ENSEIGNEMENT DE PROMOTION SOCIALE

Cours de

STRUCTURE DES ORDINATEURS

- Codage de l'information -

H. Schyns

Septembre 2005

Sommaire

1. Position du problème

Nous avons l'habitude de traiter les nombres décimaux par groupe de trois chiffres : les unités, les milliers, les millions, les milliards, etc. Chaque séquence de trois chiffres est appelée triade.

L'ordinateur travaille en binaire et traite les nombres par série de huit bits. Un paquet de **huit bits** est appelé **octet ou byte** (*ang.*). Généralement, le terme anglais est préféré.

Suivant la taille du nombre à représenter, on utilise des groupes de 1, 2, 4 ou 8 bytes. Pourquoi pas 3 ou 5 ? Parce que ce ne sont pas des puissances de 2. Un paquet **2 bytes** (16 bits) est appelé "**word**" ou "**mot**"; un paquet de **4 bytes** (32 bits) est appelé "**dword**" ou "**double mot**".

Toute information traitée par un ordinateur est toujours convertie en binaire et stockée dans des bytes.

Un nombre entier, un nombre réel (avec des décimales), un son, une couleur, une instruction de programme, etc. sont toujours représentés par des bytes.

Le processeur qui reçoit un byte à traiter (p.ex. 10110101) est incapable de dire *par lui-même* si ce byte représente un nombre entier, un nombre réel, une instruction, un son, etc.

Par conséquent, chaque fois que l'on demandera au processeur de traiter un byte, on devra aussi lui en donner le type et, très souvent, l'algorithme de traitement. C'est un des rôles de la programmation : tout programme comporte par une section dans laquelle on définit le **type** des données. La représentation de chaque type fait l'objet d'une convention internationale ou d'une convention de langage (p.ex : norme IEEE 754).

2. Les nombres entiers

2.1. Nombres entiers strictement positifs

Un nombre entier strictement positif (*ang.: unsigned integer*), aussi appelé entier non signé ([1]), est codé sur 1, 2 ou 4 bytes selon sa taille. On distingue ainsi les *entiers courts*, les *entiers* et les *entiers longs*.

Le codage s'effectue en binaire (ou en hexadécimal) de la façon exposée plus haut en ajoutant éventuellement des zéros à gauche pour remplir (*ang. padding*) le byte.

Exemple :	91 = 1011011	*(codage binaire)*
	91 = 0101 1011	*(remplissage du byte)*
	91 = $5B_h$	*(forme hexadécimale)*

En programmation, on utilise le type entier non signé pour définir une variable (zone mémoire) dont on sait qu'elle ne contiendra que des valeurs positives telles que le nombre d'habitants d'une ville, le nombre de voitures en circulation, les numéros du tirage du lotto, etc.

Voici un tableau des principaux types d'entiers non signés :

Nom	Taille (byte)	Valeur minimale	Valeur maximale
Entier court non signé *short unsigned integer*	1	0 00_h	255 FF_h
Entier non signé *unsigned integer*	2	0 $00\ 00_h$	65 535 $FF\ FF_h$
Entier long non signé *long unsigned integer*	4	0 $00\ 00\ 00\ 00_h$	4 294 967 295 $FF\ FF\ FF\ FF_h$

2.2. Nombres entiers négatifs (ou positifs)

Un nombre entier négatif (*ang.: signed integer*), aussi appelé entier signé ([2]), est aussi codé sur 1, 2 ou 4 bytes selon sa taille. On distinguera ainsi les *entiers signés courts* (*ang.: short signed integer*), les *entiers signés* et les *entiers signés longs* (*ang.: long signed integer*).

> Comme l'ordinateur ne peut traiter que des 0 et des 1, on est bien obligé de traduire les signes + et - sous forme de 1 et 0.

Comme l'ordinateur traite les données par groupe de 8, 16 ou 32 bits, il n'est pas question d'ajouter un 9ème, 17ème ou 33ème bit pour définir le signe du nombre. Par contre, il est permis de sacrifier l'un des bits du byte afin qu'il représente le signe. On prend la convention suivante (première convention) :

1 Du point de vu mathématique, les entiers non signés (sans signe) correspondent aux nombres entiers naturels, éléments de l'ensemble $\mathbb{N}$.

2 Du point de vu mathématique, les entiers signés (avec signe) correspondent aux nombres entiers éléments de l'ensemble $\mathbb{Z}$. On sait que $\mathbb{N} \subset \mathbb{Z}$.

Dans un byte (ou un nombre de plusieurs bytes) le bit situé à l'extrémité gauche représente le signe (là où on placerait le signe + ou - en décimal)

0 signifie **+**
1 signifie **–**

Dès lors, un byte se décompose en 1 bit de signe et 7 bits qui codent la valeur.

