

UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Architecture 1

Assembleur 80386

E. Dall'Olio, N. González-Deleito, 2003–2005.
E. Dall'Olio, M. De Wulf, N. González-Deleito, 2002–2003.
E. Dall'Olio, N. González-Deleito, G. Othmezzouri, 2000–2002.
L. Franck, 1998–1999.

Table des matières

Avant-propos	7
1 Les bases	9
2 Introduction à l'assembleur	11
2.1 L'architecture du 80386	11
2.1.1 Les registres	12
2.1.2 La mémoire	14
2.1.3 La boucle d'interprétation	14
2.2 Les langages	14
2.3 Utilisation de l'assembleur dans le cadre du TP	16
3 NASM	17
3.1 Les instructions	17
3.2 Les immédiats	18
3.2.1 Les nombres	18
3.2.2 Les expressions	18
3.3 Les sections	18
3.4 Les pseudo-instructions	19
3.5 Les directives	19
3.6 Autres mots-clés	20
3.7 Canevas de fichier source	20
3.8 Production d'un fichier exécutable	22
4 Les instructions de transfert de données	23
4.1 Les transferts simples	23
4.2 Les transferts avec extension	23

4.3	Les échanges	24
5	Les modes d'adressage	25
5.1	L'adressage par immédiat	25
5.2	L'adressage par registre	25
5.3	L'adressage mémoire	26
5.3.1	L'adressage direct	26
5.3.2	L'adressage indirect	26
6	Les instructions de manipulation de bits	29
6.1	L'arithmétique booléenne	29
6.2	Les masques	29
6.2.1	Peinture au pochoir sur une surface propre	29
6.2.2	Peinture au pochoir sur une surface très sale	30
6.3	Les glissements et les rotations	31
6.3.1	Les glissements	31
6.3.2	Les rotations	33
6.4	Les enregistrements	34
6.5	Les instructions spécialisées	34
6.5.1	Les instructions de recherche de bits	34
6.5.2	Les instructions de test de bits	35
6.5.3	Les glissements en double-précision	35
7	Les instructions arithmétiques	37
7.1	Codage des nombres	37
7.1.1	Notation avec bit de signe	37
7.1.2	Notation en complément à un	38
7.1.3	Notation en complément à deux	39
7.2	Les instructions arithmétiques	39
7.2.1	L'addition	39
7.2.2	La soustraction	40
7.2.3	La multiplication	41
7.2.4	La division	43
7.2.5	Le passage à l'opposé	44
7.3	Les instructions d'extension de signe	44

8 Les choix et les boucles	45
8.1 Les instructions de comparaison	45
8.2 Les instructions de saut	46
8.3 Les structures alternatives	48
8.4 Les structures itératives	48
8.5 Les instructions <i>set byte on condition</i>	49
9 Les procédures	51
9.1 La pile	51
9.2 L'appel à une procédure	52
9.3 Le passage de paramètres	53
9.4 Les variables locales	54
9.5 La valeur de retour d'une fonction	54
9.6 Le retour de procédure	54
9.7 Canevas général	54
9.8 L'édition des liens	56
A Effets des instructions sur les flags	57
Index d'instructions	59
Bibliographie	61

Avant-propos

Ces notes résultent, en partie, de la fusion des notes écrites par Laurent Franck en 1999 pour le processeur 8086 et du syllabus d'exercices écrit par Emmanuel Dall'Olio, Nicolás González-Deleito et Gabriel Othmezouri en 2001.

Nous aimerions remercier Raymond Devillers pour les avoir soigneusement relues.

Considérations pratiques

Nous allons voir dans le cadre du TP l'assembleur pour le processeur 80386 d'Intel. D'un point de vue pratique, nous utiliserons la version 0.98.34 de l'assembleur NASM¹, ainsi que la version 3.2 du compilateur C++ GCC² pour réaliser des entrées/sorties. Le système d'exploitation employé sera GNU/Linux.

Même si dans les premiers exercices l'interaction avec du code écrit en C++ se fera *en aveugle*, les détails seront expliqués en profondeur lorsque les appels de procédure seront étudiés au cours des dernières séances du TP.

¹NASM est disponible à l'adresse nasm.sourceforge.net.

²GCC est disponible à l'adresse gcc.gnu.org.

Chapitre 1

Les bases

Lorsque nous utilisons un mot tel que « eau », il nous est difficile de détacher le concept qu'il y a derrière ce mot et sa représentation symbolique sous forme de nom commun dans la langue française. Pourtant ce même concept peut être retrouvé dans toutes les autres langues sous une forme différente. Il est donc important de bien faire la distinction entre le *concept* et sa *représentation*.

Le même type de distinction existe pour les nombres. Dans ce cas-ci, la langue utilisée pour exprimer des concepts tels que mille cinq cent quinze ou septante-deux est appelée la *base*. Au même titre que des lettres (par exemple, celles de l'alphabet de la langue française) sont utilisées pour construire des mots, des symboles appelés *chiffres* sont utilisés pour construire la représentation d'un nombre en numération de position dans une base. Chaque base est, entre autres, définie par la taille de son alphabet.

Les bases les plus utilisées en informatique sont le binaire (alphabet de taille 2, ou plus succinctement base 2), l'octal (base 8), le décimal (base 10) et l'hexadécimal (base 16). Dans ce dernier cas, les chiffres usuels ne suffisant pas, on les complète avec les six premières lettres de l'alphabet ($A_{16} = 10_{10}, \dots, F_{16} = 15_{10}$).

L'être humain (dans une grande proportion) utilise la base 10 comme langage universel des nombres. Lorsque nous apprenons une langue étrangère (par exemple, l'anglais), un des principes de cet apprentissage est d'effectuer une projection des mots appris dans cette langue vers le français (water $\rightarrow$ eau). Il en va de même lorsque nous apprenons une base étrangère (c'est-à-dire différente de 10), excepté que dans le cas des nombres, faire la projection vers la base 10 est beaucoup plus simple que de passer de l'anglais au français.

Pour traduire (convertir) un nombre exprimé en base B , n_B , vers la base B' , on calcule d'abord la valeur n associée à la représentation n_B à l'aide de la formule suivante :

$$n = \sum_{i=0}^{M-1} n_i * B^i,$$

où les n_i , avec $i \in [0, M - 1]$, représentent les M chiffres de la représentation

de n en base B . Ensuite, nous pouvons représenter ce nombre en base B' grâce à la même formule :

$$n = \sum_{i=0}^{M'-1} n'_i * B'^i,$$

où les n'_i , avec $i \in [0, M' - 1]$, représentent les M' chiffres de la représentation de n en base B' , avec $0 \leq n'_i < B'$. En pratique, cette deuxième étape est réalisée en procédant à des divisions successives par B' , les restes de ces divisions constituant l'image miroir de la représentation du nombre considéré dans la nouvelle base.

Par exemple, si nous nous donnons le nombre 43_7 et que nous voulons l'écrire en base 11, la première formule nous indique que le nombre représenté est trente et un ; la seconde, nous permet de l'écrire 29_{11} .

Il va de soi que, pour nous, si l'une des deux bases est la base 10, les opérations ci-dessus sont simplifiées, puisque l'image mentale que nous nous faisons des nombres est, pour des raisons culturelles, leur représentation en base 10.

Notons que le coefficient de la plus grande puissance de la base est appelé le *chiffre le plus significatif* ou *chiffre de poids fort*. De même, le coefficient associé à la plus petite puissance de la base est appelé le *chiffre le moins significatif* ou *chiffre de poids faible*.

Comme nous le verrons au cours des exercices, il existe des méthodes plus simples pour des conversions entre certaines bases (par exemple, de la base 2 vers la base 16).

Chapitre 2

Introduction à l'assembleur

La plupart des ordinateurs personnels actuels sont basés sur des processeurs Intel de la famille 80x86 ou compatibles. Il y a cependant des exceptions, la plus notable étant les Macintosh d'Apple, basés sur des processeurs Motorola.

Nous allons ici nous intéresser plus particulièrement au processeur 80386. Les processeurs ultérieurs de la même famille (80486, Pentium, ...) sont des extensions du 80386, de même que celui-ci est une extension des processeurs antérieurs (8086, 80186 et 80286).

2.1 L'architecture du 80386

Le processeur 80386 d'Intel peut être considéré comme composé essentiellement de trois parties :

- l'unité arithmético-logique (UAL), qui, comme son nom l'indique, réalise des opérations logiques et arithmétiques de base ;
- les registres, des zones de mémoire d'accès rapide et présentes en nombre restreint ;
- et la boucle d'interprétation, un programme « câblé » dans le processeur, exécutant des programmes écrits en langage machine stockés dans la mémoire attachée au processeur.

Le processeur est relié à la mémoire par un *bus* qui permet de transporter des informations par paquets de 8, 16 ou 32 bits.

De plus, il dispose de trois modes de fonctionnement différents :

- le mode réel, dans lequel le processeur se limite à émuler le fonctionnement d'un processeur 8086 ;
- le mode protégé, sous lequel une protection de la mémoire, au niveau matériel, est mise en place afin d'empêcher les applications s'exécutant sur le processeur d'avoir accès à des zones mémoire attribuées à d'autres applications ;
- et le mode 8086 virtuel, permettant d'émuler plusieurs machines 8086 virtuelles séparées, tout en exécutant simultanément des tâches en mode protégé.

Ce système de protection de la mémoire permet de bien faire la différence entre *les programmes système* et *les programmes applicatifs*. Les premiers feront partie du système d'exploitation et auront accès à un ensemble d'instructions privilégiées que les programmes applicatifs ne pourront pas utiliser.

Bien entendu, au cours des exercices nous nous limiterons à ce second type de programmes.

Il est important de remarquer que, sous le système d'exploitation GNU/Linux, le processeur travaille en mode protégé et en *mode 32 bits*. Comme nous le verrons plus tard, afin de ne pas multiplier le nombre de codes opératoires existants, certaines instructions peuvent travailler indistinctement sur des registres de 16 ou 32 bits. Leur fonctionnement exact varie donc selon que le processeur se trouve en mode 16 ou 32 bits. Il est donc bon de savoir que, sous GNU/Linux, toutes ces instructions seront interprétées dans leur format 32 bits.

2.1.1 Les registres

Le processeur 80386 met à la disposition des programmes un nombre assez restreint de registres. Il est donc important de les utiliser au mieux, pour éviter des accès inutiles à la mémoire. Remarquons qu'un accès à la mémoire nécessite beaucoup plus de temps qu'un simple accès à un registre.

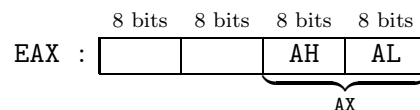
L'architecture dite *orthogonale* du 80386 permet à certains de ces registres d'être utilisés dans plusieurs contextes distincts. Par exemple, comme nous le verrons plus tard, n'importe lequel des registres généraux pourra être employé lors d'un adressage indirect. Certaines des contraintes imposées par le processeur 8086 n'auront donc plus lieu pour le processeur 80386.

Les registres généraux

Les registres généraux sont au nombre de 4 et sont tous composés de 32 bits. Ils sont identifiés en assembleur par un nom à trois caractères : EAX, EBX, ECX et EDX.

Leurs 16 premiers bits, appelés parties basses, sont identifiés, respectivement, par les noms de registre AX, BX, CX et DX. Il n'est par contre pas possible d'accéder de façon directe aux parties hautes correspondantes.

Les 8 premiers bits de AX, BX, CX et DX constituent, respectivement, les registres AL, BL, CL et DL. Symétriquement, les 8 derniers bits de AX, BX, CX et DX forment, respectivement, les registres AH, BH, CH et DH.



Les registres de segment

Il y a 6 registres de segment. Ce sont des registres de 16 bits, identifiés par les noms CS, DS, ES, FS, GS et SS.

Ils sont liés à la gestion de la mémoire. CS fait référence à l'endroit où se trouve le code exécutable d'un programme. SS se réfère à l'endroit où se trouve la pile associée à ce programme. Les 4 autres registres permettent de faire référence aux données associées au programme qui s'exécute.

