*" ou "compteur ordinal". Il contient en permanence l'adresse de la prochaine instruction et s'incrémente au fur et à mesure que les instructions sont lues pour être exécutées.

Les flags sont des bits indicateurs qui basculent dans un état ou l'autre (0 ou 1) en fonction des opérations arithmétiques ou logiques exécutées par le processeur :

ZR/NZ → le flag ZERO indique si la dernière opération donne un résultat nul.

PL/NG → le flag de SIGNE indique si le résultat est positif ou négatif.

CY/NC → le flag de CARRY passe à 1 si la dernière opération donne lieu à un report.

OV/NV → le flag d'OVERFLOW indique si le dernière opération donne un résultat incohérent pour des nombres sont signés.

Les états de ces flags sont pris en compte par les instructions de branchements conditionnels telles que JZ (jump on zero) ou JNZ (jump on non zero).

7.4 Saisie du programme d'addition

Encodons à présent quelques lignes en assembleur. Nous utilisons pour cela la commande **A** du DEBUG en indiquant à que nous plaçons le programme à l'adresse `100`. Les instructions peuvent être encodées indifféremment en minuscules ou en majuscules.

Les deux premières instructions vont mettre deux valeurs `1234` et `5678` respectivement dans les registres `AX` et `BX`. La troisième instruction indique au microprocesseur qu'il doit ajouter la valeur donnée par `BX` à celle contenue dans `AX`.
NB. Quand l'instruction se réfère à deux opérandes, la première désigne la destination et la seconde représente l'opérande source. Ainsi "`MOV AX,BX`" équivaut à `(AX)←(BX)`

```
-A 100
154F:0100 MOV AX,1234        (AX) <- 1234
154F:0103 MOV BX,5678        (BX) <- 5678
154F:0106 ADD AX,BX          (AX) <- (AX) + (BX)
154F:0108   ↵                Terminer l'encodage par [Enter]
-
```

Vérifions le travail fait à l'aide de la commande de désassemblage **U** *(unassemble)*

```
-u 100
154F:0100 B83412        MOV     AX,1234
154F:0103 BB7856        MOV     BX,5678
154F:0106 01D8          ADD     AX,BX
154F:0108 0000          ADD     [BX+SI],AL
154F:010A 0000          ADD     [BX+SI],AL
```

DEBUG affiche 16 lignes d'instruction mais seules les trois premières nous intéressent. Remarquez comment ces instructions sont transcrites en langage machine.

- La première instruction occupe trois bytes en mémoire. Le code opération (opcode) de cette instruction se trouve à l'adresse `0100`, "`B8`" est le code qui pour le Pentium signifie "*mettre dans AX le double byte suivant*" . L'opérande `1234` (en hexadécimal bien sûr) occupe deux bytes. La partie basse "`34`" se trouve dans le byte d'adresse `0101`, la partie haute "12" se trouve à l'adresse `0102`.
- La seconde instruction commence donc trois bytes plus loin => à l'adresse `103`. Elle ressemble très fort à la première.
- L'instruction "`ADD AX,BX`" occupe deux bytes en langage machine.

7.4.1 Exécution du programme :

Demandez la trace de trois instructions à partir de l'adresse `100`

```
-t =100 3
AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=154F  ES=154F  SS=154F  CS=154F  IP=0103   NV UP EI PL NZ NA PO NC
154F:0103 BB7856        MOV     BX,5678

AX=1234  BX=5678  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=154F  ES=154F  SS=154F  CS=154F  IP=0106   NV UP EI PL NZ NA PO NC
154F:0106 01D8          ADD     AX,BX

AX=68AC  BX=5678  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=154F  ES=154F  SS=154F  CS=154F  IP=0108   NV UP EI PL NZ NA PE NC
154F:0108 0000          ADD     [BX+SI],AL
```

1ière instruction : AX reçoit la valeur `1234` ; IP contient l'adresse `103`, c'est l'adresse de l'instruction suivante.

2ième instruction : BX reçoit la valeur `5678`, IP désigne l'instruction suivante.

3ième instruction : A la valeur contenue dans AX s'ajoute celle du registre BX.
`1234 + 5678 = 68AC`

7.4.2 Vérification de la validité du résultat:

Cette vérification se fait en consultant les indicateurs de report et de dépassement.

- L'indicateur de report (*carry*) concerne uniquement les opérations sur des nombres non signés. Le carry passe à 1 si la somme des deux nombres produit un report qui sort des *n* bits alloués à la réponse, autrement dit si sa valeur dépasse 2^n.
- L'indicateur de dépassement (*overflow*) concerne quant à lui uniquement les nombres signés. Il passe à 1 si pour une somme de deux nombres de même signe, le calcul donne un nombre de signe différent.
  NB. L'addition de deux nombres de signes différents correspond à une soustraction. La valeur absolue du résultat est donc moindre que celle d'au moins un des deux nombres de départ. On peut donc être certain que le résultat ne sortira pas des limites allant de -2^{n-1} à $+2^{n-1}-1$ (-3768 à +32.767 si n = 16 bits)

Dans l'essai que nous avons fait, les deux nombres `1234` et `5678` pouvaient aussi bien être signés que non signés.

- S'ils sont non signés, c'est l'indicateur de report qu'il faut vérifier. Ici l'addition `1234 + 5678` ne donne aucun report, la réponse `67AC` tient parfaitement sur 16 bits. C'est ce que confirme l'indication NC = "*no carry*"
- Dans le cas où les deux nombres additionnés sont signés, `1234` et `5678` sont tous deux positifs puisque le bit de signe = 0. Le résultat est lui aussi positif, c'est ce que confirme l'indication NV = *"no overflow"*

Lors d'une addition, le processeur évalue donc le résultat à la fois pour des nombres signés ou non signés. C'est au programmeur de l'application de savoir si les variables qu'il a déclarées sont signées ou non, en fonction de quoi il vérifie après chaque opération soit le flag de dépassement (nombres signés) soit celui de report (nombres non signés).

7.5 Exemples de calculs

- <u>Exemple 1</u>
  Imaginez les calculs suivants faits sur des nombre de 16 bits (2 octets)

```
   FFFF
+  0001
  10000
```

Carry = 1	il y a un report qui sort des 16 bits
Overflow = 0	car les deux nombres à additionner sont de signes différents.

1° Si ces nombres sont non signés

`FFFF`$_{(16)}$ = $65.535_{(10)}$ `0001`$_{(16)} = 1_{(10)}$ La réponse de la machine est `0000`

Pourtant, la somme 65.535 + 1 = 65.536 et non pas 0 !

L'erreur signalée par l'indicateur de report (*carry*) provient du fait que le résultat dépasse la valeur maximale qui puisse être écrite avec 16 bits : $65.536_{(10)} = 1\ 0000_{(16)}$

2° Si ces nombres sont signés

`FFFF`$_{(16)}$ = -1 `0001`$_{(16)}$ = +1 `0000`$_{(16)} = 0_{(10)}$

La somme (-1) + (+1) = 0 C'est bien le résultat obtenu.

Il n'y a pas d'erreur, c'est ce que confirme l'indicateur *Overflow* = 0, il n'y a pas de dépassement.

- Exemple 2

```
   8012
+  F091
  170A3
```

Carry = 1	il y a un report qui sort des 16 bits
Overflow = 1	nombres à additionner tous deux négatifs mais le résultat est positif !

1° Si ces nombres sont non signés :

8012(16) = 32768(10)   F091(16) = 61585(10)   70A3 (16) = 28.835 (10)

Pourtant, la somme 32.768 + 61.585 = 91.353 et non pas 28.835 !

L'erreur signalée par le bit de report (*carry*) provient du fait que le résultat dépasse la valeur maximale qui puisse être écrite avec 16 bits : 91.353 > 65.536
ou 1 70A3 (16) > 1 0000 (16)

⇒ Pour pouvoir additionner de telles valeurs il faut utiliser des nombres de 32 bits.

2° Si ces nombres sont signés :

8012(16) = −32.750(10)   F091(16) = −3951(10)   70A3(16) = 28.835 (10)

La somme (−32.750) + (−3.951) = -36.737   et non pas 28.835 !

L'erreur signalée par l'indicateur de dépassement (*overflow*) elle est due au fait que -36.737 < 32.768 (-2^{15}) limite inférieure des nombres signés en 16 bits.

⇒ Pour pouvoir additionner de telles valeurs il faut utiliser des nombres de 32 bits.

- Exemple 3

```
   51A4
+  620C
   B3B0
```

Carry = 0	il n'y a pas de report
Overflow = 1	les nombres à additionner sont positifs mais le résultat est négatif.

1° Si ces nombres sont non signés :

51A4(16) = 20.900(10)   620C(16) = 25.100(10)   B3B0 (16) = 46.000 (10)

La somme 20.900 + 25.100 = 46.000

Le résultat est correct, c'est ce que confirme l'indicateur de carry = 0.

Ce calcul était possible en codant ces nombres non signés sur 16 bits. En effet le résultat de la somme ne dépasse pas 2^{16}-1 soit 65.535.

2° Si ces nombres sont signés :

51A4(16) = +20.900   620C(16) = +25.100   B3B0(16) = −19.536 (10)

La somme (+20.900) + (+25.100) = +46.000   et non pas -19.536 !

L'erreur signalée par l'indicateur de dépassement (*overflow*) provient du fait que 46.000 est plus grand que 32.767, le plus grand nombre signé de 16 bits (2^{15}-1)
Pour éviter ce dépassement, ces nombres signés auraient dus être codés sur plus de 16 bits, sur 32 bits par exemple.

