Langage Assembleur PC

Paul A. Carter

4 décembre 2021

Table des matières

Préface

Objectif

L'objectif de ce livre est de permettre au lecteur de mieux comprendre comment les ordinateurs fonctionnent réellement à un niveau plus bas que les langages de programmation comme Pascal. En ayant une compréhension plus profonde de la façon dont fonctionnent les ordinateurs, le lecteur peu devenir plus productif dans le développement de logiciel dans des langages de plus haut niveau comme le C et le C++. Apprendre à programmer en assembleur est un excellent moyen d'atteindre ce but. Les autres livres d'assembleur pour PC apprennent toujours à programmer le processeur 8086 qu'utilisait le PC originel de 1981 ! Le processeur 8086 ne supportait que le mode *réel*. Dans ce mode, tout programme peu adresser n'importe quel endroit de la mémoire ou n'importe quel périphérique de l'ordinateur. Ce mode n'est pas utilisable pour un système d'exploitation sécurisé et multitâche. Ce livre parle plutôt de la façon de programmer les processeurs 80386 et plus récents en mode *protégé* (le mode dans lequel fonctionnent Windows et Linux). Ce mode supporte les fonctionnalités que les systèmes d'exploitation modernes offrent, comme la mémoire virtuelle et la protection mémoire. Il y a plusieurs raisons d'utiliser le mode protégé :

1. Il est plus facile de programmer en mode protégé qu'en mode réel 8086 que les autres livres utilisent.
2. Tous les systèmes d'exploitation PC modernes fonctionnent en mode protégé.
3. Il y a des logiciels libres disponibles qui fonctionnent dans ce mode.

Le manque de livres sur la programmation en assembleur PC en mode protégé est la principale raison qui a conduit l'auteur à écrire ce livre.

Comme nous y avons fait allusion ci-dessus, ce dexte utilise des logiciels Libres/Open Source : l'assembleur NASM et le compilateur C/C++ DJGPP. Les deux sont disponibles en téléchargement sur Internet. Ce texte parle également de l'utilisation de code assembleur NASM sous Linux et avec les compilateurs C/C++ de Borland et Microsoft sous Windows. Les

exemples pour toutes ces plateformes sont disponibles sur mon site Web : `http://www.drpaulcarter.com/pcasm`. Vous *devez* télécharger le code exemple si vous voulez assembler et exécuter la plupart des exemples de ce tutorial.

Soyez conscient que ce texte ne tente pas de couvrir tous les aspects de la programmation assembleur. L'auteur a essayé de couvrir les sujets les plus importants avec lesquels *tous* les programmeurs devraient être familiers.

Remerciements

L'auteur voutrait remercier les nombreux programmeurs qui ont contribué au mouvement Libre/Open Source. Tous les programmes et même ce livre lui-même ont été créés avec des logiciels gratuits. L'auteur voudrait remercier en particulier John S. Fine, Simon Tatham, Julian Hall et les autres développeurs de l'assembleur NASM sur lequel tous les exemples de ce livre sont basés ; DJ Delorie pour le développement du compilateur C/C++ DJGPP utilisé ; les nombreuses personnes qui ont contribué au compilateur GNU gcc sur lequel DJGPP est basé ; Donald Knuth et les autres pour avoir développé les langages composition TEX et LATEX 2_ε qui ont été utilisés pour produire le livre ; Richard Stallman (fondateur de la Free Software Foundation), Linus Torvalds (créateur du noyau Linux) et les autres qui ont créé les logiciels sous-jacents utilisés pour ce travail.

Merci aux personnes suivantes pour leurs corrections :

— John S. Fine
— Marcelo Henrique Pinto de Almeida
— Sam Hopkins
— Nick D'Imperio
— Jeremiah Lawrence
— Ed Beroset
— Jerry Gembarowski
— Ziqiang Peng
— Eno Compton
— Josh I Cates
— Mik Mifflin
— Luke Wallis
— Gaku Ueda
— Brian Heward
— Chad Gorshing
— F. Gotti
— Bob Wilkinson
— Markus Koegel
— Louis Taber

Ressources sur Internet

Page de l'auteur	`http://www.drpaulcarter.com/`
Page NASM sur SourceForge	`http://sourceforge.net/projects/nasm/`
DJGPP	`http://www.delorie.com/djgpp`
Assembleur Linux	`http://www.linuxassembly.org/`
The Art of Assembly	`http://webster.cs.ucr.edu/`
USENET	`comp.lang.asm.x86`
Documentation Intel	`http://developer.intel.com/design/Pentium4/documentation.htm`

Réactions

L'auteur accepte toute réaction sur ce travailThe author welcomes any feedback on this work.

E-mail: `pacman128@gmail.com`
WWW: `http://www.drpaulcarter.com/pcasm`

Chapitre 1

Introduction

1.1 Systèmes Numériques

La mémoire d'un ordinateur est constituée de nombres. Cette mémoire ne stocke pas ces nombres en décimal (base 10). Comme cela simplifie grandement le matériel, les ordinateurs stockent toutes les informations au format binaire (base 2). Tout d'abord, revoyons ensemble le système décimal.

1.1.1 Décimal

Les nombres en base 10 sont constitués de 10 chiffres possibles (0-9). Chaque chiffre d'un nombre est associé à une puissance de 10 selon sa position dans le nombre. Par exemple :

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 Binaire

Les nombres en base 2 sont composés de deux chiffres possibles (0 et 1). Chaque chiffre est associé à une puissance de 2 selon sa position dans le nombre (un chiffre binaire isolé est appelé bit). Par exemple :

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

Cet exemple montre comment passer du binaire au décimal. Le tableau 1.1 montre la représentation binaire des premiers nombres.

La figure 1.1 montre comment des chiffres binaires individuels (*i.e.*, des bits) sont additionnés. Voici un exemple :

Decimal	Binary		Decimal	Binary
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

TABLE 1.1 – Décimaux 0 à 15 en Binaire

No previous carry				Previous carry			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

FIGURE 1.1 – Addition binaire (c signifie *carry*, retenue)

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

Si l'on considère la division décimale suivante :

$$1234 \div 10 = 123\ r\ 4$$

on peut voir que cette division sépare le chiffre le plus à droite du nombre et décale les autres chiffres d'une position vers la droite. Diviser par deux effectue une opération similaire, mais pour les chiffres binaires du nombre. Considérons la division binaire suivante [1] :

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

Cette propriété peut être utilisée pour convertir un nombre décimal en son équivalent binaire comme le montre la Figure 1.2. Cette méthode trouve le chiffre binaire le plus à droite en premier, ce chiffre est appelée le *bit le moins significatif* (lsb, least significant bit). Le chiffre le plus à gauche est appelé le *bit le plus significatif* (msb, most significant bit). L'unité de base de la mémoire consiste en un jeu de 8 bits appelé *octet* (byte).

1. L'indice 2 est utilisé pour indiquer que le nombre est représenté en binaire, pas en décimal

Decimal	Binary
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

Donc $25_{10} = 11001_2$

FIGURE 1.2 – Conversion décimale

$$\begin{aligned} 589 \div 16 &= 36\ r\ 13 \\ 36 \div 16 &= 2\ r\ 4 \\ 2 \div 16 &= 0\ r\ 2 \end{aligned}$$

Donc $589 = 24\mathrm{D}_{16}$

FIGURE 1.3 –

1.1.3 Hexadecimal

Les nombres hexadécimaux utilisent la base 16. L'hexadécimal (ou *hexa* en abrégé) peut être utilisé comme notation pour les nombres binaires. L'hexa a 16 chiffres possibles. Cela pose un problème car il n'y a pas de symbole à utiliser pour les chiffres supplémentaires après 9. Par convention, on utilise des lettres pour les représenter. Les 16 chiffres de l'hexa sont 0-9 puis A, B, C, D, E et F. Le chiffre A équivaut 10 en décimal, B à 11, etc. Chaque chiffre d'un nombre en hexa est associé à une puissance de 16. Par exemple :

$$\begin{aligned} 2\mathrm{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

Pour convertir depuis le décimal vers l'hexa, utilisez la même idée que celle utilisée pour la conversion binaire en divisant par 16. Voyez la Figure 1.3 pour un exemple.

La raison pour laquelle l'hexa est utile est que la conversion entre l'hexa et le binaire est très simple. Les nombres binaires deviennent grands et incompréhensibles rapidement. L'hexa fournit une façon beaucoup plus compacte pour représenter le binaire.

Pour convertir un nombre hexa en binaire, convertissez simplement chaque chiffre hexa en un nombre binaire de 4 bits. Par exemple, $24D_{16}$ est converti en $0010\ 0100\ 1101_2$. Notez que les 0 de tête des 4 bits sont importants ! Si le 0 de tête pour le chiffre du milieu de $24D_{16}$ n'est pas utilisé, le résultat est faux. Le conversion du binaire vers l'hexa est aussi simple. On effectue la même chose, dans l'autre sens. Convertissez chaque segment de 4 bits du binaire vers l'hexa. Commencez à droite, pas à gauche, du nombre binaire. Cela permet de s'assurer que le procédé utilise les segments de 4 bits corrects [2]. Exemple :

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

Un nombre de 4 bits est appelé *quadruplet*(nibble). Donc, chaque chiffre hexa correspond à un quadruplet. Deux quadruplets forment un octet et donc un octet peut être représenté par un nombre hexa à deux chiffres. La valeur d'un bit va de 0 à 11111111 en binaire, 0 à FF en hexa et 0 à 255 en décimal.

1.2 Organisation de l'Ordinateur

1.2.1 Mémoire

La mémoire est mesurée en kilo octets ($2^{10} = 1024$ octets), mega octets ($2^{20} = 1048576$ octets) et giga octets ($2^{30} = 1073741824$ octets).

L'unité mémoire de base est l'octet. Un ordinateur avec 32 mega octets de mémoire peut stocker jusqu'à environ 32 millions d'octets d'informations. Chaque octet en mémoire est étiqueté par un nombre unique appelé son adresse comme le montre la Figure 1.4.

Adresse	0	1	2	3	4	5	6	7
Mémoire	2A	45	B8	20	8F	CD	12	2E

FIGURE 1.4 – Adresses Mémoire

Souvent, la mémoire est utilisée par bouts plus grand que des octets isolées. Sur l'architecture PC, on a donné des noms à ces portions de mémoire plus grandes, comme le montre le Tableau 1.2.

2. S'il n'est pas clair que le point de départ est important, essayez de convertir l'exemple en partant de la gauche

mot	2 octets
double mot	4 octets
quadruple mot	8 octets
paragraphe	16 octets

TABLE 1.2 – Unités Mémoire

Toutes les données en mémoire sont numériques. Les caractères sont stockés en utilisant un *code caractère* qui fait correspondre des nombres aux caractères. Un des codes caractère les plus connus est appelé *ASCII* (American Standard Code for Information Interchange, Code Américain Standard pour l'Echange d'Informations). Un nouveau code, plus complet, qui supplante l'ASCII est l'Unicode. Une des différences clés entre les deux codes est que l'ASCII utilise un octet pour encoder un caractère alors que l'Unicode en utilise deux (ou un *mot*). Par exemple, l'ASCII fait correspondre l'octet 41_{16} (65_{10}) au caractère majuscule *A* ; l'Unicode y fait correspondre le mot 0041_{16}. Comme l'ASCII n'utilise qu'un octet, il est limité à 256 caractères diffrérents au maximum [3]. L'Unicode étend les valeurs ASCII à des mots et permet de représenter beaucoup plus de caractères. C'est important afin de représenter les caractères de tous les langages du monde.

1.2.2 Le CPU (processeur)

Le processeur (CPU, Central Processing Unit) est le dispositif physique qui exécute les instructions. Les instructions que les processeurs peuvent exécuter sont généralement très simples. Elles peuvent nécessiter que les données sur lesquelles elles agissent soient présentes dans des emplacements de stockage spécifiques dans le processeur lui-même appelés *registres*. Le processeur peut accéder aux données dans les registres plus rapidement qu'aux données en mémoire. Cependant, le nombre de registres d'un processeur est limité, donc le programmeur doit faire attention à n'y conserver que les données actuellement utilisées.

Les instructions que peut exécuter un type de processeur constituent le *langage machine*. Les programmes machine ont une structure beaucoup plus basique que les langages de plus haut niveau. Les instructions du langage machine sont encodées en nombres bruts, pas en format texte lisible. Un processeur doit être capable de décoder les instructions très rapidement pour fonctionner efficacement. Le langage machine est conçu avec ce but en tête, pas pour être facilement déchiffrable par les humains. Les programmes écrits dans d'autres langages doivent être convertis dans le langage machine

3. En fait, l'ASCII n'utilise que les 7 bits les plus faible et donc n'a que 128 valeurs à utiliser.

natif du processeur pour s'exécuter sur l'ordinateur. Un *compilateur* est un programme qui traduit les programmes écrits dans un langage de programmation en langage machine d'une architecture d'ordinateur particulière. En général, chaque type de processeur a son propre langage machine unique. C'est une des raisons pour lesquelles un programme écrit pour Mac ne peut pas être exécuté sur un PC.

GHz *signifie gigahertz ou un milliards de cycles par seconde. Un processeur à 1,5GHz a 1,5 milliards d'impulsions horloge par seconde.*

Les ordinateurs utilisent une *horloge* pour synchroniser l'exécution des instructions. L'horloge tourne à une fréquence fixée (appelée *vitesse d'horloge*). Lorsque vous achetez un ordinateur à 1,5GHz, 1,5GHz est la fréquence de cette horloge. L'horloge ne décompte pas les minutes et les secondes. Elle bat simplement à un rythme constant. Les composants électroniques du processeur utilisent les pulsations pour effectuer leurs opérations correctement, comme le battement d'un métronome aide à jouer de la musique à un rythme correct. Le nombre de battements (ou, comme on les appelle couramment *cycles*) que recquiert une instruction dépend du modèle et de la génération du processeur. Le nombre de cycles dépend de l'instruction.

1.2.3 La famille des processeurs 80x86

Les PC contiennent un processeur de la famille des Intel 80x86 (ou un clone). Les processeurs de cette famille ont tous des fonctionnalités en commun, y compris un langage machine de base. Cependant, les membre les plus récents améliorent grandement les fonctionnalités.

8088,8086: Ces processeurs, du point de vue de la programmation sont identiques. Ils étaient les processeurs utilisés dans les tous premiers PC. Il offrent plusieurs registres 16 bits : AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Ils ne supportent que jusqu'à 1Mo de mémoire et n'opèrent qu'en *mode réel*. Dans ce mode, un programme peut accéder à n'importe quelle adresse mémoire, même la mémoire des autres programmes ! Cela rend le débogage et la sécurité très difficiles ! De plus, la mémoire du programme doit être divisée en *segments*. Chaque segment ne peut pas dépasser les 64Ko.

80286: Ce processeur était utilisé dans les PC de type AT. Il apporte quelques nouvelles instructions au langage machine de base des 8088/86. Cependant, sa principale nouvelle fonctionnalité est le *mode protégé 16 bits*. Dans ce mode, il peut accéder jusqu'à 16Mo de mémoire et empêcher les programmes d'accéder à la mémoire des uns et des autres. Cependant, les programmes sont toujours divisés en segments qui ne peuvent pas dépasser les 64Ko.

80386: Ce processeur a grandement amélioré le 80286. Tout d'abord, il étend la plupart des registres à 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) et ajoute deux nouveaux registres 16 bits : FS

FIGURE 1.5 – Le registre AX

et GS. Il ajoute également un nouveau *mode protégé 32 bits*. Dans ce mode, il peut accéder jusqu'à 4Go de mémoire. Les programmes sont encore divisés en segments mais maintenant, chaque segment peut également faire jusqu'à 4Go !

80486/Pentium/Pentium Pro: Ces membres de la famille 80x86 apportent très peu de nouvelles fonctionnalités. Il accélèrent principalement l'exécution des instructions.

Pentium MMX: Ce processeur ajoute les instructions MMX (Multi-Media eXentions) au Pentium. Ces instructions peuvent accélérer des opérations graphiques courantes.

Pentium II: C'est un processeur Pentium Pro avec les instructions MMX (Le Pentium III est grossièrement un Pentium II plus rapide).

1.2.4 Regitres 16 bits du 8086

Le processeur 8086 original fournissait quatre registres généraux de 16 bits : AX, BX, CX et DX. Chacun de ces registres peut être décomposé en deux registres de 8 bits. Par exemple, le registre AX pouvait être décomposé en AH et AL comme le montre la Figure 1.5. Le registre AH contient les 8 bits de poids fort de AX et AL contient les 8 bits de poids faivble. Souvent, AH et AL sont utilisés comme des registres d'un octet indépendants ; cependant, il est important de réaliser qu'ils ne sont pas indépendants de AX. Changer la valeur de AX changera les valeurs de AL et AL et *vice versa*. Les registres généraux sont utilisés dans beaucoup de déplacements de données et instructions arithmétiques.

Il y a deux registres d'index de 16 bits : SI et DI. Ils sont souvent utilisés comme des pointeurs, mais peuvent être utilisés pour la plupart des mêmes choses que les registres généraux. Cependant, ils ne peuvent pas être décomposés en registres de 8 bits.

Les registres 16 bits BP et SP sont utilisés pour pointer sur des données dans la pile du langage machine et sont appelés le pointeur de base et le ponteur de pile, respectivement. Nous en reparlerons plus tard.

Les registres 16 bits CS, DS, SS et ES sont des *registres de segment*. Ils indiquent quelle zone de la mémoire est utilisée pour les différentes parties d'un programme. CS signifie Code Segment, DS Data Segment, SS Stack Segment (segment de pile) et ES Extra Segment. ES est utilisé en tant que

registre de segment temporaire. Des détails sur ces registres se trouvent dans les Sections 1.2.6 et 1.2.7.

Le registre de pointeur d'instruction (IP) est utilisé avec le registre CS pour mémoriser l'adresse de la prochaine instruction à exécuter par le processeur. Normalement, lorsqu'une instruction est exécutée, IP est incrémenté pour pointer vers la prochaine instruction en mémoire.

Le registre FLAGS stocke des informations importantes sur les résultats d'une instruction précédente. Ces résultats sont stockés comme des bits individuels dans le registre. Par exemple, le bit Z est positionné à 1 si le résultat de l'instruction précédente était 0 ou à 0 sinon. Toutes les instructions ne modifient pas les bits dans FLAGS, consultez le tableau dans l'appendice pour voir comment chaque instruction affecte le registre FLAGS.

1.2.5 Registres 32 bits du 80386

Les processeurs 80386 et plus récents ont des registres étendus. Par exemple le registre AX 16 bits est étendu à 32 bits. Pour la compatibilité ascendante, AX fait toujours référence au registre 16 bits et on utilise EAX pour faire référence au registre 32 bits. AX représente les 16 bits de poids faible de EAX tout comme AL représente les 8 bits de poids faible de AX (et de EAX). Il n'y a pas de moyen d'accéder aux 16 bits de poids fort de EAX directement. Les autres registres étendus sont EBX, ECX, EDX, ESI et EDI.

La plupart des autres registres sont également étendus. BP devient EBP ; SP devient ESP ; FLAGS devient EFLAGS et IP devient EIP. Cependant, contrairement aux registres généraux et d'index, en mode protégé 32 bits (dont nous parlons plus loin) seules les versions étendues de ces registres sont utilisées.

Les registres de segment sont toujours sur 16 bits dans le 80386. Il y a également deux registres de segment supplémentaires : FS et GS. Leurs noms n'ont pas de signification particulière. Ce sont des segments temporaires supplémentaires (comme ES).

Une des définitions du terme *mot* se réfère à la taille des registres de données du processeur. Pour la famille du 80x86, le terme est désormais un peu confus. Dans le Tableau 1.2, on voit que le terme *mot* est défini comme faisant 2 octets (ou 16 bits). Cette signification lui a été attribuée lorsque le premier 8086 est apparu. Lorsque le 80386 a été développé, il a été décidé de laisser la définition de *mot* inchangée, même si la taille des registres avait changé.

1.2.6 Mode Réel

En mode réel, la mémoire est limitée à seulement un méga octet (2^{20} octet). Les adresses valides vont de 00000 à FFFFF (en hexa). Ces adresses nécessitent un nombre sur 20 bits. Cependant, un nombre de 20 bits ne tiendrait dans aucun des registres 16 bits du 8086. Intel a résolu le problème, en utilisant deux valeurs de 16 bits pour déterminer une adresse. La première valeur de 16 bits est appelée le *sélecteur*. Les valeurs du sélecteur doivent être stockées dans des registres de segment. La seconde valeur de 16 bits est appelée le *déplacement* (offset). L'adresse physique identifiée par un couple *sélecteur*:*déplacement* 32 bits est calculée par la formule

Alors d'où vient l'infâme limite des 640Ko du DOS ? Le BIOS requiert une partie des 1Mo pour son propre code et pour les périphériques matériels comme l'écran.

$$16 * \text{sélecteur} + \text{déplacement}$$

Multiplier par 16 en hexa est facile, il suffit d'ajouter un 0 à la droite du nombre. Par exemple, l'adresse physique référencée par 047C:0048 est obtenue de la façon suivante :

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

De fait, la valeur du sélecter est un numéro de paragraphe (voir Tableau 1.2).

Les adresses réelles segmentées ont des inconvénients :

— Une seule valeur de sélecteur peut seulement référencer 64Ko de mémoire (la limite supérieure d'un déplacement de 16 bits). Que se passe-t-il si un programme a plus de 64Ko de code ? Une seule valeur de CS ne peut pas être utilisée pour toute l'exécution du programme. Le programme doit être divisé en sections (appelées *segments*) de moins de 64Ko. Lorsque l'exécution passe d'un segment à l'autre, la valeur de CS doit être changée. Des problèmes similaires surviennent avec de grandes quantités de données et le registre DS. Cela peut être très gênant !

— Chaque octet en mémoire n'a pas une adresse segmentée unique. L'adresse physique 04808 peut être référencée par 047C:0048, 047D:0038, 047E:0028 ou 047B:0058. Cela complique la comparaison d'adresses segmentées.

1.2.7 Mode Protégé 16 bits

Dans le mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel. En mode réel, la valeur d'un sélecteur est un numéro de paragraphe en mémoire. En mode protégé, un sélecteur est un *indice* dans un *tableau de descripteurs*. Dans les deux modes, les programmes sont divisés en segments. En mode réel,

ces segments sont à des positions fixes en mémoire et le sélecteur indique le numéro de paragraphe auquel commence le segment. En mode protégé, les segments ne sont pas à des positions fixes en mémoire physique. De fait, il n'ont même pas besoin d'être en mémoire du tout !

Le mode protégé utilise une technique appelée *mémoire virtuelle*. L'idée de base d'un système de mémoire virtuelle est de ne garder en mémoire que les programmes et les données actuellement utilisés. Le reste des données et du code sont stockés temporairement sur le disque jusqu'à ce qu'on aie à nouveau besoin d'eux. Dans le mode protégé 16 bits, les segments sont déplacés entre la mémoire et le disque selon les besoins. Lorsqu'un segment est rechargé en mémoire depuis le disque, il est très probable qu'il sera à un endroit en mémoire différent de celui où il était avant d'être placé sur le disque. Tout ceci est effectué de façon transparente par le système d'exploitation. Le programme n'a pas a être écrit différemment pour que la mémoire virtuelle fonctionne.

En mode protégé, chaque segment est assigné à une entrée dans un tableau de descripteurs. Cette entrée contient toutes les informations dont le système a besoin à propos du segment. Ces informations indiquent : s'il est actuellement en mémoire ; s'il est en mémoire, où il se trouve ; les droits d'accès (*p.e.*, lecture seule). L'indice de l'entrée du segment est la valeur du sélecteur stockée dans les registres de segment.

Un journaliste PC bien connu a baptisé le processeur 286 "cerveau mort" (brain dead)

Un gros inconvénient du mode protégé 16 bits est que les déplacements sont toujours des quantités sur 16 bits. En conséquence, Les tailles de segment sont toujours limitées au plus à 64Ko. Cela rend l'utilisation de grands tableaux problématique.

1.2.8 Mode Protégé 32 bits

Le 80386 a introduit le mode protégé 32 bits. Il y a deux différences majeures entre les modes protégés 32 bits du 386 et 16 bits du 286 :

1. Les déplacements sont étendus à 32 bits. Cela permet à un déplacement d'aller jusqu'à 4 milliards. Ainsi, les segments peuvent avoir des tailles jusqu'à 4Go.
2. Les segments peuvent être divisés en unités plus petites de 4Ko appelées *pages*. Le système de mémoire virtuelle fonctionne maintenant avec des pages plutôt qu'avec des segments. Cela implique que seules certaines parties d'un segment peuvent être présentes en mémoire à un instant donné. En mode 16 bits du 286, soit le segment en entier est en mémoire, soit rien n'y est. Ce qui n'aurait pas été pratique avec les segments plus grands que permet le mode 32 bits.

Dans Windows 3.x, le *mode standard* fait référence au mode protégé 16 bits du 286 et le *mode amélioré* (enhanced) fait référence au mode 32 bits.

Windows 9X, Windows NT/2000/XP, OS/2 et Linux fonctionnent tous en mode protégé 32 bits paginé.

1.2.9 Interruptions

Quelques fois, le flot ordinaire d'un programme doit être interrompu pour traiter des évènements qui requièrent une réponse rapide. Le matériel d'un ordinateur offre un mécanisme appelé *interruptions* pour gérer ces évènements. Par exemple, lorsqu'une souris est déplacée, la souris interrompt le programme en cours pour gérer le déplacement de la souris (pour déplacer le curseur *etc.*). Les interruptions provoquent le passage du contrôle à un *gestionnaire d'interruptions*. Les gestionnaires d'interruptions sont des routines qui traitent une interruption. Chaque type d'interruption est assignée à un nombre entier. Au début de la mémoire physique, réside un tableau de *vecteurs d'interuptions* qui contient les adresses segmentées des gestionnaires d'interruption. Le numéro d'une interruption est essentiellement un indice dans ce tableau.

Les interruptions externes proviennent de l'extérieur du processeur (la souris est un exemple de ce type). Beaucoup de périphériques d'E/S soulèvent des interruptions (*p.e.*, le clavier, le timer, les lecteurs de disque, le CD-ROM et les cartes son). Les interruptions internes sont soulevées depuis le processeur, à cause d'une erreur ou d'une instruction d'interruption. Les interruptions erreur sont également appelées *traps*. Les interruptions générées par l'instruction d'interruption sont également appelées *interruptions logicielles*. Le DOS utilise ce type d'interruption pour implémenter son API(Application Programming Interface). Les systèmes d'exploitation plus récents (comme Windows et Unix) utilisent une interface basée sur C [4].

Beaucoup de gestionnaires d'interruptions redonnent le contrôle au programme interrompu lorsqu'ils se terminent. Ils restaurent tous les registres aux valeurs qu'ils avaient avant l'interruption. Ainsi, le programme interrompu s'exécute comme si rien n'était arrivé (excepté qu'il perd quelques cycles processeur). Les traps ne reviennent généralement jamais. Souvent, elles arrêtent le programme.

1.3 Langage Assembleur

1.3.1 Langage Machine

Chaque type de processeur comprend son propre langage machine. Les instructions dans le langage machine sont des nombres stockés sous forme

4. Cependant, ils peuvent également utiliser une interface de plus bas niveau au niveau du noyau

d'octets en mémoire. Chaque instruction a son propre code numérique unique appelé *code d'opération* ou *opcode* (operation code) en raccourci. Les instructions des processeurs 80x86 varient en taille. L'opcode est toujours au début de l'instruction. Beaucoup d'instructions comprennent également les données (*p.e.* des constantes ou des adresses) utilisées par l'instruction.

Le langage machine est très difficile à programmer directement. Déchiffrer la signification d'instructions codées numériquement est fatigant pour des humains. Par exemple, l'instruction qui dit d'ajouter les registres EAX et EBX et de stocker le résultat dans EAX est encodée par les codes hexadécimaux suivants :

```
03 C3
```

C'est très peu clair. Heureusement, un programme appelé un *assembleur* peut faire ce travail laborieux à la place du programmeur.

1.3.2 Langage d'Assembleur

Un programme en langage d'assembleur est stocké sous forme de texte (comme un programme dans un langage de plus haut niveau). Chaque instruction assembleur représente exactement une instruction machine. Par exemple, l'instruction d'addition décrite ci-dessus serait représentée en langage assembleur comme suit :

```
add eax, ebx
```

Ici, la signification de l'instruction est *beaucoup plus* claire qu'en code machine. Le mot `add` est un *mnémonique* pour l'instruction d'addition. La forme générale d'une instruction assembleur est :

```
mnémonique opérande(s)
```

Un *assembleur* est un programme qui lit un fichier texte avec des instructions assembleur et convertit l'assembleur en code machine. Les *compilateurs* sont des programmes qui font des conversions similaires pour les langages de programmation de haut niveau. Un assembleur est beaucoup plus simple qu'un compilateur. Chaque instruction du langage d'assembleur représente directement une instruction machine. Les instructions d'un langage de plus haut niveau sont *beaucoup* plus complexes et peuvent requérir beaucoup d'instructions machine.

Cela a pris plusieurs années aux scientifiques de l'informatique pour concevoir le simple fait d'écrire un compilateur !

Une autre différence importante entre l'assembleur et les langages de haut niveau est que comme chaque type de processeur a son propre langage machine, il a également son propre langage d'assemblage. Porter des programmes assembleur entre différentes architectures d'ordinateur est *beaucoup* plus difficile qu'avec un langage de haut niveau.

Les exemples de ce livre utilisent le Netwide Assembler ou NASM en raccourci. Il est disponible gratuitement sur Internet (voyez la préface pour

l'URL). Des assembleurs plus courants sont l'Assembleur de Microsoft (MASM) ou l'Assembleur de Borland (TASM). Il y a quelques différences de syntaxe entre MASM/TASM et NASM.

1.3.3 Opérandes d'Instruction

Les instructions en code machine ont un nombre et un type variables d'opérandes ; cependant, en général, chaque instruction a un nombre fixé d'opérandes (0 à 3). Les opérandes peuvent avoir les types suivants :

registre : Ces opérandes font directement référence au contenu des registres du processeur.

mémoire : Ils font référence aux données en mémoire. L'adresse de la donnée peut être une constante codée en dur dans l'instruction ou calculée en utilisant les valeurs des registres. Les adresses sont toujours des déplacements relatifs au début d'un segment.

immédiat : Ce sont des valeurs fixes qui sont listées dans l'instruction elle-même. Elles sont stockées dans l'instruction (dans le segment de code), pas dans le segment de données.

implicite : Ces opérandes ne sont pas entrés explicitement. Par exemple, l'instruction d'incrémentation ajoute un à un registre ou à la mémoire. Le un est implicite.

1.3.4 Instructions de base

L'instruction la plus basique est l'instruction `MOV`. Elle déplace les données d'un endroit à un autre (comme l'opérateur d'assignement dans un langage de haut niveau). Elle prend deux opérandes :

```
mov dest, src
```

La donnée spécifiée par *src* est copiée vers *dest*. Une restriction est que les deux opérandes ne peuvent pas être tous deux des opérandes mémoire. Cela nous montre un autre caprice de l'assembleur. Il y a souvent des règles quelque peu arbitraires sur la façon dont les différentes instructions sont utilisées. Les opérandes doivent également avoir la même taille. La valeur de AX ne peut pas être stockée dans BL.

Voici un exemple (les points-virgules marquent un commentaire) :

```
mov    eax, 3  ; stocke 3 dans le registre EAX (3 est un operande immediat)
mov    bx, ax  ; stocke la valeur de AX dans le registre BX
```

L'instruction `ADD` est utilisée pour additionner des entiers.

```
add    eax, 4   ; eax = eax + 4
add    al, ah   ; al = al + ah
```

L'instruction `SUB` soustrait des entiers.

```
sub    bx, 10   ; bx = bx - 10
sub    ebx, edi ; ebx = ebx - edi
```

Les instructions `INC` et `DEC` incrémentent ou décrémentent les valeurs de 1. Comme le un est un opérande implicite, le code machine pour `INC` et `DEC` est plus petit que celui des instructions `ADD` et `SUB` équivalentes.

```
inc    ecx      ; ecx++
dec    dl       ; dl--
```

1.3.5 Directives

Une *directive* est destinée à l'assembleur, pas au processeur. Elles sont généralement utilisées pour indiquer à l'assembleur de faire quelque chose ou pour l'informer de quelque chose. Elles ne sont pas traduites en code machine. Les utilisations courantes des directives sont :

- la définition de constantes
- la définition de mémoire pour stocker des données
- grouper la mémoire en segment
- inclure des codes sources de façon conditionnelle
- inclure d'autres fichiers

Le code NASM est analysé par un préprocesseur, exactement comme en C. Il y a beaucoup de commandes identiques à celles du préprocesseur C. Cependant, les directives du préprocesseur NASM commencent par un % au lieu d'un # comme en C.

La directive equ

La directive `equ` peut être utilisée pour définir un *symbole*. Les symboles sont des constantes nommées qui peuvent être utilisées dans le programme assembleur. Le format est le suivant :

symbole `equ` *valeur*

Les valeurs des symboles ne peuvent *pas* être redéfinies plus tard.

La directive %define

Cette directive est semblable à la directive `#define` du C. Elle est le plus souvent utilisée pour définir des macros, exactement comme en C.

```
%define SIZE 100
      mov    eax, SIZE
```

Unité	Lettre
octet	B
mot	W
double mot	D
quadruple mot	Q
dix octets	T

TABLE 1.3 – Lettres des Directives `RESX` et `DX`

Le code ci-dessus définit une macro appelée `SIZE` et montre son utilisation dans une instruction `MOV`. Les macros sont plus flexibles que les symboles de deux façons. Elles peuvent être redéfinies et peuvent être plus que de simples nombres constants.

Directives de données

Les directives de données sont utilisées dans les segments de données pour réserver de la place en mémoire. Il y a deux façons de réserver de la mémoire. La première ne fait qu'allouer la place pour les données ; la seconde alloue la place et donne une valeur initiale. La première méthode utilise une des directives `RES`*X*. Le *X* est remplacé par une lettre qui détermine la taille de l'objet (ou des objets) qui sera stocké. Le Tableau 1.3 montre les valeurs possibles.

La seconde méthode (qui définit une valeur initiale) utilise une des directives `D`*X*. Les lettres *X* sont les mêmes que celles de la directive `RES`*X*.

Il est très courant de marquer les emplacements mémoire avec des *labels* (étiquettes). Les labels permettent de faire référence facilement aux emplacements mémoire dans le code. Voici quelques exemples :

```
L1    db     0         ; octet libelle L1 avec une valeur initiale de 0
L2    dw     1000      ; mot labelle L2 avec une valeur initiale de 1000
L3    db     110101b   ; octet initialise a la valeur binaire 110101 (53 en decimal)
L4    db     12h       ; octet initialise a la valeur hexa 12 (18 en decimal)
L5    db     17o       ; octet initialise a la valeur octale 17 (15 en decimal)
L6    dd     1A92h     ; double mot initialise a la valeur hexa 1A92
L7    resb   1         ; 1 octet non initialise
L8    db     "A"       ; octet initialise avec le code ASCII du A (65)
```

Les doubles et simples quotes sont traitées de la même façon. Les définitions de données consécutives sont stockées séquentiellement en mémoire. C'est à dire que le mot L2 est stocké immédiatement après L1 en mémoire. Des séquences de mémoire peuvent également être définies.

```
L9    db     0, 1, 2, 3              ; definit 4 octets
```

```
L10   db     "w", "o", "r", 'd', 0   ; definit une chaine C = "word"
L11   db     'word', 0               ; idem L10
```

La directive `DD` peut être utilisée pour définir à la fois des entiers et des constantes à virgule flottante en simple précision[5]. Cependant, la directive `DQ` ne peut être utilisée que pour définir des constantes à virgule flottante en double précision.

Pour les grandes séquences, la directive de NASM `TIMES` est souvent utile. Cette directive répète son opérande un certain nombre de fois. Par exemple :

```
L12   times 100 db 0                 ; equivalent a 100 (db 0)
L13   resw   100                     ; reserve de la place pour 100 mots
```

Souvenez vous que les labels peuvent être utilisés pour faire référence à des données dans le code. Il y a deux façons d'utiliser les labels. Si un label simple est utilisé, il fait référence à l'adresse (ou offset) de la donnée. Si le label est placé entre crochets (`[]`), il est interprété comme la donnée à cette adresse. En d'autres termes, il faut considérer un label comme un *pointeur* vers la donnée et les crochets déréférencent le pointeur, exactement comme l'astérisque en C (MASM/TASM utilisent une convention différente). En mode 32 bits, les adresses sont sur 32 bits. Voici quelques exemples :

```
1       mov    al, [L1]       ; Copie l'octet situe en L1 dans AL
2       mov    eax, L1        ; EAX = addresse de l'octet en L1
3       mov    [L1], ah       ; copie AH dans l'octet en L1
4       mov    eax, [L6]      ; copie le double mot en L6 dans EAX
5       add    eax, [L6]      ; EAX = EAX + double mot en L6
6       add    [L6], eax      ; double mot en L6 += EAX
7       mov    al, [L6]       ; copie le premier octet du double mot en L6 dans AL
```

La ligne 7 de cet exemple montre une propriété importante de NASM. L'assembleur ne garde *pas* de trace du type de données auquel se réfère le label. C'est au programmeur de s'assurer qu'il (ou elle) utilise un label correctement. Plus tard, il sera courant de stocker des adresses dans les registres et utiliser les registres comme un pointeur en C. Là encore, aucune vérification n'est faite pour savoir si le pointeur est utilisé correctement. A ce niveau, l'assembleur est beaucoup plus sujet à erreur que le C.

Considérons l'instruction suivante :

```
mov    [L6], 1              ; stocke 1 en L6
```

Cette instruction produit une erreur `operation size not specified`. Pourquoi ? Parce que l'assembleur ne sait pas s'il doit considérer 1 comme un

5. Les nombres à virgule flottante en simple précision sont équivalent aux variables `float` en C.

octet, un mot ou un double mot. Pour réparer cela, il faut ajouter un spécificateur de taille :

```
mov    dword [L6], 1        ; stocke 1 en L6
```

Cela indique à l'assembleur de stocker un 1 dans le double mot qui commence en `L6`. Les autres spécificateurs de taille sont : `BYTE`, `WORD`, `QWORD` et `TWORD`[6].

1.3.6 Entrées et Sorties

Les entrées et sorties sont des activités très dépendantes du système. Cela implique un interfaçage avec le matériel. Les langages de plus haut niveau, comme C, fournissent des bibliothèques standards de routines pour une interface de programmation simple, uniforme pour les E/S. Les langages d'assembleur n'ont pas de bibliothèque standard. Ils doivent soit accéder directement au matériel (avec une opération privilégiée en mode protégé) ou utiliser les routines de bas niveau éventuellement fournies par le système d'exploitation.

Il est très courant pour les routines assembleur d'être interfacées avec du C. Un des avantages de cela est que le code assembleur peut utiliser les routines d'E/S de la bibliothèque standard du C. Cependant, il faut connaître les règles de passage des informations aux routines que le C utilise. Ces règles sont trop compliquées pour en parler ici (nous en parlerons plus tard !). Pour simplifier les E/S, l'auteur a développé ses propres routines qui masquent les règles complexes du C et fournissent une interface beaucoup plus simple. Le Tableau 1.4 décrit les routines fournies. Toutes les routines préservent les valeurs de tous les registres, excepté les routines read. Ces routines modifient la valeur du registre EAX. Pour utiliser ces routines, il faut inclure un fichier contenant les informations dont l'assembleur a besoin pour les utiliser. Pour inclure un fichier dans NASM, utilisez la directive du préprocesseur `%include`. La ligne suivante inclut le fichier requis par les routines d'E/S de l'auteur[7] :

```
%include "asm_io.inc"
```

Pour utiliser une de ces routines d'affichage, il faut charger EAX avec la valeur correcte et utiliser une instruction `CALL` pour l'invoquer. L'instruction `CALL` est équivalente à un appel de fonction dans un langage de haut niveau. Elle saute à une autre portion de code mais revient à son origine une fois

6. `TWORD` définit une zone mémoire de 10 octets. Le coprocesseur virgule flottante utilise ce type de données.

7. Le fichier `asm_io.inc` (et le fichier objet `asm_io` que recquiert `asm_io.inc`) sont dans les exemples de code à télécharger sur la page web de ce tutorial, `http://www.drpaulcarter.com/pcasm`

print_int	affiche à l'écran la valeur d'un entier stocké dans EAX
print_char	affiche à l'écran le caractère dont le code ASCII est stocké dans AL
print_string	affiche à l'écran le contenu de la chaîne à l'*adresse* stockée dans EAX. La chaîne doit être une chaîne de type C (*i.e.* terminée par 0).
print_nl	affiche à l'écran un caractère de nouvelle ligne.
read_int	lit un entier au clavier et le stocke dans le registre EAX.
read_char	lit un caractère au clavier et stocke son code ASCII dans le registre EAX.

TABLE 1.4 – Routine d'E/S de l'assembleur

la routine terminée. Le programme d'exemple ci-dessous montre plusieurs exemples d'appel à ces routines d'E/S.

1.3.7 Débogage

La bibliothèque de l'auteur contient également quelques routines utiles pour déboguer les programmes. Ces routines de débogage affichent des informations sur l'état de l'ordinateur sans le modifier. Ces routines sont en fait des *macros* qui sauvegardent l'état courant du processeur puis font appel à une sous-routine. Les macros sont définies dans le fichier `asm_io.inc` dont nous avons parlé plus haut. Les macros sont utilisées comme des instructions ordinaires. Les opérandes des macros sont séparés par des virgules.

Il y a quatre routines de débogage nommées `dump_regs`, `dump_mem`, `dump_stack` et `dump_math` ; elles affichent respectivement les valeurs des registres, de la mémoire, de la pile et du coprocesseur arithmétique.

dump_regs Cette macro affiche les valeurs des registres (en hexadécimal) de l'ordinateur sur `stdout` (*i.e.* l'écran). Elle affiche également les bits positionnés dans le registre FLAGS[8]. Par exemple, si le drapeau zéro (zero flag) est à 1, *ZF* est affiché. S'il est à 0, il n'est pas affiché. Elle prend en argument un entier qui est affiché également. Cela peut aider à distinguer la sortie de différentes commandes `dump_regs`.

dump_mem Cette macro affiche les valeurs d'une région de la mémoire (en hexadécimal et également en caractères ASCII). Elle prend trois arguments séparés par des virgules. Le premier est un entier utilisé pour étiqueter la sortie (comme l'argument de `dump_regs`). Le second argument est l'adresse à afficher (cela peut être un label). Le dernier

8. Le Chapitre 2 parle de ce registre

argument est le nombre de paragraphes de 16 octets à afficher après l'adresse. La mémoire affichée commencera au premier multiple de paragraphe avant l'adresse demandée.

dump_stack Cette macro affiche les valeurs de la pile du processeur (la pile sera présentée dans le Chapitre 4). La pile est organisée en double mots et cette routine les affiche de cette façon. Elle prend trois arguments délimités par des virgules. Le premier est un entier (comme pour `dump_regs`). Le second est le nombre de double mots à afficher *après* l'adresse que le registre `EBP` contient et le troisième argument est le nombre de double mots à afficher *avant* l'adresse contenue dans `EBP`.

dump_math Cette macro affiche les valeurs des registres du coprocesseur arithmétique. Elle ne prend qu'un entier en argument qui est utilisé pour étiqueter la sortie comme le fait l'argument de `dump_regs`.