Dans les modes réel ou 8086 virtuel, chacun de ces registres donne (après ajout de 4 bits nuls comme bits les moins significatifs) l'adresse de début d'une zone appelée *segment* servant de base pour les calculs d'adresse au sein de ce segment. Dans le mode protégé, ils jouent le rôle d'un index au sein d'une table décrivant les différentes zones mémoire associées au programme s'exécutant. La structure de ce type de table dépassant les limites de ce cours, nous ne donnerons donc pas plus de détails sur ce sujet.

Les registres d'indilage

Destinés essentiellement à indiquer des éléments en mémoire, les registres d'indilage sont au nombre de 4 et ont tous 32 bits. Ils sont identifiés par les noms ESI, EDI, ESP et EBP.

Même si leurs 16 bits de poids faible peuvent être identifiés, respectivement, par les noms SI, DI, SP et BP, nous utiliserons la plupart du temps les versions 32 bits de ces registres.

Les registres de contrôle

Les registres EIP et EFLAGS, tous deux sur 32 bits, représentent respectivement le pointeur vers l'instruction courante et un ensemble de variables à valeurs booléennes décrivant l'état courant du processeur.

Comme pour les autres registres de 32 bits, IP et FLAGS représentent respectivement les 16 premiers bits de ces deux registres.

Il est à noter qu'aucun de ces quatre noms de registre ne peut être directement utilisé dans une instruction assembleur.

Comme nous le verrons dans la suite, certains bits du registre EFLAGS sont modifiés au cours de l'exécution de certaines instructions, et peuvent être testés par la suite. Parmi ces bits, nous pouvons mentionner :

- le *carry flag* (CF), signalant un dépassement non signé ;
- le *parity flag* (PF), précisant si le nombre de bits à 1 dans le byte le moins significatif venant d'être traité est pair ou impair ;
- le *zero flag* (ZF), indiquant un résultat nul ;
- le *sign flag* (SF), donnant le signe du résultat de la dernière opération arithmétique ;

- et le *overflow flag* (OF), signalant un dépassement signé.

2.1.2 La mémoire

La mémoire est un espace de stockage d'informations qui est divisé en cases, chaque case pouvant contenir une quantité fixe d'information. Dans le cas du 80386, chaque case de la mémoire contient un byte, ce qui équivaut à une valeur entre 0 et 255. Ces cases sont numérotées de 0 jusqu'à 4.294.967.295, c'est-à-dire $2^{32} - 1$, dans les limites de la taille de la mémoire physique installée.

Ce numéro de case s'appelle l'*adresse* de la case mémoire. Lors de la lecture ou de l'écriture d'une information dans une case mémoire, il faut toujours fournir :

- dans le cas d'une lecture, l'adresse de la case dont l'information doit être lue ;
- dans le cas d'une écriture, l'adresse de la case où l'information doit être écrite, ainsi que cette information.

2.1.3 La boucle d'interprétation

La boucle d'interprétation du processeur se divise en 6 étapes :

1. Prendre dans la mémoire l'instruction suivante à exécuter.
2. Décoder cette instruction, en particulier séparer l'opérateur des opérandes.
3. Récupérer les valeurs des opérandes. Cela consiste, par exemple, à prendre la valeur d'un registre ou d'une case mémoire.
4. Mettre à jour le pointeur indiquant l'instruction courante à exécuter.
5. Exécuter l'opération.
6. Stocker le résultat dans l'endroit approprié (dans un registre ou en mémoire).

2.2 Les langages

Le processeur 80386 peut être programmé dans une multitude de langages. Cependant, tout comme nous avons une langue maternelle, le 80386 a également son langage natif (et rudimentaire) qui est le *langage machine 80386*. Dès lors, lorsque nous programmons en C++, par exemple, il nous faut un mécanisme qui permette de traduire du C++ vers le langage machine. Ce mécanisme, selon la manière dont il fonctionne, s'appelle compilateur ou interpréteur. Remarquons au passage qu'un compilateur ou un interpréteur est également un programme fonctionnant sur l'ordinateur, et qui lui-même a dû être écrit à un moment ou à un autre ... Cela ressemble un peu au problème de la poule et de l'oeuf.

Dans le cas de l'interpréteur, chaque ligne du programme est lue, traduite en langage machine et ensuite exécutée par le processeur. Si le programme doit passer plusieurs fois par la même ligne, elle sera traduite plusieurs fois.

Dans le cas du compilateur, le programme est traduit complètement en langage machine avant que sa première ligne ne soit exécutée. L'avantage se trouve dans un gain de vitesse au niveau de l'exécution, puisque la traduction a été effectuée préalablement.

En fonction de leur structure, certains langages sont plus adaptés à être interprétés ou compilés. Dans le cas d'un langage compilé, il est très important de bien discerner ce qui est fait lors de la phase de traduction et lors de l'exécution. Entre autres, une fois le programme traduit, tous les changements effectués sur le programme source ne seront pris en compte que lors de la prochaine traduction.

Nous l'avons dit précédemment, le langage machine est très rudimentaire. Ce n'est pas le cas de langages tels que le C++, qui dispose de structures algorithmiques et de types de données évolués. Cette différence en complexité permet de classer les langages en deux catégories : les langages de haut niveau tels que le C++ et les langages de bas niveau tels que le langage machine et l'assembleur.

Il faut remarquer que la compilation (ou l'interprétation) a toujours pour but de passer d'un langage évolué vers un langage moins évolué. Par ailleurs, en fonction de l'écart d'évolution des deux langages, le travail de l'interpréteur ou du compilateur sera plus ou moins complexe.

Bien que l'assembleur et le langage machine soient tous les deux des langages de bas niveau, nous pouvons dire que l'assembleur est un peu moins rudimentaire que le langage machine. Cependant, la relation qui existe entre ces deux langages est spéciale : l'assembleur dans sa version de base n'est rien d'autre qu'une version plus lisible du langage machine. Il n'y a donc pas de support pour des structures algorithmiques plus évoluées.

En effet, les programmes en langage machine sont écrits comme une suite de 0 et de 1. Prenons comme exemple une instruction qui va transférer le contenu du registre AX dans le registre BX. En langage machine cela s'écrirait :

10001001	11	000	011
----------	----	-----	-----

où, en lisant de gauche à droite :

- la première série de bits indique l'*opérateur*, 10001001 voulant dire *transfert depuis un registre* ;
- la deuxième série, 11, indique que la destination est aussi un registre ;
- la troisième série, 000, indique que l'opérande *source* correspond au registre AX ;
- la dernière série, 011, indique que l'opérande *destination* correspond au registre BX.

Nous pouvons facilement imaginer à quel point il est fastidieux d'écrire des programmes en langage machine (et d'y repérer des fautes) ! L'assembleur permet de pallier ce problème, car il permet d'écrire :

MOV	BX,	AX
-----	-----	----

... qui est tout de même plus lisible. Le « compilateur » d'assembleur vers langage machine (qui par un mauvais coup du sort s'appelle également *assembleur*) se chargera de traduire **MOV** en 10001001 11, **BX** en 011 et **AX** en 000.

2.3 Utilisation de l'assembleur dans le cadre du TP

Même s'il est plus facile d'écrire des programmes en assembleur qu'en langage machine, programmer en assembleur n'est pas une sinécure. Pour cette raison, l'usage de l'assembleur est souvent réservé à l'écriture

- de (morceaux de) systèmes d'exploitation ;
- de (morceaux de) compilateurs ;
- de pilotes (*drivers*) pour des périphériques ;
- ou de routines optimisées.

Dans ce TP, nous utiliserons l'assembleur pour réaliser des opérations communes qui seraient normalement écrites en langage de haut niveau. Même si cette utilisation de l'assembleur peut paraître artificielle, elle constitue un excellent support pour mieux comprendre ce qui se passe au niveau du processeur lorsqu'un programme s'exécute.

Chapitre 3

NASM

Pour traiter des programmes en langage assembleur nous utiliserons l'assembleur NASM. Ce chapitre est donc consacré à la syntaxe employée par cet outil, ainsi qu'aux étapes à suivre pour produire un fichier exécutable à partir d'un fichier source (créé au moyen de n'importe quel éditeur de texte).

3.1 Les instructions

Dans tout code source NASM, chaque ligne est structurée en quatre zones séparées par des espaces :

```
label: instruction opérande(s) ; commentaire
```

La première zone contient une chaîne de caractères, appelée *label*, permettant d'identifier la position de l'instruction en question. Le premier caractère du label doit être une lettre, un `_` ou un `?`. Les caractères restants peuvent être des lettres, chiffres, `_`, `$`, `#`, `@`, `~` et `?`. Les mots-clés (mnémoniques d'instructions, noms de registres, ...) sont bien sûr interdits. Un style de programmation propre impose que ce label soit suivi d'un `:`. Il conviendra également de choisir intelligemment ces labels, de façon à rappeler leur usage. Nous verrons aux chapitres 8 et 9 dans quels cas de tels identifiants sont utiles.

Les deux zones suivantes se composent respectivement d'un mnémonique d'instruction et des éventuels opérandes prévus pour cette instruction. Deux ou plusieurs opérandes sont séparés par des virgules. Selon l'instruction, un opérande peut être soit un registre, soit l'adresse ou le contenu d'une zone en mémoire, soit une constante (ces dernières étant également appelées *opérandes immédiats*).

La dernière zone contient un commentaire. Elle commence par un `;` et s'étend jusqu'à la fin de la ligne.

Il n'est pas obligatoire que toutes ces zones apparaissent à chaque ligne. Des lignes en blanc, des lignes de commentaire ou ne contenant qu'un label, par exemple, sont permises (ce dernier cas permet d'avoir plusieurs labels pour une même instruction).

Remarquons que NASM n'est insensible aux majuscules et aux minuscules que pour les mots-clés. Pour bien mettre en évidence ces derniers par rapport à d'autres symboles, nous représenterons les mots-clés par des majuscules et tout autre type de symboles par des minuscules. Il s'agit d'une auto-discipline (un style de programmation) courante et bien pratique.

3.2 Les immédiats

3.2.1 Les nombres

Par défaut, tout nombre est représenté sous forme décimale. Cependant, NASM peut également interpréter des nombres sous forme binaire, octale ou hexadécimale. Pour cela, il faut qu'un tel nombre soit immédiatement suivi, respectivement, par les lettres **b**, **q** et **h**. De plus, il est imposé que tous les nombres en base 16 commencent par un chiffre, et ceci afin qu'ils ne soient pas interprétables comme un nouveau symbole ou confondus avec un symbole existant dans le langage (comme pourrait l'être **Ah**, à la fois nom d'un registre et représentation de dix en hexadécimal ; en NASM, ce dernier cas sera représenté par **0Ah**).

NASM accepte également le préfixe **0x** (également utilisé en **C++**) pour représenter des nombres en hexadécimal.

Voici quelques exemples : **3**, **101b**, **11q**, **13h**, **0Bh**, **0xB**.

3.2.2 Les expressions

NASM est capable d'évaluer des expressions arithmétiques au moment de l'assemblage. Voici quelques-uns des opérateurs supportés, par ordre croissant de priorité : **|**, **^** et **&** (respectivement, *ou*, *ou exclusif* et *et*, tous bit à bit), **+**, **-**, *****, **/**, **%** (reste de la division entière ; doit obligatoirement être suivi d'un espace), **+** unaire, **-** unaire et **~** (complément à 1).

3.3 Les sections

Un code source NASM peut être divisé en plusieurs sections. Chacune de ces sections permettra d'indiquer au chargeur et au système d'exploitation comment doivent être interprétées les informations qui y sont contenues. Par exemple, la section **.text** définit, entre autres, une zone dont le contenu peut être exécuté mais ne peut pas être modifié. La section **.data** définit quant à elle une zone modifiable mais non exécutable.