□ Exemple 4

```
   241C
+  30A5
   54C1
```

Carry = 0	il n'y a pas de report
Overflow = 0	les nombre à additionner sont positifs le résultat aussi.

1° Si ces nombres sont non signés :

241C(16) = 9.244(10)   30A5(16) = 12.453(10)   54C1 (16) = 21.697 (10)

La somme 9.244 (10) + 12.453(10) = 21.697 (10)

Le résultat est correct, ce que confirme le bit de report (*carry*) qui est à 0.

2° Si ces nombres sont signés :

241C(16) = +9.244(10)   30A5(16) = +12.453(10)   54C1 (16) = +21.697 (10)

La somme (+9.244) + (+12.453) = + 21.697

Le résultat est correct, en effet le bit de dépassement (*overflow*) est à 0

EXERCICES

1. Les nombres signés suivants peuvent-ils être encodés sur un octet seulement ?
   Répondez par oui ou par non puis justifiez votre réponse.

   240 (10)

   100 (10)

   -130 (10)

   255 (10)

2. Les nombres signés suivants peuvent-ils être encodés sur un octet seulement ?
   Si oui écrivez cette valeur (en binaire ou en hexa) avec un seul byte.

16 bits	**8 bits**
0000 0000 0011 1011 (2)	
1111 1111 1011 0110 (2)	
0000 0000 1000 0101 (2)	
1111 1111 0111 0011 (2)	
FF82 (16)	
0426 (16)	
FF6A (16)	
F0C4 (16)	

3. Les nombres suivants sont donnés en hexadécimal. A quelles valeurs décimales ces nombres correspondent-ils selon qu'ils sont signés ou non ?

Code hexadécimal	Valeur en base 10	
	Si les nombres sont signés	S'ils sont non signés
EO		
63		
7F		
80		
81		
40		
D8		
14		
C0		

4. Faites les calculs suivants. (Tous les nombres sont donnés en hexadécimal)
   A votre avis, selon que ces codes représentent des nombres signés ou non, ces calculs sont-il exacts s'ils sont faits sur 8 bits uniquement ?
   Que vaudraient dans chacun des cas les indicateurs de report et de dépassement ?

a) 40 + 14 =

b) 80 + 7F =

c) D8 + C0 =

d) 40 + E0 =

e) C0 + 63 =

f) FF - 20 =

8 Codage des nombres réels

8.1 Utilité de la virgule flottante

- Imaginez un système de mesure digital qui indique des dimensions en millimètres avec une précision de trois chiffres derrière la virgule. Faut-il pour cela que le processeur de ce dispositif sache traiter des nombres en virgule flottante ? Non, des nombres entiers conviennent, il suffit de mesurer les dimensions en micromètres et la précision est la même.
Si vraiment vous tenez aux mesures en mm, il suffit d'afficher ou d'imaginer une virgule en troisième position. 12.345 µm ≡ 12,345 mm

On parle dans ce cas de la notation en « virgule fixe ». Les méthodes de calculs en virgule fixe sont identiques à celles que nous avons vues pour les nombres entiers et sont très rapides.

- Imaginons à présent que notre système doive effectuer des calculs sur les valeurs mesurées. Nous savons que l'étendue des grandeurs représentables par des nombres entiers dépend du nombre de bits utilisés :

8 bits	256
16	65536
32	$4\ 10^9$ (4 x 1024 x 1024 x 1024 exactement)
64 bits	$16\ 10^{18}$ (16×1024^6 exactement)

Lors des calculs, il arrive que des dépassements provoquent des erreurs.

Supposez par exemple que l'on doive calculer 1500 x 2000 / 4000 avec des entiers de 16 bits. Les nombres 1500, 2000 et 4000 peuvent parfaitement être codés en 16 bits puisqu'ils sont tous trois inférieurs à $2^{16} - 1 =$ 65.535
Si cependant la machine commence par calculer 1500 x 2000, le résultat intermédiaire 3.000.000 dépasse 65.353, les bits de poids forts du résultat sont ignorés et la réponse est fausse : $1500_{(10)} \times 2000_{(10)} = 5DC_{(16)} \times 7D0_{(16)} = 2CC6C0_{(16)}$
En ne lisant que les 16 bits cela donne : $C6C0_{(16)} = 50.880_{(10)}$ c'est bien moins qu 3.000.000 !
Si, pour ne pas dépasser cette limite vers le haut, on commence par calculer 2000/4000, la division entière donne 0 ce qui ne vaut guerre mieux !

Le problème vient du fait que les nombres entiers (ou en virgules fixe) ne permettent que la représentation d'une étendue limitée de nombres.

La solution consiste à accroître cette étendue en distinguant d'une part les chiffres significatifs et d'autre part les ordres de grandeur comme nous le faisons en notation scientifique. Le calcul $1,5 \cdot 10^3 \times 2 \cdot 10^3$ se fait dès lors en deux étapes, le calcul des chiffres significatifs d'une part 1,5 x 2 = 6 et les ordres de grandeurs $10^3 \times 10^3 = 10^6$ d'autre part. Ces calculs 1,5 x 2 et 3+3 (la somme des exposants de 10) portent sur de petits nombres.

La représentation, dite en « virgule flottante » permet une plus grande disparité des ordres de grandeur qu'en virgule fixe. On peut grâce aux « nombres flottants » écrire à la fois des nombres très grands (suivis d'un grand nombre de zéros) et des fractions très petites (précédés d'un grand nombre de zéro).

C'est en quelque sorte une technique de compression de l'écriture des nombres : au lieu d'écrire tous les 0 on en indique simplement leur nombre. C'est par rapport aux nombres entiers, une compression où l'on ignore les détails. Ainsi, ce n'est pas parce qu'il est permis d'écrire 10^{38} que le système est capable de compter 1 à 1 depuis 0 jusque 10^{38}

Rappelons en guise de préparation comment nous écrivons les nombres réels en décimal.

8.2 Notation scientifique

Voici comment écrire un nombre en notation scientifique :
1° placer la virgule juste après le premier chiffre significatif
2° multiplier ce nombre par la puissance de 10 qui convient.

Exemple : 204.875 = 2,04875 10^5 ou 0,0273 = 2,73 10^{-2}

L'exposant donné à 10 compense l'opération faite en déplaçant la virgule.
Chaque déplacement de la virgule vers la gauche diviserait le nombre par 10, il doit donc être contrebalancé en ajoutant 1 à l'exposant de 10.
Inversement, chaque déplacement de la virgule vers la droite multiplierait le nombre par 10 s'il n'était pas compensé en retranchant 1 de l'exposant de 10.

Exercices

Ecrire en notation scientifique
9364,88 381,25 0,000356 0,0356 0,554 0,0003

Transformer en notation décimale
2,7654 10^6 4,42 10^{-1} 6,05 10^4 3,02 10^{-2} 1,25 10^9 1,5 10^{-5}

8.3 Nombres fractionnaires binaires

- En décimal, les chiffres placés derrière la virgule, représentent des dixièmes, des centièmes, des millièmes et ainsi de suite.
  Le $n^{\text{ième}}$ chiffre derrière la virgule a pour poids 10^{-n}.
- En binaire, les principes de numération de positions sont les même. Les chiffres derrière la virgule représentent cette fois des demis, des quarts, des huitièmes etc. Le $n^{\text{ième}}$ chiffre derrière la virgule à pour poids 2^{-n}.

Binaire	Décimal
0,1	= 0,5
0,01	= 0,25
0,001	= 0,125
0,0001	= 0,0625
0,00001	= 0,03125
0,000001	= 0,015625

Remarques :

- Les « décimales » en base dix ne permettent une représentation finie des nombres rationnels que si ceux-ci peuvent s'écrire sous la forme $\frac{N}{10^n}$

  C'est la cas pour la valeur de un quart : 1/4 = 25/10² = 0,25
  ou pour un cinquième : 1/5 = 2/10 = 0,2
  mais pas pour un tiers 1/3 = 0,333.333.333.333.333.333.333.333.333 .... !

- De même, en binaire, les nombres réels et rationnels n'auront une représentation finie que s'ils peuvent être écrits sous la forme $\frac{N}{2^n}$

  C'est la cas pour un demi : $1/10_{(2)} = 0{,}1_{(2)}$
  ou pour un quart : $1/100_{(2)} = 1/100_{(2)} = 0{,}01_{(2)}$
  mais pas pour un tiers : 1/11 = 0,0101 0101 0101 0101 0101 0101 0101….
  ni pour un cinquième : 1/101 = 0,0110 0110 0110 0110 0110 0110 …

Conclusion : Puisque les nombres sont représentés par des codes de dimensions finies, il est très souvent nécessaire d'achever l'écriture de ces codes par un arrondi. La représentation des nombre réels par des nombres en virgule flottante n'est bien souvent qu'approximative.

8.3.1 Conversion de nombre décimaux fractionnaires en binaire

Il est bien rare que l'on doive faire ce genre de calcul. Nous ne le faisons qu'à titre de curiosité. Soit à convertir 0,123 en binaire.

$$\begin{aligned} 0{,}123 &= 0{,}123 \text{ x } 1024 / 1024 \\ &= 125{,}952 / 1024 \\ &\approx 126 / 1024 \\ &\approx 111\ 1110_{(2)} / 10\ 0000\ 0000_{(2)} \\ &\approx 0{,}0001\ 1111\ 1 \end{aligned}$$

! L'arrondi est inévitable car 0,123 ne peut pas être écrit exactement sous la forme $\frac{N}{2^n}$

Autre méthode : multiplications successives par 2

```
0 , 6875   = valeur à convertir
1 , 3750   = 1/2  + 0,375 qui reste à convertir
0 , 750    = 0/4 + 0,750
1 , 50     = 1/8 + 0,5
1 , 0      = 1/16            => Résultat = 0,1011 (2)
```

8.3.2 Exercices

Convertir en décimal :

$1101{,}1_{(2)}$	$11{,}01_{(2)}$	$101{,}11_{(2)}$	$1{,}0101_{(2)}$

Convertir en binaire

2,5	3,75	0,475	15,5625

8.4 Nombres binaires en virgule flottante "*Floating point*"

Le principe est le même que celui de la notation scientifique. Le nombre est écrit en deux parties : les chiffres significatifs et un exposant donné à la base pour faire varier la position de la virgule. La seule différence est qu'ici nous sommes en binaire et non plus en décimal. Les chiffres significatifs ainsi que l'exposant sont codés en binaire. La base des exposants est la base 2.

