COURS DE CIRCUITS NUMERIQUES

Poly **4**

Systèmes de Numération

Codes binaires

Arithmétique Binaire

Par G. PALLOT

SOMMAIRE

1. NUMERATION BINAIRE, NOMBRES POSITIFS

1.1. Principe

EN DECIMAL:

$427,123 = \underline{4}.10^2 + \underline{2}.10^1 + \underline{7}.10^0 + \underline{1}.10^{-1} + \underline{2}.10^{-2} + \underline{3}.10^{-3}$

Poids :	100	10	1	0,1	0,01	0,001

Le code est Pondéré, il permet les opérations arithmétiques. Les chiffres sont de 0 à 9 . La base est 10.

Décaler un nombre = décaler en **laissant la virgule en place** :
Un décalage du nombre à droite divise le nombre par 10
Un décalage à gauche multiplie le nombre par 10

EN BINAIRE:

$1001,101 = \underline{1}.2^3 + \underline{0}.2^2 + \underline{0}.2^1 + \underline{1}.2^0 + \underline{1}.2^{-1} + \underline{0}.2^{-2} + \underline{1}.2^{-3}$

Poids :	8	4	2	1	0,5	0,25	0,125

= 9, 625 décimal .

REMARQUE IMPORTANTE : La virgule n' est pas en mémoire!!! . L' utilisateur doit savoir à tout instant où se situe la virgule, et réaliser les traitements suivants en fonction de celle ci. Ceci est primordial quand on travaille dans le langage de base de la machine (assembleur) .

Dans les langages évolués (FORTRAN, COBOL, C ....) , l'utilisateur n'a pratiquement pas à s'en soucier .

La virgule peut être en mémoire (sous forme d' un code particulier) , lorsqu' un nombre décimal est écrit en mémoire ou dans un fichier sous forme de CARACTERES , mais jamais ou niveau du code binaire du nombre .

Pour définir la façon dont est codé un nombre, on doit donner:

-son **FORMAT** : nombre d'octets et position de la virgule (pour un format tel que celui présenté, nommé " VIRGULE FIXE")

-Le **type de code** d'un nombre (binaire non signé, binaire signé et son mode...). Nous verrons les notations virgules flottantes et les codes signés ultérieurement.

Ce code est également pondéré et permettra les opérations arithmétiques.
Les chiffres sont seulement **0** et **1**, appelés "**bits**" ou éléments binaires.
La base est 2.
Un décalage à droite divise le nombre par 2
Un décalage à gauche multiplie le nombre par 2
Le bit le plus à droite est le "**bit de faible poids**" ou LSB (least significant bit)
Le bit le plus à gauche est le "**bit de fort poids**" ou MSB (most significant bit)
Un groupement de 8 bits = un **octet .** Les bits sont groupés par 8, ou 16 bits dans les ordinateurs . Une valeur peut être codée sur 1 , 2 , 3 ,4 ,5 ... etc octets .

Un groupement de 4 bits est un quartet, on parle des quartets bas et haut de l'octet.
2^{10} = **1024 = le "kilo"**
2^{20} = 1024.1024 = le "Méga" (un peu plus de 10^6 soit 1048576).
(exemples: mémoire de 64 kilo octets, disques durs de 200 Méga octets)
La suite des entiers codée en binaire fournit "**le code binaire naturel"** .

1.2. Conversions

1.2.1. Conversion binaire › décimal

A la main, on obtient évidemment un résultat décimal. L'ordinateur lui ne peut travailler qu'avec des 0 et des 1, il fournira donc chaque chiffre décimaux de 0 à 9 en binaire.

1.2.1.1. Partie entière

a) A la main: - Directement d'après la définition . (calcul à la main). On additionne les poids présents. On peut se servir d'une calculette de poche si le mode binaire existe.

b) L'ordinateur: - Par comptage et soustraction, en commençant par la puissance de 10 la plus haute possible (exemple: comptage des mille, des cents, des dizaines, il reste les unités

- Par des division (en binaire) par 10 décimal, les restes successifs fournissent les unités, dizaines, centaines.

1.2.1.2. Partie fractionnaire (m bits fractionnaires)

a) A la main: - Par les puissances négatives de 2, pratique pour voir l'ordre de grandeur (en tenant compte des 2 ou 3 premiers bits), pénible obtenir la valeur exacte.

- On convertit l'entier sans tenir compte de la virgule, on divise par 2^m. Une calculette est pratique.

b) L'ordinateur: - Des produits successifs par 10, avec prise à chaque fois de la partie entière, fournissent dans l'ordre les chiffres des dixième, des centièmes, des millièmes ...

- On peut aussi obtenir en binaire directement le nombre de $1000^{èmes}$ par exemple en multipliant par 10^3, on convertit alors ce nombre entier en décimal.

1.2.2. Conversion décimal › binaire

1.2.2.1. Partie entière

a) A la main:

- Soustractions successives des puissances de 2 (en commençant par la plus grande) : méthode des puissances de 2.

	puissances de 2 successives		
Exemple 121 ?	1	(LSB en dernier))	
121	2	0	
-64	4	0	
57	8	1	
-32	16	1	
25	32	1	
-16	64	**1**	(on trouve le **MSB** en premier)
9 (= 8 + 1)	128		

Donc **121 = 1111001** sur 7 bits (suffisant) ou **0111 1001** sur un octet.
C'est la méthode manuelle la plus rapide pour convertir directement en binaire.
Par contre ce n'est pas la plus rapide lors d'une conversion effectuée par l'ordinateur.

- Divisions successives par deux. (à la main)

Démonstration : soit -|- - - - - - -, | (un tiret = 1 bit)

Si on décale petit à petit ce nombre vers la droite (division par 2) on recueille chaque bit en commençant par le LSB.

Cet algorithme peut aussi se programmer (divisions par 2 et récupération de la partie fractionnaire à chaque fois).

A la main: 121 I_2_.

1 60 I_2_. On trouve le LSB en premier .

0 30 I_2_. On aime ou on n'aime pas !

0 15 I_2_.

1 7 I_2_.

1 3 I_2

1 1 I_2

1 0

c) L'ordinateur: Directement par la formule de définition (tous les opérandes étant évidemment en binaire)

127 s'écrit $1.10^2 + 2.10^1 + 7$, l'ordinateur calculera donc:

1 . 1100100 + 10 . 1010 + 111 = 11111111 normalement !

1.2.2.2.Décimal-Binaire, partie fractionnaire

a) A la main (et calculette)

Selon la précision voulue, on se limitera à n bits (précision absolue $\pm 2^{-(n-1)}$): Exemple sur 4 bits n = ,1010 = 0,625 à $\pm 2^{-5}$ ou ± 0,031.

- Puissances négatives de 2. Peu pratique au delà de 2^{-4} (0,0625) ou 2^{-5} (0,03125).

- Puissances de deux , en se ramenant à un entier.

Le nombre n s'écrit en fait n = ,- etc en fonction de la précision.

Si par exemple on se limite à **6** bits, on veut n sous la forme ,- - - - - - (en acceptant éventuellement une erreur).

On raisonne sur un nombre $2^{\mathbf{6}}$ fois plus grand :

$N=2^{\mathbf{6}}.n=$ - - - - - -, - - - - - - - - -

On ne garde que l'entier le plus proche

$N \simeq$ - - - - - - ,

On convertit cet entier . Le code binaire (virgule mise à part) est le même pour ce nombre entier arrondi que pour le nombre de départ approché .

Exemple: 0,371 sur **8** bits , code et erreur?

On cherche le code de $2^{\mathbf{8}}$. 0,371 = 256 . 0,371 = 94,976 Ce nombre n'est pas entier donc n ne pourra être codé sur 8 bits que moyennant une approximation.

L'entier le plus proche est 95 qui ,converti en binaire sur 8 bits, donne 01011111 .

Le code approché sur 8 bits de 0,371 est le même soit $n \simeq$,01011111 .

(Encore une fois, la virgule est marquée ici pour la compréhension , elle n'est pas en mémoire !!).

On obtient une valeur arrondie au plus près, et on peut trouver facilement l'erreur commise: on a pris 95 ou lieu de 94,976 donc une erreur de 95-94,976 = 0,024 sur 256.n

L'erreur sur n est donc de 0,024/256 = $+ 9{,}37\ 10^{-5}$.

