Première section

Organisation de la machine

Représentation des données Chapitre 1

La plus grande difficulté que la plupart des débutants rencontrent en apprenant le langage assembleur est probablement l'usage commun des systèmes de numération binaire et hexadécimal. Beaucoup de programmeurs pensent que les nombres hexadécimaux (ou hex[1]) représentent la preuve définitive que Dieu n'a jamais destiné personne à travailler en assembleur. Quoi que ces nombres ne ressemblent pas à ce que vous voyez tous les jours, leurs avantages dépassent de loin les inconvénients. Donc, comprendre ces systèmes de numération est important parce que leur utilisation simplifie d'autres sujets complexes, par exemple, l'algèbre booléenne et la conception logique, les représentations numériques signées, les codes des caractères et les paquets de données.

1.0 Vue d'ensemble du chapitre

Ce chapitre décrit plusieurs concepts importants, dont les systèmes de numération binaire et hexadécimal, l'organisation binaire des données (bits, quartets, octets, mots et doubles-mots), nombres signés et non signés, opérations arithmétiques, logiques, de décalage et de rotation sur les valeurs binaires, champs de bits, paquets de données et l'ensemble des caractères ASCII. Ceci constitue la matière de base, ainsi que l'essence de la compréhension du contenu du reste du livre, laquelle dépend de la maîtrise de ces notions. Si vous avez déjà abordé ces sujets dans d'autres cours, vous devriez tout de même les parcourir avant de passer au prochain chapitre. Si vous n'êtes pas familiarisé avec cette matière ou si vous n'en avez qu'une vague connaissance, vous devriez l'étudier soigneusement avant d'aller plus loin. *Tout le contenu du chapitre est important !* N'en sautez donc aucune partie.

1.1 Systèmes de numération

La plupart des systèmes informatiques ne représentent pas les valeurs numériques de façon décimale. Au contraire, ils font usage du système de numération binaire ou de complément à deux. Pour comprendre les limitations de l'arithmétique informatique vous devez comprendre comment les ordinateurs représentent les nombres.

1.1.1 Une révision du système décimal

Vous avez utilisé le système de numération décimal (base 10) pendant tellement de temps que vous le considérez désormais évident au point d'en oublier les propriétés intrinsèques. Quand vous voyez un nombre comme "123" vous ne pensez pas à la valeur 123, mais vous vous faites une image mentale sur le nombre d'objets que la valeur représente. En réalité, cependant, le nombre 123 représente :

$$1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

ou

$$100 + 20 + 3$$

Chaque chiffre apparaissant à la gauche de la virgule décimale représente une valeur de zéro à neuf fois une puissance croissante de dix. Et chaque nombre apparaissant à la droite de la virgule décimale représente une valeur de zéro à neuf fois une puissance décroissante de dix. Par exemple, la valeur 123,456 peut s'écrire :

$$1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2} + 6 * 10^{-3}$$

ou

$$100 + 20 + 3 + 0,4 + 0,05 + 0,006$$

1.1.2 Le système de numération binaire

[1]Le terme *hexadécimal* est parfois abrégé par *hex*, même si, techniquement parlant, hex veut dire base six et non base seize.

La plupart des systèmes informatiques modernes (en incluant l'IBM PC) fonctionnent en utilisant la logique binaire. L'ordinateur représente les données en utilisant deux niveaux de voltage (normalement 0v et 5v). Avec ces deux niveaux, on peut représenter exactement deux valeurs. Celles-ci peuvent être représentées par tout couple de nombres mais, par convention, on utilise les valeurs 0 et 1. Et ces dernières correspondent exactement aux deux chiffres utilisés dans le système de numération binaire. Puisqu'il y a une correspondance entre les niveaux logiques utilisés par les 80x86 et les deux termes utilisés en numération binaire, il n'est par surprenant que l'IBM PC s'en serve.

Le système de numération binaire fonctionne tout comme le système de numération décimale, avec deux exceptions : les nombres binaires comprennent seulement les chiffres 0 et 1 (au lieu de 0-9) et utilisent des puissances de deux à la place des puissances de dix. Par conséquent, il est facile de convertir un nombre binaire en décimal. Pour chaque "1" de la chaîne binaire, on ajoute 2^n, où *n* est la position du bit. Par exemple, la valeur binaire 11001010_2 représente :

$$1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$
$$= 128 + 64 + 8 + 2$$
$$= 202_{10}$$

Pour convertir un nombre décimal en binaire, c'est un peu plus difficile. Vous devez trouver les puissances de deux qui, additionnées, produisent le résultat décimal. La méthode la plus facile est de procéder de la plus grande puissance de deux dans le nombre jusqu'à 2^0. Considérez la valeur décimale 1359 :

- $2^{10} = 1024$, $2^{11} = 2048$. Donc 1024 est la plus grande puissance de deux inférieure à 1359. Soustrayez 1024 de 1359 et commencez la valeur binaire avec un "1" à gauche. Binaire = "1", résultat décimal : 1359 - 1024 = 335.
- La prochaine puissance de deux ($2^9 = 512$) est plus grande que le résultat obtenu ci-dessus, donc, ajouter "0" à la fin de la chaîne binaire. Nombre binaire = "10" et le résultat décimal est encore 335.
- La prochaine puissance de deux est 256 (2^8). Soustraire ceci à 335 et ajouter "1" à la fin de la chaîne binaire. Nombre binaire = "101", avec un résultat décimal de 79.
- 128 (2^7) est plus grand que 79, donc ajouter "0" à la chaîne binaire. Nombre binaire = "1010" et le résultat décimal reste 79.
- La prochaine plus petite puissance de deux ($2^6 = 64$) est inférieure à 79, donc lui soustraire 64 et ajouter "1" à la chaîne binaire. Nombre binaire = "10101", avec un résultat décimal de 15.
- 15 est inférieur à la prochaine puissance de deux ($2^5 = 32$), donc ajouter simplement "0" à la fin de la chaîne binaire. Nombre binaire = "101010" et le résultat décimal est encore 15.
- 16 (2^4) est plus grand que 15, donc ajouter un autre "0" à la chaîne. Nombre binaire = "1010100", avec un résultat binaire encore de 15.
- 2^3 (8) est inférieur à 15, donc ajouter un autre "1". Nombre binaire = "10101001" avec un résultat décimal de 7.
- 2^2 est plus petit que 7, donc le lui soustraire en ajoutant un autre "1" à la chaîne. Nombre binaire = "101010011", résultat décimal : 3.
- 2^1 est inférieur à 3, donc ajouter un autre "1" au nombre binaire et soustraire 2 au nombre décimal. Nombre binaire = "1010100111" et le résultat décimal est maintenant 1.
- Finalement le résultat décimal est 1, qui correspond à 2^0, donc ajouter "1" final à la fin de la chaîne binaire. Le résultat final binaire est : "10101001111".

Quoique les nombres binaires apparaissent sans trop d'importance dans les langages de haut niveau, ils sont partout dans les programmes en assembleur.

1.1.3 Les formats binaires

De façon purement abstraite, chaque nombre binaire contient un nombre infini de chiffres (ou *bits* qui est une contraction du terme anglais *binary digit*). Par exemple, on peut représenter le nombre cinq par :

101 00000101 0000000000101 000000000000101 …

Tout nombre arbitraire de zéros peut précéder le nombre binaire sans changer sa valeur.

On adopterait la convention d'ignorer tout zéro superflu. Par exemple, 101^2 représenterait le nombre cinq. Cependant, par le fait que les processeurs 80x86 travaillent avec des groupes de huit bits, l'on trouvera beaucoup plus simple d'exprimer tous les nombres avec des zéros de remplissage, de façon à les faire apparaître comme s'ils étaient des nombres de quatre ou de huit bits. Par conséquent, en adoptant cette convention, on écrira le nombre cinq comme 0101^2 ou 00000101^2.

La plupart des gens séparent par un point[2] tout groupe de trois chiffres pour rendre plus simple la lecture des gros nombres décimaux. Par exemple, 1.023.435.208 est beaucoup plus facile à lire et à comprendre que 1023435208. Dans ce texte, on adoptera une convention similaire pour la lecture des nombres binaires. Nous séparerons chaque groupe de quatre chiffres binaires par un espace. Par exemple, la valeur binaire 1010111110110010 sera écrite : 1010 1111 1011 0010.

Souvent, on empaquette plusieurs valeurs dans le même nombre binaire. Une forme de l'instruction MOV sur 80x86 (voir annexe D) utilise l'encodage binaire 10110 *rrr dddd dddd* pour emboîter trois objets dans 16 bits : un code d'opération de cinq bits (10110), un registre dans un champ de trois bits (rrr) et une valeur immédiate de huit bits (dddd dddd). Par convention, on assignera une valeur numérique à chaque position de bit. Nous allons numéroter chaque bit comme suit :

1) Le bit le plus à la droite du nombre binaire est la position de bit zéro.
2) Chaque bit à gauche est donné par la prochaine et successive position de bit.

Une valeur binaire de huit bits utilise les bits zéro à sept :

$$X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

Une valeur binaire de seize bits utilise les positions de zéro à quinze :

$$X_{15}\ X_{14}\ X_{13}\ X_{12}\ X_{11}\ X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

Le bit zéro est normalement référé comme le *bit de poids faible* ou *bit le moins significatif* (low order bit ou L.O. en anglais). Alors que le bit le plus à gauche est nommé le *bit de poids fort* ou *bit le plus significatif* (high order bit ou H.O. en anglais). L'on se référera aux bits intermédiaires avec leur nombre de position respectif.

1.2 Organisation des données

En mathématiques pures, une valeur est formée par un nombre arbitraire de bits. Cependant, les ordinateurs travaillent avec un certain nombre spécifique (et fini) de bits. Des groupes quelconques sont des bits isolés, des bits groupés par quatre (nommés *quartets* ou *quarters* en anglais), par huit (nommés *octets* ou *bytes* en anglais), par seize (qu'on nomme *mots*, ou *words* en anglais), et ainsi de suite. Les tailles ne sont pas arbitraires et ces valeurs particulières ont toutes une raison d'être. Cette section décrira les groupes de bits d'usage le plus commun dans les puces Intel 80x86.

1.2.1 Bits

La plus petite "unité" de données dans un ordinateur binaire est le simple *bit*[3]. Puisqu'une telle unité est capable de représenter seulement deux valeurs différentes (typiquement 0 et 1) vous aurez l'impression qu'il y a un très petit nombre d'objets qu'on peut représenter avec un simple bit. C'est faux ! Il y a une infinité de nombres qu'on peut représenter avec deux états opposés.

En effet, avec un seul bit, vous pouvez représenter toute paire de deux objets différents. Des exemples comprennent 0 et 1, Vrai et Faux, Allumé ou Éteint, Mâle ou Femelle, Bon ou Mauvais, etc. Cependant, on n'est pas limité à représenter seulement deux états opposés comme des types de données binaires (c'est-à-dire, ces objets qui prennent seulement deux valeurs distinctes). Vous pourriez utiliser un bit pour représenter les nombres 723 et 1245. Ou bien 6254 et 5. Ou même les couleurs rouge et bleu. Ou encore deux objets sans relation, par exemple, le rouge et le nombre 3256. Bref, vous pouvez représenter n'importe quelle sorte de couple de valeurs avec un bit. Par contre, on peut représenter au plus deux valeurs.

[2]Les notations changent selon les pays, par exemple, aux Etats-Unis et dans nombre d'autres pays, on utilise la virgule, n.d.t.
[3]"bit" est la contraction anglaise de *binary digit* (nombre binaire), ndt.

Aussi, différents bits peuvent représenter différentes choses. Par exemple, un bit pourrait être utilisé pour représenter les valeurs 0 et 1, alors qu'un bit adjacent pourrait parfaitement représenter les valeurs Vrai et Faux. Que peut-on dire en regardant simplement les bits ? La réponse, assurément, est : «rien». Mais ceci illustre toute l'idée qui se trouve derrière les structures de données des ordinateurs : *les données sont ce que vous définissez.* Si vous vous servez d'un bit pour représenter une valeur booléenne (Vrai ou Faux), alors ce bit représentera cette valeur. Et afin que ce bit maintienne une signification véritable, vous devrez être cohérent. Cela étant, si vous utilisez un bit pour représenter Vrai ou Faux, il va de soi que vous ne devriez pas utiliser ce même bit pour indiquer une relation rouge/bleu.