Cela revient à écrire les nombres sous la forme $\pm M \,.\, 2^{\pm E}$

'M' est la mantisse, 'E' est l'exposant

La mantisse et l'exposant se déterminent de la même manière que pour écrire en notation scientifique avec un seul chiffre significatif avant la virgule.

Exemple écrire « en binaire » sous la forme mantisse exposant les nombres binaires suivants :

$11{,}0111_{(2)} = 1{,}10111_{(2)}\ 10^{1}_{(2)}$	10010 / 100 = ...
$1000 = 1{,}0 \,.\, 10^{3}$	111 / 1000 = ...
$0{,}00101 = 1{,}01 \,.\, 10^{-3}$	10101 x 10000 = ...

8.5 Représentation en machine

Tout comme les nombres entiers, les nombres flottants ont en machine une taille fixée à l'avance.
La norme IEEE 754 a, pour garantir la portabilité des applications, défini deux représentations standardisées : les formats simples et double précision, codés sur respectivement 32 et 64 bits.

Chacun des ces formats comprend 3 champs :

- un bit de signe (0=positif, 1 = négatif),
- quelques bits pour l'exposant
- les autres bits pour la mantisse.

En ce qui concerne la mantisse, puisqu'on est en binaire, le premier chiffre significatif vaut systématiquement 1. On peut donc ne pas écrire ce bit pour gagner un bit de précision en ne conservant que la partie fractionnaire de la mantisse.

L'exposant au lieu d'être un nombre signé a une représentation biaisée. La valeur biaisée est la valeur signée + 127 pour les réels simples ou + 1023 pour les réels doubles.

$$N = (-1)^s \times (1+F) \times 2^{(E-\text{ biais})}$$

En machine les réels simples sont codés sur 32 bits :

1 bit pour le signe
8 bits pour l'exposant (biais = 127)
23 bits pour la partie fractionnaire de la mantisse

s	e0 e1. . . . e7	f_1 f_2 . . . . f_{23}

Les réels doubles sont codés sur 64 bits

1 pour le signe
11 bits pour l'exposant (biais = 1023)
52 bits pour la partie fractionnaire de la mantisse

s	e_0 e_1 . . . . e_{10}	f_1 f_2 . . . . f_{52}

Les conversions de décimal à binaire en virgule flottante sont d'autant plus difficiles que l'exposant est à cheval sur plusieurs bytes. Voici un aperçu de ce que ça donne :

Décimal	Hexa (simple précision)	Hexa (double précision)
-10^{+3}	C4 7A 00 00	C0 8F 40 00 00 00 00 00
-1	BF 80 00 00	BF F0 00 00 00 00 00 00
-10^{-3}	B6 83 12 6F	BF 50 62 4D D2 F1 A9 FC
0	**00 00 00 00**	**00 00 00 00 00 00 00 00**
10^{-6}	35 86 37 BD	3E B0 C6 F7 A0 B5 ED 8D
10^{-3}	36 83 12 6F	3F 50 62 4D D2 F1 A9 FC
0,5	3F 00 00 00	3F E0 00 00 00 00 00 00
1	3F 80 00 00	3F F0 00 00 00 00 00 00
2	40 00 00 00	40 00 00 00 00 00 00 00
3	40 40 00 00	40 08 00 00 00 00 00 00
5	40 A0 00 00	40 14 00 00 00 00 00 00
10	41 20 00 00	40 24 00 00 00 00 00 00
10^{3}	44 7A 00 00	40 8F 40 00 00 00 00 00
10^{6}	49 74 24 00	41 2E 84 80 00 00 00 00
10^{12}	57 68 D4 A5	48 6D 1A 94 A2 00 00 00

Remarques :

- Notez la ressemblance qu'il y a entre les nombres positifs et négatifs
  - Seul change le bit de signe
  - Ce bit de signe est exactement à la même place que pour les nombres entiers signés.
- Remarquez aussi comment la disposition Signe/Exposant/Mantisse adoptée par l'IEEE facilite la comparaison des nombres. La comparaison de deux nombres de signes différents est immédiate, le test du premier bit suffit.
  Ce bit étant mis à zéro, il reste la valeur absolue du nombre. Une comparaison telle qu'on la ferait avec des nombres entiers suffit puisque l'exposant en première position influence la comparaison comme si tout simplement il s'agissait des bits plus significatifs.

Vous trouverez des formulaires de conversions aux adresses suivantes :

Décimal → Virgule flottante	http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html
VF 32 bits → Décimal	http://babbage.cs.qc.edu/courses/cs341/IEEE-754hex32.html
VF 64 bits → Décimal	http://babbage.cs.qc.edu/courses/cs341/IEEE-754hex64.html

8.6 Valeurs particulières

Outre la valeur qui se calcule comme suit $N = (-1)^s \times (1+F) \times 2^{(E-\text{biais})}$ la norme prévoit la représentation de valeurs particulières.

Voici différents cas en simple précision :

Si E=FF et F <> 0	=> ce code n'est pas un nombre : NaN "*Not a number*"
Si S=1, E=FF et F=0	=> N = - ∞
Si S=0, E=FF et F=0	=> N = + ∞
Si E=0 F<>0	=> $N = (-1)^S \times 2^{-126} \times$ 0.ffffff (valeurs non normalisées)
Si S=1 ou 0, E=0, F=0	=> N = 0

S	**E**xposant	Partie **f**ractionaire de la mantisse	
0	1......8	9.....................31	
0	11111111	01010101010000000000000	= NaN
1	11111111	00000000000000000111111	= NaN
0	00000000	00000000000000000000000	= 0
1	00000000	00000000000000000000000	= -0
0	11111111	00000000000000000000000	= + ∞
1	11111111	00000000000000000000000	= - ∞
0	11111110	11111111111111111111111	$= 2^{254-127} \times 1{,}1111..._{(2)}$ $= 2^{127} \times 2 = 2^{128} = 3{,}4\ 10^{38}$ = la plus grande valeur possible <> ∞
0	00000001	00000000000000000000000	$= 2^{1-127} \times 1{,}0 = 2^{-126}$ = la plus petite valeur normalise possible
			Valeurs non normalisées:
0	00000000	10000000000000000000000	$= 2^{-126} \times 0{,}1 = 2^{-127}$
0	00000000	00000000000000000000001	$= 2^{-126} \times$ 0.00000000000000000000001 $= 2^{-149}$ = la plus petite valeur possible <> 0

Bien que ce type de calculs se fasse rarement à la main, exerçons nous tout de même à faire quelques conversions afin de bien saisir le mécanisme de ce codage.

Conversions décimal → binaire

1.0 $= 1 \times 2^{0}$ exposant 0 => sa valeur biaisée = 0 +7F = 7F

```
Signe| Exposant | Partie fractionnaire de la mantisse
    0|011  1111  1|000  0000  0000  0000  0000  0000
       3    F       8    0      0    0      0    0
```

-0.5 $= -1 \times 2^{-1}$ exposant -1 => valeur biaisée = FF +7F = ^{1}7E

```
Signe| Exposant | Partie fractionnaire de la mantisse
    1|011  1111  0|000  0000  0000  0000  0000  0000
       B    F       0    0      0    0      0    0
```

Conversions binaire → décimal

40 40 00 00

```
Signe| Exposant | Partie fractionnaire de la mantisse
    0|100  0000  0|100  0000  0000  0000  0000  0000
       4    0       4    0      0    0      0    0
```

Signe : 0 =+

Valeur biaisée de l'exposant = 1000 0000 = 128

Exposant = 128-127 = 1

Partie fractionnaire de la mantisse : 0,10000000

=> Mantisse = 1,1

Résultat : $N = + 2^{1} \times 1{,}1_{(2)} = 3$

9 Les fonctions logiques

La logique est une forme d'opération de la pensée qui nous permet de raisonner. C'est par exemple la démarche qui vous permettrait de résoudre l'énigme suivante :

Un homme regarde un portrait et dit : « Je n'ai ni frère ni sœur mais le père de cet homme est le fils de mon père ». Qui est représenté sur le portrait ?

Les questions de logique sont parfois moins énigmatiques. :

S'il fait beau ce soir et si j'ai fini ma préparation, j'irai me promener.

Nous tenterons de les ramener à des choix simples où les affirmations sont soit vraies soit fausses, les réponses aux questions sont oui ou non, il n'y a pas de valeurs intermédiaires.
La promenade est liée à deux conditions : la météo et le travail qui reste à faire. Les différentes situations sont représentées dans ce qu'on appelle une table de vérité :

Il fera beau	*Mon travail sera achevé*	*J'irai me promener*
Non	*Non*	***Non***
Non	*Oui*	***Non***
Oui	*Non*	***Non***
Oui	*Oui*	***Oui***

Essayez avec la proposition suivante :

L'accusé sera disculpé si l'enquête révèle qu'il s'agit d'un suicide ou s'il peut faire la preuve qu'il était ailleurs au moment des faits.