- Multiplications successives par 2.

On multiplie par deux successivement, chaque bit est égal à la partie entière, que l'on enlève progressivement .

Démonstration: ,|- - - - - - - -| (un tiret = 1 bit)

Des décalages à gauche font progressivement défiler les bits, en commençant par le bit de fort poids.

Exemple 0,371	0,742	0,484	0,968	0,936 _ _ _ _ _
2	2	2	2	2
0, 742	**1**, 484	**0**, 968	**1**, 936	**1** ,872

Résultat sur 5 bits: ,01011 . On n'a pas directement l'erreur, on obtient une valeur par défaut et non au plus prés.

b) L'ordinateur: (D chiffres fractionnaires)

A partir des chiffres décimaux codés en binaire, on peut raisonner en entier, convertir en binaire, diviser (en binaire) par 10^D, le quotient (< 1 ici) fournit le résultat.

Remarque, la division entière classique peut donner donner 0 si le numérateur est inférieur au dénominateur, il faut pouvoir obtenir la partie fractionnaire !

1.3. Précision

Si on arrondit à l'entier le plus proche, la précision absolue est de ± 0,5 .

Si on généralise, un arrondi de la partie fractionnaire à **N** bits, founit une **précision absolue de ± $2^{-(N+1)}$** . Exemple sur 9 bits après la virgule, la précision est de $\pm 2^{-10}$ = 1/1024 _ 0,001 (3 chiffres décimaux significatifs après la virgule) .

Choix du nombre de bits fractionnaires pour une précision décimal voulue:

Soit une partie fractionnaire en décimal, de **D** chiffres fractionnaires , _ _ _ _ _ . . .

On veut la coder en binaire au moyen de m bits, sous la forme: ,- - - - - - . . .

Dans le cas général, il faut que $2^{-m} <= 10^{-D}$

Donc $m \geq D\frac{Ln10}{Ln2}$ (Ln ou log décimal)

Pour certains cas particuliers, avec peu de bits, on peut coder un nombre fractionnaire sans erreurs, lorsque ce nombre tombe juste sur un pas de quantification binaire: exemple 0,75 se code sur 2 bits, avec une erreur nulle (,11)

1.4. Dynamique

On appelle DYNAMIQUE sur N bits, et dans un code ou un format donné, l'étendue des nombres que l'on peut ainsi représenter.

Exemple En **entier N bits non signés,** on peut coder $\mathbf{2^N}$ nombres de **0** à $\mathbf{2^N - 1}$
(sur 4 bits 2^4 = 16 nombres de 0 à 15)
(sur 8 bits 2^8 = 256 nombres de 0 à 255)
(sur 16 bits 2^{16} = 65536 nombres de 0 à 65535)

Cas général: pour coder sur n bits, une partie entière de valeur max Nmax, il faut que $2^n - 1 >= Nmax$ et donc $n \geq \frac{Ln(N\max+1)}{Ln2}$

1.5. Formats Virgule Fixe et Virgule Flottante

En décimal : 1256,57 (virgule fixe) s' écrit aussi: $0{,}1256570000.10^4$ (virgule flottante 10 chiffres significatifs) .

En binaire : 1011,1101 (virgule fixe) peut s' écrire de la même manière en une mantisse et un exposant, mais sous forme de puissance de 2:

Mantisse ,1011110100000000 (exemple 16 bits)

Exposant 00000100 (exemple 8 bits)

Du **nombre de bits de la mantisse** dépend la **précision,** et donc le nombre de chiffres décimaux significatifs.

Du **nombre de bits de l'exposant** dépend la **dynamique.**

Les conversions virgule fixe - virgule flottante (out réciproquement)**, s'effectuent par l'ordinateur**, et ne sont pas simples ! (passage des puissances de 2 aux puissances de 10, la mantisse avant d'être convertie est donc modifiée).

	VIRGULE FIXE	**VIRGULE FLOTTANTE**
Avantages	**Calculs rapides** Conversions simples	**Grande dynamique** Programmation aisée en langage évolués
Inconvénients	**Dynamique relativement faible** pour un nombre de bits donné Programmation délicate	**Calculs plus lents** Nécessité d'un langage évolué, de nombreux sous programmes de calcul et de conversion, ou de circuits supplémentaires ("coprocesseur virgule flottante"), plus onéreux.
Utilisation	Domaine industriel: Petits systèmes en Automatisme, Robotique, machines outils, appareils de mesure,..., traitement de signal. Grand public: jeux, electroménager, HIFI (lecteur compact disques)	Calculs scientifiques en informatique, Gros systèmes en Automatisme, contrôlés par ordinateurs. Traitement du signal.

2. NOTATION HEXADECIMALE

2.1. Principe

Pour écrire des nombres binaires plus rapidement, on utilise fréquemment la notation hexadécimale. Sur un ordinateur, pour visualiser du binaire en mémoire, on utilise ce format.

2.1.1. Cas de nombres entiers non signés

Cette notation est dans ce cas un système de numération:
On utilise les 16 chiffres :

0 1 2 3 4 5 6 7 8 9	et	A B C D E F
correspondant à **0 1 ... 9** du décimal		correspondant à **10 11 12 13 14 15** du décimal

Soit un nombre binaire, dont le nombre de bits est multiple de 4 (on rajoutera des bits si besoin est du coté des poids forts)

0101011110101011

Cette écriture est longue et difficile à lire !

On forme des tranches de 4 bits : 0101 0111 1010 1011

Chaque groupement de 4 bits à pour poids: 16^3 16^2 16^1 16^0

A chaque groupement de 4 bits ,de valeur décimale entre 0 et 15, on fait alors correspondre un chiffre :

0000	0	
0001	1	
0010	2	
0011	3	chiffres décimaux de 0 à 9
0100	4	
etc		
-		
1001	9	
1010	A	(10 décimal)
1011	B	(11 décimal)
1100	C	(12 décimal)
1101	D	(13 décimal)
1110	E	(14 décimal)
1111	F	(15 décimal)

Le nombre binaire 0101 0111 1010 1011 s' écrit **57AB** en hexadécimal.
Un octet s'écrit avec deux chiffres : 1000 1110 = 8E

2.1.2. Cas de code binaire quelconque

On peut toujours grouper 4 par 4 les bits à partir de la droite, et faire une correspondance 4 bits = 1 caractère héxadécimal. Il n'y a plus de système de numération, mais seulement alors une écriture rapide ! (écriture 4x4, passe partout .....) !

« L'écriture hexadécimale» est donc un moyen d'écrire rapidement un nombre ou un code binaire, ***en l'interprétant comme un entier positif*** (la virgule ne se voit plus si elle n'est pas à une position multiple de 4 !, le signe non plus comme nous le verrons plus loin, et le code binaire peut représenter même n'importe quoi, pas forcément un nombre, exemple code d'instruction d'un microprocesseur d'ordinateur).

Les nombres dans un ordinateur sont toujours formés d'une suite de 0 et de 1, on peut cependant, au moyen d'un logiciel approprié de conversion binaire, introduire des valeurs hexadécimales au clavier (à partir des caractères de 0 à F).

Pour examiner les contenus des mémoires (le plus souvent 8 ou 16 bits), on pourra afficher la forme hexadécimale, plus facile à lire. La donnée 8 ou 16 bits voyage en direction de la console de visualisation sous forme de 2 ou de 4 caractères (parmi 0,1,2 , ... A,B,... F) et non pas sous forme de leur valeur binaire.

2.2. Conversions

2.2.1. Hexadécimal ⇒ Binaire et Binaire ⇒ Hexadécimal

Pour tout code, mais en fait, numériquement parlant, l'égalité n'a lieu que pour des entiers positifs ! Par lecture diecte :

7C = 0111 1100

0111 1000 = 78hexa (bien marquer hexa pour ne pas confondre avec 78 décimal!)

2.2.2. Décimal ⇒ Hexadécimal directe:

Pour des nombres entiers positifs seulement !

Un peu comme en binaire:

a) Essai des puissances de 16 successives (en commençant par la plus grande possible).