La première idée qui vient à l'esprit quand il s'agit de coder les nombre +91 et -91 est simplement d'ajouter le bit de signe :

+91 =	*0*101 1011	($5B_h$)
–91 =	*1*101 1011	(DB_h)

Ceci pose cependant un certain nombre de problèmes :

- la manière de traiter une addition et une soustraction varie selon que l'on a affaire à deux nombres négatifs, deux positifs ou un positif et un négatif.
- la suite des nombres présente deux valeurs 0 alors que 0 doit être unique :

+0 =	*0*000 0000
–0 =	*1*000 0000

Pour résoudre ces problèmes, on choisit une deuxième convention, appelée "complément à 2" :

Comment coder correctement le nombre –91 ?

Partons du nombre positif :	+91 =	0101 1011
Remplaçons les 0 par des 1 et réciproquement :		1010 0100
Ajoutons 1 :	–91 =	1010 0101

L'algorithme de changement de signe s'applique exactement de la même manière à un nombre négatif; ce qui est particulièrement intéressant :

Partons du nombre négatif :	–91 =	1010 0101
Remplaçons les 0 par des 1 et réciproquement :		0101 1010
Ajoutons 1 :	+91 =	0101 1011

Ceci peut sembler assez tordu mais cette manière de coder les nombres va grandement simplifier le problème de l'addition et de la soustraction. Par exemple, l'addition d'un nombre et de son opposé donne bien 0 :

(report)		*1 1111 111*	
	+91 =	0101 1011	($5B_h$)
	<u>–91 =</u>	<u>1010 0101</u>	($A5_h$)
	0 =	0000 0000	

Lors de l'addition du dernier bit, à gauche, le report de 1 devrait être placé dans un 9ème bit. Comme le byte ne contient que 8 bits, on laisse tomber ce report. On parle de dépassement de capacité (*ang.: overflow*).

Nous pouvons également vérifier que l'oppose de 0 reste 0, grâce à cette astuce de l'overflow :

Partons du 0 "positif" :	+0 =	0000 0000
Remplaçons les 0 par des 1 et réciproquement :		1111 1111
Ajoutons 1 :	–0 =	*(1)*0000 0000

Voici comment se présentent les nombres entiers autour de 0 :

+3 =	0000 0011	(03_h)
+2 =	0000 0010	(02_h)
+1 =	0000 0001	(01_h)
+0 =	0000 0000	(00_h)
–1 =	1111 1111	(FF_h)
–2 =	1111 1110	(FE_h)
–3 =	1111 1101	(FD_h)

Il peut sembler étrange que –1 se traduise par 1111 1111. Nous avons pourtant déjà tous rencontré un cas similaire dans la vie quotidienne :

Imaginons une voiture neuve, qui n'a pas encore roulé. Son compteur kilométrique affiche 000000 km. Montons dans cette voiture et roulons un kilomètre *en marche arrière*. Ceci revient à soustraire un kilomètre de la distance déjà parcourue ou, en d'autres mots, à rouler –1 kilomètre. Au terme de cette distance, le compteur affiche 999999 km. Autrement dit, pour le compteur kilométrique de la voiture

–1 =	999999
–2 =	999998
–3 =	999997

De même, si le compteur affiche 999999 km et que nous roulons un kilomètre en marche avant, nous obtenons un dépassement de capacité et le compteur revient à 000000; le dernier report, celui du chiffre de gauche est ignoré.

Cette comparaison met en évidence le fait que, lors du codage des entiers, les bytes, mots et double mots se comportent comme des compteurs à 8, 16 ou 32 chiffres binaires.

La série des nombres positifs va de 0000 0000 (00_h) à 0111 1111 ($7F_h$), soit de 0 à 127. La série des nombres négatifs va de 1111 1111 (FF_h) à 1000 0000 (80_h), soit de -1 à -128. Il y a donc un nombre de plus du côté des négatifs. Cette dissymétrie existe dans tous les cas, que le nombre soit codé sur 1, 2 ou 4 bytes.

Ceci posera un petit problème lors du changement de signe de la valeur la plus négative

Partons du nombre négatif :	–128 =	1000 0000	(80_h)
Remplaçons les 0 par des 1 et réciproquement :		0111 1111	
Ajoutons 1 :	–128 =	1000 0000	

Donc –(–128) = –128, ce qui est contraire aux règles de l'arithmétique. Les algorithmes utilisent une branche spéciale pour détecter ce cas particulier.

En programmation, on utilise le type entier signé pour définir une variable dont on sait qu'elle contiendra des valeurs entières qui, selon les cas, pourront être positives ou négatives. Par exemple les numéros des étages d'un immeuble, les flux de personnes qui entrent ou sortent d'une zone et, en général, tous les nombres sur lesquels on a l'intention de faire des soustractions.