Puisque la plupart des objets que vous modelez demandent plus de deux valeurs pour être décrits, il en découle que les bits ne sont pas exactement les types de données les plus populaires qu'on peut utiliser. Cependant, ces derniers joueront tout de même un rôle important dans vos programmes, car tous les autres objets sont constitués par des groupes de bits. Bien sûr, il y a plusieurs types de données qui demandent deux valeurs distinctes, ce qui fait déduire que les bits sont également importants en eux-mêmes. Mais, vous allez bientôt découvrir que les bits individuels sont difficiles à manipuler, par conséquent, on utilise souvent d'autres types de données pour représenter des valeurs booléennes.

1.2.2 Quartets

Un *quartet* ou *nibble* en anglais, est un groupe de quatre bits. Il ne constitue pas une très importante structure de données, sauf pour deux types d'objets : nombres de type BCD (sigle du terme anglais *binary coded decimal*[4]), et nombres hexadécimaux. Quatre bits sont requis pour représenter ces nombres. Avec un quartet, on peut représenter jusqu'à seize valeurs différentes. Dans le cas des nombres hexadécimaux, les valeurs 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F sont représentées par quatre bits (voir "Le système de numération hexadécimal au paragraphe 1.3). Alors que BCD utilise dix différents chiffres (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) et demande également quatre bits. En fait, tout groupe de seize valeurs différentes peut être représenté par un quartet, mais le système hexadécimal et le système BCD sont les outils primaires qu'on représente avec une telle structure.

1.2.3 Octets

Sans conteste, la plus importante structure de données utilisée dans les microprocesseurs 80x86 est *l'octet* (ou *byte* en anglais). Un octet contient huit bits et c'est la plus petite donnée adressable dans un microprocesseur 80x86. La mémoire principale et les emplacements d'entrées/sorties sont tous constituées par des adresses de huit bits. Ceci veut dire que l'objet le plus petit pouvant être accédé individuellement par un programme tournant sur 80x86 est une valeur d'octet. Accéder à toute unité plus petite demande la lecture de l'octet contenant la donnée et le masquage dans celui-ci des bits superflus. Les bits dans un octet sont normalement dénombrés de zéro à sept, via la convention illustrée dans la figure 1.1.

Le bit 0 est dit *le bit de poids le plus faible* ou *le bit faible*, alors que le bit 7 est indiqué comme étant *le bit de poids le plus fort* ou *le bit fort* de l'octet. L'on se réfère à tous les autres bits par leur numéro.

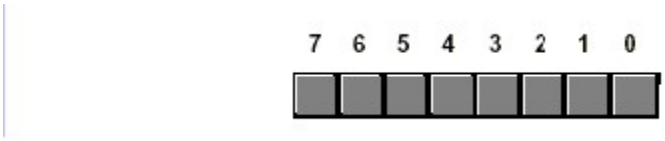


Figure 1.1 Dénombrement dans un octet

Notez qu'un octet contient aussi exactement deux quartets (voir figure 1.2).

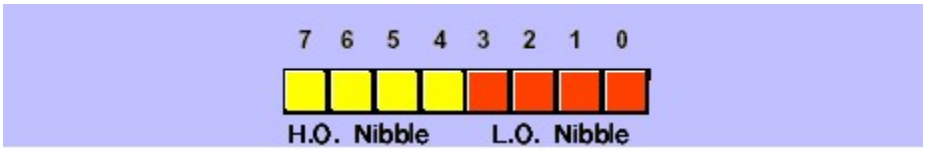


Figure 1.2 Les deux quartets dans un octet

[4]Binaire codé décimal, en français, ntd.

Les bits de la plage 0…3 comprennent le *quartet de poids faible*, alors que les bits de la plage 4…7 constituent le *quartet de poids fort*. Puisqu'un octet contient exactement deux quartets, le représenter requiert deux chiffres hexadécimaux.

Puisqu'un octet contient huit bits, il peut représenter 2^8, ou 256, valeurs différentes. Généralement, l'on utilise les octets pour représenter des valeurs numériques dans la plage 0…255, des valeurs non signées dans la plage -128…+127 (voir "Nombres signés et non signés", au paragraphe 1.7), des codes de caractères ASCII/IBM et autres types de données spéciaux qui demandent au plus 256 valeurs différentes. Certains ensembles ont moins de 256 éléments, donc, huit bits sont plus que suffisants.

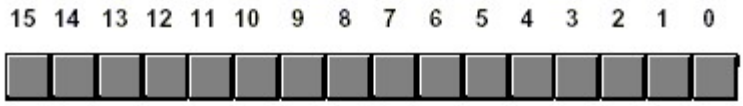
La famille 80x86 est faite de machines adressables par octets (voir "Disposition de la mémoire et accès" au début du chapitre 4), donc il s'avère plus efficace de manipuler un octet tout entier qu'un bit isolé ou un quartet. Pour cette raison, beaucoup de programmeurs utilisent un octet complet pour représenter des types de données qui demandent non plus de 256 valeurs, même si moins de huit bits pourraient suffire. Par exemple, on représente souvent les valeurs Vrai ou Faux par 00000001_2 et 00000000_2 (respectivement).

L'application la plus importante des octets est probablement le stockage des codes de caractères (voir "Le jeu de caractères ASCII", au paragraphe 1.11). Il y a 128 codes définis dans ce jeu. IBM utilise les 128 possibles valeurs restantes pour un jeu étendu, qui comprend des caractères européens, des symboles graphiques, des lettres grecques et des symboles mathématiques. Voir l'appendice A pour l'assignation des codes aux caractères[5].

1.2.4 Mots

Un *mot* (*word*, en anglais) est un groupe de seize bits. On dénombre ces bits de zéro à quinze. Ce dénombrement apparaît à la figure 1.3.

Figure 1.3 Le nombre de bits dans un mot.



Comme l'octet, le bit 0 est le bit de poids le plus faible et le bit 15 est le bit de poids le plus fort. Quand on fait référence aux autres bits dans le mot, on les référence par le numéro correspondant à leur position. Notez qu'un mot contient exactement deux octets. Par conséquent, les bits 0..7 forment l'*octet faible*, alors que les octets 8…15 constituent l'*octet fort* du mot (voir figure 1.4).

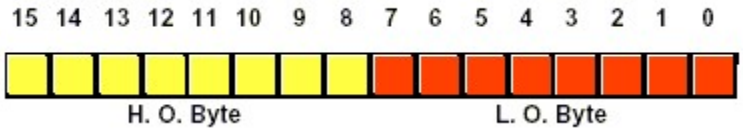


Figure 1.4 Les deux octets dans un mot

Naturellement, un mot peut être successivement divisé en quatre nibbes, comme montré à la figure 1.5.

[5]Au moment de la traduction de ce chapitre, l'original anglais de cet appendix n'était pas encore disponible. Référez-vou au projet original https://www.plantation-productions.com/Webster/

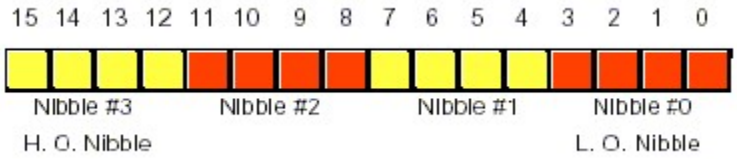


Figure 1.5 Les quartets dans un mot.

Le quartet 0 est le quartet de poids le plus faible est le quartet 3 est le quartet fort du mot. Les deux autres quartets sont le "quartet un" et le "quartet deux".

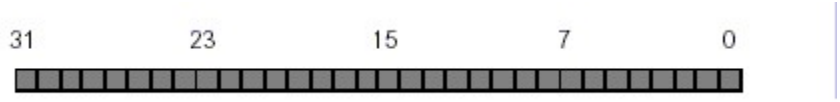
Avec 16 bits, vous pouvez représenter 2^{16} (65 536) valeurs différentes. Celles-ci peuvent être les valeurs de la plage 0…65535 (ou, comme c'est souvent le cas, -32 768...+32 767), ou tout autre type de données ne dépassant les 65 536 valeurs. Les trois utilisations majeures pour les mots sont les valeurs entières, les offsets et les segments (voir encore "Disposition de la mémoire et accès", au début du chapitre 4, pour une description des segments et des offsets).

Les mots peuvent représenter des valeurs entières de la plage 0…65 536 ou -32 768…+32 767. Les valeurs numériques non signées sont représentées par la valeur binaire correspondant au nombre des bits dans le mot. Les valeurs signées utilisent la forme à deux compléments (voir encore "Nombres signées et non signés", au paragraphe 1.7). Les valeurs de segment, qui ont toujours 16 bits de long, constituent l'*adresse de paragraphe* des segments de code, de données, extra-segment ou segment de pile dans la mémoire.

1.2.5 Doubles-mots

Un double-mot est exactement ce que son nom implique, une paire de mots. Par conséquent, une telle quantité est formée de 32 bits, comme montré à la figure 1.6.

Figure 1.6 Le nombre des bits dans un double-mot



Naturellement, ce double-mot peut être divisé en en mot fort et en un mot faible, ou en quatre différents octets, ou encore en seize différents quartets (voir figure 1.7).

Les doubles-mots peuvent représenter tout genre de choses différentes. Ils répresentent d'abord les adresses segmentées. Un autre objet commun représenté par un double-mot est une valeur entière de 32 bits (qui permet des nombres non signés dans la plage 0…4294967295 ou la plage signée -2147483648…+2147483647). Les valeurs à virgule flottante de 32 bits s'arrangent aussi dans des doubles-mots. La plupart du temps, on utilise des doubles-mots pour stocker des adresses segmentées.

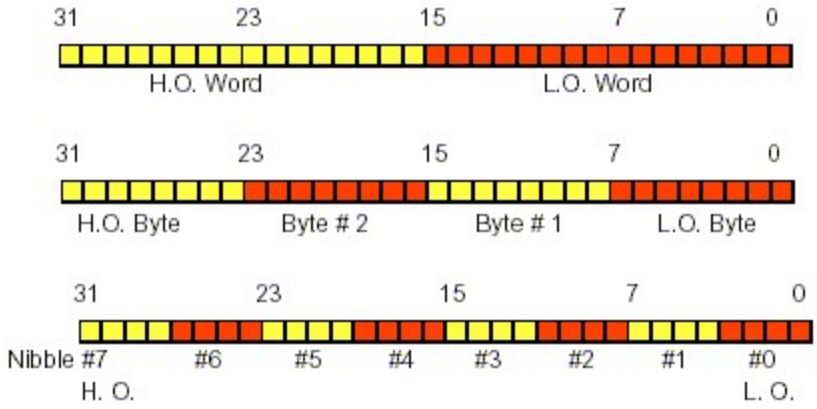


Figure 1.7 Quartets, Octets et mots dans un double-mot

1.3 Le système de numération hexadécimal

Un grand problème avec le système binaire est leur longueur. Pour représenter la valeur 202_{10}, il faut huit chiffres binaires. La version décimale demande seulement trois chiffres et, donc, peut représenter des nombres de façon beaucoup plus compacte. Ceci n'a pas échappé aux ingénieurs à l'heure de concevoir des systèmes informatiques binaires. En faisant face à de grandes valeurs, les nombres binaires deviennent rapidement inutilisables. Malheureusement, un ordinateur pense en binaire et il convient le plus souvent d'utiliser ce système. Bien qu'on peut convertir entre décimal et binaire, la conversion n'est pas aussi simple. Le système de numération hexadécimal (en base 16) résout ces problèmes. Les nombres hexadécimaux offrent les deux caractéristiques qu'on cherche : ils sont encore plus compacts que les nombres décimaux et il est facile de les convertir en binaire et vice-versa. C'est pourquoi, la plupart des systèmes informatiques binaires utilisent le système de numération hexadécimal[6]. Etant donné que la base d'un tel nombre est 16, chaque chiffre à la gauche du point décimal représente une puissance successive de seize. Par exemple, le nombre 1234_{16} est égal à :

$$1 * 16^3 + 2 * 16^2 + 3 * 16^1 + 4 * 16^0$$

ou

$$4096 + 512 + 48 + 4 = 4660_{10}$$

Chaque chiffre hexadécimal peut représenter une des seize valeurs allant de 0 à 15_{10}. Puisque notre système numérique n'a que dix chiffres, on a besoin d'inventer six unités additionnelles pour représenter les valeurs dans la plage de 10_{10} à 15_{10}. Au lieu de créer de nouveaux symboles, on utilise les lettres de l'alphabet, de A à F. Ce qui suit, représente des exemples valides de nombres hexadécimaux :

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

Puisqu'on a besoin souvent d'entrer des nombres hexadécimaux dans un système informatique, il nous faut un mécanisme différent pour les représenter. Après tout, dans la plupart des ordinateurs, il n'est pas possible d'inclure un index pour indiquer la base de la valeur associée. On adoptera donc les conventions suivantes :

- Toutes les valeurs numériques (peu importe la base) commencent avec un chiffre.
- Toutes les valeurs hexadécimales terminent avec la lettre "h", par exemple 123A4h[7].
- Toutes les valeurs binaires terminent avec la lettre "b".
- Les nombres décimaux peuvent avoir le suffixe "t" ou "d".