1.4 Créer un Programme

Aujourd'hui, il est très peu courant de créer un programme autonome écrit complètement en langage assembleur. L'assembleur est habituellement utilisé pour optimiser certaines routines critiques. Pourquoi ? Il est *beaucoup* plus simple de programmer dans un langage de plus haut niveau qu'en assembleur. De plus, utiliser l'assembleur rend le programme très dur à porter sur d'autres plateformes. En fait, il est rare d'utiliser l'assembleur tout court.

Alors, pourquoi apprendre l'assembleur ?

1. Quelques fois, le code écrit en assembleur peut être plus rapide et plus compact que le code généré par un compilateur.
2. L'assembleur permet l'accès à des fonctionnalités matérielles du système directement qu'il pourrait être difficile ou impossible à utiliser depuis un langage de plus haut niveau.
3. Apprendre à programmer en assembleur aide à acquérir une compréhension plus profonde de la façon dont fonctionne un ordinateur.
4. Apprendre à programmer en assembleur aide à mieux comprendre comment les compilateurs et les langage de haut niveau comme C fonctionnent.

Ces deux dernier points démontrent qu'apprendre l'assembleur peut être utile même si on ne programme jamais dans ce langage plus tard. En fait, l'auteur programme rarement en assembleur, mais il utilise les leçons qu'il en a tiré tous les jours.

```
1 int main()
2 {
3   int ret_status;
4   ret_status = asm_main();
5   return ret_status;
6 }
```

FIGURE 1.6 – Code de `driver.c`

1.4.1 Premier programme

Les programmes qui suivront dans ce texte partiront tous du programme de lancement en C de la Figure 1.6. Il appelle simplement une autre fonction nommée `asm_main`. C'est la routine qui sera écrite en assemble proprement dit. Il y a plusieurs avantages à utiliser un programme de lancement en C. Tout d'abord, cela laisse le système du C initialiser le programme de façon à fonctionner correctement en mode protégé. Tous les segments et les registres correspondants seront initialisés par le C. L'assembleur n'aura pas à se préoccuper de cela. Ensuite, la bibliothèque du C pourra être utilisée par le code assembleur. Les routines d'E/S de l'auteur en tirent partie. Elles utilisent les fonctions d'E/S du C (`printf`, *etc.*). L'exemple suivant montre un programme assembleur simple.

first.asm

```
1  ; fichier: first.asm
2  ; Premier programme assembleur. Ce programme attend la saisie de deux
3  ; entiers et affiche leur somme.
4  ;
5  ; Pour creer l'executable en utilisant djgpp :
6  ; nasm -f coff first.asm
7  ; gcc -o first first.o driver.c asm_io.o
8
9  %include "asm_io.inc"
10 ;
11 ; Les donnees initialisees sont placees dans le segment .data
12 ;
13 segment .data
14 ;
15 ; Ces labels referencent les chaines utilisees pour l'affichage
16 ;
17 prompt1 db    "Entrez un nombre : ", 0      ; N'oubliez pas le 0 final
18 prompt2 db    "Entrez un autre nombre : ", 0
19 outmsg1 db    "Vous avez entre ", 0
```

```
20 outmsg2 db    " et ", 0
21 outmsg3 db    ", leur somme vaut ", 0
22
23 ;
24 ; Les donnees non initialisees sont placees dans le segment .bss
25 ;
26 segment .bss
27 ;
28 ; Ces labels referencent les doubles mots utilises pour stocker les entrees
29 ;
30 input1  resd 1
31 input2  resd 1
32
33 ;
34 ; Le code est place dans le segment .text
35 ;
36 segment .text
37         global  _asm_main
38 _asm_main:
39         enter   0,0             ; initialisation
40         pusha
41
42         mov     eax, prompt1    ; affiche un message
43         call    print_string
44
45         call    read_int        ; lit un entier
46         mov     [input1], eax   ; le stocke dans input1
47
48         mov     eax, prompt2    ; affiche un message
49         call    print_string
50
51         call    read_int        ; lit un entier
52         mov     [input2], eax   ; le stocke dans input2
53
54         mov     eax, [input1]   ; eax = dword en input1
55         add     eax, [input2]   ; eax += dword en input2
56         mov     ebx, eax        ; ebx = eax
57
58         dump_regs 1             ; affiche les valeurs des registres
59         dump_mem  2, outmsg1, 1 ; affiche le contenu de la memoire
60 ;
61 ; Ce qui suit affiche le message resultat en plusieurs etapes
```

```
62 ;
63         mov     eax, outmsg1
64         call    print_string     ; affiche le premier message
65         mov     eax, [input1]
66         call    print_int        ; affiche input1
67         mov     eax, outmsg2
68         call    print_string     ; affiche le second message
69         mov     eax, [input2]
70         call    print_int        ; affiche input2
71         mov     eax, outmsg3
72         call    print_string     ; affiche le troisieme message
73         mov     eax, ebx
74         call    print_int        ; affiche la somme (ebx)
75         call    print_nl         ; affiche une nouvelle ligne
76
77         popa
78         mov     eax, 0           ; retourne dans le programme C
79         leave
80         ret
```

first.asm

La ligne 13 du programme définit une section qui spécifie la mémoire à stocker dans le segment de données (dont le nom est `.data`). Seules les données initialisées doivent être définies dans ce segment. Dans les lignes 17 à 21, plusieurs chaînes sont déclarées. Elle seront affichées avec la bibliothèque C et doivent donc se terminer par un caractère *null* (code ASCII 0). Souvenez vous qu'il y a une grande différence entre `0` et `'0'`.

Les données non initialisées doivent être déclarées dans le segment bss (appelé `.bss` à la ligne 26). Ce segment tient son nom d'un vieil opérateur assembleur UNIX qui signifiait "block started by symbol". Il y a également un segment de pile. Nous en parlerons plus tard.

Le segment de code est appelé `.text` historiquement. C'est là que les instructions sont placées. Notez que le label de code pour la routine main (ligne 38) a un préfixe de soulignement. Cela fait partie de la *convention d'appel C*. Cette convention spécifie les règles que le C utilise lorsqu'il compile le code. Il est très important de connaître cette convention lorsque l'on interface du C et de l'assembleur. Plus loin, la convention sera présentée dans son intégralité ; cependant, pour l'instant, il suffit de savoir que tous les symboles C (*i.e.*, les fonctions et les variables globales) on un préfixe de soulignement qui leur est ajouté par le compilateur C (cette règle s'applique spécifiquement pour DOS/Windows, le compilateur C Linux n'ajoute rien du tout aux noms des symboles).

La directive `global` ligne 37 indique à l'assembleur de rendre le label

`_asm_main` global. Contrairement au C, les labels ont une *portée interne* par défaut. Cela signifie que seul le code du même module peut utiliser le label. La directive `global` donne au(x) label(s) spécifié(s) une *portée externe*. Ce type de label peut être accédé par n'importe quel module du programme. Le module `asm_io` déclare les labels `print_int`, *et.al.* comme étant globaux. C'est pourquoi l'on peut les utiliser dans le module `first.asm`.

1.4.2 Dépendance vis à vis du compilareur

Le code assembleur ci-dessus est spécifique au compilateur C/C++ GNU [9] gratuit DJGPP [10]. Ce compilateur peut être téléchargé gratuitement depuis Internet. Il nécessite un PC à base de 386 ou plus et tourne sous DOS, Windows 95/98 ou NT. Ce compilateur utilise des fichiers objets au format COFF (Common Object File Format). Pour assembler le fichier au format COFF, utilisez l'option `-f coff` avec `nasm` (comme l'indiquent les commentaires du code). L'extension du fichier objet sera `o`.

Le compilateur C Linux est également un compilateur GNU. Pour convertir le code ci-dessus afin qu'il tourne sous Linux, retirez simplement les préfixes de soulignement aux lignes 37 et 38. Linux utilise le format ELF (Executable and Linkable Format) pour les fichiers objet. Utilisez l'option `-f elf` pour Linux. Il produit également un fichier objet avec l'extension `o`.

Les fichiers spécifiques au compilateur, disponibles sur le site de l'auteur, ont déjà été modifiés pour fonctionner avec le compilateur approprié.

Borland C/C++ est un autre compilateur populaire. Il utilise le format Microsoft OMF pour les fichier objets. Utilisez l'option `-f obj` pour les compilateurs Borland. L'extension du fichier objet sera `obj`. Le format OMF utilise des directives `segment` différentes de celles des autres formats objet. Le segment data (ligne 13) doit être changé en :

```
segment _DATA public align=4 class=DATA use32
```

Le segment bss (ligne 26) doit être changé en :

```
segment _BSS public align=4 class=BSS use32
```

Le segment text (ligne 36) doit être changé en :

```
segment _TEXT public align=1 class=CODE use32
```

De plus, une nouvelle ligne doit être ajoutée avant la ligne 36 :

```
group DGROUP _BSS _DATA
```

Le compilateur Microsoft C/C++ peut utiliser soit le format OMF, soit le format Win32 pour les fichiers objet (Si on lui passe un format OMF, il convertit les informations au format Win32 en interne). Le format Win32 permet de définir les segment comme pour DJGPP et Linux. Utilisez l'option `-f win32` pour produire un fichier objet dans ce format. L'extension du fichier objet sera `obj`.

9. GNU est un projet de Free Software Foundation (`http://www.fsf.org`)
10. `http://www.delorie.com/djgpp`

1.4.3 Assembler le code

La première étape consiste à assembler le code. Depuis la ligne de commande, tappez :

```
nasm -f format-objet first.asm
```

où *format-objet* est soit *coff*, soit *elf*, soit *obj* soit *win32* selon le compilateur C utilisé (Souvenez vous que le fichier source doit être modifié pour Linux et pour Borland).

1.4.4 Compiler le code C

Compilez le fichier `driver.c` en utilisant un compilateur C. Pour DJGPP, utilisez :

```
gcc -c driver.c
```

L'option `-c` signifie de compiler uniquement, sans essayer de lier. Cette option fonctionne à la fois sur les compilateurs Linux, Borland et Microsoft.

1.4.5 Lier les fichiers objets

L'édition de liens est le procédé qui consiste à combiner le code machine et les données des fichiers objet et des bibliothèques afin de créer un fichier exécutable. Comme nous allons le voir, ce processus est compliqué.

Le code C nécessite la bibliothèque standard du C et un *code de démarrage* spécial afin de s'exécuter. Il est *beaucoup* plus simple de laisser le compilateur C appeler l'éditeur de liens avec les paramètres corrects que d'essayer d'appeler l'éditeur de liens directement. Par exemple, pour lier le code du premier programme en utilisant DJGPP, utilisez :

```
gcc -o first driver.o first.o asm_io.o
```

Cela crée un exécutable appelé `first.exe` (ou juste `first` sous Linux).

Avec Borland, on utiliserait :

```
bcc32 first.obj driver.obj asm_io.obj
```

Borland utilise le nom du premier fichier afin de déterminer le nom de l'exécutable. Donc, dans le cas ci-dessus, le programme s'appelera `first.exe`.

Il est possible de combiner les étapes de compilation et d'édition de liens. Par exemple :

```
gcc -o first driver.c first.o asm_io.o
```

Maintenant `gcc` compilera `driver.c` puis liera.

1.4.6 Comprendre un listing assembleur

L'option `-l` *`fichier-listing`* peut être utilisée pour indiquer à `nasm` de créer un fichier listing avec le nom donné. Ce fichier montre comment le code a été assemblé. Voici comment les lignes 17 et 18 (dans le segment data) apparaîssent dans le fichier listing (les numéros de ligne sont dans le fichier listing ; cependant, notez que les numéro de ligne dans le fichier source peuvent ne pas être les mêmes).

Les nombres diffèrent sur la version française car ce ne sont pas les mêmes caractères.

```
48 00000000 456E7465722061206E-     prompt1 db    "Entrez un nombre : ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Entrez un autre nombre : ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

La première colonne de chaque ligne est le numéro de la ligne et la seconde est le déplacement (en hexa) de la donnée dans le segment. La troisième colonne montre les données hexa brutes qui seront stockées. Dans ce cas, les données hexa correspondent aux codes ASCII. Enfin, le texte du fichier source est affiché sur la droite. Les déplacements listés dans la seconde colonne ne sont très probalement *pas* les déplacements réels auxquels les données seront placées dans le programme terminé. Chaque module peut définir ses propres labels dans le segment de données (et les autres segments également). Lors de l'étape d'édition de liens (voir Section 1.4.5), toutes ces définitions de labels de segment de données sont combinées pour ne former qu'un segment de données. Les déplacements finaux sont alors calculés par l'éditeur de liens.

Voici une petite portion (lignes 54 à 56 du fichier source) du segment text dans le fichier listing :

```
94 0000002C A1[00000000]            mov     eax, [input1]
95 00000031 0305[04000000]          add     eax, [input2]
96 00000037 89C3                    mov     ebx, eax
```

La troisième colonne montre le code machine généré par l'assembleur. Souvent le code complet pour une instruction ne peut pas encore être calculé. Par exemple, ligne 94, le déplacement (ou l'adresse) de `input1` n'est pas connu avant que le code ne soit lié. L'assembleur peut calculer l'opcode pour l'instruction `mov` (qui, d'après le listing, est A1), mais il écrit le déplacement entre crochets car la valeur exacte ne peut pas encore être calculée. Dans ce cas, un déplacement temporaire de 0 est utilisé car `input1` est au début du segment bss déclaré dans ce fichier. Souvenez vous que cela ne signifie *pas* qu'il sera au début du segment bss final du programme. Lorsque le code est lié, l'éditeur de liens insèrera le déplacement correct à la place. D'autres instructions, comme ligne 96, ne font référence à aucun label. Dans ce cas, l'assembleur peut calculer le code machine complet.

Réprésentations Big et Little Endian

Si l'on regarde attentivement la ligne 95, quelque chose semble très bizarre à propos du déplacement enrte crochets dans le code machine. Le label `input2` est au déplacement 4 (comme défini dans ce fichier) ; cepedant, le déplacement qui apparaît en mémoire n'est pas 00000004 mais 04000000. Pourquoi ? Des processeurs différents stockent les entiers multioctets dans des ordres différents en mémoire. Il y a deux méthodes populaires pour stocker les entiers : *big endian* et *little endian*. Le big endian est la méthode qui semble la plus naturelle. L'octet le plus fort (*i.e.* le plus significatif) est stocké en premier, puis le second plus fort, *etc.* Par exemple, le dword 00000004 serait stocké sous la forme des quatre octets suivants : 00 00 00 04. Les mainframes IBM, la plupart des processeurs RISC et les processeur Motorola utilisent tous cette méthode big endian. Cependant, les processeurs de type Intel utilisent la méthode little endian ! Ici, l'octet le moins significatif est stocké en premier. Donc, 00000004 est stocké en mémoire sous la forme 04 00 00 00. Ce format est codé en dur dans le processeur et ne peut pas être changé. Normalement, le programmeur n'a pas besoin de s'inquiéter du format utilisé. Cependant, il y a des circonstances où c'est important.

Endian est prononcé comme indian.

1. Lorsque des données binaires sont transférées entre différents ordinateurs (soit via des fichiers, soit via un réseau).
2. Lorsque des données binaires sont écrites en mémoire comme un entier multioctet puis relues comme des octets individuels ou *vice versa*.

Le caractère big ou little endian ne s'applique pas à l'ordre des éléments d'un tableau. Le premier élément d'un tableau est toujours à l'adresse la plus petite. C'est également valable pour les chaînes (qui sont juste des tableaux de caractères). Cependant, le caractère big ou little endian s'applique toujours aux éléments individuels des tableaux.

1.5 Fichier Squelette

La Figure 1.7 montre un fichier squelette qui peut être utilisé comme point de départ pour l'écriture de programmes assembleur.

skel.asm

```
1  %include "asm_io.inc"
2  segment .data
3  ;
4  ; Les donnees initialisees sont placees dans ce segment de donnees
5  ;
6
7  segment .bss
8  ;
9  ; Les donnees non initialisees sont placees dans le segment bss
10 ;
11
12 segment .text
13         global  _asm_main
14 _asm_main:
15         enter   0,0               ; initialisation
16         pusha
17
18 ;
19 ; Le code est place dans le segment text. Ne modifiez pas le code
20 ; place avant ou apres ce commentaire.
21 ;
22
23         popa
24         mov     eax, 0            ; retour au C
25         leave
26         ret
```

skel.asm

FIGURE 1.7 – Squelette de Programme

Chapitre 2

Bases du Langage Assembleur

2.1 Travailler avec les Entiers

2.1.1 Représentation des entiers

Les entiers se décomposent en deux catégories : signés et non signés. Les entiers non signés (qui sont positifs) sont représentés d'une manière binaire très intuitive. Les nombre 200 en tant qu'entier non signé sur un octet serait représenté par 11001000 (ou C8 en hexa).

Les entiers signés (qui peuvent être positifs ou négatifs) sont représentés d'une façon plus compliquée. Par exemple, considérons -56. $+56$ serait représenté par l'octet 00111000. Sur papier, on peut représenter -56 comme -111000, mais comment cela serait-il représenté dans un octet en mémoire de l'ordinateur. Comment serait stocké le signe moins ?

Il y a trois techniques principales qui ont été utilisées pour représenter les entiers signés dans la mémoire de l'ordinateur. Toutes ces méthodes utilisent le bit le plus significatif de l'entier comme *bit de signe*. Ce bit vaut 0 si le nombre est positif et 1 s'il est négatif.

Grandeur signée

La première méthode est la plus simple, elle est appelée *grandeur signée*. Elle représente l'entier en deux parties. La première partie est le bit de signe et la seconde est la grandeur entière. Donc 56 serait représenté par l'octet $\underline{0}$0111000 (le bit de signe est souligné) et -56 serait $\underline{1}$0111000. La valeur d'octet la plus grande serait $\underline{0}$1111111 soit $+127$ et la plus petite valeur sur un octet serait $\underline{1}$1111111 soit -127. Pour obtenir l'opposé d'une valeur, on inverse le bit de signe. Cette méthode est intuitive mais a ses inconvénients. Tout d'abord, il y a deux valeurs possibles pour 0 : $+0$ ($\underline{0}$0000000) et -0 ($\underline{1}$0000000). Comme zéro n'est ni positif ni négatif, ces deux représentations devraient se comporter de la même façon. Cela complique la logique de

l'arithmétique pour le processeur. De plus, l'arithmétique générale est également compliquée. Si l'on ajoute 10 à −56, cela doit être transformé en 10 moins 56. Là encore, cela complique la logique du processeur.

Complément à 1

La seconde méthode est appelée représentation en *complément à un*. Le complément à un d'un nombre est trouvé en inversant chaque bit du nombre (une autre façon de l'obtenir est que la valeur du nouveau bit est $1 - \text{anciennevaleurdubit}$). Par exemple, le complément à un de $\underline{0}0111000$ (+56) est $\underline{1}1000111$. Dans la notation en complément à un, calculer le complément à un est équivalent à la négation. Donc $\underline{1}1000111$ est la représentation de −56. Notez que le bit de signe a été automatiquement changé par le complément à un et que, comme l'on s'y attendait, appliquer le complément à un deux fois redonne le nombre de départ. Comme pour la première méthode, il y a deux représentation pour 0 : $\underline{0}0000000$ (+0) et $\underline{1}1111111$ (−0). L'arithmétique des nombres en complément à un est compliquée.

Voici une astuce utile pour trouver le complément à un d'un nombre en hexadécimal sans repasser en binaire. L'astuce est d'ôter le chiffre hexa de F (ou 15 en décimal). Cette méthode suppose que le nombre de bits dans le nombre est multiple de 4. Voici un exemple : +56 est représenté par 38 en hexa. pour trouver le complément à un, ôtez chaque chiffre de F pour obtenir C7 en hexa. Cela concorde avec le résultat ci-dessus.

Complément à deux

Les deux premières méthodes décrites ont été utilisées sur les premiers ordinateurs. Les ordinateurs modernes utilisent une troisième méthode appelée représentation en *complément à deux*. On trouve le e complément à deux d'un nombre en effectuant les deux opérations suivantes :

1. Trouver le complément à un du nombre
2. Ajouter un au résultat de l'étape 1

Voici un exemple en utilisant $\underline{0}0111000$ (56). Tout d'abord, on calcule le complément à un : $\underline{1}1000111$. Puis, on ajoute un :

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad 1 \\ \hline & \underline{1}1001000 \end{array}$$

Dans la notation en complément à deux, calculer le complément à deux est équivalent à trouver l'opposé d'un nombre. Donc, $\underline{1}1001000$ est la représentation en complément à deux de −56. Deux négations devraient reproduire le nombre original. Curieusement le complément à deux ne rempli pas

Nombre	Représentation Hexa
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

TABLE 2.1 – Two's Complement Representation

cette condition. Prenez le complément à deux de $\underline{1}1001000$ en ajoutant un au complément à un.

$$\begin{array}{rr} & \underline{0}0110111 \\ + & \qquad 1 \\ \hline & \underline{0}0111000 \end{array}$$

Lorsque l'on effectue l'addition dans l'opération de complémentation à deux, l'addition du bit le plus à gauche peut produire une retenue. Cette retenue n'est *pas* utilisée. Souvenez vous que toutes les données sur un ordinateur sont d'une taille fixe (en terme de nombre de bits). Ajouter deux octets produit toujours un résultat sur un octet (comme ajouter deux mots donne un mot, *etc.*) Cette propriété est importante pour la notation en complément à deux. Par exemple, considérons zéro comme un nombre en complément à deux sur un octet ($\underline{0}0000000$). Calculer son complément à deux produit la somme :

$$\begin{array}{rr} & \underline{1}1111111 \\ + & \qquad 1 \\ \hline c & \underline{0}0000000 \end{array}$$

où c représente une retenue (Plus tard, nous montrerons comment détecter cette retenue, mais elle n'est pas stockée dans le résultat). Donc, en notation en complément à deux, il n'y a qu'un zéro. Cela rend l'arithmétique en complément à deux plus simple que les méthodes précédentes.

En utilisant la notation en complément à deux, un octet signé peut être utilisé pour représenter les nombres -128 à $+127$. Le Tableau 2.1 montre quelques valeurs choisies. Si 16 bits sont utilisés, les nombres signés $-32,768$ à $+32,767$ peuvent être représentés. $+32,767$ est représenté par 7FFF, $-32,768$ par 8000, -128 par FF80 et -1 par FFFF. Les nombres en complément à deux sur 32 bits vont de -2 milliards à $+2$ milliards environ.

Le processeur n'a aucune idée de ce qu'un octet en particulier (ou un mot ou un double mot) est supposé représenté. L'assembleur n'a pas le concept de types qu'un langage de plus haut niveau peut avoir. La façon dont les données sont interprétées dépendent de l'instruction dans laquelle on les

utilise. Que la valeur FF soit considérée comme représentant un -1 signé ou un $+255$ non signé dépend du programmeur. Le langage C définit des types entiers signés et non signés. Cela permet au compilateur C de déterminer les instructions correctes à utiliser avec les données.

2.1.2 Extension de signe

En assembleur, toutes les données ont une taille bien spécifiée. Il n'est pas rare de devoir changer la taille d'une donnée pour l'utiliser avec d'autres. Réduire la taille est le plus simple.

Réduire la taille des données

Pour réduire la taille d'une donnée, il suffit d'en retirer les bits les plus significatifs. Voici un exemple trivial :

```
mov    ax, 0034h      ; ax = 52 (stocké sur 16 bits)
mov    cl, al         ; cl = les 8 bits de poids faible de ax
```

Bien sûr, si le nombre ne peut pas être représenté correctement dans une plus petite taille, réduire la taille ne fonctionne pas. Par exemple, si `AX` valait 0134h (ou 308 en décimal) alors le code ci-dessus mettrait quand même `CL` à 34h. Cette méthode fonctionne à la fois avec les nombres signés et non signés. Considérons les nombres signés, si `AX` valait FFFFh (-1 sur un mot), alors `CL` vaudrait FFh (-1 sur un octet). Cependant, notez que si la valeur dans `AX`) était signé, elle aurait été tronqué et le résultat aurait été faux !

La règle pour les nombres non signés est que tous les bits retirés soient à 0 afin que la convertion soit correcte. La règle pour les nombres signés est que les bits retirés doivent soit tous être des 1 soit tous des 0. De plus, Le premier bit à ne pas être retiré doit valoir la même chose que ceux qui l'ont été. Ce bit sera le bit de signe pour la valeur plus petite. Il est important qu'il ait la même valeur que le bit de signe original !

Augmenter la taille des données

Augmenter la taille des données est plus compliqué que la réduire. Considérons l'octet hexa FF. S'il est étendu à un mot, quelle valeur aura-t-il ? Cela dépend de la façon dont FF est interprété. Si FF est un octet non signé (255 en décimal), alors le mot doit être 00FF ; cependant, s'il s'agit d'un octet signé (-1 en décimal), alors le mot doit être FFFF.

En général, pour étendre un nombre non signé, on met tous les bits supplémentaires du nouveau nombre à 0. Donc, FF devient 00FF. Cependant, pour étendre un nombre signé, on doit *étendre* le bit de signe. Cela signifie que les nouveaux bits deviennent des copies du bit de signe. Comme le bit

de signe de FF est à 1, les nouveaux bits doivent également être des uns, pour donner FFFF. Si le nombre signé 5A (90 en décimal) était étendu, le résultat serait 005A.

Il y a plusieurs instructions fournies par le 80386 pour étendre les nombres. Souvenez vous que l'ordinateur ne sait pas si un nombre est signé ou non. C'est au programmeur d'utiliser l'instruction adéquate.

Pour les nombres non signés, on peut placer des 0 dans les bits de poids fort de façon simple en utilisant l'instruction `MOV`. Par axemple pour étendre l'octet de AL en un mot non signé dans AX :

```
mov    ah, 0   ; met les 8 bits de poids for à 0
```

Cependant, il n'est pas possible d'utiliser une instruction `MOV` pour convertir le mot non signé de AX en un double mot non signé dans EAX. Pourquoi pas ? Il n'y a aucune façon d'accéder aux 16 bits de poids fort de EAX dans un `MOV`. Le 80386 résoud ce problème en fournissant une nouvelle instruction : `MOVZX`. Cette instruction a deux opérandes. La destination (premier opérande) doit être un registre de 16 ou 32 bits. La source (deuxième opérande) doit être un registre 8 ou 16 bits. L'autre restriction est que la destination doit être plus grande que la source (la plupart des instructions recquièrent que la source soit de la même taille que la destination). Voici quelques exemples :

```
movzx  eax, ax      ; étend ax à eax
movzx  eax, al      ; étend al à eax
movzx  ax, al       ; étend al à ax
movzx  ebx, ax      ; étend ax à ebx
```

Pour les nombres signés, il n'y a pas de façon simple d'utiliser l'instruction `MOV` quelque soit le cas. Le 8086 fournit plusieurs instructions pour étendre les nombres signés. L'instruction `CBW` (Convert Byte to Word, Convertir un Octet en Mot) étend le signe du registre AL dans AX. Les opérandes sont implicites. L'instruction `CWD` (Convert Word to Double word, Convertir un Mot en Double mot) étend le signe de AX dans DX:AX. La notation DX:AX signifie qu'il faut considérer les registres DX et AX comme un seul registre 32 bits avec les 16 bits de poids forts dans DX et les bits de poids faible dans AX (souvenez vous que le 8086 n'avait pas de registre 32 bits du tout !). Le 80386 a apporté plusieurs nouvelles instructions. L'instruction `CWDE` (Convert Word to Double word Extended, convertir un mot en double mot étendue) étend le signe de AX dans EAX. L'instruction `CDQ` (Convert Double word to Quad word, Convertir un Double mot en Quadruple mot) étend le signe de EAX dans EDX:EAX (64 bits !) Enfin, l'instruction `MOVSX` fonctionne comme `MOVZX` excepté qu'elle utilise les règles des nombres signés.

```
1 unsigned char uchar = 0xFF;
2 signed char   schar = 0xFF;
3 int a = (int) uchar;      /* a = 255 (0x000000FF) */
4 int b = (int) schar;      /* b = -1 (0xFFFFFFFF) */
```

FIGURE 2.1 – Extension de signe en C

```
char ch;
while( (ch = fgetc(fp)) != EOF ) {
  /* faire qqch avec ch */
}
```

FIGURE 2.2 – Utilisation de la fonction fgetc

Application à la programmation en C

Le C ANSI ne définit pas si le type `char` *est signé ou non, c'est à chaque compilateur de le décider. C'est pourquoi on définit le type explicitement dans la Figure 2.1.*

L'extension d'entiers signés et non signés a également lieu en C. Les variables en C peuvent être déclarées soit comme signées soit comme non signées (les `int` sont signés). Considérons le code de la Figure 2.1. A la ligne 3, la variable `a` est étendue en utilisant les règles pour les valeurs non signées (avec `MOVZX`), mais à la ligne 4, les règles signées sont utilisées pour `b` (avec `MOVSX`).

Il existe un bug de programmation courant en C qui est directement lié à notre sujet. Considérons le code de la Figure 2.2. La prototype de `fgetc()`est :

```
int fgetc( FILE * );
```

On peut se demander pourquoi est-ce que la fonction renvoie un `int` puisqu'elle lit des caractères ? La raison est qu'elle renvoie normalement un `char` (étendu à une valeur `int` en utilisant l'extension de zéro). Cependant, il y a une valeur qu'elle peut retourner qui n'est pas un caractère, `EOF`. C'est une macro habituellement définie comme valant -1. Donc, `fgetc()` retourne soit un `char` étendu à une valeur `int` (qui ressemble à `000000`xx en hexa) soit `EOF` (qui ressemble à `FFFFFFFF` en hexa).

Le problème avec le programme de la Figure 2.2 est que `fgetc()` renvoie un `int`, mais cette valeur est stockée dans un `char`. Le C tronquera alors les bits de poids fort pour faire tenir la valeur de l'`int` dans le `char`. Le seul problème est que les nombres (en hexa) `000000FF` et `FFFFFFFF` seront tous deux tronqué pour donner l'octet `FF`. Donc, la boucle while ne peut pas distinguer la lecture de l'octet `FF` dans le fichier et la fin de ce fichier.

Ce que fait exactement le code dans ce cas change selon que le `char` est signé ou non. Pourquoi ? Parce que ligne 2, `ch` est comparé avec `EOF`. Comme

`EOF` est un `int` [1], `ch` sera étendu à un `int` afin que les deux valeurs comparées soient de la même taille [2]. Comme la Figure 2.1 le montrait, le fait qu'une variable soit signée ou non est très important.

Si le `char` n'est pas signé, `FF` est étendu et donne `000000FF`. Cette valeur est comparée à `EOF` (`FFFFFFFF`) et est différente. Donc la boucle ne finit jamais !

So le `char` est signé, `FF` est étendu et donne `FFFFFFFF`. Les deux valeurs sont alors égale et la boucle se termine. Cependant, comme l'octet `FF` peut avoir été lu depuis le fichier, la boucle peut se terminer prématurément.

La solution à ce problème est de définir la variable `ch` comme un `int`, pas un `char`. Dans ce cas, aucune troncation ou extension n'est effectuée à la ligne 2. Dans la boucle, il est sans danger de tronquer la valeur puisque `ch` *doit* alors être réellement un octet.

2.1.3 Arithmétique en complément à deux

Comme nous l'avons vu plus tôt, l'instruction `add` effectue une addition et l'instruction `sub` effectue une soustraction. Deux des bits du registre EFLAGS que ces instructions positionnent sont *overflow* (dépassement de capacité) et *carry flag* (retenue). Le drapeau d'overflow est à 1 si le résultat réel de l'opération est trop grand et ne tient pas dans la destination pour l'arithmétique signée. Le drapeau de retenue est à 1 s'il y a une retenue dans le bit le plus significatif d'une addition d'une soustraction. Donc, il peut être utilisé pour détecter un dépassement de capacité en arithmétique non signée. L'utilisation du drapeau de retenue pour l'arithmétique signée va être vu sous peu. Un des grands avantages du complément à 2 est qu les règles pour l'addition et la soustraction sont exactement les mêmes que pour les entiers non signés. Donc, `add` et `sub` peuvent être utilisées pour les entiers signés ou non.

$$\begin{array}{rr} 002\mathrm{C} & 44 \\ +\ \mathrm{FFFF} & +\ (-1) \\ \hline 002\mathrm{B} & 43 \end{array}$$

Il y a une retenue de générée mais elle ne fait pas partie de la réponse.

Il y a deux instructions de multiplication et de division différentes. Tout d'abord, pour multiplier, utilisez les instructions `MUL` ou `IMUL`. L'instruction `MUL` est utilisée pour multiplier les nombres non signés et `IMUL` est utilisée pour multiplier les entiers signés. Pourquoi deux instructions différentes sont nécessaires ? Les règles pour la multiplication sont différentes pour les nombres non signés et les nombres signés en complément à 2. Pourquoi cela ?

1. Il est courant de penser que les fichiers ont un carctère EOF comme dernier caractère. Ce n'est *pas* vrai !
2. La raison de cette nécessité sera exposée plus tard.

dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

TABLE 2.2 – `Instructions imul`

Considérons la multiplication de l'octet FF avec lui-même donnant un résultat sur un mot. En utilisant la multiplication non signée, il s'agit de 255 fois 255 soit 65025 (ou FE01 en hexa). En utilisant la multiplication signée, il s'agit de -1 fois -1 soit 1 (ou 0001 en hexa).

Il y a plusieurs formes pour les instructions de multiplication. La plus ancienne ressemble à cela :

```
mul   source
```

La *source* est soit un registre soit une référence mémoire. Cela ne peut pas être une valeur immédiate. Le type exact de la multiplication dépend de la taille de l'opérande source. Si l'opérande est un octet, elle est multipliée par l'octet situé dans le registre AL et le résultat est stocké dans les 16 bits de AX. Si la source fait 16 bits, elle est multipliée par le mot situé dans AX et le résultat 32 bits est stocké dans DX:AX. Si la source fait 32 bits, elle est multipliée par EAX et le résultat 64 bits est stocké dans EDX:EAX.

L'instruction `IMUL` a les mêmes formats que `MUL`, mais ajoute également d'autres formats d'instruction. Il y a des formats à deux et à trois opérandes :

```
imul   dest, source1
imul   dest, source1, source2
```

Le Tableau 2.2 montre les combinaisons possibles.

Les deux opérateurs de division sont `DIV` et `IDIV`. Il effectuent respectivement une division entière non signée et une division entière signée. Le format général est :

```
div   source
```

Si la source est sur 8 bits, alors AX est divisé par l'opérande. Le quotient est stocké dans AL et le reste dans AH. Si la source est sur 16 bits, alors DX:AX est divisé par l'opérande. Le quotient est stocké dans AX et le reste dans DX. Si la source est sur 32 bits, alors EDX:EAX est divisé par l'opérande, le quotient est stocké dans EAX et le reste dans EDX. L'instruction `IDIV` fonctionne de la même façon. Il n'y a pas d'instructions `IDIV` spéciales comme pour `IMUL`. Si le quotient est trop grand pour tenir dans son registre ou que le diviseur vaut 0, le programme et interrompu et se termine. Une erreur courante est d'oublier d'initialiser DX ou EDX avant la division.

L'instruction `NEG` inverse son opérande en calculant son complément à deux. L'opérande peut être n'importe quel registre ou emplacement mémoire sur 8 bits, 16 bits, or 32 bits.

2.1.4 Programme exemple

math.asm

```
%include "asm_io.inc"
segment .data         ; Chaines affichees
prompt          db    "Entrez un nombre : ", 0
square_msg      db    "Le carre de l'entree vaut ", 0
cube_msg        db    "Le cube de l'entree vaut ", 0
cube25_msg      db    "Le cube de l'entree fois 25 vaut ", 0
quot_msg        db    "Le quotient de cube/100 vaut ", 0
rem_msg         db    "Le reste de cube/100 vaut ", 0
neg_msg         db    "La négation du reste vaut ", 0

segment .bss
input   resd 1

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; routine d'initialisation
        pusha

        mov     eax, prompt
        call    print_string

        call    read_int
        mov     [input], eax

        imul    eax               ; edx:eax = eax * eax
        mov     ebx, eax          ; sauvegarde le résultat dans ebx
```

```
28          mov     eax, square_msg
29          call    print_string
30          mov     eax, ebx
31          call    print_int
32          call    print_nl
33
34          mov     ebx, eax
35          imul    ebx, [input]      ; ebx *= [entree]
36          mov     eax, cube_msg
37          call    print_string
38          mov     eax, ebx
39          call    print_int
40          call    print_nl
41
42          imul    ecx, ebx, 25      ; ecx = ebx*25
43          mov     eax, cube25_msg
44          call    print_string
45          mov     eax, ecx
46          call    print_int
47          call    print_nl
48
49          mov     eax, ebx
50          cdq                       ; initialise edx avec extension de signe
51          mov     ecx, 100          ; on ne peut pas diviser par une valeur immédiate
52          idiv    ecx               ; edx:eax / ecx
53          mov     ecx, eax          ; sauvegarde le résultat dans ecx
54          mov     eax, quot_msg
55          call    print_string
56          mov     eax, ecx
57          call    print_int
58          call    print_nl
59          mov     eax, rem_msg
60          call    print_string
61          mov     eax, edx
62          call    print_int
63          call    print_nl
64
65          neg     edx               ; inverse le reste
66          mov     eax, neg_msg
67          call    print_string
68          mov     eax, edx
69          call    print_int
```

```
70          call    print_nl
71
72          popa
73          mov     eax, 0            ; retour au C
74          leave
75          ret
```
math.asm

2.1.5 Arithmétique en précision étendue

Le langage assembleur fournit également des instructions qui permettent d'effectuer des additions et des soustractions sur des nombres plus grands que des doubles mots. Ces instructions utilisent le drapeau de retenue. Comme nous l'avons dit plus haut, les instructions `ADD` et `SUB` modifient le drapeau de retenue si une retenue est générée. Cette information stockée dans le drapeau de retenue peut être utilisée pour additionner ou soustraire de grands nombres en morcelant l'opération en doubles mots (ou plus petit).

Les instructions `ADC` et `SBB` utilisent les informations données par le drapeau de retenue. L'instruction `ADC` effectue l'opération suivante :

opérande1 = *opérande1* + drapeau de retenue + *opérande2*

L'instruction `SBB` effectue :

opérande1 = *opérande1* - drapeau de retenue - *opérande2*

Comment sont elles utilisées ? Considérons la somme d'entiers 64 bits dans EDX:EAX et EBX:ECX. Le code suivant stockerait la somme dans EDX:EAX :

```
1       add     eax, ecx        ; additionne les 32 bits de poids faible
2       adc     edx, ebx        ; additionne les 32 bits de poids fort et la retenue
```

La soustraction est très similaire. Le code suivant soustrait EBX:ECX de EDX:EAX :

```
1       sub     eax, ecx        ; soustrait les 32 bits de poids faible
2       sbb     edx, ebx        ; soustrait les 32 bits de poids fort et la retenue
```

Pour les nombres *vraiment* grands, une boucle peut être utilisée (voir Section 2.2). Pour une boucle de somme, il serait pratique d'utiliser l'instruction `ADC` pour toutes les itérations (au lieu de toutes sauf la première). Cela peut être fait en utilisant l'instruction `CLC` (CLear Carry) juste avant que la boucle ne commence pour initialiser le drapeau de retenue à 0. Si le drapeau de retenue vaut 0, il n'y a pas de différence entre les instructions `ADD` et `ADC`. La même idée peut être utilisée pour la soustraction.

2.2 Structures de Contrôle

Les langages de haut niveau fournissent des structures de contrôle de haut niveau (*p.e.*, les instructions *if* et *while*) qui contrôlent le flot d'exécution. Le langage assembleur ne fournit pas de structures de contrôle aussi complexes. Il utilise à la place l'infâme *goto* et l'utiliser de manière inappropriée peut conduire à du code spaghetti ! Cependant, il *est* possible d'écrire des programmes structurés en assembleur. La procédure de base est de concevoir la logique du programme en utilisant les structures de contrôle de haut niveau habituelles et de traduire cette conception en langage assembleur (comme le ferait un compilateur).

2.2.1 Comparaison

Les structures de contrôle décident ce qu'il faut faire en comparant des données. En assembleur, le résultat d'une comparaison est stocké dans le registre FLAGS pour être utilisé plus tard. Le 80x86 fournit l'instruction `CMP` pour effectuer des comparaisons. Le registre FLAGS est positionné selon la différence entre les deux opérandes de l'instruction `CMP`. Les opérandes sont soustraites et le registre FLAGS est positionné selon le résultat, mais ce résultat n'est stocké *nulle part*. Si vous avez besoin du résultat, utilisez l'instruction `SUB` au lieu de `CMP`.

Pour les entiers non signés, il y a deux drapeaux (bits dans le registre FLAGS) qui sont importants : les drapeaux zéro (ZF) et retenue (CF) . Le drapeau zero est allumé (1) si le résultat de la différence serait 0. Considérons une comparaison comme :

```
cmp    vleft, vright
```

La différence `vleft - vright` est calculée et les drapeaux sont positionnés en fonction. Si la différence calculée par `CMP` vaut 0, `vleft = vright`, alors ZF est allumé(*i.e.* 1) et CF est éteint (*i.e.* 0). Si `vleft > vright`, ZF est éteint et CF également. Si `vleft < vright`, alors ZF est éteint et CF est allumé (retenue).

Pour les entiers signés, il y a trois drapeaux importants : le drapeau zéro (ZF), le drapeau d'overflow (OF) et le drapeau de signe (SF). Le drapeau d'overflow est allumé si le résultat d'une opération dépasse la capacité (de trop ou de trop peu). Le drapeau de signe est allumé si le résultat d'une opération est négatif. Si `vleft = vright`, ZF est allumé (comme pour les entiers non signés). Si `vleft > vright`, ZF est éteint et SF = OF. Si `vleft < vright`, ZF est éteint et SF $\neq$ OF.

N'oubliez pas que d'autres instructions peuvent aussi changer le registre FLAGS, pas seulement `CMP`.

Pourquoi SF = OF si `vleft > vright` *? S'il n'y a pas d'overflow, alors la différence aura la valeur correcte et doit être positive ou nulle. Donc, SF = OF = 0. Par contre, s'il y a un overflow, la différence n'aura pas la valeur correcte (et sera en fait négative). Donc, SF = OF = 1.*

2.2.2 Instructions de branchement

Les instructions de branchement peuvent transférer l'exécution à n'importe quel point du programme. En d'autres termes, elles agissent comme un *goto*. Il y a deux types de branchements : conditionnel et inconditionnel. Un branchement inconditionnel est exactement comme un goto, il effectue toujours le branchement. Un branchement conditionnel peut effectuer le branchement ou non selon les drapeaux du registre FLAGS. Si un branchement conditionnel n'effecute pas le branchement, le contrôle passe à l'instruction suivante.

L'instruction `JMP` (abbréviation de *jump*) effectue les branchements inconditionnels. Son seul argument est habituellement l'*étiquette de code* de l'instruction à laquelle se brancher. L'assembleur ou l'éditeur de liens remplacera l'étiquette par l'adresse correcte de l'instruction. C'est une autre des opérations pénibles que l'assembleur fait pour rendre la vie du programmeur plus simple. Il est important de réaliser que l'instruction immédiatement après l'instruction `JMP` ne sera jamais exécutée à moins qu'une autre instruction ne se branche dessus !

Il y a plusieurs variantes de l'instruction de saut :

SHORT Ce saut est très limité en portée. Il ne peut bouger que de plus ou moins 128 octets en mémoire. L'avantage de ce type de saut est qu'il utilise moins de mémoire que les autres. Il utilise un seul octet signé pour stocker le *déplacement* du saut. Le déplacement est le nombre d'octets à sauter vers l'avant ou vers l'arrière (le déplacement est ajouté à EIP). Pour spécifier un saut court, utilisez le mot clé `SHORT` immédiatement avant l'étiquette dans l'instruction `JMP`.

NEAR Ce saut est le saut par défaut, que ce soit pour les branchements inconditionnels ou conditionnels, il peut être utilisé pour sauter vers n'importe quel emplacement dans un segment. En fait, le 80386 supporte deux types de sauts proches. L'un utilise 2 ocets pour le déplacement. Cela permet de se déplacer en avant ou en arrière d'environ 32 000 octets. L'autre type utilise quatre octets pour le déplacement, ce qui, bien sûr, permet de se déplacer n'importe où dans le segment. Le type de saut à deux octets peut être spécifié en plaçant le mot clé `WORD` avant l'étiquette dans l'instruction `JMP`.

FAR Ce saut permet de passer le contrôle à un autre segment de code. C'est une chose très rare en mode protégé 386.

Les étiquettes de code valides suivent les mêmes règles que les étiquettes de données. Les étiquettes de code sont définies en les plaçant dans le segment de code au début de l'instruction qu'ils étiquettent. Deux points sont placés à la fin de l'étiquette lors de sa définition. Ces deux points ne font *pas* partie du nom.

JZ	saute uniquement si ZF est allumé
JNZ	saute uniquement si ZF est éteint
JO	saute uniquement si OF est allumé
JNO	saute uniquement si OF est éteint
JS	saute uniquement si SF est allumé
JNS	saute uniquement si SF est éteint
JC	saute uniquement si CF est allumé
JNC	saute uniquement si CF est éteint
JP	saute uniquement si PF est allumé
JNP	saute uniquement si PF est éteint

TABLE 2.3 – Branchements Conditionnels Simples

Il y a beaucoup d'instructions de branchement conditionnel différentes. Elles prennent toutes une étiquette de code comme seule opérande. Les plus simples se contentent de regarder un drapeau dans le registre FLAGS pour déterminer si elles doivent sauter ou non. Voyez le Tableau 2.3 pour une liste de ces instructions (PF est le *drapeau de parité* qui indique si le nombre de bits dans les 8 bits de poids faible du résultat est pair ou impair).

Le pseudo-code suivant :

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

peut être écrit de cette façon en assembleur :

```
1       cmp    eax, 0            ; positionne les drapeaux (ZF allumé si eax - 0 = 0)
2       jz     thenblock         ; si ZF est allumé, branchement vers thenblock
3       mov    ebx, 2            ; partie ELSE du IF
4       jmp    next              ; saute par dessus la partie THEN du IF
5  thenblock:
6       mov    ebx, 1            ; partie THEN du IF
7  next:
```

Les autres comparaisons ne sont pas si simples si l'on se contente des branchements conditionnels du Tableau 2.3. Pour illustrer, considérons le pseudo-code suivant :

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

Signé		Non Signé	
JE	saute si `vleft = vright`	JE	saute si `vleft = vright`
JNE	saute si `vleft` $\neq$ `vright`	JNE	saute si `vleft` $\neq$ `vright`
JL, JNGE	saute si `vleft < vright`	JB, JNAE	saute si `vleft < vright`
JLE, JNG	saute si `vleft` $\leq$ `vright`	JBE, JNA	saute si `vleft` $\leq$ `vright`
JG, JNLE	saute si `vleft > vright`	JA, JNBE	saute si `vleft > vright`
JGE, JNL	saute si `vleft` $\geq$ `vright`	JAE, JNB	saute si `vleft` $\geq$ `vright`

TABLE 2.4 – Instructions de Comparaison Signées et Non Signées

Si EAX est plus grand ou égal à 5, ZF peut être allumé ou non et SF sera égal à OF. Voici le code assembleur qui teste ces conditions (en supposant que EAX est signé) :

```
1        cmp    eax, 5
2        js     signon            ; goto signon si SF = 1
3        jo     elseblock         ; goto elseblock si OF = 1 et SF = 0
4        jmp    thenblock         ; goto thenblock si SF = 0 et OF = 0
5   signon:
6        jo     thenblock         ; goto thenblock si SF = 1 et OF = 1
7   elseblock:
8        mov    ebx, 2
9        jmp    next
10  thenblock:
11       mov    ebx, 1
12  next:
```

Le code ci-dessus est très confus. Heureusement, le 80x86 fournit des instructions de branchement additionnelles pour rendre ce type de test *beaucoup* plus simple. Il y a des versions signées et non signées de chacune. Le Tableau 2.4 montre ces instructions. Les branchements égal et non égal (JE et JNE) sont les mêmes pour les entiers signés et non signés (en fait, JE et JNE sont réellement identiques à JZ et JNZ, respectivement). Chacune des autres instructions de branchement a deux synonymes. Par exemple, regardez JL (jump less than, saut si inférieur à) et JNGE (jump not greater than or equal to, saut si non plus grand ou égal). Ce sont les mêmes instructions car :

$$x < y \Longrightarrow \mathbf{not}(x \geq y)$$

Les branchements non signés utilisent A pour *above* (au dessus) et B pour *below* (en dessous) à la place de L et G.

En utilisant ces nouvelles instructions de branchement, le pseudo-code ci-dessus peut être traduit en assembleur plus facilement.