Nom	Taille (byte)	Valeur minimale	Valeur maximale
Entier court signé *short signed integer*	1	–128 80_h	127 $7F_h$
Entier signé *signed integer*	2	–32 768 $10\ 00_h$	32 767 $7F\ FF_h$
Entier long signé *long signed integer*	4	–2 147 483 648 $10\ 00\ 00\ 00_h$	2 147 483 647 $7F\ FF\ FF\ FF_h$

Un problème fréquent en programmation est le changement de type : un nombre stocké dans un byte est recopié dans un mot de deux bytes. On lui adjoint donc un byte à gauche. Comment faut-il remplir ce byte supplémentaire pour ne pas changer la valeur stockée ?

Soit le nombre *positif* +91 stocké dans un byte :

+91 = 0101 1011 ($5B_h$)

Pour le copier dans une variable qui contient deux bytes, il suffit d'ajouter 0000 0000 à sa gauche :

+91 = 0000 0000 0101 1011 ($00\ 5B_h$)

Soit le nombre *négatif* –91 stocké dans un byte :

–91 = 1010 0101 ($A5_h$)

Si on ajoute 0000 0000 à sa gauche on change le signe et la valeur car, par convention, le premier bit à gauche est toujours le bit de signe.

–91 ≠ 0000 0000 1010 0101 = 165

En fait, il suffit de coller 1111 1111 :

–91 = 1111 1111 1101 1011 ($FF\ A5_h$)

3. Les nombres réels

3.1. La virgule fixe

La première idée qui vient à l'esprit quand on parle de nombres réels est de fixer une fois pour toutes le nombre de chiffres que l'on veut conserver derrière la virgule.

Avantage : une fois ce choix posé, les opérations sur les nombres fractionnaires se ramènent à peu de choses près à des opérations sur des entiers.

Inconvénient : il est impossible de coder des nombres plus petits que la dernière décimale. Si on a décidé de garder six chiffres derrière la virgule il est impossible de stocker un nombre tel que 0, 000 000 567.

Les nombres en virgule fixe ont généralement été abandonnés dans les années 1960. Ils sont cependant encore utilisés dans certaines applications financières ([1]).

3.2. La virgule flottante

Un nombre réel (*ang.: real* ou *float*) est codé sur 4 ou 8 bytes selon le nombre de chiffres significatifs désirés par l'utilisateur ([2]). On distingue ainsi les *réels simples* (*ang.: single real*) et les *réels doubles* (*ang.: double real* ou *double float*).

En programmation, on utilise le type réel dès qu'une variable est susceptible de contenir une partie fractionnaire, c'est-à-dire presque tout le temps : prix d'un article, température, distance parcourue, etc.

3.2.1. Notation scientifique en système décimal

Le concept à la base du réel en virgule flottante est la notation scientifique. Pour écrire un nombre en notation scientifique, on ramène la virgule derrière le premier chiffre significatif et on multiplie par la puissance de 10 adéquate ([3]). Par exemple :

$$579 = 5{,}79 \bullet 100 = 5{,}79 \bullet 10^{2}$$

Plus rapidement, on compte le nombre de rangs dont on déplace la virgule pour la ramener juste derrière les plus grand chiffre significatif :

579,	départ
57,9	1 rang
5,79	**2** rangs

$$579 = 5{,}79 \bullet 10^{\mathbf{2}}$$

On procède de même quand le nombre contient une partie entière et une partie décimale :

$$42\,854{,}9 = 4{,}28549 \bullet 10^{4}$$

1 Il faut cependant être prudent : selon la norme américaine, le type "virgule fixe" utilise 4 décimales et permet de travailler au centième de "cent" près. Ceci n'a pas empèché un hacker fameux de collecter ces fractions de cent et de devenir millionnaire. Forte de cette aventure, la norme EURO impose de travailler avec 6 décimales. Il en résulte que le type "virgule fixe" ou "monétaire" est ***interdit*** dans les applications financières qui utilisent l'Euro.

2 Du point de vu mathématique, les réels sont les éléments de l'ensemble $\mathbb{R}$. On sait que $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$.

3 Par premier chiffre significatif d'un nombre, on entend le plus grand chiffre qui ne soit pas un zéro de position.

Quand le nombre ne contient qu'une partie décimale, on se retrouve avec un exposant de 10 négatif :

$$0{,}00371 = 3{,}71 \bullet 0.001 = 3{,}71 \bullet 10^{-3}$$

Ici, la virgule s'est déplacée de 3 rangs vers les petits chiffres; l'exposant de 10 sera donc négatif.