Ces puissances sont 16^0 16^1 16^2 16^3 16^4

ou 1 16 256 4096 65536

En binaire, le poids était présent (bit 0) ou non (bit 1), ici, il faut compter combien de fois est présent le poids (de 0 à 15 fois)

exemple: 751 ?

La plus grande puissance de 16 possible : 256

On divise : 751 I 256 . puis 239 I 16 .

239 **2** **15** **14**

| |

chiffre de poids fort chiffre de poids faible (le dernier reste < 16)

Résultat 2 E F

(avec une calculatrice, on divise 751 par 256, on trouve **2**, 933..

La partie entière **2** est le chiffre de fort poids.

On trouve le reste par 751 - 2 . 256 = 239 et on continue .)

b) Divisions successives par 16 : (on obtient le chiffre de poids faible en premier)

exemple: 751 ?

très rapide car le chiffre est petit:

```
751  I__16_.
111     46  I__16_.
        15     14      2
         |              |
poids faible    poids fort
```

c) Usage d'une **calculette** en mode hexadécimal. Certaines calculettes peuvent effectuer directement cette conversion, comme la suivante d'ailleurs, mais attention certaines aussi travaillent toujours en code signé, et au delà d'une certaine valeur, si le poids fort est 1, la valeur est interprétée comme négative, comme nous le verrons plus loin...

2.2.3. Hexadécimal $\Rightarrow$ Décimal directement

Pour des entiers positifs seulement !

D'après la définition, $2EF = 2 . 16^2 + 14 . 16^1 + 15 = 2 * 256+14*16+15 = 751$

3. NOTATION DECIMAL CODE BINAIRE

3.1. Définitions

Nommé code DCB ou BCD (abréviations françaises ou anglaises).

Le monde extérieur étant un monde décimal, l'utilisateur doit pouvoir introduire au clavier et afficher sur l'écran directement des nombres décimaux.

Une valeur négative est écrite évidemment tout bêtement avec un signe - devant. Un signe + peut aussi se mettre devant une valeur positive.

3.1.1. Codage 1 chiffre = 1 octet

- ***Codage chiffres:***

Les chiffres décimaux de 0 à 9, tenant normalement sur 4 bits, sont codés sur 8 bits (l'octet étant le groupement de base dans un ordinateur).

Le nombre non signé 126 par exemple se codera par les trois octets successifs

0000 0001
0000 0010
0000 0110

Un nombre tel que +46,77 ou -46,77 peut se coder par 5 ou 6octets:

Code du signe (souvent 0000 0000 pour + et 1111 1111 pour -)
0000 0100
0000 0110
Code de virgule (absent si le programme de traitement connaît la position de celle ci)
0000 0111
0000 0111

Ce codage est **pratique**, un octet = directement la valeur du chiffre, les conversions sont donc rapides.

- ***Codage caractères:***

En ajoutant à chaque code chiffre le code 0011 0000 on passe du chiffre au caractère clavier code ASCII). On rappelle que ne peuvent **circuler** sur des liaisons entre ordinateur et consoles que des **codes caractères**, sinon des valeurs binaires quelconques peuvent correspondre à des codes de commande tels que arrêt et reprise de transmission, code de programmation des consoles ... et tout planter !

Exemple:

chiffre 3 = code chiffre binaire 0000 0011 (code arithmétique, pondéré)

caractère 3 = code (ASCII) binaire 0011 0011 (écriture héxa 33)

Ce code est évidemment non arithmétique, il faut mette à zéro le quartet haut pour retrouver l'entier correspondant).

3.1.2. Code compact: 1 octet = 2 chiffres décimaux

On regroupe les chiffres 2 par 2, et on code chaque chiffre décimal en binaire, sur 4 bits:

Exemple 347,26 = 0011 0100 0111 , 0010 0110

Centaines dizaines unités dixièmes centièmes

La virgule peut être toujours à la même place, donc inutile alors de la coder. Sinon il faut prévoir un quartet spécial virgule (par exemple 1010). Chaque chiffre étant < 10, sauf erreur de calcul on ne peut confondre le code de la virgule avec un chiffre.

Pour des nombres non signés, le signe est omis, sinon on peut prévoir un quartet signe (par exemple 0000 pour + et 1111 pour -), en sachant que le premier quartet est toujours le signe.

Utilisation: ce regroupement 4 par 4 est pratique pour sortir des valeurs numériques sur des afficheurs 7 segments, 16 bits fournissent 4 chiffres. Par contre, pour les conversions binaires, on doit revenir à 1 chiffre = 1 octet. On s'en sert aussi, avec une variante pour coder le signe en langage COBOL dans le format type « COMP-3 »

3.2. Conversions

3.2.1. Décimal ⇔ DCB

1) Code compact (2 chiffres par octet)

Les conversions dans les deux sens sont immédiates, par lecture 4 par 4 à partir de la droite. Il faut donc étendre à un nombre pair de chiffres.

Exemple : 347d = 0000 **0011 0100 0111** ou **écriture hexa** 03_h 47_h

Attention, ne pas confondre le nombre DCB 0011 0100 0111 (347 décimal), et le nombre binaire 0011 0100 0111 (347 hexadécimal) = 839 décimal !!!. La suite n' est pas pondérée de la même façon .

Un nombre DCB décalé d'un cran dans un sens ou dans l'autre donne n'importe quoi ! Par contre, un décalage de 4 bits le multiplie ou le divise par 10 .

2) Code 1 chiffre par octet

Evident, exemple : 347d = 00000011 00000100 00000111 3 octets minimum

Ecriture hexa : 3 octets 03_h 04_h 07_h

3.2.2. DCB ⇔ Binaire

Comme précédemment, mais dans l'autre sens par lecture directe, 2 chiffres par octet ou un seul selon le codage.

3.3. Comparaison Binaire --- Décimal Codé Binaire

Les deux sont nécessaires, on passe de l'un à l'autre par des programmes de conversion.

	Binaire	**DCB**
Avantages	Proche de la machine Calculs rapides	Proche et nécessaire pour passer au monde décimal !
Inconvénients	Conversions nécessaires en DCB pour être comprises par l'utilisateur.	Les calculs seraient plus lents, (par algorithmes mélangeant les puissances de 2 et de 10.

4. NOMBRE NEGATIFS EN CODE COMPLEMENT A 2

(complément vrai)

4.1. But : le calcul algébrique !

Pour effectuer les calcul de par exemple 6 + 3 ou 6 + (-3) vous ne pouvez le faire qu'en réfléchissant : en effet le première addition se fait bien par une addition, mais la seconde se fait en réalité par une soustraction !

Donc perte de temps pour une machine qui ne pense pas et qui doit travailler le plus vote possible ! Il lui faut en effet exécuter une instruction de test avant d'effectuer l'opération, et en outre 2 opérateurs sont nécessaires: l'additionneur et le soustracteur.

On cherche un code représentant directement le nombre –3 qui évitera donc à l'ordinateur de réfléchir, et donc à ne faire que des additions !

4.2. En Binaire

Il faut se fixer un nombre N de bits .

Ex sur 4 bits :

0110 - 0011 = 0110 +(10000 - 0011) - 10000 = 0110 + (0111 - 0011) + 1 - 10000

	Complément VRAI Complément à 2^N **Complément à 2**
	Complément RESTREINT Complément à 2^{N-1} , appelé "Complément à 1"

Pour calculer:	0110 (6 décimal)	On calcule:	0110	
	- 0011 (3 décimal)		+ 1100	
	?		+ 1	
			1 **0011**	3 en ignorant la retenue

Le **complément restreint** s'obtient juste par **inversion des bits** .

0011 donne 1100

Le **complément vrai s' obtient ensuite en rajoutant 1.**

le nombre 1100 + 1 = 1101 peut donc représenter le nombre - 0011.

Un nombre négatif se représente donc par le complément VRAI du nombre positif correspondant.

A - B = A + (-B) = A + Complément vrai de B = A + 2^n - B (modulo 2^n)

On peut noter CR = complément restreint

CV = complément VRAI, ou « complément à 2 » , = CR +1 .