```
1        cmp    eax, 5
2        jge    thenblock
3        mov    ebx, 2
4        jmp    next
5  thenblock:
6        mov    ebx, 1
7  next:
```

2.2.3 Les instructions de boucle

Le 80x86 fournit plusieurs instructions destinées à implémenter des boucles de style *for*. Chacune de ces instructions prend une étiquette de code comme seule opérande.

LOOP Decrémente ECX, si ECX $\neq 0$, saute vers l'étiquette

LOOPE, LOOPZ Décremente ECX (le registre FLAGS n'est pas modifié), si ECX $\neq 0$ et ZF $= 1$, saute

LOOPNE, LOOPNZ Décremente ECX (FLAGS inchangé), si ECX $\neq 0$ et ZF $= 0$, saute

Les deux dernières instructions de boucle sont utiles pour les boucles de recherche séquentielle. Le pseudo-code suivant :

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

peut être traduit en assembleur de la façon suivante :

```
1        mov    eax, 0          ; eax est sum
2        mov    ecx, 10         ; ecx est i
3  loop_start:
4        add    eax, ecx
5        loop   loop_start
```

2.3 Traduire les Structures de Contrôle Standards

Cette section décrit comment les structures de contrôle standards des langages de haut niveau peuvent être implémentées en langage assembleur.

2.3.1 Instructions if

Le pseudo-code suivant :

```
if ( condition )
  then_block;
else
  else_block;
```

peut être implémenté de cette façon :

```
1        ; code pour positionner FLAGS
2        jxx    else_block    ; choisir xx afin de sauter si la condition est fausse
3        ; code pour le bloc then
4        jmp    endif
5  else_block:
6        ; code pour le bloc else
7  endif:
```

S'il n'y a pas de else, le branchement `else_block` peut être remplacé par un branchement à `endif`.

```
1        ; code pour positionner FLAGS
2        jxx    endif          ; choisir xx afin de sauter si la condition est fausse
3        ; code pour le bloc then
4  endif:
```

2.3.2 Boucles while

La boucle *while* est une boucle avec un test au début :

```
while( condition ) {
  corps de la boucle;
}
```

Cela peut être traduit en :

```
1  while:
2        ; code pour positionner FLAGS selon la condition
3        jxx    endwhile       ; choisir xx pour sauter si la condition est fausse
4        ; body of loop
5        jmp    while
6  endwhile:
```

2.3.3 Boucles do while

La boucle *do while* est une boucle avec un test à la fin :

```
do {
  corps de la boucle;
} while( condition );
```

```
1   unsigned guess;   /* candidat courant pour être premier  */
2   unsigned factor;  /*  diviseur  possible  de guess */
3   unsigned limit;   /* rechercher les nombres premiers jusqu'à cette valeur */
4
5    printf ("Rechercher les  nombres premier jusqu'a  :  ");
6   scanf("%u", &limit);
7    printf ("2\n");     /*  traite  les  deux premier  nombres premiers comme
    */
8    printf ("3\n");     /* des cas spéciaux  */
9   guess = 5;           /* candidat  initial  */
10  while ( guess <= limit ) {
11    /* recherche  un  diviseur  du candidat */
12     factor  = 3;
13     while (  factor*factor  < guess &&
14              guess  % factor  != 0 )
15      factor  += 2;
16     if  ( guess  % factor  != 0 )
17       printf ("%d\n", guess);
18    guess  += 2;    /* ne  regarder  que  les  nombres impairs */
19  }
```

FIGURE 2.3 –

Cela peut être traduit en :

```
1  do:
2         ; corps de la boucle
3         ; code pour positionner FLAGS selon la condition
4         jxx    do          ; choisir xx pour se brancher si la condition est vraie
```

2.4 Exemple : Trouver des Nombres Premiers

Cette section détaille un programme qui trouve des nombres premiers. Rappelez vous que les nombres premiers ne sont divisibles que par 1 et par eux-mêmes. Il n'y a pas de formule pour les rechercher. La méthode de base qu'utilise ce programme est de rechercher les diviseurs de tous les nombres impairs [3] sous une limite donnée. Si aucun facteur ne peut être trouvé pour un nombre impair, il est premier. La Figure 2.3 montre l'algorithme de base écrit en C.

Voici la version assembleur :

3. 2 est le seul nombre premier pair.

prime.asm

```
%include "asm_io.inc"
segment .data
Message         db      "Rechercher les nombres premiers jusqu'a : ", 0

segment .bss
Limit           resd    1               ; rechercher les nombres premiers jusqu'à cette limite
Guess           resd    1               ; candidat courant pour être premier

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; routine d'intialisation
        pusha

        mov     eax, Message
        call    print_string
        call    read_int            ; scanf("%u", &limit);
        mov     [Limit], eax

        mov     eax, 2              ; printf("2\n");
        call    print_int
        call    print_nl
        mov     eax, 3              ; printf("3\n");
        call    print_int
        call    print_nl

        mov     dword [Guess], 5    ; Guess = 5;
while_limit:                        ; while ( Guess <= Limit )
        mov     eax,[Guess]
        cmp     eax, [Limit]
        jnbe    end_while_limit     ; on utilise jnbe car les nombres ne sont pas signes

        mov     ebx, 3              ; ebx est factor = 3;
while_factor:
        mov     eax,ebx
        mul     eax                 ; edx:eax = eax*eax
        jo      end_while_factor    ; si la réponse ne tient pas dans eax seul
        cmp     eax, [Guess]
        jnb     end_while_factor    ; si !(factor*factor < guess)
        mov     eax,[Guess]
        mov     edx,0
```

```
42          div     ebx                 ; edx = edx:eax % ebx
43          cmp     edx, 0
44          je      end_while_factor    ; si !(guess % factor != 0)
45
46          add     ebx,2               ; factor += 2;
47          jmp     while_factor
48  end_while_factor:
49          je      end_if              ; si !(guess % factor != 0)
50          mov     eax,[Guess]         ; printf("%u\n")
51          call    print_int
52          call    print_nl
53  end_if:
54          add     dword [Guess], 2    ; guess += 2
55          jmp     while_limit
56  end_while_limit:
57
58          popa
59          mov     eax, 0            ; retour au C
60          leave
61          ret
```

prime.asm

Chapitre 3

Opérations sur les Bits

3.1 Opérations de Décalage

Le langage assembleur permet au programmeur de manipuler individuellement les bits des données. Une opération courante sur les bits est appelée *décalage*. Une opération de décalage déplace la position des bits d'une donnée. Les décalages peuvent être soit vers la gauche (*i.e.* vers les bits les plus significatifs) soit vers la droite (les bits les moins significatifs).

3.1.1 Décalages logiques

Le décalage logique est le type le plus simple de décalage. Il décale d'une manière très simple. La Figure 3.1 montre l'exemple du décalage d'un nombre sur un octet.

Original	1	1	1	0	1	0	1	0
Décalé à gauche	1	1	0	1	0	1	0	0
Décalé à droite	0	1	1	1	0	1	0	1

FIGURE 3.1 – Décalages logiques

Notez que les nouveaux bits sont toujours à 0. Les instructions `SHL` et `SHR` sont respectivement utilisées pour décaler à gauche et à droite. Ces instructions permettent de décaler de n'importe quel nombre de positions. Le nombre de positions à décaler peut soit être une contante, soit être stocké dans le registre `CL`. Le dernier bit décalé de la donnée est stocké dans le drapeau de retenue. Voici quelques exemples :

```
1  mov    ax, 0C123H
2  shl    ax, 1          ; décale d'1 bit à gauche,  ax = 8246H, CF = 1
3  shr    ax, 1          ; décale d'1 bit à droite,  ax = 4123H, CF = 0
4  shr    ax, 1          ; décale d'1 bit à droite,  ax = 2091H, CF = 1
```

```
5       mov    ax, 0C123H
6       shl    ax, 2           ; décale de 2 bits à gauche,  ax = 048CH, CF = 1
7       mov    cl, 3
8       shr    ax, cl          ; décale de 3 bits à droite, ax = 0091H, CF = 1
```

3.1.2 Utilisation des décalages

La multiplication et la division rapides sont les utilisations les plus courantes des opérations de déclage. Rappelez vous que dans le système décimal, la multiplication et la division par une puissance de dix sont simples, on décale simplement les chiffres. C'est également vrai pour les puissances de deux en binaire. Par exemple, pour multiplier par deux le nombre binaire 1011_2 (ou 11 en décimal), décalez d'un cran vers la gauche pour obtenir 10110_2 (ou 22). Le quotient d'un division par une puissance de deux est le résultat d'un décalage à droite. Pour diviser par deux, utilisez un décalage d'une position à droite ; pour diviser par 4 (2^2), décalez à droite de 2 positions ; pour diviser par 8 (2^3), décalez de 3 positions vers la droite, *etc.* Les instructions de décalage sont très basiques et sont *beaucoup* plus rapides que les instructions `MUL` et `DIV` correspondantes !

En fait, les décalages logiques peuvent être utilisés pour multiplier ou diviser des valeurs non signées. Il ne fonctionnent généralement pas pour les valeurs signées. Considérons la valeur sur deux octets FFFF (-1 signé). Si on lui applique un décalage logique d'une position vers la droite, le résultat est 7FFF ce qui fait $+32,767$! Un autre type de décalage doit être utilisé pour les valeurs signées.

3.1.3 Décalages arithmétiques

Ces décalages sont conçus pour permettre à des nombres signés d'être rapidement multipliés et divisés par des puissances de 2. Il assurent que le bit de signe est traité correctement.

SAL Shift Arithmetic Left (Décalage Arithmétique à Gauche) - Cette instruction est juste un synonyme pour `SHL`. Elle est assemblée pour donner exactement le même code machine que `SHL`. Tant que le bit de signe n'est pas changé par le décalage, le résultat sera correct.

SAR Shift Arithmetic Right (Décalage Arithmétique à Droite) - C'est une nouvelle instruction qui ne décale pas le bit de signe (*i.e.* le msb) de son opérande. Les autres bits sont décalés normalement sauf que les nouveaux bits qui entrent par la gauche sont des copies du bit de signe (c'est-à-dire que si le bit de signe est à 1, les nouveaux bits sont également à 1). Donc, si un octet est décalé avec cette instruction, seuls les 7 bits de poids faible sont décalés. Comme pour les autres décalages, le dernier bit sorti est stocké dans le drapeau de retenue.

```
1       mov     ax, 0C123H
2       sal     ax, 1           ; ax = 8246H, CF = 1
3       sal     ax, 1           ; ax = 048CH, CF = 1
4       sar     ax, 2           ; ax = 0123H, CF = 0
```

3.1.4 Décalages circulaires

Les instructions de décalage circulaire fonctionnent comme les décalages logiques excepté que les bits perdus à un bout sont réintégrés à l'autre. Donc, la donnée est traitée comme s'il s'agissait d'une structure circulaire. Les deux instructions de rotation les plus simples sont `ROL` et `ROR` qui effectuent des rotations à gauche et à droite, respectivement. Comme pour les autres décalages, ceux-ci laissent une copie du dernier bit sorti dans le drapeau de retenue.

```
1       mov     ax, 0C123H
2       rol     ax, 1           ; ax = 8247H, CF = 1
3       rol     ax, 1           ; ax = 048FH, CF = 1
4       rol     ax, 1           ; ax = 091EH, CF = 0
5       ror     ax, 2           ; ax = 8247H, CF = 1
6       ror     ax, 1           ; ax = C123H, CF = 1
```

Il y a deux instructions de rotation supplémentaires qui décalent les bits de la donnée et le drapeau de retenue appelées `RCL` et `RCR`. Par exemple, si on applique au registre `AX` une rotation avec ces instructions, elle appliquée aux 17 bits constitués de `AX` et du drapeau de retenue.

```
1       mov     ax, 0C123H
2       clc                     ; eteint le drapeau de retenue (CF = 0)
3       rcl     ax, 1           ; ax = 8246H, CF = 1
4       rcl     ax, 1           ; ax = 048DH, CF = 1
5       rcl     ax, 1           ; ax = 091BH, CF = 0
6       rcr     ax, 2           ; ax = 8246H, CF = 1
7       rcr     ax, 1           ; ax = C123H, CF = 0
```

3.1.5 Application simple

Voici un extrait de code qui compte le nombre de bits qui sont "allumés" (*i.e.* à 1) dans le registre EAX.

```
1       mov     bl, 0           ; bl contiendra le nombre de bits ALLUMES
2       mov     ecx, 32         ; ecx est le compteur de boucle
```

X	Y	X ET Y
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 3.1 – L'opération ET

	1	0	1	0	1	0	1	0
ET	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

FIGURE 3.2 – Effectuer un ET sur un octet

```
3  count_loop:
4        shl    eax, 1          ; decale un bit dans le drapeau de retenue
5        jnc    skip_inc        ; si CF == 0, goto skip_inc
6        inc    bl
7  skip_inc:
8        loop   count_loop
```

Le code ci-dessus détruit la valeur originale de `EAX` (`EAX` vaut zéro à la fin de la boucle). Si l'on voulait conserver la valeur de `EAX`, la ligne 4 pourrait être remplacée par `rol eax, 1`.

3.2 Opérations Booléennes Niveau Bit

Il y a quatre opérateurs booléens courants : *AND*, *OR*, *XOR* et *NOT*. Une *table de vérité* montre le résultat de chaque opération pour chaque valeur possible de ses opérandes.

3.2.1 L'opération *ET*

Le résultat d'un *ET* sur deux bits vaut 1 uniquement si les deux bits sont à 1, sinon, le résultat vaut 0, comme le montre la table de vérité du Tableau 3.1.

Les processeurs supportent ces opérations comme des instructions agissant de façon indépendante sur tous les bits de la donnée en parallèle. Par exemple, si on applique un *ET* au contenu de `AL` et `BL`, l'opération *ET* de base est appliquée à chacune des 8 paires de bits correspondantes dans les deux registres, comme le montre la Figure 3.2. Voici un exemple de code :

```
1        mov    ax, 0C123H
2        and    ax, 82F6H          ; ax = 8022H
```

X	Y	X OU Y
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 3.2 – L'opération OU

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 3.3 – L'opération XOR

3.2.2 L'opération *OU*

Le *OU* de 2 bits vaut 0 uniquement si les deux bits valent 0, sinon le résultat vaut 1 comme le montre la table de vérité du Tableau 3.2. Voici un exemple de code :

```
1        mov    ax, 0C123H
2        or     ax, 0E831H          ; ax = E933H
```

3.2.3 L'opération *XOR*

Le *OU* exclusif de 2 bits vaut 0 uniquement si les deux bits sont égaux, sinon, le résultat vaut 1 comme le montre la table de vérité du Tableau 3.3. Voici un exemple de code :

```
1        mov    ax, 0C123H
2        xor    ax, 0E831H          ; ax = 2912H
```

3.2.4 L'opération *NOT*

L'opération *NOT* est une opération *unaire* (*i.e.* elle agit sur une seule opérande, pas deux comme les opérations *binaires* de type *ET*). Le *NOT* d'un bit est la valeur opposée du bit comme le montre la table de vérité du Tableau 3.4. Voici un exemple de code :

```
1        mov    ax, 0C123H
2        not    ax                  ; ax = 3EDCH
```

X	NOT X
0	1
1	0

TABLE 3.4 – L'opération NOT

Allumer le bit i	Effectuer un OU sur le nombre avec 2^i (qui est le nombre binaire avec uniquement le bit i d'allumé)
Eteindre le bit i	Effectuer un ET sur le nombre avec le nombre binaire qui n'a que le bit i d'éteint. Cette opérande est souvent appelée *masque*
Inverser le bit i	Effectuer un XOR sur le nombre avec 2^i

TABLE 3.5 – Utilisation des opérations booléennes

Notez que le NOT donne le complément à un. Contrairement aux autres opérations niveau bit, l'instruction `NOT` ne change aucun des bits du registre `FLAGS`.

3.2.5 L'instruction `TEST`

L'instruction `TEST` effectue une opération AND mais ne stocke pas le résultat. Elle positionne le registre `FLAGS` selon ce que ce dernier aurait été après un AND (comme l'instruction `CMP` qui effectue une soustraction mais ne fait que positionner `FLAGS`). Par exemple, si le résultat était zéro, `ZF` serait allumé.

3.2.6 Utilisation des opérations sur les bits

Les opérations sur les bits sont très utiles pour manipuler individuellement les bits d'une donnée sans modifier les autres bits. Le Tableau 3.5 montre trois utilisations courantes de ces opérations. Voici un exemple de code, implémentant ces idées.

```
1  mov    ax, 0C123H
2  or     ax, 8           ; allumer le bit 3,   ax = C12BH
3  and    ax, 0FFDFH      ; éteindre le bit 5,  ax = C10BH
4  xor    ax, 8000H       ; inverser le bit 31,   ax = 410BH
5  or     ax, 0F00H       ; allumer un quadruplet,  ax = 4F0BH
6  and    ax, 0FFF0H      ; éteindre un quadruplet, ax = 4F00H
7  xor    ax, 0F00FH      ; inverser des quadruplets,  ax = BF0FH
8  xor    ax, 0FFFFH      ; complément à 1,  ax = 40F0H
```

L'opération *ET* peut aussi être utilisée pour trouver le reste d'une division par une puissance de deux. Pour trouver le reste d'une division par 2^i, effectuez un *ET* sur le nombre avec un masque valant $2^i - 1$. Ce masque contientra des uns du bit 0 au bit $i-1$. Ce sont tout simplement ces bits qui correspondent au reste. Le resultat du *ET* conservera ces bits et mettra les autres à zéro. Voici un extrait de code qui trouve le quotient et le reste de la division de 100 par 16.

```
1  mov    eax, 100        ; 100 = 64H
2  mov    ebx, 0000000FH  ; masque = 16 - 1 = 15 ou F
3  and    ebx, eax        ; ebx = reste = 4
```

En utilisant le registre `CL` il est possible de modifier n'importe quel(s) bit(s) d'une donnée. Voici un exemple qui allume un bit de `EAX`. Le numéro du bit à allumer est stocké dans `BH`.

```
1  mov    cl, bh          ; tout d'abord, construire le nombre pour le OU
2  mov    ebx, 1
3  shl    ebx, cl         ; décalage à gauche cl fois
4  or     eax, ebx        ; allume le bit
```

Eteindre un bit est un tout petit peut plus dur.

```
1  mov    cl, bh          ; tout d'abord, construire le nombre pour le ET
2  mov    ebx, 1
3  shl    ebx, cl         ; décalage à gauche cl fois
4  not    ebx             ; inverse les bits
5  and    eax, ebx        ; éteint le bit
```

Le code pour inverser un bit est laissé en exercice au lecteur.

Il n'est pas rare de voir l'instruction déroutante suivante dans un programme 80x86 :

```
xor    eax, eax         ; eax = 0
```

Un nombre auquel on applique un *XOR* avec lui-même donne toujours zéro. Cette instruction est utilisée car son code machine est plus petit que l'instruction `MOV` correspondante.

3.3 Eviter les Branchements Conditionnels

Les processeurs modernes utilisent des techniques très sophistiquées pour exécuter le code le plus rapidement possible. Une technique répandue est appelée *exécution spéculative*. Cette technique utilise les possibilités de traitement en parallèle du processeur pour exécuter plusieurs instructions à la

```
1       mov    bl, 0           ; bl contiendra le nombre de bits allumés
2       mov    ecx, 32         ; ecx est le compteur de boucle
3  count_loop:
4       shl    eax, 1          ; décale un bit dans le drapeau de retenue
5       adc    bl, 0           ; ajoute le drapeau de retenue à bl
6       loop   count_loop
```

FIGURE 3.3 – Compter les bits avec `ADC`

fois. Les branchements conditionnels posent un problème à ce type de fonctionnement. Le processeur, en général, ne sait pas si le branchement sera effectué ou pas. Selon qu'il est effectué ou non, un ensemble d'instructions différent sera exécuté. Les processeurs essaient de prévoir si le branchement sera effectué. Si la prévision est mauvaise, le processeur a perdu son temps en exécutant le mauvais code.

Une façon d'éviter ce problème est d'éviter d'utiliser les branchements conditionnels lorsque c'est possible. Le code d'exemple de 3.1.5 fournit un exemple simple de la façon de le faire. Dans l'exemple précédent, les bits "allumés" du registre EAX sont décomptés. Il utilise un branchement pour éviter l'instruction `INC`. La Figure 3.3 montre comment le branchement peut être retiré en utilisant l'instruction `ADC` pour ajouter directement le drapeau de retenue.

Les instructions `SET`*xx* fournissent un moyen de retirer les branchements dans certains cas. Ces instructions positionnent la valeur d'un registre ou d'un emplacement mémoire d'un octet à zéro ou à un selon l'état du registre FLAGS. Les caractères après `SET` sont les mêmes que pour les branchements conditionnels. Si la condition correspondante au `SET`*xx* est vraie, le résultat stocké est un, s'il est faux, zéro est stocké. Par exemple :

```
setz   al        ; AL = 1 si ZF est allume, sinon 0
```

En utilisant ces instructions, il est possible de développer des techniques ingénieuses qui calculent des valeurs sans branchement.

Par exemple, considérons le problème de la recherche de la plus grande de deux valeurs. L'approche standard pour résoudre ce problème serait d'utiliser un `CMP` et un branchement conditionnel pour déterminer la valeur la plus grande. Le programme exemple ci dessous montre comment le maximum peut être trouvé sans utiliser aucun branchement.

```
1  ; file: max.asm
2  %include "asm_io.inc"
```

```
3  segment .data
4
5  message1 db "Entrez un nombre : ",0
6  message2 db "Entrez un autre nombre : ", 0
7  message3 db "Le plus grand nombre est : ", 0
8
9  segment .bss
10
11 input1  resd    1        ; premier nombre entre
12
13 segment .text
14         global  _asm_main
15 _asm_main:
16         enter   0,0              ; routine d'initialisation
17         pusha
18
19         mov     eax, message1    ; affichage du premier message
20         call    print_string
21         call    read_int         ; saisie du premier nombre
22         mov     [input1], eax
23
24         mov     eax, message2    ; affichage du second message
25         call    print_string
26         call    read_int         ; saisie du second nombre (dans eax)
27
28         xor     ebx, ebx         ; ebx = 0
29         cmp     eax, [input1]    ; compare le premier et le second nombre
30         setg    bl               ; ebx = (input2 > input1) ?          1 : 0
31         neg     ebx              ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
32         mov     ecx, ebx         ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
33         and     ecx, eax         ; ecx = (input2 > input1) ?     input2 : 0
34         not     ebx              ; ebx = (input2 > input1) ?          0 : 0xFFFFFFFF
35         and     ebx, [input1]    ; ebx = (input2 > input1) ?          0 : input1
36         or      ecx, ebx         ; ecx = (input2 > input1) ?     input2 : input1
37
38         mov     eax, message3    ; affichage du résultat
39         call    print_string
40         mov     eax, ecx
41         call    print_int
42         call    print_nl
43
44         popa
```

```
45          mov     eax, 0              ; retour au C
46          leave
47          ret
```

L'astuce est de créer un masque de bits qui peut être utilisé pour sélectionner la valeur correcte pour le maximum. L'instruction `SETG` à la ligne 30 positionne BL à 1 si la seconde saisie est la plus grande ou à 0 sinon. Ce n'est pas le masque de bits désiré. Pour créer le masque nécessaire, la ligne 31 utilise l'instruction `NEG` sur le registre EBX (notez que EBX a été positionné à 0 précédemment). Si EBX vaut 0, cela ne fait rien ; cependant, si EBX vaut 1, le résultat est la représentation en complément à deux de -1 soit 0xFFFFFFFF. C'est exactement le masque de bits désiré. Le code restant utilise ce masque de bits pour sélectionner la saisie correspondant au plus grand nombre.

Une autre astuce est d'utiliser l'instruction `DEC`. Dans le code ci-dessus, si `NEG` est remplacé par `DEC`, le résultat sera également 0 ou 0xFFFFFFFF. Cependant, les valeurs sont inversées par rapport à l'utilisation de l'instruction `NEG`.

3.4 Manipuler les bits en C

3.4.1 Les opérateurs niveau bit du C

Contrairement à certains langages de haut niveau, le C fournit des opérateurs pour les opérations niveau bit. L'opération *ET* est représentée par l'opérateur binaire `&`[1]. L'opération *OU* est représentée par l'opérateur binaire `|`. L'opération *XOR* est représentée par l'opérateur binaire `^` . Et l'opération *NOT* est représentée par l'opérateur unaire `~` .

Les opérations de décalage sont effectuées au moyen des opérateurs binaires `<<` et `>>` du C. L'opérateur `<` effectue les décalages à gauche et l'opérateur `>>` effectue les décalages à droite. Ces opérateurs prennent deux opérandes. L'opérande de gauche est la valeur à décaler et l'opérande de droite est le nombre de bits à décaler. Si la valeur à décaler est d'un type non signé, un décalage logique est effectué. Si la valeur est d'un type signé (comme `int`), alors un décalage arithmétique est utilisé. Voici un exemple en C utilisant ces opérateurs :

```
1 short int s;          /* on suppose que les short int font 16 bits */
2 short unsigned u;
3 s = -1;               /* s = 0xFFFF (complément à 2) */
4 u = 100;              /* u = 0x0064 */
```

1. Ces opérateur est différent des opérateurs `&&` binaire et `&` unaire !

Macro	Signification
S_IRUSR	l'utilisateur peut lire
S_IWUSR	l'utilisateur peut écrire
S_IXUSR	l'utilisateur peut exécuter
S_IRGRP	le groupe peut lire
S_IWGRP	le groupe peut écrire
S_IXGRP	le groupe peut exécuter
S_IROTH	les autres peuvent lire
S_IWOTH	les autres peuvent écrire
S_IXOTH	les autres peuvent exécuter

TABLE 3.6 – Macros POSIX pour les Permissions de Fichiers

```
5 u = u | 0x0100;          /* u = 0x0164 */
6 s = s & 0xFFF0;          /* s = 0xFFF0 */
7 s = s ^ u;               /* s = 0xFE94 */
8 u = u << 3;              /* u = 0x0B20 (décalage logique) */
9 s = s >> 2;              /* s = 0xFFA5 (décalage arithmetique) */
```

3.4.2 Utiliser les opérateurs niveau bit en C

Les opérateurs niveau bit sont utilisés en C pour les mêmes raisons qu'ils le sont en assembleur. Ils permettent de manipuler les bits d'une donnée individuellement et peuvent être utilisés pour des multiplications et des divisions rapides. En fait, un compilateur C malin utilisera automatiquement un décalage pour une multiplication du type `x *= 2`.

Beaucoup d'API[2] de systèmes d'exploitation (comme *POSIX*[3] et Win32) contiennent des fonctions qui utilisent des opérandes donc les données sont codées sous forme de bits. Par exemple, les systèmes POSIX conservent les permissions sur les fichiers pour trois différents types d'utilisateurs : *utilisateur* (un nom plus approprié serait *propriétaire*), *groupe* et *autres*. Chaque type d'utilisateur peut recevoir la permission de lire, écrire et/ou exécuter un fichier. Pour changer les permissions d'un fichier, le programmeur C doit manipuler des bits individuels. POSIX définit plusieurs macros pour l'aider (voir Tableau 3.6). La fonction `chmod` peut être utilisée pour définir les permissions sur un fichier. Cette fonction prend deux paramètres, une chaine avec le nom du fichier à modifier et un entier[4] avec les bits appropriés d'al-

2. Application Programming Interface, Interface pour la Programmation d'Applications

3. signifie Portable Operating System Interface for Computer Environments, Interface Portable de Système d'Exploitation pour les Environnements Informatiques. Un standard développé par l'IEEE basé sur UNIX.

4. En fait, un paramètre de type `mode_t` qui est définit comme entier par un typedef.

lumés pour les permissions désirées. Par exemple, le code ci-dessous définit les permissions pour permettre au propriétaire du fichier de lire et d'écrire dedans, au groupe de lire le fichier et interdire l'accès aux autres.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

La fonction POSIX `stat` peut être utilisée pour récupérer les bits de permission en cours pour un fichier. En l'utilisant avec la fonction `chmod`, il est possible de modifier certaines des permissions sans changer les autres. Voici un exemple qui retire l'accès en écriture aux autres et ajoute les droits de lecture pour le propriétaire. Les autres permissions ne sont pas altérées.

```
1 struct stat file_stats ;     /* structure utilisee par stat() */
2 stat("foo", &file_stats );  /* lit les infos du fichier
3                                file_stats.st_mode contient les bits de permission */
4 chmod("foo", (file_stats.st_mode & ~S_IWOTH) | S_IRUSR);
```

3.5 Représentations Big et Little Endian

Le Chapitre 1 a introduit les concepts de représentations big et little endian des données multioctets. Cependant, l'auteur s'est rendu compte que pour beaucoup de gens, ce sujet est confus. Cette section couvre le sujet plus en détails.

Le lecteur se souvient sûrement que le caractère big ou little endian fait référence à l'ordre dans lequel les octets (*pas* les bits) d'un élément de données multi-octets sont stockés en mémoire. La représentation big endian est la méthode la plus intuitive. Elle stocke l'octet le plus significatif en premier, puis le second octet le plus significatif, etc. En d'autres termes, les *gros* (big) bits sont stockés en premier. La méthode little endian stocke les octets dans l'ordre inverse (moins signficatif en premier). La famille des processeurs x86 utilise la représentation little endian.

Par exemple, considérons le double mot représentant 12345678_{16}. En représentation big endian, les octets seraient stockés 12 34 56 78. En représentation little endian, les octets seraient stockés 78 56 34 12.

Le lecteur est probablement en train de se demander pourquoi n'importe quelle concepteur de puce sain d'esprit utiliserait la représentation little endian ? Les ingénieurs de chez Intel étaient-ils sadiques pour infliger à une multitude de programmeurs cette représentation qui prête à confusion ? Il peut sembler que le processeur ait à faire du travail supplémentaire pour stocker les octets en mémoire dans l'odre inverse (et pour les réinverser lorsqu'il lit à partir de la mémoire). En fait, le processeur ne fait aucun travail supplémentaire pour lire et écrire en mémoire en utilisant le format little endian. Il faut comprendre que le processeur est constitué de beaucoup de

```
unsigned short word = 0x1234; /* on suppose que sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
   printf("Machine Big Endian\n");
else
   printf("Machine Little Endian\n");
```

FIGURE 3.4 – Comment déterminer le caractère big ou little endian

circuits électroniques qui ne travaillent que sur des valeurs de un bit. Les bits (et les octets) peuvent être dans n'importe quel ordre dans le processeur.

Considérons le registre de deux octets `AX`. Il peut être décomposé en deux registres d'un octet : `AH` et `AL`. Il y a des circuits dans le processeur qui conservent les valeurs de `AH` and `AL`. Ces circuits n'ont pas d'ordre dans le processeur. C'est-à-dire que les circuits pour `AH` ne sont pas avant ou après les circuits pour `AL`. Une instruction `MOV` qui copie la valeur de `AX` en mémoire copie la valeur de `AL` puis de `AH`. Ce n'est pas plus dur pour le processeur que de stocker `AH` en premier.

Le même argument s'applique aux bits individuels d'un octet. Il ne sont pas réellement dans un ordre déterminé dans les circuits du processeur (ou en mémoire en ce qui nous concerne). Cependant, comme les bits individuels ne peuvent pas être adressés dans le processeur ou en mémoire, il n'y a pas de façon de savoir (et aucune raison de s'en soucier) l'ordre dans lequel ils sont conservés à l'intérieur du processeur.

Le code C de la Figure 3.4 montre comment le caractère big ou little endian d'un processeur peut être déterminé. Le pointeur `p` traite la variable `word` comme un tableau de caractères de deux éléments. Donc, `p[0]` correspond au premier octet de `word` en mémoire dont la valeur dépend du caractère big ou little endian du processeur.

3.5.1 Quand se Soucier du Caractère Big ou Little Endian

Pour la programmation courante, le caractère big ou little endian du processeur n'est pas important. Le moment le plus courant où cela devient important est lorsque des données binaires sont transférées entre différents systèmes informatiques. Cela se fait habituellement en utilisant un type quelconque de média physique (comme un disque) ou via un réseau. Comme les données ASCII sont sur un seul octet, le caractère big ou little endian n'est pas un problème.

Avec l'avènement des jeux de caractères multioctets comme UNICODE, le caractère big ou little endian devient important même pour les données texte. UNICODE supporte les deux types de représentation et a un mécanisme pour indiquer celle qui est utilisée pour représenter les données.

Tous les en-têtes internes de TCP/IP stockent les entiers au format big endian (appelé *ordre des octets réseau*). Les bibliothèques TCP/IP offrent

```
1  unsigned invert_endian( unsigned x )
2  {
3    unsigned invert;
4    const unsigned char * xp = (const unsigned char *) &x;
5    unsigned char * ip = (unsigned char *) & invert;
6
7    ip[0]  = xp[3];    /* inverse les octets individuels */
8    ip[1]  = xp[2];
9    ip[2]  = xp[1];
10   ip[3]  = xp[0];
11
12   return invert;     /* renvoie les octets inverses */
13 }
```

FIGURE 3.5 – Fonction invert_endian

des fonctions C pour résoudre les problèmes de représentation big ou little endian d'une façon portable. Par exemple, la fonction `htonl()` convertit un double mot (ou un entier long) depuis le format hôte vers le format réseau. La fonction function `ntohl()` effectue la transformation inverse[5]. Pour un système big endian les deux fonctions retournent leur paramètre inchangé. Cela permet d'écrire des programmes réseau qui compileront et s'exécuteront correctement sur n'importe quel système sans tenir compte de la représentation utilisée. Pour plus d'information sur les représentations big et little endian et la programmation réseau, voyez l'excellent livre de W. Richard Steven : *UNIX Network Programming.*

La Figure 3.5 montre une fonction C qui passe d'une représentation à l'autre pour un double mot. Le processeur 486 a introduit une nouvelle instruction machine appelée `BSWAP` qui inverse les octets de n'importe quel registre 32 bits. Par exemple,

```
bswap   edx          ; échange les octets de edx
```

L'instruction ne peut pas être utilisée sur des registres de 16 bits. Cependant, l'instruction `XCHG` peut être utilisée pour échanger les octets des registres 16 bits pouvant être décomposés en registres de 8 bits. Par exemple :

```
xchg    ah,al        ; échange les octets de ax
```

5. En fait, passer d'une représentation à l'autre pour entier consiste à inverser l'ordre des octets ; donc, convertir de little vers big ou de big vers little est la même opération. Donc, ces deux fonctions font la même chose.

```
1  int count_bits( unsigned int data )
2  {
3    int cnt = 0;
4
5    while( data != 0 ) {
6      data = data & (data − 1);
7      cnt++;
8    }
9    return cnt;
10 }
```

FIGURE 3.6 – Compter les Bits : Method Une

3.6 Compter les Bits

Plus haut, nous avons donné une technique intuitive pour compter le nombre de bits "allumés" dans un double mot. Cette section décrit d'autres méthodes moins directes de le faire pour illustrer les opérations sur les bits dont nous avons parlé dans ce chapitre.

3.6.1 Méthode une

La première méthode est très simple, mais pas évidente. La Figure 3.6 en montre le code.

Comment fonctionne cette méthode ? A chaque itération de la boucle, un bit est éteint dans `data`. Quand tous les bits sont éteints ((*i.e.* lorsque `data` vaut zéro), la boucle s'arrête. Le nombre d'itération requises pour mettre `data` à zéro est égal au nombre de bits dans la valeur original de `data`.

La ligne 6 est l'endroit où un bit de `data` est éteint. Comment cela marche ? Considérons la forme générale de la représentation binaire de `data` et le 1 le plus à droite dans cette représentation. Par définition, chaque bit après ce 1 est à zéro. Maintenant, que sera la représentation de `data - 1` ? Les bits à gauche du 1 le plus à droite seront les même que pour `data`, mais à partir du 1 le plus à droite, les bits seront les compléments des bits originaux de `data`. Par exemple :

`data` = xxxxx10000
`data - 1` = xxxxx01111

où les x sont les mêmes pour les deux nombres. Lorsque l'on applique un *ET* sur `data` avec `data - 1`, le résultat mettra le 1 le plus à droite de `data` à zéro et laissera les autres bits inchangés.