0,00371	départ
00,0371	-1 rang
000,371	-2 rangs
0003,71	**-3** rangs

$$0{,}00371 = 3{,}71 \bullet 10^{\mathbf{-3}}$$

On en conclut que tout nombre décimal *différent de 0* peut s'écrire sous la forme d'une mantisse constituée des chiffres significatifs et d'une puissance de 10.

3.2.2. Notation scientifique en système binaire et hexadécimal

Le même principe peut être appliqué aux nombres binaires et hexadécimaux mais cette fois, on multiplie la mantisse par une puissance de 2 ou de 16 :

$$1011011{,}10011_b = 1{,}01101110011_b \bullet 2^{\mathbf{6}}$$

$$0{,}00001101011_b = 1{,}101011_b \bullet 2^{\mathbf{-5}}$$

$$A5B{,}98_h = A{,}5B98_h \bullet 16^{\mathbf{2}}$$

$$0{,}0007CF3_h = 7{,}CF3_h \bullet 16^{\mathbf{-4}}$$

A l'inverse, on peut choisir de ramener systématiquement la virgule derrière le plus petit chiffre :

$$1011011{,}10011_b = 101101110011_b \bullet 2^{\mathbf{-5}}$$

$$0{,}00001101011_b = 1101011_b \bullet 2^{\mathbf{-11}}$$

$$A5B{,}98_h = A5B98_h \bullet 16^{\mathbf{-2}}$$

$$0{,}0007CF3_h = 7CF3_h \bullet 16^{\mathbf{-7}}$$

On peut aussi se définir d'office un certain nombre de bits de mantisse, quitte à donner la valeur 0 aux bits non utilisés <u>*à droite*</u>. Si nous décidons de travailler systématiquement avec 20 bits, les nombres binaires ci-dessus deviendront :

$$1011011{,}10011_b = 1011\ 0111\ 0011\ 0000\ 0000_b \bullet 2^{\mathbf{-13}}$$

$$0{,}00001101011_b = 1101\ 0110\ 0000\ 0000\ 0000_b \bullet 2^{\mathbf{-24}}$$

Pour que ce principe reste cohérent, il faut évidemment aussi coder l'exposant dans le système binaire.

3.2.3. Taille du nombre réel

Il nous reste à structurer la mantisse et l'exposant dans un groupe de 8, 16 ou 32 bits (1, 2 ou 4 bytes). Or,

il faut 10 digits binaires pour représenter 3 chiffres décimaux

Par exemple, le plus grand nombre décimal sur 3 chiffres est 999, ce qui se traduit en binaire par un nombre de 10 digits : 11 1110 0111 ([1]).

Or, dans la vie courante, nous utilisons généralement des nombres avec cinq chiffres significatifs. Pensez au total d'un ticket de caisse, à la distance entre deux villes, aux heures et minutes, etc. Autrement dit, pour stocker un nombre avec une précision "habituelle" de cinq chiffres, il nous faut au moins quinze digits binaires sans compter le bit de signe. En ce qui concerne exposants, nous travaillons quotidiennement avec des grandeurs allant des millièmes (10^{-3}, millimètres) à des millions (10^{+6}, lotto). Ceci exige au moins quatre bits. Nous voici donc à 20 bits. Impossible donc de travailler de manière courante avec des nombres réels codés sur 8 ou 16 bits.

Les nombres réels (flottants) sont codés sur 32 bits au moins.

3.2.4. Stucture du nombre réel

Un nombre réel "simple" codé sur 32 bits a la structure suivante :

seee eeee emmm mmmm mmmm mmmm mmmm mmmm

- 1 bit de signe, à l'extrême gauche
- 8 bits d'exposant, à cheval sur deux bytes
- 23 bits de mantisse

Un nombre réel "double" est codé sur 64 bits selon la structure

- 1 bit de signe, à l'extrême gauche
- 11 bits d'exposant, à cheval sur deux bytes
- 52 bits de mantisse

Un nombre réel "long double" est codé sur 80 bits selon la structure

- 1 bit de signe, à l'extrême gauche
- 14 bits d'exposant, à cheval sur deux bytes
- 64 bits de mantisse et un 1 *explicite* dans le 65ème bit

Pour augmenter la précision et accélérer les comparaisons de nombres, les concepteurs du système ont utilisé deux astuces :

- la mantisse n'affiche pas le premier 1.

  Dans un nombre décimal, par définition, le premier chiffre significatif est différent de 0.