La première de ces deux sections contiendra généralement les instructions d'un programme, tandis que la seconde se prêtera bien pour accueillir les variables éventuellement utilisées par ce programme.

Une section est déclarée à l'aide du mot-clé **SECTION**, suivi du nom de la section, dans notre cas, soit `.text` soit `.data`. Tout ce qui suit ce mot-clé fait partie de la section correspondante, et cela soit jusqu'à la prochaine occurrence de **SECTION**, soit jusqu'à la fin du fichier source. Notons qu'il est tout à fait possible, par exemple, de commencer par définir le début de la section `.text`, puis définir la section `.data` et enfin compléter la section de code. Plusieurs occurrences de **SECTION** avec le même nom de section peuvent donc exister au sein d'un même fichier source, les contenus étant alors concaténés.

3.4 Les pseudo-instructions

En plus des instructions propres à un processeur, NASM définit plusieurs mots-clés, appelés pseudo-instructions, permettant de remplir le contenu d'une section. Parmi ceux-ci nous détaillons **DB**, **DW** et **DD**, qui vont nous servir pour définir des zones en mémoire pour des variables. Ces trois pseudo-instructions initialisent respectivement des zones de taille égale à un byte, un mot (deux bytes) et un double-mot (quatre bytes). Comme les instructions, elles peuvent être précédées d'un label (ce qui permet de nommer la zone qui est définie) et suivies d'un commentaire. Pour bien faire la différence entre un nom de variable et un identifiant d'instruction, nous omettons le caractère `:` à la fin des labels associés à des variables.

DB, **DW** et **DD** acceptent toutes les trois un nombre illimité d'opérandes, fournissant les valeurs auxquelles les zones mémoire allouées vont être initialisées. Notons enfin que ces différents blocs sont alloués de manière contiguë, dans leur ordre d'apparition.

Voici quelques exemples :

```
i DW 0                                ; un mot initialisé à 0

v DB 0xFFFF, 0x0000, 0xFFFF          ; trois bytes : le premier et
                                       ; le dernier ne sont composés
                                       ; que de bits à 1, le deuxième
                                       ; de bits à 0

w DD 0x01234567                        ; un premier double-mot ...
   DD 0x89ABCDEF                       ; ... suivi d'un deuxième
```

3.5 Les directives

Contrairement aux instructions (qui sont exécutées par le processeur) et aux pseudo-instructions (qui sont interprétées par NASM, mais qui modifient le contenu d'une section), une directive est un mot-clé adressé à NASM pour que la phase d'assemblage se fasse d'une façon bien précise.

Nous avons déjà étudié plus haut un tel mot-clé. Il s'agit de la directive **SECTION**,

qui indique à NASM de générer tout ce qui est nécessaire afin que le système d'exploitation sache, lors de l'exécution d'un programme par exemple, comment gérer chacune des sections qui composent celui-ci.

Nous étudierons la signification des directives `GLOBAL` et `EXTERN`, liées à l'édition des liens, au chapitre 9, lorsque nous verrons comment effectuer des appels de procédure. Cependant, nous les utiliserons dès les premiers exercices pour pouvoir réaliser aisément des entrées/sorties, en interagissant avec du code écrit en C++.

La dernière directive que nous allons employer dans le cadre du TP est `CPU 386`, qui indique à NASM de restreindre le jeu d'instructions disponibles à celles supportées par le processeur 80386 d'Intel.

3.6 Autres mots-clés

Comme nous le verrons dans les prochains chapitres, les différentes combinaisons de type et taille des opérandes admises pour chaque instruction sont bien définies en assembleur. Toutefois, il est parfois possible qu'une même écriture d'une instruction puisse donner lieu à plusieurs interprétations distinctes, suivant la taille de ses opérandes. Afin que NASM puisse encoder correctement cette instruction, nous devons lever une telle ambiguïté en précisant explicitement la taille des opérandes à l'aide d'un des mots-clés suivants : `BYTE`, `WORD` et `DWORD`. Ceux-ci indiquent, respectivement, que la taille de l'opérande qui les suit fait 8, 16 et 32 bits. Nous verrons plus précisément au cours des exercices à quels moments ces mots-clés doivent être employés.

3.7 Canevas de fichier source

Voici le canevas de fichier source NASM que nous utiliserons pour les programmes dont le point d'entrée principal se trouve dans la partie du code écrite en assembleur :

```
CPU 386

GLOBAL main
EXTERN afficherUns32

SECTION .data

        ; variables du programme

SECTION .text
```

```

main:
    ; instructions du programme

    ; (1)
    PUSHA
    PUSH EAX
    CALL afficherUns32
    POP  EAX
    POPA

    ; (2)
    MOV EBX, 0      ; syscall argument: exit code
    MOV EAX, 1      ; system call number: sys_exit
    INT 0x80        ; call kernel

```

Une version électronique de ce canevas est disponible dans le fichier `canevas.asm` fourni avec ces notes.

Le point d'entrée principal est défini à l'aide du label `main:` et de la directive `GLOBAL main`. Les instructions du point (1) appellent une fonction écrite en C++, `afficherUns32`, qui affiche le contenu du registre `EAX` en tant qu'entier non signé.

D'autres fonctions d'affichage et de saisie de données sont définies dans le fichier `IO.cpp` fourni avec ces notes. Pour les utiliser il suffit de changer le nom de l'opérande de l'instruction `CALL` et de la directive `EXTERN` par le nom de la fonction à appeler. Les deux occurrences de `EAX` peuvent bien sûr être remplacées par tout autre registre ou variable de 32 bits.

Il est important de noter que ces fonctions d'entrée/sortie modifient la valeur des registres `EAX`, `EBX`, `ECX` et `EDX`. Si, après un appel à une de ces fonctions, nous voulons conserver les valeurs qui étaient contenues dans ces registres avant l'appel, il faudra les protéger pour qu'elles ne soient pas écrasées. Ceci peut être très facilement réalisé à l'aide des instructions `PUSHA` et `POPA`, que nous étudierons en détail au chapitre 9.

Les trois instructions du point (2) permettent de signaler la fin du programme au système d'exploitation, afin qu'il puisse libérer les ressources utilisées par le programme. Ces instructions seront les dernières instructions qui seront exécutées.

Le fonctionnement exact de ces deux bouts de code deviendra clair lorsque nous aborderons la matière du chapitre 9. Nous verrons également à ce moment-là quelles modifications nous devons apporter au canevas présenté ci-dessus lorsque le point d'entrée principal du programme se trouve dans la partie du code écrite en C++.

3.8 Production d'un fichier exécutable

Pour créer un fichier exécutable à partir des fichiers source, nous allons toujours procéder en deux étapes, peu importe l'endroit où se trouve le point d'entrée principal. Nous commencerons par assembler le module écrit en assembleur à l'aide de la commande :

```
nasm -f elf nom_fichier.asm
```

L'option `-f elf` indique à NASM de produire un fichier objet `nom_fichier.o` de type ELF, compatible avec le système d'exploitation GNU/Linux. Notons au passage que c'est le type du fichier objet utilisé qui définit les sections qui, par défaut, peuvent être utilisés par un programme. Les sections `.text` et `.data` font partie de cette liste de sections pour le format ELF.

Après cette première étape, il reste à compiler le module C++ et à réaliser une édition des liens entre ce module et le fichier objet du module assembleur. Tout ceci peut très facilement être fait par la commande :

```
g++ nom_fichier_1.cpp nom_fichier_2.o
```

qui génère un fichier exécutable nommé `a.out`.

Chapitre 4

Les instructions de transfert de données

4.1 Les transferts simples

L’instruction `MOV` permet de transférer des données à partir d’une zone source vers une zone destination, par blocs de 8, 16 ou 32 bits. Sa syntaxe est la suivante :

`MOV dst, src`

où `dst` est un registre ou une zone en mémoire et `src` un registre, une zone en mémoire ou un immédiat (comme nous allons le détailler au chapitre suivant).

Remarquons que les deux opérandes doivent avoir la même longueur. De plus, il n’est pas possible de faire un transfert entre deux zones en mémoire.

Par exemple, si nous voulons assigner au registre `AX` la valeur 5, nous devons faire un `MOV AX, 5`. De manière similaire, pour copier le contenu du registre `SI` dans `AX`, nous devons faire un `MOV AX, SI`. Par contre, un `MOV AL, SI` ne sera pas une instruction valide puisque `AL` n’a que 8 bits.

4.2 Les transferts avec extension

À partir du processeur 80386, deux instructions de transfert supplémentaires sont apparues. La première, `MOVZX` (*move with zero-extend*), permet de transférer vers l’opérande destination le contenu de l’opérande source, en mettant à zéro les bits de poids fort du premier opérande. Pour cela, la longueur de la destination doit être strictement supérieure à celle de la source. La syntaxe de cette instruction reste très similaire à celle du `MOV` :

`MOVZX dst, src`

où `dst` est un registre de 16 ou de 32 bits et où `src` est un registre ou une zone en mémoire de 8 ou 16 bits.

Par exemple, l'instruction `MOVZX CX, AL` placera dans le registre `CL` le contenu de `AL` et mettra à zéro le registre `CH`.

D'autre part, l'instruction `MOVSX` (*move with sign-extend*), réalise un transfert du contenu de l'opérande source vers l'opérande destination, en faisant une extension du signe du premier opérande vers l'opérande destination (comme nous le verrons au chapitre 7, cela veut dire que les bits de poids fort ajoutés dans la destination seront tous à 0 si la source est positive, et à 1 si elle est négative). La syntaxe de cette instruction est la même que celle de l'instruction `MOVZX`.

4.3 Les échanges

L'instruction `XCHG` échange le contenu de ses deux opérandes. Sa syntaxe est la suivante :

```
XCHG op1, op2
```

où `op1` et `op2` peuvent être un registre ou une zone en mémoire de 8, 16 ou 32 bits de même taille. Elles ne peuvent bien évidemment pas être en même temps deux zones en mémoire, de même qu'aucun des opérandes ne peut être un immédiat.

À titre d'exemple, après l'exécution des trois instructions suivantes nous aurons que le registre `AX` contiendra la valeur 7 et que `BX` vaudra 5 :

```
MOV  AX, 5
MOV  BX, 7
XCHG AX, BX
```

Chapitre 5

Les modes d'adressage

Les règles décrivant comment doivent se faire les accès aux informations disponibles sur le système, notamment celles se trouvant en mémoire, s'appellent *les modes d'adressage*. Le processeur 80386 offre une plus grande variété de modes d'adressage que le 8086. À titre informatif, ce dernier processeur imposait que seuls les registres BX, BP, SI et DI servent lors d'un adressage indirect. Cette contrainte est désormais levée, comme nous le verrons dans le présent chapitre.

Nous allons illustrer les différents modes d'adressage existants à l'aide de l'instruction MOV.

5.1 L'adressage par immédiat

L'adressage par immédiat a lieu lorsqu'un des opérandes est une constante pouvant être évaluée au moment de l'assemblage.

Exemples :

```
MOV AX, 1515
```

```
MOV AX, 5+7
```

5.2 L'adressage par registre

L'adressage par registre a lieu lorsque les opérandes de l'instruction correspondante sont des registres.

Exemple :

```
MOV AX, BX
```

5.3 L'adressage mémoire

Dans le cas de l'adressage mémoire, un des opérandes fait référence à une adresse en mémoire.

5.3.1 L'adressage direct

Dans le mode d'adressage mémoire direct, l'adresse considérée est constante et est déterminée *avant* l'exécution du programme.

Par exemple, l'instruction

```
MOV AX, [buffer]
```

place dans **AX** le contenu du mot de 16 bits débutant à l'adresse en mémoire indiquée par la variable **buffer**.

Il est important de préciser comment ce transfert de deux bytes est réalisé par le processeur. **buffer** est une adresse, c'est-à-dire, le numéro d'une case en mémoire de la taille d'un byte. L'instruction ci-dessus placera le contenu de cette case dans **AL** (la partie basse de **AX**) et le contenu du byte suivant (d'adresse **buffer+1**) dans **AH** (la partie haute de **AX**). Cette façon de faire est appelée *petit boutiste* (*little endian* en anglais).