Exemples de nombres hexadécimaux valides :

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

[6]La société Digital Equipment est la seule qui encore résiste. Ils utilisent encore les nombres octals dans la plupart de leurs systèmes. Un héritage des jours où ils produisaient des machines de 12 bits.

[7]Actuellement, le suffixe "h" des nombres hexadécimaux est une convention de Intel, non une convention générale. Les assembleurs 68000 et 65c816 utilisés dans les systèmes Macintosh et Apple II préfixent ces nombres avec le symbole "$".

Comme vous pouvez voir, les nombres hexadécimaux sont compacts et faciles à lire. De plus, vous pouvez effectuer facilement les conversions entre hexadécimal et binaire. Considérez la table suivante :

Table 1 : conversion binaires/hexa

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

Cette table donne toute l'information dont vous avez besoin pour convertir tout nombre hexadécimal en nombre binaire ou vice-versa.

Pour convertir un nombre hexadécimal en binaire, substituez simplement dans le nombre les quatre bits correspondant à chaque chiffre hexadécimal. Par exemple, pour convertir 0ABCDh en valeur binaire, convertissez chaque chiffre conformément à la table ci-dessus :

0	A	B	C	D	Hexadécimal
0000	1010	1011	1100	1101	Binaire

Pour convertir un nombre binaire en format hexadécimal est également facile. La première étape est de remplir le côté gauche du nombre binaire de zéros pour s'assurer qu'il y a un multiple de quatre bits dans le nombre de chiffres. Par exemple, étant donné le nombre 1011001010, il faudrait commencer par ajouter deux bits à la gauche du nombre de façon qu'il contienne 12 bits. La valeur binaire convertie devient 001011001010. La prochaine étape consiste à subdiviser le nombre en groupes de quatre bits, c'-à-d., 0010 1100 1010. Finalement comparez ces valeurs dans la table est substituez-les avec les chiffres hexadécimaux appropriés, donc, 2CA. Comparez à cela la difficulté de conversion entre nombres décimaux et binaires ou entre décimaux et hexadécimaux !

Puisque convertir entre hexadécimal et binaire est une opération que vous ferez répétitivement, il convient de mémoriser les nombres binaires de 1 à 15 (illustré dans la table ci-dessus). Même si vous avez une calculatrice

qui fait la conversion pour vous, vous trouverez que les conversions manuelles entre nombres binaires et hexadécimaux sont beaucoup plus rapides et beaucoup plus efficaces.

1.4 Opérations arithmétiques sur les nombres binaires et hexadécimaux

Il y a diverses opérations qu'on peut effectuer avec des nombres binaires et hexadécimaux. Par exemple, on peut additionner, soustraire, multiplier, diviser et effectuer d'autres opérations arithmétiques. Même si vous n'avez pas besoin de devenir précisément un expert dans ce domaine, il faudrait être tout de moins capable d'effectuer ces opérations manuellement avec papier et crayon. Ceci vous donnera clarté, même si, à la fin, vous ferez la plupart de ces convertions à l'aide d'une calculatrice. Il y a plusieurs de ces outils dans me marché ; la liste suivante donne un certain nombre de fabriquants qui produisent ces appareils :

Manufacturiers de calculatrices hexadécimales :

- Casio
- Hewlett-Packard
- Sharp
- Texas Instruments

Cette liste n'est pas du tout exhaustive. D'autres constructeurs peuvent également en produire. Certes, Hewlett-Packard est la meilleure du genre, mais c'est plus chère. Sharp et Casio produisent des calculatrices qui se vendent à moins de 50 $. Si vous comptez de programmer en assembleur, avoir une de ces calculatrices est essentiel.

Une autre alternative pour avoir une calculatrice hexadécimale est d'obtenir un programme TSR (Terminate and Stay Resident), comme SideKick™ qui contient un calculateur intégré. Cependant, à moins que vous ne l'ayez déjà, ou à moins de n'avoir besoin des autres caractéristiques qu'il offre, cela ne vaut en définitive pas la peine, étant donné qu'il coûte plus qu'une calculatrice est il n'est pas aussi commode à utiliser.

Pour comprendre pourquoi vous devriez investir une petite somme en une calculatrice, considérez le problème arithmétique suivant :

```
9h +
1h =
----
```

Vous seriez tentés d'écrire la réponse "10h" comme solution du problème. Mais, c'est faux ! La réponse correcte est dix, qui s'écrit "0Ah", et non seize, qui s'écrit "10h". Un problème semblable existe avec cette autre opération :

```
10h -
01h =
-------
```

Vous êtes peut-être portés à répondre "9h", même si la réponse correcte est cependant "0Fh". Souvenez-vous que ce problème pose la question : «quelle est la différence entre seize et un ?». La réponse est évidemment quinze, qui se note "0Fh".

Même si les deux problèmes mentionnés ne vous impressionnent pas particulièrement, dans une situation de stress, votre cerveau passera au mode décimal, tout moment de distraction amènera à des résultats incorrects. Morale de l'histoire - si vous devez effectuer une opération arithmétique à la main sur des nombres hexadécimaux, accordez-vous un moment et faites attention. Vous pourriez à la limite convertir les nombres en décimal, effectuer l'opération et les reconvertir en hexadécimal.

Vous ne devriez jamais effectuer des calculs arithmétiques directement en binaire. Puisque ces nombres contiennent souvent de longues chaînes de bits, vous aurez trop d'opportunités de vous tromper. Si vous tombez

sur des nombres binaires, convertissez-les toujours en hex, effectuez les calculs en hex (de préférence avec une calculatrice), et reconvertissez le résultat en binaire, si nécessaire.

1.5 Opérations logiques sur les bits

Il y a quatre opérations logiques fondamentales qu'on peut effectuer sur des nombres binaires ou hexadécimaux : AND, OR, XOR (OR exclusif) et NOT[8]. Contrairement aux opérations arithmétiques, un calculateur n'est pas nécessaire pour effectuer ces opérations. C'est souvent plus facile de les faire à la main. L'opération AND logique est dyadique[9] (elle accepte exactement deux opérandes). Ces opérandes sont de simples bits binaires (en base 2). L'opération AND consiste en :

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

Une manière compacte de représenter un AND est de se servir d'une table de vérité :

Table 2 : Table de vérité pour AND

AND	0	1
0	0	0
1	0	1

C'est semblable à la table de multiplication de l'école primaire. Les colonnes à la gauche et en haut représentent les valeurs d'entrée de l'opération AND. La valeur située à l'intersection d'une ligne et d'une colonne (pour une paire spécifique de valeurs d'entrée), est le résultat de l'opération logique entre ces deux valeurs. En français cette opération se traduit par : « Si la première opérande est 1 et la seconde opérande est 1, le résultat est 1 ; sinon, c'est 0 ».

Un fait important à noter à propos de l'opération AND est que vous pouvez l'utiliser pour forcer un résultat de zéro. Si l'une des deux opérandes est nulle, le résultat sera toujours 0, peu importe la valeur de l'autre opérande. Par exemple, la ligne contenant une entrée de zéro ne contient que des zéros et la colonne qui contient une entrée de zéro ne contient également que des zéros. Inversement, si une opérande contient 1, le résultat est exactement la valeur de l'autre opérande. Ces caractéristiques de l'opération AND sont très importantes, en particulier quand vous travaillez avec des chaînes de bits et vous voulez forcer des bits individuels à être nuls. On étudiera ces applications de l'opération AND dans la prochaine section.

L'opération logique OR est également dyadique. Sa définition est :

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

La table de vérité du OR prend la forme suivante :

Table 3 : Table de vérité pour OR

[8]Il est d'usage commun de nommer ces opérateurs en anglais, tout le monde les comprend et ils correspondent exactement aux commandes commandes logiques du langage assembleur, n.d.t.
[9]Dans beaucoup de textes, ces opérations sont dites "binaires". Cependant, le terme "dyadique" veut dire la même chose et évite la confusion avec le système de numération binaire.

OR	0	1
0	0	1
1	1	1

En peu de termes, l'opération OR s'explique par : « Si une des deux opérandes ou les deux valent 1, alors le résultat c'est 1 ; sinon, c'est zéro ». Cette opération est également connue comme *OR inclusif*.

Donc, si une seule des deux opérandes est 1, le résultat sera toujours 1, peu importe la valeur de l'autre opérande. Et si une opérande vaut 0, le résultat est toujours la valeur de l'autre opérande. Comme l'opération AND, ceci a un effet de bord important qu'on peut exploiter à l'heure d'effectuer une opération de chaînes de bits (voir la prochaine section). Notez qu'il y a une différence entre le OR inclusif le la signification littéraire du mot "ou". Considérez la phrase : « Je vais au magasin ou je vais au parc ». Une telle affirmation implique que vous allez soit au magasin, soit au parc, pas aux deux endroits en même temps. Par conséquent, la version littéraire du mot "ou" est légèrement différente de l'opération OR inclusive ; elle est beaucoup plus proche de l'opération OR *exclusif*.

XOR (OR eXclusif) est également une opération dyadique. Et elle est définie comme suit :

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

La table de vérité de l'opération XOR prend la forme suivante :

Table 4 : Table de vérité pour XOR

XOR	0	1
0	0	1
1	1	0

Donc, l'opération XOR peut s'exprimer par : « Si l'une des deux opérandes, mais pas les deux, vaut 1, alors le résultat sera 1 ; sinon, c'est 0 ». Notez que la version exclusive de OR est identique à la signification littéraire du mot "ou".

Si l'une des deux opérandes dans un XOR vaut 1, alors le résultat est toujours l'*inverse* de l'autre opérande ; autrement dit, si une opérande vaut 1, le résultat est 0 si l'autre opérande vaut 1, ou bien c'est 1, si l'autre opérande vaut 0. Si la première opérande contient un zéro, alors le résultat sera exactement la valeur de l'autre opérande. Cette propriété vous permet d'invertir sélectivement les bits dans une chaîne de bits.

L'opération NOT est monadique (c'est-à-dire qu'elle n'accepte q'une opérande). C'est :

NOT 0 = 1

NOT 1 = 0

La table de vérité de NOT prend la forme suivante :

Table 5 : Table de vérité pour NOT

NOT	0	1

	1	0

1.6 Opérations logiques sur les nombres binaires et les chaînes de bits

Comme il est décrit dans la section précédente, les opérations logiques fonctionnent seulement avec des opérandes d'un seul bit. Etant donné que les processeurs 80x86 utilisent des groupes de huit, seize ou trente-deux bits, on doit faire une extension de la signification de ces fonctions pour pouvoir traiter plus de deux bits à la fois. Les fonctions logiques 80x86 opèrent sur une base de bit-à-bit (on appelle cela en anglais *bitwise opération*). Etant donné deux valeurs numériques, ces fonctions traitent individuellement les bits qui les composent, bit à bit. Par exemple, si vous voulez calculer le AND des deux valeurs suivantes de huit bits, alors l'opération sera effectuée sur chaque colonne prise individuellement[10] :

```
1011 0101
1110 1110
---------
1010 0100
```

La forme bit-à-bit de l'exécution peut facilement être appliquée à d'autres opérations logiques.

On a défini les opérations logiques en termes de valeurs binaires ; en effet, vous trouverez beaucoup plus facile de les effectuer en binaire, plutôt qu'en d'autres bases. Par conséquent, si vous devez calculer une opération logique entre deux nombres hexadécimaux, vous devriez d'abord les convertir en binaire. Ceci s'applique avec toutes les opérations logiques (par exemple, AND, OR, XOR, etc.).

La possibilité de forcer des bits individuels à valoir 0 ou 1 via AND ou OR, ainsi que la possibilité d'inverser des bits à l'aide de XOR est très importante dans le travail avec des chaînes de bits. Ces opérations vous permettent de manipuler sélectivement certains bits dans une valeur donnée en laissant les autres inchangés. Par exemple, si vous avez une valeur binaire X de huit bits et vous voulez vous assurer que les bits de 4 à 7 ne contiennent que des zéros, vous pouvez soumettre X à un AND contre la valeur binaire 0000 1111. Ceci force le quartet le plus significatif à valoir zéro tout en laissant les premiers quatre bits inchangés. D'autre part, vous pourriez forcer le bit le moins significatif à 1 en laissant les autres intacts via un OR entre X et la valeur binaire 0000 0001, puis effectuer un XOR entre X et 0000 0100, respectivement. L'utilisation de AND, OR et XOR pour manipuler des chaînes de bits de cette manière est connue comme *masquage de chaînes de bits*. On utilise ce terme parce qu'on peut se servir de certaines valeurs (1 pour AND, 0 pour OR ou XOR) pour "masquer" certains bits et les préserver du changement lors d'une opération qui a pour but de modifier une valeur.