  Il en va de même pour un nombre binaire : le premier chiffre significatif est différent de 0; le premier chiffre significatif est donc toujours 1. Puisque c'est toujours le cas, on peut décider de ne pas écrire ce premier 1. On parle alors de 1 "implicite". C'est pourquoi la structure du nombre flottant est souvent notée

  seee eeee e^{1}mmm mmmm mmmm mmmm mmmm mmmm

  La seule exception est le zéro (0.000) qui ne compte *aucun* chiffre significatif. Nous illustrerons ce principe ci-après
- l'exposant est donné avec un excès ou biais (*ang.: bias*).

  On pourrait bien sûr coder les exposants négatifs en complément à 2. Comme la structure dispose de 8 bits d'exposant, on irait de

1 En toute rigueur, la rapport est donné par log(10) / log(2) = 1 / 0.30103 = 3.32

1000 0000 (-128) à 0111 1111 (+127)

Cependant, pour faciliter le tri et la comparaison des nombres, on a décidé de biaiser les exposants. On leur ajoute une valeur constante de manière à les ramener dans la plage de 0 à 255.

Le biais est égal à 127 + nombre de bits de la mantisse, soit 150

3.2.5. Exemple d'encodage

Encodons le nombre 445.9 sous la forme d'un réel à 32 bits.

Transformons d'abord la partie entière en binaire comme expliqué ci avant :

445 = 1 1011 1101

Passons ensuite à la partie décimale. On tombe sur une forme périodique

0,9 = 0,1110 0110 0110 0110 0110 0...

En mettant les deux morceaux ensemble, on a

445,9 = 1 1011 1101,1110 0110 0110 0110 0110 0...

Or, selon la norme, nous avons droit à 1 bit implicite (le premier 1) et 23 bits de mantisse, soit 24 bits au total :

1 1011 1101,1110 0110 0110 011

Pour ramener la virgule à l'extrémité droite, nous devons la déplacer de 15 rangs, soit un facteur 2^{-15},donc un exposant -15

1101 1110 1111 0011 0011 0011•2^{-15}

Réglons le problème de l'exposant. Le biais étant de 127_d et la mantisse comptant 23 bits, la constante à appliquer vaut 150_d, ce qui donne un exposant

$-15_d + 150_d = 135_d = 1000\ 0111_b$

Par ailleurs, dans la mantisse, nous pouvons supprimer le premier 1 significatif.

101 1110 1111 0011 0011 0011

Reste à fixer le bit de signe : 0 puis qu'il s'agit d'un nombre positif

Dès lors, en rassemblant les morceaux, on a

0 1000 0111101 1110 1111 0011 0011 0011

Regroupons finalement les bits par paquets de 4 et transformons en hexadécimal

0100 0011 1101 1110 1111 0011 0011 0011

43 DE F3 33

Le nombre 445.9 est donc codé en mémoire sous la forme 43 DE F3 33

Ce travail d'encodage est fait par le compilateur lorsqu'il traduit le programme et les constantes qui y apparaissent dans un langage intelligible par le processeur. Il est également effectué par le programme lorsqu'il lit les données encodées dans les différents champs des fenêtres de saisie.

3.2.6. Exemple de décodage

Soit le nombre 43 DE F3 33 à transformer dans le système décimal.

Passons à la notation binaire :

43 DE F3 33 = 0100 0011 1101 1110 1111 0011 0011 0011

Isolons

- le bit de signe : 0
- les huit bits d'exposant : 100 0011 1
- les 23 bits de mantisse : 101 1110 1111 0011 0011 0011

Nous savons déjà qu'il s'agit d'un nombre positif.

Décodons l'exposant et soustrayons le biais de 150_d

$$10000111_b = 135_d \text{ ð } 135_d - 150_d = -15_d$$

Le nombre en question s'exprime donc en terme de

$$2^{-15} = 1 / 32\,768$$

Reprenons la mantisse et ajoutons le 1 implicite à gauche

101 1110 1111 0011 0011 0011 **ð** **1**101 1110 1111 0011 0011 0011 = 14 611 251

Le nombre en question est donc

$$\frac{14\,611\,251}{32\,768} = 445{,}899993896484375 \cong 445{,}9$$

On notera avec intérêt qu'il y a une erreur d'arrondi par rapport au nombre initial. C'est assez logique puisque le nombre flottant disposant de 24 digits binaires, sa précision est de

$$24 / 3.322 = 7.2 \text{ chiffres décimaux}$$

En double précision, on dispose de 53 bits de mantisse et la précision devient

$$53 / 3.322 \approx 15.96 \text{ chiffres décimaux}$$

3.2.7. Plages et exceptions

Les nombres codés en virgule flottante vont de

Simple :	$\pm 3{,}4 \cdot 10^{-38}$ à $\pm 3{,}4 \cdot 10^{+38}$	(7 chiffres sign.)
Double :	$\pm 1{,}7 \cdot 10^{-308}$ à $\pm 1{,}7 \cdot 10^{+308}$	(15 chiffres sign.)
Long double :	$\pm 3{,}4 \cdot 10^{-4932}$ à $\pm 3{,}4 \cdot 10^{+4932}$	(19 chiffres sign.)