On est amené a effectuer des calculs algébriques en utilisant ce code "Binaire signé mode complément à 2 ". C'est mâgique…

4.3. Code Binaire signé, mode Complément vrai (ou complément à 2)

4.3.1. Entiers: Exemple sur 3 bits

0 11	+3
0 10	+2
0 01	+1
0 00	0
1 11	-1
1 10	-2
1 01	-3
1 00	-4

On peut coder sur 3 bits,
2^3 = 8 nombres de -4 à +3

Attention, les bits à droite du bit de signe ne sont pas la valeur absolue !!

bit de signe.

La **dynamique** sur N bits (pour des entiers) est de

2^N nombres de -2^{N-1} à $2^{N-1} - 1$

Sur 4 bits, on aurait 16 nombres de -8 à + 7
Sur 8 bits, on aurait 256 nombres de - 128 à + 127.

On notera le dissymétrie (une valeur possible de plus en négatif)

Le nombre -1 s'écrit	Sur 3 bits	111
	Sur 4 bits	1111
	Sur 8 bits	11111111

Pour coder un même nombre avec plus de bits, on effectue une "**EXTENSION DE SIGNE "**: On rajoute des 0 en tête pour un nombre positif, des 1 pour un nombre négatif.

Le code est pondéré :

Sur 4 bits

1	0	0	0	=	-8
|	|	|	|		
-8	4	2	1		

bit de signe.

Si x <0 code_x = 2^n -x
-x = code_x - 2^n
décalage de -2^n par rapport à code_x, entier non signé

Le bit de signe a pour poids -2^{N-1}

On peut utiliser directement cette pondération pour convertir :
Exemple 10011000 = -128 + 16 + 8 = - 104

En **écriture hexadécimale**, on peut écrite ce nombre : 98 hexa . Attention, on ne peut pas le convertir directement de l'hexa en décimal par les puissances de 16, il faut passer par le binaire. Seul le poids fort à un poids négatif .

Une calculette en mode hexadécimal travaille le plus souvent en entier signé mode complément à 2. Un nombre est négatif si le poids fort est à 1, donc si le chiffre hexa **le plus à gauche** est supérieur ou égal à 8 de 8 à F, le plus à gauche voulant dire le niéme chiffre pour une calculette n chiffres. Donc pour une calculette 10 chiffres, FE donnera 254 tandis que FFFFFFFFFE donnera -2.

4.3.2. nombres non entiers.

pas de problème : 10,100000 = -2 + 0,5 = -1,5
01,100000 =1,5

Il suffit de jongler avec la virgule

4.3.3. Avantage du code "complément à 2".

-Rapidité des calculs, le code étant pondéré, il permet le calcul algébrique pour les additions et soustractions, par contre pour les multiplications et divisions, l'algorithme de calcul est différent !

-La suite binaire signée (mode complément vrai) est facile à obtenir à partir de la suite des entiers non signés (code binaire naturel): simple inversion du bit de signe.

Exemple sur 4 bits:

1111	(15)		0111	(+7)
1110			0110	
1101		Par INVERSION	0101	
1100		du bit de signe :	0100	
1011			0011	
1010			0010	
1001			0001	
1000	(8)		----------0000----------	(0)------
0111			1111	
0110			1110	
0101			1101	
0100			1100	
0011			1011	
0010			1010	
0001			1001	(- 7)
0000	(0)		1000	(- 8)
Code binaire Naturel			Code binaire signé, mode complément vrai (ou à 2)	

5. NOMBRES NEGATIFS EN CODE SIGNE ET VALEUR ABSOLUE

La valeur absolue d'un nombre en binaire signé, mode complément à 2, n'est pas directement visible pour les nombres négatifs:

Exemple sur 4 bits: 0111 (+7) valeur absolue 7
1001 (- 7) valeur absolue ?

Pour la trouver, il suffit, pour un nombre négatif, de prendre le complément vrai de 1001 soit 0110 + 1 = 0111, on trouve 7.

Le code signe et valeur absolue est donc un code intermédiaire obligatoire .

Pour les calculs, ce code se prête mal aux additions - soustractions (code non pondéré entièrement). Il est cependant souvent utilisé pour les multiplications et divisions (le signe est alors traité à part).

5.1. Bit de signe en tête du nombre

Un nombre sur **N** bits, peut se coder avec **N -1 bits pour la valeur absolue**, **et un bit à gauche représentant le signe** (0 pour + , 1 pour -)

Exemple sur 4 bits - 7 = 1 001 en mode complément à 2
- 7 = 1 111 en mode valeur absolue signe

sur 8 bits - 7 = 1 1111001 en complément à 2
- 7 = 1 0000111 en mode valeur absolue-signe
|
bit de signe

En mode signe et valeur absolue, le bit de signe n'a pas de poids. Ce bit à un signifie juste le signe moins.

5.2. Autres codes pour le signe

On peut coder le signe sur 1 octet ou sur 4 bits :

Exemple : 1, 2 , 3 ...etc octets pour la valeur absolue et **un octet pour le signe** (par exemple 00 pour + et FF pour le -). Le signe peut alors être mis à part dans une mémoire de 8 bits.

Alors : - 7, sur 2 octets de valeur absolue peut s'écrire 0007_{hexa}, octet de signe FF .

6. ARITHMETIQUE BINAIRE (VIRGULE FIXE)

6.1. Nombres NON SIGNES

6.1.1. Addition

Comme en décimal, mais 1 + 1 donne déjà une retenue:

0+0 = 0
0+1 = 1
1+0= 1
1+1 = 0 retenue de 1

On doit prendre garde au débordement (dépassement de capacité, résultat trop grand pour le nombre de bits utilisé).:

Exemple sur 4 bits :

```
        0111   (7)
+       0011   (3)
        1 1 1   retenues dans l' addition de chaque bit
        1010   (10)        pas de débordement, le résultat tient sur 4 bits .

        0111 | (7)
        1010 | (10)
        1 1  |  retenues
   1    0001 | (17 > 15 ) dépassement de capacité sur 4 bits .
   |
```

bit "dépassement" (**retenue du poids fort**)
souvent nommé **bit C ou Carry** sur 8, ou 16 bits , ou plus .

Si le bit C est à un, le résultat est juste avec ce bit en plus , il faut le prendre en compte ou éventuellement prévoir un octet de plus (la taille des opérandes dans un ordinateur étant multiple de 8).

Pour des nombres non entiers, aucun changement, les nombres doivent être alignés sur la virgule avant addition (comme en décimal)

```
01,11  + 0,11   :    01 ,1 1 (1,75)
                     00, 1 1 (0,75)
                     10, 1 0 (2,5)
```

6.1.2. Soustraction

les règles ressemblent au calcul en décimal:

```
    | 10111 |   (23)        Mais      |  1011 |  (11)
-   | 1011  |  -(11)                  | -1100 | -(12)
    | 01100 |   (12)                1 |  1111 |
```

dépassement de capacité indiqué par le bit C .
(le résultat est négatif, il faudrait être en binaire signé)
(avec au moins un bit de plus)

6.1.3. Multiplication

```
    10111        23
    01011        11
    10111
   10111
  10111.
 0011111101   soi t  253   ( 10 bits au maximum pour un produit 5 bits par 5 bits)
```

6.1.4. Division

```
46/5  101110      101 |      .        On peut s' arrêter à l'entier ou
     -101                             continuer jusqu' a la précision
      000110      1001,001            souhaitée
       -101
       1,000
       -,101
        ,011      Soit   46/5  = 9,125 + 0,375/5
```

L'algorithme de division, comme celui de multiplication peut être câblé (machines ultra rapides) ou programmé (calculatrices simples de poche) .

6.2. Nombres SIGNES, en COMPLEMENT A 2

Seuls l'addition et la soustraction peuvent s'effectuer comme pour les nombres toujours positifs .

Pour la multiplication et la division , des algorithmes spéciaux existent , sinon, on peut traiter le signe à part, en passant par les codes valeur absolue-signe.

Débordement en signé : il se nomme l'OVERFLOW, bit **V .**

Prenons les deux cas de débordement (sur 4 bits, on ne peut normalement écrire que de -8 à + 7) :

	5		0101	
+	3		0011	
	8		1000	faux , bit de signe détruit par une retenue
	-5		1011	
+	-4		1100	
	-9	1	0111	faux, le bit de signe devrait être 1 .

Effectuons l' addition A + B = R . Soit SA et SB et SR les bits de signe de A,B et R.