1.7 Nombres signés et non signés

Tout à l'heure, on a considéré les nombres binaires comme des valeurs non signées. Le nombre binaire ... 00000 représente zéro, ...00001 représente un, ...00010 représente deux et ainsi de suite. Que se passe-t-il alors pour les nombres négatifs ? Les valeurs signées ont bien été mentionnées dans les sections précédentes et on a également parlé du système de numération par complément à deux ; mais l'on n'a pas expliqué comment représenter les nombres négatifs en binaire. On va justement le voir !

Pour représenter des nombres signés par le système binaire, on doit placer une restriction à ces nombres : ils doivent avoir un nombre fini et fixe de bits. Cette restriction est naturelle, car les microprocesseurs comme le 80x86 peuvent adresser seulement un nombre fini de bits. Donc, pour représenter les nombres négatifs, nous devons limiter les nombres à des tailles prédéfinies, comme huit, seize, trente-deux bits ou plus encore.

Avec un nombre fixe de bits, on peut seulement représenter un certain nombre d'objets. Par exemple, avec huit bits, l'on peut représenter seulement 256 objets différents. Et les nombres négatifs, sont des objects tout comme les nombres positifs. Par conséquent, avec huit bits, on doit forcément utiliser certaines des 256 valeurs à disposition pour représenter les nombres négatifs. En d'autres termes, on doit sacrifier certains nombres positifs pour pouvoir définir les nombres négatifs. Pour réaliser cela, on peut assigner la moitié du nombre total de possibilités aux nombres négatifs, et laisser l'autre aux nombres positifs. Donc, avec un octet, on peut

[10]Ceci peut toujours se faire, parce qu'il n'y a jamais de retenue dans les opérations logiques, n.d.t.

représenter les valeurs négatives de la plage -128…-1 et les valeurs positives de la plage 0…127[11]. Avec un mot (16 bits), on peut représenter de -32768 à +32767. Avec un double-mot (de 32 bits), notre plage sera [-2147483648, +2147483647]. En général, avec *n* bits, on peut représenter les valeurs signées dans la plage -2^{n-1} à $+2^{n-1}-1$.

Nous avons donc trouvé la façon de représenter les valeurs négatives. Mais, de quelle façon le fait-on exactement ? Il y a divers moyens, mais la technologie 80x86 utilise la complémentation à deux. Dans un tel système, le bit de poids le plus fort d'un nombre est le *bit du signe*. Si ce bit est zéro, le nombre est positif, sinon, il est négatif. Exemples :

Pour des nombres de 16 bits :

8000h est négatif, parce que le bit le plus significatif est à 1.
100h est positif, parce que le bit le plus significatif est à 0.
7FFFh est positif.
0FFFFh est négatif.
0FFFh est positif.

Si le bit fort est zéro, alors le nombre est positif et, par conséquent, il est stocké dans une valeur binaire standard. Si, au contraire, il est à un, alors le nombre est négatif et, par conséquent, il est stocké en forme de complément à deux. Pour convertir un nombre positif dans sa version négative (en forme de complément à deux), vous utiliserez cet algorithme :

1) Inverser tous les bits du nombre, c'est-à-dire, lui appliquer la fonction NOT.
2) Ajoutez 1 au résultat.

Par exemple, pour calculer l'équivalent 8 bits de -5 :

```
0000 0101           Cinq, en binaire.
1111 1010           Après une inversion de tous les bits.
1111 1011           Résultat obtenu après l'ajout de 1.
```

Si on prend ce résultat et on effectue encore un complément à deux, alors on obtient de nouveau notre valeur originale, 00000101, tout comment on s'attendait :

```
1111 1011           Complément à deux de -5.
0000 0100           Après inversion.
0000 0101           Après ajout de 1, nous obtenons +5.
```

Les exemples qui suivent fournissent certaines valeurs positives et négatives de 16 bits :

7FFFh : +32767, le plus gros nombre positif sur 16 bits.
8000h : -32768, le plus petit nombre négatif sur 16 bits.
40000h : 16 384.

Pour changer le signe des nombres ci-dessus, faites ce qui suit :

```
7FFFh :       0111 1111 1111 1111       +32767d.
              1000 0000 0000 0000       Inversion de tous les bits.
              1000 0000 0000 0001       Ajout de 1 (8001h ou –32767d).

8000h :       1000 0000 0000 0000       -32768d.
              0111 1111 1111 1111       Inversion de tous les bits.
              1000 0000 0000 0000       Ajout de 1 (8000h ou –32768d).

4000h :       0100 0000 0000 0000       16384d.
              1011 1111 1111 1111       Inversion de tous les bits.
              1100 0000 0000 0000       Ajout de 1 (0C00h ou –16384d).
```

8000h inversé devient 7FFFh. Après avoir ajouté 1, on obtient 8000h ! Qu'est-ce qui arrive ? -(- 32768) vaudrait donc –32768 ? Certainement non. Mais, la valeur +32768 ne peut pas être représenté avec un nombre signé de 16 bits ; donc, on ne peut pas changer le signe de la valeur la plus petite. Si vous tentez cette opération, les microprocesseurs 80x86 provoqueront un dépassement de capacité (*overflow*).

[11]En théorie, zéro n'est ni négatif ni positif. Pour des raisons techniques (dues à des implications matérielles), on considère zéro comme étant un nombre positif.

Pourquoi donc se tracasser avec un système de numération aussi minable ? Pourquoi ne pas utiliser le bit fort comme un drapeau de signe (*sign flag*) et stocker l'équivalent positif du nombre dans les bits qui restent ? La réponse réside dans le matériel. Il s'avère que changer les signes des nombres est le seul travail ennuyeux. Avec la technique du complément à deux, la plupart des autres opérations se révèlent aussi faciles que travailler en système binaire non signé. Par exemple, supposez devoir effectuer l'addition 5 + (-5). Le résultat est zéro. Considérez ce qui arrive quand on effectue cette addition en mode complément à deux :

```
  00000101
  11111011
  --------
1 00000000
```

On termine avec une retenue dans le neuvième bit et tous les autres bits à zéro. Evidemment, si l'on ignore cette retenue, ajouter deux valeurs signées produit toujours le bon résultat quand on utilise le mode complément à deux. Ce qui veut dire qu'on peut utiliser le même composant matériel pour les additions et pour les soustractions signées et non signées, ce qui ne serait pas possible avec des modes différents.

Les questions à la fin de ce chapitre mises à part, vous n'aurez jamais besoin d'effectuer à la main les opérations en mode complément à deux. Les processeurs 80x86 fournissent une instruction, NEG (negate), qui effectue cette opération pour vous. De plus, toutes les calculatrices hexadécimales sont en mesure de l'effectuer en pressant simplement la touche de changement de signe (+/- ou CHS). Néanmoins, effectuer la complémentation à deux à la main est facile, et vous devez savoir comment le faire.

Une fois encore, vous devez noter que les données représentées par un ensemble de bits dépendent entièrement du contexte. La valeur binaire de huit bits 11000000b pourrait représenter un caractère IBM/ASCII ou bien la valeur décimale non signée 192, ou la valeur décimale signée –64, etc. En tant que programmeur, c'est votre responsabilité d'utiliser cette donnée convenablement.

1.8 Extensions des valeurs signées et non signées

Etant donné que les entiers formatés en mode complément à deux ont une longueur fixe, un petit problème surgit. Qu,est-ce qu'arrive-t-il si vous avez besoin de convertir un entier signé de 8 bits en un entier signé de 16 bits ? Ce problème et son inverse (conversion d'une valeur de 16 bits en une valeur de 8 bits), peut être envisagé par les opérations *d'extension* et de *contraction*. De la même manière, les 80x86 travaillent avec des valeurs d'une longueur fixe, même si nous sommes en train de travailler avec des nombres non signés. L'extension de zéros (*zero extension*) permet de convertir de petites valeurs non signées en des valeurs plus grandes.

Considérez la valeur "-64". La version complémentaire sur 8 bits de cette valeur est 0C0h. L'équivalent 16 bits de ce nombre est 0FFC0h. Maintenant, considérez la valeur "+64". Les versions 8 et 16 bits de cette valeur sont 40h et 0040h. La différence entre les nombres de 8 et de 16 bits peut être décrite par cette règle : "Si le nombre est négatif, l'octet fort de celui-ci contiendra 0FFh ; s'il est positif, son octet fort de sa version 16 bits sera formé par des zéros.

Elargir une valeur signée d'un certain nombre de bits à un format plus grand est facile. Il suffit de copier le bit du signe dans tous les bits additionnels du nouveau format. Par exemple, pour élargir un nombre de 8 bits à un nombre de 16 bits, copiez simplement le bit 7 du nombre de 8 bits dans tous les bits de la plage 8...15. Et pour élargir un nombre de 16 bits à un double-mot, copiez simplement le bit 15 dans tous les bits de la plage 16...31.

L'extension des nombres signés est nécessaire quand on manipule des valeurs signées ayant différentes longueurs. Souvent, vous avez besoin de convertir un octet en un mot. Vous devriez appliquer cette extension avant d'effectuer toute opération avec des nombres plus grands. D'autres opérations (spécialement les multiplications et les divisions) peuvent nécessiter un élargissement allant jusqu'à 32 bits. Naturellement, les valeurs non signées ne sont pas soumises à la même règle que les valeurs signées[12].

Exemples d'extension signée :

```
Huit bits          Seize bits          Trente-deux bits
```

[12]Autrement dit, en élargissant les valeurs considérées d'avance comme non signées, les bits de remplissage doivent toujours valoir zéro, même si le bit fort du petit format était différent de zéro, n.d.t.

```
80h                 FF80h               FFFFFF80h
28h                 0028h               00000028h
9Ah                 FF9Ah               FFFFFF9Ah
7Fh                 007Fh               0000007Fh
---                 1020h               00001020h
---                 8088h               FFFF8088h
```

Pour élargir un octet non signé, il faut *zéro-élargir* la valeur. Ce genre d'extension est très facile, il suffit simplement d'ajouter des zéros de remplissage là où l'opération est requise. Par exemple, pour élargir la valeur non signée de 8 bits 82h à une valeur non signée de 16 bits, ajoutez simplement des zéros dans l'octet fort du mot : 0082h.

```
Huit bits           Seize bits          Trente-deux bits

80h                 0080h               00000080h
28h                 0028h               00000028h
9Ah                 009Ah               0000009Ah
7Fh                 007Fh               0000007Fh
---                 1020h               00001020h
---                 8088h               00008088h
```

Contracter une valeur signée en un format plus petit est un peu plus délicat. Les extensions n'échouent jamais. Etant donnée une valeur signée de *m* bits, vous pourrez toujours la convertir en un nombre de *n* bits (où n > m). Malheureusement, étant donné un nombre de *n* bits, vous ne pouvez pas toujours le convertir en un nombre de *m* bits quand m < n. Par exemple, considérez la valeur -448. Sa représentation hexadécimale sur 16 bits est 0FE40h. Ce nombre a une taille trop grande pour être contenu dans un espace de 8 bits. C'est un exemple de condition de dépassement, qui a lieu lors des opérations de conversion.

Pour comprimer efficacement des valeurs, il faut jeter un coup d'oeil à l'octet ou aux octets forts que vous voulez liquider. Les octets forts que vous voulez supprimer doivent contenir soit des zéros, soit 0FFh. Si vous rencontrez toute autre valeur, vous ne pourrez pas contracter le nombre sans le tronquer. Finalement, l'octet fort de votre valeur résultante doit toujours correspondre à la valeur d'origine. Exemples (16 bits à 8 bits) :

```
FF80h peut être converti en 80h.
0040h peut être converti en 40h.
FE40h ne peut pas être converti en une valeur de 8 bits.
0100h ne peut pas être converti en une valeur de 8 bits.
```

1.9 Décalages et rotations

Un autre ensemble d'opérations logiques - qui s'applique aux chaînes de bits - constitue les opérations de *décalage* et de *rotation*. Les deux catégories peuvent à leur tour se subdiviser en *décalage à gauche*, *rotation à gauche*, *décalage à droite* et *rotation à droite*. Ces opérations s'avèrent extrêmement utiles à la programmation en assembleur.

Le décalage à gauche déplace chaque bit d'une chaîne d'une position vers la gauche (voir figure 1.8).