L'exposant maximum FF_h est réservé pour les situations spéciales :

- Quand la mantisse ne contient que des zéros, on considère qu'il s'agit de l'infini. C'est notamment le cas lors d'une division par 0.

0 11111111 00000…000 = +INF
1 11111111 00000…000 = -INF

- Quand la mantisse est différente de zéro, on considère qu'il ne s'agit pas d'un nombre (Not a Number ou Nan). C'est notamment le cas lors d'une opération illicite telle que la racine carrée d'un nombre négatif.

La valeur 0 est représentée par

0000 0000 0000 0000 0000 0000 0000 0000

4. Les caractères alphabétiques

4.1. Les différents codes

De nombreux codes ont été utilisés pour représenter les caractères :

- le code BAUDOT

  Il utilisait 5 bits (0-31), ce qui s'est très vite révélé insuffisant pour coder les chiffres et lettres ou les minuscules et les majuscules;
- le code EBCDIC

  Introduit par IBM, il utilisait 8 bits, ce qui permettait d'encoder tous les caractères (a-z), les majuscules (A-Z), les chiffres (0-9) et les caractères spéciaux (espace, ponctuation, parenthèses, $, etc);
- le code ASCII (*American Standard, Code for Information Interchange*)

  Issu d'une convention rassemblant plusieurs experts; il s'est peu à peu imposé en raison de son utilisation dans les micro-ordinateurs.

  La première version du code ASCII représentait les caractères sur 7 bits.

  Les 32 premiers (00-1F) représentent des caractères de contrôle utilisés, soit dans des protocoles d'échange d'informations (ENQ, ACK, NAK, …), soit pour le contrôle de certains terminaux (LF : Line Feed, CR : Carriage Return, BS : BackSpace, …).

  Les codes suivants (20-7F), sont utilisés pour coder les caractères alphabétiques de manière consécutive et les caractères de ponctuation.

  A = 41_h = 65_d, B = 42_h = 66_d, etc.

  L'astuce de ce code est qu'il suffit de faire +32 ($5^{ème}$ bit à 1) pour passer aux minuscules, ce qui simplifie grandement les algorithmes de conversion.

  a = 61_h = 97_d, b = 62_h = 98_d, etc.

  Dans une deuxième version, on a étendu le code ASCII à 8 bits (ASCII étendu) ce qui a permis d'introduire les caratères accentués (à, ä, é, è,…) et des caractères semi-graphiques (┴┤├└). Ceci a aussi permis de définir des pages de codes spéciaux (*ang.: codepage*) pour les différents langages basés sur l'alphabet latin ou cyrillique.
- le code ANSI

  Ce code utilisé par les applications Microsoft est une variante du code ASCII.
- le code UNICODE

  Avec l'apparition d'Internet et l'internationalisation de l'usage des ordinateurs, il devient nécessaire de disposer d'un code permettant de traiter aussi les alphabets non latins tels que le japonais ou le chinois qui possèdent des milliers de caractères.

  L'astuce (prévisible) a été de passer d'un codage sur 8 bits (256 caractères) à un codage sur 16 bits (65 536 caractères). Pour maintenir la compatibilité avec les systèmes antérieurs, dans le cas de l'alphabet latin le premier byte est mis à 00_h tandis que le second reprend l'ancien code ASCII ou ANSI.