La façon de faire opposée, placer le contenu de la case en mémoire d'adresse **buffer** dans **AH** et celui de la case **buffer+1** dans **AL**, est appelée *gros boutiste* (*big endian* en anglais). Cette convention est notamment utilisée dans les processeurs de la gamme Motorola.

De la même manière, l'instruction

```
MOV [buffer+2], AX
```

placera dans la case en mémoire d'adresse **buffer+2** le contenu de **AL** et dans celle d'adresse **buffer+3** le contenu de **AH**.

De façon générale, le processeur 80386 fera correspondre les bytes de plus en plus significatifs en mémoire aux bytes de plus en plus significatifs dans un registre.

Avec l'assembleur NASM, l'utilisation de crochets est obligatoire pour accéder au contenu d'une variable. S'ils sont omis, l'occurrence de cette variable fait référence à son adresse. Par exemple, l'instruction

```
MOV EAX, buffer
```

placera dans **EAX** l'adresse de **buffer**. La directive **OFFSET** disponible sur d'autres assembleurs n'est donc pas nécessaire (et n'existe pas) sous NASM.

5.3.2 L'adressage indirect

Lorsque l'adresse d'une zone en mémoire n'est connue qu'au moment de l'exécution du programme, il faut utiliser le mode d'adressage mémoire indirect pour

accéder à cette zone. Cette adresse doit être placée dans un registre avant de pouvoir référencer la zone vers laquelle elle pointe.

Supposons que **EBX** contienne l'adresse de ladite zone en mémoire. L'instruction

MOV AX, [EBX]

placera dans **AX** le contenu des 16 premiers bits accessibles à partir de l'adresse se trouvant dans **EBX** (toujours en suivant la politique petit boutiste).

De nombreuses possibilités sont offertes pour accéder ainsi à la mémoire de façon indirecte. Le schéma suivant reprend tous les modes d'adressage mémoire indirect possibles pour le processeur 80386. Sauf pour le *segment*, l'ordre des termes est quelconque. De plus, il est permis de ne faire apparaître que certains d'entre-eux dans l'expression pour le calcul d'adresse.

$$\begin{array}{cccccc}
 \left(\begin{array}{c} \text{CS} \\ \text{SS} \\ \text{DS} \\ \text{ES} \\ \text{FS} \\ \text{GS} \end{array} \right) & : & \left(\begin{array}{c} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right) & + & \left(\begin{array}{c} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right) & * & \left(\begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right) & \pm & \left(\begin{array}{c} 8 \text{ bits} \\ 16 \text{ bits} \\ 32 \text{ bits} \end{array} \right) \\
 \text{segment} & & \text{base} & & \text{index} & & \text{facteur} & & \text{déplacement} \\
 & & & & & & \text{d'échelle} & & \text{codé sur}
 \end{array}$$

L'adresse à laquelle débute une zone en mémoire est généralement appelée la *base*. Cette adresse peut se trouver dans n'importe lequel des 4 registres généraux ou des 4 registres d'indexage.

Si la zone en question est structurée en sous-zones, comme c'est par exemple le cas pour un vecteur, il est possible d'accéder à une de ces dernières en précisant son indice dans un des registres d'*index* ci-dessus. (Remarquons que **ESP** ne peut pas être utilisé à cet effet.) De plus, il est possible de multiplier cet index par un des *facteurs d'échelle* indiqués afin de tenir compte de la taille (en bytes) des sous-zones considérées.

Un déplacement signé sur 8, 16 ou 32 bits peut également être ajouté dans le calcul de l'adresse.

Par défaut, lorsque le registre de base est soit **ESP** soit **EBP**, l'adresse à calculer fait référence à la pile¹ associée au programme courant, le registre de segment **SS** interviendra donc dans le calcul de cette adresse. Autrement, c'est le registre **DS** qui est utilisé². Il est possible de changer ce comportement standard en précisant lequel des registres de segment doit être utilisé dans le calcul d'adresse. Voici un exemple :

MOV AX, [DS:EBP]

¹Nous verrons dans un chapitre ultérieur comment fonctionne le stack système.

²Ceci n'est pas toujours le cas pour les instructions de manipulation de chaînes de caractères ; celles-ci ne seront cependant pas étudiées dans ce TP.

Dans l'instruction

```
MOV AX, [EDX*4+EBP]
```

le registre de base est EBP, puisque seul l'index peut être accompagné du facteur d'échelle. Par contre, dans l'instruction

```
MOV AX, [EDX+EBP]
```

n'importe lequel des deux pourrait être la base. Le registre de segment à utiliser dépendrait alors du choix ayant été fait. Afin de lever cette ambiguïté, NASM utilise les règles suivantes :

1. si aucun des registres présents n'emploie le facteur d'échelle, le premier registre dans l'expression est considéré comme étant la base ;
2. si un des registres présents utilise le facteur d'échelle, c'est l'autre registre (s'il existe) qui constitue la base ;
3. si un seul registre est présent dans l'expression, le registre de segment employé lors du calcul d'adresse est celui associé par défaut au registre apparaissant dans cette expression.

Dans l'exemple précédent, DS aurait donc été choisi comme registre de segment à employer.

Bien que cette utilisation des registres de segment soit toujours respectée par le processeur 80386, il convient de remarquer que le système d'exploitation GNU/Linux place, pour chaque application, le code, les données et la pile dans un même segment³. (Les valeurs des registres CS, DS, ES et SS sont donc toutes les mêmes.) En conséquence, les expressions [EDX+EBP] et [EBP+EDX] ci-dessus feront toutes deux référence à la même zone en mémoire.

Pour conclure, remarquons que NASM impose que, lors d'un adressage mémoire, l'entièreté de l'expression pour le calcul d'adresse soit contenue dans des crochets. La notation

```
MOV AX, v[SI]
```

permise dans d'autres assembleurs, doit donc ici être écrite comme suit :

```
MOV AX, [v+SI]
```

³La protection de la mémoire est assurée au sein d'une même application par le système de pagination.

Chapitre 6

Les instructions de manipulation de bits

6.1 L'arithmétique booléenne

L'assembleur 80386 offre les instructions d'arithmétique booléenne que sont AND, OR, XOR et NOT. Les trois premières suivent la même syntaxe que le MOV et réalisent, respectivement, un *et*, un *ou* et un *ou exclusif* bit à bit entre ses deux opérandes. Le résultat est, dans les trois cas, placé dans l'opérande destination.

L'instruction NOT, par contre, est une opération unaire qui n'accepte comme opérande qu'un registre ou une zone en mémoire. Elle fait une complémentation à 1 bit à bit de cet opérande.

Pour rappel, voici les tables de vérité pour ces 4 opérateurs :

AND	0	1	OR	0	1	XOR	0	1	NOT	
0	0	0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	1	1	0	1	0

6.2 Les masques

En assembleur, ces opérateurs de l'arithmétique booléenne sont souvent utilisés pour réaliser du masquage d'information. Le masquage (ou utilisation de masques) est très similaire à la peinture avec un pochoir.

6.2.1 Peinture au pochoir sur une surface propre

La peinture avec pochoir vise à recouvrir de peinture la zone non protégée par le pochoir et à laisser intacte la surface protégée. Si nous considérons que la surface est propre, l'application de la peinture sur la partie non protégée va faire apparaître par contraste le logo. De la même manière, le masquage assembleur

visé à forcer à 1 certains bits dans un mot. Les autres bits (la partie sous le pochoir), sont laissés intacts.

L'utilisation d'un pochoir est décomposée en deux étapes. La première est la réalisation du pochoir, la seconde est son utilisation proprement dite. En assembleur, il n'y a pas de raison que cela change.

Lors de la réalisation du pochoir (création du masque), il faut délimiter la zone qui va être recouverte de peinture du reste. En assembleur, c'est la même chose. Si nous voulons forcer à 1 les bits 4, 5 et 6 d'un mot, nous devons construire un masque qui contiendra des 1 aux emplacements 4, 5 et 6 et qui contiendra des 0 ailleurs.

Voici un tel masque sur un byte de données. La première ligne représente les numéros (ou poids) des bits, la seconde contient le masque.

7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0

Une fois le masque (pochoir) construit, il reste à appliquer la peinture. C'est à ce moment qu'interviennent les opérations booléennes. Si nous voulons forcer à 1 les bits 4, 5 et 6, nous utiliserons l'instruction **OR** entre le mot dont nous voulons forcer les bits (supposons qu'il soit stocké dans **AL**) et le masque construit à cet effet : **OR AL, 01110000b**.

Une propriété intéressante des pochoirs (et des masques par la même occasion) est que si nous désirons l'image négative, il suffit de prendre l'autre partie du pochoir. Idem avec les masques : il suffit de prendre le **NOT** du masque.

Le masque suivant (construit en faisant un **NOT** du masque précédent) va laisser intacts les bits 4, 5 et 6 et forcer à 1 les autres.

7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	1

L'utilisation du masque est identique : nous faisons un **OR** entre le mot et le masque.

6.2.2 Peinture au pochoir sur une surface très sale

Dans le cas où nous essayons de faire de la peinture au pochoir sur une surface très sale dont les taches sont de la même couleur que la peinture que nous utilisons, cela marche moins bien : le contraste sera alors faible et les formes du logo n'apparaîtront plus très clairement. Qu'à cela ne tienne, il suffit alors d'utiliser un pochoir avec une peinture qui est en fait du détachant de sorte à faire disparaître la crasse qui est dans la partie exposée du pochoir. Si nous traduisons cela en assembleur, cela revient à forcer à 0 (faire disparaître) certains bits et à conserver les autres (les laisser intacts).

Nous construisons donc un masque qui va contenir des 0 là où il faut forcer les bits à 0 et des 1 ailleurs.

Par exemple, le masque suivant va permettre de forcer à 0 les bits 4, 5 et 6 d'un byte de données :

7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	1

Une fois le masque ainsi obtenu, nous utilisons un **AND** entre le mot à traiter (supposons à nouveau qu'il soit stocké dans **AL**) et le masque : **AND AL, 10001111b**. Le résultat est bien que les bits 4, 5 et 6 sont forcés à 0, tandis que les autres restent inchangés.

En utilisant le même masque transformé par un **NOT**, nous pouvons laisser les bits 4, 5 et 6 intacts et forcer à 0 les autres.

Et voilà comment on fait de l'art en informatique ...

6.3 Les glissements et les rotations

Les processeurs de la même famille que le 80386 disposent d'une grande variété d'instructions permettant de manipuler le contenu d'un mot. Parmi ces manipulations, il y a deux catégories qui sont extrêmement utiles : les glissements (*shifts* en anglais) et les rotations.

6.3.1 Les glissements

Un glissement de k positions d'un mot de n bits est une opération visant à décaler de k positions vers la gauche ou vers la droite tous les bits de ce mot.

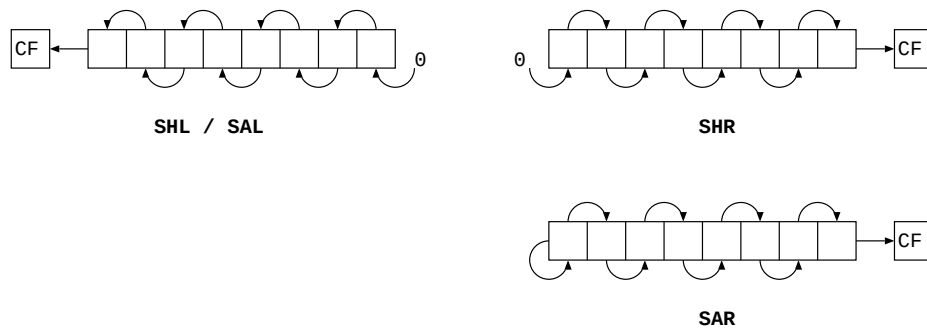
Si nous imaginons une pièce avec n chaises en ligne et certaines des chaises occupées par des personnes assises, lorsque nous faisons un glissement de 1 position *vers la droite*, chaque personne va quitter sa chaise pour occuper la chaise qui se trouve à droite. Évidemment, quelqu'un assis sur la chaise la plus à droite ne pourra plus s'asseoir. Cette personne quitte donc la pièce et va s'installer dans une chaise (unique) qui se trouve dans le couloir. Le 80386 dispose également de cette chaise unique. Elle s'appelle le *carry flag* (CF) et est en fait un des bits du registre **EFLAGS**. Lors d'un tel glissement, le bit le plus à droite (le bit le moins significatif) sera alors placé dans le *carry flag*.