```
1  static unsigned char byte_bit_count[256]; /* tableau de recherche */
2
3  void initialize_count_bits ()
4  {
5    int cnt, i, data;
6
7    for( i = 0; i < 256; i++ ) {
8      cnt = 0;
9      data = i;
10     while( data != 0 ) {          /* methode une */
11       data = data & (data - 1);
12       cnt++;
13     }
14     byte_bit_count[i] = cnt;
15   }
16 }
17
18 int count_bits( unsigned int data )
19 {
20   const unsigned char * byte = ( unsigned char *) & data;
21
22   return byte_bit_count[byte[0]] + byte_bit_count[byte[1]] +
23          byte_bit_count[byte[2]] + byte_bit_count[byte[3]];
24 }
```

FIGURE 3.7 – Méthode Deux

3.6.2 Méthode deux

Un tableau de recherche peut également être utilisé pour contrer les bits de n'importe quel double mot. L'approche intuitive serait de précalculer le nombre de bits de chaque double mot et de le stocker dans un tableau. Cependant, il y a deux problèmes relatifs à cette approche. Il y a à peu près *4 milliards* de valeurs de doubles mots ! Cela signifie que le tableau serait très gros et que l'initialiser prendrait beaucoup de temps (en fait, à moins que l'on ait l'intention d'utiliser le tableau plus de 4 milliards de fois, cela prendrait plus de temps de l'initialiser que cela n'en prendrait de calculer le nombre de bits avec la méthode une !).

Une méthode plus réaliste calculerait le nombre de bits pour toutes les valeurs possibles d'octet et les stockerait dans un tableau. Le double mot peut alors être décomposé en quatre valeurs d'un octet. Le nombre de bits pour chacune de ces 4 valeurs d'un octet est recherché dans le tableau et

```
1  int count_bits(unsigned int x )
2  {
3    static unsigned int mask[] = { 0x55555555,
4                                   0x33333333,
5                                   0x0F0F0F0F,
6                                   0x00FF00FF,
7                                   0x0000FFFF };
8    int i;
9    int shift;    /* nombre de position à décaler à droite */
10
11   for( i=0, shift=1; i < 5; i++, shift *= 2 )
12     x = (x & mask[i]) + ( (x >> shift) & mask[i] );
13   return x;
14 }
```

FIGURE 3.8 – Méthode Trois

additionné aux autres pour trouver le nombre de bits du double mot original. La Figure 3.7 montre le code implémentant cette approche.

La fonction `initialize_count_bits` doit être appelée avant le premier appel à la fonction `count_bits`. Cette fonction initialise le tableau global `byte_bit_count`. La fonction `count_bits` ne considère pas la variable `data` comme un double mot mais comme un tableau de quatre octets. Donc, `dword[0]` est un des octets de `data` (soit le moins significatif, soit le plus significatif selon que le matériel est little ou big endian, respectivement). Bien sûr, on peut utiliser une instruction comme :

```
(data >> 24) & 0x000000FF
```

pour trouver la valeur de l'octet le plus significatif et des opérations similaires pour les autres octets ; cependant, ces opérations seraient plus lentes qu'une référence à un tableau.

Un dernier point, une boucle `for` pourrait facilement être utilisée pour calculer la somme des lignes 22 et 23. Mais, cela inclurait le supplément de l'initialisation de l'indice, sa comparaison après chaque itération et son incrémentation. Calculer la somme comme une somme explicite de quatre valeurs est plus rapide. En fait, un compilateur intelligent convertirait la version avec une boucle `for` en une somme explicite. Ce procédé de réduire ou éliminer les itérations d'une boucle est une technique d'optimisation de compilateur appelée *loop unrolling* (déroulage de boucle).

3.6.3 Méthode trois

Il y a encore une méthode intelligente de compter les bits allumés d'une donnée. Cette méthode ajoute littéralement les uns et les zéros de la donnée. La somme est égale au nombre de 1 dans la donnée. Par exemple, considérons le comptage des uns d'un octet stocké dans une variable nommée `data`. La première étape est d'effectuer l'opération suivante :

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

Pourquoi faire cela ? La constante hexa `0x55` vaut 01010101 en binaire. Dans la première opérande de l'addition, on effectue un *ET* sur `data` avec elle, les bits sur des positions impaires sont supprimés. La seconde opérande `((data >> 1) & 0x55)` commence par déplacer tous les bits à des positions paires vers une position impaire et utilise le même masque pour supprimer ces bits. Maintenant, la première opérande contient les bits impairs et la seconde les bits pairs de `data`. Lorsque ces deux opérandes sont additionnées, les bits pairs et les bits impairs de `data` sont additionnés. Par exemple, si `data` vaut 10110011_2, alors :

`data &` 01010101_2		00	01	00	01
+ `(data >> 1) &` 01010101_2	soit +	01	01	00	01
		01	10	00	10

L'addition à droite montre les bits additionnés ensembles. Les bits de l'octet sont divisés en deux champs de 2 bits pour montrer qu'il y a en fait quatre additions indépendantes. Comme la plus grande valeur que ces sommes peuvent prendre est deux, il n'est pas possibile qu'une somme déborde de son champ et corrompe l'une des autres sommes.

Bien sûr, le nombre total de bits n'a pas encore été calculé. Cependant, la même technique que celle qui a été utilisée ci-dessus peut être utilisée pour calculer le total en une série d'étapes similaires. L'étape suivante serait :

```
data = (data & 0x33) + ((data >> 2) & 0x33);
```

En continuant l'exemple du dessus (souvenez vous que `data` vaut maintenant 01100010_2):

`data &` 00110011_2		0010	0010
+ `(data >> 2) &` 00110011_2	soit +	0001	0000
		0011	0010

Il y a maintenant deux champs de 4 bits additionnés individuellement.

La prochaine étape est d'additionner ces deux sommes de bits ensemble pour former le résultat final :

```
data = (data & 0x0F) + ((data >> 4) & 0x0F);
```

En utilisant l'exemple ci-dessus (avec `data` égale à 00110010_2):

$$\begin{array}{r} \texttt{data \& } 00001111_2 \\ +\ \texttt{(data >> 4) \& } 00001111_2 \\ \hline \end{array} \quad \text{soit} \quad \begin{array}{r} 00000010 \\ +\ \ 00000011 \\ \hline 00000101 \end{array}$$

Maintenant, `data` vaut 5 ce qui est le résultat correct. La Figure 3.8 montre une implémentation de cette méthode qui compte les bits dans un double mot. Elle utilise une boucle `for` pour calculer la somme. Il serait plus rapide de dérouler la boucle ; cependant, la boucle rend plus claire la façon dont la méthode se généralise à différentes tailles de données.

Chapitre 4

Sous-Programmes

Ce chapitre explique comment utiliser des sous-programmes pour créer des programmes modulaires et s'interfacer avec des langages de haut niveau (comme le C). Les fonctions et les procédures sont des exemples de sous-programmes dans les langages de haut niveau.

Le code qui appelle le sous-programme et le sous-programme lui-même doivent se mettre d'accord sur la façon de se passer les données. Ces règles sur la façon de passer les données sont appelées *conventions d'appel*. Une grande partie de ce chapitre traitera des conventions d'appel standards du C qui peuvent être utilisées pour interfacer des sous-programmes assembleur avec des programmes C. Celle-ci (et d'autres conventions) passe souvent les adresses des données (*i.e.* des pointeurs) pour permettre au sou-programme d'accéder aux données en mémoire.

4.1 Adressage Indirect

L'adressage indirect permet aux registres de se comporter comme des pointeurs. Pour indiquer qu'un registre est utilisé indirectement comme un pointeur, il est entouré par des crochets (`[]`). Par exemple :

```
1  mov    ax, [Data]      ; adressage mémoire direct normal d'un mot
2  mov    ebx, Data       ; ebx = & Data
3  mov    ax, [ebx]       ; ax = *ebx
```

Comme AX contient un mot, la ligne 3 lit un mot commençant à l'adresse stockée dans EBX. Si AX était remplacé par AL, un seul octet serait lu. Il est important de réaliser que les registres n'ont pas de types comme les variable en C. Ce sur quoi EBX est censé pointer est totalement déterminé par les instructions utilisées. Si EBX est utilisé de manière incorrecte, il n'y aura souvent pas d'erreur signalée de la part de l'assembleur ; cependant, le programme ne fonctionnera pas correctement. C'est une des nombreuses

raisons pour lesquelles la programmation assembleur est plus sujette à erreur que la programmation de haut niveau.

Tous les registres 32 bits généraux (EAX, EBX, ECX, EDX) et d'index (ESI, EDI) peuvent être utilisés pour l'adressage indirect. En général, les registres 16 et 8 bits ne le peuvent pas.

4.2 Exemple de Sous-Programme Simple

Un sous-programme est une unité de code indépendante qui peut être utilisée depuis différentes parties du programme. En d'autres termes, un sous-programme est comme une fonction en C. Un saut peut être utilisé pour appeler le sous-programme, mais le retour présente un problème. Si le sous-programme est destiné à être utilisé par différentes parties du programme, il doit revenir à la section de code qui l'a appelé. Donc, le retour du sous-programme ne peut pas être codé en dur par un saut vers une étiquette. Le code ci-dessous montre comment cela peut être réalisé en utilisant une forme indirecte de l'instruction `JMP`. Cette forme de l'instruction utilise la valeur d'un registre pour déterminer où sauter (donc, le registre agit plus comme un *pointeur de fonction* du C). Voici le premier programme du Chapitre 1 réécrit pour utiliser un sous-programme.

sub1.asm

```
1  ; fichier : sub1.asm
2  ; Programme d'exemple de sous-programme
3  %include "asm_io.inc"
4
5  segment .data
6  prompt1 db    "Entrez un nombre : ", 0       ; ne pas oublier le zéro terminal
7  prompt2 db    "Entrez un autre nombre : ", 0
8  outmsg1 db    "Vous avez entré ", 0
9  outmsg2 db    " et ", 0
10 outmsg3 db    ", la somme des deux vaut ", 0
11
12 segment .bss
13 input1  resd 1
14 input2  resd 1
15
16 segment .text
17         global  _asm_main
18 _asm_main:
19         enter   0,0               ; routine d'initialisation
20         pusha
21
```

```
22          mov     eax, prompt1      ; affiche l'invite
23          call    print_string
24
25          mov     ebx, input1       ; stocke l'adresse de input1 dans ebx
26          mov     ecx, ret1         ; stocke l'adresse de retour dans ecx
27          jmp     short get_int     ; lit un entier
28  ret1:
29          mov     eax, prompt2      ; affiche l'invite
30          call    print_string
31
32          mov     ebx, input2
33          mov     ecx, $ + 7        ; ecx = cette addresse + 7
34          jmp     short get_int
35
36          mov     eax, [input1]     ; eax = dword dans input1
37          add     eax, [input2]     ; eax += dword dans input2
38          mov     ebx, eax          ; ebx = eax
39
40          mov     eax, outmsg1
41          call    print_string      ; affiche le premier message
42          mov     eax, [input1]
43          call    print_int         ; affiche input1
44          mov     eax, outmsg2
45          call    print_string      ; affiche le second message
46          mov     eax, [input2]
47          call    print_int         ; affiche input2
48          mov     eax, outmsg3
49          call    print_string      ; affiche le troisième message
50          mov     eax, ebx
51          call    print_int         ; affiche la somme (ebx)
52          call    print_nl          ; retour à la ligne
53
54          popa
55          mov     eax, 0            ; retour au C
56          leave
57          ret
58  ; subprogram get_int
59  ; Paramètres:
60  ;   ebx - addresse du dword dans lequel stocker l'entier
61  ;   ecx - addresse de l'instruction vers laquelle retourner
62  ; Notes:
63  ;   la valeur de eax est perdue
```

```
64 get_int:
65         call    read_int
66         mov     [ebx], eax          ; stocke la saisie en mémoire
67         jmp     ecx                 ; retour à l'appelant
```

sub1.asm

Le sous-programme `get_int` utilise une convention d'appel simple, basée sur les registres. Il s'attend à ce que le registre EBX contienne l'adresse du DWORD dans lequel stocker le nombre saisi et à ce que le registre ECX contiennent l'adresse de l'instruction vers laquelle retourner. Dans les lignes 25 à 28, l'étiquette `ret1` est utilisée pour calculer cette adresse de retour. Dans les lignes 32 à 34, l'opérateur `$` est utilisé pour calculer l'adresse de retour. L'opérateur `$` retourne l'adresse de la ligne sur laquelle il apparaît. L'expression `$ + 7` calcule l'adresse de l'instruction `MOV` de la ligne 36.

Ces deux calculs d'adresses de code de retour sont compliqués. La première méthode requiert la définition d'une étiquette pour tout appel de sous-programme. La seconde méthode ne requiert pas d'étiquette mais nécessite de réfléchir attentivement. Si un saut proche avait été utilisé à la place d'un saut court, le nombre à ajouter à `$` n'aurait pas été 7! Heureusement, il y a une façon plus simple d'appeler des sous-programmes. Cette méthode utilise la *pile*.

4.3 La pile

Beaucoup de processeurs ont un support intégré pour une pile. Une pile est une liste Last-In First-Out (*LIFO*, dernier entré, premier sorti). La pile est une zone de la mémoire qui est organisée de cette façon. L'instruction `PUSH` ajoute des données à la pile et l'instruction `POP` retire une donnée. La donnée retirée est toujours la dernière donnée ajoutée (c'est pourquoi on appelle ce genre de liste dernier entré, premier sorti).

Le registre de segment SS spécifie le segment qui contient la pile (habituellement c'est le même que celui qui contient les données). Le registre ESP contient l'adresse de la donnée qui sera retirée de la pile. On dit que cette donnée est au *sommet* de la pile. Les données ne peuvent être ajoutées que par unités de double mots. C'est-à-dire qu'il est impossible de placer un octet seul sur la pile.

L'instruction `PUSH` insère un double mot [1] sur la pile en ôtant 4 de ESP puis en stockant le double mot en `[ESP]`. L'instruction `POP` lit le double mot en `[ESP]` puis ajoute 4 à ESP. Le code ci-dessous montre comment fonctionnent ces instructions en supposant que ESP vaut initialement `1000H`.

1. En fait, il est également possible d'empiler des mots, mais en mode protégé 32 bits, il est mieux de ne travailler qu'avec des doubles mots sur la pile.

```
1       push   dword 1     ; 1 est stocké en 0FFCh, ESP = 0FFCh
2       push   dword 2     ; 2 est stocké en 0FF8h, ESP = 0FF8h
3       push   dword 3     ; 3 est stocké en 0FF4h, ESP = 0FF4h
4       pop    eax         ; EAX = 3, ESP = 0FF8h
5       pop    ebx         ; EBX = 2, ESP = 0FFCh
6       pop    ecx         ; ECX = 1, ESP = 1000h
```

La pile peut être utilisée comme un endroit approprié pour stocker des données temporairement. Elle est également utilisée pour effectuer des appels de sous-programmes, passer des paramètres et des variables locales.

Le 80x86 fournit également une instruction, `PUSHA`, qui empile les valeurs des registres EAX, EBX, ECX, EDX, ESI, EDI et EBP (pas dans cet ordre). L'instruction `POPA` peut être utilisée pour les dépiler tous.

4.4 Les Instructions CALL et RET

Le 80x86 fournit deux instructions qui utilisent la pile pour effectuer des appels de sous-programmes rapidement et facilement. L'instruction CALL effectue un saut inconditionnel vers un sous-programme et *empile* l'adresse de l'instruction suivante. L'instruction RET *dépile* une adresse et saute à cette adresse. Lors de l'utilisation de ces instructions, il est très important de gérer la pile correctement afin que le chiffre correct soit dépilé par l'instruction RET !

Le programme précédent peut être réécrit pour utiliser ces nouvelles instructions en changeant les lignes 25 à 34 en ce qui suit :

```
      mov    ebx, input1
      call   get_int

      mov    ebx, input2
      call   get_int
```

et en changeant le sous-programme `get_int` en :

```
get_int:
      call   read_int
      mov    [ebx], eax
      ret
```

Il y a plusieurs avantages à utiliser CALL et RET:

— C'est plus simple !

— Cela permet d'imbriquer des appels de sous-programmes facilement. Notez que `get_int` appelle `read_int`. Cet appel empile une autre adresse. à la fin du code de `read_int` se trouve un RET qui dépile l'adresse de retour et saute vers le code de `get_int`. Puis, lorsque le RET de `get_int` est exécuté, il dépile l'adresse de retour qui revient vers `asm_main`. Cela fonctionne correctement car il s'agit d'une pile LIFO.

Souvenez vous, il est *très* important de dépiler toute donnée qui est empilée. Par exemple, considérons le code suivant :

```
1 get_int:
2       call   read_int
3       mov    [ebx], eax
4       push   eax
5       ret          ; dépile la valeur de EAX, pas l'adresse de retour !!
```

Ce code ne reviendra pas correctement !

4.5 Conventions d'Appel

Lorsqu'un sous-programme est appelé, le code appelant et le sous-programme (l'*appelé*) doivent s'accorder sur la façon de se passer les données. Les langages de haut niveau ont des manières standards de passer les données appelées *conventions d'appel*. Pour interfacer du code de haut niveau avec le langage assembleur, le code assembleur doit utiliser les mêmes conventions que le langage de haut niveau. Les conventions d'appel peuvent différer d'un compilateur à l'autre ou peuvent varier selon la façon dont le code est compilé (*p.e.* selon que les optimisations sont activées ou pas). Une convention universelle est que le code est appelé par une instruction `CALL` et revient par un `RET`.

Tous les compilateurs C PC supportent une convention d'appel qui sera décrite dans le reste de ce chapitre par étape. Ces conventions permettent de créer des sous-programmes *réentrants*. Un sous-programme réentrant peut être appelé depuis n'importe quel endroit du programme en toute sécurité (même depuis le sous-programme lui-même).

4.5.1 Passer les paramètres via la pile

Les paramètres d'un sous-programme peuvent être passés par la pile. Ils sont empilés avant l'instruction `CALL`. Comme en C, si le paramètre doit être modifié par le sous-programme, l'*adresse* de la donnée doit être passée, pas sa *valeur*. Si la taille du paramètre est inférieure à un double mot, il doit être converti en un double mot avant d'être empilé.

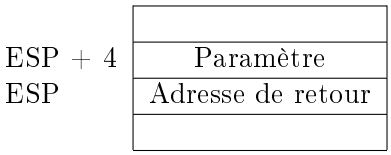

FIGURE 4.1 – état de la pile lors du passage d'un paramètre

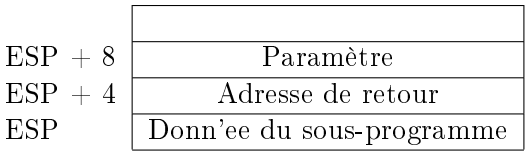

FIGURE 4.2 – Etat de la pile après empilage d'une donnée

Les paramètres sur la pile ne sont pas dépilés par le sous-programme, à la place, ils sont accédés depuis la pile elle-même. Pourquoi ?

— Comme ils doivent être empilés avant l'instruction `CALL`, l'adresse de retour devrait être dépilée avant tout (puis réempilée ensuite).
— Souvent, les paramètres sont utilisés à plusieurs endroits dans le sous-programme. Habituellement, ils ne peuvent pas être conservés dans un registre durant toute la durée du sous-programme et devront être stockés en mémoire. Les laisser sur la pile conserve une copie de la donnée en mémoire qui peut être accédée depuis n'importe quel endroit du sous-programme.

Considérons un sous-programme auquel on passe un paramètre unique via la pile. Lorsque le sous-programme est appelé, la pile ressemble à la Figure 4.1. On peut accéder au paramètre en utilisant l'adressage indirect (`[ESP+4]` [2]).

Lors de l'utilisation de l'adressage indirect, le processeur 80x86 accède à différents segments selon les registres utilisés dans l'expression d'adressage indirect. ESP (et EBP) utilisent le segment de pile alors que EAX, EBX, ECX et EDX utilisent le segment de données. Cependant, ce n'est habituellement pas important pour la plupart des programmes en mode protégé, car pour eux, les segments de pile et de données sont les mêmes.

Si la pile est également utilisée dans le sous-programme pour stocker des données, le nombre à ajouter à ESP changera. Par exemple, la Figure 4.2 montre à quoi ressemble la pile si un DWORD est empilé. Maintenant, le paramètre se trouve en `ESP + 8`, plus en `ESP + 4`. Donc, utiliser ESP lorsque l'on fait référence à des paramètres peut être une source d'erreurs. Pour résoudre ce problème, Le 80386 fournit un autre registre à utiliser : EBP. La seule utilité de ce registre est de faire référence à des données sur la pile. La convention d'appel C stipule qu'un sous-programme doit d'abord empiler la valeur de EBP puis définir EBP pour qu'il soit égal à ESP. Cela permet à ESP de changer au fur et à mesure que des données sont empilées ou dépilées sans modifier EBP. A la fin du programme, la valeur originale de EBP doit être restaurée (c'est pourquoi elle est sauvegardée au début du sous-programme).

2. Il est autorisé d'ajouter une constante à un registre lors de l'utilisation de l'adressage indirect. Des expressions plus complexes sont également possible. Ce sujet est traité dans le chapitre suivant

```
1 etiquette_sousprogramme:
2       push   ebp          ; empile la valeur originale de EBP
3       mov    ebp, esp     ; EBP = ESP
4 ; code du sousprogramme
5       pop    ebp          ; restaure l'ancienne valeur de EBP
6       ret
```

FIGURE 4.3 – Forme générale d'un sous-programme

ESP + 8	EBP + 8	Parametre
ESP + 4	EBP + 4	Adresse de retour
ESP	EBP	EBP sauvegardé

FIGURE 4.4 – Etat de la pile en suivant la convention d'appel C

La Figure 4.3 montre la forme générale d'un sous-programme qui suit ces conventions.

Les lignes 2 et 3 de la Figure 4.3 constituent le *prologue* générique d'un sous-programme. Les lignes 5 et 6 consituent l'*épilogue*. La Figure 4.4 montre à quoi ressemble la pile immédiatement après le prologue. Maintenant, le paramètre peut être accédé avec `[EBP + 8]` depuis n'importe quel endroit du programme sans se soucier de ce qui a été empilé entre temps par le sous-programme.

Une fois le sous-programme terminé, les paramètres qui ont été empilés doivent être retirés. La convention d'appel C spécifie que c'est au code appelant de le faire. D'autres conventions sont différentes. Par exemple, la convention d'appel Pascal spécifie que c'est au sous-programme de retirer les paramètres (Il y une autre forme de l'instruction RET qui permet de le faire facilement). Quelques compilateurs C supportent cette convention également. Le mot clé `pascal` est utilisé dans le prototype et la définition de la fonction pour indiquer au compilateur d'utiliser cette convention. En fait, la convention `stdcall`, que les fonctions de l'API C MS Windows utilisent, fonctionne également de cette façon. Quel est son avantage ? Elle est un petit peu plus efficace que la convention C. Pourquoi toutes les fonctions C n'utilisent pas cette convention alors ? En général, le C autorise une fonction a avoir un nombre variable d'arguments (*p.e.*, les fonctions `printf` et `scanf`). Pour ce type de fonction, l'opération consistant à retirer les paramètres de la pile varie d'un appel à l'autre. La convention C permet aux instructions nécessaires à la réalisation de cette opération de varier facilement d'un appel à l'autre. Les conventions Pascal et stdcall rendent cette opération très compli-

```
1       push   dword 1        ; passe 1 en paramètre
2       call   fun
3       add    esp, 4         ; retire le paramètre de la pile
```

FIGURE 4.5 – Exemple d'appel de sous-programme

quée. Donc, la convention Pascal (comme le langage Pascal) n'autorise pas ce type de fonction. MS Windows peut utiliser cette convention puisqu'aucune de ses fonctions d'API ne prend un nombre variable d'arguments.

La Figure 4.5 montre comment un sous-programme utilisant la convention d'appel C serait appelé. La ligne 3 retire le paramètre de la pile en manipulant directement le pointeur de pile. Une instruction `POP` pourrait également être utilisée, mais cela nécessiterait le stockage d'un paramètre inutile dans un registre. En fait, dans ce cas particulier, beaucoup de compilateurs utilisent une instruction `POP ECX` pour retirer le paramètre. Le compilateur utilise un `POP` plutot qu'un `ADD` car le `ADD` nécessite plus d'octets pour stocker l'instruction. Cependant, le `POP` change également la valeur de ECX ! Voici un autre programme exemple avec deux sous-programmes qui utilisent les conventions d'appel C dont nous venons de parler. La ligne 54 (et les autres) montre que plusieurs segments de données et de texte peuvent être déclarés dans un même fichier source. Ils seront combinés en des segments de données et de texte uniques lors de l'édition de liens. Diviser les données et le code en segments séparés permet d'avoir les données d'un sous-programme définies à proximité de celui-ci.

sub3.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  ;
10 ; algorithme en pseudo-code
11 ; i = 1;
12 ; sum = 0;
13 ; while( get_int(i, &input), input != 0 ) {
14 ;   sum += input;
15 ;   i++;
```

```
16 ; }
17 ; print_sum(num);
18 segment .text
19         global  _asm_main
20 _asm_main:
21         enter   0,0               ; routine d'initialisation
22         pusha
23
24         mov     edx, 1            ; edx est le 'i' du pseudo-code
25 while_loop:
26         push    edx               ; empile i
27         push    dword input       ; empile l'adresse de input
28         call    get_int
29         add     esp, 8            ; dépile i et &input
30
31         mov     eax, [input]
32         cmp     eax, 0
33         je      end_while
34
35         add     [sum], eax        ; sum += input
36
37         inc     edx
38         jmp     short while_loop
39
40 end_while:
41         push    dword [sum]       ; empile la valeur de sum
42         call    print_sum
43         pop     ecx               ; dépile [sum]
44
45         popa
46         leave
47         ret
48
49 ; sous-programme get_int
50 ; Paramètres (dans l'ordre de l'empilement)
51 ;   nombre de saisies (en [ebp + 12])
52 ;   adresse du mot où stocker la saisie (en [ebp + 8])
53 ; Notes:
54 ;   les valeurs de eax et ebx sont détruites
55 segment .data
56 prompt  db      ") Entrez un nombre entier (0 pour quitter): ", 0
57
```

```
58 segment .text
59 get_int:
60         push    ebp
61         mov     ebp, esp
62
63         mov     eax, [ebp + 12]
64         call    print_int
65
66         mov     eax, prompt
67         call    print_string
68
69         call    read_int
70         mov     ebx, [ebp + 8]
71         mov     [ebx], eax          ; stocke la saisie en mémoire
72
73         pop     ebp
74         ret                         ; retour à l'appelant
75
76 ; sous-programme print_sum
77 ; affiche la somme
78 ; Paramètre:
79 ;   somme à afficher (en [ebp+8])
80 ; Note: détruit la valeur de eax
81 ;
82 segment .data
83 result  db      "La somme vaut ", 0
84
85 segment .text
86 print_sum:
87         push    ebp
88         mov     ebp, esp
89
90         mov     eax, result
91         call    print_string
92
93         mov     eax, [ebp+8]
94         call    print_int
95         call    print_nl
96
97         pop     ebp
98         ret
```

sub3.asm

```
1  etiquette_sousprogramme:
2        push   ebp                ; empile la valeur originale de EBP
3        mov    ebp, esp           ; EBP = ESP
4        sub    esp, LOCAL_BYTES   ; = nb octets nécessaires pour les locales
5  ; code du sousprogramme
6        mov    esp, ebp           ; désalloue les locales
7        pop    ebp                ; restaure la valeur originalede EBP
8        ret
```

FIGURE 4.6 – Forme générale d'un sous-programme avec des variables locales

```
1  void calc_sum( int n, int * sump )
2  {
3    int i, sum = 0;
4
5    for( i=1; i <= n; i++ )
6      sum += i;
7    *sump = sum;
8  }
```

FIGURE 4.7 – Version C de sum

4.5.2 Variables locales sur la pile

La pile peut être utilisée comme un endroit pratique pour stocker des variables locales. C'est exactement ce que fait le C pour les variables normales (ou *automatiques* en C lingo). Utiliser la pile pour les variables est important si l'on veut que les sous-programmes soient réentrants. Un programme réentrant fonctionnera qu'il soit appelé de n'importe quel endroit, même à partir du sous-programme lui-même. En d'autres termes, les sous-programmes réentrants peuvent être appelés *récursivement*. Utiliser la pile pour les variables économise également de la mémoire. Les données qui ne sont pas stockées sur la pile utilisent de la mémoire du début à la fin du programme (le C appelle ce type de variables *global* ou *static*). Les données stockées sur la pile n'utilisent de la mémoire que lorsque le sous-programme dans lequel elles sont définies est actif.

Les variables locales sont stockées immédiatement après la valeur de EBP sauvegardée dans la pile. Elles sont allouées en soustrayant le nombre d'octets requis de ESP dans le prologue du sous-programme. La Figure 4.6 montre le nouveau squelette du sous-programme. Le registre EBP est utilisé pour accéder à des variables locales. Considérons la fonction C de la Figure 4.7. La Figure 4.8 montre comment le sous-programme équivalent pourrait être

```
1  cal_sum:
2        push   ebp
3        mov    ebp, esp
4        sub    esp, 4           ; fait de la place pour le sum local
5
6        mov    dword [ebp - 4], 0   ; sum = 0
7        mov    ebx, 1               ; ebx (i) = 1
8  for_loop:
9        cmp    ebx, [ebp+12]        ; i <= n ?
10       jnle   end_for
11
12       add    [ebp-4], ebx         ; sum += i
13       inc    ebx
14       jmp    short for_loop
15
16 end_for:
17       mov    ebx, [ebp+8]         ; ebx = sump
18       mov    eax, [ebp-4]         ; eax = sum
19       mov    [ebx], eax           ; *sump = sum;
20
21       mov    esp, ebp
22       pop    ebp
23       ret
```

FIGURE 4.8 – Version assembleur de sum

écrit en assembleur.

La Figure 4.9 montre à quoi ressemble la pile après le prologue du programme de la Figure 4.8. Cette section de la pile qui contient les paramètres, les informations de retour et les variables locales est appelée *cadre de pile* (stack frame). Chaque appel de fonction C crée un nouveau cadre de pile sur la pile.

Le prologue et l'épilogue d'un sous-programme peuvent être simplifiés en utilisant deux instructions spéciales qui sont conçues spécialement dans ce but. L'instruction `ENTER` effectue le prologue et l'instruction `LEAVE` l'épilogue. L'instruction `ENTER` prend deux opérandes immédiates. Dans la convention d'appel C, la deuxième opérande est toujours 0. La première opérande est le nombre d'octets nécessaires pour les variables locales. L'instruction `LEAVE` n'a pas d'opérande. La Figure 4.10 montre comment ces instructions sont utilisées. Notez que le squelette de programme (Figure 1.7) utilise également `ENTER` et `LEAVE`.

En dépit du fait que `ENTER` *et* `LEAVE` *simplifient le prologue et l'épilogue, ils ne sont pas utilisés très souvent. Pourquoi ? Parce qu'ils sont plus lent que les instructions plus simples équivalentes ! C'est un des exemples où il ne faut pas supposer qu'une instruction est plus rapide qu'une séquence de plusieurs instructions.*

ESP + 16	EBP + 12	`n`
ESP + 12	EBP + 8	`sump`
ESP + 8	EBP + 4	Adresse de retour
ESP + 4	EBP	EBP sauvé
ESP	EBP - 4	`sum`

FIGURE 4.9 – état de la pile après le prologue de sum

```
1  etiquette_sousprogramme:
2        enter  LOCAL_BYTES, 0     ; = nb octets pour les locales
3  ; code du sous-programme
4        leave
5        ret
```

FIGURE 4.10 – Forme générale d'un sous-programme avec des variables locales utilisant `ENTER` et `LEAVE`

4.6 Programme Multi-Modules

Un *programme multi-modules* est un programme composé de plus d'un fichier objet. Tous les programmes présentés jusqu'ici sont des programmes multi-modules. Il consistent en un fichier objet C pilote et le fichier objet assembleur (plus les fichiers objet de la bibliothèque C). Souvenez vous que l'éditeur de liens combine les fichier objets en un programme exécutable unique. L'éditeur de liens doit rapprocher toutes les références faites à chaque étiquette d'un module (*i.e.* un fichier objet) de sa définition dans un autre module. Afin que le module A puisse utiliser une étiquette définie dans le module B, la directive `extern` doit être utilisée. Après la directive `extern` vient une liste d'étiquettes délimitées par des virgules. La directive indique à l'assembleur de traiter ces étiquettes comme *externes* au module. C'est-à-dire qu'il s'agit d'étiquettes qui peuvent être utilisées dans ce module mais sont définies dans un autre. Le fichier `asm_io.inc` définit les routines `read_int`, *etc.* comme externes.

En assembleur, les étiquettes ne peuvent pas être accédées de l'extérieur par défaut. Si une étiquette doit pouvoir être accédée depuis d'autres modules que celui dans lequel elle est définie, elle doit être déclarée comme *globale* dans son module, par le biais de la directive `global` . La ligne 13 du listing du programme squelette de la Figure 1.7 montre que l'étiquette `_asm_main` est définie comme globale. Sans cette déclaration, l'éditeur de liens indiquerait une erreur. Pourquoi ? Parce que le code C ne pourrait pas faire référence à l'étiquette *interne* `_asm_main`.

Voici le code de l'exemple précédent réécrit afin d'utiliser deux modules. Les deux sous-programmes (`get_int` et `print_sum`) sont dans des fichiers sources distincts de celui de la routine `_asm_main`.

main4.asm

```
%include "asm_io.inc"

segment .data
sum     dd   0

segment .bss
input   resd 1

segment .text
        global  _asm_main
        extern  get_int, print_sum
_asm_main:
        enter   0,0               ; routine d'initialisation
        pusha

        mov     edx, 1            ; edx est le 'i' du pseudo-code
while_loop:
        push    edx               ; empile i
        push    dword input       ; empile l'adresse de input
        call    get_int
        add     esp, 8            ; dépile i et &input

        mov     eax, [input]
        cmp     eax, 0
        je      end_while

        add     [sum], eax        ; sum += input

        inc     edx
        jmp     short while_loop

end_while:
        push    dword [sum]       ; empile la valeur de sum
        call    print_sum
        pop     ecx               ; dépile [sum]

        popa
        leave
```

```
39          ret
```
main4.asm

sub4.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   prompt  db      ") Entrez un nombre entier (0 pour quitter): ", 0
5
6   segment .text
7           global  get_int, print_sum
8   get_int:
9           enter   0,0
10
11          mov     eax, [ebp + 12]
12          call    print_int
13
14          mov     eax, prompt
15          call    print_string
16
17          call    read_int
18          mov     ebx, [ebp + 8]
19          mov     [ebx], eax          ; stocke la saisie en mémoire
20
21          leave
22          ret                         ; retour à l'appelant
23
24  segment .data
25  result  db      "La somme vaut ", 0
26
27  segment .text
28  print_sum:
29          enter   0,0
30
31          mov     eax, result
32          call    print_string
33
34          mov     eax, [ebp+8]
35          call    print_int
36          call    print_nl
37
38          leave
39          ret
```
sub4.asm

L'exemple ci-dessus n'a que des étiquettes de code globales ; cependant, les étiquettes de donnée globales fonctionnent exactement de la même façon.

4.7 Interfacer de l'assembleur avec du C

Aujourd'hui, très peu de programmes sont écrits complètement en assembleur. Les compilateurs sont très performants dans la conversion de code de haut niveau en code machine efficace. Comme il est plus facile d'écrire du code dans un langage de haut niveau, ils sont plus populaires. De plus, le code de haut niveau est *beaucoup* plus portable que l'assembleur !

Lorsque de l'assembleur est utilisé, c'est souvent pour de petites parties du code. Cela peut être fait de deux façons : en appelant des sous-routines assembleur depuis le C ou en incluant de l'assembleur. Inclure de l'assembleur permet au programmeur de placer des instructions assembleur directement dans le code C. Cela peut être très pratique ; cependant, il y a des inconvénients à inclure l'assembleur. Le code assembleur doit être écrit dans le format que le compilateur utilise. Aucun compilateur pour le moment ne supporte le format NASM. Des compilateurs différents demandent des formats différents. Borland et Microsoft demandent le format MASM. DJGPP et gcc sous Linux demandent le format GAS[3]. La technique d'appel d'une sous-routine assembleur est beaucoup plus standardisée sur les PC.

Les routines assembleur sont habituellement utilisées avec le C pour les raisons suivantes :

— Un accès direct au fonctionnalités matérielles de l'ordinateur est nécessaire car il est difficle ou impossible d'y accéder en C.
— La routine doit être la plus rapide possible et le programmeur peut optimiser le code à la main mieux que le compilateur.

La dernière raison n'est plus aussi valide qu'elle l'était. La technologie des compilateurs a été améliorée au fil des ans et les compilateurs génèrent souvent un code très performant (en particulier si les optimisations activées). Les inconvénients des routines assembleur sont une portabilité et une lisibilité réduites.

La plus grande partie des conventions d'appel C a déjà été présentée. Cependant, il y a quelques fonctionnalités supplémentaires qui doivent être décrites.

3. GAS est l'assembleur GNU que tous les compilateurs GNU utilisent. Il utilise la syntaxe AT&T qui est très différentes des syntaxes relativement similaires de MASM, TASM et NASM.

```
1  segment .data
2  x            dd     0
3  format       db     "x = %d\n", 0
4
5  segment .text
6  ...
7        push   dword [x]     ; empile la valeur de x
8        push   dword format  ; empile l'adresse de la chaîne format
9        call   _printf       ; notez l'underscore!
10       add    esp, 8        ; dépile les paramètres
```

FIGURE 4.11 – Appel de `printf`

EBP + 12	Valeur de `x`
EBP + 8	Adresse de la chaine format
EBP + 4	Adresse de retour
EBP	Sauvegarde de EBP

FIGURE 4.12 – Pile à l'intérieur de `printf`

4.7.1 Sauvegarder les registres

Le mot clé `register` *peut être utilisé dans une déclaration de variable C pour suggérer au compilateur d'utiliser un registre pour cette variable plutot qu'un emplacement mémoire. On appelle ces variables, variables de registre. Les compilateurs modernes le font automatiquement sans qu'il y ait besoin d'une suggestion.*

Tout d'abord, Le C suppose qu'une sous-routine maintient les valeurs des registres suivants : EBX, ESI, EDI, EBP, CS, DS, SS, ES. Cela ne signifie pas que la sous-routine ne les change pas en interne. Cela signifie que si elle change leur valeurs, elle doit les restaurer avant de revenir. Les valeurs de EBX, ESI et EDI doivent être inchangées car le C utilise ces registres pour les *variables de registre*. Habituellement, la pile est utilisée pour sauvegarder les valeurs originales de ces registres.

4.7.2 Etiquettes de fonctions

La plupart des compilateurs C ajoutent un caractère underscore(_) au début des noms des fonctions et des variables global/static. Par exemple, à une fonction appelée `f` sera assignée l'étiquette `_f`. Donc, s'il s'agit d'une routine assembleur, elle *doit* être étiquetée `_f`, pas `f`. Le compilateur Linux gcc, n'ajoute *aucun* caractère. Dans un executable Linux ELF, on utiliserait simplement l'étiquette `f` pour la fonction `f`. Cependant, le gcc de DJGPP ajoute un underscore. Notez que dans le squelette de programme assembleur (Figure 1.7), l'étiquette de la routine principale est `_asm_main`.

4.7.3 Passer des paramètres

Dans les conventions d'appel C, les arguments d'une fonction sont empilés sur la pile dans l'ordre *inverse* de celui dans lequel ils apparaîssent dans l'appel de la fonction.

Considérons l'expression C suivante : `printf("x = %d\n",x);` la Figure 4.11 montre comment elle serait compilée (dans le format NASM équivalent). La Figure 4.12 montre à quoi ressemble la pile après le prologue de la fonction `printf`. La fonction `printf` est une des fonctions de la bibliothèque C qui peut prendre n'importe quel nombre d'arguments. Les règles des conventions d'appel C ont été spécialement écrite pour autoriser ce type de fonctions. Comme l'adresse de la chaîne format est empilée en dernier, son emplacement sur la pile sera *toujours* `EBP + 8` quel que soit le nombre de paramètres passés à la fonction. Le code de `printf` peut alors analyser la chaîne format pour déterminer combien de paramètres ont du être passés et les récupérer sur la pile.

Il n'est pas nécessaire d'utiliser l'assembleur pour gérer un nombre aléatoire d'arguments en C. L'en-tête `stdarg.h` *définit des macros qui peuvent être utilisées pour l'effectuer de façon portable. Voyez n'importe quel bon livre sur le C pour plus de détails.*

Bien sûr, s'il y a une erreur, `printf("x = %d\n")`, le code de `printf` affichera quand même la valeur double mot en `[EBP + 12]`. Cependant, ce ne sera pas la valeur de `x` !

4.7.4 Calculer les adresses des variables locales

Trouver l'adresse d'une étiquette définie dans les segments `data` ou `bss` est simple. Basiquement, l'éditeur de liens le fait. Cependant, calculer l'adresse d'une variable locale (ou d'un paramètre) sur la pile n'est pas aussi intuitif. Néanmoins, c'est un besoin très courant lors de l'appel de sous-routines. Considérons le cas du passage de l'adresse d'une variable (appelons la `x`) à une fonction (appelons la `foo`). Si `x` est situé en EBP $-$ 8 sur la pile, on ne peut pas utiliser simplement :

```
mov     eax, ebp - 8
```

Pourquoi ? La valeur que `MOV` stocke dans EAX doit être calculée par l'assembleur (c'est-à-dire qu'elle doit donner une constante). Cependant, il y a une instruction qui effectue le calcul désiré. Elle est appelée `LEA` (pour *Load Effective Address*, Charger l'Adresse Effective). L'extrait suivant calculerait l'adresse de `x` et la stockerait dans EAX :

```
lea     eax, [ebp - 8]
```

Maintenant, EAX contient l'adresse de `x` et peut être placé sur la pile lors de l'appel de la fonction `foo`. Ne vous méprenez pas, au niveau de la syntaxe, c'est comme si cette instruction lisait la donnée en [EBP$-$8] ; cependant, ce n'est *pas* vrai. L'instruction `LEA` ne lit *jamais* la mémoire ! Elle calcule simplement l'adresse qui sera lue par une autre instruction et stocke cette

adresse dans sa première opérande registre. Comme elle ne lit pas la mémoire, aucune taille mémoire (*p.e.* `dword`) n'est nécessaire ni autorisée.

4.7.5 Retourner des valeurs

Les fonctions C non void retournent une valeur. Les conventions d'appel C spécifient comment cela doit être fait. Les valeurs de retour sont passées via les registres. Tous les types entiers (`char`, `int`, `enum`, *etc.*) sont retournés dans le registre EAX. S'ils sont plus petits que 32 bits, ils sont étendus à 32 bits lors du stockage dans EAX (la façon dont ils sont étendus dépend du fait qu'ils sont signés ou non). Les valeurs 64 bits sont retournées dans la paire de registres EDX:EAX. Les valeurs de pointeurs sont également stockées dans EAX. Les valeurs en virgule flottante sont stockées dans le registre STP du coprocesseur arithmétique (ce registre est décrit dans le chapitre sur les nombres en virgule flottante).

4.7.6 Autres conventions d'appel

Les règles ci-dessus décrivent les conventions d'appel C supportées par tous les compilateurs C 80x86. Souvent, les compilateurs supportent également d'autres conventions d'appel. Lorsqu'il y a une interface avec le langage assembleur, il est *très* important de connaître les conventions utilisées par le compilateur lorsqu'il appelle votre fonction. Habituellement, par défaut, ce sont les conventions d'appel standard qui sont utilisées ; cependant, ce n'est pas toujours le cas[4]. Les compilateurs qui utilisent plusieurs conventions ont souvent des options de ligne de commende qui peuvent être utilisés pour changer la convention par défaut. Il fournissent également des extensions à la syntaxe C pour assigner explicitement des conventions d'appel à des fonctions de manière individuelle. Cependant, ces extensions ne sont pas standardisées et peuvent varier d'un compilateur à l'autre.

Le compilateur GCC autorise différentes conventions d'appel. La convention utilisée par une fonction peut être déclarée explicitement ent utilisant l'extension `__attribute__` . Par exemple, pour déclarer une fonction void qui utilise la convention d'appel standard appelée `f` qui ne prend qu'un paramètre `int`, utilisez la syntaxe suivante pour son prototype :