Dans le premier cas, la promenade dépend de deux conditions qui doivent être simultanées. Dans le second cas, une seule condition suffit pour disculper l'accusé.

Bon nombre de chercheurs ont tenté de trouver une manière infaillible de raisonner. George Boole traduisit les relations logiques en équations ce qui donna l'***algèbre booléenne***. Il définit ainsi les règles qui permettent de faire des raisonnements valides pour autant que les « variables logiques » ne puissent avoir que deux valeurs possibles : Oui ou Non, Vrai ou faux, 1 ou 0. Ce doivent être des « ***variables binaires*** »
Shannon (1916-2001), l'inventeur du mot « bit », démontra que l'algèbre de Boole était applicable aux circuits électriques. Cela permit à cette époque d'automatiser les centraux téléphoniques. Cette analogie est reprise ci-dessous pour matérialiser le fonctionnement des opérations logiques.

Les fonctions logiques sont en informatique aussi courantes si pas plus que les opérations arithmétiques. La logique combinatoire tout comme l'arithmétique repose sur quelques opérations élémentaires.
En **arithmétique**, ces opérations sont l'addition, la soustraction, la multiplication et la division (**+, -, x, /**). Il est possible à partir de là d'imaginer toutes les autres opérations telles que les exposants, les racines, les logarithmes etc.
En **logique**, les opérations fondamentales sont le **ET**, le **OU** et le **NON**.

La manière la plus simple de comprendre les fonctions logiques est de se les représenter par des schémas électriques qui comportent un ou plusieurs boutons poussoirs et une lampe. Cette lampe s'allume à condition que les contacts électriques y laissent passer le courant.

Le schéma ci-contre traduit la condition la plus simple :
La lampe s'allume si le bouton poussoir A est actionné.
Autrement dit (S = 1) si (A = 1)

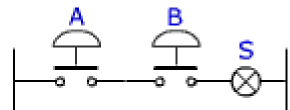
Le fonctionnement de ce circuit s'exprime par l'<u>équation logique</u> $S = A$

Il n'y a que deux cas possibles. Ils sont représentés dans une <u>table de vérité</u>
Une table de vérité a pour le rôle de montrer la correspondance entre la sortie et toutes les combinaisons de valeurs que peuvent prendre la ou les entrées.

A	S
0	0
1	1

Plaçons maintenant deux contacts dans le circuit. La condition nécessaire pour allumer la lampe dépend de la manière dont les contacts sont connectés. Suivant les cas, la condition pour allumer la lampe fait appel aux opérateurs logiques ET ou OU.

9.1 La fonction ET



La lampe s'allume si on active simultanément les contacts A <u>et</u> B

(S = 1) si (A = 1) ET (B = 1)

<u>Tables de vérités</u>

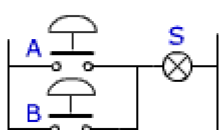
A	B	A et B
0	0	0
0	1	0
1	0	0
1	1	1

<u>Equation logique de la fonction ET</u>

$$S = A \cdot B$$

Le point représente l'opérateur ET.
Ce signe convient parfaitement puisque la fonction ET donne le même résultat qu'une multiplication.

9.2 La fonction OU



La lampe s'allume si on active le contact A <u>ou</u> le contact B

(S = 1) si (A = 1) OU (B = 1)

<u>Tables de vérités</u>

A	B	A ou B
0	0	0
0	1	1
1	0	1
1	1	1

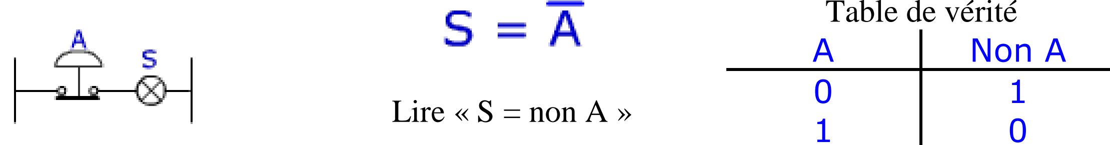
Equation logique de la fonction OU

$$S = A \dot{+} B$$

L'opérateur OU est représenté par un signe plus surmonté d'un point. Observons les trois premières ligne de la table de vérité, le résultat de l'opération OU y est semblable au résultat d'une addition.
Le résultat de 1 ou 1 diffère cependant de 1+1. Nous mettons un point au-dessus du signe '+' pour indiquer que l'opération n'est pas identique à l'addition.

9.3 La fonction NON

Les contacts que nous avons utilisés jusqu'ici, sont des contacts « normalement ouverts ». Quand le bouton poussoir est relâché (quand A = 0) le courant ne passe pas.
Nous utilisons maintenant un contact « normalement fermé » pour illustrer la fonction NON. Le courant traverse le bouton poussoir quand il est relâché mais le courant se coupe quand on actionne le contact (quand A = 1).



$$S = \overline{A}$$

Lire « S = non A »

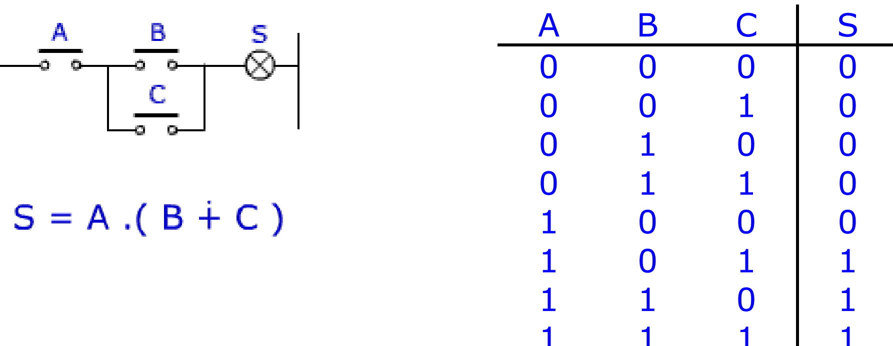
Table de vérité

A	Non A
0	1
1	0

9.4 Combinaisons de fonctions logiques

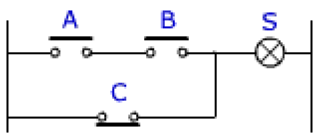
Les trois fonctions de base que nous venons de voir se combinent de multiples façons. A chaque schéma imaginable correspond une équation. La correspondance entre un schéma et une fonction logique est systématique :
- Des contacts en parallèle correspondent à la fonction OU
- Des contacts en série correspondent à la fonction ET
- Un contact normalement fermé représente la fonction NON

Exemples :



$$S = A .(B \dot{+} C)$$

A	B	C	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

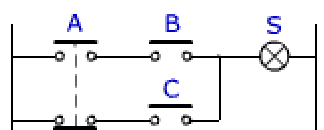


$S = (A \cdot B) + \overline{C}$

A	B	C	S
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tout comme en algèbre la même variable peut apparaître plusieurs fois dans une même équation. C'est le cas dans l'exemple suivant : $S = (A \cdot B) + (\overline{A} \cdot C)$

Quand A vaut 1 dans le premier terme, son complément vaut simultanément 0 dans le second.



$S = (A \cdot B) + (\overline{A} \cdot C)$

A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

9.5 Propriétés des fonctions logiques

OR

Commutativité	$A + B = B + A$
Associativité	$A + B + C = A + (B + C) = (A + B) + C$
Cas particuliers	$A + 1 = 1$
	$A + 0 = A$
	$A + \overline{A} = 1$

ET

Commutativité	$A \cdot B = B \cdot A$
Associativité	$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$
Cas particuliers	$A \cdot 1 = A$
	$A \cdot 0 = 0$
	$A \cdot \overline{A} = 0$

Distributivité

$$A \cdot (B + C) = A \cdot B + A \cdot C$$
$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