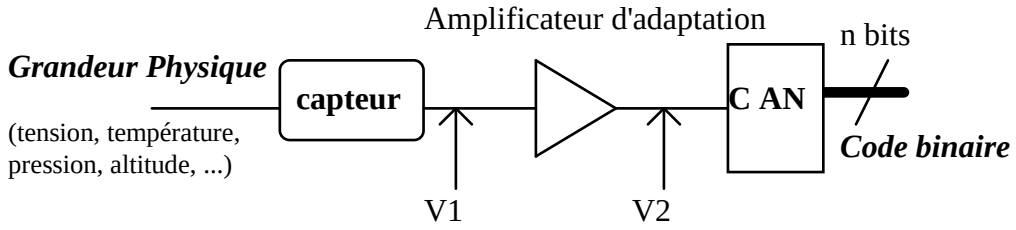
Les deux cas ci dessus de débordement fournissent une équation possible pour le bit V:

$$V = SA.SB.\overline{SR} + \overline{SA}.\overline{SB}.SR$$

Un débordement apporte toujours de fortes erreurs de calculs, il faut utiliser assez de bits pour coder toutes les valeurs utilisées.

Si V est à un, le résultat est juste avec un bit de plus, le bit C, qui devient alors bit de signe . Une extension de signe est nécessaire pour fournir le résultat avec davantage de bits : 0111 = -16 + 7 = -9 , résultat correct . Une extension de signe fournirait 11110111 = -9 sur un octet .

7. CODAGE D'UNE GRANDEUR PHYSIQUE



Selon le capteur utilisé, V1 peut varier de seulement quelques mv, et autour d'une tension moyenne parfois non nulle. Un ampli d'adaptation est indispensable pour faire travailler le convertisseur Analogique-numérique (CNA) dans la plage de tension prévue par le constructeur (par exemple 0 à 10 en « mode monopolaire », ou -5 à +5 en « bipolaire »).

7.1. Codage proportionnel

7.1.1. Principe

Nous étudierons tout d'abord la cas ou on peut faire correspondre le zéro du code avec le zéro de la grandeur physique .

On cherche une relation

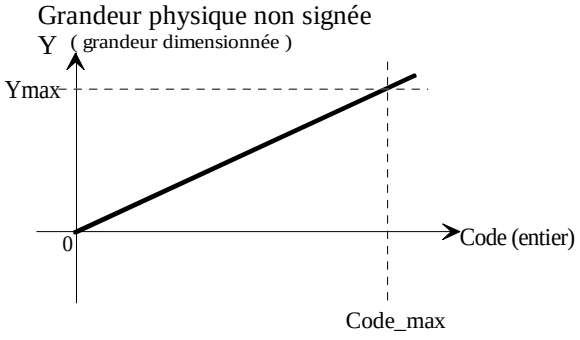
Grandeur physique = k . code

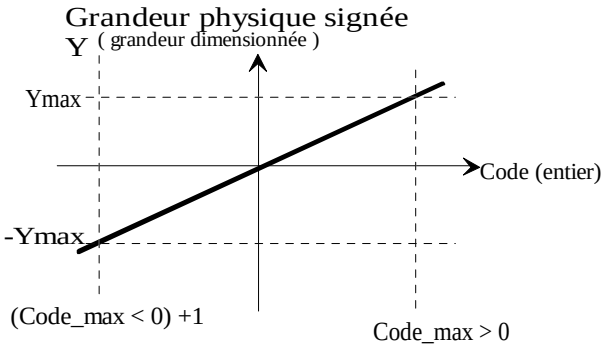
(exprimée avec son unité) (code binaire lu en entier)

(le cas général serait k1.code + k2 : relation linéaire quelconque**)**

k est donc le pas de quantification exprimé en unité de la grandeur physique (par exemple 0,23mm ou 0,077°C .....)

On peut travailler en signé (mode complément vrai) ou en non signé. (par exemple, une température peut aller de 0 à 100°C , un angle de -45° à +45 °.





On fait donc correspondre:

-le **zéro du code** avec le **zéro de la grandeur**

-le **maximum positif du code** avec le **maximum positif de la grandeur**.

on obtient k par : $$k = \frac{Max\_grandeur}{Max\_Code}$$

On prend un nombre N de bits permettant d' obtenir un pas plus petit ou égal à celui demandé, en fonction de la précision absolue souhaitée .

On rappelle que : **Précision = ± 1/2 pas de quantification**

7.1.2. Exemples :

1) On veut coder un angle de 0 à 90° avec une précision meilleur que ± 0,1° .

La grandeur est non signée, on choisit le code binaire pour les **entiers non signés.**

le pas de quantification doit être inférieur à 2.0,1 = 0,2° .

Il faut au moins 90/0,2 + 1 = 451 valeurs (le 0 compris) . Avec 8 bits, on ne code que 256 valeurs, il faudra donc 9 bits .

On fait correspondre : grandeur code

90°	2^N-1 (code max) soit 511 sur 9 bits
-	-
0°	0 (zéro)

On aura ainsi 512 valeurs, et le pas de quantification (k) sera alors :

k = 90/511 = 0,176 °C

On aura la relation **angle (°) = 0,176.code**

La précision finale sera de ±0,176/2 = ±0,088° (meilleure que 0,1°)

Remarque : le nombre de chiffres décimaux significatifs à conserver pour k , donc le nombre de bits nécessaires pour son codage , est évidemment fonction de la précision voulue sur l'angle, nous étudierons ce problème plus loin).

2) On veut coder un angle de -45° à 45° avec une précision meilleur que ± 0,05° .

On choisit ici un **code entiers signés** .

Le pas doit être inférieur à 0,1° . Il faudra (45 - (-45))/0,1 + 1 = 901 valeurs . Donc N = 10 bits .

On fait correspondre: grandeur code

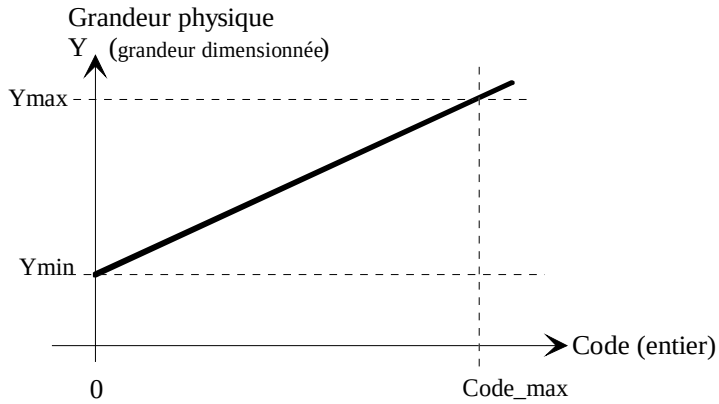
+45	2^{N-1}-1 (code max positif) soit 511 sur 10 bits
-	-
0°	0 (zéro)
-	-
-45	-2^{N-1+1} (code max négatif +1) soit -512 sur 10 bits

D' ou k = 45/511 = 0,0881° , et **Angle (°) = 0,0881.code**

Précision = ±0,0881/2 = ± 0,044 °.

7.2. Cas général linéaire.

7.2.1. Principe



La grandeur (**Y**) peut être **signée ou non**, *on peut choisir un code non signé* (un code signé n'est pas utile, le signe du code n'étant pas toujours celui de la grandeur physique).

Pour coder cette grandeur physique **Y**, variant par exemple de **Ymin** à **Ymax** avec la meilleur précision possible, il faut un codage **Y = K.Code + K'**

On trouve K et K' en faisant correspondre le 0 du code avec Ymin, et le Code_max avec Ymax.

Le nombre de bits nécessaire s'obtient en cherchant le nombre de pas minimum entre Ymin et Ymax.

7.2.2. Exemple

On veut coder une **distance Y variant de 1 à 5,5 m** , avec une précision absolue supérieure **à ± 5mm**.

Le pas sur Y est de 10mm, il faut donc au minimum $\frac{5,5-1}{0,01}+1$ = 451 pas . Donc au minimum 9 bits.

Pour un codage sur **10 bits** (circuits standards), le pas sera de $\frac{5,5-1}{1023}$ = 4,39mm et la **précision maximale de ± 2,2mm**, donc mieux que prévu.