4.2. Le code ASCII

Dec	Hex	Asc	Dec	Hex	Asc	Dec	Hex	Asc	Dec	Hex	Asc	Dec	Hex	Asc
0	0	NUL	51	33	3	102	66	f	153	99	™	204	CC	Ì
1	1	SOH	52	34	4	103	67	g	154	9A	š	205	CD	Í
2	2	STX	53	35	5	104	68	h	155	9B	›	206	CE	Î
3	3	ETX	54	36	6	105	69	i	156	9C	œ	207	CF	Ï
4	4	EOT	55	37	7	106	6A	j	157	9D	•	208	D0	Ð
5	5	ENQ	56	38	8	107	6B	k	158	9E	ž	209	D1	Ñ
6	6	ACK	57	39	9	108	6C	l	159	9F	Ÿ	210	D2	Ò
7	7	BEL	58	3A	:	109	6D	m	160	A0		211	D3	Ó
8	8	BS	59	3B	;	110	6E	n	161	A1	¡	212	D4	Ô
9	9	HT	60	3C	<	111	6F	o	162	A2	¢	213	D5	Õ
10	A	LF	61	3D	=	112	70	p	163	A3	£	214	D6	Ö
11	B	VT	62	3E	>	113	71	q	164	A4	¤	215	D7	×
12	C	FF	63	3F	?	114	72	r	165	A5	¥	216	D8	Ø
13	D	CR	64	40	@	115	73	s	166	A6	¦	217	D9	Ù
14	E	SO	65	41	A	116	74	t	167	A7	§	218	DA	Ú
15	F	SI	66	42	B	117	75	u	168	A8	¨	219	DB	Û
16	10	SLE	67	43	C	118	76	v	169	A9	©	220	DC	Ü
17	11	CS1	68	44	D	119	77	w	170	AA	ª	221	DD	Ý
18	12	DC2	69	45	E	120	78	x	171	AB	«	222	DE	Þ
19	13	DC3	70	46	F	121	79	y	172	AC	¬	223	DF	ß
20	14	DC4	71	47	G	122	7A	z	173	AD	-	224	E0	à
21	15	NAK	72	48	H	123	7B	{	174	AE	®	225	E1	á
22	16	SYN	73	49	I	124	7C	|	175	AF	¯	226	E2	â
23	17	ETB	74	4A	J	125	7D	}	176	B0	°	227	E3	ã
24	18	CAN	75	4B	K	126	7E	~	177	B1	±	228	E4	ä
25	19	EM	76	4C	L	127	7F	•	178	B2	²	229	E5	å
26	1A	SIB	77	4D	M	128	80	€	179	B3	³	230	E6	æ
27	1B	ESC	78	4E	N	129	81	•	180	B4	´	231	E7	ç
28	1C	FS	79	4F	O	130	82	‚	181	B5	µ	232	E8	è
29	1D	GS	80	50	P	131	83	ƒ	182	B6	¶	233	E9	é
30	1E	RS	81	51	Q	132	84	„	183	B7	·	234	EA	ê
31	1F	US	82	52	R	133	85	…	184	B8	¸	235	EB	ë
32	20		83	53	S	134	86	†	185	B9	¹	236	EC	ì
33	21	!	84	54	T	135	87	‡	186	BA	º	237	ED	í
34	22	"	85	55	U	136	88	ˆ	187	BB	»	238	EE	î
35	23	#	86	56	V	137	89	‰	188	BC	¼	239	EF	ï
36	24	$	87	57	W	138	8A	Š	189	BD	½	240	F0	ð
37	25	%	88	58	X	139	8B	‹	190	BE	¾	241	F1	ñ
38	26	&	89	59	Y	140	8C	Œ	191	BF	¿	242	F2	ò
39	27	'	90	5A	Z	141	8D	•	192	C0	À	243	F3	ó
40	28	(	91	5B	[	142	8E	Ž	193	C1	Á	244	F4	ô
41	29	)	92	5C	\	143	8F	•	194	C2	Â	245	F5	õ
42	2A	*	93	5D	]	144	90	•	195	C3	Ã	246	F6	ö
43	2B	+	94	5E	^	145	91	‘	196	C4	Ä	247	F7	÷
44	2C	,	95	5F	_	146	92	’	197	C5	Å	248	F8	ø
45	2D	-	96	60	`	147	93	“	198	C6	Æ	249	F9	ù
46	2E	.	97	61	a	148	94	”	199	C7	Ç	250	FA	ú
47	2F	/	98	62	b	149	95	•	200	C8	È	251	FB	û
48	30	0	99	63	c	150	96	–	201	C9	É	252	FC	ü
49	31	1	100	64	d	151	97	—	202	CA	Ê	253	FD	ý
50	32	2	101	65	e	152	98	˜	203	CB	Ë	254	FE	þ
												255	FF	ÿ

Codepage 437

5. Les instructions

Pour développer une application, le programmeur utilise un langage, dit de haut niveau, tel que C++ ou Visual Basic. Ce langage, fait de mots, est incompréhensible pour la machine qui, elle, n'est capable que de comprendre une suite de 0 et de 1.

Le programme C++ est traduit en Assembler. C'est un langage déjà beaucoup plus proche de la machine mais qui utilise encore des mots, appelés mnémoniques, pour représenter les opérations.

Il y a six groupes d'opérations :

- Les **transferts de données** entre la mémoire RAM et les registres
  MOV, POP, PUSH,...
- Les **opérations arithmétiques**
  ADD, SUB, IMUL, IDIV, ...
- Les **opérations logiques**
  AND, OR, NOT, XOR, ...
- Les **contrôles de séquence** qui expriment à quel moment le processeur doit exécuter une autre partie du programme
  JMP, CALL, RET, JNZ, JLE, ...
- Les **entrées/sorties** qui lisent ou envoient des bytes vers les différents ports
  IN, OUT, ...
- Les **commandes** et **manipulations diverses**
  NOP, WAIT, INT,...