OU exclusif

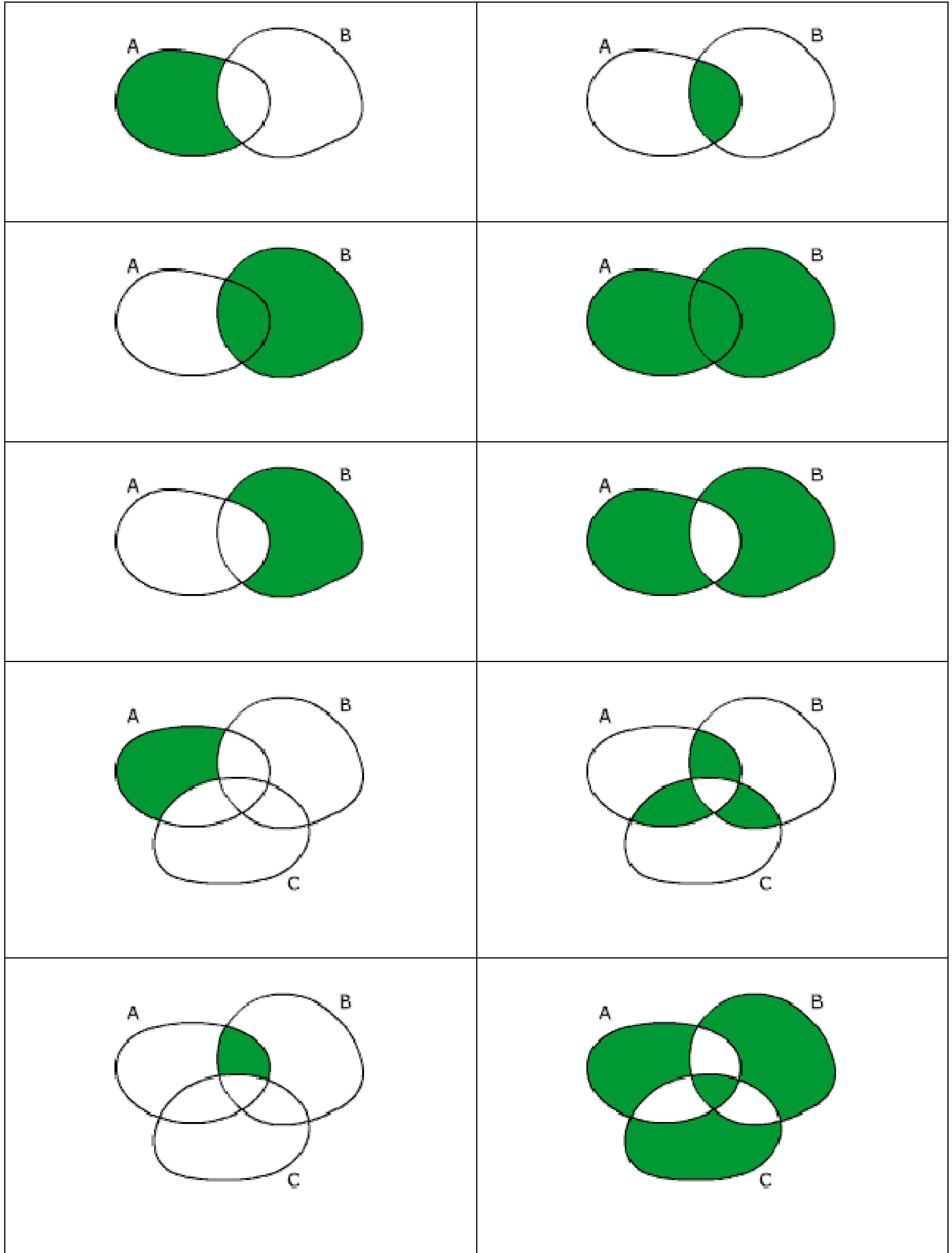
Si $A \oplus B = C$ alors $A \oplus C = B$ et $B \oplus C = A$

Théorème de de Morgan

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$
$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

EXERCICES

1. Donnez les équations logiques que représentent ces diagrammes.



NB. Sachez retracer les tables de vérité pour chacun de ces diagrammes.

2. Dessinez les schémas qui correspondent aux équations logiques suivantes.
Vous devez aussi savoir en tracer les tables de vérité.

$S = A . B$

$S = A + B$

$S = \overline{A}$

$S = \overline{A} . \overline{B}$

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

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

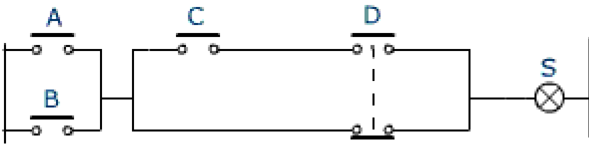
$S = \overline{A} . (B + C)$

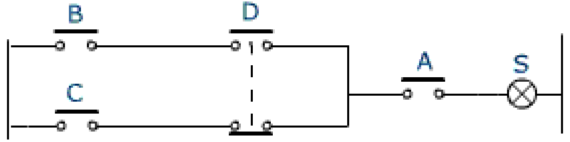
$S = A . (\overline{B} + C)$

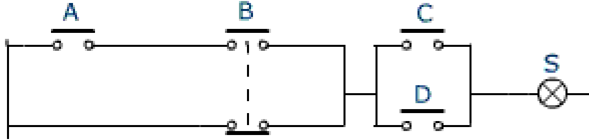
$S = \overline{A . B}$

$S = \overline{A + B}$

3. Donnez les équations logiques qui correspondent aux schémas suivants :





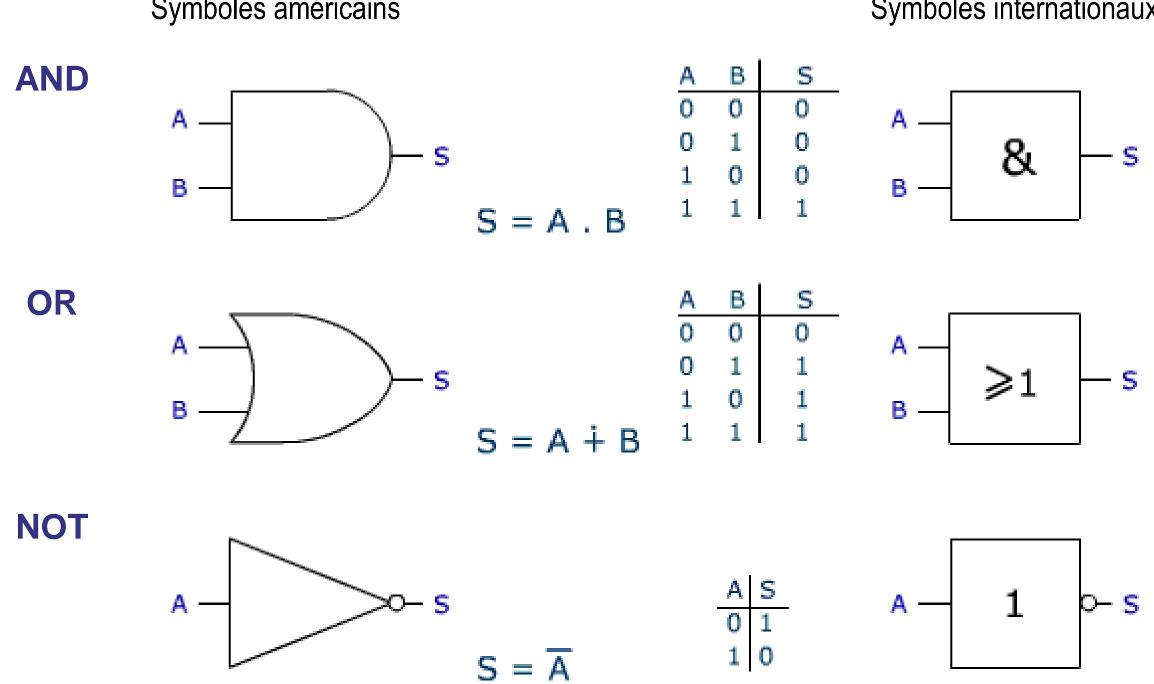


10 Les portes logiques

10.1 Fonctions de base

Nous avons jusqu'ici utilisé des boutons poussoirs et une lampe pour illustrer le fonctionnement des opérateurs logiques. En électronique digitale, les opérations logiques sont effectuées par des portes logiques. Ce sont des circuits qui combinent les signaux logiques présentés à leurs entrées sous forme de tensions. On aura par exemple 5V pour représenter l'état logique 1 et 0V pour représenter l'état 0.

Voici les symboles des trois fonctions de base.



A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

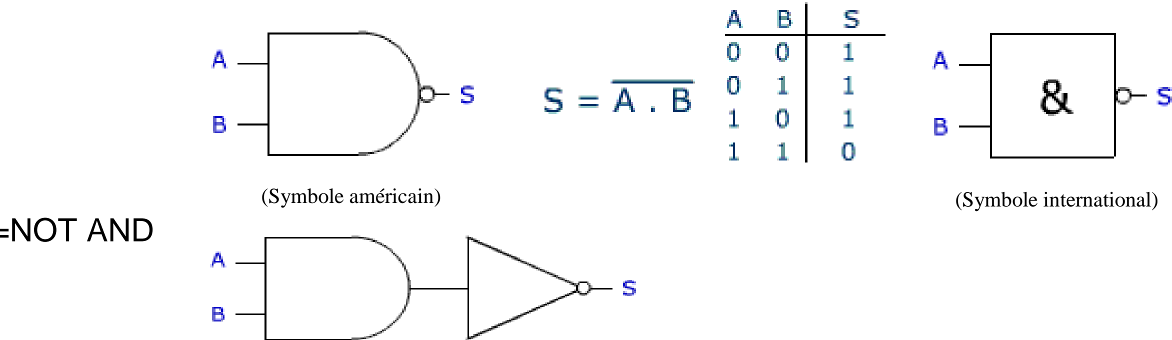
A	S
0	1
1	0

Le nombre d'entrées des fonctions AND et OR n'est pas limité. Voici par exemple une représentation de ces portes avec trois entrées :

10.2 Combinaisons de portes logiques.

Ces trois fonctions logiques de base peuvent être combinées pour réaliser des opérations plus élaborées en interconnectant les entrées et les sorties des portes logiques.