On a les relations: pour Ymin: 1= K.0 + K'
Ymax: 5,5= K.1023+ K'

D'ou K'=1m et K= 0,004398826979 m On conserve pour l'instant visiblement plus de chiffres significatifs que nécessaire !!.

Donc **Y (en mètres) = 0,00439882697.Code +1**

7.2.3. Précision suffisante de K (et K'). Nombre de bits pour les coder.

7.2.3.1.Principe

Ce qui suit est valable bien évidemment pour le codage proportionnel de départ, prenons l'exemple précédent plus général.

Si K et K' ne sont pas codés exactement, ils introduisent une erreur:

dY = K.dCode + Code.dK + dK'.

dCode = 0,5 (valeur arrondie à l'entier le plus proche, en effet, les circuits "CAN" ou convertisseurs analogique-numérique fournissent le code à 1/2 LSB près)

Cette erreur est maximale pour Code_max et vaut:

dYmax=K.O,5 + Code_max.dK + dK'

Pour assurer la précision max possible du CNA soit ±K/2 (moitié du pas), il faut bien sur avoir dK=dK'=0 ce qui n'est pas souvent possible si K et K' ne sont pas entiers! (il faut parfois un très grand nombre de bits !).

Avec une tolérance légèrement plus grande, on va pouvoir limiter le nombre de bits nécessaires:

On choisit dK' (qui est souvent nul si K' est un entier), et on trouve dK par:

$$dK = \frac{dY\max - dK' - 0,5.K}{Code\_\max}$$

On en déduit alors aisément le nombre de bits nécessaires pour coder K (en virgule fixe ou en virgule flottante). (On ne doit pas trouver dK <0 , ceci signifierait que l'on veut une précision supérieure à ce que peut donner le convertisseur !!).

7.2.3.2.Application à l'exemple numérique précédent:

a) <u>calcul de Y dans l'unité (ici le mètre) prévu au départ</u>.

K'=1 donc dK'=0.

Pour avoir une précision sur Y de ±2,2mm, il faudrait mathématiquement coder K avec un nombre infini de bits!. Si on se borne à celle de départ soit dYmax = ±5mm, on en déduit:

$$dK = \frac{0,005 - 0 - 0,5 * 0,004398826}{1023} \approx 2,7.10^{-6}$$

On peut donc limiter K à **7 chiffres après la virgule** soit K ≃ 0,0043988 m (arrondi au plus près). Cette précision semble élevée.

b) Calcul de Y dans une autre unité

On remarque dans cet exemple que les deux chiffres après la virgule sont des zéros, si on se servait de 100.K au lieu de K, on n'écrirait plus que 5 chiffres après la virgule !!.

Ceci revient à coder Y en cm , en effet:

Y (en cm) = 0,439882697.Code + 100

K' = 100 entier à deux chiffres.

On se contente de dY=5mm = 0,5cm. Donc $dK=2,7.10^{-4}$. On peut donc limiter K à **5 chiffres après la virgule** soit K ≃ 0,43988 m

c) Codage de K en virgule flottante

C'est le cas de l'acquisition d'une grandeur physique par un ordinateur, et de programmes travaillant en flottant (le plus usuel sur des nombres non entiers).

On peut choisir d'exprimer Y en m ou en cm, cela ne change rien car

$0,0043988 = 0,43988.10^{-2}$ et $0,43988=0,43988.10^{0}$, le nombre de chiffres significatifs de la mantisse est le même , soit 5.

On choisit la simple précision (qui permet déjà une grande précision et dynamique), ou sinon la double précision.

d) Codage de K en virgule fixe

C'est le cas des automatismes opérant sur des grandeurs physiques, et travaillant en virgule fixe surtout pour des problèmes de vitesse d'exécution et de côut.

Pour coder K dans le **premier cas**, **(Y en m)** ,il faudra n bits tel que $2^{-n} \le 2,7.10^{-6}$ donc $n \ge \frac{6 * \log(10) - \log(2,7)}{\log(2)}$ = 18,5 soit **19 bits**. (ce qui est beaucoup, on cherche à travailler avec 8 ou 16 bits maximum).

Dans le **second cas, (Y en cm)**, $2^{-n} \le 2,7.10^{-4}$ et $n \ge \frac{4 * \log(10) - \log(2,7)}{\log(2)}$ = 11,8 soit **12 bits** . On voit donc l'intérêt du choix de l'unité. (à ce stade du calcul).

---

8. AUTRES CODES

Un code est par définition une correspondance biunivoque entre un ensemble de symboles (en binaire le 0 et le 1) , et un ensemble d'objets (lettres, chiffres, instructions, valeurs numériques ...).

Cette correspondance, à priori arbitraire, est conçue pour faciliter le traitement de tel ou tel type de problème .

8.1. Codes Arithmétiques

Ce sont les codes déjà étudiés, permettant les calculs.

8.2. Codes Alphanumériques : le code ASCII

Ils n'ont aucune propriété arithmétique directe. Ils servent à représenter des "caractères" (chiffres, lettres....)

Un standard est le **Code ASCII** : Il permet de représenter en 7 (ou 8) bits toutes les touches d' une machine à écrire , y compris des touches "commandes " tels que les CTRL... et ALT.... , "Pomme" sur les Macintosh ...

Le code n' est pas tout à fait quelconque:

1) chiffres de 0 à 9 : Le quartet bas est égal à la valeur binaire du chiffre

Le quartet haut est 0011 (3 hexa)

Exemple le caractère "7" donne le code ASCII 37 hexa (0011 0111 binaire).

On peut ainsi facilement passer du code ASCII du chiffre à sa valeur numérique et vice versa.

2) **lettres** . Les codes sont placés par ordre croissant, dans le même ordre que l' alphabet, pour faciliter les classements(il suffit de comparer les codes ASCII).

Exemple: A,B,C,D.. donnent 40,41,42,43 ...(hexa).

3) **Utilisation** : Ne pas confondre le code binaire du 9 (09 hexa) et celui du caractère "9" (39 hexa) !!! .

On se sert des codes ASCII pour

-l'écriture d'un langage (assembleur ou évolué), ou d'un texte quelconque

-la transmission vers une imprimante, vers un écran de visualisation

-les stockages sur disques.

4) **Traitement de texte et éditeur de Texte**

Editeur de texte (Exemple : NotePad.exe sur les PC) Ecriture de simples caractères ASCII, (espace et saut de ligne compris) aucune autre mise en page, aucun style de caractères (gras, italique …)

Traitement de texte (Exemple : Word) Permet toutes les mises en page possibles, avec tableaux, images … (ou presque ..) Le fichier contient les caractères de texte, mais un grand nombre d'autres octets de mise en page et d'information.

Editeur Hexadécimal (exemple Hexedit.exe pour PC) : permet de lire les octets d'un fichier quelconque en Hexadécimal. Si les octets correspondent par moment à des caractères ASCII, ceux ci sont affichées en clair (dans une colonne de droite généralement)

Codes ASCCI de base sur 7 bits
Et iso latin sur 8 bits.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
1	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del
8	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
9	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
a		¡	ć	£	ď	ě	ę	ğ	Í	©	ł			ŋ	ő	
b	°	±	²	³	ť	μ	¶	ů	ÿ	ı	ž	ż	ij	¡	¿	¿
c	À	Á	Â	Ã	Ä	Å	Æ	Ç	É	È	Ê	Ë	Ì	Í	Î	Ï
d	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Œ	Ø	Ù	Ú	Û	Ü	Ý	Þ	β
e	à	á	â	ã	ä	ȧ	æ	ç	è	é	ê	ë	ì	í	î	ï
f	ǒ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Les 32 premiers codes sont utilisés comme caractères de contrôle pour représenter, par exemple, une fin de ligne ou une tabulation, un arrêt transmission (par touches Contrôle truc ou machin …), la touche Escape ….

Le code ASCII ne contient pas de caractères accentués et il a été complété par le code ISO-8859-1 (ou Latin 1). Ce n’est hélas pas le seul. Les 128 premiers caractères correspondent au code ASCII, les 128 suivants aux caractères accentués et caractères spéciaux (voir les 8 dernières lignes du tableau).