Figure 1.8 Opération de décalage à gauche

Le bit zéro se déplace à la position 1, la valeur qui était à la position 1 passe à la position 2 et ainsi de suite. Il y a sans doute deux questions qui surgissent naturellement : « Qu'est-ce qui arrive à la position du bit zéro ? » et « Qu'est-ce qui devient le bit qui occupait la position sept ? ». Eh bien, ceci dépend du contexte. Le bit à la position zéro prendra la valeur 0 et le bit de la position 7 sera perdu.

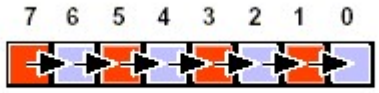
Noter que décaler une valeur vers la gauche, c'est faire la même chose que la multiplier par sa base. Par exemple, décaler un nombre décimal d'une position vers la gauche (ajouter un zéro à la droite du nombre) équivaut à multiplier ce nombre par dix (sa base) :

```
1234 SHL 1 = 12340  (SHL 1 = décaler d'une position vers la gauche)
```

Puisque la base d'un nombre binaire est deux, le décaler vers la gauche c'est le multiplier par deux. Et si vous décalez une valeur binaire vers la gauche deux fois, vous la multipliez en fait par 2 deux fois (autrement dit par quatre). Et si vous le faites trois fois, vous la multipliez par huit (2*2*2). En général, si vous décalez une valeur par la gauche *n* fois, vous multipliez cette dernière par 2^n.

Un décalage vers la droite fonctionne de la même manière, sauf qu'on déplace les bits vers la direction opposée. Le bit 7 passe au bit 6, le bit 6 au bit 5, le bit 5 au bit 4, etc. Pendant un décalage vers la droite, le bit 7 prend la valeur 0 et le bit zéro est perdu (voir figure 1.9).

Figure 1.9 Opération de décalage à droite



Puisqu'un décalage vers la gauche est l'équivalent de la multiplication par deux, il ne devrait pas surprendre qu'un décalage vers la droite soit l'équivalent d'une division par deux (ou en général d'une division par la base du nombre). Si vous exécutez *n* décalages à droite, vous divisez pratiquement ce nombre par 2^n.

Il y a pourtant un problème avec les décalages à droite : comme décrit ci-dessus, un décalage vers la droite est équivalent à une division *non signée* par deux. Par exemple, si vous décalez la représentation non signée de 254 (0FEh) d'une position vers la droite, vous obtiendrez 127 (07Fh), exactement ce que vous vous attendiez. Cependant, si vous décalez la représentation binaire de -2 (0FEh) vers la droite d'une position, vous obtiendrez encore 127 (07Fh), qui *n'est pas* correct. Ce problème se produit parce que cette opération comporte l'insertion d'un zéro dans le bit sept. Si la valeur qu'il contenait était le bit du signe valant 1, alors le nombre est devenu positif. Pas un bon résultat quand on divise par deux.

Pour utiliser le décalage à droite comme un opérateur de division, on doit définir une troisième opération de décalage : *le décalage à droite arithmétique*[13]. Un décalage arithmétique fonctionne tout comme un décalage normal (un *décalage logique à droite*) avec une exception : au lieu de placer zéro dans le bit sept il le laisse simplement dans son état initial sans le changer, comme montré à la figure 1.10.

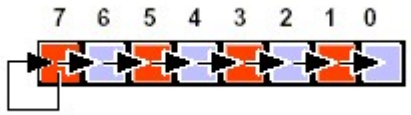


Figure 1.10 Opération de décalage à droite arithmétique

Ceci produit généralement le résultat attendu. Par exemple, si vous effectuez le décalage arithmétique à droite de –2 (0FEh) vous obtiendrez –1 (0FFh). Gardez cependant une chose à l'esprit. Cette opération arrondit toujours les nombres à l'entier le plus proche *inférieur ou égal au résultat actuel*. En se basant sur leur expérience avec les langages de haut niveau et les règles standard de troncation, beaucoup de gens prennent pour acquis qu'une division arrondit toujours vers zéro. Mais ce n'est simplement pas le cas. Par exemple, si vous appliquez le décalage arithmétique sur –1 (0FFh), le résultat sera –1 et non zéro. –1 est inférieur à 0, donc l'opération de décalage arithmétique arrondit le résultat à –1. Ceci n'est pas une bogue dans le décalage arithmétique. C'est la manière dont les divisions entières sont normalement effectuées sur une machine. Et le 80x86 ne fait pas d'exception.

[13]Nous n'avons pas besoin d'un décalage arithmétique gauche. L'opération standard de décalage à gauche fonctionne pour les nombres signés et non signés.

Autres deux opérations utiles sont la *rotation vers la gauche* et la *rotation vers la droite*. Ces opérations ressemblent aux opérations de décalage, mais avec une différence principale : le bit balayé d'une extrémité revient dans l'autre.

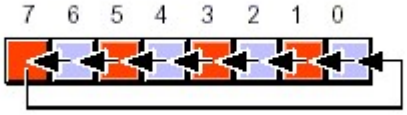


Figure 1.11 Opération de rotation à gauche

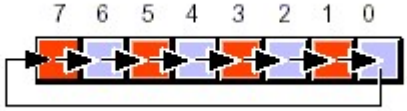
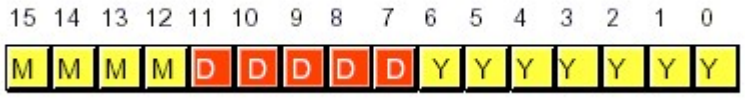


Figure 1.12 Opération de rotation à droite

1.10 Champs de bits et compactage de données

Quoique la technologie 80x86 travaille de façon optimale avec des octets, des mots et des doubles-mots, vous aurez occasionnellement besoin de travailler avec un type de données qui se sert d'un nombre d'octets autre que huit, seize ou trente-deux. Par exemple, considérez une date dans le format : "4/2/88". Pour représenter cette date, il faut trois valeurs numériques : une valeur pour le mois, une pour le jour et une pour l'année[14]. Les mois, sans doute, prennent les valeurs de 1 à 12 et donc demandent au mois quatre bits (pour un maximum de 16 valeurs différentes). Les jours vont de 1 à 31, donc cinq bits pourraient suffire (pour un maximum de 32 valeurs représentables). Et, finalement, les années, en présumant adopter la plage 0...99, demandent sept bits (qu'on peut utiliser pour représenter jusqu'à un maximum de 128 valeurs). Quatre plus cinq plus sept, fait 16 bits, ou deux octets. En d'autres termes, on peut compacter notre date en deux octets au lieu de trois (comme on aurait dû faire si on utilisait un octet pour chaque valeur de mois, de jour et d'année). Ceci économise un octet de mémoire pour chaque date stockée, ce qui peut représenter un gain substantiel d'espace si vous avez à travailler avec beaucoup de dates. Les bits peuvent être arrangés selon la figure 1.13 :

Figure 1.13 Format de données compactées



MMMM représente les quatre bits nécessaires à représenter la valeur du mois, DDDDD représente les cinq bits pour la valeur du jour et YYYYYYY représente les sept bits pour l'année. Chaque groupe de bits représentant un objet est un *champ de bits* (*bit field*). Une date comme "4/2/88" serait donc exprimée par 4158h :

```
0100 00010 1011000  = 0100 0001 0101 1000b ou 4158h
 4     2      88
```

Quoique les données compactées soient efficaces du point de vue de l'espace (elles occupent moins d'espace en mémoire), elles sont inefficaces du point de vue calcul (lenteur!). La raison ? Des instructions additionnelles sont nécessaires pour décompacter (*unpack*) les données rangées dans les champs de bits. Ces instructions supplémentaires requièrent plus de temps pour s'exécuter (et des octets additionnels pour stocker les

[14]Notez qu'aux Etats-Unis, les dates s'écrivent dans le format mois-jour-année. J'ai voulu garder cette représentation pour ne pas altérer les exemples et les exercices, n.d.t.

instructions elles-mêmes), donc vous devez soigneusement considérer quelles données compactées dans des champs peuvent vous faire économiser quelque chose.

Les exemples de paquets de données utiles abondent. Vous pourriez compacter huit valeurs booléennes dans un seul octet, vous pouvez également compacter des codes BCD (binaire codé décimal), etc.

1.11 Le jeu de caractères ASCII

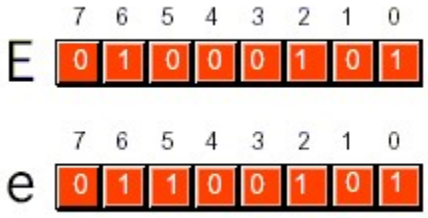
Le jeu de caractères ASCII (en excluant la version étendue définie par IBM), est divisé entre quatre groupes de 32 caractères. Le premier groupe, les codes ASCII allant de 0 à 1Fh (31), constitue un ensemble spécial de caractères non imprimables, qu'on appelle caractères de contrôle. On les appelle de cette manière parce que, au lieu d'afficher des caractères ou des symboles, ils réalisent diverses opérations de contrôle d'affichage. Des exemples comprennent le retour chariot qui positionne le curseur au bord gauche de la ligne courante[15], le "saut de ligne" (*line feed*), qui déplace le curseur vers le bas d'une ligne à la fois) et l'effacement arrière (*backspace*), qui fait retourner le curseur d'une position vers la gauche et efface le caractère qui s'y trouvait. Malheureusement, différents caractères de contrôle effectuent différentes opérations sur différents périphériques de sortie. Il y a pourtant une très petite standardisation. Pour savoir comment un caractère de contrôle affecte exactement un périphérique particulier, vous aurez besoin de consulter son manuel.

Le second groupe de 32 codes de caractères ASCII comprend divers symboles de ponctuation, de caractères spéciaux et de chiffres numériques. Les caractères de ce groupe les plus remarquables sont le caractère d'espacement (code ASCII 20h) et les valeurs numériques (codes ASCII de 30h à 39h). Notez que les chiffres diffèrent de leurs valeurs numériques correspondante seulement dans le quartet fort. En soustrayant 30h du code ASCII de tout nombre vous pouvez obtenir l'équivalent numérique de celui-ci.

Le troisième groupe est réservé aux caractères alphabétiques majuscules. Les codes ASCII pour les caractères de A à Z se trouvent dans la plage 41h...5Ah (65...90). Puisqu'il y a seulement 26 différents caractères alphabétiques, les six codes restants hebergent des symboles spéciaux.

Le quatrième et dernier groupe de 32 codes ASCII est réservé aux lettres alphabétiques minuscules, à cinq symboles spéciaux additionnels et à un autre caractère de contrôle (delete). Notez que les symboles minuscules utilisent les codes ASCII de 61h à 7Ah. Si vous convertissez les codes des majuscules et des minuscules en binaire, vous remarquerez que les lettres majuscules diffèrent des lettres minuscules exactement par une position binaire. Par exemple, considérez les codes des caractères "E" et "e" dans la figure 1.14 :

Figure 1.14 Les codes ASCII pour les lettres "E" et "e".



La seule place où ces codes diffèrent est le bit 5. Les caractères majuscules contiennent toujours un zéro dans la position du bit 5. Vous pouvez exploiter ce fait pour facilement convertir entre lettres majuscules et minuscules. Si vous avez un caractère capital, vous pouvez le forcer à devenir minuscule en mettant le cinquième bit à 1. Ou bien, si vous avez un caractère minuscule et vous voulez le forcer à devenir capital, vous pouvez le faire en mettant le bit 5 à zéro. Vous pouvez faire passer un caractère d'un état à l'autre en invertissant simplement le cinquième bit.

Les bits 5 et 6 vous permettent aussi de savoir rapidement dans lequel des quatre groupes des caractères ASCII vous vous trouvez :

[15]Historiquement, le terme "retour chariot" (emprunté de l'anglais *carriage return*) se réfère au chariot du papier dans la machine à écrire. Un retour chariot consistait à mouvoir physiquement le chariot vers l'extrême droite, de sorte que le prochain caractère tapé aurait été imprimé sur la marge gauche de la feuille.

Bit 6	Bit 5	Groupe
0	0	Caractères de contrôle
0	1	Chiffres et ponctuation
1	0	Majuscules et spéciaux
1	1	Minuscules et spéciaux

Donc vous pouvez, par exemple, convertir toute minuscule ou majuscule (ou caractère spécial correspondant) en un caractère de contrôle occupant une position respective en mettant les bits 5 et 6 à zéro.