```
void f( int ) __attribute__((cdecl));
```

GCC supporte également la convention d'appel *standard call* . La fonction ci-dessus pourrait être déclarée afin d'utiliser cette convention en remplaçant le `cdecl` par `stdcall`. La différence entre `stdcall` et `cdecl` est que `stdcall` impose à la sous-routine de retirer les paramètres de la pile (comme le fait la

4. Le compiltateur C Watcom est un exemple qui n'utilise *pas* les conventions standards par défaut. Voyez les sources exemple pour Watcom pour plus de détails

convention d'appel Pascal). Donc, la convention `stdcall` ne peut être utilisée que par des fonctions qui prennent un nombre fixe d'arguments (*i.e.* celles qui ne sont pas comme `printf` et `scanf`).

GCC supporte également un attribut supplémentaire appelé `regparm` qui indique au compilateur d'utiliser les registres pour passer jusqu'à 3 arguments entiers à une fonction au lieu d'utiliser la pile. C'est un type d'optimisation courant que beaucoup de compilateurs supportent.

Borland et Microsoft utilisent une syntaxe commune pour déclarer les conventions d'appel. Ils ajoutent les mots clés `__cdecl` et `__stdcall` au C. Ces mots clés se comportent comme des modificateurs de fonction et appraissent immédiatement avant le nom de la fonction dans un prototype. Par exemple, la fonction `f` ci-dessus serait définie comme suit par Borland et Microsoft:

```
void __cdecl f( int );
```

Il y a des avantages et des inconvénients à chacune des conventions d'appel. Les principaux avantages de `cdecl` est qu'elle est simple et très flexible. Elle peut être utilisée pour n'importe quel type de fonction C et sur n'importe quel compilateur C. Utiliser d'autres conventions peut limiter la portabilité de la sous-routine. Son principal inconvénient est qu'elle peut être plus lente que certaines autres et utilise plus de mémoire (puisque chaque appel de fonction nécessite du code pour retirer les paramètres de la pile).

L'avantage de la convention `stdcall` est qu'elle utilise moins de mémoire que `cdecl`. Aucun nettoyage de pile n'est requis après l'instruction `CALL`. Son principal inconvénient est qu'elle ne peut pas être utilisée avec des fonctions qui ont un nombre variable d'arguments.

L'avantage d'utiliser une convention qui se sert des registres pour passer des paramètres entiers est la rapidité. Le principal inconvénient est que la convention est plus complexe. Certains paramètres peuvent se trouver dans des registres et d'autres sur la pile.

4.7.7 Exemples

Voici un exemple qui montre comment une routine assembleur peut être interfacée avec un programme C (notez que ce programme n'utilise pas le programme assembleur squelette (Figure 1.7) ni le module driver.c).

main5.c

```
1 #include <stdio.h>
2 /* prototype de la routine assembleur */
3 void calc_sum( int, int * ) __attribute__((cdecl));
4
```

```c
int main( void )
{
  int n, sum;

   printf ("Somme des entiers jusqu'à : ");
  scanf("%d", &n);
  calc_sum(n, &sum);
   printf ("La somme vaut %d\n", sum);
  return 0;
}
```

main5.c

sub5.asm

```
; sous-routine _calc_sum
; trouve la somme des entiers de 1 à n
; Paramètres:
;   n    - jusqu'où faire la somme (en [ebp + 8])
;   sump - pointeur vers un entier dans lequel stocker la somme (en [ebp + 12])
; pseudo-code C:
; void calc_sum( int n, int * sump )
; {
;   int i, sum = 0;
;   for( i=1; i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }

segment .text
        global  _calc_sum
;
; variable locale :
;   sum en [ebp-4]
_calc_sum:
        enter   4,0             ; Fait de la place pour sum sur la pile
        push    ebx             ; IMPORTANT !

        mov     dword [ebp-4],0 ; sum = 0
        dump_stack 1, 2, 4      ; affiche la pile de ebp-8 à ebp+16
        mov     ecx, 1          ; ecx est le i du pseudocode
for_loop:
        cmp     ecx, [ebp+8]    ; cmp i et n
        jnle    end_for         ; si non i <= n, quitter
```

```
Somme des entiers jusqu'a : 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
 +16  BFFFFB80  080499EC
 +12  BFFFFB7C  BFFFFB80
  +8  BFFFFB78  0000000A
  +4  BFFFFB74  08048501
  +0  BFFFFB70  BFFFFB88
  -4  BFFFFB6C  00000000
  -8  BFFFFB68  4010648C
La somme vaut 55
```

FIGURE 4.13 – Exécution du programme sub5

```
30
31          add     [ebp-4], ecx        ; sum += i
32          inc     ecx
33          jmp     short for_loop
34
35  end_for:
36          mov     ebx, [ebp+12]       ; ebx = sump
37          mov     eax, [ebp-4]        ; eax = sum
38          mov     [ebx], eax
39
40          pop     ebx                 ; restaure ebx
41          leave
42          ret
```

sub5.asm

Pourquoi la ligne 22 de `sub5.asm` est si importante ? Parce que les conventions d'appel C imposent que la valeur de EBX ne soit pas modifiée par l'appel de fonction. Si ce n'est pas respecté, il est très probable que le programme ne fonctionne pas correctement.

La ligne 25 montre la façon dont fonctionne la macro `dump_stack`. Souvenez vous que le premier paramètre est juste une étiquette numérique et les deuxième et troisième paramètres déterminent respectivement combien de doubles mots elle doit afficher en-dessous et au-dessus de EBP. La Figure 4.13 montre une exécution possible du programme. Pour cette capture, on peut voir que l'adresse du dword où stocker la somme est BFFFFB80 (en EBP + 12) ; le nombre jusqu'auquel additionner est 0000000A (en EBP + 8) ; l'adresse de retour pour la routine est 08048501 (en EBP + 4) ; la valeur sauvegardée de EBP est BFFFFB88 (en EBP); la valeur de la variable locale est

0 en (EBP - 4) ; et pour finir, la valeur sauvegardée de EBX est 4010648C (en EBP - 8).

La fonction `calc_sum` pourrait être réécrite pour retourner la somme plutôt que d'utiliser un pointeur. Comme la somme est une valeur entière, elle doit être placée dans le registre EAX. La ligne 11 du fichier `main5.c` deviendrait :

```
sum = calc_sum(n);
```

De plus, le prototype de `calc_sum` devrait être altéré. Voici le code assembleur modifié :

sub6.asm

```
1  ; sous-routine _calc_sum
2  ; trouve la somme des entiers de 1 à n
3  ; Paramètres:
4  ;   n    - jusqu'où faire la somme (en [ebp + 8])
5  ; Valeur de retour :
6  ;   valeur de la somme
7  ; pseudo-code C :
8  ; int calc_sum( int n )
9  ; {
10 ;   int i, sum = 0;
11 ;   for( i=1; i <= n; i++ )
12 ;     sum += i;
13 ;   return sum;
14 ; }
15 segment .text
16         global  _calc_sum
17 ;
18 ; variable locale :
19 ;   sum en [ebp-4]
20 _calc_sum:
21         enter   4,0               ; fait de la place pour la somme sur la pile
22
23         mov     dword [ebp-4],0   ; sum = 0
24         mov     ecx, 1            ; ecx est le i du pseudocode
25 for_loop:
26         cmp     ecx, [ebp+8]      ; cmp i et n
27         jnle    end_for           ; si non i <= n, quitter
28
29         add     [ebp-4], ecx      ; sum += i
30         inc     ecx
31         jmp     short for_loop
```

```
segment .data
format       db "%d", 0

segment .text
...
      lea    eax, [ebp-16]
      push   eax
      push   dword format
      call   _scanf
      add    esp, 8
...
```

FIGURE 4.14 – Appeler `scanf` depuis l'assembleur

```
end_for:
        mov     eax, [ebp-4]       ; eax = sum

        leave
        ret
```

sub6.asm

4.7.8 Appeler des fonctions C depuis l'assembleur

Un des avantages majeurs d'interfacer le C et l'assembleur est que cela permet au code assembleur d'accéder à la grande bibliothèque C et aux fonctions utilisateur. Par exemple, si l'on veut appeler la fonction `scanf` pour lire un entier depuis le clavier ? La Figure 4.14 montre comment le faire. Une chose très importante à se rappeler est que `scanf` suit les conventions d'appel C standards à la lettre. Cela signifie qu'elle préserve les valeurs des registres EBX, ESI et EDI ; cependant, les registres EAX, ECX et EDX peuvent être modifiés ! En fait, EAX sera modifié car il contiendra la valeur de retour de l'appel à `scanf`. Pour d'autres exemple d'interface entre l'assembleur et le C, observez le code dans `asm_io.asm` qui a été utilisé pour créer `asm_io.obj`.

4.8 Sous-Programmes Réentrants et Récursifs

Un sous-programme réentrant rempli les critères suivants :

— Il ne doit pas modifier son code. Dans un langage de haut niveau, cela serait difficile mais en assembleur, il n'est pas si dur que cela pour un programme de modifier son propre code. Par exemple :

```
mov    word [cs:$+7], 5      ; copie 5 dans le mot 7 octets plus loin
add    ax, 2                 ; l'expression précédente change 2 en 5 !
```

Ce code fonctionnerait en mode réel, mais sur les systèmes d'exploitation en mode protégé, le segment de code est marqué en lecture seule. Lorsque la première ligne ci-dessus s'exécute, le programme est interrompu sur ces systèmes. Cette façon de programmer est mauvaise pour beaucoup de raison. Elle porte à confusion, est difficile à maintenir et ne permet pas le partage de code (voir ci-dessous).

— Il ne doit pas modifier de données globales (comme celles qui se trouvent dans les segments `data` et `bss`). Toutes les variables sont stockées sur la pile.

Il y a plusieurs avantages à écrire du code réentrant.

— Un sous-programme réentrant peut être appelé récursivement.

— Un programme réentrant peut être partagé par plusieurs processus. Sur beaucoup de systèmes d'exploitation multi-tâches, s'il y a plusieurs instances d'un programme en cours, seule *une* copie du code se trouve en mémoire. Les bibliothèques partagées et les DLLs (*Dynamic Link Libraries*, Bibliothèques de Lien Dynamique) utilisent le même principe.

— Les sous-programmes réentrants fonctionnent beaucoup mieux dans les programmes multi-threadés[5]. Windows 9x/NT et la plupart des systèmes d'exploitation de style Unix (Solaris, Linux, *etc.*) supportent les programmes multi-threadés.

4.8.1 Sous-programmes récursifs

Ce type de sous-programmes s'appellent eux-mêmes. La récursivité peut être soit *directe* soit *indirecte*. La récursivité directe survient lorsqu'un sous-programme, disons `foo`, s'appelle lui-même dans le corps de `foo`. La récursivité indirecte survient lorsqu'un sous-programme ne s'appelle pas directement lui-même mais via un autre sous-programme qu'il appelle. Par exemple, le sous-programme `foo` pourrait appeler `bar` et `bar` pourrait appeler `foo`.

Les sous-programmes récursifs doivent avoir une *condition de terminaison*. Lorsque cette condition est vraie, il n'y a plus d'appel récursif. Si une routine récursive n'a pas de condition de terminaison ou que la condition n'est jamais remplie, la récursivité ne s'arrêtera jamais (exactement comme une boucle infinie).

La Figure 4.15 montre une fonction qui calcule une factorielle récursivement. Elle peut être appelée depuis le C avec :

```
x = fact(3);          /* trouve 3! */
```

5. Un programme multi-threadé a plusieurs threads d'exécution. C'est-à-dire que le programme lui-même est multi-tâches.

```
1  ; trouve n!
2  segment .text
3        global _fact
4  _fact:
5        enter  0,0
6
7        mov    eax, [ebp+8]     ; eax = n
8        cmp    eax, 1
9        jbe    term_cond        ; si n <= 1, terminé
10       dec    eax
11       push   eax
12       call   _fact            ; appel fact(n-1) récursivement
13       pop    ecx              ; réponse dans eax
14       mul    dword [ebp+8]    ; edx:eax = eax * [ebp+8]
15       jmp    short end_fact
16 term_cond:
17       mov    eax, 1
18 end_fact:
19       leave
20       ret
```

FIGURE 4.15 – Fonction factorielle récursive

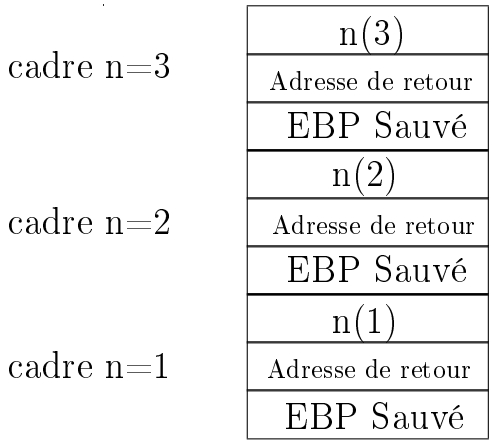

FIGURE 4.16 – Cadres de pile pour la fonction factorielle

```
1 void f( int x )
2 {
3   int i;
4   for( i=0; i < x; i++ ) {
5     printf ("%d\n", i);
6     f(i);
7   }
8 }
```

FIGURE 4.17 – Un autre exemple (version C)

La Figure 4.16 montre à quoi ressemble la pile au point le plus profond pour l'appel de fonction ci-dessus.

Les Figures 4.17 et 4.18 montrent un exemple récursif plus compliqué en C et en assmbleur, respectivement. Quelle est la sortie pour `f(3)` ? Notez que l'instruction `ENTER` crée un nouveau `i` sur la pile pour chaque appel récursif. Donc, chaque instance récursive de `f` a sa propre variable `i` indépendante. Définir `i` comme un double mot dans le segment `data` ne fonctionnerait pas pareil.

4.8.2 Révision des types de stockage des variables en C

Le C fournit plusieurs types de stockage des variables.

global Ces variables sont déclarées en dehors de toute fonction et sont stockées à des emplacements mémoire fixes (dans les segments `data` ou `bss`) et existent depuis le début du programme jusqu'à la fin. Par défaut, on peut y accéder de n'importe quelle fonction dans le programme ; cependant, si elles sont déclarées comme `static`, seules les fonctions dans le même module peuvent y accéder (*i.e.* en termes assembleur, l'étiquette est interne, pas externe).

static Il s'agit des variables *locales* d'une fonctions qui sont déclarées `static` (Malheureusement, le C utilise le mot clé `static` avec deux sens différents !) Ces variables sont également stockées dans des emplacements mémoire fixes (dans `data` ou `bss`), mais ne peuvent être accédées directement que dans la fonction où elles sont définies.

automatic C'est le type par défaut d'une variable C définie dans une fonction. Ces variables sont allouées sur la pile lorsque la fonction dans laquelle elles sont définies est appelée et sont désallouées lorsque la fonction revient. Donc, elles n'ont pas d'emplacement mémoire fixe.

register Ce mot clé demande au compilateur d'utiliser un registre pour la donnée dans cette variable. Il ne s'agit que d'une *requête*. Le com-

```
1  %define i ebp-4
2  %define x ebp+8          ; macros utiles
3  segment .data
4  format       db "%d", 10, 0     ; 10 = '\n'
5  segment .text
6        global _f
7        extern _printf
8  _f:
9        enter  4,0           ; alloue de la place sur la pile pour i
10
11       mov    dword [i], 0  ; i = 0
12 lp:
13       mov    eax, [i]      ; i < x?
14       cmp    eax, [x]
15       jnl    quit
16
17       push   eax           ; appelle printf
18       push   format
19       call   _printf
20       add    esp, 8
21
22       push   dword [i]     ; appelle f
23       call   _f
24       pop    eax
25
26       inc    dword [i]     ; i++
27       jmp    short lp
28 quit:
29       leave
30       ret
```

FIGURE 4.18 – Un autre exemple (version assembleur)

pilateur n'a *pas* à l'honorer. Si l'adresse de la variable est utilisée à un endroit quelconque du programme, elle ne sera pas respectée (puisque les registres n'ont pas d'adresse). De plus, seuls les types entiers simples peuvent être des valeurs registres. Les types structures ne peuvent pas l'être ; ils ne tiendraient pas dans un registre ! Les compilateurs C placent souvent les variables automatiques normales dans des registres sans aide du programmeur.

volatile Ce mot clé indique au compilateur que la valeur de la variable peut changer à tout moment. Cela signifie que le compilateur ne peut faire aucune supposition sur le moment où la variable est modifiée. Souvent, un compilateur stocke la valeur d'une variable dans un registre temporairement et utilise le registre à la place de la variable dans une section de code. Il ne peut pas faire ce type d'optimisations avec les variables `volatile`. Un exemple courant de variable volatile serait une variable qui peut être modifiée par deux threads d'un programme multi-threadé. Considérons le code suivant :

```
1 x = 10;
2 y = 20;
3 z = x;
```

Si `x` pouvait être altéré par un autre thread, il est possible que l'autre thread change `x` entre les lignes 1 et 3, alors `z` ne vaudrait pas 10. Cependant, si `x` n'a pas été déclaré comme volatile, le compilateur peut supposer que `x` est inchangé et positionner `z` à 10.

Une autre utilisation de `volatile` est d'empêcher le compilateur d'utiliser un registre pour une variable.

Chapitre 5

Tableaux

5.1 Introduction

Un *tableau* est un bloc contigu de données en mémoire. Tous les éléments de la liste doivent être du même type et occuper exactement le même nombre d'octets en mémoire. En raison de ces propriétés, les tableaux permettent un accès efficace à une donnée par sa position (ou indice) dans le tableau. L'adresse de n'importe quel élément peut être calculée en connaissant les trois choses suivantes :

— L'adresse du premier élément du tableau
— Le nombre d'octets de chaque élément
— L'indice de l'élément

Il est pratique de considérer l'indice du premier élément du tableau comme étant 0 (comme en C). Il est possible d'utiliser d'autres valeurs pour le premier indice mais cela complique les calculs.

5.1.1 Définir des tableaux

Définir des tableaux dans les segments `data` et `bss`

Pour définir un tableau initialisé dans le segment `data`, utilisez les directives `db`, `dw`, *etc.* normales. directives. NASM fournit également une directive utile appelée `TIMES` qui peut être utilisée pour répéter une expression de nombreuses fois sans avoir à la dupliquer à la main. La Figure 5.1 montre plusieurs exemples.

Pour définir un tableau non initialisé dans le segment `bss`, utilisez les directives `resb`, `resw`, *etc.* Souvenez vous que ces directives ont une opérande qui spécifie le nombre d'unités mémoire à réserver. La Figure 5.1 montre également des exemples de ces types de définitions.

```
1  segment .data
2  ; définit un tableau de 10 doubles mots initialisés à 1,2,..,10
3  a1           dd   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
4  ; définit un tableau de 10 mots initialisés à 0
5  a2           dw   0, 0, 0, 0, 0, 0, 0, 0, 0, 0
6  ; idem mais en utilisant TIMES
7  a3           times 10 dw 0
8  ; définit un tableau d'octets avec 200 0 puis 100 1
9  a4           times 200 db 0
10             times 100 db 1
11
12 segment .bss
13 ; définit un tableau de 10 doubles mots non initialisés
14 a5           resd  10
15 ; définit un tableau de 100 mots non initialisés
16 a6           resw  100
```

FIGURE 5.1 – Définir des tableaux

Définir des tableaux comme variables locales

Il n'y a pas de manière directe de définir un tableau comme variable locale sur la pile. Comme précédemment, on calcule le nombre total d'octets requis pour *toutes* les variables locales, y compris les tableaux, et on l'ôte de ESP (soit directement, soit en utilisant l'instruction `ENTER`). Par exemple, si une fonction a besoin d'une variable caractère, deux entiers double-mot et un tableau de 50 éléments d'un mot, il faudrait $1 + 2 \times 4 + 50 \times 2 = 109$ octets. Cependant, le nombre ôté de ESP doit être un multiple de quatre (112 dans ce cas) pour maintenir ESP sur un multiple de double mot. On peut organiser les variables dans ces 109 octets de plusieurs façons. La Figure 5.2 montre deux manières possibles. La partie inutilisée sert à maintenir les doubles mots sur des adresses multiples de doubles mots afin d'accélérer les accès mémoire.

5.1.2 Accéder aux éléments de tableaux

Il n'y a pas d'opérateur `[ ]` en langage assembleur comme en C. Pour accéder à un élément d'un tableau, son adresse doit être calculée. Considérons les deux définitions de tableau suivantes :

```
array1       db     5, 4, 3, 2, 1     ; tableau d'octets
array2       dw     5, 4, 3, 2, 1     ; tableau de mots
```

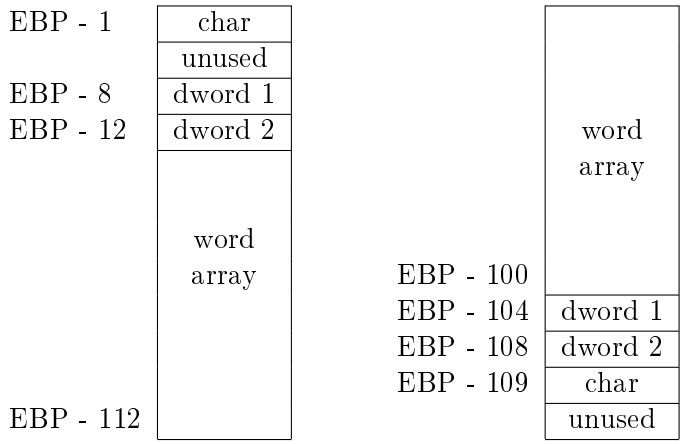

FIGURE 5.2 – Organisations possibles de la pile

Voici quelques exemple utilisant ces tableaux :

```
1  mov    al, [array1]             ; al = array1[0]
2  mov    al, [array1 + 1]         ; al = array1[1]
3  mov    [array1 + 3], al         ; array1[3] = al
4  mov    ax, [array2]             ; ax = array2[0]
5  mov    ax, [array2 + 2]         ; ax = array2[1] (PAS array2[2]!)
6  mov    [array2 + 6], ax         ; array2[3] = ax
7  mov    ax, [array2 + 1]         ; ax = ??
```

A la ligne 5, l'élément 1 du tableau de mots est référencé, pas l'élément 2. Pourquoi ? Les mots sont des unités de deux octets, donc pour se déplacer à l'élément suivant d'un tableau de mots, on doit se déplacer de deux octets, pas d'un seul. La ligne 7 lit un octet du premier élément et un octet du suivant. En C, le compilateur regarde la type de pointeur pour déterminer de combien d'octets il doit se déplacer dans une expression utilisant l'arithmétique des pointeurs, afin que le programmeur n'ait pas à le faire. Cependant, en assembleur, c'est au programmeur de prendre en compte a taille des éléments du tableau lorsqu'il se déplace parmi les éléments.

La Figure 5.3 montre un extrait de code qui additionne tous les éléments de `array1` de l'exemple de code précédent. A la ligne 7, AX est ajouté à DX. Pourquoi pas AL ? Premièrement, les deux opérandes de l'instruction `ADD` doivent avoir la même taille. Deuxièmement, il serait facile d'ajouter des octets et d'obtenir une somme qui serait trop grande pour tenir sur un octet. En utilisant DX, la somme peut atteindre 65 535. Cependant, il est important de réaliser que AH est également additionné. C'est pourquoi, AH est positionné à zéro [1] à la ligne 3.

1. Positionner AH à zéro équivaut à supposer implicitement que AL est un nombre

```
1       mov     ebx, array1             ; ebx = adresse de array1
2       mov     dx, 0                   ; dx contiendra sum
3       mov     ah, 0                   ; ?
4       mov     ecx, 5
5   lp:
6       mov     al, [ebx]               ; al = *ebx
7       add     dx, ax                  ; dx += ax (pas al!)
8       inc     ebx                     ; bx++
9       loop    lp
```

FIGURE 5.3 – Faire la somme des éléments d'un tableau (Version 1)

```
1       mov     ebx, array1             ; ebx = adresse de array1
2       mov     dx, 0                   ; dx contiendra la somme
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       jnc     next                    ; si pas de retenue goto next
7       inc     dh                      ; incrémente dh
8   next:
9       inc     ebx                     ; bx++
10      loop    lp
```

FIGURE 5.4 – Faire la somme des éléments d'un tableau (Version 2)

Les Figures 5.4 et 5.5 montrent deux manières alternatives de calculer la somme. Les lignes en italique remplacent les lignes 6 et 7 de la Figure 5.3.

5.1.3 Adressage indirect plus avancé

Ce n'est pas étonnant, l'adressage indirect est souvent utilisé avec les tableaux. La forme la plus générale d'une référence mémoire indirecte est :

[*reg de base + facteur*reg d'index + constante*]

où :

reg de base est un des registres EAX, EBX, ECX, EDX, EBP, ESP, ESI ou EDI.

non signé. S'il est signé, l'action appropriée serait d'insérer une instruction `CBW` entre les lignes 6 et 7.

```
1       mov     ebx, array1             ; ebx = adresse de array1
2       mov     dx, 0                   ; dx contiendra la somme
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       adc     dh, 0                   ; dh += drapeau de retenue + 0
7       inc     ebx                     ; bx++
8       loop    lp
```

FIGURE 5.5 – Faire la somme des éléments d'un tableau (Version 3)

facteur est 1, 2, 4 ou 8 (S'il vaut 1, le facteur est omis).

reg d'index est un des registres EAX, EBX, ECX, EDX, EBP, ESI, EDI (Notez que ESP n'est pas dans la liste).

constante est une constante 32 bits. Cela peut être une étiquette (ou une expression d'étiquette).

5.1.4 Exemple

Voici un exemple qui utilise un tableau et le passe à une fonction. Il utilise le programme `array1c.c` (dont le listing suit) comme pilote, pas le programme `driver.c`.

array1.asm

```
1   %define ARRAY_SIZE 100
2   %define NEW_LINE 10
3
4   segment .data
5   FirstMsg        db   "10 premiers éléments du tableau", 0
6   Prompt          db   "Entrez l'indice de l'élément à afficher : ", 0
7   SecondMsg       db   "L'élément %d vaut %d", NEW_LINE, 0
8   ThirdMsg        db   "Eléments 20 à 29 du tableau", 0
9   InputFormat     db   "%d", 0
10
11  segment .bss
12  array           resd ARRAY_SIZE
13
14  segment .text
15          extern  _puts, _printf, _scanf, _dump_line
16          global  _asm_main
17  _asm_main:
```

```
18          enter   4,0                ; variable locale dword en EBP - 4
19          push    ebx
20          push    esi
21
22  ; initialise le tableau à 100, 99, 98, 97, ...
23
24          mov     ecx, ARRAY_SIZE
25          mov     ebx, array
26  init_loop:
27          mov     [ebx], ecx
28          add     ebx, 4
29          loop    init_loop
30
31          push    dword FirstMsg         ; affiche FirstMsg
32          call    _puts
33          pop     ecx
34
35          push    dword 10
36          push    dword array
37          call    _print_array           ; affiche les 10 premiers éléments du tableau
38          add     esp, 8
39
40  ; demande à l'utilisateur l'indice de l'élément
41  Prompt_loop:
42          push    dword Prompt
43          call    _printf
44          pop     ecx
45
46          lea     eax, [ebp-4]       ; eax = adresse du dword local
47          push    eax
48          push    dword InputFormat
49          call    _scanf
50          add     esp, 8
51          cmp     eax, 1               ; eax = valeur de retour de scanf
52          je      InputOK
53
54          call    _dump_line  ; ignore le reste de la ligne et recommence
55          jmp     Prompt_loop          ; si la saisie est invalide
56
57  InputOK:
58          mov     esi, [ebp-4]
59          push    dword [array + 4*esi]
```

```
60          push    esi
61          push    dword SecondMsg      ; affiche la valeur de l'élément
62          call    _printf
63          add     esp, 12
64
65          push    dword ThirdMsg       ; affiche les éléments 20 à 29
66          call    _puts
67          pop     ecx
68
69          push    dword 10
70          push    dword array + 20*4     ; adresse de array[20]
71          call    _print_array
72          add     esp, 8
73
74          pop     esi
75          pop     ebx
76          mov     eax, 0            ; retour au C
77          leave
78          ret
79
80  ;
81  ; routine _print_array
82  ; Routine appelable depuis le C qui affiche les éléments d'un tableau de doubles mots
83  ; comme des entiers signés.
84  ; Prototype C:
85  ; void print_array( const int * a, int n);
86  ; Paramètres:
87  ;   a - pointeur vers le tableau à afficher (en ebp+8 sur la pile)
88  ;   n - nombre d'entiers à afficher (en ebp+12 sur la pile)
89
90  segment .data
91  OutputFormat    db   "%-5d %5d", NEW_LINE, 0
92
93  segment .text
94          global  _print_array
95  _print_array:
96          enter   0,0
97          push    esi
98          push    ebx
99
100         xor     esi, esi                ; esi = 0
101         mov     ecx, [ebp+12]           ; ecx = n
```

```
102         mov     ebx, [ebp+8]              ; ebx = adresse du tableau
103 print_loop:
104         push    ecx                       ; printf change ecx !
105
106         push    dword [ebx + 4*esi]       ; empile tableau[esi]
107         push    esi
108         push    dword OutputFormat
109         call    _printf
110         add     esp, 12                   ; retire les paramètres (laisse ecx !)
111
112         inc     esi
113         pop     ecx
114         loop    print_loop
115
116         pop     ebx
117         pop     esi
118         leave
119         ret
```

array1.asm

array1c.c

```
1  #include <stdio.h>
2
3  int asm_main( void );
4  void dump_line( void );
5
6  int main()
7  {
8    int ret_status;
9    ret_status = asm_main();
10   return ret_status;
11 }
12
13 /*
14  * fonction dump_line
15  * retire tous les caractères restant sur la ligne courante dans le buffer d'entrée
16  */
17 void dump_line()
18 {
19   int ch;
20
21   while( (ch = getchar()) != EOF && ch != '\n')
```

```
22      /* null body*/ ;
23  }
```

array1c.c

L'instruction `LEA` revisitée

L'instruction `LEA` peut être utilisée dans d'autres cas que le calcul d'adresse. Elle est assez couramment utilisée pour les calculs rapides. Considérons le code suivant :

```
lea     ebx, [4*eax + eax]
```

Il stocke la valeur de $5 \times$ `EAX` dans EBX. Utiliser `LEA` dans ce cas est à la fois plus simple et plus rapide que d'utiliser `MUL`. Cependant, il faut être conscient du fait que l'expression entre crochets *doit* être une adresse indirecte légale. Donc, par exemple, cette instruction ne peut pas être utilisée pour multiplier par 6 rapidement.

5.1.5 Tableaux Multidimensionnels

Les tableaux multidimensionnels ne sont pas vraiments différents de ceux à une dimension dont nous avons déjà parlé. En fait, il sont représentés en mémoire exactement comme cela, un tableau à une dimension.

Tableaux à Deux Dimensions

Ce n'est pas étonnant, le tableau multidimensionnel le plus simple est celui à deux dimensions. Un tableau à deux dimensions est souvent représenté comme une grille d'éléments. Chaque élément est identifié par une paire d'indices. Par convention, le premier indice est associé à la ligne et le second à la colonne.

Considérons un tableau avec trois lignes et deux colonnes défini de la manière suivante :

```
int a [3][2];
```

Le compilateur C réserverait de la place pour un tableau d'entiers de 6 ($= 2 \times 3$) et placerait les éléments comme suit :

Indice	0	1	2	3	4	5
Elément	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

Ce que ce tableau tente de montrer et que l'élément réferencé comme `a[0][0]` est stocké au début du tableau de 6 éléments à une dimension. L'élément

```
1    mov    eax, [ebp - 44]          ; ebp - 44 est l'emplacement de i
2    sal    eax, 1                   ; multiplie i par 2
3    add    eax, [ebp - 48]          ; ajoute j
4    mov    eax, [ebp + 4*eax - 40]  ; ebp - 40 est l'adresse de a[0][0]
5    mov    [ebp - 52], eax          ; stocke le résultat dans x (en ebp - 52)
```

FIGURE 5.6 – Assembleur correspondant à `x = a[i][j]`

`a[0][1]` est stocké à la position suivante (indice 1) *etc.* Chaque ligne du tableau à deux dimensions est stockée en mémoire de façon contigüe. Le dernier élément d'une ligne est suivi par le premier élément de la suivante. On appelle cela la représentation *au niveau ligne* (rowwise) du tableau et c'est comme cela qu'un compilateur C/C++ représenterait le tableau.

Comment le compilateur détermine où `a[i][j]` se trouve dans la représentation au niveau ligne ? Une formule simple calcule l'indice à partir de `i` et `j`. La formule dans ce cas est $2i+j$. Il n'est pas compliqué de voir comment on obtient cette formule. Chaque ligne fait deux éléments de long ; donc le premier élément de la ligne i est à l'emplacement $2i$. Puis on obtient l'emplacement de la colonne j en ajoutant j à $2i$. Cette analyse montre également comment la formule est généralisée à un tableau de `N` colonnes : $N \times i + j$. Notez que la formule ne dépend *pas* du nombre de lignes.

Pour illustrer, voyons comment *gcc* compile le code suivant (utilisant le tableau `a` défini plus haut) :

```
x = a[i][j];
```

La Figure 5.6 montre le code assembleur correspondant. Donc, le compilateur convertit grossièrement le code en :

```
x = *(&a[0][0] + 2*i + j);
```

et en fait, le programmeur pourrait l'écrire de cette manière et obtenir le même résultat.

Il n'y a rien de magique à propos du choix de la représentation niveau ligne du tableau. Une représentation niveau colonne fonctionnerait également :

Indice	0	1	2	3	4	5
Elément	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

Dans la représentation niveau colonne, chaque colonne est stockée de manière contigüe. L'élément `[i][j]` est stocké à l'emplacement $i+3j$. D'autres langages (FORTRAN, par exemple) utilisent la représentation niveau colonne. C'est important lorsque l'on interface du code provenant de multiples langages.

Dimensions Supérieures à Deux

Pour les dimensions supérieures à deux, la même idée de base est appliquée. Considérons un tableau à trois dimensions :

```
int b [4][3][2];
```

Ce tableau serait stocké comme s'il était composé de trois tableaux à deux dimensions, chacun de taille `[3][2]` stockés consécutivement en mémoire. Le tableau ci-dessous montre comment il commence :

Indice	0	1	2	3	4	5
Elément	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
Indice	6	7	8	9	10	11
Elément	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

La formule pour calculer la position de `b[i][j][k]` est $6i + 2j + k$. Le 6 est déterminé par la taille des tableaux `[3][2]`. En général, pour un tableau de dimension `a[L][M][N]` l'emplacement de l'élément `a[i][j][k]` sera $M \times N \times i + N \times j + k$. Notez, là encore, que la dimension `L` n'apparaît pas dans la formule.

Pour les dimensions plus grandes, le même procédé est généralisé. Pour un tableau à n dimensions de dimension D_1 à D_n, l'emplacement d'un élément repéré par les indices i_1 à i_n est donné par la formule :

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

ou pour les fanas de maths, on peut l'écrire de façon plus concise :

$$\sum_{j=1}^{n} \left(\prod_{k=j+1}^{n} D_k \right) i_j$$

La première dimension, D_1, n'apparaît pas dans la formule.

C'est là que vous comprenez que l'auteur est un major de physique (ou la référence à FORTRAN vous avait déjà mis sur la voie ?)

Pour la représentation au niveau colonne, la formule générale serait :

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

ou dans la notation des fanas de maths :

$$\sum_{j=1}^{n} \left(\prod_{k=1}^{j-1} D_k \right) i_j$$

Dans ce cas, c'est la dernière dimension, D_n, qui n'apparaît pas dans la formule.

Passer des Tableaux Multidimensionnels comme Paramètres en C

La représentation au niveau ligne des tableaux multidimensionnels a un effet direct sur la programmation C. Pour les tableaux à une dimension, la taille du tableau n'est pas nécessaire pour calculer l'emplacement en mémoire de n'importe quel élément. Ce n'est pas vrai pour les tableaux multidimensionnels. Pour accéder aux éléments de ces tableaux, le compilateur doit connaître toutes les dimensions sauf la première. Cela saute aux yeux lorsque l'on observe le prototype d'une fonction qui prend un tableau multidimensionnel comme paramètre. Ce qui suit ne compilera pas :

```
void f( int a[ ][ ] );  /* pas d'information sur la dimension */
```

Cependant, ce qui suit compile :

```
void f( int a[ ][2] );
```

Tout tableau à deux dimensions à deux colonnes peut être passé à cette fonction. La première dimension n'est pas nécessaire [2].

Ne confondez pas avec une fonction ayant ce prototype :

```
void f( int * a[ ] );
```

Cela définit un tableau à une dimension de pointeurs sur des entiers (qui peut à son tour être utilisé pour créer un tableau de tableaux qui se comportent plus comme un tableau à deux dimensions).

Pour les tableaux de dimensions supérieures, toutes les dimensions sauf la première doivent être données pour les paramètres. Par exemple, un paramètre tableau à quatre dimensions peut être passé de la façon suivante :

```
void f( int a[ ][4][3][2] );
```

5.2 Instructions de Tableaux/Chaînes

La famille des processeurs 80x86 fournit plusieurs instructions conçues pour travailler avec les tableaux. Ces instructions sont appelées *instructions de chaînes*. Elles utilisent les registres d'index (ESI et EDI) pour effectuer une opération puis incrémentent ou décrémentent automatiquement l'un des registres d'index ou les deux. Le *drapeau de direction* (DF) dans le registre FLAGS détermine si les registres d'index sont incrémentés ou décrémentés. Il y a deux instructions qui modifient le drapeau de direction :

CLD éteint le drapeau de direction. Les registres d'index sont alors incrémentés.

STD allume le drapeau de direction. Les registres d'index sont alors décrémentés.

2. On peut indiquer une taille mais elle n'est pas prise en compte par le compilateur.

LODSB AL = [DS:ESI] ESI = ESI ± 1	STOSB [ES:EDI] = AL EDI = EDI ± 1
LODSW AX = [DS:ESI] ESI = ESI ± 2	STOSW [ES:EDI] = AX EDI = EDI ± 2
LODSD EAX = [DS:ESI] ESI = ESI ± 4	STOSD [ES:EDI] = EAX EDI = EDI ± 4

FIGURE 5.7 – Instructions de lecture et d'écriture de chaîne

```
1  segment .data
2  array1  dd  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4  segment .bss
5  array2  resd 10
6
7  segment .text
8        cld                    ; à ne pas oublier !
9        mov    esi, array1
10       mov    edi, array2
11       mov    ecx, 10
12 lp:
13       lodsd
14       stosd
15       loop  lp
```

FIGURE 5.8 – Exemples de chargement et de stockage

Une erreur *très* courante dans la programmation 80x86 est d'oublier de positionner le drapeau de direction. Cela conduit souvent à un code qui fonctionne la plupart du temps (lorsque le drapeau de direction est dans la position désirée), mais ne fonctionne pas *tout* le temps.

5.2.1 Lire et écrire en mémoire

Les instructions de chaîne les plus simples lisent ou écrivent en mémoire, ou les deux. Elles peuvent lire ou écrire un octet, un mot ou un double-mot à la fois. La Figure 5.7 montre ces instructions avec une brève description de ce qu'elles font en pseudo-code. Il y a plusieurs choses à noter ici. Tout d'abord, ESI est utilisé pour lire et EDI pour écrire. C'est facile à retenir si l'on se souvient que SI signifie *Source Index* (Indice Source) et DI *Destination Index* (Indice de Destination). Ensuite, notez que le registre qui contient la

MOVSB	byte [ES:EDI] = byte [DS:ESI] ESI = ESI ± 1 EDI = EDI ± 1
MOVSW	word [ES:EDI] = word [DS:ESI] ESI = ESI ± 2 EDI = EDI ± 2
MOVSD	dword [ES:EDI] = dword [DS:ESI] ESI = ESI ± 4 EDI = EDI ± 4

FIGURE 5.9 – Instructions de déplacement de chaîne en mémoire

```
1  segment .bss
2  array  resd 10
3
4  segment .text
5        cld                    ; à ne pas oublier !
6        mov    edi, array
7        mov    ecx, 10
8        xor    eax, eax
9        rep stosd
```

FIGURE 5.10 – Exemple de tableau mise à zéro d'un tableau

donnée est fixe (AL, AX ou EAX). Enfin, notez que les instructions de stockage utilisent ES pour déterminer le segment dans lequel écrire, pas DS. En programmation en mode protégé, ce n'est habituellement pas un problème, puisqu'il n'y a qu'un segment de données et ES est automatiquement initialisé pour y faire référence (tout comme DS). Cependant, en programmation en mode réel, il est *très* important pour le programmeur d'initialiser ES avec la valeur de sélecteur de segment correcte [3]. La Figure 5.8 montre un exemple de l'utilisation de ces instructions qui copie un tableau dans un autre.

La combinaison des instructions `LODSx` et `STOSx` (comme aux lignes 13 et 14 de la Figure 5.8) est très courante. En fait, cette combinaison peut être effectuée par une seule instruction de chaîne `MOVSx`. La Figure 5.9 décrit les opérations effectuées par cette instruction. Les lignes 13 et 14 de la Figure 5.8 pourraient être remplacées par une seule instruction `MOVSD` avec le même

3. Une autre complication est qu'on ne peut pas copier la valeur du registre DS dans ES directement en utilisant une instruction `MOV`. A la place, la valeur de DS doit être copiée dans un registre universel (comme AX) puis être copié depuis ce registre dans ES, ce qui utilise deux instruction `MOV`.

CMPSB	compare l'octet en [DS:ESI] avec celui en [ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	compare le mot en [DS:ESI] avec celui en [ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	compare de double-mot en [DS:ESI] avec celui en [ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	compare AL et [ES:EDI] EDI ± 1
SCASW	compare AX et [ES:EDI] EDI ± 2
SCASD	compare EAX et [ES:EDI] EDI ± 4

FIGURE 5.11 – Instructions de comparaison de chaînes

résultat. La seule différence serait que le registre EAX ne serait pas utilisé du tout dans la boucle.

5.2.2 Le préfixe d'instruction `REP`

La famille 80x86 fournit un préfixe d'instruction spécial[4] appelé `REP` qui peut être utilisé avec les instructions de chaîne présentées ci-dessus. Ce préfixe indique au processeur de répéter l'instruction de chaîne qui suit un nombre précis de fois. Le registre ECX est utilisé pour compter les itérations (exactement comme pour l'instruction `LOOP`). En utilisant le préfixe `REP`, la boucle de la Figure 5.8 (lignes 12 à 15) pourrait être remplacée par une seule ligne :

```
rep movsd
```

La Figure 5.10 montre un autre exemple qui met à zéro le contenu d'un tableau.

4. Un préfixe d'instruction n'est pas une instruction, c'est un octet spécial qui est placé avant une instruction de chaîne qui modifie son comportement. D'autres préfixes sont également utilisés pour modifier le segment par défaut des accès mémoire