Certains mnémoniques sont utilisée seules (p.ex.: RET), d'autres prennent un seul paramètre (p.ex.: JMP), d'autres deux ou trois (p.ex.: ADD).

Pour passer des mnémoniques à un langage binaire, on associe chaque mnémonique à un nombre appelé OpCode. Malheureusement, cet OpCode diffère selon les familles de processeurs. Les paramètres sont soit des nombres, facilement représentés en binaire, soit des adresses mémoire, elles aussi facilement représentés en binaire.

Finalement, une instruction apparaît sous la forme d'une suite de bytes :

```
C746FE0500       = mov word ptr [bp-02], 0005
```

En fait, une instruction peut comprendre de 1 à 9 zones d'informations et sa taille varie de 1 à 16 bytes en fonction de sa complexité ([1]).

Instruction prefix	Address size prefix	Operand size prefix	Segment override	**OpCode**	Mode r/M	SIB	Displacement	Immediate
0 or 1	0 or 1	0 or 1	0 or 1	**1 or 2**	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2 or 4
Number of bytes								

Dans ce système, l'instruction la plus courte compte 1 byte et la plus longue est une phrase de 16 bytes. Selon les cas, le code d'instruction (OpCode) peut aussi bien se trouver dans le premier byte que dans le cinquième (ou n'importe où entre les deux). Il est clair que le temps de lecture, de décodage et d'exécution d'une instruction variera largement en fonction de l'instruction à traiter.

1 Cette partie est beaucoup plus développée dans le chapitre consacré au fonctionnement du processeur.

6. Endianisme

6.1. big-endian or small-endian ?

Tous les langages écrits ont dû faire un choix : écrit-on horizontalement ou verticalement ? Si on décide d'écrire horizontalement, est-ce de gauche à droite ou de droite à gauche ?

On sait que les choix diffèrent selon les cultures et les populations.

L'informatique, qui est une manière d'écrire l'information dans un ordinateur, n'échappe pas à ces questions… ni à des différences de choix selon les populations de processeurs.

Au sein d'**un byte**, pas de problème, tout les processeurs suivent la même convention :

$$1 \text{ Byte} = b_7b_6b_5b_4b_3b_2b_1b_0$$

- le bit numéro 0 est à droite. Ce bit représente les unités; c'est le **bit de poids faible**.
- le bit numéro 7 est à gauche. Ce bit représente les multiples de 128; c'est le **bit de poids fort**.

Les choses se compliquent quand un type utilise **plusieurs bytes** comme c'est le cas pour les entiers, les entiers longs et les réels.

La convention **big-endian** (*fr. litt.: gros boutien*), suit la logique de lecture de bits; le **byte de poids fort** est placé à gauche et le byte de poids faible à droite :

$$1 \text{ Réel} = 4 \text{ Bytes} = B_3B_2B_1B_0 = b_{31}\ldots b_{24} \mid b_{23}\ldots b_{16} \mid b_{15}\ldots b_8 \mid b_7\ldots b_0$$

Selon cette convention, le nombre réel converti au paragraphe précédent est stocké en mémoire et sur disque selon la succession

$445{,}9_d$ ð 43 DE F3 33_h

Cette convention est notamment utilisée par les processeurs Motorola qui équipent la famille Apple/Mac.

La convention **small-endian** (*fr. litt.: petit boutien*), suit la logique inverse; le **byte de poids faible** est placé à gauche et le byte de poids fort à droite *mais au sein de chaque byte, les bits se lisent toujours dans l'ordre habituel* :

$$1 \text{ Réel} = 4 \text{ Bytes} = B_0B_1B_2B_3 = b_7\ldots b_0 \mid b_{15}\ldots b_8 \mid b_{23}\ldots b_{16} \mid b_{24}\ldots b_{31}$$

Selon cette convention, le nombre réel converti au paragraphe précédent est stocké en mémoire et sur disque selon la succession

$445{,}9_d$ ð 33 F3 DE 43_h

Les processeurs des familles Intel et AMD qui équipent les PC sont de type small-endian.

Exercices du chapitre

♦ Exercice 1

Encodez le nombre 187 en tant que

- entier non signé,
- entier signé,
- réel.

♦ Exercice 2

Un nombre de 4 bytes (big-endian) s'écrit B4 C2 F5 56_h. De quel nombre s'agit-il si c'est

- un entier long non signé,
- un entier long signé,
- un réel.

♦ Exercice 3

On sait que le nombre réel 0 constitue une exception à la règle de codage des réels. Quelle serait la valeur du nombre codé

0000 0000 0000 0000 0000 0000 0000 0000

si on appliquait la règle de décodage sans tenir compte de ce cas particulier ?