Considérez un moment les codes ASCII des carctères numériques :

"0"	48	30h
"1"	49	31h
"2"	50	32h
"3"	51	33h
"4"	52	34h
"5"	53	35h
"6"	54	36h
"7"	55	37h
"8"	56	38h
"9"	57	39h

La représentation décimale de ces codes n'est pas très éclaircissante. Cependant leur représentation hexadécimale révèle quelque chose de très important - le quartet faible du code ASCII est l'équivalent binaire du nombre représenté. En liquidant (c'est-à-dire en mettant à zéro) le quartet fort, vous pourrez convertir ce code en sa représentation binaire. Inversement, vous pouvez convertir une valeur binaire de la plage 0…9 en sa représentation ASCII simplement en mettant le quartet fort à trois. Remarquez que vous pouvez utiliser AND pour forcer les bits forts à valoir zéro ; de même, vous pouvez utiliser OR pour mettre le quartet fort à trois (0011).

Notez que vous ne pouvez pas convertir une chaîne de caractères numériques en leur représentation binaire équivalente en modifiant simplement le quartet fort de chaque chiffre de la chaîne. Convertir 123 (31h, 32h, 33h) de cette manière donne trois octets : 010203h et non le résultat correct, qui serait 7Bh. Convertir une chaîne de bits en un entier binaire demande plus de travail que cela ; la conversion ci-dessus fonctionne seulement pour des unités isolées.

Le septième bit dans le standard ASCII vaut toujours zéro. Ceci veut dire que le jeu de caractères occupe seulement la moitié des possibles codes de caractères qu'un octet pourrait contenir. IBM utilise les 128 codes restants pour divers caractères spéciaux, incluant les caractères internationaux (caractères accentués etc.), les symboles mathématiques et les caractères de traçage des lignes. Notez que ces caractères additionnels sont une extension non standard du jeu de caractères ASCII. Sans aucun doute, le nom IBM a une influence considérable, donc au moins tous les ordinateurs personnels modernes basés sur 80x86 et sur un affichage supportent le jeu de caractères ASCII étendu. Et il en va de même pour la plupart des imprimantes.

Dans l'éventualité où vous deviez échanger des données avec d'autres machines qui ne sont pas compatibles avec le PC, vous aurez deux alternatives : ou bien vous vous limitez au code ASCII standard ou bien vous vous assurez que la machine cible supporte le code ASCII étendu. Certaines machines, comme Apple Machintosh ne fournissent pas un support clé en main à l'ensemble du jeu de caractères étendu de l'IBM PC ; cependant, vous pouvez toujours obtenir une police vous permettant d'afficher ce jeu. D'autres machines (comme l'Amiga et l'Atari ST) ont des possibilités semblables. Mais, en définitive, le standard ASCII de 128 caractères est le seul code sur lequel vous pouvez compter pour un transfert fiable de système à système.

Malgré le fait que c'est un « standard », encoder vos données en utilisant simplement des caractères ASCII conventionnels, ne garantit néanmoins pas la compatibilité entre tous les systèmes. Si en règle générale un "A" dans une machine sera tout probablement un "A" dans une autre, il n'y a pas par contre beaucoup de normalisation pour ce qui concerne les caractères de contrôle. Des 32 codes de contrôles plus delete, il y en a seulement quatre d'universellement supportés : backspace (BS), tab, carriage return (CR) et line feed (LF). Et comme s'il ne suffisait pas, différentes machines utilisent souvent ces contrôles de différentes manières. La fin de ligne est un exemple particulièrement tracassant. MS-DOS, CP/M et d'autres systèmes le marquent avec la séquence des deux caractères CR/LF. Apple, Machintosh, Apple II et certains autres systèmes le marquent avec le seul CR. Les systèmes UNIX le font avec un simple LF. Inutile de dire que chercher d'échanger des fichiers de texte simple entre ces systèmes peut s'avérer une expérience bien frustrante. Même si vous utilisez des caractères ASCII standard dans tous vos fichiers, vous aurez encore besoin de convertir des données. Heureusement, de telles conversions sont simples.

Malgré quelques défauts majeurs, les codes ASCII sont le standard pour l'échange des données entre les systèmes et les programmes. La plupart de ces derniers acceptent des données ASCII ; de la même façon, beaucoup de programmes peuvent également en produire. Puisque, en assembleur, vous aurez affaire aux caractères ASCII, il serait sage d'en étudier la disposition et d'en mémoriser un petit nombre (comme, "0", "A", "a", etc.).

1.12 Résumé

La plupart des systèmes informatiques modernes utilisent le système de numération binaire pour représenter des valeurs. Puisque les nombres binaires sont quelque peu difficiles à manier, on utilise souvent la représentation hexadécimale. Ceci parce qu'il est très facile de convertir entre nombres hexadécimaux et binaires, ce qui n'est pas le cas entre le système décimal (plus familier) et le système binaire. Un simple chiffre hexadécimal représente quatre unités binaires (bits), et on appelle *quartet* un groupe de quatre bits. Voir :

- *Le Système de numération binaire*, au paragraphe 1.1.2
- *Les formats binaires*, au paragraphe 1.1.3
- *Le système de numération hexadécimal*, au paragraphe 1.3

Les processeurs 80x86 travaillent mieux avec des groupes de huit, seize et trente-deux bits. On appelle les objets de cette taille *octets*, *mots* et *doubles-mots* respectivement. Avec un octet, on peut représenter 256 valeurs uniques. Avec un mot, on peut représenter 65536 valeurs différentes. Avec un double-mot on peut représenter plus de quatre milliards de valeurs. Souvent, on représente des valeurs entières (signées ou non signées) avec des octets, des mots ou des doubles-mots ; cependant toute autre quantité peut être représentée aussi facilement. Voir :

- *Organisation des données*, au pargraphe 1.2
- *Octets*, au paragraphe 1.2.3
- *Mots*, au paragraphe 1.2.4
- *Doubles-mots*, au paragraphe 1.2.5

Pour pouvoir nous référer à des bits spécifiques à l'intérieur d'un quartet, d'un octet, d'un mot, d'un double-mot ou d'une autre structure encore, on numérote les bits à partir de zéro (le bit le moins significatif) jusqu'à n-1 (où *n* est le nombre de bits dont objet est constitué). On peut aussi numéroter de façon similaire les quartets, les octets et les mots à l'intérieur d'une plus grande structure. Voir :

- *Les formats binaires*, au paragraphe 1.1.3

Il y a diverses opérations que l'on peut effectuer sur les valeurs binaires, incluant l'arithmétique ordinaire (+, -, * et /) et les opérations logiques (AND, OR, XOR, NOT, décalages à gauche et à droite (shift left, shift right) et les rotations à gauche et à droite (rotate left, rotate right)). Les opérations AND, OR, XOR et NOT sont normalement définies pour des opérations entre deux bits isolés. On peut les élargir à *n* bits en les traitant de bit à bit (ou bitwise). Les opérations de décalage et de rotation sont toujours définies pour une longueur fixe de chaînes de bits. Voir :

- *Opérations arithmétiques sur les nombres binaires et hexadécimaux*, au paragraphe 1.4
- *Opérations logiques sur les bits*, au paragraphe 1.5
- *Opérations logiques sur les nombres binaires et les vhaînes de bits*, au paragraphe 1.6
- *Décalages et rotations*, au paragraphe 1.9

Il y a deux types de valeurs entières qu'on peut représenter avec des chaînes binaires sur 80x86 : entiers signés et non signés. Le 80x86 représente les nombres non signés à l'aide du format binaire standard. Mais il représente les entiers signés à l'aide du format de complément à deux. Alors que les nombres non signés peuvent être d'une longueur arbitraire, les nombres signés ont du sens seulement avec des longueurs fixes. Voir :

- *Nombres signés et non signés*, au paragraphe 1.7
- *Extensions des valeurs signées et non signées*, au paragraphe 1.8

Parfois, il peut ne pas être pratique de stocker des données en groupes de huit, seize ou trente-deux bits. Pour économiser de l'espace vous pourriez vouloir compacter différentes pièces de données dans le même octet, mot ou double-mot. Ceci reduit les exigences d'espace, mais aux frais d'avoir à exécuter des opérations supplémentaires pour compacter ou décompacter. Voir :

- *Champs de bits et compactage de données*, au paragraphe 1.10

Les données caractère sont probablement le type d'objet le plus commun que l'on trouve en arrière des valeurs entières. L'IBM/PC et compatibles utilisent une variante du jeu de caractères ASCII - le jeu ASCII/IBM étendu. Les premiers 128 caractères de cette catégorie sont les caractères ASCII standard et les 128 codes restants sont des caractères spéciaux créés par IBM pour les langues internationales, les mathématiques et le traçage des lignes. Puisque l'usage du jeu de caractères ASCII est aussi commun dans les programmes modernes, une certaine familiarité avec cet ensemble est essentielle. Voir :

- *Le jeu de caractères ASCII*, au paragraphe 1.11

1.13 Exercices de laboratoire

Ce livre est accompagné par un support logiciel significatif. Ce dernier est divisé en quatre catégories de base : le code source des exemples du texte, la bibliothèque standard UCR pour programmer en assembleur 80x86, des échantillons de code que vous aurez à modifier pour divers exercices de laboratoire et des applications logicielles servant à assister ces exercices. Ces applications ont été écrites en assembleur, en C++, Flex/Bison et Dephi (Pascal orienté objets). Beaucoup de ces applications incluent le code source autant que les fichiers exécutables.

La plupart des logiciels accompagnant ce livre fonctionnent sur Windows 3.1, Windows 9x ou Windows NT. Certains autres, cependant, manipulent directement le matériel et s'exécutent seulement sous DOS ou sous une fenêtre DOS dans Windows 3.1. Cet ouvrage présume que vous êtes familiarisé avec ces systèmes d'exploitation ; si ce n'est pas le cas et vous désirez plus de détails, veuillez vous référer à un ouvrage approprié.

1.13.1 Installation des logiciels

Les logiciels qui accompagnent cet ouvrage sont généralement offerts sur cédérom[16]. Vous pouvez donc vous servir de la plupart directement à partir du CD. Cependant, pour plus de facilité, vous voudrez probablement installer tout le paquetage sur votre disque dur[17]. Pour faire ceci, vous aurez besoin de créer deux sous-répertoires : ARTOFASM et STDLIB. Le premier contiendra les fichiers spécifiques pour ce livre, le second sera

[16]Ils sont également disponibles via ftp anonyme, quoiqu'il y a divers fichiers associés à ce texte.

pour les fichiers associés à la bibliothèque standard UCR pour les machines 80x86. Une fois ces répertoires créés, copiez-y tous les fichiers et les sous-répertoires à partir des répertoires analogues sur le CD. Ou bien, à partir du DOS ou d'une fenêtre DOS de Windows vous pouvez utiliser la commande XCOPY :

```
xcopy r:\artofasm\*.*      c:\artofasm   /s
xcopy r:\stdlib\*.*        c:\stdlib     /s
```

Ces lignes présument que votre lecteur de CD ait la lettre R: et que vous soyez en train d'installer vos logiciels sur le lecteur C: Elles présument aussi que vous avez déjà créé les dossiers ARTOFASM et STDLIB avant d'exécuter XCOPY.

Pour utiliser la bibliothèque standard avec les projets de programmation vous aurez besoin d'ajouter ou de modifier deux lignes dans votre fichier AUTOEXEC.BAT. Si de semblables lignes n'y sont déjà présentes, ajoutez ceci :

```
set lib=c:\stdlib\lib
set include=c:\stdlib\include
```

Ces commandes indiquent à MASM (Microsoft Macro Assembler) où il peut trouver la bibliothèque standard UCR et les fichiers d'inclusion pour celle-ci. Sans ces lignes, MASM produira une erreur toutes les fois où, dans vos programmes, vous utiliserez les routines de la bibliothèque.

S'il y a déjà "set include=..." et "set lib=..." dans AUTOEXEC.BAT, vous n'aurez pas à les remplacer. Mais vous devrez ajouter la chaîne ";c:\stdlib\lib" à la fin du "set lib=..." et ";c:\stdlib\include" à la fin du "set include=...". Certains langages (comme le C++) utilisent également ces variables d'environnement "set" ; si vous les remplacez arbitrairement par les variables ci-dessus, vos programmes en assembleur fonctionneront bien, mais toute tentative de compiler un programme C++ (ou d'un autre langage) pourrait échouer. Si vous oubliez d'insérer ces lignes dans votre AUTOEXEC.BAT, vous pourrez encore activer ces commandes temporairement (tant que l'ordinateur ne sera pas éteint) en les tapant simplement à l'invite DOS. En tapant *set*, vous pourrez voir si ces commandes sont actives ou non.

Si vous n'avez pas un lecteur de CD, vous pouvez obtenir les logiciels associés à ce livre par un site ftp anonyme depuis cs.ucr.edu[18]. Regardez dans le sous-répertoire "pub/pc/ibmpc". Les fichiers dans le serveur ftp sont compressés. Un fichier "README" vous décrira comment les décompresser[19].