```
1  segment .bss
2  array        resd 100
3
4  segment .text
5        cld
6        mov    edi, array    ; pointeur vers le début du tableau
7        mov    ecx, 100      ; nombre d'éléments
8        mov    eax, 12       ; nombre à rechercher
9  lp:
10       scasd
11       je     found
12       loop   lp
13   ; code à exécuter si non trouvé
14       jmp    onward
15 found:
16       sub    edi, 4         ; edi pointe maintenant vers 12
17   ; code à exécuter si trouvé
18 onward:
```

FIGURE 5.12 – Exemple de recherche

5.2.3 Instructions de comparaison de chaînes

La Figure 5.11 montre plusieurs nouvelles instructions de chaînes qui peuvent être utilisées pour comparer des données en mémoire entre elles ou avec des registres. Elles sont utiles pour comparer des tableaux ou effectuer des recherches. Elles positionnent le registre FLAGS exactement comme l'instruction `CMP`. Les instructions `CMPSx` comparent les emplacements mémoire correspondants et les instructions `SCASx` scannent des emplacements mémoire pour une valeur particulière.

La Figure 5.12 montre un court extrait de code qui recherche le nombre 12 dans un tableau de double mots. L'instruction `SCASD` à la ligne 10 ajoute toujours 4 à EDI, même si la valeur recherchée est trouvée. Donc, si l'on veut trouver l'adresse du 12 dans le tableau, il est nécessaire de soustraire 4 de EDI (comme le fait la ligne 16).

5.2.4 Les préfixes d'instruction `REPx`

Il y a plusieurs autres préfixes d'instruction du même genre que `REP` qui peuvent être utilisés avec les instructions de comparaison de chaînes. La Figure 5.13 montre les deux nouveaux préfixes et décrit ce qu'ils font.

`REPE`, `REPZ`	répète l'instruction tant que le drapeau Z est allumé ou au plus ECX fois
`REPNE`, `REPNZ`	répète l'instruction tant que le drapeau Z est éteint ou au plus ECX fois

FIGURE 5.13 – Préfixes d'instruction `REPx`

```
1  segment .text
2        cld
3        mov    esi, block1        ; adresse du premier bloc
4        mov    edi, block2        ; adresse du second bloc
5        mov    ecx, size          ; taille des blocs en octets
6        repe   cmpsb              ; répéter tant que Z est allumé
7        je     equal              ; Z est allumé => blocs égaux
8     ; code à exécuter si les blocs ne sont pas égaux
9        jmp    onward
10 equal:
11    ; code à exécuter s'ils sont égaux
12 onward:
```

FIGURE 5.14 – Comparer des blocs mémoire

`REPE` et `REPZ` sont simplement des synonymes pour le même préfixe (comme le sont `REPNE` et `REPNZ`). Si l'instruction de comparaison de chaînes répétée stoppe à cause du résultat de la comparaison, le ou les registres d'index sont quand même incrémentés et ECX décrémenté ; cependant, le registre FLAGS contient toujours l'état qu'il avait à la fin de la répétition. Donc, il est possible d'utiliser le drapeau Z pour déterminer si la comparaison s'est arrêtée à cause de son résultat ou si c'est parce que ECX a atteint zéro.

Pourquoi ne peut pas pas simplement regarder si ECX est à zéro après la comparaison répétée ?

La Figure 5.14 montre un extrait de code qui détermine si deux blocs de mémoire sont égaux. Le `JE` de la ligne 7 de l'exemple vérifie le résultat de l'instruction précédente. Si la comparaison s'est arrêtée parce qu'elle a trouvée deux octets différents, le drapeau Z sera toujours éteint et aucun branchement n'est effectué ; cependant, si la comparaison s'est arrêtée parce que ECX a atteint zéro, le drapeau Z sera toujours allumé et le code se branche à l'étiquette `equal`.

5.2.5 Exemple

Cette section contient un fichier source assembleur avec plusieurs fonctions qui implémentent des opérations sur les tableaux en utilisant des instructions de chaîne. Beaucoup de ces fonctions font doublons avec des fonc-

tions familières de la bibliothèque C.

memory.asm

```
global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy

segment .text
; fonction _asm_copy
; copie deux blocs de mémoire
; prototype C
; void asm_copy( void * dest, const void * src, unsigned sz);
; paramètres:
;   dest - pointeur sur le tampon vers lequel copier
;   src  - pointeur sur le tampon depuis lequel copier
;   sz   - nombre d'octets à copier

; ci-dessous, quelques symboles utiles sont définis

%define dest [ebp+8]
%define src  [ebp+12]
%define sz   [ebp+16]
_asm_copy:
        enter   0, 0
        push    esi
        push    edi

        mov     esi, src        ; esi = adresse du tampon depuis lequel copier
        mov     edi, dest       ; edi = adresse du tampon vers lequel copier
        mov     ecx, sz         ; ecx = nombre d'octets à copier

        cld                     ; éteint le drapeau de direction
        rep     movsb           ; exécute movsb ECX fois

        pop     edi
        pop     esi
        leave
        ret


; fonction _asm_find
; recherche un octet donné en mémoire
; void * asm_find( const void * src, char target, unsigned sz);
; paramètres :
;   src    - pointeur sur le tampon dans lequel chercher
```

```
41 ;   target - octet à rechercher
42 ;   sz     - nombre d'octets dans le tampon
43 ; valeur de retour :
44 ;   si l'élément recherché est trouvé, un pointeur vers sa première occurence dans le tampon
45 ;   est retourné
46 ;   sinon, NULL est retourné
47 ; NOTE : l'élément à rechercher est un octet, mais il est empilé comme une valeur dword.
48 ;       La valeur de l'octet est stockée dans les 8 bits de poids faible.
49 ;
50 %define src    [ebp+8]
51 %define target [ebp+12]
52 %define sz     [ebp+16]
53
54 _asm_find:
55         enter   0,0
56         push    edi
57
58         mov     eax, target     ; al a la valeur recherchée
59         mov     edi, src
60         mov     ecx, sz
61         cld
62
63         repne   scasb           ; scanne jusqu'à ce que ECX == 0 ou [ES:EDI] == AL
64
65         je      found_it        ; si le drapeau zéro est allumé, on a trouvé
66         mov     eax, 0          ; si pas trouvé, retourner un pointeur NULL
67         jmp     short quit
68 found_it:
69         mov     eax, edi
70         dec     eax              ; si trouvé retourner (DI - 1)
71 quit:
72         pop     edi
73         leave
74         ret
75
76
77 ; fonction _asm_strlen
78 ; retourne la taille d'une chaîne
79 ; unsigned asm_strlen( const char * );
80 ; paramètre :
81 ;   src - pointeur sur la chaîne
82 ; valeur de retour :
```

```
83  ;   nombre de caractères dans la chaîne (sans compter le 0 terminal) (dans EAX)
84
85  %define src [ebp + 8]
86  _asm_strlen:
87          enter   0,0
88          push    edi
89
90          mov     edi, src        ; edi = pointeur sur la chaîne
91          mov     ecx, 0FFFFFFFFh ; utilise la plus grande valeur possible de ECX
92          xor     al,al           ; al = 0
93          cld
94
95          repnz   scasb           ; recherche le 0 terminal
96
97  ;
98  ; repnz ira un cran trop loin, donc la longueur vaut FFFFFFFE - ECX,
99  ; pas FFFFFFFF - ECX
100 ;
101         mov     eax,0FFFFFFFEh
102         sub     eax, ecx        ; longueur = 0FFFFFFFEh - ecx
103
104         pop     edi
105         leave
106         ret
107
108 ; fonction _asm_strcpy
109 ; copie une chaîne
110 ; void asm_strcpy( char * dest, const char * src);
111 ; paramètres :
112 ;   dest - pointeur sur la chaine vers laquelle copier
113 ;   src  - pointeur sur la chaîne depuis laquelle copier
114 ;
115 %define dest [ebp + 8]
116 %define src  [ebp + 12]
117 _asm_strcpy:
118         enter   0,0
119         push    esi
120         push    edi
121
122         mov     edi, dest
123         mov     esi, src
124         cld
```

```
125 cpy_loop:
126         lodsb                   ; charge AL & incrémente SI
127         stosb                   ; stocke AL & incrémente DI
128         or      al, al          ; positionne les drapeaux de condition
129         jnz     cpy_loop        ; si l'on est pas après le 0 terminal, on continue
130
131         pop     edi
132         pop     esi
133         leave
134         ret
```

memory.asm

memex.c

```
1  #include <stdio.h>
2
3  #define STR_SIZE 30
4  /* prototypes */
5
6  void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
7  void * asm_find( const void *,
8                   char target, unsigned ) __attribute__((cdecl));
9  unsigned asm_strlen( const char * ) __attribute__((cdecl));
10 void asm_strcpy( char *, const char * ) __attribute__((cdecl));
11
12 int main()
13 {
14   char st1[STR_SIZE] = "chaîne test";
15   char st2[STR_SIZE];
16   char * st;
17   char   ch;
18
19   asm_copy(st2, st1, STR_SIZE); /* copie les 30 caractères de la chaîne */
20    printf ("%s\n", st2);
21
22    printf ("Entrez un caractère : ");  /* recherche un octet dans la chaîne */
23   scanf("%c%*[^\n]", &ch);
24   st = asm_find(st2, ch, STR_SIZE);
25   if ( st )
26     printf ("Trouvé : %s\n", st);
27   else
28     printf ("Pas trouvé\n");
29
```

```
30    st1[0] = 0;
31    printf("Entrez une chaîne :");
32    scanf("%s", st1);
33    printf("longueur = %u\n", asm_strlen(st1));
34
35    asm_strcpy( st2, st1 );       /* copie des données dans la chaîne */
36    printf("%s\n", st2 );
37
38    return 0;
39  }
```

memex.c

Chapitre 6

Virgule Flottante

6.1 Représentation en Virgule Flottante

6.1.1 Nombres binaires non entiers

Lorsque nous avons parlé des systèmes numériques dans le premier chapitre, nous n'avons abordé que le nombres entiers. Evidemment, il est possible de représenter des nombres non entiers dans des bases autres que le décimal. En décimal, les chiffres à droite de la virgule sont associés à des puissances négatives de dix :

$$0,123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

Ce n'est pas étonnant, les nombres binaires fonctionnent de la même façon :

$$0,101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,625$$

Cette idée peut être associée avec les méthodes appliquées aux entiers dans le Chapitre 1 pour convertir un nombre quelconque :

$$110,011_2 = 4 + 2 + 0,25 + 0,125 = 6,375$$

Convertir du binaire vers le décimal n'est pas très difficile non plus. Divisez le nombre en deux parties : entière et décimale. Convertissez la partie entière en binaire en utilisant les méthodes du Chapitre 1, la partie décimale est convertie en utilisant la méthode décrite ci-dessous.

Considérons une partie décimale binaire dont les bits sont notés $a, b, c, \ldots$ Le nombre en binaire ressemble alors à :

$$0, abcdef \ldots$$

Multipliez le nombre par deux. La représentation du nouveau nombre sera :

$$a, bcdef \ldots$$

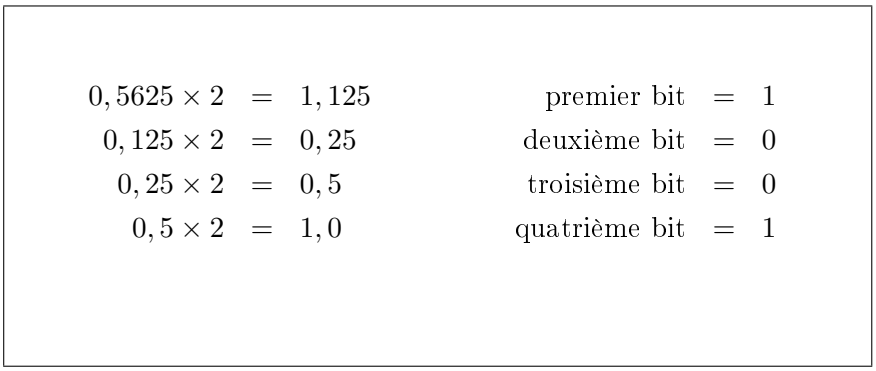

FIGURE 6.1 – Convertir 0,5625 en binaire

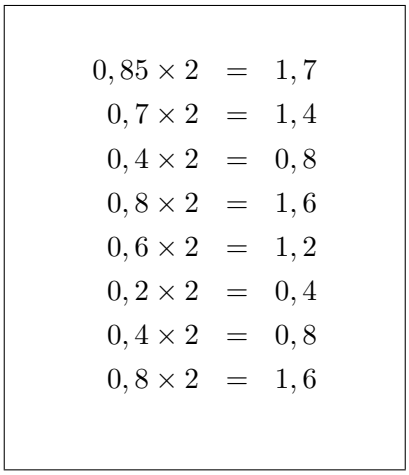

FIGURE 6.2 – Convertir 0,85 en binaire

Notez que le premier bit est maintenant en tête. Remplacez le a par 0 pour obtenir :

$$0, bcdef \ldots$$

et multipliez à nouveau par deux, vous obtenez :

$$b, cdef \ldots$$

Maintenant le deuxième bit (b) est à la première place. Cette procédure peut être répétée jusqu'à ce qu'autant de bits que nécessaires soient trouvés. La Figure 6.1 montre un exemple réel qui converti 0,5625 en binaire. La méthode s'arrête lorsque la partie décimale arrive à zéro.

Prenons un autre exemple, considérons la conversion de 23,85 en binaire, il est facile de convertir la partie entière ($23 = 10111_2$), mais qu'en est-il de la partie décimale $(0,85)$? La Figure 6.2 montre le début de ce calcul. Si

l'on regarde les nombres attentivement, on trouve une boucle infinie ! Cela signifie que 0,85 est un binaire répétitif (par analogie à un décimal répétitif en base 10) [1]. Il y a un motif dans les nombres du calcul. En regardant le motif, on peut voir que $0,85 = 0,11\overline{0110}_2$. Donc, $23,85 = 10111,11\overline{0110}_2$.

Une conséquence importante du calcul ci-dessus est que 23,85 ne peut pas être représenté *exactement* en binaire en utilisant un nombre fini de bits (Tout comme $\frac{1}{3}$ ne peut pas être représenté en décimal avec un nombre fini de chiffres). Comme le montre ce chapitre, les variables `float` et `double` en C sont stockées en binaire. Donc, les valeurs comme 23,85 ne peuvent pas être stockées exactement dans ce genre de variables. Seule une approximation de 23,85 peut être stockée.

Pour simplifier le matériel, les nombres en virgule flottante sont stockés dans un format standard. Ce format utilise la notation scientifique (mais en binaire, en utilisant des puissances de deux, pas de dix). Par exemple, 23,85 ou $10111,11011001100110\ldots_2$ serait stocké sous la forme :

$$1,011111011001100110\ldots \times 2^{100}$$

(où l'exposant (100) est en binaire). Un nombre en virgule flottante *normalisé* a la forme :

$$1,sssssssssssssss \times 2^{eeeeeee}$$

où $1,sssssssssssss$ est la *mantisse* et *eeeeeeee* est l'*exposant*.

6.1.2 Représentation en virgule flottante IEEE

L'IEEE (Institute of Electrical and Electronic Engineers) est une organisation internationale qui a conçu des formats binaires spécifiques pour stocker les nombres en virgule flottante. Ce format est utilisé sur la plupart des ordinateurs (mais pas tous !) fabriqué de nos jours. Souvent, il est supporté par le matériel de l'ordinateur lui-même. Par exemple, les coprocesseurs numériques (ou arithmétiques) d'Intel (qui sont intégrés à tous leurs processeurs depuis le Pentium) l'utilisent. L'IEEE définit deux formats différents avec des précisions différentes : simple et double précision. La simple précision est utilisée pour les variables `float` en C et la double précision pour les variables `double`.

Le coprocesseur arithmétique d'Intel utilise également une troisième précision plus élevée appelée *précision étendue*. En fait toutes les données dans le coprocesseur sont dans ce format. Lorsqu'elles sont stockées en mémoire depuis le coprocesseur, elles sont converties soit en simple, soit en double

1. Cela n'est pas surprenant qu'un nombre puisse être répétitif dans une base, mais pas dans une autre. Pensez à $\frac{1}{3}$, il se répète en décimal, mais en ternaire (base 3) il vaudrait $0,1_3$.

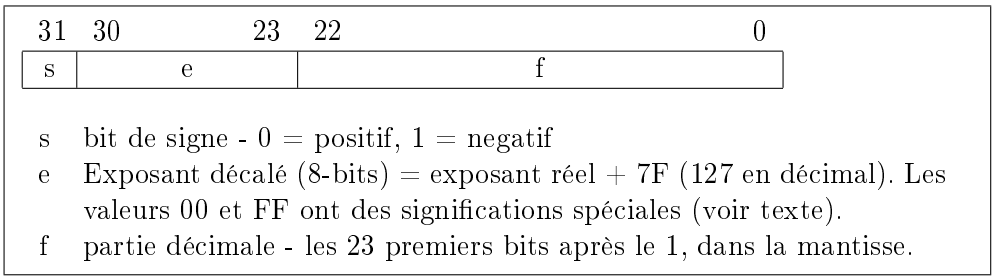

FIGURE 6.3 – Simple précision IEEE

précision automatiquement [2]. La précision étendue utilise un format général légèrement différent des formats float et double de l'IEEE, nous n'en parlerons donc pas ici.

Simple précision IEEE

La virgule flottante simple précision utilise 32 bits pour encoder le nombre. Elle est généralement précise jusqu'à 7 chiffres significatifs. Les nombres en virgule flottante sont généralement stockés dans un format beaucoup plus compliqué que celui des entiers. La Figure 6.3 montre le format de base d'un nombre en simple précision IEEE. Ce format a plusieurs inconvénients. Les nombres en virgule flottante n'utilisent pas la représentation en complément à deux pour les nombres négatifs. Ils utilisent une représentation en grandeur signée. Le bit 31 détermine le signe du nombre.

L'exposant binaire n'est pas stocké directement. A la place, la somme de l'exposant et de 7F est stockée dans les bits 23 à 30, Cet *exposant décalé* est toujours positif ou nul.

La partie décimale suppose une mantisse normalisée (de la forme $1,sssssssss$). Comme le premier bit est toujours un 1, le 1 de gauche n'est *pas stocké !* Cela permet le stockage d'un bit additionnel à la fin et augmente donc légèrement la précision. Cette idée est appelée *représentation en un masqué*.

Il faut toujours garder à l'esprit que les octets 41 BE CC CD peuvent être interprétés de façons différentes selon ce qu'en fait le programme ! Vus comme un nombre en virgule flottante en simple précision, ils représentent 23,850000381, mais vus comme un entier double mot, ils représentent 1,103,023,309 ! Le processeur ne sait pas quelle est la bonne interprétation !

Comment serait stocké 23,85 ? Tout d'abord, il est positif, donc le bit de signe est à 0, Ensuite, l'exposant réel est 4, donc l'exposant décalé est $7\text{F} + 4 = 83_{16}$. Enfin, la fraction vaut 01111101100110011001100 (souvenez vous que le un de tête est masqué). En les mettant bout à bout (pour clarifier les différentes sections du format en virgule flottante, le bit de signe et la fraction on été soulignés et les bits ont été regroupés en groupes de 4 bits) :

$$\underline{0}\,100\ 0001\ 1\,\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41\text{BECCCC}_{16}$$

2. Le type `long double` de certains compilateurs (comme celui de Borland) utilise cette précision étendue. Par contre, d'autres compilateurs utilisent la double précision à la fois pour les `double` et `long double` (C'est autorisé par le C ANSI).

$e = 0$ et $f = 0$	indique le nombre zéro (qui ne peut pas être normalisé). Notez qu'il y a +0 et -0.
$e = 0$ et $f \neq 0$	indique un *nombre dénormalisé*. Nous en parlerons dans la section suivante.
$e =$ FF et $f = 0$	indique l'infini (∞). Il y a un infini positif et un infini négatif.
$e =$ FF et $f \neq 0$	indique un résultat indéfini, appelé *NaN* (Not a Number, pas un nombre).

TABLE 6.1 – Valeurs spéciales de f et e

Cela ne fait pas exactement 23,85 (puisqu'il s'agit d'un binaire répétitif). Si l'on convertit le chiffre ci-dessus en décimal, on constate qu'il vaut approximativement 23,849998474, ce nombre est très proche de 23,85 mais n'est pas exactement le même. En réalité, en C, 23,85 ne serait pas représenté exactement comme ci-dessus. Comme le bit le plus à gauche qui a été supprimé de la représentation exacte valait 1, le dernier bit est arrondi à 1, donc 23,85 serait représenté par 41 BE CC CD en hexa en utilisant la simple précision. En décimal, ce chiffre vaut 23,850000381 ce qui est une meilleure approximation de 23,85.

Comment serait représenté -23,85 ? Il suffit de changer le bit de signe : C1 BE CC CD. Ne prenez *pas* le complément à deux !

Certaines combinaisons de e et f ont une signification spéciale pour les flottants IEEE. Le Tableau 6.1 décrit ces valeurs. Un infini est produit par un dépassement de capacité ou une division par zéro. Un résultat indéfini est produit par une opération invalide comme rechercher la racine carée d'un nombre négatif, additionner deux infinis, *etc.*

Les nombres en simple précision normalisés peuvent prendre des valeurs de $1,0 \times 2^{-126}$ ($\approx 1,1755 \times 10^{-35}$) à $1,11111\ldots \times 2^{127}$ ($\approx 3,4028 \times 10^{35}$).

Nombres dénormalisés

Les nombres dénormalisés peuvent être utilisés pour représenter des nombres avec des grandeurs trop petites à normaliser (*i.e.* inférieures à $1,0 \times 2^{-126}$). Par exemple, considérons le nombre $1,001_2 \times 2^{-129}$ ($\approx 1,6530 \times 10^{-39}$). Dans la forme normalisée, l'exposant est trop petit. Par contre, il peut être représenté sous une forme non normalisée : $0,01001_2 \times 2^{-127}$. Pour stocker ce nombre, l'exposant décalé est positionné à 0 (voir Tableau 6.1) et la partie décimale est la mantisse complète du nombre écrite comme un produit avec 2^{-127} (*i.e.* tous les bits sont stockés, y compris le un à gauche du point décimal). La représentation de $1,001 \times 2^{-129}$ est donc :

0 000 0000 0 001 0010 0000 0000 0000 0000

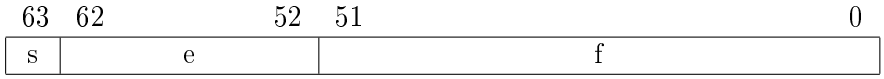

FIGURE 6.4 – Double précision IEEE

Double précision IEEE

La double précision IEEE utilise 64 bits pour représenter les nombres et est précise jusqu'à environ 15 chiffres significatifs. Comme le montre la Figure 6.4, le format de base est très similaire à celui de la simple précision. Plus de bits sont utilisés pour l'exposant décalé (11) et la partie décimale (52).

La plage de valeurs plus grande pour l'exposant décalé a deux conséquences. La première est qu'il est calculé en faisant la somme de l'exposant réel et de 3FF (1023) (pas 7F comme pour la simple précision). Deuxièmement, une grande plage d'exposants réels (et donc une plus grande plage de grandeurs) est disponible. Les grandeurs en double précision peuvent prendre des valeurs entre environ 10^{-308} et 10^{308}.

C'est la plus grande taille du champ réservé à la partie décimale qui est reponsable de l'augmentation du nombre de chiffres significatifs pour les valeurs doubles.

Pour illustrer cela, reprenons 23,85. L'exposant décalé sera 4+3FF = 403 en hexa. Donc, la représentation double sera :

0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

ou 40 37 D9 99 99 99 99 9A en hexa. Si l'on reconvertit ce nombre en décimal, on trouve 23,8500000000000014 (il y a 12 zéros !) ce qui est une approximation de 23,85 nettement meilleure.

La double précision a les mêmes valeurs spéciales que la simple précision [3]. Les nombres dénormalisés sont également très similaires. La seule différence principale est que les nombres doubles dénormalisés utilisent 2^{-1023} au lieu de 2^{-127}.

6.2 Arithmétique en Virgule Flottante

L'arithmétique en virgule flottante sur un ordinateur est différente de celle des mathématiques. En mathématiques, tous les nombres peuvent être

3. La seule différence est que pour l'infini et les valeurs indéfinies, l'exposant décalé vaut 7FF et non pas FF.

considérés comme exacts. Comme nous l'avons montré dans la section précédente, sur un ordinateur beaucoup de nombres ne peuvent pas être représentés exactement avec un nombre fini de bits. Tous les calculs sont effectués avec une précision limitée. Dans les exemples de cette section, des nombres avec une mantisse de 8 bits seront utilisés pour plus de simplicité.

6.2.1 Addition

Pour additionner deux nombres en virgule flottante, les exposants doivent être égaux. S'ils ne le sont pas déjà, alors, il faut les rendre égaux en décalant la mantisse du nombre ayant le plus petit exposant. Par exemple, considérons $10,375 + 6,34375 = 16,71875$ ou en binaire :

$$\begin{array}{rr} & 1,0100110 \times 2^3 \\ + & 1,1001011 \times 2^2 \\ \hline \end{array}$$

Ces deux nombres n'ont pas le même exposant, donc on décale la mantisse pour rendre les exposants égaux, puis on effectue l'addition :

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

Notez que le décalage de $1,1001011 \times 2^2$ supprime le un de tête et arrondi plus tard le résultat en $0,1100110 \times 2^3$. Le résultat de l'addition, $10,0001100 \times 2^3$ (ou $1,00001100 \times 2^4$), est égal à $10000,110_2$ ou 16,75, ce n'est *pas* égal à la réponse exacte (16,71875) ! Il ne s'agit que d'une approximation due aux erreurs d'arrondi du processus d'addition.

Il est important de réaliser que l'arithmétique en virgule flottante sur un ordinateur (ou une calculatrice) est toujours une approximation. Les lois des mathématiques ne fonctionnent pas toujours avec les nombres en virgule flottante sur un ordinateur. Les mathématiques supposent une précision infinie qu'aucun ordinateur ne peut atteindre. Par exemple, les mathématiques nous apprennent que $(a + b) - b = a$; cependant, ce n'est pas forcément exactement vrai sur un ordinateur !

6.2.2 Soustraction

La soustraction fonctionne d'une façon similaire à l'addition et souffre des mêmes problèmes. Par exemple, considérons $16,75 - 15,9375 = 0,8125$:

$$\begin{array}{rr} & 1,0000110 \times 2^4 \\ - & 1,1111111 \times 2^3 \\ \hline \end{array}$$

Décaler $1,1111111 \times 2^3$ donne (en arrondissant) $1,0000000 \times 2^4$

$$\begin{array}{rr} & 1,0000110 \times 2^4 \\ - & 1,0000000 \times 2^4 \\ \hline & 0,0000110 \times 2^4 \end{array}$$

$0,0000110 \times 2^4 = 0,11_2 = 0,75$ ce qui n'est pas exactement correct.

6.2.3 Multiplication et division

Pour la multiplication, les mantisses sont multiplées et les exposants sont additionnés. Considérons $10,375 \times 2,5 = 25,9375$:

$$\begin{array}{rr} & 1,0100110 \times 2^3 \\ \times & 1,0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1,10011111000000 \times 2^4 \end{array}$$

Bien sûr, le résultat réel serait arrondi à 8 bits pour donner :

$$1,1010000 \times 2^4 = 11010,000_2 = 26$$

La division est plus compliquée, mais souffre des mêmes problèmes avec des erreurs d'arrondi.

6.2.4 Conséquences sur la programmation

Le point essentiel de cette section est que les calculs en virgule flottante ne sont pas exacts. Le programmeur doit en être conscient. Une erreur courante que les programmeurs font avec les nombres en virgule flottante est de les comparer en supposant qu'un calcul est exact. Par exemple, considérons une fonction appelée `f(x)` qui effectue un calcul complexe et un programme qui essaie de trouver la racine de la fonction[4]. On peut être tenté d'utiliser l'expression suivante pour vérifier si `x` est une racine :

```
if ( f(x) == 0.0 )
```

Mais, que se passe-t-il si `f(x)` retourne 1×10^{-30} ? Cela signifie très probablement que `x` est une *très* bonne approximation d'une racine réelle ; cependant, l'égalité ne sera pas vérifiée. Il n'y a peut être aucune valeur en virgule flottante IEEE de `x` qui renvoie exactement zéro, en raison des erreurs d'arrondi dans `f(x)`.

Une meilleur méthode serait d'utiliser :

4. La racine d'une fonction est une valeur x telle que $f(x) = 0$

```
if ( fabs(f(x)) < EPS )
```

où EPS est une macro définissant une très petite valeur positive (du genre 1×10^{-10}). Cette expression est vraie dès que f(x) est très proche de zéro. En général, pour comparer une valeur en virgule flottante (disons x) à une autre (y), on utilise :

```
if ( fabs(x − y)/fabs(y) < EPS )
```

6.3 Le Coprocesseur Arithmétique

6.3.1 Matériel

Les premiers processeurs Intel n'avaient pas de support matériel pour les opérations en virgule flottante. Cela ne signifie pas qu'ils ne pouvaient pas effectuer de telles opérations. Cela signifie seulement qu'elles devaient être réalisées par des procédures composées de beaucoup d'instructions qui n'étaient pas en virgule flottante. Pour ces systèmes, Intel fournissait une puce appelée *coprocesseur mathématique*. Un coprocesseur mathématique a des instructions machine qui effectuent beaucoup d'opérations en virgule flottante beaucoup plus rapidement qu'en utilisant une procédure logicielle (sur les premiers processeurs, au moins 10 fois plus vite !). Le coprocesseur pour le 8086/8088 s'appelait le 8087, pour le 80286, il y avait un 80287 et pour le 80386, un 80387 ; le processeur 80486DX intégrait le coprocesseur mathématique dans le 80486 lui-même [5]. Depuis le Pentium, toutes les générations de processeurs 80x86 on un coprocesseur mathématique intégré ; cependant, on le programme toujours comme s'il s'agissait d'une unité séparée. Même les systèmes plus anciens sans coprocesseur peuvent installer un logiciel qui émule un coprocesseur mathématique. Cet émulateur est automatiquement activé lorsqu'un programme exécute une instruction du coprocesseur et lance la procédure logicielle qui produit le même résultat que celui qu'aurait donné le coprocesseur (bien que cela soit plus lent, bien sûr).

Le coprocesseur arithmétique a huit registres de virgule flottante. Chaque registre contient 80 bits de données. Les nombres en virgule flottante sont *toujours* stockés sous forme de nombres 80 bits en précision étendue dans ces registres. Les registres sont appelés `ST0`, `ST1`, `ST2`, ... `ST7`. Les registres en virgule flottante sont utilisés différemment des registres entiers du processeur principal. Ils sont organisés comme une *pile*. Souvenez vous qu'une pile est une liste *Last-In First-Out* (LIFO, dernier entré, premier sorti). `ST0` fait toujours référence à la valeur au sommet de la pile. Tous les nouveaux nombres

5. Cependant, le 80486SX n'avait *pas* de coprocesseur intégré. Il y avait une puce 80487SX pour ces machines.

sont ajoutés au sommet de la pile. Les nombres existants sont décalés vers le bas pour faire de la place au nouveau.

Il y a également un registre de statut dans le coprocesseur arithmétique. Il a plusieurs drapeaux. Seuls les 4 drapeaux utilisés pour les comparaisons seront traités : C_0, C_1, C_2 et C_3. Leur utilisation est traitée plus tard.

6.3.2 Instructions

Pour faciliter la distinction entre les instructions du processeur normal et celles du coprocesseur, tous les mnémoniques du coprocesseur commencent par un `F`.

Chargement et stockage

Il y a plusieurs instructions qui chargent des données au sommet de la pile du coprocesseur :

`FLD` *source*	Charge un nombre en virgule flottante depuis la mémoire vers le sommet de la pile. La *source* peut être un nombre en simple, double ou précision étendue ou un registre du coprocesseur.
`FILD` *source*	Lit un *entier* depuis la mémoire, le convertit en flottant et stocke le résultat au sommet de la pile. La *source* peut être un mot, un double mot ou un quadruple mot.
`FLD1`	Stocke un un au sommet de la pile.
`FLDZ`	Stocke un zéro au sommet de la pile.

Il y a également plusieurs instructions qui déplacent les données depuis la pile vers la mémoire. Certaines de ces instruction retirent le nombre de la pile en même temps qu'elles le déplacent.

`FST` *dest* — Stocke le sommet de la pile (`ST0`) en mémoire. La *destination* peut être un nombre en simple ou double précision ou un registre du coprocesseur.

`FSTP` *dest* — Stocke le sommet de la pile en mémoire, exactement comme `FST` ; cependant, une fois le nombre stocké, sa valeur est retirée de la pile. La *destination* peut être un nombre en simple, double ou précision étendue ou un registre du coprocesseur.

`FIST` *dest* — Stocke la valeur au sommet de la pile convertie en entier en mémoire. La *destination* peut être un mot ou un double mot. La pile en elle-même reste inchangée. La façon dont est converti le nombre en entier dépend de certains bits dans le *mot de contrôle* du coprocesseur. Il s'agit d'un registre d'un mot spécial (pas en virgule flottante) qui contrôle le fonctionnement du coprocesseur. Par défaut, le mot de contrôle est initialisé afin qu'il arrondisse à l'entier le plus proche lorsqu'il convertit vers un entier. Cependant, les instructions `FSTCW` (Store Control Word, stocker le mot de contrôle) et `FLDCW` (Load Control Word, charger le mot de contrôle) peuvent être utilisées pour changer ce comportement.

`FISTP` *dest* — Identique à `FIST` sauf en deux points. Le sommet de la pile est supprimé et la *destination* peut également être un quadruple mot.

Il y a deux autres instructions qui peuvent placer ou retirer des données de la pile.

`FXCH ST`*n* — échange les valeurs de `ST0` et `ST`*n* sur la pile (où *n* est un numéro de registre entre 1 et 7).

`FFREE ST`*n* — libère un registre sur la pile en le marquant comme inutilisé ou vide.

Addition et soustraction

Chacune des instructions d'addition calcule la somme de `ST0` et d'un autre opérande. Le résultat est toujours stocké dans un registre du coprocesseur.

```
1  segment .bss
2  array        resq SIZE
3  sum          resq 1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                  ; ST0 = 0
9  lp:
10       fadd   qword [esi]    ; ST0 += *(esi)
11       add    esi, 8         ; passe au double suivant
12       loop   lp
13       fstp   qword sum      ; stocke le résultat dans sum
```

FIGURE 6.5 – Exemple de somme d'un tableau

`FADD` *source*	`ST0 +=` *source*. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FADD` *dest*`, ST0`	*dest* `+= ST0`. La *destination* peut être n'importe quel registre du coprocesseur.
`FADDP` *dest* ou `FADDP` *dest*`, ST0`	*dest* `+= ST0` puis ST0 est retiré de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FIADD` *source*	`ST0 += (float)` *source*. Ajoute un entier à `ST0`. La *source* doit être un mot ou un double mot en mémoire.

Il y a deux fois plus d'instructions pour la soustraction que pour l'addition car l'ordre des opérandes est important pour la soustraction (*i.e.* $a+b = b+a$, mais $a-b \neq b-a$!). Pour chaque instruction, il y a un mirroir qui effectue la soustraction dans l'ordre inverse. Ces instructions inverses se finissent toutes soit par `R` soit par `RP`. La Figure 6.5 montre un court extrait de code qui ajoute les éléments d'un tableau de doubles. Au niveau des lignes 10 et 13, il faut spécifier la taille de l'opérande mémoire. Sinon l'assembleur ne saurait pas si l'opérande mémoire est un float (dword) ou un double (qword).

`FSUB` *source*	`ST0 -=` *source*. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FSUBR` *source*	`ST0 =` *source* `- ST0`. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FSUB` *dest*`, ST0`	*dest* `-= ST0`. La *destination* peut être n'importe quel registre du coprocesseur.
`FSUBR` *dest*`, ST0`	*dest* `= ST0 -` *dest*. La *destination* peut être n'importe quel registre du coprocesseur.
`FSUBP` *dest* ou `FSUBP` *dest*`, ST0`	*dest* `-= ST0` puis retire ST0 de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FSUBRP` *dest* ou `FSUBRP` *dest*`, ST0`	*dest* `= ST0 -` *dest* puis retire ST0 de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FISUB` *source*	`ST0 -= (float)` *source*. Soustrait un entier de `ST0`. La *source* doit être un mot ou un double mot en mémoire.
`FISUBR` *source*	`ST0 = (float)` *source* `- ST0`. Soustrait `ST0` d'un entier. La *source* doit être un mot ou un double mot en mémoire.

Multiplication et division

Les instructions de multiplication sont totalement analogues à celles d'addition.

`FMUL` *source*	`ST0 *=` *source*. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FMUL` *dest*`, ST0`	*dest* `*= ST0`. La *destination* peut être n'importe quel registre du coprocesseur.
`FMULP` *dest* ou `FMULP` *dest*`, ST0`	*dest* `*= ST0` puis retire ST0 de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FIMUL` *source*	`ST0 *= (float)` *source*. Multiplie un entier avec `ST0`. La *source* peut être un mot ou un double mot en mémoire.

Ce n'est pas étonnant, les instructions de division sont analogues à celles de soustraction. La division par zéro donne l'infini.

`FDIV` *source*	`ST0 /=` *source*. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FDIVR` *source*	`ST0 =` *source* `/ ST0`. La *source* peut être n'importe quel registre du coprocesseur ou un nombre en simple ou double précision en mémoire.
`FDIV` *dest*`, ST0`	*dest* `/= ST0`. La *destination* peut être n'importe quel registre du coprocesseur.
`FDIVR` *dest*`, ST0`	*dest* `= ST0 /` *dest*. La *destination* peut être n'importe quel registre du coprocesseur.
`FDIVP` *dest* ou `FDIVP` *dest*`, ST0`	*dest* `/= ST0` puis retire ST0 de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FDIVRP` *dest* ou `FDIVRP` *dest*`, ST0`	*dest* `= ST0 /` *dest* puis retire ST0 de la pile. La *destination* peut être n'importe quel registre du coprocesseur.
`FIDIV` *source*	`ST0 /= (float)` *source*. Divise `ST0` par un entier. La *src* doit être un mot ou un double mot en mémoire.
`FIDIVR` *source*	`ST0 = (float)` *source* `/ ST0`. Divise un entier par `ST0`. La *source* doit être un mot ou un double mot en mémoire.

Comparaisons

Le coprocesseur effectue également des comparaisons de nombres en virgule flottante. La famille d'instructions `FCOM` est faite pour ça.

`FCOM` *source*	compare `ST0` et *source*. La *source* peut être un registre du coprocesseur ou un float ou un double en mémoire.
`FCOMP` *source*	compare `ST0` et *source*, puis retire ST0 de la pile. La *source* peut être un registre du coprocesseur ou un float ou un double en mémoire.
`FCOMPP`	compare `ST0` et `ST1`, puis retire ST0 et ST1 de la pile.
`FICOM` *source*	compare `ST0` et `(float)` *source*. La *source* peut être un entier sur un mot ou un double mot en mémoire.
`FICOMP` *source*	compare `ST0` et `(float)`*source*, puis retire ST0 de la pile. La *source* peut être un entier sur un mot ou un double mot en mémoire.
`FTST`	compare `ST0` et 0,

Ces instructions changent les bits C_0, C_1, C_2 et C_3 du registre de statut du coprocesseur. Malheureusement, il n'est pas possible pour le processeur d'accéder à ces bits directement. Les instructions de branchement conditionnel utilisent le registre FLAGS, pas le registre de statut du coprocesseur. Cependant, il est relativement simple de transférer les bits du mot de statut dans les bits correspondants du registre FLAGS en utilisant quelques ins-

```
1  ;     if ( x > y )
2  ;
3        fld    qword [x]       ; ST0 = x
4        fcomp  qword [y]       ; compare ST0 et y
5        fstsw  ax              ; place les bits C dans FLAGS
6        sahf
7        jna    else_part       ; si x non < y goto else_part
8  then_part:
9        ; code si vrai
10       jmp    end_if
11 else_part:
12       ; code si faux
13 end_if:
```

FIGURE 6.6 – Exemple de comparaison

tructions nouvelles :

`FSTSW` *destination*	Stocke le mot de statut du coprocesseur soit dans un mot en mémoire soit dans le registre AX.
`SAHF`	Stocke le registre AH dans le registre FLAGS.
`LAHF`	Charge le registre AH avec les bits du registre FLAGS.

La Figure 6.6 montre un court extrait de code en exemple. Les lignes 5 et 6 transfèrent les bits C_0, C_1, C_2 et C_3 du mot de statut du coprocesseur dans le registre FLAGS. Les bits sont transférés de façon a être identiques au résultat d'une comparaison de deux entiers *non signés*. C'est pourquoi la ligne 7 utilise une instruction `JNA`.

Les Pentium Pro (et les processeurs plus récents (Pentium II and III)) supportent deux nouveaux opérateurs de comparaison qui modifient directement le registre FLAGS du processeur.

`FCOMI` *source*	compare `ST0` et *source*. La *source* doit être un registre du coprocesseur.
`FCOMIP` *source*	compare `ST0` et *source*, puis retire ST0 de la pile. La *source* doit être un registre du coprocesseur.

La Figure 6.7 montre une sous-routine qui trouve le plus grand de deux doubles en utilisant l'instruction `FCOMIP`. Ne confondez pas ces instructions avec les fonctions de comparaisons avec un entier (`FICOM` et `FICOMP`).

Instructions diverses

Cette section traite de diverses autres instructions fournies par le coprocesseur.

`FCHS`	`ST0 = - ST0` Change le signe de `ST0`
`FABS`	`ST0` $= \|$`ST0`$\|$ Extrait la valeur absolue de `ST0`
`FSQRT`	`ST0` $= \sqrt{$`ST0`$}$ Extrait la racine carrée de `ST0`
`FSCALE`	`ST0` $=$ `ST0` $\times 2^{\lfloor$`ST1`$\rfloor}$ multiplie `ST0` par une puissance de 2 rapidement. `ST1` n'est pas retiré de la pile du coprocesseur. La Figure 6.8 montre un exemple d'utilisation de cette instruction.

6.3.3 Exemples

6.3.4 Formule quadratique

Le premier exemple montre comment la formule quadratique peut être codée en assembleur. Souvenez vous, la formule quadratique calcule les solutions de l'équation :

$$ax^2 + bx + c = 0$$

La formule donne deux solutions pour x: x_1 et x_2.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

L'expression sous la racine carrée ($b^2 - 4ac$) est appelée le *déterminant*. Sa valeur est utile pour déterminer laquelle des trois possibilités suivantes est vérifiée pour les solutions.

1. Il n'y a qu'une seule solution double. $b^2 - 4ac = 0$
2. Il y a deux solutions réelles. $b^2 - 4ac > 0$
3. Il y a deux solutions complexes. $b^2 - 4ac < 0$

Voici un court programme C qui utilise la sous-routine assembleur :

quadt.c

```
1  #include <stdio.h>
2
3  int quadratic( double, double, double, double *, double *);
4
5  int main()
6  {
7    double a,b,c, root1, root2;
8
9     printf ("Entrez a, b, c : ");
10    scanf("%lf %lf %lf", &a, &b, &c);
11    if (quadratic( a, b, c, &root1, &root2) )
12       printf ("racines : %.10g %.10g\n", root1, root2);
```

```
13    else
14       printf("pas de racine  réelle \n");
15    return 0;
16 }
```

quadt.c

Voici la routine assembleur :

quad.asm