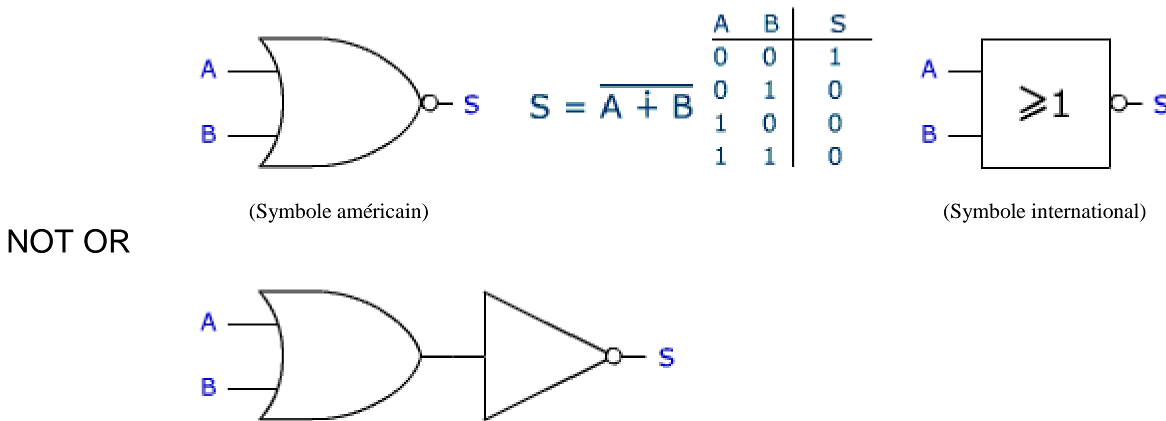
10.2.1 La porte NAND (Non ET)



A	B	S
0	0	1
0	1	1
1	0	1
1	1	0

(Symbole américain) (Symbole international)

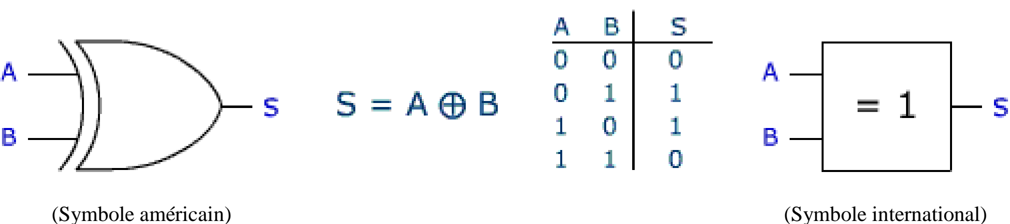
10.2.2 Porte NOR (Non OU)



A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

(Symbole américain) (Symbole international)

10.2.3 Porte XOR



A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

(Symbole américain) (Symbole international)

Porte XOR à deux entrées

La fonction "OU Exclusif" est en principe d'une fonction de deux variables :

$$S = A \oplus B$$

La sortie est à 1 si une seule des deux entrées vaut 1, d'où son appellation « Ou exclusif ».

Porte XOR à plusieurs entrées

Pour calculer le résultat de $S = A \oplus B \oplus C$, on doit pouvoir faire d'abord l'opération entre deux termes, puis refaire un ou exclusif entre le résultat obtenu et le troisième terme.

Ce qui se traduit par $S = (A \oplus B) \oplus C$ ou par $S = A \oplus (B \oplus C)$

On constate que l'appellation "Ou exclusif" n'est plus aussi ben à propos puisque avec trois variables, le résultat vaut 1 si une seule entrée ou toutes les trois valent 1.

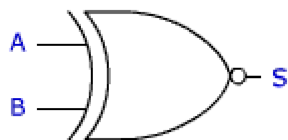
A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Le résultat est en fin de compte un bit de parité. Il vaut 1 si le nombre d'entrées à 1 est impair.

L'inverse de la porte XOR à 2 entrées

Voyons ce que donne la table de vérité si on inverse la sortie d'une porte XOR :

A	B	$S = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1



Le résultat est vaut 1 si les deux entrées sont identiques.

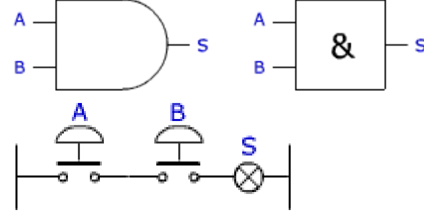
Cette porte teste donc l'équivalence des deux entrées. Certains appellent cette fonction logique, "fonction équivalence", d'autres l'appelle "XNOR"

11 Chronogrammes

Nous avons vu différentes représentations des fonctions logiques :

❖ Les symboles des portes :

(NB. Ces symboles désignent les fonctions sans en illustrer ni les états ni le fonctionnement)



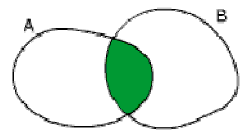
❖ Les schémas électriques :

❖ Les équations et l'algèbre de BOOLE : S = A . B

❖ Les tables de vérité :

A	B	A et B
0	0	0
0	1	0
1	0	0
1	1	1

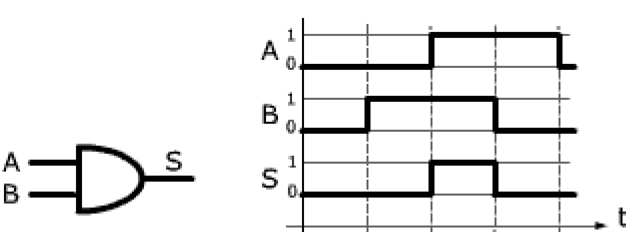
❖ Les diagrammes de VENN



Il existe une autre représentation encore qui devrait vous aider à vous familiariser encore plus avec les fonctions logique : le chronogramme.

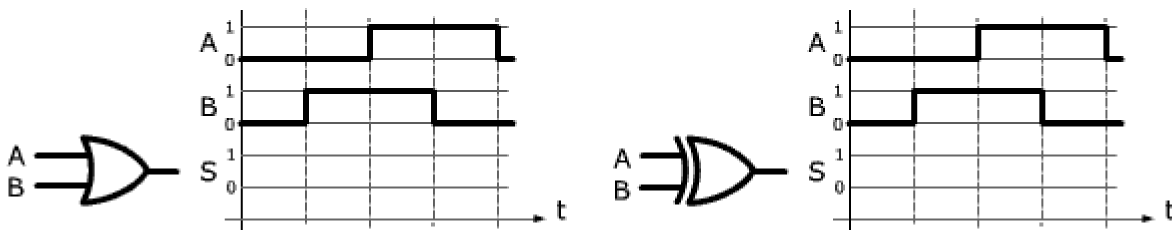
Un **chronogramme** est un diagramme temporel, il représente l'évolution des signaux logiques au cours du temps.

Exemple :



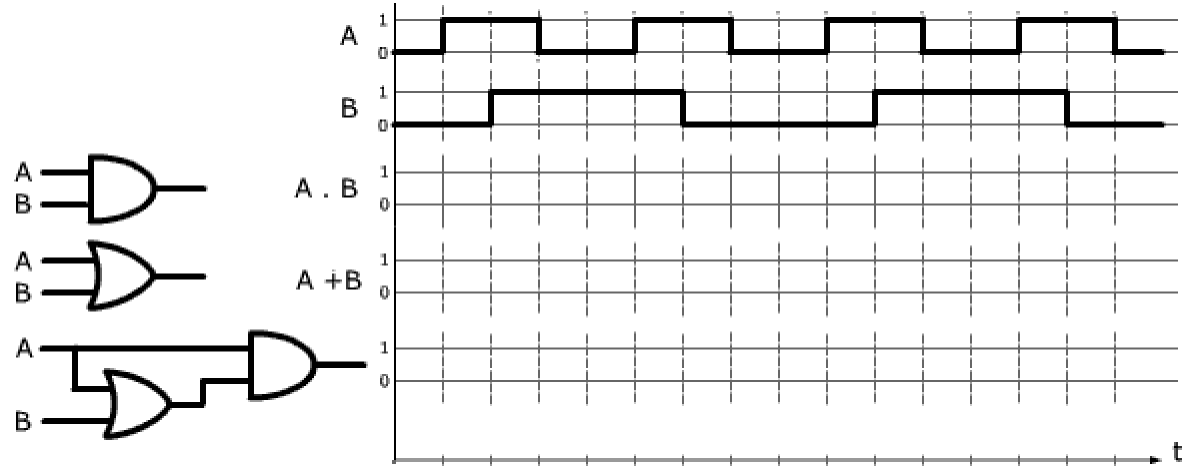
Les électroniciens utilisent les chronogrammes pour étudier le timing des signaux comme nous le faisons pour l'étude des indices de latence pour l'étude du timing des RAM dynamiques http://www.courstechinfo.be/Hard/Memoire.html#TimingRAM

Exerçons nous de suite avec des fonctions simples :



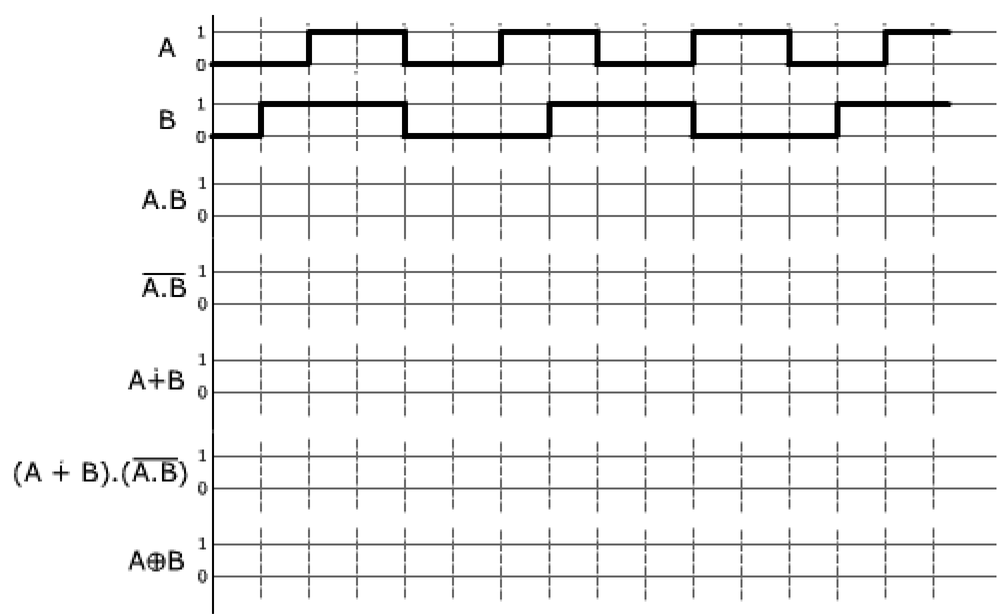
Exercices :

1 **a**) Tracer les chronogrammes des circuits suivants en tenant compte des signaux A et B



b) Prouvez par l'algèbre de Boole que $A.(A+B) \equiv A$

2 **a**)

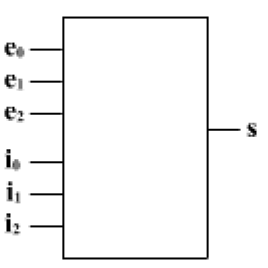


b) Prouvez par l'algèbre de Boole que $(A+B) . (\overline{A . B}) = (A \oplus B)$

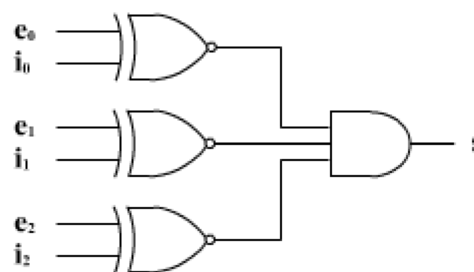
12 Circuits logiques

12.1 Comparateur

Le comparateur est un circuit qui compare deux mots de n bits. En sortie, un bit indique le résultat de la comparaison : 1 s’il y a égalité entre les deux codes présents à l’entrée,
0 si ces codes sont différents.