Le dossier STDLIB que vous avez créé, contient le code source et les fichiers pour la bibliothèque UCR standard. Il s'agit d'un ensemble essentiel de sous-routines en assembleur que vous pouvez appeler tout comme on appelle les fonctions de la bibliothèque standard du C. Ces routines simplifient grandement l'écriture des programmes. De plus, elles sont du domaine public, par conséquent vous pouvez les utiliser à loisir, sans restrictions de license, dans tout programme que vous écrirez.

Le dossier ARTOFASM contient des fichiers spécifiques au texte. A l'intérieur de ce répertoire, vous verrez une séquence de sous-dossiers, nommés ch1, ch2, ch3, etc. Ces sous-dossiers contiennent les fichiers associés aux chapitres 1, 2, 3, etc. Dans certains de ces sous-répertoires, vous trouverez autres deux sous-dossiers nommés "DOS" et "WINDOWS". Si ces dossiers sont présents, ils séparent les fichiers qui doivent s'exécuter sous MS-Windows des fichiers qui doivent s'exécuter sous DOS. Beaucoup des programmes du répertoire DOS *demandent un environnement de mode de lecture* "read-mode" *et n'exécutent pas sous le DOS d'une fenêtre Windows 95 ou Windows NT*. Vous aurez alors besoin d'exécuter les logiciels de ce répertoire directement en DOS pur. Les applications Windows requièrent un écran couleur.

Il y a souvent un troisième sous-répertoire dans chaque répertoire des chapitres : SOURCES. Celui-ci contient le listing des codes sources (quand ceci s'applique) pour les logiciels du chapitre. Beaucoup de ces applications sont écrites en assembleur via MASM 6.x, C++ générique, Turbo Pascal ou Borland Delphi (visual object Pascal). Si vous avez un intérêt à voir comment ces logiciels fonctionnent, vous pourrez jeter un coup d'oeil à ce sous-répertoire.

---

[17]Si vous êtes en train de vous servir de ces logiciels dans un laboratoire d'école, votre professeur les a probablement déjà installés. Comme règle générale, vous ne devriez jamais installer quoi que ce soit dans un laboratoire. Renseignez-vous auprès de votre professeur avant de prendre toute décision.

[18]Mais également par le site web du livre, http://webster.cs.ucr.edu/index.html, n.d.t.

[19]Veuillez noter cependant, que toutes ces informations ne seront pas disponibles en français, et il en va de même pour le support logiciel, n.d.t.

Cet ouvrage présume que vous connaissez déjà comment exécuter des programmes en mode DOS et Windows et que vous êtes familiarisé avec la terminologie commune de ces systèmes. Il présume également que vous connaissez certaines commandes simples du DOS, comme DIR, COPY, DEL, RENAME et ainsi de suite. Si vous êtes novice avec Windows ou avec DOS, vous devriez vous procurer un manuel de référence approprié.

Les fichiers des exercices de laboratoire du chapitre 1 apparaissent dans ARTOFASM\CH1. Ce sont des programmes sur environnement Windows, donc vous aurez besoin de Windows 3.1, 95 ou NT ou quelques version plus récente(et compatible) pour les exécuter.

1.13.2 Exercices de conversion des données

Dans ces exercices, vous utiliserez le programme "convert.exe" situé dans le sous-répertoire ARTOFASM\CH1. Ce programme affiche et convertit des entiers de 16 bits en utilisant la notation décimale, décimale non signée, hexadécimale et binaire.

Quand vous exécutez ce programme, il ouvre une fenêtre avec quatre champs de texte (une pour chaque type de données). Changer une valeur dans un champ provoque la mise à jour immédiate des autres champs, de sorte qu'ils affichent leur représentation correspondante de la nouvelle valeur. Si vous commettez une erreur en entrant une donnée, le programme émet un beep et colore le champ en rouge jusqu'à ce que vous ne l'ayez pas corrigée. Notez que vous pouvez utiliser la souris, les touches de contrôle du curseur et les touches d'édition (par exemple, del et backspace) pour changer les valeurs individuelles des champs de texte.

Pour cet exercice et votre rapport de laboratoire, vous devrez explorer la relation entre les diverses valeurs binaires, hexadécimales, décimales non signées et décimales signées. Par exemple, vous pourriez entrer les valeurs décimales 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16 384 et 32 768 et remarquer les valeurs qui apparaissent dans les autres champs.

Le premier objectif de cet exercice est de vous familiariser avec les équivalents décimaux de certaines valeurs binaires et hexadécimales communes. Dans votre rapport, par exemple, vous pourriez expliquer qu'est-ce qu'il y a de spécial avec les équivalents binaires (et hexadécimaux) des nombres décimaux ci-dessus.

Un autre groupe d'expériences possibles est d'essayer de choisir divers nombres binaires ayant exactement un ensemble de deux bits, par exemple, 11, 110, 1100, 11000, 110000, etc. Assurez-vous de noter les résultats décimaux et hexadécimaux que ces entrées produisent.

Essayez d'entrer divers nombres binaires dont les huit bits faibles sont à zéro. Annotez les résultats dans votre rapport de laboratoire. Tentez la même expérience avec les nombres hexadécimaux, en utilisant zéro pour le premier ou les deux premiers chiffres faibles.

Dans le champ des nombres décimaux signés vous devriez également expérimenter les nombres négatifs ; essayez d'utiliser des valeurs comme -1, -2, -3, -256, -1024, etc. Expliquez les résultats que vous obtenez utilisant vos connaissances du mode de complément à deux.

Essayez d'entrer des nombres pairs et impairs dans le champ des décimaux non signés. Découvrez et décrivez la différence entre nombres pairs et impairs dans leur représentation binaire. Tentez d'entrer des multiples d'autres valeurs (par exemple, de trois : 3, 6, 9, 12, 15, 18, 21, ...) et voyez si vous pouvez détecter un schéma dans les résultats binaires.

Vérifiez les conversions *hexadécimal* <-> *binaire* que ce chapitre décrit. En particulier entrez le même chiffre hexadécimal dans chacune des quatre positions d'une valeur de 16 bits et prenez note sur la position des bits correspondants de la représentation binaire. Essayez d'entrer différentes valeurs binaires, comme 1111, 11110, 111100, 1111000 et 11110000. Expliquez les résultats que vous obtenez et décrivez pourquoi, avant d'effectuer une conversion, vous devriez élargir les valeurs binaires de sorte que leur longueur soit toujours un multiple de quatre.

Dans votre rapport de laboratoire, listez les expériences ci-dessus plus d'autres que vous tenterez vous-mêmes. Expliquez les résultats que vous vous attendez en incluant le résultat que le programme convert.exe aura produit. Expliquez toute astuce que vous pourriez rencontrer en vous servant de ce programme.

1.13.3 Exercices sur les opérations logiques

Le programme logical.exe est un simple calculateur qui effectue diverses fonctions logiques. Il accepte des valeurs binaires ou hexadécimales et puis, il calcule le résultat ou bien il exécute certaines opérations logiques sur les entrées. Le calculateur supporte les opérations dyadiques AND, OR, XOR ou monadiques NOT, NEG (deux compléments), SHL (décalage à gauche), SHR (décalage à droite), ROL (rotation à gauche), ROR (rotation à droite).

Quand vous exécutez logical.exe, il affiche une série de boutons sur le côté gauche de la fenêtre. Ces boutons permettent de sélectionner le type d'opération désiré. Par exemple, presser le bouton AND indique au calculateur d'effectuer un AND entre les deux valeurs d'entrée. Si vous sélectionnez une opération monadique (unaire), comme NOT, SHL, etc., alors vous aurez à entrer seulement une valeur ; pour les opérations dyadiques, les deux champs de texte seront actifs.

logical.exe permet, en outre, d'entrer des valeurs binaires ou hexadécimales. Notez que ce programme modifie automatiquement le champ des nombres hexadécimaux quand tout changement au champ des nombres binaires se produit. Et la même chose arrive dans le champ des binaires, si un changement de la valeur hexadécimale a lieu. Si vous entrez une valeur incorrecte dans un champ comme dans l'autre, celui-ci devient rouge tant que vous n'aurez pas corrigé l'erreur.

Pour cet exercice de laboratoire, vous devrez explorer chacune des opérations logiques. Essayez diverses expériences en choisissant soigneusement certaines valeurs, calculez manuellement le résultat que vous vous attendez, puis, exécutez votre calcul dans le programme logical.exe pour vérifier vos résultats. Vous devriez spécialement expérimenter les capacités de masquage des opérations AND, OR et XOR. Essayez d'effectuer ces opérations sur différentes valeurs, comme 000F, 00FF, 00F0, 0FFF, FF00, etc. Notez les résultats et mentionnez-les dans votre rapport de laboratoire.

Certaines expériences que vous pourriez essayer, sont :

- Concevez un masque pour convertir les valeurs ASCII de 0 à 9 en leur contrepartie entière binaire, en utilisant l'opération AND. Pendant que vous utiliserez le masque, essayez d'entrer les codes ASCII de chacune de ces unités. Décrivez vos résultats. Qu'arrive-t-il si vous entrez des codes ASCII qui ne codifient pas des nombres ?
- Concevez un masque pour convertir des valeurs entières de la plage 0...9 en leurs codes ASCII correspondants, via une opération OR. Entrez chacune des valeurs binaires de cette plage et décrivez les résultats. Qu'est-ce qu'il arrive si vous entrez des valeurs en dehors des bornes 0-9 ? En particulier, qu'arrive-t-il si vous entrez des valeurs en dehors de 0h...0Fh ?
- Concevez un masque qui détermine quand une valeur entière de 16 bits est positive ou négative à l'aide de l'opération AND. Le résultat devrait être 0 si l'entier est positif (ou nul) et une valeur différente de zéro si le nombre est négatif. Entrez plusieurs valeurs positives et négatives pour tester votre masque. Expliquez comment vous pourriez utiliser AND pour vérifier le signe de tout bit isolé.
- Concevez un masque pour effectuer une opération logique XOR qui produise le même résultat qu'on avait appliqué un NOT à la seconde opérande.
- Vérifiez que les opérations SHL et SHR correspondent à une multiplication entière par deux et à une division entière par deux, respectivement. Qu'arrive-t-il arrive si vous décalez les données au-delà du bit fort ou du bit faible ? En termes de multiplication et de division entières à quoi cela correspond ?
- Appliquez l'opération ROL sur un ensemble de nombres positifs et négatifs. En vous basant sur l'observation du paragraphe 1.13.3, qu'est-ce que vous pourriez dire à propos du résultat quand vous faites la rotation à gauche d'un nombre positif ou d'un nombre négatif ?
- Appliquez les opérations NEG et NOT sur une valeur. Discutez sur les similarités et les différences des résultats. Décrivez-les en vous basant sur votre connaissance du mode complément à deux.

1.13.4 Exercices sur les extensions signées et non signées

Le programme signext.exe accepte des valeurs binaires ou hexadécimales de huit bits et les élargit à des valeurs de 16 bits conformément à leur signe. Comme logical.exe, ce programme accepte une valeur en format soit binaire soit hexadécimal et il lui applique immédiatement l'extension appropriée.

Pour votre rapport de laboratoire, donnez diverses valeurs d'entrée de huit bits et décrivez les résultats que vous obtenez. Exécutez ces valeurs dans signext.exe et vérifiez ce qui se produit. Pour chaque valeur que vous testez, assurez-vous de lister tous les résultats. Et assurez-vous également d'essayer des valeurs comme 0, 7Fh, 80h et 0FFh.

En exécutant des valeurs, découvrez quelles unités hexadécimales apparaissant dans le quartet fort constituent des nombres négatifs de 16 bits et lesquelles constituent des nombres positifs de 16 bits. Documentez l'ensemble dans votre rapport.

Entrez des ensembles de valeurs comme (1, 10), (2, 20), (3, 30), ..., (7, 70), (8, 80), (9, 90), (A, A0), ... (F, F0). Expliquez les résultats que vous obtenez. Pourquoi "F" s'élargit avec des 0 et "F0" avec des 1 ?

Expliquez dans votre rapport de laboratoire comment l'on pourrait élargir les valeurs de 16 bits à des valeurs de 32 bits. Expliquez pourquoi les extensions signées ou non signées sont utiles.

1.13.5 Exercices sur le compactage des données

Le programme packdata.exe utilise le type de date qu'on a vu dans ce chapitre (regardez le paragraphe 1.10). Il accepte une valeur de date en format binaire ou décimal et l'empaquette dans une valeur de 16 bits.