```
1  ; fonction quadratic
2  ; trouve les solutions à l'équation quadratique :
3  ;       a*x**2 + b*x + c = 0
4  ; prototype C :
5  ;   int quadratic( double a, double b, double c,
6  ;                  double * root1, double *root2 )
7  ; Paramètres:
8  ;   a, b, c - coefficients des puissances de l'équation quadratique (voir ci-dessus)
9  ;   root1   - pointeur vers un double où stocker la première racine
10 ;   root2   - pointeur vers un double où stocker la deuxième racine
11 ; Valeur de retour :
12 ;   retourne 1 si des racines réelles sont trouvées, sinon 0
13
14 %define a               qword [ebp+8]
15 %define b               qword [ebp+16]
16 %define c               qword [ebp+24]
17 %define root1           dword [ebp+32]
18 %define root2           dword [ebp+36]
19 %define disc            qword [ebp-8]
20 %define one_over_2a     qword [ebp-16]
21
22 segment .data
23 MinusFour       dw      -4
24
25 segment .text
26         global  _quadratic
27 _quadratic:
28         push    ebp
29         mov     ebp, esp
30         sub     esp, 16         ; alloue 2 doubles (disc & one_over_2a)
31         push    ebx             ; on doit sauvegarder l'ebx original
32
33         fild    word [MinusFour]; pile : -4
```

```
34          fld     a               ; pile : a, -4
35          fld     c               ; pile : c, a, -4
36          fmulp   st1             ; pile : a*c, -4
37          fmulp   st1             ; pile : -4*a*c
38          fld     b
39          fld     b               ; pile : b, b, -4*a*c
40          fmulp   st1             ; pile : b*b, -4*a*c
41          faddp   st1             ; pile : b*b - 4*a*c
42          ftst                    ; compare avec 0
43          fstsw   ax
44          sahf
45          jb      no_real_solutions ; si < 0, pas de solution réelle
46          fsqrt                   ; pile : sqrt(b*b - 4*a*c)
47          fstp    disc            ; stocke et décale la pile
48          fld1                    ; pile : 1,0
49          fld     a               ; pile : a, 1,0
50          fscale                  ; pile : a * 2(1,0) = 2*a, 1
51          fdivp   st1             ; pile : 1/(2*a)
52          fst     one_over_2a     ; pile : 1/(2*a)
53          fld     b               ; pile : b, 1/(2*a)
54          fld     disc            ; pile : disc, b, 1/(2*a)
55          fsubrp  st1             ; pile : disc - b, 1/(2*a)
56          fmulp   st1             ; pile : (-b + disc)/(2*a)
57          mov     ebx, root1
58          fstp    qword [ebx]     ; stocke dans *root1
59          fld     b               ; pile : b
60          fld     disc            ; pile : disc, b
61          fchs                    ; pile : -disc, b
62          fsubrp  st1             ; pile : -disc - b
63          fmul    one_over_2a     ; pile : (-b - disc)/(2*a)
64          mov     ebx, root2
65          fstp    qword [ebx]     ; stocke dans *root2
66          mov     eax, 1          ; la valeur de retour est 1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; la valeur de retour est 0
71
72  quit:
73          pop     ebx
74          mov     esp, ebp
75          pop     ebp
```

```
76          ret
```
quad.asm

6.3.5 Lire un tableau depuis un fichier

Dans cet exemple, une routine assembleur lit des doubles depuis un fichier. Voici un court programme de test en C :

readt.c

```
1  /*
2   * Ce programme teste la procédure assembleur 32 bits read_doubles().
3   * Il lit des doubles depuis stdin ( Utilisez une redirection pour lire depuis un fichier ).
4   */
5  #include <stdio.h>
6  extern int read_doubles( FILE *, double *, int  );
7  #define MAX 100
8
9  int main()
10 {
11   int i,n;
12   double a[MAX];
13
14   n = read_doubles(stdin, a, MAX);
15
16   for( i=0; i < n; i++ )
17     printf ("%3d %g\n", i, a[i]);
18   return 0;
19 }
```

readt.c

Voici la routine assembleur

read.asm

```
1  segment .data
2  format  db      "%lf", 0        ; format pour fscanf()
3
4  segment .text
5          global  _read_doubles
6          extern  _fscanf
7
8  %define SIZEOF_DOUBLE   8
9  %define FP              dword [ebp + 8]
```

```
10 %define ARRAYP          dword [ebp + 12]
11 %define ARRAY_SIZE      dword [ebp + 16]
12 %define TEMP_DOUBLE     [ebp - 8]
13
14 ;
15 ; fonction _read_doubles
16 ; prototype C :
17 ;   int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; Cette fonction lit des doubles depuis un fichier texte dans un tableau, jusqu'à
19 ; EOF ou que le tableau soit plein.
20 ; Paramètres :
21 ;   fp         - FILE pointeur à partir duquel lire (doit être ouvert en lecture)
22 ;   arrayp     - pointeur vers le tableau de double vers lequel lire
23 ;   array_size - nombre d'éléments du tableau
24 ; Valeur de retour :
25 ;   nombre de doubles stockés dans le tableau (dans EAX)
26
27 _read_doubles:
28         push    ebp
29         mov     ebp,esp
30         sub     esp, SIZEOF_DOUBLE      ; définit un double sur la pile
31
32         push    esi                     ; sauve esi
33         mov     esi, ARRAYP             ; esi = ARRAYP
34         xor     edx, edx                ; edx = indice du tableau (initialement 0)
35
36 while_loop:
37         cmp     edx, ARRAY_SIZE         ; edx < ARRAY_SIZE ?
38         jnl     short quit              ; si non, quitte la boucle
39 ;
40 ; appelle fscanf() pour lire un double dans TEMP_DOUBLE
41 ; fscanf() peut changer edx, donc on le sauvegarde
42 ;
43         push    edx                     ; sauve edx
44         lea     eax, TEMP_DOUBLE
45         push    eax                     ; empile &TEMP_DOUBLE
46         push    dword format            ; emplie &format
47         push    FP                      ; emplie file pointer
48         call    _fscanf
49         add     esp, 12
50         pop     edx                     ; restaure edx
51         cmp     eax, 1                  ; fscanf a retourné 1?
```

```
52          jne     short quit              ; si non, on quitte la boucle
53
54  ;
55  ; copie TEMP_DOUBLE dans ARRAYP[edx]
56  ; (Les 8 octets du double sont copiés en deux fois 4 octets)
57  ;
58          mov     eax, [ebp - 8]
59          mov     [esi + 8*edx], eax      ; copie des 4 octets de poids faible
60          mov     eax, [ebp - 4]
61          mov     [esi + 8*edx + 4], eax  ; copie des 4 octets de poids fort
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67          pop     esi                     ; restaure esi
68
69          mov     eax, edx                ; stocke la valeur de retour dans eax
70
71          mov     esp, ebp
72          pop     ebp
73          ret
```

read.asm

6.3.6 Renchercher les nombres premiers

Ce dernier exemple recherche les nombres premiers, une fois de plus. Cette implémentation est plus efficace que la précédente. Elle stocke les nombres premiers déjà trouvés dans un tableau et ne divise que par ceux-ci au lieu de diviser par tous les nombres impairs pour trouver de nouveaux premiers.

Une autre différence est qu'il calcule la racine carrée du candidat pour le prochain premier afin de déterminer le moment où il peut s'arrêter de rechercher des facteurs. Il altère le fonctionnement du coprocesseur afin que lorsqu'il stocke la racine sous forme d'entier, il la tronque au lieu de l'arrondir. Ce sont les bits 10 et 11 du mot de contrôle qui permettent de paramétrer cela. Ces bits sont appelées les bits RC (Rounding Control, contrôle d'arrondi). S'ils sont tous les deux à 0 (par défaut), le coprocesseur arrondit lorsqu'il convertit vers un entier. S'ils sont tous les deux à 1, le copresseur tronque les conversions en entiers. Notez que la routine fait attention à sauvegarder le mot de contrôle original et à le restaurer avant de quitter.

Voici le programme C pilote :

fprime.c

```
#include <stdio.h>
#include <stdlib.h>
/*
 * fonction find_primes
 * recherche le nombre indiqué de nombres premiers
 * Paramètres:
 *   a – tableau pour contenir les nombres premiers
 *   n – nombre de nombres premiers à trouver
 */
extern void find_primes( int * a, unsigned n );

int main()
{
  int statut;
  unsigned i;
  unsigned max;
  int * a;

   printf ("Combien de nombres premiers voulez vous trouver ? ");
  scanf("%u", &max);

  a = calloc( sizeof(int), max);

  if ( a ) {

    find_primes(a,max);

    /* affiche les 20 derniers nombres premiers trouvés */
    for( i= ( max > 20 ) ? max – 20 : 0; i < max; i++ )
       printf ("%3d %d\n", i+1, a[i]);

    free (a);
    statut = 0;
  }
  else {
     fprintf (stderr, "Impossible de créer un tableau de %u entiers\n", max);
    statut = 1;
  }

  return statut;
}
```

fprime.c

Voici la routine assembleur :

prime2.asm

```
segment .text
        global  _find_primes
;
; fonction find_primes
; trouve le nombre indiqué de nombre premiers
; Paramètres:
;   array  - tableau pour contenir les nombres premiers
;   n_find - nombre de nombres premiers à trouver
; Prototype C :
;extern void find_primes( int * array, unsigned n_find )
;
%define array         ebp + 8
%define n_find        ebp + 12
%define n             ebp - 4           ; Nombre de nombres premiers trouvés
%define isqrt         ebp - 8           ; racine du candidat
%define orig_cntl_wd  ebp - 10          ; mot de contrôle original
%define new_cntl_wd   ebp - 12          ; nouveau mot de contrôle

_find_primes:
        enter   12,0                    ; fait de la place pour les variables locales

        push    ebx                     ; sauvegarde les variables registre éventuelles
        push    esi

        fstcw   word [orig_cntl_wd]     ; récupère le mot de contrôle courant
        mov     ax, [orig_cntl_wd]
        or      ax, 0C00h               ; positionne les bits d'arrondi à 11 (tronquer)
        mov     [new_cntl_wd], ax
        fldcw   word [new_cntl_wd]

        mov     esi, [array]            ; esi pointe sur array
        mov     dword [esi], 2          ; array[0] = 2
        mov     dword [esi + 4], 3      ; array[1] = 3
        mov     ebx, 5                  ; ebx = guess = 5
        mov     dword [n], 2            ; n = 2
;
; Cette boucle externe trouve un nouveau nombre premier à chaque itération qu'il
; ajoute à la fin du tableau. Contrairement au programme de recherche de nombres
```

```
39 ; premiers précédent, cette fonction ne détermine pas la primauté du nombre en
40 ; divisant par tous les nombres impairs. Elle ne divise que par les nombres
41 ; premiers qu'elle a déjà trouvé (c'est pourquoi ils sont stockés dans un tableau).
42 ;
43 while_limit:
44         mov     eax, [n]
45         cmp     eax, [n_find]           ; while ( n < n_find )
46         jnb     short quit_limit
47
48         mov     ecx, 1                  ; ecx est utilisé comme indice
49         push    ebx                     ; stocke le candidat sur la pile
50         fild    dword [esp]             ; le charge  sur la pile du coprocesseur
51         pop     ebx                     ; retire le candidat de la pile
52         fsqrt                           ; calcule sqrt(guess)
53         fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
54 ;
55 ; Cette boucle interne divise le candidat (ebx) par les nombres premiers
56 ; calculés précédemment jusqu'à ce qu'il trouve un de ses facteurs premiers
57 ; (ce qui signifie que le candidat n'est pas premier) ou jusqu'à ce que le
58 ; nombre premier par lequel diviser soit plus grand que floor(sqrt(guess))
59 ;
60 while_factor:
61         mov     eax, dword [esi + 4*ecx]        ; eax = array[ecx]
62         cmp     eax, [isqrt]                    ; while ( isqrt < array[ecx]
63         jnbe    short quit_factor_prime
64         mov     eax, ebx
65         xor     edx, edx
66         div     dword [esi + 4*ecx]
67         or      edx, edx                        ; && guess % array[ecx] != 0 )
68         jz      short quit_factor_not_prime
69         inc     ecx                             ; essaie le nombre premier suivant
70         jmp     short while_factor
71
72 ;
73 ; found a new prime !
74 ;
75 quit_factor_prime:
76         mov     eax, [n]
77         mov     dword [esi + 4*eax], ebx        ; ajoute le candidat au tableau
78         inc     eax
79         mov     [n], eax                        ; incrémente n
80
```

```
81  quit_factor_not_prime:
82          add     ebx, 2                          ; essaie le nombre impair suivant
83          jmp     short while_limit
84
85  quit_limit:
86
87          fldcw   word [orig_cntl_wd]             ; restaure le mot de contrôle
88          pop     esi                             ; restaure les variables registre
89          pop     ebx
90
91          leave
92          ret
```

prime2.asm

```
1  global _dmax
2
3  segment .text
4  ; fonction _dmax
5  ; retourne le plus grand de ses deux arguments double
6  ; prototype C
7  ; double dmax( double d1, double d2 )
8  ; Paramètres :
9  ;   d1   - premier double
10 ;   d2   - deuxième double
11 ; Valeur de retour :
12 ;   le plus grand de d1 et d2 (dans ST0)
13 %define d1   ebp+8
14 %define d2   ebp+16
15 _dmax:
16         enter   0, 0
17
18         fld     qword [d2]
19         fld     qword [d1]          ; ST0 = d1, ST1 = d2
20         fcomip  st1                 ; ST0 = d2
21         jna     short d2_bigger
22         fcomp   st0                 ; retire d2 de la pile
23         fld     qword [d1]          ; ST0 = d1
24         jmp     short exit
25 d2_bigger:                          ; si d2 est le plus grand
26                                     ; rien à faire
27 exit:
28         leave
29         ret
```

FIGURE 6.7 – Exemple de `FCOMIP`

```
1  segment .data
2  x            dq  2,75          ; converti au format double
3  five         dw  5
4
5  segment .text
6        fild   dword [five]      ; ST0 = 5
7        fld    qword [x]         ; ST0 = 2,75, ST1 = 5
8        fscale                   ; ST0 = 2,75 * 32, ST1 = 5
```

FIGURE 6.8 – Exemple d'utilisation de `FSCALE`

Chapitre 7

Structures et C++

7.1 Structures

7.1.1 Introduction

Les structures sont utilisées en C pour regrouper des données ayant un rapport entre elles dans une variable composite. Cette technique a plusieurs avantages :

1. Cela clarifie le code en montrant que les données définies dans la structure sont intimement liées.
2. Cela simplifie le passage des donneés aux fonctions. Au lieu de passer plusieurs variables séparément, elles peuvent être passées en une seule entité.
3. Cela augmente la *localité* [1] du code.

Du point de vue de l'assembleur, une structure peut être considérée comme un tableau avec des éléments de taille *variable*. Les éléments des vrais tableaux sont toujours de la même taille et du même type. C'est cette propriété qui permet de calculer l'adresse de n'importe quel élément en connaissant l'adresse de début du tableau, la taille des éléments et l'indice de l'élément voulu.

Les éléments d'une structure ne sont pas nécessairement de la même taille (et habituellement, ils ne le sont pas). A cause de cela, chaque élément d'une structure doit être explicitement spécifié et doit recevoir un *tag* (ou nom) au lieu d'un indice numérique.

En assembleur, on accède à un élément d'une structure d'une façon similaire à l'accès à un élément de tableau. Pour accéder à un élément, il faut connaître l'adresse de départ de la structure et le *déplacement relatif* de cet

1. Votez le chapitre sur la gestion de la mémoire virtuelle de n'importe quel livre sur les Systèmes d'Exploitation pour une explication de ce terme.

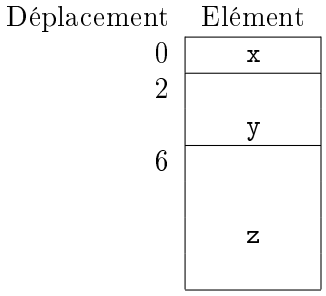

FIGURE 7.1 – Structure S

élément par rapport au début de la structure. Cependant, contrairement à un tableau où ce déplacement peut être calculé grâce à l'indice de l'élément, c'est le compilateur qui affecte un déplacement aux éléments d'une structure.

Par exemple, considérons la structure suivante :

```
struct S {
  short int x;    /* entier sur 2 octets */
  int       y;    /* entier sur 4 octets */
  double    z;    /* flottant sur 8 octets */
};
```

La Figure 7.1 montre à quoi pourrait ressembler une variable de type `S` pourrait ressembler en mémoire. Le standard ANSI C indique que les éléments d'une structure sont organisés en mémoire dans le même ordre que celui de leur définition dans le `struct`. Il indique également que le premier élément est au tout début de la structure (*i.e.* au déplacement zéro). Il définit également une macro utile dans le fichier d'en-tête `stddef.h` appelée `offsetof()`. Cette macro calcule et renvoie le déplacement de n'importe quel élément d'une structure. La macro prend deux paramètres, le premier est le nom du *type* de la structure, le second est le nom de l'élément dont on veut le déplacement. Donc le résultat de `offsetof(S, y)` serait 2 d'après la Figure 7.1.

7.1.2 Alignement en mémoire

Souvenez vous qu'une adresse est sur un multiple de double mot si elle est divisible par 4

Si l'on utilise la macro `offsetof` pour trouver le déplacement de `y` en utilisant le compilateur gcc, on s'aperçoit qu'elle renvoie 4, pas 2 ! Pourquoi ? Parce que *gcc* (et beaucoup d'autre compilateurs) aligne les variables sur des multiples de doubles mots par défaut. En mode protégé 32 bits, le processeur lit la mémoire plus vite si la donnée commence sur un multiple de double mot. La Figure 7.2 montre à quoi ressemble la structure `S` en utilisant *gcc*. Le compilateur insère deux octets inutilisés dans la structure pour aligner `y` (et `z`) sur un multiple de double mot. Cela montre pourquoi c'est une bonne

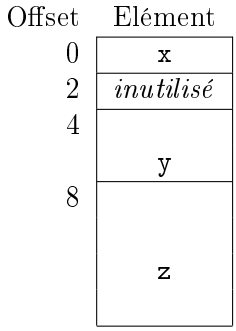

FIGURE 7.2 – Structure S

idée d'utiliser `offsetof` pour calculer les déplacements, au lieu de les calculer soi-même lorsqu'on utilise des structures en C.

Bien sûr, si la structure est utilisée uniquement en assembleur, le programmeur peut déterminer les déplacements lui-même. Cependant, si l'on interface du C et de l'assembleur, il est très important que l'assembleur et le C s'accordent sur les déplacements des éléments de la structure ! Une des complications est que des compilateurs C différents peuvent donner des déplacements différents aux éléments. Par exemple, comme nous l'avons vu, le compilateur *gcc* crée une structure `S` qui ressemble à la Figure 7.2 ; cependant, le compilateur de Borland créerait une structure qui ressemble à la Figure 7.1. Les compilateurs C fournissent le moyen de spécifier l'alignement utilisé pour les données. Cependant, le standard ANSI C ne spécifie pas comment cela doit être fait et donc, des compilateurs différents procèdent différemment.

Le compilateur *gcc* a une méthode flexible et compliquée de spécifier l'alignement. Le compilateur permet de spécifier l'alignement de n'importe quel type en utilisant une syntaxe spéciale. Par exemple, la ligne suivante :

```
typedef short int unaligned_int __attribute__((aligned(1)));
```

définit un nouveau type appelé `unaligned_int` qui est aligné sur des multiples d'octet (Oui, toutes les parenthèses suivant `__attribute__` sont nécessaires !) Le paramètre 1 de `aligned` peut être remplacé par d'autres puissances de deux pour spécifier d'autres alignements (2 pour s'aligner sur les mots, 4 sur les doubles mots, *etc.*). Si l'élément `y` de la structure était changé en un type `unaligned_int`, *gcc* placerait `y` au déplacement 2. Cependant, `z` serait toujours au déplacement 8 puisque les doubles sont également alignés sur des doubles mots par défaut. La définition du type de `z` devrait aussi être changée pour le placer au déplacement 6.

Le compilateur *gcc* permet également de *comprimer* (pack) une structure. Cela indique au compilateur d'utiliser le minimum d'espace possible pour la structure. La Figure 7.3 montre comment `S` pourrait être réécrite de cette

```
struct S {
  short int x;    /* entier sur 2 octets */
  int       y;    /* entier sur 4 octets */
  double    z;    /* flottant sur 8 octets */
} __attribute__((packed));
```

FIGURE 7.3 – Structure comprimée sous *gcc*

```
#pragma pack(push) /* sauve l'état de l'alignement */
#pragma pack(1)    /* définit un alignement sur octet*/

struct S {
  short int x;    /* entier sur 2 octets */
  int       y;    /* entier sur 4 octets */
  double    z;    /* flottant sur 8 octets */
};

#pragma pack(pop)  /* restaure l'alignement original */
```

FIGURE 7.4 – Structure comprimée sous les compilateurs Microsoft ou Borland

façon. Cette forme de `S` utiliserait le moins d'octets possible, soit 14 octets.

Les compilateurs de Microsoft et Borland supportent tous les deux la même méthode pour indiquer l'alignement par le biais d'une directive `#pragma`.

`#pragma pack(1)`

La directive ci-dessus indique au compilateur d'aligner les éléments des structures sur des multiples d'un octet (*i.e.*, sans décalage superflu). Le un peut être remplacé par deux, quatre, huit ou seize pour spécifier un alignement sur des multiples de mots, doubles mots, quadruples mots ou de paragraphe, respectivement. La directive reste active jusqu'à ce qu'elle soit écrasée par une autre. Cela peut poser des problèmes puisque ces directives sont souvent utilisées dans des fichiers d'en-tête. Si le fichier d'en-tête est inclus avant d'autres fichiers d'en-tête définissant des structures, ces structures peuvent être organisées différemment de ce qu'elles auraient été par défaut. Cela peut conduire à des erreurs très difficiles à localiser. Les différents modules d'un programmes devraient organiser les éléments des structures à *différents* endroits !

Il y a une façon d'éviter ce problème. Microsoft et Borland permettent la sauvegarde de l'état de l'alignement courant et sa restauration. La Figure 7.4 montre comment on l'utilise.

```
struct S {
  unsigned f1 : 3;    /* champ de 3 bits */
  unsigned f2 : 10;   /* champ de 10 bits */
  unsigned f3 : 11;   /* champ de 11 bits */
  unsigned f4 : 8;    /* champ de 8 bits */
};
```

FIGURE 7.5 – Exemple de Champs de Bits

7.1.3 Champs de Bits

Les champs de bits permettent de déclarer des membres d'une structure qui n'utilisent qu'un nombre de bits donné. La taille en bits n'a pas besoin d'être un multiple de huit. Un membre champ de bit est défini comme un **unsigned int** ou un **int** suivi de deux-points et de sa taille en bits. La Figure 7.5 en montre un exemple. Elle définit une variable 32 bits décomposée comme suit :

8 bits	11 bits	10 bits	3 bits
f4	f3	f2	f1

Le premier champ de bits est assigné aux bits les moins significatifs du double mot [2].

Néanmoins, le format n'est pas si simple si l'on observe comment les bits sont stockés en mémoire. La difficulté apparaît lorsque les champs de bits sont à cheval sur des multiples d'octets. Car les octets, sur un processeur little endian seront inversés en mémoire. Par exemple, les champs de bits de la structure `S` ressembleront à cela en mémoire :

5 bits	3 bits	3 bits	5 bits	8 bits	8 bits
f2l	f1	f3l	f2m	f3m	f4

L'étiquette *f2l* fait référence aux cinq derniers bits (*i.e.*, les cinq bits les moins significatifs) du champ de bits *f2*. L'étiquette *f2m* fait référence aux cinq bits les plus significatifs de *f2*. Les lignes verticales doubles montrent les limites d'octets. Si l'on inverse tous les octets, les morceaux des champs *f2* et *f3* seront réunis correctement.

L'organisation de la mémoire physique n'est habituellement pas importante à moins que des données de soient transférées depuis ou vers le programme (ce qui est en fait assez courant avec les champs de bits). Il est courant que les interfaces de périphériques matériels utilisent des nombres impairs de bits dont les champs de bits facilitent la représentation.

2. En fait, le standard ANSI/ISO C laisse une certaine liberté au compilateur sur la façon d'organiser les bits. Cependant, les compilateurs C courants (*gcc*, *Microsoft* et *Borland*) organisent les champs comme cela.

Byte \ Bit	7	6	5	4	3	2	1	0
0	Code Opération (08h)							
1	N° d'Unité Logique			msb de l'ABL				
2	milieu de l'Adresse de Bloc Logique							
3	lsb de l'Adresse de Bloc Logique							
4	Longueur du Transfert							
5	Contrôle							

FIGURE 7.6 – Format de la Commande de Lecture SCSI

Un bon exemple est SCSI[3]. Une commande de lecture directe pour un périphérique SCSI est spécifiée en envoyant un message de six octets au périphérique selon le format indiqué dans la Figure 7.6. La difficulté de représentation en utilisant les champs de bits est l'*adresse de bloc logique* qui est à cheval sur trois octets différents de la commande. D'après la Figure 7.6, on constate que les données sont stockées au format big endian. La Figure 7.7 montre une définition qui essaie de fonctionner avec tous les compilateurs. Les deux premières lignes définissent une macro qui est vraie si le code est compilé avec un compilateur Borland ou Microsoft. La partie qui peut porter à confusion va des lignes 11 à 14. Tout d'abord, on peut se demander pourquoi les champs lba_mid et lba_lsb sont définis séparément et non pas comme un champ unique de 16 bits. C'est parce que les données sont stockées au format big endian. Un champ de 16 bits serait stocké au format little endian par le compilateur. Ensuite, les champs lba_msb et logical_unit semblent être inversés ; cependant, ce n'est pas le cas. Ils doivent être placés dans cet ordre. La Figure 7.8 montre comment les champs sont organisés sous forme d'une entité de 48 bits (les limites d'octets sont là encore représentées par des lignes doubles). Lorsqu'elle est stockée en mémoire au format little endian, les bits sont réarrangés au format voulu (Figure 7.6).

Pour compliquer encore plus le problème, la définition de SCSI_read_cmd ne fonctionne pas correctement avec le C Microsoft. Si l'expression **sizeof**(SCSI_read_cmd) est évaluée, le C Microsoft renvoie 8 et non pas 6 ! C'est parce que le compilateur Microsoft utilise le type du champ de bits pour déterminer comment organiser les bits. Comme tous les bits sont déclarés comme **unsigned**, le compilateur ajoute deux octets à la fin de la structure pour qu'elle comporte un nombre entier de double mots. Il est possible d'y remédier en déclarant tous les champs **unsigned short**. Maintenant, le compilateur Microsoft n'a plus besoin d'ajouter d'octets d'alignement puisque six octets forment un nombre entier de mots de deux octets[4]. Les autres compilateurs fonc-

3. Small Computer Systems Interface, un standard de l'industrie pour les disques durs, *etc.*

4. Mélanger différents types de champs de bits conduit à un comportement très

```
#define MS_OR_BORLAND (defined(__BORLANDC__) \
                       || defined(_MSC_VER))

#if MS_OR_BORLAND
# pragma pack(push)
# pragma pack(1)
#endif

struct SCSI_read_cmd {
  unsigned opcode : 8;
  unsigned lba_msb : 5;
  unsigned logical_unit : 3;
  unsigned lba_mid : 8;    /* bits du milieu */
  unsigned lba_lsb : 8;
  unsigned transfer_length : 8;
  unsigned control : 8;
}
#if defined(__GNUC__)
   __attribute__((packed))
#endif
;

#if MS_OR_BORLAND
# pragma pack(pop)
#endif
```

FIGURE 7.7 – Structure du Format de la Commande de Lecture SCSI

tionnent également correctement avec ce changement. La Figure 7.9 montre une autre définition qui fonctionne sur les trois compilateurs. Il ne déclare plus que deux champs de bits en utilisant le type **unsigned char**.

Le lecteur ne doit pas se décourager s'il trouve la discussion ci-dessus confuse. C'est confus ! L'auteur trouve souvent moins confus d'éviter d'utiliser des champs de bits en utilisant des opérations niveau bit pour examiner et modifier les bits manuellement.

7.1.4 Utiliser des structures en assembleur

Comme nous l'avons dit plus haut, accéder à une structure en assembleur ressemble beaucoup à accéder à un tableau. Prenons un exemple simple, regardons comment l'on pourrait écrire une routine assembleur qui mettrait

étrange ! Le lecteur est invité à tester.

8 bits	8 bits	8 bits	8 bits	3 bits	5 bits	8 bits
control	transfer_length	lba_lsb	lba_mid	logical_unit	lba_msb	opcode

FIGURE 7.8 – Organisation des champs de `SCSI_read_cmd`

```
1  struct SCSI_read_cmd {
2    unsigned char opcode;
3    unsigned char lba_msb : 5;
4    unsigned char logical_unit : 3;
5    unsigned char lba_mid;   /* bits du milieu */
6    unsigned char lba_lsb;
7    unsigned char transfer_length;
8    unsigned char control;
9  }
10 #if defined(__GNUC__)
11    __attribute__((packed))
12 #endif
13 ;
```

FIGURE 7.9 – Structure du Format de la Commande de Lecture SCSI Alternative

à zéro l'élément `y` d'une structure `S`. Supposons que le prototype de la routine soit :

```
void zero_y( S * s_p );
```

La routine assembleur serait :

```
1  %define      y_offset  4
2  _zero_y:
3        enter  0,0
4        mov    eax, [ebp + 8]      ; récupère s_p depuis la pile
5        mov    dword [eax + y_offset], 0
6        leave
7        ret
```

Le C permet de passer une structure par valeur à une fonction ; cependant, c'est une mauvaise idée la plupart du temps. Toutes les données de la structure doivent être copiées sur sur la pile puis récupérées par la routine. Il est beaucoup plus efficace de passer un pointeur vers la structure à la place.

Le C permet aussi qu'une fonction renvoie une structure. Evidemment, une structure ne peut pas être retournée dans le registre `EAX`. Des compilateurs différents gèrent cette situation de façon différente. Une situation

```
1  #include <stdio.h>
2
3  void f( int x )
4  {
5     printf ("%d\n", x);
6  }
7
8  void f( double x )
9  {
10    printf ("%g\n", x);
11 }
```

FIGURE 7.10 – Deux fonctions `f()`

courante que les compilateurs utilisent est de réécrire la fonction en interne de façon à ce qu'elle prenne un pointeur sur la structure en paramètre. Le pointeur est utilisé pour placer la valeur de retour dans une structure définie en dehors de la routine appelée.

La plupart des assembleur (y compris NASM) ont un support intégré pour définir des structures dans votre code assembleur. Reportez vous à votre documentation pour plus de détails.

7.2 Assembleur et C++

Le langage de programmation C++ est une extension du langage C. Beaucoup des règles valables pour interfacer le C et l'assembleur s'appliquent également au C++. Cependant, certaines règles doivent être modifiées. De plus, certaines extension du C++ sont plus faciles à comprendre en connaissant le langage assembleur. Cette section suppose une connaissance basique du C++.

7.2.1 Surcharge et Décoration de Noms

Le C++ permet de définir des fonctions (et des fonctions membres) différentes avec le même nom. Lorsque plus d'une fonction partagent le même nom, les fonctions sont dites *surchargées*. Si deux fonctions sont définies avec le même nom en C, l'éditeur de liens produira une erreur car il trouvera deux définitions pour le même symbole dans les fichiers objets qu'il est en train de lier. Par exemple, prenons le code de la Figure 7.10. Le code assembleur équivalent définirait deux étiquettes appelées `_f` ce qui serait bien sûr une erreur.

Le C++ utilise le même procédé d'édition de liens que le C mais évite cette erreur en effectuant une *décoration de nom* (name mangling) ou en modifiant le symbole utilisé pour nommer une fonction. D'une certaine façon, le C utilise déjà la décoration de nom. Il ajoute un caractère de soulignement au nom de la fonction C lorsqu'il crée l'étiquette pour la fonction. Cependant, il décorera le nom des deux fonctions de la Figure 7.10 de la même façon et produira une erreur. Le C++ utilise un procédé de décoration plus sophistiqué qui produit deux étiquettes différentes pour les fonctions. Par exemple, la première fonction de la Figure 7.10 recevrait l'étiquette `_f__Fi` et la seconde, `_f__Fd`, sous DJGPP. Cela évite toute erreur d'édition de liens.

Malheureusement, il n'y a pas de standard sur la gestion des noms en C++ et des compilateurs différents décorent les noms de façon différente. Par exemple, Borland C++ utiliserait les étiquettes `@f$qi` et `@f$qd` pour les deux fonctions de la Figure 7.10. Cependant, les règles ne sont pas totalement arbitraires. Le nom décoré encode la *signature* de la fonction. La signature d'une fonction est donnée par l'ordre et le type de ses paramètres. Notez que la fonction qui ne prend qu'un argument `int` a un *i* à la fin de son nom décoré (à la fois sous DJGPP et Borland) et que celle qui prend un argument `double` a un *d* à la fin de son nom décoré. S'il y avait une fonction appelée `f` avec le prototype suivant :

```
void f( int x, int y, double z);
```

DJGPP décorerait son nom en `_f__Fiid` et Borland en `@f$qiid`.

Le type de la fonction ne fait *pas* partie de la signature d'une fonction et n'est pas encodé dans nom décoré. Ce fait explique une règle de la surcharge en C++. Seules les fonctions dont les signatures sont uniques peuvent être surchargées. Comme on le voit, si deux fonctions avec le même nom et la même signature sont définies en C++, elle donneront le même nom décoré et créeront une erreur lors de l'édition de liens. Par défaut, toutes les fonctions C++ sont décorées, même celles qui ne sont pas surchargées. Lorsqu'il compile un fichier, le compilateur n'a aucun moyen de savoir si une fonction particulière est surchargée ou non, il décore donc touts les noms. En fait, il décore également les noms des variables globales en encodant le type de la variable d'une façon similaire à celle utilisée pour les signatures de fonctions. Donc, si l'on définit une variable globale dans un fichier avec un certain type puis que l'on essaie de l'utiliser dans un autre fichier avec le mauvais type, l'éditeur de liens produira une erreur. Cette caractéristique du C++ est connue sous le nom de *typesafe linking* (édition de liens avec respect des types) . Cela crée un autre type d'erreurs, les prototypes inconsistants. Cela arrive lorsque la définition d'une fonction dans un module ne correspond pas avec le prototype utilisé par un autre module. En C, cela peut être un problème très difficile à corriger. Le C ne détecte pas cette erreur. Le programme compilera et sera lié mais aura un comportement imprévisible car le

code appelant placera sur la pile des types différents de ceux que la fonction attend. En C++ cela produira une erreur lors de l'édition de liens.

Lorsque le compilateur C++ analyse un appel de fonction, il recherche la fonction correspondante en observant les arguments qui lui sont passés [5]. S'il trouve une correspondance, il crée un `CALL` vers la fonction adéquate en utilisant les règles de décoration du compilateur.

Comme des compilateurs différents utilisent différentes règles de décoration de nom, il est possible que des codes C++ compilés par des compilateurs différents ne puissent pas être liés ensemble. C'est important lorsque l'on a l'intention d'utiliser une bibliothèque C++ précompilée ! Si l'on veut écrire une fonction en assembleur qui sera utilisée avec du code C++, il faut connaître les règles de décoration de nom du compilateur C++ utilisé (ou utiliser la technique expliquée plus bas).

L'étudiant astucieux pourrait se demander si le code de la Figure 7.10 fonctionnera de la façon attendue. Comme le C++ décore toutes les fonctions, alors la fonction `printf` sera décorée et le compilateur ne produira pas un `CALL` à l'étiquette `_printf`. C'est une question pertinente. Si le prototype de `printf` était simplement placé au début du fichier, cela arriverait. Son prototype est :

```
int printf( const char *, ...);
```

DJGPP décorerait ce nom en `_printf__FPCce` (`F` pour *fonction*, `P` pour *pointeur*, `C` pour *const*, `c` pour *char* et `e` pour ellipse). Cela n'appelerait pas la fonction `printf` de la bibliothèque C standard ! Bien sûr, il doit y avoir un moyen pour que le C++ puisse appeler du code C. C'est très important car il existe une *énorme* quantité de vieux code C utile. En plus de permettre l'accès au code hérité de C, le C++ permet également d'appeler du code assembleur en utilisant les conventions de décoration standards du C.

Le C++ étend le mot-clé `extern` pour lui permettre de spécifier que la fonction ou la variable globale qu'il modifie utilise les conventions C normales. Dans la terminologie C++, la fonction ou la variable globale utilise une *édition de liens C*. Par exemple, pour déclarer la fonction `printf` comme ayant une édition de liens C, utilisez le prototype :

```
extern "C" int printf( const char *, ... );
```

Cela impose au compilateur de ne pas utiliser les règles de décoration de nom du C++ sur la fonction, mais d'utiliser les règles C à la place. Cependant, en faisant cela, la fonction `printf` ne peut pas être surchargée. Cela constitue la façon la plus simple d'interfacer du C++ et de l'assembleur, définir une

5. La correspondance n'a pas à être exacte, le compilateur prendra en compte les correspondances trouvées en transtypant les arguments. Les règles de ce procédé sont en dehors de la portée de ce livre. Consultez un livre sur le C++ pour plus de détails.

```
1  void f( int \& x )      // le & indique un paramètre par référence
2  { x++; }
3
4  int main()
5  {
6    int y = 5;
7    f(y);                 // une référence sur y est passée, pas de & ici !
8    printf ("%d\n", y);   // affiche 6 !
9    return 0;
10 }
```

FIGURE 7.11 – Exemple de référence

fonction comme utilisant une édition de liens C puis utiliser la convention d'appel C.

Pour plus de facilité, le C++ permet également de définir une édition de liens C sur un bloc de fonctions et de variables globales. Le bloc est indiqué en utilisant les accolades habituelles.

```
extern "C" {
  /* variables globales et prototypes des fonction ayant une édition de liens C */
}
```

Si l'on examine les fichiers d'en-tête ANSI C fournis avec les compilateur C/C++ actuels, on trouve ce qui suit vers le début de chaque fichier d'en-tête :

```
#ifdef __cplusplus
extern "C" {
#endif
```

Et une construction similaire, vers la fin, contenant une accolade fermante. Les compilateurs C++ définissent la macro `__cplusplus` (avec *deux* caractères de soulignement au début). L'extrait ci-dessus entoure tout le fichier d'en-tête dans un bloc `extern "C"` si le fichier d'en-tête est compilé en C++, mais ne fait rien s'il est compilé en C (puisqu'un compilateur C génèrerait une erreur de syntaxe sur `extern "C"`). La même technique peut être utilisée par n'importe quel programmeur pour créer un fichier d'en-tête pour des routines assembleur pouvant être utilisées en C ou en C++.

7.2.2 Références

Les *références* sont une autre nouvelle fonctionnalité du C++. Elles permettent de passer des paramètres à une fonction sans utiliser explicitement

de pointeur. Par exemple, considérons le code de la Figure 7.11. En fait, les paramètres par référence sont plutôt simples, ce sont des pointeurs. Le compilateur masque simplement ce fait aux yeux du programmeur (exactement de la même façon que les compilateurs Pascal qui implémentent les paramètres `var` comme des pointeurs). Lorsque le compilateur génère l'assembleur pour l'appel de fonction ligne 7, il passe l'*adresse* de `y`. Si l'on écrivait la fonction `f` en assembleur, on ferait comme si le prototype était [6] :

```
void f( int * xp);
```

Les références sont juste une facilité qui est particulièrement utile pour la surcharge d'opérateurs. C'est une autre fonctionnalité du C++ qui permet de donner une signification aux opérateurs de base lorsqu'ils sont appliqués à des structures ou des classes. Par exemple, une utilisation courante est de définir l'opérateur plus (+) de façon à ce qu'il concatène les objets string. Donc, si `a` et `b` sont des strings, `a + b` renverra la concaténation des chaînes `a` et `b`. Le C++ appelerait en réalité une fonction pour ce faire (en fait, cette expression pourrait être réécrite avec une notation fonction sous la forme `operator +(a,b)`). Pour plus d'efficacité, il est souhaitable de passer l'adresse des objets string à la place de les passer par valeur. Sans référence, cela pourrait être fait en écrivant `operator +(&a,&b)`, mais cela imposerait d'écrire l'opérateur avec la syntaxe `&a + &b`. Cela serait très maladroit et confus. Par contre, en utilisant les références, il est possible d'écrire `a + b`, ce qui semble très naturel.

7.2.3 Fonctions inline

Les *fonctions inline* sont encore une autre fonctionnalité du C++ [7]. Les fonctions inline sont destinées à remplacer les macros du préprocesseur qui prennent des paramètres, sources d'erreur. Sourvenez vous en C, une macro qui élève un nombre au carré ressemble à cela :

```
#define SQR(x) ((x)*(x))
```

Comme le préprocesseur ne comprend pas le C et ne fait que de simples substitutions, les parenthèses sont requises pour calculer le résultat correct dans la plupart des cas. Cependant, même cette version ne donnerait pas la bonne réponse pour `SQR(x++)`.

Les macros sont utilisées car elles éliminent la surcharge d'un appel pour une fonction simple. Comme le chapitre sur les sous-programmes l'a démontré, effectuer un appel de fonction implique plusieurs étapes. Pour une fonction très simple, le temps passé à l'appeler peut être plus grand que celui

6. Bien sûr, il faudrait déclarer la fonction avec une édition de liens en C, comme nous en avons parlé dans la Section 7.2.1

7. Les compilateurs supportent souvent cette fonctionnalité comme une extension du C ANSI.

```
1  inline int inline_f( int x )
2  { return x*x; }
3
4  int f( int x )
5  { return x*x; }
6
7  int main()
8  {
9    int y, x = 5;
10   y = f(x);
11   y = inline_f(x);
12   return 0;
13 }
```

FIGURE 7.12 – Exemple d'inlining

passé dans la fonction ! Les fonctions inline sont une façon beaucoup plus pratique d'écrire du code qui ressemble à une fonction mais qui n'effectue *pas* de `CALL` à un bloc commun. Au lieu de cela, les appels à des fonctions inline sont remplacés par le code de la fonction. Le C++ permet de rendre une fonction inline en plaçant le mot-clé `inline` au début de sa définition. Par exemple, considérons les fonctions déclarées dans la Figure 7.12. L'appel à la fonction `f`, ligne 10, est un appel de fonction normal (en assembleur, en supposant que `x` est à l'adresse `ebp-8` et `y` en `ebp-4`):

```
1       push   dword [ebp-8]
2       call   _f
3       pop    ecx
4       mov    [ebp-4], eax
```

Cependant, l'appel à la fonction `inline_f`, ligne 11 ressemblerait à :

```
1       mov    eax, [ebp-8]
2       imul   eax, eax
3       mov    [ebp-4], eax
```

Dans ce cas, il y a deux avantages à inliner. Tout d'abord, la fonction inline est plus rapide. Aucun paramètre n'est placé sur la pile, aucun cadre de pile n'est créé puis détruit, aucun branchement n'est effectué. Ensuite, l'appel à la fonction inline utilise moins de code ! Ce dernier point est vrai pour cet exemple, mais ne reste pas vrai dans tous les cas.