Notons que dans le cas d'un glissement vers la droite, la chaise se trouvant le plus à gauche sera toujours vide, ce qui, en termes d'assembleur, veut dire que le bit le plus à gauche (le bit le plus significatif) sera forcé à 0.

Le glissement de 1 position vers la gauche va envoyer le bit le plus à gauche (le bit le plus significatif) dans le *carry flag*. En conséquence, le bit le plus à droite (le bit le moins significatif) sera forcé à 0.

Les figures suivantes reprennent les différentes instructions de glissement disponibles. La flèche signifie la copie d'un bit. Pour les glissements vers la droite, les figures se lisent de la droite vers la gauche (on commence par copier le bit

le moins significatif dans le *carry flag*). Pour les glissements vers la gauche, le diagramme se lit de la gauche vers la droite. Nous supposons des glissements de 1 bit.



Il existe deux types de glissements vers la droite : le shift « normal » (**SHR**) et le shift arithmétique (**SAR**). Ce dernier laisse le bit de signe (qui, comme nous le verrons, est le bit le plus significatif) intact, alors que le premier l'écrase avec un 0. Il existe également deux instructions de shift vers la gauche (**SHL** et **SAL**), mais elles se comportent toutes deux de la même manière.

Un glissement de k positions est obtenu en appliquant k fois un glissement de 1 position, le *carry flag* étant remplacé à chaque fois.

Au niveau de l'assembleur, à partir du processeur 80186 la syntaxe de ces instructions est la suivante :

```
SHL/SAL/SHR/SAR dst, 1
SHL/SAL/SHR/SAR dst, amp
```

où **dst** est un registre ou une zone en mémoire de 8, 16 ou 32 bits. Le premier format permet de faire des glissements d'une position. Dans le second, **amp** représente l'amplitude du glissement ; elle peut être soit contenue dans le registre **CL** soit représentée par un immédiat sur 8 bits.

Nous distinguons deux syntaxes distinctes pour ces instructions, même si la première peut être vue comme un cas particulier de la seconde. Ce choix est dû au fait que la première nécessitait moins de cycles d'horloge dans les processeurs antérieurs au 80386. Ceci n'est plus le cas pour ce processeur, cependant des codes opératoires différents ont été maintenus pour des raisons de compatibilité.

Interprétation mathématique

Il est important de comprendre que lorsque nous glissons une donnée d'une position vers la gauche, nous multiplions sa valeur par deux. Symétriquement, lorsque nous réalisons un glissement d'une position vers la droite sur une donnée, nous effectuons une division entière par deux de la valeur de cette donnée. De plus, si nous utilisons l'instruction **SAR**, la valeur du signe est conservée dans le résultat.

6.3.2 Les rotations

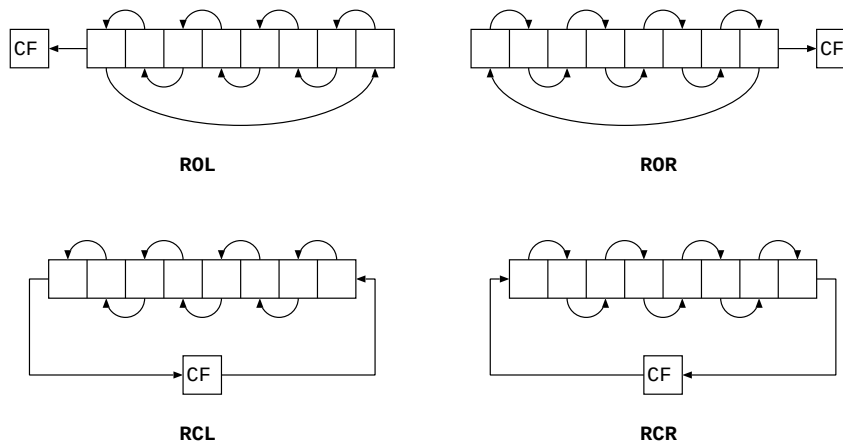
Les rotations sont identiques aux glissements à un détail près : les bits qui quittent le mot d'un côté sont réinjectés de l'autre côté du mot. Il existe deux variantes de rotation (sans considérer les variantes dues au sens de rotation) : celles qui utilisent le *carry flag* et celles qui ne l'utilisent pas.

Reprenons l'analogie de la pièce avec les chaises. Nous désirons effectuer une rotation vers la droite d'une position sans *carry flag*, c'est-à-dire, sans utiliser la chaise dans le couloir. Nous allons procéder comme pour le glissement d'une position vers la droite : chaque personne assise sur une chaise va se déplacer d'une chaise vers la droite. Cependant, et c'est ici que cela change, la personne devant quitter la pièce (celle qui était assise sur la chaise la plus à droite) va aller occuper la chaise la plus à gauche. (Pour rappel, après un glissement vers la droite, cette chaise est toujours libre.)

Si nous utilisons le *carry flag*, la personne la plus à droite va aller s'asseoir dans la chaise du couloir (le *carry flag*). Il se peut que la chaise du couloir soit déjà occupée (le *carry flag* vaudrait 1). Dans ce cas, son occupant va au préalable rentrer dans la pièce et occuper la chaise la plus à gauche (le *carry flag* est donc copié dans le bit le plus à gauche). Somme toute, cela revient à considérer une rotation sur $n + 1$ chaises. Contrairement aux glissements, nous pouvons voir que l'état du *carry flag* avant l'opération est important et influence le résultat final.

Les instructions **CLC** (*clear carry*) et **STC** (*set carry*) permettent, respectivement, de forcer le *carry flag* à 0 et à 1. Elles n'acceptent pas d'opérandes.

La rotation vers la gauche fonctionne de la même manière. Les schémas suivants décrivent les différentes instructions de rotation supportées par la famille des processeurs 80x86. Lorsque la seconde lettre du mnémonique d'une de ces instructions est un **C**, la rotation utilise le *carry flag*. À nouveau, la flèche signifie la copie d'un bit. Cependant, le bit qui va « faire le tour » est mémorisé avant d'être écrasé.



De même que pour les glissements, une rotation de k positions est obtenue en

effectuant k fois la rotation d'une position correspondante.

La syntaxe des instructions de rotation est identique à celle des instructions de glissement, à savoir :

```
ROL/RCL/ROR/RCR dst, 1
ROL/RCL/ROR/RCR dst, amp
```

Pour les mêmes raisons que dans le cas des glissements, nous faisons la distinction entre ces deux syntaxes, bien qu'elles soient maintenant équivalentes en termes de performances.

Il n'y a pas d'interprétation mathématique immédiate pour les rotations ; par contre, elles peuvent être très précieuses pour la création des masques.

6.4 Les enregistrements

Nous avons vu au début de ce chapitre comment isoler ou forcer à 1 des zones de bits. En utilisant le masquage combiné aux glissements et/ou aux rotations, nous allons être capables de déplacer des zones de bits au sein d'un mot. En fonction de la complexité de l'opération, il sera parfois possible de réaliser la manipulation en place. Dans d'autres cas, nous devrons passer par une zone de stockage temporaire.

De telles manipulations sont utiles car elles permettent de considérer un mot comme étant un enregistrement. Les champs de cet enregistrement peuvent donc être définis en indiquant les bits de début et de fin pour chaque champ.

Cette représentation est très compacte, ce qui est souvent le but recherché en assembleur. Dans d'autres cas, elle peut nous être imposée par le matériel ou le système d'exploitation utilisés.

6.5 Les instructions spécialisées

Le processeur 80386 a introduit plusieurs familles d'instructions à usage spécialisé. Nous les présentons, dans ce qui suit, à titre informatif.

6.5.1 Les instructions de recherche de bits

Les instructions de recherche de bits renvoient dans l'opérande destination l'indice du *premier* bit à 1 trouvé dans l'opérande source. Si un tel bit existe, le *zero flag* (ZF) est mis à 0 ; autrement, lorsque la valeur de la source est nulle, il est mis à 1 et la valeur de l'opérande destination devient indéfinie.

L'instruction **BSF** (*bit-scan forward*) réalise cette recherche du bit le moins significatif au bit le plus significatif. À l'opposé, **BSR** (*bit-scan reverse*) la fait dans le sens inverse. Leur syntaxe est la suivante :

```
BSF/BSR dst, src
```

où **dst** est un registre de 16 ou 32 bits et **src** un registre ou une zone en mémoire de 16 ou 32 bits. Comme pour le **MOV**, les deux opérandes doivent toujours avoir la même longueur.

Notons que ces deux instructions sont extrêmement lentes.

6.5.2 Les instructions de test de bits

Il est possible de récupérer dans le *carry flag* la valeur d'un bit particulier dans une zone de données grâce à l'instruction **BT** (*bit-test*). Sa syntaxe est la suivante :

BT src, ind

où **src** représente la zone dans laquelle la recherche doit être faite (un registre ou une zone en mémoire de 16 ou 32 bits) et où **ind** est l'indice du bit à placer dans le *carry flag* ; il peut être soit un registre de 16 ou 32 bits, selon la longueur du premier opérande, soit un immédiat sur 8 bits. Comme d'habitude, la valeur de l'indice du bit le moins significatif de **src** est 0.

Les instructions **BTC** (*bit-test and complement*), **BTR** (*bit-test and reset*) et **BTS** (*bit-test and set*), se comportent de la même manière que **BT**, mais réalisent ensuite, respectivement, un complément, une mise à 0 et une mise à 1 du bit en question. Leur syntaxe reste la même que celle de l'instruction **BT**.

Lorsque le premier opérande est une zone en mémoire et le second un registre, ces quatre instructions ont un comportement particulièrement intéressant : elles vont chercher le bit donné par le second opérande dans une zone en mémoire dont l'adresse de base est donnée par le premier opérande. Par exemple, si **EAX** vaut 100, l'instruction

BT [EBX], EAX

est équivalente à

BT [EBX+12], 4

Dans tous les autres cas, la valeur du second opérande est interprétée modulo 16 ou 32, selon la longueur du premier opérande.

6.5.3 Les glissements en double-précision

Les instructions de glissement en double-précision permettent d'effectuer un glissement d'une certaine amplitude **amp** sur leur premier opérande. Les bits qui seront injectés dans ce dernier seront les **amp** bits de poids fort du deuxième opérande dans le cas d'un glissement vers la gauche ou les **amp** bits de poids faible de ce même opérande lors d'un glissement vers la droite. Contrairement à ce que nous pourrions penser, ce deuxième opérande n'est pas modifié. La syntaxe de ces instructions est la suivante :

SHLD/SHRD dst, src, amp

où **dst** est soit un registre soit une zone en mémoire de 16 ou 32 bits. **src** ne peut être qu'un registre de même longueur que le premier opérande. L'amplitude du

glissement, contenue dans l'opérande `amp`, peut être donnée par un immédiat sur 8 bits ou par le registre `CL`.

Le *carry flag* (`CF`) est mis à jour de la même façon que pour les glissements simples.

Supposons que `EAX` et `EBX` contiennent respectivement les valeurs `01234567h` et `89ABCDEFh`. Après l'instruction `SHLD EAX, EBX, 8`, le registre `EAX` aura la valeur `23456789h` et le `CF` sera mis à 1. Similairement, après `SHRD EAX, EBX, 8`, le registre `EAX` contiendra la valeur `EF012345h` et le `CF` sera mis à 0. Dans les deux cas, `EBX` n'est pas modifié.