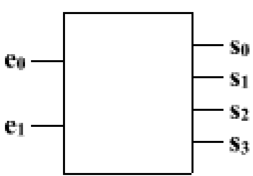


S = 1 si
$e_1=i_1$
et $e_2=i_2$
et $e_3=i_3$

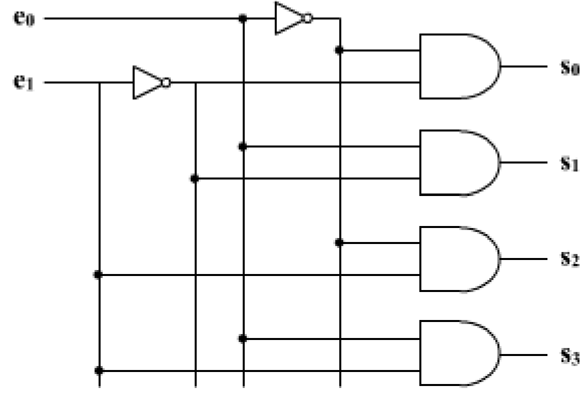


12.2 Décodeur

Le décodeur est un circuit qui possède n bits à d’entrées et 2^n bits en sortie. Parmi toutes ces sorties une seule est active, son numéro est formé par les n bits en entrée.
Exemple : Décodeur "1 parmi 4"



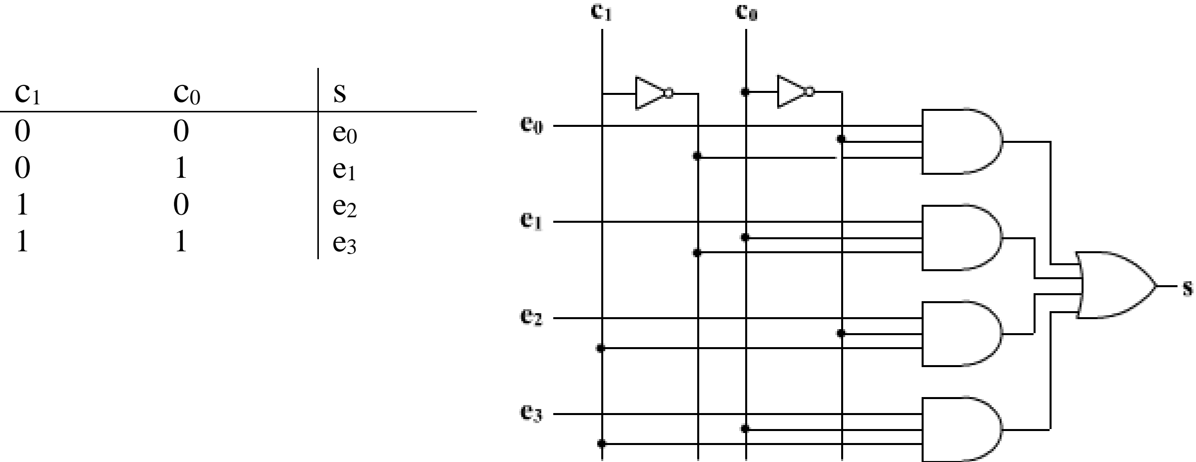
$$s_0 = \overline{e_1} \cdot \overline{e_0}$$
$$s_1 = \overline{e_1} \cdot e_0$$
$$s_2 = e_1 \cdot \overline{e_0}$$
$$s_3 = e_1 \cdot e_0$$



e_1	e_0	s_3	s_2	s_1	s_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

12.3 Multiplexeur

Le multiplexage est une opération qui consiste à utiliser un équipement unique pour traiter plusieurs signaux. Exemple : une ligne de transmission unique pour des signaux multiples. On parle alors de multiplexage temporel : De mêmes intervalles de temps sont accordés successivement pour chacun des signaux à transmettre.

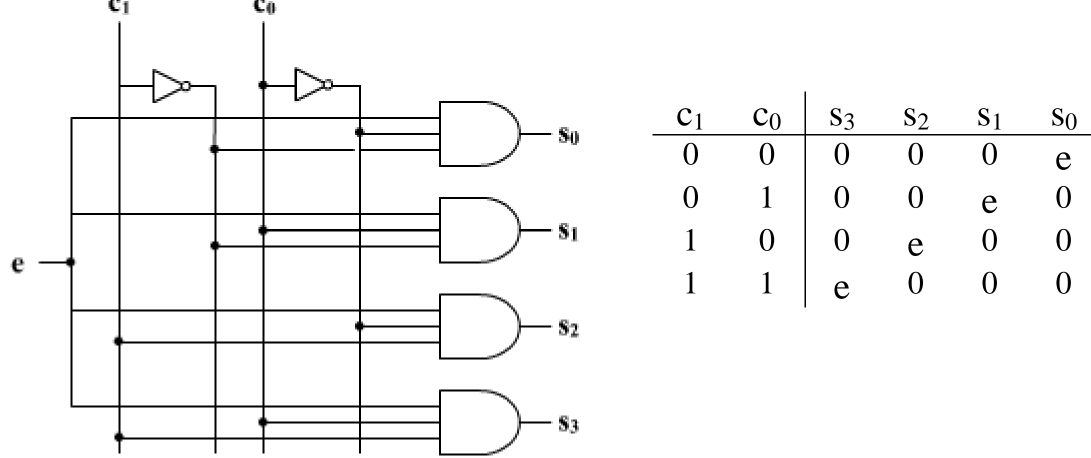


c_1	c_0	s
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

Le multiplexeur agit comme un "commutateur" qui transmet à la sortie le signal d'une entrée sélectionnée par un code binaire[1]..

12.4 Démultiplexeur

Ce circuit réalise la fonction inverse du multiplexeur. Il possède plusieurs sorties (2^n) , un signal en entrée et n bits pour désigner la sortie vers laquelle sera aiguillé le signal d'entrée.

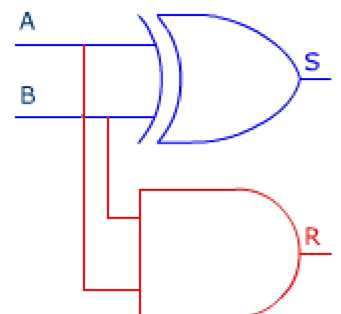


c_1	c_0	S_3	S_2	S_1	S_0
0	0	0	0	0	e
0	1	0	0	e	0
1	0	0	e	0	0
1	1	e	0	0	0

[1] C'est ce genre de circuit qu'il y a pour l'adressage des colonnes d'une RAM dynamique en mode lecture.

12.5 *Le demi* additionneur *half adder*

→ Addition de 2 bits = circuit à 2 entrées



1 bit + 1 bit

S = A ⊕ B
est la somme

R = A . B
est le report

A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

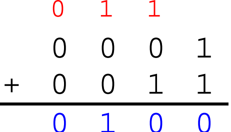
Le demi-additionneur effectue la somme de deux bits. S est la somme et R le report. (*carry*)
Ce schéma ne convient cependant que pour additionner 2 nombres de 1 bit.

0	1	0	1
+0	+0	+1	+1
00	01	01	10

12.6 Le plein additionneur *full adder*

Pour additionner de deux nombres de plusieurs bits il faut mettre en cascade des additionneurs qui additionnent les bits correspondant des deux nombres plus les reports R_{i-1} issus des additions des bits précédents.