```
1  class Simple {
2  public:
3    Simple();                  // constructeur par défaut
4    ~Simple();                 // destructeur
5    int get_data() const;      // fonctions membres
6    void set_data( int  );
7  private:
8    int data;                  // données membres
9  };
10
11 Simple::Simple()
12 { data = 0; }
13
14 Simple::~Simple()
15 { /* rien */ }
16
17 int Simple::get_data() const
18 { return data; }
19
20 void Simple::set_data( int x )
21 { data = x; }
```

FIGURE 7.13 – Une classe C++ simple

La principal inconvénient de l'inlining est que le code inline n'est pas lié et donc le code d'une fonction inline doit être disponible pour *tous* les fichiers qui l'utilisent. L'exemple de code assembleur précédent le montre. L'appel de la fonction non-inline ne nécessite que la connaissance des paramètres, du type de valeur de retour, de la convention d'appel et du nom de l'étiquette de la fonction. Toutes ces informations sont disponibles par le biais du prototype de la fonction. Cependant, l'utilisation de la fonction inline nécessite la connaissance de tout le code de la fonction. Cela signifie que si *n'importe quelle* partie de la fonction change, *tous* les fichiers source qui utilisent la fonction doivent être recompilés. Souvenez vous que pour les fonctions non-inline, si le prototype ne change pas, souvent les fichiers qui utilisent la fonction n'ont pas besoin d'être recompilés. Pour toutes ces raisons, le code des fonctions inline est généralement placé dans les fichiers d'en-tête. Cette pratique est contraire à la règle stricte habituelle du C selon laquelle on ne doit *jamais* placer de code exécutable dans les fichiers d'en-tête.

```
void set_data( Simple * object, int x )
{
  object->data = x;
}
```

FIGURE 7.14 – Version C de Simple::set_data()

```
1  _set_data__6Simplei:           ; nom décoré
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp + 8]   ; eax = pointeur vers l'objet (this)
6        mov    edx, [ebp + 12]  ; edx = paramètre entier
7        mov    [eax], edx       ; data est au déplacement 0
8
9        leave
10       ret
```

FIGURE 7.15 – Traduction de Simple::set_data(int) par le compilateur

7.2.4 Classes

En fait, le C++ utilise le mot clé `this` *pour accéder au pointeur vers l'objet lorsque l'on se trouve à l'intérieur d'une fonction membre.*

Une classe C++ décrit un type d'*objet*. Un objet possède à la fois des membres données et des membres fonctions[8]. En d'autres termes, il s'agit d'une `struct` à laquelle sont associées des données et des fonctions. Considérons la classe simple définie dans la Figure 7.13. Une variable de type `Simple` ressemblerait à une `struct` C normale avec un seul membre `int`. Les fonctions ne sont *pas* affectées à la structure en mémoire. Cependant, les fonctions membres sont différentes des autres fonctions. On leur passe un paramètre *caché*. Ce paramètre est un pointeur vers l'objet sur lequel agit la fonction.

Par exemple, considérons la méthode `set_data` de la classe `Simple` de la Figure 7.13. Si elle était écrite en C, elle ressemblerait à une fonction à laquelle on passerait explicitement un pointeur vers l'objet sur lequel elle agit comme le montre le code de la Figure 7.14. L'option `-S` du compilateur *DJGPP* (et des compilateurs *gcc* et Borland également) indique au compilateur de produire un fichier assembleur contenant l'équivalent assembleur du code. Pour *DJGPP* et *gcc* le fichier assembleur possède une extension `.s` et utilise malheureusement la syntaxe du langage assembleur AT&T qui est

8. Souvent appelés *fonctions membres* en C++ ou plus généralement *méthodes*.

assez différente des syntaxes NASM et MASM[9] (les compilateurs Borland et MS génèrent un fichier avec l'extension `.asm` utilisant la syntaxe MASM.) La Figure 7.15 montre la sortie de *DJGPP* convertie en syntaxe NASM avec des commentaires supplémentaires pour clarifier le but des instructions. Sur la toute première ligne, notez que la méthode `set_data` reçoit une étiquette décorée qui encode le nom de la méthode, le nom de la classe et les paramètres. Le nom de la classe est encodé car d'autres classes peuvent avoir une méthode appelée `set_data` et les deux méthodes *doivent* recevoir des étiquettes différentes. Les paramètres sont encodés afin que la classe puisse surcharger la méthode `set_data` afin qu'elle prenne d'autres paramètres, comme les fonctions C++ normales. Cependant, comme précédemment, des compilateurs différents encoderont ces informations différemment dans l'étiquette.

Ensuite, aux lignes 2 et 3 nous retrouvons le prologue habituel. A la ligne 5, le premier paramètre sur la pile est stocké dans `EAX`. Ce n'est *pas* le paramètre `x` ! Il s'agit du paramètre caché[10] qui pointe vers l'objet sur lequel on agit. La ligne 6 stocke le paramètre `x` dans `EDX` et la ligne 7 stocke `EDX` dans le double mot sur lequel pointe `EAX`. Il s'agit du membre `data` de l'objet `Simple` sur lequel on agit, qui, étant la seule donnée de la classe, est stocké au déplacement 0 de la structure `Simple`.

Exemple

Cette section utilise les idées de ce chapitre pour créer une classe C++ qui représente un entier non signé d'une taille arbitraire. Comme l'entier peut faire n'importe quelle taille, il sera stocké dans un tableau d'entiers non signés (doubles mots). Il peut faire n'importe quelle taille en utilisant l'allocation dynamique. Les doubles mots sont stockés dans l'ordre inverse[11] (*i.e.* le double mot le moins significatif est au déplacement 0). La Figure 7.16 montre la définition de la classe `Big_int`[12]. La taille d'un `Big_int` est mesurée par la taille du tableau d'`unsigned` utilisé pour stocker les données. La donnée membre `size_` de la classe est affectée au déplacement 0 et le membre `number_` est affecté au déplacement 4.

Pour simplifier l'exemple, seuls les objets ayant des tableaux de la même taille peuvent être additionnés entre eux.

---

9. Le compilateur *gcc* inclut son propre assembleur appelé *gas*. L'assembleur *gas* utilise la syntaxe AT&T et donc le compilateur produit un code au format *gas*. Il y a plusieurs sites sur le web qui expliquent les différences entre les formats INTEL et AT&T. Il existe également un programme gratuit appelé `a2i` (http://www.multimania.com/placr/a2i.html), qui passe du format AT&T au format NASM.

10. Comme d'habitude, *rien* n'est caché dans le code assembleur !

11. Pourquoi ? Car les opérations d'addition commenceront ainsi toujours par le début du tableau et avanceront.

12. Voyez le code source d'exemple pour obtenir le code complet de cet exemple. Le texte ne se réfèrera qu'à certaines parties du code.

```
1  class Big_int {
2  public:
3    /*
4     * Paramètres :
5     *    size            - taille de l'entier exprimée en nombre d'unsigned
6     *                      int normaux
7     *    initial_value   - valeur initiale du Big_int sous forme d'un
8     *                      unsigned int normal
9     */
10   explicit Big_int( size_t    size,
11                     unsigned initial_value = 0);
12   /*
13    * Paramètres :
14    *    size            - taille de l'entier exprimée en nombre d'unsigned
15    *                      int normaux
16    *    initial_value   - valeur initiale du Big_int sous forme d'une chaîne
17    *                      contenant une représentation hexadécimale de la
18    *                      valeur.
19    */
20   Big_int( size_t         size,
21            const char * initial_value );
22
23   Big_int( const Big_int & big_int_to_copy);
24   ~Big_int();
25
26   // renvoie la taille du Big_int (en termes d'unsigned int)
27   size_t size() const;
28
29   const Big_int & operator = ( const Big_int & big_int_to_copy);
30   friend Big_int operator + ( const Big_int & op1,
31                               const Big_int & op2 );
32   friend Big_int operator - ( const Big_int & op1,
33                               const Big_int & op2);
34   friend bool operator == ( const Big_int & op1,
35                             const Big_int & op2 );
36   friend bool operator < ( const Big_int & op1,
37                            const Big_int & op2);
38   friend ostream & operator << ( ostream &      os,
39                                  const Big_int & op );
40 private:
41   size_t       size_;     // taille du tableau d'unsigned
42   unsigned * number_; // pointeur vers un tableau d'unsigned contenant
43                            la valeur
44 };
```

FIGURE 7.16 – Définition de la classe Big_int

```
1  // prototypes des routines assembleur
2  extern "C" {
3    int add_big_ints( Big_int &       res,
4                      const Big_int & op1,
5                      const Big_int & op2);
6    int sub_big_ints( Big_int &       res,
7                      const Big_int & op1,
8                      const Big_int & op2);
9  }
10
11 inline Big_int operator + ( const Big_int & op1, const Big_int & op2)
12 {
13   Big_int result(op1.size());
14   int res = add_big_ints(result, op1, op2);
15   if (res == 1)
16     throw Big_int::Overflow();
17   if (res == 2)
18     throw Big_int::Size_mismatch();
19   return result;
20 }
21
22 inline Big_int operator - ( const Big_int & op1, const Big_int & op2)
23 {
24   Big_int result(op1.size());
25   int res = sub_big_ints(result, op1, op2);
26   if (res == 1)
27     throw Big_int::Overflow();
28   if (res == 2)
29     throw Big_int::Size_mismatch();
30   return result;
31 }
```

FIGURE 7.17 – Code de l'Arithmétique sur la Classe Big_int

La classe a trois constructeurs : le premier (ligne 9) initialise l'instance de la classe en utilisant un entier non signé normal ; le second, (ligne 19) initalise l'instance en utilisant une chaîne qui contient une valeur hexadécimale. Le troisième constructeur (ligne 22) est le *constructeur par copie*.

Cette explication se concentre sur la façon dont fonctionnent les opérateurs d'addition et de soustraction car c'est là que l'on utilise de l'assembleur. La Figure 7.17 montre les parties du fichier d'en-tête relatives à ces opérateurs. Elles montrent comment les opérateurs sont paramétrés pour appeler des routines assembleur. Comme des compilateurs différents utilisent des règles de décoration radicalement différentes pour les fonctions opérateur, des fonctions opérateur inline sont utilisées pour initialiser les appels aux routines assembleur liées au format C. Cela les rend relativement simples à porter sur des compilateurs différents et est aussi rapide qu'un appel direct. Cette technique élimine également le besoin de soulever une exception depuis l'assembleur !

Pourquoi l'assembleur n'est-il utilisé qu'ici ? Souvenez vous que pour effectuer de l'arithmétique en précision multiple, la retenue doit être ajoutée au double mot significatif suivant. Le C++ (et le C) ne permet pas au programmeur d'accéder au drapeau de retenue du processeur. On ne pourrait effectuer l'addition qu'en recalculant indépendamment en C++ la valeur du drapeau de retenue et en l'ajoutant de façon conditionnelle au double mot suivant. Il est beaucoup plus efficace d'écrire le code en assembleur à partir duquel on peut accéder au drapeau de retenue et utiliser l'instruction `ADC` qui ajoute automatiquement le drapeau de retenue.

Par concision, seule la routine assembleur `add_big_ints` sera expliquée ici. Voici le code de cette routine (contenu dans `big_math.asm`) :

big_math.asm

```
1  segment .text
2          global  add_big_ints, sub_big_ints
3  %define size_offset 0
4  %define number_offset 4
5
6  %define EXIT_OK 0
7  %define EXIT_OVERFLOW 1
8  %define EXIT_SIZE_MISMATCH 2
9
10 ; Paramètres des routines add et sub
11 %define res ebp+8
12 %define op1 ebp+12
13 %define op2 ebp+16
14
15 add_big_ints:
```

```
16          push    ebp
17          mov     ebp, esp
18          push    ebx
19          push    esi
20          push    edi
21          ;
22          ; initialise esi pour pointer vers op1
23          ;            edi pour pointer vers op2
24          ;            ebx pour pointer vers res
25          mov     esi, [op1]
26          mov     edi, [op2]
27          mov     ebx, [res]
28          ;
29          ; s'assure que les 3 Big_int ont la même taille
30          ;
31          mov     eax, [esi + size_offset]
32          cmp     eax, [edi + size_offset]
33          jne     sizes_not_equal                 ; op1.size_ != op2.size_
34          cmp     eax, [ebx + size_offset]
35          jne     sizes_not_equal                 ; op1.size_ != res.size_
36
37          mov     ecx, eax                        ; ecx = taille des Big_int
38          ;
39          ; initialise les registres pour qu'ils pointent vers leurs tableaux respectifs
40          ;       esi = op1.number_
41          ;       edi = op2.number_
42          ;       ebx = res.number_
43          ;
44          mov     ebx, [ebx + number_offset]
45          mov     esi, [esi + number_offset]
46          mov     edi, [edi + number_offset]
47
48          clc                                     ; met le drapeau de retenue à 0
49          xor     edx, edx                        ; edx = 0
50          ;
51          ; boucle d'addition
52  add_loop:
53          mov     eax, [edi+4*edx]
54          adc     eax, [esi+4*edx]
55          mov     [ebx + 4*edx], eax
56          inc     edx                             ; ne modifie pas le drapeau de retenue
57          loop    add_loop
```

```
58
59          jc      overflow
60  ok_done:
61          xor     eax, eax                        ; valeur de retour = EXIT_OK
62          jmp     done
63  overflow:
64          mov     eax, EXIT_OVERFLOW
65          jmp     done
66  sizes_not_equal:
67          mov     eax, EXIT_SIZE_MISMATCH
68  done:
69          pop     edi
70          pop     esi
71          pop     ebx
72          leave
73          ret
```

big_math.asm

Heureusement, la majorité de ce code devrait être compréhensible pour le lecteur maintenant. Les lignes 25 à 27 stockent les pointeurs vers les objets `Big_int` passés à la fonction via des registres. Souvenez vous que les références sont des pointeurs. Les lignes 31 à 35 vérifient que les tailles des trois objets sont les mêmes (Notez que le déplacement de `size_` est ajouté au pointeur pour accéder à la donnée membre). Les lignes 44 à 46 ajustent les registres pour qu'ils pointent vers les tableaux utilisés par leurs objets respectifs au lieu des objets eux-mêmes (là encore, le déplacement du membre `number_` est ajouté au pointeur sur l'objet).

La boucle des lignes 52 à 57 additionne les entiers stockés dans les tableaux en additionnant le double mot le moins significatif en premier, puis les doubles mots suivants, *etc.* L'addition doit être effectuée dans cet ordre pour l'arithmétique en précision étendue (voir Section 2.1.5). La ligne 59 vérifie qu'il n'y a pas de dépassement de capacité, lors d'un dépassement de capacité, le drapeau de retenue sera allumé par la dernière addition du double mot le plus significatif. Comme les doubles mots du tableau sont stockés dans l'ordre little endian, la boucle commence au début du tableau et avance jusqu'à la fin.

La Figure 7.18 montre un court exemple utilisant la classe `Big_int`. Notez que les constantes de `Big_int` doivent être déclarées explicitement comme à la ligne 16. C'est nécessaire pour deux raisons. Tout d'abord, il n'y a pas de constructeur de conversion qui convertisse un entier non signé en `Big_int`. Ensuite, seuls les `Big_int` de même taille peuvent êtr additionnés. Cela rend la conversion problématique puisqu'il serait difficile de savoir vers quelle taille convertir. Une implémentation plus sophistiquée de la classe permettrait d'additionner n'importe quelle taille avec n'importe quelle autre.

```
#include "big_int.hpp"
#include <iostream>
using namespace std;

int main()
{
  try {
    Big_int b(5,"8000000000000a00b");
    Big_int a(5,"80000000000010230");
    Big_int c = a + b;
    cout << a << " + " << b << " = " << c << endl;
    for( int i=0; i < 2; i++ ) {
      c = c + a;
      cout << "c = " << c << endl;
    }
    cout << "c-1 = " << c - Big_int(5,1) << endl;
    Big_int d(5, "12345678");
    cout << "d = " << d << endl;
    cout << "c == d " << (c == d) << endl;
    cout << "c > d " << (c > d) << endl;
  }
  catch( const char * str ) {
    cerr << "Caught : " << str << endl;
  }
  catch( Big_int::Overflow ) {
    cerr << "Dépassement de capacité " << endl;
  }
  catch( Big_int::Size_mismatch ) {
    cerr << "Non concordance de taille" << endl;
  }
  return 0;
}
```

FIGURE 7.18 – Utilisation Simple de `Big_int`

L'auteur ne voulait pas compliquer inutilement cet exemple en l'implémentant ici (cependant, le lecteur est encouragé à le faire).

7.2.5 Héritage et Polymorphisme

L'*héritage* permet à une classe d'hériter des données et des méthodes d'une autre. Par exemple, considérons le code de la Figure 7.19. Il montre deux classes, `A` et `B`, où la classe `B` hérite de `A`. La sortie du programme est :

```
Taille de a : 4 Déplacement de ad : 0
Taille de b : 8 Déplacement de ad: 0 Déplacement de bd: 4
A::m()
A::m()
```

Notez que les membres `ad` des deux classes (`B` l'hérite de `A`) sont au même déplacement. C'est important puisque l'on peut passer à la fonction `f` soit un pointeur vers un objet `A` soit un pointeur vers un objet de n'importe quel type dérivé de (*i.e.* qui hérite de) `A`. La Figure 7.20 montre le code assembleur (édité) de la fonction (généré par *gcc*).

Notez que la sortie de la méthode `m` de `A` a été produite à la fois par l'objet `a` et l'objet `b`. D'après l'assembleur, on peut voir que l'appel à `A::m()` est codé en dur dans la fonction. Dans le cadre d'une vraie programmation orientée objet, la méthode appelée devrait dépendre du type d'objet passé à la fonction. On appelle cela le *polymorphisme*. Le C++ désactive cette fonctionnalité par défaut. On utilise le mot-clé *virtual* pour l'activer. La Figure 7.21 montre comment les deux classes seraient modifiées. Rien dans le restet du code n'a besoin d'être changé. Le polymorphisme peut être implémenté de beaucoup de manières. Malheureusement, l'implémentation de *gcc* est en transition au moment d'écrire ces lignes et devient beaucoup plus compliquée que l'implémentation initiale. Afin de simplifier cette explication, l'auteur ne couvrira que l'implémentation du polymorphisme que les compilateurs Microsoft et Borland utilisent. Cette implémentation n'a pas changé depuis des années et ne changera probablement pas dans un futur proche.

Avec ces changements, la sortie du programme change :

```
Size of a: 8 Déplacement de ad : 4
Size of b: 12 Déplacement de ad : 4 Déplacement de bd : 8
A::m()
B::m()
```

Maintenant, le second appel à `f` appelle la méthode `B::m()` car on lui passe un objet `B`. Ce n'est pas le seul changement cependant. La taille d'un `A` vaut maintenant 8 (et 12 pour `B`). De plus, le déplacement de `ad` vaut maintenant 4, plus 0. Qu'y a-t-il au déplacement 0 ? La réponse à cette question est liée à la façon dont est implémenté le polymorphisme.

```
1  #include <cstddef>
2  #include <iostream>
3  using namespace std;
4
5  class A {
6  public:
7    void __cdecl m() { cout << "A::m()" << endl; }
8    int ad;
9  };
10
11 class B : public A {
12 public:
13   void __cdecl m() { cout << "B::m()" << endl; }
14   int bd;
15 };
16
17 void f( A * p )
18 {
19   p->ad = 5;
20   p->m();
21 }
22
23 int main()
24 {
25   A a;
26   B b;
27   cout << "Taille de a : " << sizeof(a)
28        << " Déplacement de ad : " << offsetof(A,ad) << endl;
29   cout << "Taille de b : " << sizeof(b)
30        << " Déplacement de ad : " << offsetof(B,ad)
31        << " Déplacement de bd : " << offsetof(B,bd) << endl;
32   f(&a);
33   f(&b);
34   return 0;
35 }
```

FIGURE 7.19 – Héritage Simple

```
1  _f__FP1A:                        ; nom de fonction décoré
2        push   ebp
3        mov    ebp, esp
4        mov    eax, [ebp+8]        ; eax pointe sur l'objet
5        mov    dword [eax], 5      ; utilisation du déplacement 0 pour ad
6        mov    eax, [ebp+8]        ; passage de l'adresse de l'objet à A::m()
7        push   eax
8        call   _m__1A              ; nom décoré de la méthode A::m()
9        add    esp, 4
10       leave
11       ret
```

FIGURE 7.20 – Code Assembleur pour un Héritage Simple

```
1  class A {
2  public:
3    virtual void __cdecl m() { cout << "A::m()" << endl; }
4    int ad;
5  };
6
7  class B : public A {
8  public:
9    virtual void __cdecl m() { cout << "B::m()" << endl; }
10   int bd;
11 };
```

FIGURE 7.21 – Héritage Polymorphique

```
1  ?f@@YAXPAVA@@@Z:
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp+8]
6        mov    dword [eax+4], 5  ; p->ad = 5;
7
8        mov    ecx, [ebp + 8]    ; ecx = p
9        mov    edx, [ecx]        ; edx = pointeur sur la vtable
10       mov    eax, [ebp + 8]    ; eax = p
11       push   eax               ; empile le pointeur "this"
12       call   dword [edx]       ; appelle la première fonction de la vtable
13       add    esp, 4            ; nettoie la pile
14
15       pop    ebp
16       ret
```

FIGURE 7.22 – Code Assembleur de la Fonction `f()`

Une classe C++ qui a une (ou plusieurs) méthode(s) virutelle(s) a un champ caché qui est un pointeur vers un tableau de pointeurs sur des méthodes[13]. Cette table est souvent appelée la *vtable*. Pour les classes `A` et `B` ce pointeur est stocké au déplacement 0. Les compilateurs Windows placent toujours ce pointeur au début de la classe au sommet de l'arbre d'héritage. En regardant le code assembleur (Figure 7.22) généré pour la fonction `f` (de la Figure 7.19) dans la version du programme avec les méthodes virtuelles, on peut voir que l'appel à la méthode `m` ne se fait pas via une étiquette. La ligne 9 trouve l'adresse de la vtable de l'objet. L'adresse de l'objet est placée sur la pile ligne 11. La ligne 12 appelle la méthode virtuelle en se branchant à la première adresse dans la vtable[14]. Cet appel n'utilise pas d'étiquette, il se branche à l'adresse du code sur lequel pointe `EDX`. Ce type d'appel est un exemple de *liaison tardive* (late binding). La liaison tardive repousse le choix de la méthode à appeler au moment de l'exécution du code. Cela permet d'appeler la méthode correspondant à l'objet. Le cas normal (Figure 7.20) code en dur un appel à une certaine méthode et est appelé *liaison précoce* (early binding), car la méthode est liée au moment de la compilation.

13. Pour les classes sans méthode virtuelle, les compilateurs C++ rendent toujours la classe compatible avec une structure C normale qui aurait les mêmes données membres.

14. Bien sûr, la valeur est déjà dans le registre `ECX`. Elle y a été placée à la ligne 8 et le ligne 10 pourrait être supprimée et la ligne suivante changée de façon à empiler `ECX`. Le code n'est pas très efficace car il a été généré en désactivant les optimisations du compilateur.

```
1  class A {
2  public:
3    virtual void __cdecl m1() { cout << "A::m1()" << endl; }
4    virtual void __cdecl m2() { cout << "A::m2()" << endl; }
5    int ad;
6  };
7
8  class B : public A {     // B hérite du m2() de A
9  public:
10   virtual void __cdecl m1() { cout << "B::m1()" << endl; }
11   int bd;
12 };
13 /* affiche la vtable de l'objet fourni */
14 void print_vtable( A * pa )
15 {
16   // p voit pa comme un tableau de doubles mots
17   unsigned * p = reinterpret_cast<unsigned *>(pa);
18   // vt voit la vtable comme un tableau de pointeurs
19   void ** vt = reinterpret_cast<void **>(p[0]);
20   cout << hex << "adresse de la vtable = " << vt << endl;
21   for( int i=0; i < 2; i++ )
22     cout << "dword " << i << " : " << vt[i] << endl;
23
24   // appelle les fonctions virtuelle d'une façon ABSOLUMENT non portable
25   void (*m1func_pointer)(A *);   // variable pointeur de fonction
26   m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
27   m1func_pointer(pa);            // appelle m1 via le pointeur de fonction
28
29   void (*m2func_pointer)(A *);   // variable pointeur de fonction
30   m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
31   m2func_pointer(pa);            // appelle m2 via le pointeur de fonction
32 }
33
34 int main()
35 {
36   A a;    B b1;   B b2;
37   cout << "a: " << endl;   print_vtable(&a);
38   cout << "b1: " << endl; print_vtable(&b1);
39   cout << "b2: " << endl; print_vtable(&b2);
40   return 0;
41 }
```

FIGURE 7.23 – Exemple plus compliqué

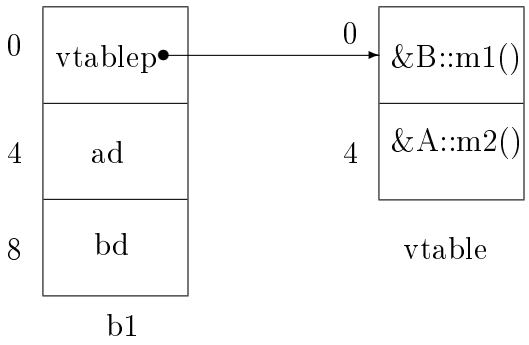

FIGURE 7.24 – Représentation interne de `b1`

Le lecteur attentif se demandera pourquoi les méthodes de classe de la Figure 7.21 sont explicitement déclarées pour utiliser la convention d'appel C en utilisant le mot-clé `__cdecl`. Par défaut, Microsoft utilise une convention différente de la convention C standard pour les méthodes de classe C++. Il passe le pointeur sur l'objet sur lequel agit la méthode via le registre `ECX` au lieu d'utiliser la pile. La pile est toujours utilisée pour les autres paramètres explicites de la méthode. Le modificateur `__cdecl` demande l'utilisation de la convention d'appel C standard. Borland C++ utilise la convention d'appel C par défaut.

Observons maintenant un exemple légèrement plus compliqué (Figure 7.23). Dans celui-là, les classes `A` et `B` ont chacune deux méthodes : `m1` et `m2`. Souvenez vous que comme la classe `B` ne définit pas sa propre méthode `m2`, elle hérite de la méthode de classe de `A`. La Figure 7.24 montre comment l'objet `b` apparaît en mémoire. La Figure 7.25 montre la sortie du programme. Tout d'abord, regardons l'adresse de la vtable de chaque objet. Les adresses des deux objets `B` sont les mêmes, ils partagent donc la même vtable. Une vtable appartient à une classe pas à un objet (comme une donnée membre `static`). Ensuite, regardons les adresses dans les vtables. En regardant la sortie assembleur, on peut déterminer que le pointeur sur la méthode `m1` est au déplacement 0 (ou dword 0) et celui sur `m2` est au déplacement 4 (dword 1). Les pointeurs sur la méthode `m2` sont les mêmes pour les vtables des classes `A` et `B` car la classe `B` hérite de la méthode `m2` de la classe `A`.

Les lignes 25 à 32 montrent comment l'on pourrait appeler une fonction virtuelle en lisant son adresse depuis la vtable de l'objet [15]. L'adresse de la méthode est stockée dans un pointeur de fonction de type C avec un pointeur *this* explicite. D'après la sortie de la Figure 7.25, on peut voir comment cela fonctionne. Cependant, s'il vous plaît, n'écrivez *pas* de code de ce genre ! Il

15. Souvenez vous, ce code ne fonctionne qu'avec les compilateurs MS et Borland, pas *gcc*.

```
a:
Adresse de la vtable = 004120E8
dword 0: 00401320
dword 1: 00401350
A::m1()
A::m2()
b1:
Adresse de la vtable = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
b2:
Adresse de la vtable = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
```

FIGURE 7.25 – Sortie du programme de la Figure 7.23

n'est utilisé que pour illustrer le fait que les méthodes virtuelles utilisent la vtable.

Il y a plusieurs leçons pratiques à tirer de cela. Un fait important est qu'il faut être très attentif lorsque l'on écrit ou que l'on lit des variables de type classe depuis un fichier binaire. On ne peut pas utiliser simplement une lecture ou une écriture binaire car cela lirait ou écrirait le pointeur vtable depuis le fichier ! C'est un pointeur sur l'endroit où la vtable réside dans la mémoire du programme et il varie d'un programme à l'autre. La même chose peut arriver avec les structures en C, mais en C, les structures n'ont des pointeurs que si le programmeur en définit explicitement. Il n'y a pas de pointeurs explicites déclarés dans les classes `A` ou `B`.

Une fois encore, il est nécessaire de réaliser que des compilateurs différents implémentent les méthodes virtuelles différemment. Sous Windows, les objets de la classe COM (Component Object Model) utilisent des vtables pour implémenter les interfaces COM [16]. Seuls les compilateurs qui implémentent les vtables des méthodes virtuelles comme le fait Microsoft peuvent créer des classes COM. C'est pourquoi Borland utilise la même implémentation que Microsoft et une des raisons pour lesquelles *gcc* ne peut pas être utilisé pour

16. Les classes COM utilisent également la convention d'appel `__stdcall`, pas la convention C standard.

créer des classes COM.

Le code des méthodes virtuelles est identique à celui des méthodes non-virtuelles. Seul le code d'appel est différent. Si le compilateur peut être absolument sûr de la méthode virtuelle qui sera appelée, il peut ignorer la vtable et appeler la méthode directement (*p.e.*, en utilisant la liaison précoce).

7.2.6 Autres fonctionnalités C++

Le fonctionnement des autres fonctionnalités du C++ (*p.e.*, le RunTime Type Information, la gestion des exceptions et l'héritage multiple) dépasse le cadre de ce livre. Si le lecteur veut aller plus loin, *The Annotated C++ Reference Manual* de Ellis et Stroustrup et *The Design and Evolution of C++* de Stroustrup constituent un bon point de départ.

Annexe A

Instructions 80x86

A.1 Instructions hors Virgule Flottante

Cette section liste et décrit les actions et les formats des instructions hors virgule flottante de la famille de processeurs Intel 80x86.

Les formats utilisent les abbréviations suivantes :

R	registre général
R8	registre 8 bits
R16	registre 16 bits
R32	registre 32 bits
SR	registre de segment
M	mémoire
M8	octet
M16	mot
M32	double mot
I	valeur immédiate

Elles peuvent être combinées pour les instructions à plusieurs opérandes. Par exemple, le format *R,R* signifie que l'instruction prend deux opérandes de type registre. Beaucoup d'instructions à deux opérandes acceptent les mêmes opérandes. L'abbréviation *O2* est utilisée pour représenter ces opérandes : *R,R R,M R,I M,R M,I*. Si un registre 8 bits ou un octet en mémoire peuvent être utilisés comme opérande, l'abbréviation *R/M8* est utilisée.

Le tableau montre également comment les différents bits du registre FLAGS sont modifiés par chaque instruction. Si la colonne est vide, le bit correspondant n'est pas modifié du tout. Si le bit est toujours positionné à une valeur particulière, un 1 ou un 0 figure dans la colonne. Si le bit est positionné à une valeur qui dépend des opérandes de l'instruction, un *C* figure dans la colonne. Enfin, si le bit est modifié de façon indéfinie, un *?* figure dans la colonne. Comme les seules instructions qui changent le drapeau de direction sont `CLD` et `STD`, il ne figure pas parmi les colonnes FLAGS.

Nom	Description	Formats	Flags					
			O	S	Z	A	P	C
ADC	Ajout avec Retenue	O2	C	C	C	C	C	C
ADD	Addition entière	O2	C	C	C	C	C	C
AND	ET niveau bit	O2	0	C	C	?	C	0
BSWAP	Echange d'Octets	R32						
CALL	Appel de Routine	R M I						
CBW	Conversion Octet-Mot							
CDQ	Conversion DMot-QMot							
CLC	Eteindre la retenue							0
CLD	Eteindre la direction							
CMC	Inverser la retenue							C
CMP	Comparaison Entière	O2	C	C	C	C	C	C
CMPSB	Comparaison d'Octets		C	C	C	C	C	C
CMPSW	Comparaison de Mots		C	C	C	C	C	C
CMPSD	Comparaison de DMots		C	C	C	C	C	C
CWD	Conversion Mot-DMot dans DX:AX							
CWDE	Conversion Mot-DMot dans EAX							
DEC	Decrémentation d'Entier	R M	C	C	C	C	C	
DIV	Division non Signée	R M	?	?	?	?	?	?
ENTER	Création de cadre de pile	I,0						
IDIV	Division Signée	R M	?	?	?	?	?	?
IMUL	Multiplication Signée	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
INC	Incrémentation d'Entier	R M	C	C	C	C	C	
INT	Générer une Interruption	I						
JA	Saut si Au-dessus	I						
JAE	Saut si Au-Dessus ou Egal	I						
JB	Saut si En-Dessous	I						

Nom	Description	Formats	Flags					
			O	S	Z	A	P	C
JBE	Saut si En-Dessous ou Egal	I						
JC	Saut si Retenue	I						
JCXZ	Saut si CX = 0	I						
JE	Saut si Egal	I						
JG	Saut si Supérieur	I						
JGE	Saut si Supérieur ou Egal	I						
JL	Saut si Inférieur	I						
JLE	Saut si Inférieur ou Egal	I						
JMP	Saut Inconditionnel	R M I						
JNA	Saut si Non Au-Dessus	I						
JNAE	Saut si Non Au-Dessus ou Egal	I						
JNB	Saut si Non En-Dessous	I						
JNBE	Saut si Non En-Dessous ou Egal	I						
JNC	Saut si pas de Retenue	I						
JNE	Saut si Non Egal	I						
JNG	Saut si Non Supérieur	I						
JNGE	Saut si Non Supérieur ou Egal	I						
JNL	Saut si Non Inférieur	I						
JNLE	Saut si Non Inférieur ou Egal	I						
JNO	Saut si Non Overflow	I						
JNS	Saut si Non Signe	I						
JNZ	Saut si Non Zéro	I						
JO	Saut si Overflow	I						
JPE	Saut si Pair	I						
JPO	Saut si Impair	I						
JS	Saut si Signe	I						
JZ	Saut si Zéro	I						
LAHF	Charge FLAGS dans AH							
LEA	Charge l'Adresse Effective	R32,M						

Nom	Description	Formats	Flags					
			O	S	Z	A	P	C
LEAVE	Quitter le Cadre de Pile							
LODSB	Charger un Octet							
LODSW	Charger un Mot							
LODSD	Charger un Double Mot							
LOOP	Boucler	I						
LOOPE/LOOPZ	Boucler si Egal	I						
LOOPNE/LOOPNZ	Boucler si Non Egal	I						
MOV	Déplacement de données	O2 SR,R/M16 R/M16,SR						
MOVSB	Déplacement d'Octet							
MOVSW	Déplacement de Mot							
MOVSD	Déplacement de Double Mot							
MOVSX	Déplacement Signé	R16,R/M8 R32,R/M8 R32,R/M16						
MOVZX	Déplacement Non Signé	R16,R/M8 R32,R/M8 R32,R/M16						
MUL	Multiplication Non Signée	R M	C	?	?	?	?	C
NEG	Inverser	R M	C	C	C	C	C	C
NOP	Pas d'Opération							
NOT	Complément à 1	R M						
OR	OU niveau bit	O2	0	C	C	?	C	0
POP	Retirer de la pile	R/M16 R/M32						
POPA	Tout retirer de la pile							
POPF	Retirer FLAGS de la pile		C	C	C	C	C	C
PUSH	Empiler	R/M16 R/M32 I						
PUSHA	Tout Empiler							
PUSHF	Empiler FLAGS							
RCL	Rotation à Gauche avec Retenue	R/M,I R/M,CL	C					C

Nom	Description	Formats	Flags					
			O	S	Z	A	P	C
RCR	Rotation à Droite avec Retenue	R/M,I R/M,CL	C					C
REP	Répéter							
REPE/REPZ	Répéter Si Egal							
REPNE/REPNZ	Répéter Si Non Egal							
RET	Retour							
ROL	Rotation à Gauche	R/M,I R/M,CL	C					C
ROR	Rotation à Droite	R/M,I R/M,CL	C					C
SAHF	Copie AH dans FLAGS			C	C	C	C	C
SAL	Décalage Arithmétique à Gauche	R/M,I R/M, CL						C
SBB	Soustraire Avec Retenue	O2	C	C	C	C	C	C
SCASB	Recherche d'Octet		C	C	C	C	C	C
SCASW	Recherche de Mot		C	C	C	C	C	C
SCASD	Recherche de Double-Mot		C	C	C	C	C	C
SETA	Allumer Si Au-Dessus	R/M8						
SETAE	Allumer Si Au-Dessus ou Egal	R/M8						
SETB	Allumer Si En-Dessous	R/M8						
SETBE	Allumer Si En-Dessous ou Egal	R/M8						
SETC	Allumer Si Retenue	R/M8						
SETE	Allumer Si Egal	R/M8						
SETG	Allumer Si Plus Grand	R/M8						
SETGE	Allumer Si Plus Grand ou Egal	R/M8						
SETL	Allumer Si Plus Petit	R/M8						
SETLE	Allumer Si Plus Petit ou Egal	R/M8						
SETNA	Allumer Si Non Au Dessus	R/M8						
SETNAE	Allumer Si Non Au-Dessus ou Egal	R/M8						
SETNB	Allumer Si Non En-Dessous	R/M8						

Nom	Description	Formats	Flags					
			O	S	Z	A	P	C
SETNBE	Allumer Si Non En-Dessous ou Egal	R/M8						
SETNC	Allumer Si Pas de Retenue	R/M8						
SETNE	Allumer Si Non Egal	R/M8						
SETNG	Allumer Si Non Plus Grand	R/M8						
SETNGE	Allumer Si Non Plus Grand ou Egal	R/M8						
SETNL	Allumer Si Non Inférieur	R/M8						
SETNLE	Allumer Si Non Inférieur ou Egal	R/M8						
SETNO	Allumer Si Non Overflow	R/M8						
SETNS	Allumer Si Non Signe	R/M8						
SETNZ	Allumer Si Non Zéro	R/M8						
SETO	Allumer Si Overflow	R/M8						
SETPE	Allumer Si Parité Paire	R/M8						
SETPO	Allumer Si Parité Impaire	R/M8						
SETS	Allumer Si Signe	R/M8						
SETZ	Allumer Si Zéro	R/M8						
SAR	Décalage Arithmétique à Droite	R/M,I R/M, CL						C
SHR	Décalage Logique à Droite	R/M,I R/M, CL						C
SHL	Décalage Logique à Gauche	R/M,I R/M, CL						C
STC	Allumer la Retenue							1
STD	Allumer le Drapeau de Direction							
STOSB	Stocker l'Octet							
STOSW	Stocker le Mot							
STOSD	Stocker le Double-Mot							
SUB	Soustraction	O2	C	C	C	C	C	C
TEST	Comparaison Logique	R/M,R R/M,I	0	C	C	?	C	0
XCHG	Echange	R/M,R R,R/M						

Nom	Description	Formats	Flags					
			O	**S**	**Z**	**A**	**P**	**C**
XOR	Ou Exclusif Niveau Bit	O2	0	C	C	?	C	0

A.2 Instruction en Virgule Flottante

Dans cette section, la plupart des instructions du coprocesseur mathématique du 80x86 son décrites. La section description décrit brièvement l'opération effectuée par l'instruction. Pour économiser de la place, il n'est pas précisé si l'instruction décale la pile ou non.

La colonne format indique le type d'opérande pouvant être utilisé avec chaque instruction. Les abbréviations suivantes sont utilisées :

STn	Registre du coprocesseur
F	Nombre simple précision en mémoire
D	Nombre double précision en mémoire
E	Nombre précision étendue en mémoire
I16	Mot entier en mémoire
I32	Double mot entier en mémoire
I64	Quadruple mot entier en mémoire

Les instructions nécessitant un Pentium Pro ou ou supérieur sont signalées par une astérisque(*).

Instruction	Description	Format
FABS	ST0 = |ST0|	
FADD *src*	ST0 += *src*	STn F D
FADD *dest*, ST0	*dest* += ST0	STn
FADDP *dest*[,ST0]	*dest* += ST0	STn
FCHS	ST0 = −ST0	
FCOM *src*	Compare ST0 et *src*	STn F D
FCOMP *src*	Compare ST0 et *src*	STn F D
FCOMPP *src*	Compare ST0 et ST1	
FCOMI* *src*	Compare dans FLAGS	STn
FCOMIP* *src*	Compare dans FLAGS	STn
FDIV *src*	ST0 /= *src*	STn F D
FDIV *dest*, ST0	*dest* /= ST0	STn
FDIVP *dest*[,ST0]	*dest* /= ST0	STn
FDIVR *src*	ST0 = *src*/ST0	STn F D
FDIVR *dest*, ST0	*dest* = ST0/*dest*	STn
FDIVRP *dest*[,ST0]	*dest* = ST0/*dest*	STn
FFREE *dest*	Marquer comme vide	STn
FIADD *src*	ST0 += *src*	I16 I32
FICOM *src*	Compare ST0 et *src*	I16 I32
FICOMP *src*	Compare ST0 et *src*	I16 I32
FIDIV *src*	ST0 /= *src*	I16 I32
FIDIVR *src*	ST0 = *src*/ST0	I16 I32

Instruction	Description	Format
FILD *src*	Place *src* sur la Pile	I16 I32 I64
FIMUL *src*	ST0 *= *src*	I16 I32
FINIT	Initialise le Coprocesseur	
FIST *dest*	Stocke ST0	I16 I32
FISTP *dest*	Stocke ST0	I16 I32 I64
FISUB *src*	ST0 -= *src*	I16 I32
FISUBR *src*	ST0 = *src* - ST0	I16 I32
FLD *src*	Place *src* sur la Pile	STn F D E
FLD1	Place 1.0 sur la Pile	
FLDCW *src*	Charge le Registre du Mot de Contrôle	I16
FLDPI	Place π sur la Pile	
FLDZ	Place 0.0 sur la Pile	
FMUL *src*	ST0 *= *src*	STn F D
FMUL *dest*, ST0	*dest* *= ST0	STn
FMULP *dest*[,ST0]	*dest* *= ST0	STn
FRNDINT	Arrondir ST0	
FSCALE	ST0 = ST0 $\times 2^{\lfloor \text{ST1} \rfloor}$	
FSQRT	ST0 = $\sqrt{\text{ST0}}$	
FST *dest*	Stocke ST0	STn F D
FSTP *dest*	Stocke ST0	STn F D E
FSTCW *dest*	Stocke le Registre du Mot de Contrôle	I16
FSTSW *dest*	Stocke le Registre du Mot de Statut	I16 AX
FSUB *src*	ST0 -= *src*	STn F D
FSUB *dest*, ST0	*dest* -= ST0	STn
FSUBP *dest*[,ST0]	*dest* -= ST0	STn
FSUBR *src*	ST0 = *src*-ST0	STn F D
FSUBR *dest*, ST0	*dest* = ST0-*dest*	STn
FSUBP *dest*[,ST0]	*dest* = ST0-*dest*	STn
FTST	Compare ST0 avec 0.0	
FXCH *dest*	Echange ST0 et *dest*	STn