Chapitre 7

Les instructions arithmétiques

7.1 Codage des nombres

L'arithmétique définit la notion de nombres positifs et négatifs. Il doit donc être possible de manipuler des nombres signés au niveau du processeur 80386. La première question qui se pose est la représentation qui va être choisie pour exprimer le signe d'un nombre. Remarquons que, comme les nombres signés seront également une séquence de bits, rien, sinon l'interprétation qu'on en fera, ne permettra de les distinguer des nombres non signés. Les trois solutions les plus courantes pour les représenter sont les suivantes.

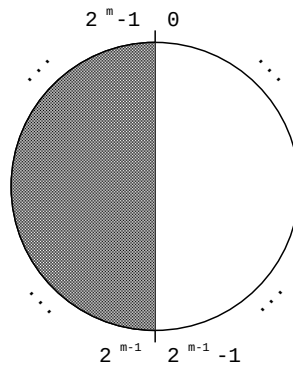
7.1.1 Notation avec bit de signe

Une première solution consiste à indiquer la valeur du signe au moyen d'un bit. Généralement, le bit de poids fort d'un mot est dédié à la représentation du signe. S'il vaut un, le nombre dont la valeur absolue est indiquée par les bits restants est négatif. S'il vaut zéro, ce nombre est positif. À titre d'exemple, si nous considérons des mots de 8 bits avec le bit le plus significatif comme bit de signe, nous obtenons :

$$01011110 = +94$$

$$11011110 = -94$$

Supposons que l'ensemble des 2^m valeurs exprimables avec un mot de m bits soient placées sur un cadran d'horloge :



Nous pouvons observer l'organisation suivante :

- les nombres positifs utilisent la moitié droite du cadran, les négatifs la moitié gauche ;
- lorsqu'un nombre positif se trouve n positions après le zéro, son négatif est également n positions *après* la valeur correspondant au mot ayant 1 comme bit le plus significatif et 0 ailleurs (2^{m-1}) ;
- le nombre constitué de m zéros représente la valeur 0, et la valeur centrale du cadran (celle ayant le bit le plus significatif à 1 et les autres bits à 0) représente, selon les conventions ci-dessus, un *zéro négatif* qui peut donc être utilisé pour représenter *Not a Number* (NaN) ;
- les $2^{m-1} - 1$ autres valeurs commençant par un 0 sont attribuées à des nombres strictement positifs, les $2^{m-1} - 1$ valeurs restantes le sont à des nombres strictement négatifs.

7.1.2 Notation en complément à un

En notation en complément à un, l'opposé d'un nombre est obtenu en soustrayant ce nombre de $2^m - 1$. Cette opération peut être très facilement réalisée en inversant tous les bits du nombre considéré. En travaillant à nouveau avec des mots de m bits et en utilisant la même représentation sur un cadran d'horloge, nous observons que :

- les nombres positifs utilisent à nouveau la moitié droite du cadran, les nombres négatifs le reste ;
- lorsqu'un nombre positif se trouve n positions après le zéro, son négatif se trouve $n + 1$ positions *avant* le zéro ;
- le nombre constitué de m zéros représente la valeur 0, et celui ne contenant que des 1, correspondant à l'opposé de 0, peut à nouveau être utilisé pour coder NaN ;
- les $2^{m-1} - 1$ autres valeurs commençant par un 0 sont attribuées à des nombres strictement positifs, les $2^{m-1} - 1$ valeurs restantes le sont à des nombres strictement négatifs.

Nous pouvons de plus observer que le bit le plus significatif indique, ici aussi, le signe du nombre représenté.

7.1.3 Notation en complément à deux

En notation en complément à deux, l'opposé d'un nombre est obtenu en soustrayant ce nombre de 2^m . Cette opération peut être facilement réalisée en deux étapes : tout d'abord passer en complément à un (c'est-à-dire, inverser tous les bits) et ensuite ajouter 1. Toujours en utilisant la représentation sur un cadran d'horloge, nous voyons que :

- les nombres positifs occupent la moitié droite du cadran, les nombres négatifs le reste ;
- lorsqu'un nombre positif se trouve n positions après le zéro, son négatif se trouve n positions *avant* le zéro (ce résultat n'est pas surprenant vu le mode de construction de la représentation en complément à deux) ;
- le nombre constitué de m zéros représente la valeur 0, et il n'y a cette fois-ci aucune valeur invalide pouvant servir à représenter NaN ;
- les $2^{m-1} - 1$ autres valeurs commençant par un 0 sont attribuées à des nombres strictement positifs, les 2^{m-1} valeurs restantes le sont à des nombres strictement négatifs.

Comme pour la notation en complément à un, nous pouvons une fois de plus observer que le bit le plus significatif indique le signe du nombre représenté.

Le 8086 et ses successeurs ne *comprennent* que la notation en complément à deux (du moins d'un point de vue matériel). Vu le foisonnement des différentes notations, nous pouvons nous demander ce qui justifie un tel choix, d'autant plus que la notation en complément à deux n'est pas spécialement la plus lisible ou la plus aisée à comprendre. La raison est simple : c'est la seule qui permet de conserver les mêmes circuits pour l'arithmétique sur nombres signés et non signés. Cet argument de choc, qui évite la duplication de tous les circuits dans l'UAL, est prépondérant.

7.2 Les instructions arithmétiques

Le processeur 80386 fournit un ensemble d'instructions permettant de réaliser les opérations arithmétiques courantes que sont l'addition, la soustraction, la multiplication et la division.

7.2.1 L'addition

L'instruction ADD permet d'additionner deux opérandes. Son format est à rapprocher de celui du MOV :

`ADD dst, src`

où **dst** est un registre ou une zone en mémoire de 8, 16 ou 32 bits et **src** est un registre, une zone en mémoire ou un immédiat de même taille. Les opérandes **dst** et **src** ne peuvent pas être deux zones en mémoire simultanément. Le résultat de l'addition de **dst** avec **src** est placé dans **dst**.

Lors d'une telle opération, il se peut que le résultat dépasse ce qui est représentable sur 8, 16 ou 32 bits. Les valeurs du *carry flag* (CF) et de l'*overflow flag* (OF) sont donc mises à jour par le processeur afin de détecter les différents cas qui peuvent se produire. Le *carry flag* sera mis à 1 si le résultat obtenu en considérant les opérandes comme des valeurs non signées est incorrect. De même, l'*overflow flag* vaudra 1 si le résultat obtenu en considérant des opérandes à valeurs signées est incorrect.

Le tableau suivant reprend les différents cas de dépassements pouvant se présenter lors d'une addition. Nous supposons des opérandes de 16 bits. Nous considérons à chaque fois les cas signé et non signé.

FFFFh	65535	-1
0002h	2	2
0001h	1	1
CF=1 OF=0	FAUX	JUSTE

7FFEh	32766	32766
0002h	2	2
8000h	32768	-32768
CF=0 OF=1	JUSTE	FAUX

8000h	32768	-32768
8000h	32768	-32768
0000h	0	0
CF=1 OF=1	FAUX	FAUX

Dans certains cas, comme celui de la multiprécision, il est nécessaire d'effectuer une addition en prenant en compte la valeur du *carry flag*, c'est-à-dire en ajoutant au résultat de l'addition la valeur qu'avait le *carry flag* avant cette addition. Ceci peut être réalisé à l'aide de l'instruction **ADC** (*add with carry*) ayant la même syntaxe que le **ADD**. Son effet est le suivant : $\text{dst} \leftarrow \text{dst} + \text{src} + \text{CF}$.

L'incrémentation

L'instruction **INC** est un cas particulier d'addition, dans lequel la valeur du seul opérande est augmentée de 1. Cette instruction est plus rapide que son équivalent en **ADD** et *ne* modifie *pas* CF. Son opérande peut être soit un registre soit une zone en mémoire de 8, 16 ou 32 bits.

7.2.2 La soustraction

Une soustraction peut être réalisée à l'aide de l'instruction **SUB**. Celle-ci a la même syntaxe que le **ADD**. Au même titre qu'il est possible de faire une addition

avec report (en utilisant la valeur du *carry flag*), il est possible de faire une soustraction avec emprunt. L'instruction permettant cela s'appelle SBB (*subtract with borrow*). Son effet est le suivant : $\text{dst} \leftarrow \text{dst} - \text{src} - \text{CF}$.

Comme pour l'addition, le *carry flag* sera mis à 1 si la soustraction se termine avec un emprunt qui ne peut pas être traité. L'*overflow flag* sera à son tour mis à 1 si le signe du résultat de la soustraction avec des opérandes signés est incohérent.

La décrémentation

Au même titre que INC, l'instruction DEC est un cas particulier de soustraction, dans lequel la valeur du seul opérande est diminuée de 1. Cette instruction jouit de la même syntaxe et des mêmes propriétés que son correspondant INC.

7.2.3 La multiplication

L'instruction MUL permet de multiplier deux nombres non signés. Sa syntaxe est la suivante :

MUL *src*

où *src* peut être soit un registre soit une zone en mémoire de 8, 16 ou 32 bits.

La sémantique de cette instruction diffère suivant la taille de l'opérande. Dans le cas où *src* est sur 8 bits, nous aurons : $\text{AX} \leftarrow \text{AL} * \text{src}$. Le résultat de la multiplication de deux opérandes sur 8 bits est donc toujours stocké sur 16 bits. A titre d'exemple, MUL CL va donc placer dans AX le résultat de la multiplication de AL et CL.

Dans le cas d'un opérande sur 16 bits, nous aurons : $\text{DX:AX} \leftarrow \text{AX} * \text{src}$. C'est-à-dire que AX contiendra la partie basse du résultat et DX la partie haute. Nous avons ici que le résultat de la multiplication de deux opérandes sur 16 bits est stocké sur 32 bits. Deux registres de 16 bits sont utilisés pour le résultat, au lieu d'un seul de 32 bits, afin de rester compatible avec les processeurs précédents de la même famille.

Finalement, lorsque *src* est sur 32 bits, MUL aura le comportement suivant : $\text{EDX:EAX} \leftarrow \text{EAX} * \text{src}$. Le résultat de la multiplication de deux opérandes sur 32 bits est constitué de 64 bits. Le processeur 80386 ne disposant pas de registres de telle taille, deux registres de 32 bits sont utilisés pour stocker le résultat.

Cette instruction met les flags CF et OF à 1 lorsque le résultat prend plus de bits que l'opérande *src* pour être représenté. Autrement, ces deux flags sont mis à 0.

Nous avons vu précédemment que la représentation en complément à deux permettait de conserver les mêmes circuits (et les mêmes instructions) pour l'addition et la soustraction de nombres signés et non signés. Cela n'est malheureusement pas le cas pour la multiplication. Une instruction spécialisée,

IMUL (*integer multiply*), existe donc pour multiplier deux opérandes signés. Sa syntaxe et son fonctionnement sont identiques à ceux de **MUL**.

À partir du processeur 80386, des versions à deux et à trois opérandes de **IMUL** sont apparues.

La version à deux opérandes :

IMUL dst, src

place dans le premier des deux le résultat de la multiplication de **dst** par **src**. **dst** est soit un registre de 16 bits soit un registre de 32 bits. Dans le premier cas, **src** peut être un registre ou une zone en mémoire de 16 bits, ou encore un immédiat sur 8 ou 16 bits. Dans le second, **src** doit être un registre ou une zone en mémoire de 32 bits, ou encore un immédiat sur 8 ou 32 bits (et pas 16).

Dans la version à trois opérandes :

IMUL dst, src1, src2

le résultat de la multiplication du deuxième et du troisième opérande est placé dans le premier. Comme pour la version à deux opérandes, **dst** ne peut être qu'un registre de 16 ou de 32 bits. Dans le premier cas, **src1** est soit un registre soit une zone en mémoire de 16 bits et **src2** un immédiat de 8 ou 16 bits. Dans le second, **src1** est soit un registre soit une zone en mémoire de 32 bits et **src2** un immédiat de 8 ou 32 bits.