8.3. Codes d'Instructions

1) **Programme 'objet"**, nommé aussi **"code machine "** , **directement exécutable** . C'est le seul langage que peut comprendre, et exécuter directement un microprocesseur d'ordinateur.

Tout langage évolué (programme source) est obligatoirement transcris en code objet , avant exécution .

Un code, exemple 86 , 2C (hexa) correspond à un simple chargement d'un octet, une simple addition sur un ou deux octets ....

2) **Exemple de langage ou de code :**

Langage C	Langage assembleur	Code Objet
Langage évolué indépendant des machines (sauf instructions systèmes) **Texte (ASCII)**	Propre de base, rudimentaire, spécifique au type de processeur de la machine. **Texte (ASCII)**	**Binaire pur**, exécutable directement par le processeur de l'ordinateur. Vaguement lisible avec un éditeur hexa (en restant très ésotérique ..)

8.4. Codes détecteurs d'erreurs

Ils permettent de détecter une erreur lors de la transmission d'information, ou lors des écritures lectures sur disquettes ou disques durs. Certains codes permettent même de corriger automatiquement certains erreurs.

a) **principe du " bit de parité"** .

Soit un nombre de N bits, la "parité" P du nombre (sens logique du terme) est égale à la parité du nombre de 1: P= 0 pair, P=1 impair.

exemple pour un code ASCII sur 7 bits : 1001001 parité impaire (trois 1).

On ajoute en tête un bit pour avoir toujours un octet de parité pair.

on obtient 1 1001001 .

|

bit de parité

A la réception, ou à la lecture du nombre, si un octet à une parité impaire, un bit ou un nombre impair de bits est erroné. Le programme redemande alors l'émission ou la relecture de l'octet .

Le cas le plus fréquent est le cas d'erreurs isolées (un bit erroné sur 8 ou 16 bits) , cette méthode simple suffit alors. Pour une coupure de ligne, des centaines de bits peuvent être faux, mais il existe d' autres techniques pour le détecter.

b) **Codes complexes** : Pour des liaisons ou stockages performants, et à haute fiabilité, des codes sophistiqués ont été étudiés, et certains peuvent corriger automatiquement 1 ou 2 erreurs successives.

8.5. Codes Gray

Sans état parasite transitoire .

Un seul bit varie à chaque fois d'un code au suivant (propriété d'adjacence) Le cas classique d'utilisation sur 2 bits est l'emploi des diagrammes de Karnaugh en logique.

a) Construction générale du code . Pour un nombre de bits >2 , il est pratique d'utiliser la symétrie :

On part de	0 , on recopie par rapport à un premier axe :		0
	1		1
			1
			0

Pour différencier les deux code supérieurs des deux codes inférieurs, on ajoute un bit (0 en haut, et 1 en bas) ,et on continue avec un autre axe.

00
01
11
. 10 . etc

on obtient	0000	(valeur décimale)	0
la	0001		1
suite	0011		3
"code Gray"	0010		2
sur	0110		6
4	0111		7
bits:	0101		5
	0100		4
	1100		12
	1101		13
	1111		15
	1110		14
	1010		10
	1011		11
	1001		9
	1000		8

Le code est circulaire : on passe de la dernière valeur à la première en ne changeant qu' un bit.

Le code n'est pas pondéré ! : (la suite de valeurs décimales n' a aucun sens)

Pour passer d'un code binaire naturel à un code Gray, il faut faire un transcodage, (logiciel ou matériel) .

b) Applications du code: (en dehors des diagrammes de Karnaugh)

-Elimination de tout parasite entre deux états successifs.

-Fabrication de pistes électriques pour mesurer une position longitudinale ou une rotation . Un code binaire peut être dessiné sur du cuivre, et des balais (conducteurs de l'électricité) (un par bit) permettent de lire ce code .

Dans un code normal la transition de 0011 à 0100 peut fournir un parasite 0111 avant de fournir vraiment 0100 (le bit de poids 2 passant en premier à 1).

Aucun parasite n'est à craindre dans le cas d'une suite de Gray .

9. INTRODUCTIONS AUX MEMOIRES

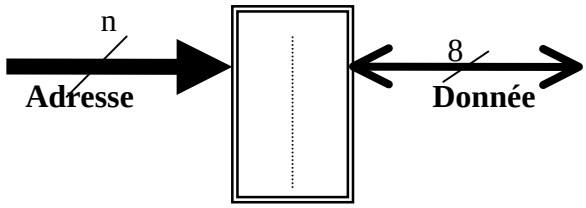
Nous ne voyons ici qu'une très brève introduction !

On veut stocker quelque part des codes aussi divers que du texte, des valeurs numériques, un programme (en langage source C, assembleur, ou directement le code d'instruction). L'ordinateur comporte évidemment un « processeur », sorte de chef d'orchestre, qui gère son fonctionnement, et il doit pouvoir accéder le plus rapidement possible à ces codes pour lire ou écrire une information existante ou nouvelle.

Le circuit permettant ceci se nomme « la **mémoire vive de l'ordinateur** », de capacité pouvant aller de quelque k octets (simples appareils electroniques) à des Méga octets.

L'information élémentaire est le plus souvent l'octet (par exemple un caractère égal un octet). Les octets sont placés successivement dans la zone mémoire qui est en fait comme une série de « tiroirs » que l'on peut numéroter.

On peut représenter une mémoire par le dessin ci contre.



Le processeur accède à chaque octet, que l'on peut nommer « donnée », en envoyant une « adresse » (qui est le numéro du tiroir !) il peut écrire ou lire la donnée.

Adresse et donnée sont en binaire, que l'on écrit évidemment en hexa pour simplifier. Une donnée de 16 ou 32 bits sera mise en mémoire sous la forme de 2 ou 4 octets successifs (que l'ordinateur peut lire en même temps ou suivant une séquence selon les architecture).

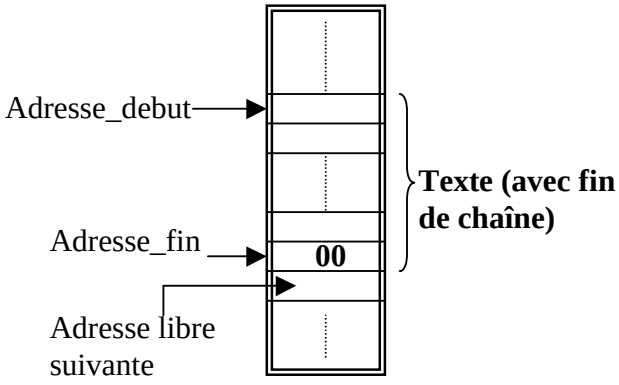
Pour des raisons physiques évidentes (adresse en binaire) la taille max d'une mémoire est une puissance de deux : **Taille max = 2^n**

(Par exemple pour 16 bits d'adresses, on aura au maximum 65536 octets)

Remarque : on peut aussi avoir un groupement de base qui n'est pas l'octet, mais le « mot » de 16 bits, à chaque adresse se trouve alors directement 16 bits de donnée.

Exemple de calcul de taille de zone mémoire :

On veut stocker une « **chaîne de caractères** » :



Une chaîne de caractère est formée des caractères ASCII correspondant au texte, et est terminée par un caractère de « fin de chaîne » (souvent le code binaire 00). La chaîne débute à la position adresse_début, et se termine à adresse_fin, on peut calculer sa taille, le problème peut être donnée dans l'ordre inverse.

Exemple de chaîne de caractères :

Si il n'y a pas de solution, c'est qu'il n'y pas de problème

Elle comprend 60 caractères (sauf erreur …. !) espace, virgule, apostrophes compris, plus un caractère invisible de fin de chaîne, donc sa **taille** est de **61 octets.**

On peut écrire la formule:

taille = adresse_fin – adresse_début +1

L'adresse libre suivante est donc :

Adresse libre suivante = adresse_début + taille

Si la chaîne débute à l'adresse C000, l'adresse libre suivante sera :
C000h + 61d = C000h + 3Dh = C03D.

Il est souvent nécessaire de faire de tels calculs, (ou de programmer des instructions qui les effectuent) car si ne on réserve pas assez de place pour les données que l'on veut stocker, des données importantes peuvent être détruites !