Exemple : Calculons 1 + 3
En binaire cela donne : 0001 + 0011 →



L'addition des bits de droite est une addition de deux bits,
elle peut être réalisée avec le demi additionneur
Il faut tenir compte d'un éventuel report pour les bits suivants.
Ainsi dès le deuxième bit de notre exemple (en comptant les bits de droite à gauche) il a fallu faire 2 additions (1 + 0 + 1 = 10 ⇒ on pose 0 et on reporte 1)

Table de vérité du circuit plein additionneur

Cette table de vérité comporte 3 entrées : R_{n-1} (le report de l'addition précédente), A et B

Il lui faut deux sorties : S = la somme de 3 bits $(A + B + R_{n-1})$
R = le nouveau report

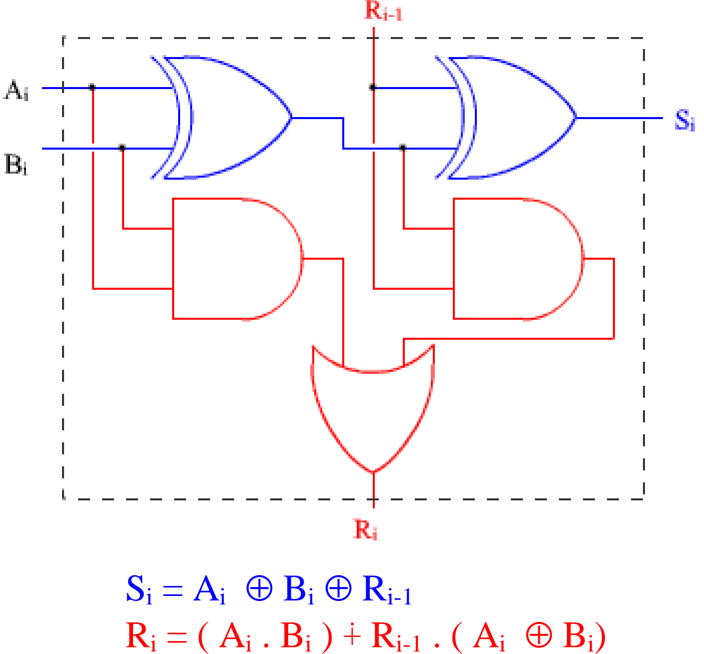
R_{i-1}	A	B	S	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Equations du circuit

$$S_i = A_i \oplus B_i \oplus R_{i-1}$$

$$R_i = (A_i \cdot B_i) + R_{i-1} \cdot (A_i \oplus B_i)$$

Schéma du circuit plein additionneur



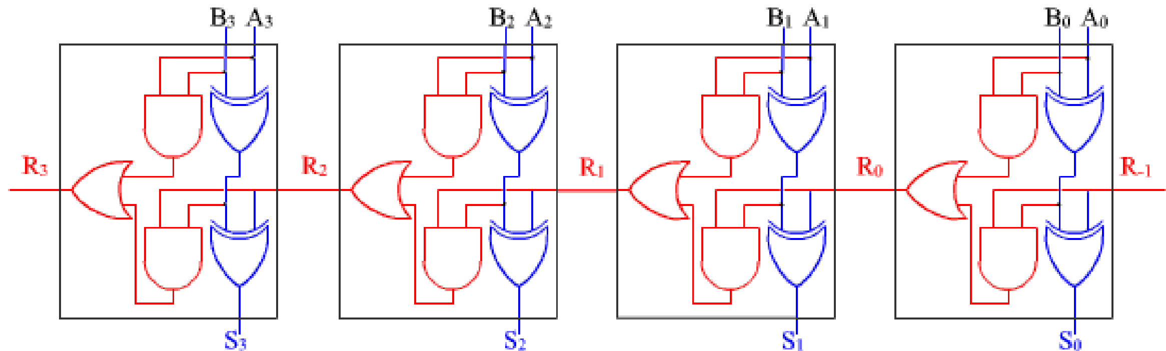
$$S_i = A_i \oplus B_i \oplus R_{i-1}$$

$$R_i = (A_i \cdot B_i) + R_{i-1} \cdot (A_i \oplus B_i)$$

Le plein additionneur est un circuit à 3 entrées. Il se compose de 2 demi additionneurs et d'une porte OU qui génère le report quand la somme vaut 2 ou 3

12.7 Addition de deux nombres de n bits

Exemple : Mise en cascade de 4 additionneurs pour l'addition de deux nombres de 4 bits
Le circuit peut tenir compte de l'éventuel report précédent R_{-1}
Le report $R_3 = 1$ dès que l'écriture de la somme nécessite plus de 4 bits



EXERCICES

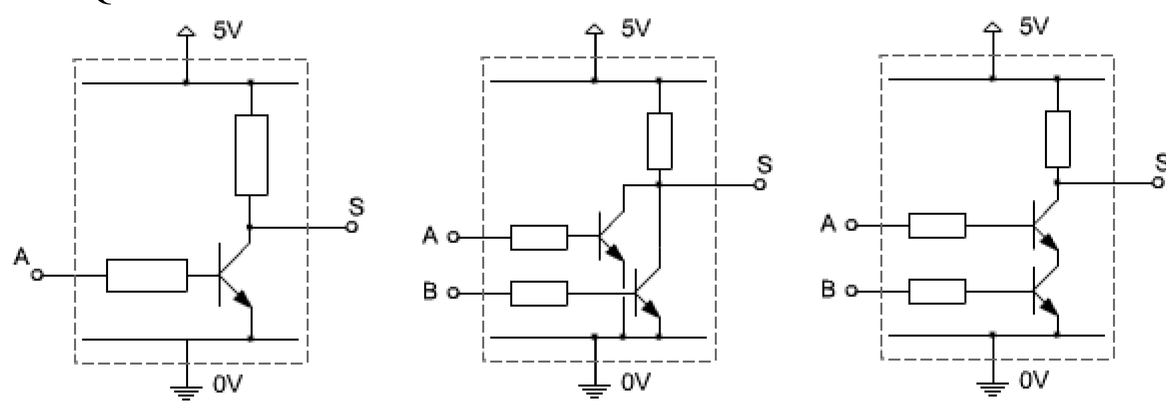
1 La table de vérité ci-contre représente les états d'un circuit à 3 entrées (A, B et C) et 2 sorties (S_0 et S_1)

Observez les états des sorties S1 et S0 et donnez-en les équations

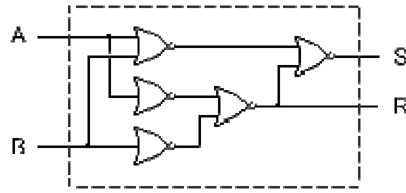
Dessinez le montage de portes logiques pour réaliser ces fonctions.

A	**B**	**C**	**S_1**	**S_0**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2 Quelles sont les fonctions des circuits suivants :



3 Montrez que ce circuit est un demi-additionneur

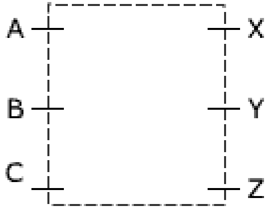


4 Le code binaire réfléchi encore appelé "code Gray" est tel que d'une valeur à la suivante il n'y a jamais qu'un seul bit qui change. La correspondance entre code binaire naturel de 3 bits et code binaire réfléchi est donnée par la table suivante :

Binaire naturel			Binaire réfléchi		
C	B	A	Z	Y	X
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

1° Ecrivez les équations de X, Y et Z en fonction de A, B et C

2° Dessiner les circuits qui pourraient faire ce décodage à l'aide de portes OR, AND et/ou XOR



Annexe A Changements de bases pour nombres de 8 bits

Conversions Binaire ↔ **Hexadécimal** ↔ Décimal Nombres non signés

		00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0000	**0**	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0001	**1**	0	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
0010	**2**	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
0011	**3**	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
0100	**4**	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
0101	**5**	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
0110	**6**	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
0111	**7**	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
1000	**8**	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
1001	**9**	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
1010	**A**	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
1011	**B**	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
1100	**C**	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
1101	**D**	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
1110	**E**	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
1111	**F**	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Conversions **Hexadécimal** ↔ Décimal Nombres signés

	00	10	20	30	40	50	60	70	*80*	*90*	*A0*	*B0*	*C0*	*D0*	*E0*	*F0*
0	0	16	32	48	64	80	96	112	*-128*	*-112*	*-96*	*-80*	*-64*	*-48*	*-32*	*-16*
1	0	17	33	49	65	81	97	113	*-127*	*-111*	*-95*	*-79*	*-63*	*-47*	*-31*	*-15*
2	2	18	34	50	66	82	98	114	*-126*	*-110*	*-94*	*-78*	*-62*	*-46*	*-30*	*-14*
3	3	19	35	51	67	83	99	115	*-125*	*-109*	*-93*	*-77*	*-61*	*-45*	*-29*	*-13*
4	4	20	36	52	68	84	100	116	*-124*	*-108*	*-92*	*-76*	*-60*	*-44*	*-28*	*-12*
5	5	21	37	53	69	85	101	117	*-123*	*-107*	*-91*	*-75*	*-59*	*-43*	*-27*	*-11*
6	6	22	38	54	70	86	102	118	*-122*	*-106*	*-90*	*-74*	*-58*	*-42*	*-26*	*-10*
7	7	23	39	55	71	87	103	119	*-121*	*-105*	*-89*	*-73*	*-57*	*-41*	*-25*	*-9*
8	8	24	40	56	72	88	104	120	*-120*	*-104*	*-88*	*-72*	*-56*	*-40*	*-24*	*-8*
9	9	25	41	57	73	89	105	121	*-119*	*-103*	*-87*	*-71*	*-55*	*-39*	*-23*	*-7*
A	10	26	42	58	74	90	106	122	*-118*	*-102*	*-86*	*-70*	*-54*	*-38*	*-22*	*-6*
B	11	27	43	59	75	91	107	123	*-117*	*-101*	*-85*	*-69*	*-53*	*-37*	*-21*	*-5*
C	12	28	44	60	76	92	108	124	*-116*	*-100*	*-84*	*-68*	*-52*	*-36*	*-20*	*-4*
D	13	29	45	61	77	93	109	125	*-115*	*-99*	*-83*	*-67*	*-51*	*-35*	*-19*	*-3*
E	14	30	46	62	78	94	110	126	*-114*	*-98*	*-82*	*-66*	*-50*	*-34*	*-18*	*-2*
F	15	31	47	63	79	95	111	127	*-113*	*-97*	*-81*	*-65*	*-49*	*-33*	*-17*	*-1*