Dans ces deux nouvelles versions de **IMUL** le résultat est stocké dans un registre ayant la même taille qu'un des opérandes intervenant dans la multiplication. Ceci peut entraîner une perte d'information dans le résultat, auquel cas, les flags **CF** et **OF** seront mis à 1.

Contrairement aux instructions **MUL** et **IMUL** à un opérande, les versions à deux et trois opérandes de **IMUL** fournissent le résultat escompté aussi bien dans le cas signé et dans le cas non signé, à condition qu'il n'y ait pas dépassement, c'est-à-dire que, si n est le nombre de bits de la représentation, le résultat tienne sur n bits dans le cas non signé, et la valeur absolue du résultat sur $n - 1$ bits dans le cas signé.

Vérifions ceci en supposant que **IMUL** donne, sous ces hypothèses, le bon résultat pour la multiplication de deux nombres non signés. Supposons que a et b soient deux entiers signés, tels que le produit de leur valeur absolue ne provoque pas de dépassement, c'est-à-dire qu'il fasse au maximum $n - 1$ bits. Trois cas sont possibles.

- Si a et b sont positifs, la situation est exactement la même que lors d'une multiplication non signée.
- Si l'un des deux nombres est négatif (disons a) et l'autre est positif, a , interprété comme un nombre non signé, est codé comme $2^n - |a|$. La multiplication de a par b donne donc :

$$(2^n - |a|) \cdot b = 2^n \cdot b - |a| \cdot b,$$

ce qui, modulo 2^n , est égal à $2^n - |a| \cdot b$, ou encore à $2^n - |a \cdot b|$, puisque b est positif.

- Si a et b sont négatifs, comme dans le deuxième cas, nous pouvons remplacer a et b par leur représentation. On a alors :

$$(2^n - |a|) \cdot (2^n - |b|) = 2^n \cdot (2^n - |a| - |b|) + |a| \cdot |b| = 2^n \cdot (2^n - |a| - |b|) + |a \cdot b|,$$

ce qui, modulo 2^n , est égal à $|a \cdot b|$, ou encore à $a \cdot b$, puisque a et b sont deux nombres négatifs.

Remarque

Même si, sur le 80386, une multiplication prend sensiblement moins de temps que sur les processeurs antérieurs, cette instruction demeure extrêmement lente. Elle ne doit donc être utilisée que lorsqu'aucune alternative n'est possible.

Par exemple, nous avons vu au chapitre 6 qu'une multiplication par une puissance de 2 pouvait être très efficacement réalisée à l'aide d'un glissement vers la gauche. De même, lorsque nous avons étudié les modes d'adressage, nous avons vu que le processeur était capable de réaliser certaines opérations arithmétiques lors d'un calcul d'adresse. L'instruction **LEA** (*load effective address*), permet d'exploiter ceci afin de réaliser des multiplications un peu plus complexes. Cette instruction, dont la syntaxe est la suivante :

LEA res, adr

où **res** est un registre de 16 bits en mode 16 bits ou de 32 bits en mode 32 bits, charge dans son premier opérande l'adresse résultant du calcul de l'expression contenue dans **adr**.

Le deux instructions suivantes multiplient donc le contenu de **EAX** par 10 en 5 cycles d'horloge, 2 pour la première et 3 pour la seconde.

```
LEA EAX, [EAX+EAX*4]
SHL EAX, 1
```

Le **MUL** correspondant aurait eu besoin d'entre 9 et 38 cycles d'horloge !

7.2.4 La division

DIV et **IDIV** permettent de diviser deux nombres, respectivement non signés et signés. Voici leur syntaxe :

DIV/IDIV src

où le seul opérande peut être soit un registre soit une zone en mémoire de 8, 16 ou 32 bits.

Comme pour l'instruction **MUL**, leur fonctionnement change selon la taille des opérandes. Dans le cas où l'opérande, c'est-à-dire le diviseur (ce par quoi nous divisons), tient sur 8 bits, la sémantique est $AL \leftarrow AX \text{ div } src$ (le résultat de la division entière) et $AH \leftarrow AX \text{ mod } src$ (le reste).

Lorsque l'opérande est sur 16 bits, le résultat de la division entière de **DX:AX** avec **src** est placé dans **AX** et le reste cette division dans **DX**.

Finalement, si `src` est sur 32 bits, $EAX \leftarrow EDX:EAX \text{ div } src$ et $EDX \leftarrow EDX:EAX \text{ mod } src$.

En procédant ainsi il est possible que le registre alloué pour placer le quotient de la division ait une taille trop petite. Par exemple, si nous divisons 20000 par 2 en utilisant la version 8 bits de `DIV`, la valeur du quotient, 10000, sera trop grande pour être placée dans `AL`. Contrairement aux instructions de multiplication, le 80386 ne signale pas ce dépassement en basculant la valeur de certains flags (ces derniers ont une valeur indéfinie après une division), mais il lance par contre une exception de type *divide overflow* (`int 00h`).

7.2.5 Le passage à l’opposé

La complémentation à deux d’une valeur peut être réalisée aisément grâce à l’instruction `NEG`, n’acceptant qu’un seul opérande : un registre ou une zone en mémoire de 8, 16 ou 32 bits.

7.3 Les instructions d’extension de signe

Depuis le processeur 8086, les instructions `CBW` (*convert byte to word*) et `CWD` (*convert word to doubleword*) étendent les valeurs respectivement de `AL` à `AX` et de `AX` à `DX:AX`, en faisant une extension de leur signe sur la moitié haute du registre résultat.

À partir du processeur 80386 sont apparues deux instructions analogues. `CWDE` (*convert word to double (extended)*) convertit le mot dans `AX` en un double mot dans `EAX`, en faisant une extension du signe sur la moitié haute de ce dernier registre. `CDQ` (*convert doubleword to qword*) fait de même avec `EAX` et `EDX:EAX`.

Chapitre 8

Les choix et les boucles

Les comparaisons (ou tests) sont une brique de base des structures algorithmiques telles que les choix et les boucles. Alors que des langages comme le C++ possèdent des instructions pour ce type de structures, l'assembleur n'en dispose pas ; elles doivent être réalisées à l'aide de tests et de branchements conditionnels.

8.1 Les instructions de comparaison

Les comparaisons en assembleur fonctionnent de manière différente de celles de la plupart des langages de haut niveau, comme le C++. En effet, alors que dans ces langages il existe un opérateur pour chaque type de comparaison (plus grand, différent, plus petit ou égal, ...), en assembleur une instruction de comparaison se chargera d'effectuer toutes les comparaisons possibles en un coup. Après son exécution, les flags du processeur seront mis à jour pour refléter le résultat de ces comparaisons. Des instructions de branchement conditionnel pourront dès lors être utilisées suivant la valeur de ces flags afin de réaliser une structure algorithmique évoluée.

Le processeur 80386 offre deux instructions permettant de comparer deux valeurs : `CMP` et `TEST`. Elles partagent la même syntaxe :

`CMP/TEST op1, op2`

où `op1` est un registre ou une zone en mémoire de 8, 16 ou 32 bits et `op2` un registre, une zone en mémoire ou un immédiat de même taille. Comme d'habitude, `op1` et `op2` ne peuvent pas être en même temps deux zones en mémoire.

`CMP` (*compare*) soustrait de `op1` la valeur de `op2` et `TEST` réalise un `AND` entre ses deux opérandes. Ces deux instructions laissent inchangés leurs paramètres, mais modifient par contre la valeur de certains flags.

Après un `CMP`, le *carry flag* (`CF`) et le *overflow flag* (`OF`) indiquent respectivement si un dépassement non signé ou signé est survenu. Après un `TEST`, ces deux flags

sont toujours mis à 0. Dans les deux cas, le *sign flag* fournit le signe du résultat et le *zero flag* indique si ce résultat est nul ou pas.

8.2 Les instructions de saut

Il existe deux types de sauts : les sauts inconditionnels et les sauts conditionnels. Les premiers effectuent le branchement à l'adresse en mémoire fournie, quelle que soit la valeur des flags de comparaison. Les seconds effectuent également un branchement, mais uniquement si la condition qu'ils doivent tester est vérifiée, c'est-à-dire si les flags ont les bonnes valeurs.

L'instruction **JMP** permet de faire un saut inconditionnel. Elle a pour syntaxe :

```
JMP dst
```

où **dst** est une adresse en mémoire.

L'assembleur offre une facilité, appelée *label*, qui permet de placer devant une ligne d'assembleur un signet portant un nom. Par après, ce nom pourra être utilisé pour faire référence à l'adresse à laquelle se trouve la ligne d'instruction correspondante. Ce label peut prendre la place d'une adresse en mémoire explicite dans n'importe quelle instruction, notamment le **JMP**, et permet de ne pas devoir calculer à la main les adresses. Voici un exemple :

```
debut_bcle:    MOV AX, 1515
               ...
               JMP debut_bcle
```

L'adresse de saut peut également être exprimée par un registre (**JMP EBX**, par exemple) ou un opérande en mémoire. Ce dernier cas de figure permet de faire un saut indirect, l'adresse de branchement est elle même contenue dans une case mémoire dont on fournit l'adresse, comme dans l'exemple suivant : **JMP [ESI]**.

Comme dit plus haut, les sauts conditionnels ne font le branchement que si la condition testée par l'instruction est vérifiée. Si elle ne l'est pas, le branchement n'est pas réalisé et l'instruction suivante est exécutée. La table 8.1 reprend les instructions de saut conditionnel existantes ainsi que la condition testée pour chacune de ces instructions. Les mnémoniques appartenant à la première section du tableau concernent l'arithmétique non signée, ceux de la deuxième l'arithmétique signée, ceux des deux autres n'ayant pas de champ d'action spécifique.

La syntaxe de ces instructions de saut conditionnel est :

```
Jxxxx dst
```

où **dst** doit pouvoir se traduire par un déplacement signé à partir de l'instruction courante d'au plus 128 bytes en arrière ou 127 bytes en avant. Le 80386 introduit un deuxième format pour ces instructions n'imposant pas cette limitation. Pour pouvoir l'utiliser, NASM exige d'employer (contre toute intuition) le mot-clé **NEAR** derrière le mnémonique de l'instruction :

```
JE NEAR lbl
```

Mnémonique	Signification	Condition testée
JA/JNBE	above ($>$)	CF = 0 et ZF = 0
JAЕ/JNB	above or equal ($\geq$)	CF = 0
JB/JNAE	below ($<$)	CF = 1
JBE/JNA	below or equal ($\leq$)	CF = 1 ou ZF = 1
JC	carry	CF = 1
JNC	no carry	CF = 0
JG/JNLE	greater ($>$)	SF = 0F et ZF = 0
JGE/JNL	greater or equal ($\geq$)	SF = 0F
JL/JNGE	less ($<$)	SF $\neq$ 0F et ZF = 0
JLE/JNG	less or equal ($\leq$)	SF $\neq$ 0F
JO	overflow	OF = 1
JNO	no overflow	OF = 0
JE/JZ	equal/zero	ZF = 1
JNE/JNZ	not equal/zero	ZF = 0
JP/JPE	parity / parity even	PF = 1
JNP/JPO	no parity / parity odd	PF = 0
JS	sign	SF = 1
JNS	no sign	SF = 0
JCXZ	CX equals zero	CX = 0
JECXZ	ECX equals zero	ECX = 0

Table 8.1 – Les instructions de saut conditionnel

8.3 Les structures alternatives

Une structure alternative, un *if*, pourra donc être réalisée avec une instruction de comparaison (ou avec une autre instruction qui modifie les flags) suivie d'une instruction de saut conditionnel pour accéder à l'autre branche. Voici un exemple illustrant le test $AX \geq BX$:

```

        CMP AX, BX
        JB  sinon

        ; instructions de la branche then
        ; (cas  $AX \geq BX$ )

        JMP fsi

sinon:   ; instructions de la branche else
        ; (cas  $AX < BX$ )

fsi:

```

Le saut inconditionnel à la fin de la branche *then* permet de sauter les instructions de l'autre branche et aller directement à l'instruction suivant le *if*.