Très important : Lorsque l'on réserve la place minimale pour une chaîne de caractère en mémoire, ne jamais oublier son caractère fin de chaîne, sinon la chaîne se poursuit sur d'autres caractères si la position mémoire suivante est aussi une chaîne, ou sinon sur n'importe quoi (valeur numérique, code quelconque ! !), avec des affichages incohérents. Souvent , on réserve une place plus grande que la plus grande des chaînes que l'on manipulera.

10. ENTREES - SORTIES DE VALEURS DECIMALES

Nous allons tout en revoyant l'usage des principaux codes, voir l'important travail effectué par un langage évolué lors de la manipulation de grandeurs décimales dans les ordres apparemment très simples tels que PRINT ou INPUT ...

10.1.Sorties de valeurs décimales sur une imprimante.

10.1.1.Entiers

Après un calcul, un nombre considéré comme entier se trouve en mémoire en code binaire signé, on veut le sortir sur une imprimante ou un écran (ordre PRINT en langage évolué)

Pour imprimer la valeur décimale de 82hexa (= -126 décimal) par exemple, il devra envoyer le code ASCII du signe moins, suivi des codes ASCII des chiffres 1 2 et 6. D'ou les étapes suivantes:

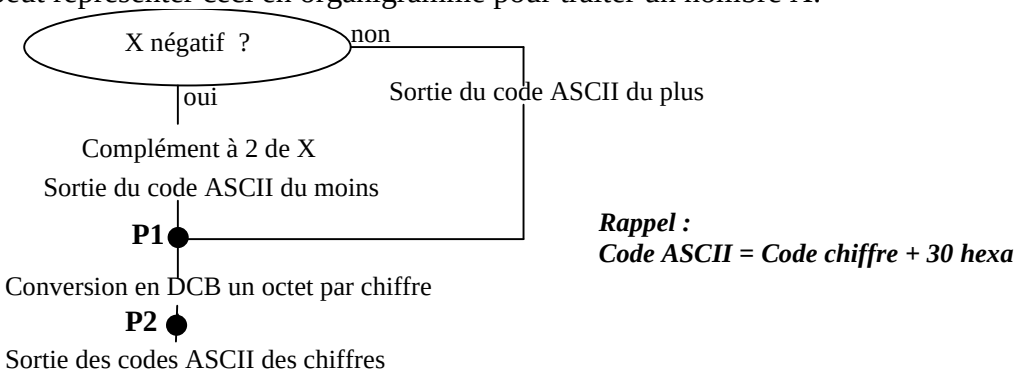
-Conversion en code valeur absolue-signe (si le bit de signe est à 1, prise du complément à 2 pour obtenir la valeur absolue, sortie du code ASCII du signe. (parfois si le signe est + le caractère « plus » n'est pas envoyé).

-Conversion en DCB. (un octet par chiffre, c'est le plus simple).

-Conversion ASCII (on ajoute 30hexa).

-envoi des codes ASCII des chiffres.

On peut représenter ceci en organigramme pour traiter un nombre X:



(exemple: le cas précédent fournira: le code ASCII du moins, suivi des codes 31, 32, et 36 hexa)

Remarque : il s'agit en simplifié, et pour des entiers seulement, le programme lancé par l'instruction C : **printf(**"%*d*"**,valeur) ;**

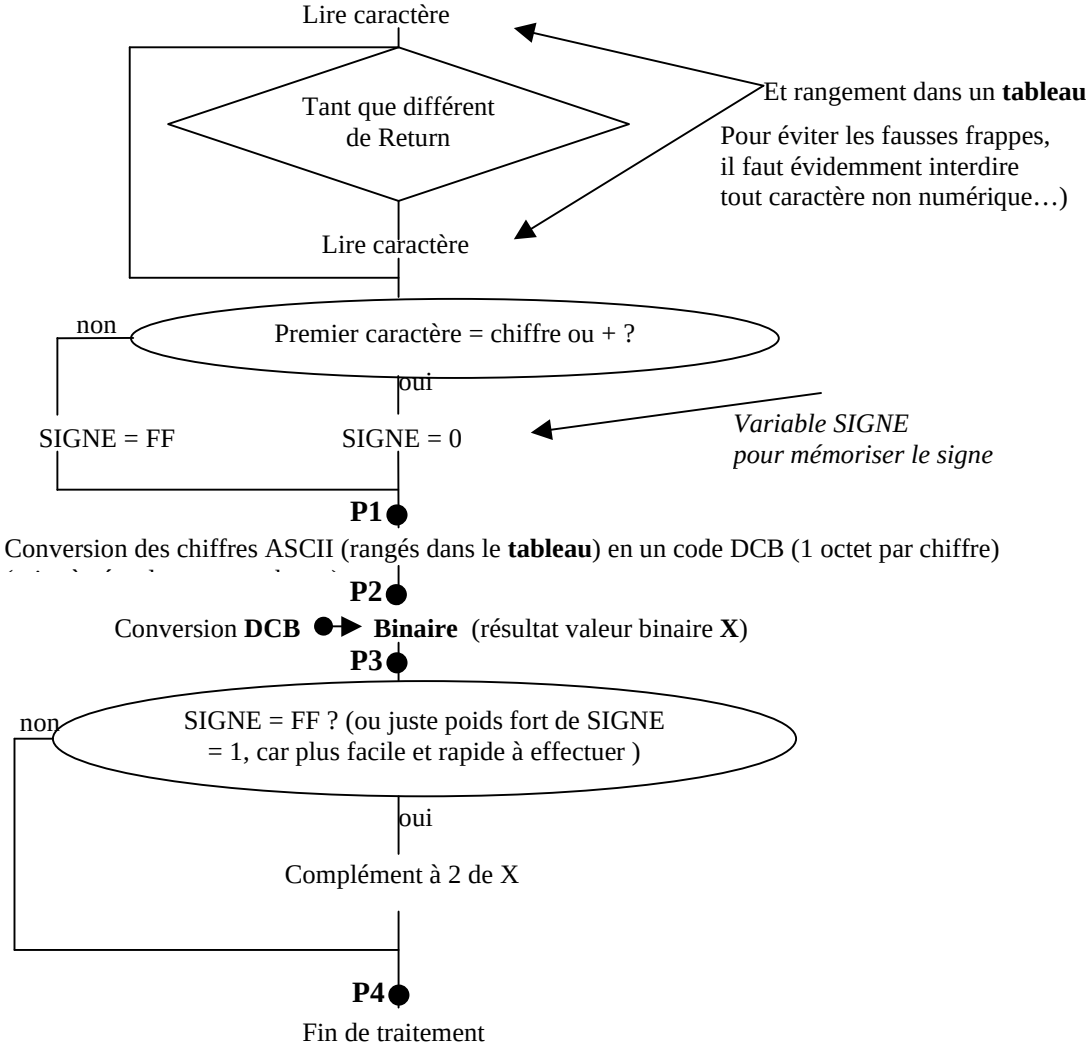
10.1.2.Nombres quelconques

Le format doit être évidemment connu du programme, celui ci traitera à part la partie entière et la partie fractionnaire, il enverra le code ASCII de la virgule pour séparer. Si un format virgule flottant est utilisé, des conversions plus complexes devront s'effectuer, (pour afficher non pas un format flottant puissance de 2, mais un format flottant puissance de 10) .

10.2.Introduction de valeurs décimales (Virgule Fixe) par clavier

10.2.1.Entiers

Remarque : Il s'agit en simplifié du programme de la fonction C : **scanf(&valeur) ;**
On tape sur un nombre quelconque par exemple 134 " return"
Exemple possible d'organigramme :



10.2.2.Nombres quelconques

Si on frappe un nombre quelconque, par exemple 134,45 : la partie entière est convertie en N octets, la partie fractionnaire en M octets, elles seront alors rassemblées dans un mot de N+M octets. Le caractère virgule, une fois frappé, permettra de différencier les deux traitements.

L'introduction de valeurs décimales en "virgule flottante" est beaucoup plus complexe.

Il ne faut évidemment pas oublier de programmer des lignes évitant toutes les erreurs de frappe, ou les frappes fantaisistes …, que donnerait en effet comme code binaire une valeur tapée par exemple une valeur – 456,8,6j ! L'organisme quoique assez bête finalement est loin d'être simple.