Quand vous exécutez ce programme, vous verrez une fenêtre avec six champs de texte pour l'entrée de la date : trois pour l'entrer en format décimal et trois pour l'entrer en format binaire. La valeur du mois doit être dans la plage 1...12, celle du jour dans la plage 1...31 et celle de l'année dans la plage 0...99. Si vous entrez une valeur en dehors de ces plages (ou toute autre valeur incorrecte), le champ incriminé deviendra rouge comme toujours, jusqu'à ce que vous n'aurez pas corrigé l'erreur.

Choisissez diverses dates et effectuez d'abord la conversion manuellement (si vous avez des problèmes avec la conversion décimal/binaire, utilisez le programme de conversion utilisé dans le premier ensemble d'exercices de ce chapitre). Successivement, entrez ces valeurs dans le programme pour vérifier vos réponses. Assurez-vous d'inclure tous les résultats dans votre rapport.

Il est essentiel que dans vos expériences vous incluiez les dates suivantes :

2/4/68 1/1/80, 8/16/64, 7/20/60, 11/2/72, 12/25/99, la date d'aujourd'hui, une date d'anniversaire (non nécessairement le vôtre), la date du début de votre rapport, etc.

1.14 Questions

1. Convertissez en binaire les valeurs hexadécimales suivantes :

a) 128	b) 4096	c) 256	d) 65536	e) 254
f) 9	g) 1024	h) 15	i) 344	j) 998
k) 255	l) 512	m) 1023	n) 2048	o) 4095
p) 8192	q) 16384	r) 32768	s) 6334	t) 12334
u) 23465	v) 5643	w) 464	x) 67	y) 888

2. Convertisez en décimal les valeurs binaires suivantes :

a) 1001 1001	b) 1001 1101	c) 1100 0011	d) 0000 1001	e) 1111 1111
f) 0000 1111	g) 0111 1111	h) 1010 0101	i) 0100 0101	j) 0101 1010
k) 1111 0000	l) 1011 1101	m) 1100 0010	n) 0111 1110	o) 1110 1111
p) 0001 1000	q) 1001 1111	r) 0100 0010	s) 1101 1100	t) 1111 0001
u) 0110 1001	v) 0101 1011	w) 1011 1001	x) 1110 0110	y) 1001 0111

3. Convertissez les valeurs binaires du problème 2 en hexadécimal.

4. Convertissez en binaire les valeurs hexadécimales suivantes :

a) 0ABCD	b) 1024	c) 0DEAD	d) 0ADD	e) 0BEEF
f) 8	g) 05AAF	h) 0FFFF	i) 0ACDB	j) 0CDBA
k) 0FEBA	l) 35	m) 0BA	n) 0ABA	o) 0BAD
p) 0DAB	q) 4321	r) 334	s) 45	t) 0E65
u) 0BEAD	v) 0ABE	w) 0DEAF	x) 0DAD	y) 9876

Effectuez les calculs hexadécimaux suivants (laissez le résultat en hex) :

5. 1234 + 9876
6. 0FFF - 0F34
7. 100 - 1
8. 0FFE -1
9. Quelle est l'importance d'un quartet ?
10. Il y a combien d'unités hexadécimales dans un :

    a) octet b) mot c) double-mot
11. Il y a combien de bits dans un :

    a) quartet b) octet c) mot d) double-mot
12. Quel est le numéro de bit du bit le plus significatif dans un :

    a) quartet b) octet c) mot d) double-mot
13. Quel caractère utilise-t-on comme suffixe dans un nobre hexadécimal ? Binaire ? Décimal ?
14. En présumant utiliser un format de 16 bits à deux compléments (signé), déterminez quelles valeurs de la question 4 sont positives et lesquelles sont négatives.
15. Élargissez toutes les valeurs de la question 2 à 16 bits (selon le signe). Donnez votre réponse en hexadécimal.
16. Effectuez une opération AND dans les paires de valeurs hexadécimales suivantes. Donnez votre réponse en hexadécimal (astuce : convertissez les valeurs hex en binaire, effectuez l'opération, puis reconvertissez-les en hex).

a) 0FF00, 0FF0	b) 0FOOF, 1234	c) 4321, 1234	d) 2341, 3241	e) 0FFFF, 0EDCB
f) 1111, 5789	f) 0FABA, 4322	h) 5523, 0F572	i) 2355, 7466	j) 4765, 6543
k) 0ABCD, 0EFDC	l) 0DDDD, 1234	m) 0CCCC, 0ABCD	n) 0BBBB, 1234	o) 0AAAA, 1234
p) 0EEEE, 1248	q) 8888, 1248	r) 8086, 124F	s) 8086, 0CFA7	t) 8765, 3456
u) 7089, 0FEDC	v) 2435, 0BCDE	w) 6355, 0EFDC	x) 0CBA, 6884	y) 0AC7, 365
17. Effectuez une opération OR dans les paires de nombres ci-dessus.
18. Effectuez une opération XOR dans les paires de nombres ci-dessus.
19. Effectuez une opération NOT dans toutes les valeurs de la question 4. Considérez-les toutes comme étant de 16 bits.
20. Effectuez l'opération de complémentation à deux dans toutes les valeurs de la question 4. Présumez encore que toutes ces valeurs soient de 16 bits.
21. Élargissez les valeurs hexadécimales suivantes de huit à seize bits (en tenant compte du signe, naturellement). Présentez votre réponse en hex.

    a) FF b) 82 c) 12 d) 56 e) 98

f) BF	g) 0F	h) 78	i) 7F	j) F7
k) 0E	l) AE	m) 45	n) 93	o) C0
p) 8F	q) DA	r) 1D	s) 0D	t) DE
u) 54	v) 45	w) F0	x) AD	y) DD

22. Faites une contraction de 16 à 8 bits des valeurs suivantes (en tenant compte du signe). Si vous ne pouvez pas le faire, expliquez pourquoi.

a) FF00	b) FF12	c) FFF0	d) 12	e) 80
f) FFFF	g) FF88	h) FF7F	i) 7F	j) 2
k) 8080	l) 80FF	m) FF80	n) FF	o) 8
p) F	q) 1	r) 834	s) 34	t) 23
u) 67	v) 89	w) 98	x) FF98	y) F98

23. Élargissez les valeurs signées dans les questions 22 à des valeurs de 32 bits.

24. En présumant que les valeurs de la question 22 sont de 16 bits, effectuez un décalage à gauche.

25. En présumant que les valeurs de la question 22 sont de 16 bits, effectuez un décalage à droite.

26. En présumant que les valeurs de la question 22 sont de 16 bits, effectuez une rotation à gauche.

27. En présumant que les valeurs de la question 22 sont de 16 bits, effectuez une rotation à droite.

28. Convertissez les dates suivantes en un format compacté comme décrit dans ce chapitre (voir le paragraphe 1.10). Présentez vos résultats comme des valeurs hexadécimales de 16 bits :

a) 1/1/92	b) 2/4/56	c) 6/19/60	d) 6/16/86	e) 1/1/99

29. Décrivez comment utiliser le décalage et les opérations logiques pour *extraire* le champ *jour* de l'enregistrement de donnée compactée de la question 28. Cela fait, exprimez ce jour par une valeur entière de 16 bits dans la plage 1…31[20].

30. Supposez avoir une valeur de la plage 0…9. Expliquez comment vous pourriez la convertir en un code de caractères ASCII en n'utilisant que des opérations logiques.

31. La fonction C++ suivante localise le premier bit activé dans un paramètre BitMap et le fait à partir de la première position jusqu'au bit fort. Si un tel bit n'existe pas, elle retourne -1. Expliquez en détail comment cette fonction travaille :

```
int FindFirstSet(unsigned BitMap, unsigned start)
{
       unsigned Mask = (1 << start);

       while(Mask)
       {
              if(BitMap & Mask) return start;
              ++start;
              Mask <<= 1;
       }
       return -1;
}
```

32. Le langage de programmation C++ ne spécifie pas combien de bits on peut trouver dans un entier non signé. Expliquez pourquoi le code ci-dessus fonctionnera indépendamment du nombre de bits trouvés dans le nombre.

[20]Il y avait une erreur dans l'original. La plage était indiquée comme 0…31, mais puisque le jour 0 n'a jamais été utilisé ni dans ce chapitre, ni dans la réalité j'ai préféré m'en tenir à une majeure exactitude. De plus, n'oubilez pas que, dans ce livre, les dates sont présentée en format américain (mm-jj-aa), n.d.t.

33. La fonction C++ suivante est le complément de la fonction ci-dessus. Elle localise le premier bit désactivé dans un paramètre BitMap. Expliquez, en détail, comment elle accomplit cette tâche :

```
int FindFirstClr(unsigned BitMap, unsigned start)
{
       return FindFirstSet(~BitMap, start);
}
```

34. Les deux fonctions suivantes activent ou désactivent (respectivement) un bit particulier et renvoient le nouveau résultat. Expliquez, en détail, comment ces fonctions fonctionnent.

```
unsigned SetBit(unsigned BitMap, unsigned position)
{
       return BitMap | (1 << position);
}

unsigned ClrBit(unsigned BitMap, unsigned position)
{
       return BitMap & ~(1 << position);
}
```

35. Dans le code de la question précédente, expliquez ce qui arriverait si les paramètres *start* et *position* contenaient une valeur plus grande ou égale au nombre de bits dans un entier non signé.

1.15 Projets de programmation

Les projets de programmation suivants prennent pour acquis que vous utilisez C, C++, Turbo Pascal, Borland Pascal, Delphi ou quelque autre langage qui supporte les opérations logiques binaires. Notez que le C++ utilise les opérateurs "&", "|" et "^" pour AND, OR et XOR respectivement. Les produits Borland Pascal permettent d'utiliser les opérateurs "and", "or", et "xor" sur les entiers pour effectuer des opérations logiques sur les bits. Les projets suivants requièrent l'usage de ces opérateurs. Il y a d'autres solutions à ces problèmes où ces opérateurs ne sont pas nécessaires, mais **n'employez pas de telles solutions**. Le but de ces exercices et de vous présenter les opérations logiques disponibles dans les langages de haut niveau (**voyez avec votre professeur, quel langage vous devez utiliser**).

Les descriptions suivantes décrivent les fonctions que vous écrirez. Cependant, vous aurez besoin d'un programme principal pour appeler et tester chacune d'elles.

1) Ecrivez deux fonctions *toupper* et *tolower* qui prennent un caractère comme argument et convertissent ce caractère en majuscule (s'il était en minuscule) ou en minuscule (s'il était en majuscule), respectivement. Utilisez les opérations logiques pour effectuer cette conversion. Les utilisateurs de Pascal pourraient avoir besoin des fonctions chr() et ord() pour résoudre ce problème.

2) Ecrivez une fonction "CharToInt" qui reçoit une chaîne de caractères et retourne la valeur entière correspondante. *N'utilisez pas une routine prédéfinie de bibliothèque comme atoi (c) ou strtoint (Pascal)*. Vous devez traiter chaque caractère passé, le convertir en entier à l'aide des opérations logiques et accumuler les résultats jusqu'à ce que vous ne soyez arrivé à la fin de la chaîne. Un algorithme facile pour cette tâche est de multiplier le résultat accumulé par 10 et lui ajouter la prochaine unité convertie. Répétez ceci jusqu'à la fin de la chaîne. Les utilisateurs de Pascal auront probablement besoin d'utiliser la fonction ord().

3) Écrivez une fonction "ToDate" qui accepte trois paramètres, un mois, un jour et une année. Cette fonction doit retourner une valeur compactée de 16 bits en utilisant le format donné dans ce chapitre (au paragraphe 1.10). Écrivez trois fonctions correspondantes, ExtractMonth, ExtractDay et ExtractYear qui attendent une valeur de 16 bits et retournent le mois, le jour ou l'année correspondante. La fonction ToDate devrait convertir automatiquement les dates de la plage 1900-19999 en des dates de la plage 00-99.

4) Écrire une fonction "CntBits" qui compte le nombre de bits ayant la valeur 1 dans un entier de 16 bits. *N'utilisez pas une fonction de bibliothèque prédéfinie*.

5) Écrire une fonction "TestBit" qui prend deux arguments. Le premier est une valeur de 16 bits à tester ; le second est une valeur dans la plage 0...15 décrivant quel bit tester. La fonction doit retourner vrai si le bit

correspondant contient 1 et faux dans le cas contraire. La fonction retourne faux également si le second paramètre contient une valeur en dehors de la plage prévue.

6) Pascal et C/C++ fournissent les opérations de décalage (SHL/SHR en Pascal, "<<" et ">>" en C/C++). Cependant, ils n'offrent pas les opérations de rotation. Ecrivez une paire de fonctions, ROL et ROR pour effectuer ces tâches. Astuce : utilisez la fonction de l'exercice 5 pour tester le bit fort. Puis utilisez l'opération de décalage correspondante et un OR pour réaliser la rotation.