8.4 Les structures itératives

Une boucle de type *while* pourra être codée en assembleur à l'aide d'une instruction de comparaison (ou d'une autre instruction modifiant la valeur des flags) suivie d'une instruction de saut conditionnel vers l'instruction suivant la boucle ; la dernière instruction du corps de la boucle sera un saut inconditionnel vers la condition à tester à chaque itération. Voici un exemple :

```

bcle:   CMP AX, BX
        JB  fin_bcle

        ; instructions du corps de la boucle
        ; (tant que  $AX \geq BX$ )

        JMP bcle

fin_bcle:

```

Une boucle de type *repeat-until* (ou *do-while*) peut être construite de façon similaire, en plaçant la comparaison et le saut conditionnel à la fin.

Bien que les boucles de type *for* puissent aussi être réalisées en introduisant un incrément ou un décrétement d'une des valeurs intervenant dans la condition d'une des deux boucles décrites ci-dessus, la famille des processeurs 80x86 dispose d'une instruction spécialisée utilisant comme compteur de boucle le re-

gistre ECX¹ : LOOP. Sa syntaxe est identique à celle des sauts conditionnels, sauf pour le mot-clé NEAR, qui ne peut pas être utilisé ici :

LOOP **dst**

Cette instruction fonctionne de la manière suivante :

1. le registre ECX est décrémenté de 1 ;
2. si sa valeur est différente de 0, un branchement est fait sur l'adresse indiquée par **dst** ;
3. sinon, l'instruction suivant le LOOP est exécutée.

Avec une telle sémantique, ECX contient toujours le nombre d'itérations qu'il reste à faire dans la boucle. Il devra donc être initialisé en conséquence.

Des variantes un peu plus évoluées de LOOP existent. La table 8.2 reprend leur nom et signification.

Mnémonique	Signification	Condition testée
LOOP	ECX not zero	ECX $\neq$ 0
LOOPE/LOOPZ	ECX not zero and equality	ECX $\neq$ 0 et ZF = 1
LOOPNE/LOOPNZ	ECX not zero and not equality	ECX $\neq$ 0 et ZF = 0

Table 8.2 – La famille des instructions LOOP

8.5 Les instructions *set byte on condition*

À partir du 80386 il existe une famille d'instructions permettant de mettre à 1 un byte si une certaine condition est satisfaite, ou de le mettre à 0 si elle ne l'est pas. Ceci est pratique si le résultat d'un test doit être sauvegardé pour une utilisation ultérieure.

Il existe une instruction pour chacune des conditions étudiées ci-dessus. La syntaxe générale de ces instructions, nommées *set byte on condition*, est la suivante :

SETxxx **dst**

où xxx est le nom de la condition et **dst** est un registre ou une zone en mémoire de 8 bits.

À titre d'exemple, l'instruction SETNZ AL initialisera AL à 1 si la valeur du *zero flag* (ZF) vaut 0.

¹Jusqu'au 80386 c'était le registre CX qui était employé. C'est toujours le cas si le processeur travaille en mode 16 bits.

Chapitre 9

Les procédures

Les langages de programmation de haut niveau offrent la possibilité de découper un problème en un ensemble de sous-problèmes plus ou moins indépendants, à l'aide de *procédures*, *fonctions* ou *méthodes*. Nous allons voir dans le présent chapitre comment ces mécanismes peuvent être mis en place en assembleur.

9.1 La pile

Le processeur 80386 est capable de gérer une pile (*stack*) de mots de 16 ou 32 bits. Le segment où celle-ci se trouve est donné par le registre **SS** (*stack segment*) et a la particularité d'être remplie de l'adresse la plus significative à l'adresse la moins significative. L'adresse où se trouve le sommet de la pile est à tout moment donnée par le registre **ESP**, le *stack pointer*.

L'instruction **PUSH** permet d'empiler une donnée sur le stack. Sa syntaxe est la suivante :

PUSH src

où **src** est soit un registre ou une zone en mémoire de 16 ou 32 bits soit un immédiat sur 8, 16 ou 32 bits. Elle décrémente la valeur de **ESP** de 2 ou de 4 selon que l'opérande soit sur 16 ou 32 bits, respectivement, avant de placer le contenu de **src** à l'adresse fournie par la nouvelle valeur de **ESP**. Il est important de remarquer qu'en mode 32 bits, lorsque l'opérande est un immédiat sur 8 bits, le *stack pointer* est décrémenté de 4 unités.

L'opération inverse est réalisée par l'instruction **POP**, dont la syntaxe est la suivante :

POP dst

où **dst** est un registre ou une zone en mémoire sur 16 ou 32 bits. Elle copie les 2 ou 4 premiers bytes accessibles à partir du sommet de la pile avant d'incrémenter la valeur de **ESP** de 2 ou 4 unités, respectivement. Bien évidemment, son comportement exact dépend de la longueur de son opérande.

L'instruction **PUSHAD** (*push all double*) permet de sauvegarder sur la pile tous

les registres généraux et d'indilage sur 32 bits. Ces derniers sont empilés dans l'ordre : **EAX**, **ECX**, **EDX**, **EBX**, **ESP** (c'est la valeur qu'il avait avant l'exécution de cette instruction qui est empilée), **EBP**, **ESI**, **EDI**. L'opération inverse peut être réalisée par l'instruction **POPAD** (*pop all double*).

De même, les instructions **PUSHAW** (*push all word*) et **POPAW** (*pop all word*) s'occupent du même travail pour les versions 16 bits de ces mêmes registres.

Cependant, nous utiliserons plus volontiers les instructions **PUSHA** et **POPA**, synonymes de **PUSHAD** et **POPAD** si le processeur travaille en mode 32 bits, mais de **PUSHAW** et **POPAW** si le processeur travaille en mode 16 bits.

Enfin, le registre **EFLAGS** peut également être empilé et dépilé du stack à l'aide des instructions **PUSHF** (*push flags*) et **POPF** (*pop flags*). Les instructions **PUSHFD**, **POPFD**, **PUSHFW** et **POPFW** fonctionnent de manière similaire aux instructions **PUSHA** et **POPA**.

9.2 L'appel à une procédure

Lorsque nous appelons une procédure, nous voulons exécuter une série d'instructions qui ne se trouvent pas à la suite de l'instruction courante. Nous devons donc faire un saut vers l'adresse où se trouve la première instruction à exécuter de la procédure que nous appelons. La différence avec un simple **JMP** réside dans le fait que, à la fin de l'exécution de cette procédure, le programme appelant devra reprendre son exécution là où il l'avait laissée. Nous devons pour ce faire pouvoir mémoriser l'adresse de retour, c'est-à-dire l'adresse de l'instruction suivante dans le programme appelant.

Le processeur 80386 permet de réaliser ceci à l'aide de l'instruction **CALL**. Cette dernière va donc faire un saut vers l'adresse indiquée dans son seul opérande, en empilant d'abord la valeur du registre **EIP**, donnant l'adresse de l'instruction physiquement suivante, sur le stack.

Deux types d'appels peuvent se produire :

- le *call near* ou *intra-segment* a lieu lorsque la procédure à exécuter se trouve à proximité, c'est-à-dire dans le même segment que le programme appelant ; dans ce cas, seul **EIP** est empilé avant l'appel ;
- le *call far* ou *extra-segment* permet d'appeler une procédure résidant dans un autre segment ; les registres **CS** et **EIP** seront donc empilés sur le stack, dans cet ordre, avant l'appel.

Nous n'étudierons que le premier type d'appel au cours des TP.

Le retour au programme appelant se fait par l'instruction **RET**¹. Celle-ci va donc dépiler l'adresse de retour du stack et faire un saut vers l'instruction suivant le **CALL** dans le programme appelant. Bien évidemment, **RET** sera la dernière instruction de toute procédure.

¹Sous NASM, cette instruction est un synonyme de **RETN**, l'instruction de retour d'une procédure de type *near*. L'instruction **RETF** permet de réaliser un retour au programme appelant pour une procédure de type *far*.

9.3 Le passage de paramètres

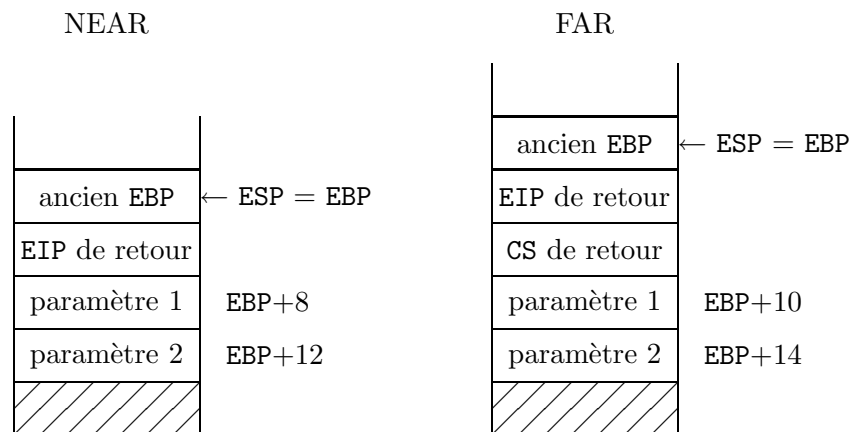
Plusieurs méthodes existent pour passer des paramètres à une procédure. Les plus classiques consistent à les faire passer soit par les registres, soit par la pile.

Un des inconvénients de la première technique est le nombre limité de registres disponibles sur un processeur. Elle est par contre souvent employée dans des interruptions.

Lors de passages de paramètres par le stack, ceux-ci sont empilés avant l'appel à la procédure. Vu que l'adresse de retour est empilée juste après, nous pouvons nous demander comment la procédure accèdera à ses paramètres. La solution est simple. Lorsque nous avons étudié les différents modes d'adressage indirect, nous avons vu que le registre **EBP** faisait, par défaut, référence au segment du stack lorsqu'il était utilisé comme base lors d'un accès à la mémoire. Nous allons donc l'utiliser comme décrit ci-dessous pour accéder aux paramètres d'une procédure.

En premier lieu, nous allons initialiser **EBP** de telle manière qu'il pointe vers le sommet de la pile. Pour cela, nous devrons copier le contenu de **ESP** dans **EBP**. Cependant, comme le programme appelant pourrait lui-même être en train d'utiliser **EBP** comme registre de travail, nous devons donc préalablement sauvegarder sa valeur à l'aide d'un **PUSH**.

La figure suivante montre l'état de la pile après un appel de procédure avec paramètres, tant pour le cas d'un *call near* que pour celui d'un *call far*. Nous voyons que dans le premier cas les paramètres sont accessibles à partir de l'adresse **EBP+8**, tandis que pour le second, ceux-ci se trouvent à partir de **EBP+10**.



Enfin, précisons que la convention **C** pour le passage de paramètres² consiste à les empiler de droite à gauche dans l'ordre dans lequel ils apparaissent dans l'en-tête de la fonction. Par conséquent, le premier paramètre de la fonction sera celui se trouvant le plus près du sommet de la pile. Ceci permet à des fonctions

²Cette convention doit impérativement être respectée lorsqu'une procédure écrite en assembleur interagit avec une fonction écrite en **C** ou en **C++**.

avec un nombre variable de paramètres de retrouver très facilement leur premier paramètre, celui qui précise souvent le nombre exact de paramètres effectifs.

9.4 Les variables locales

Parfois il est nécessaire pour une procédure d'avoir des variables locales. De la place peut être réservée à cet effet sur la pile en décrémentant de façon pertinente la valeur de **ESP**. Ces variables locales pourront ainsi être accessibles par des déplacements négatifs de plus en plus grands à partir de l'adresse **EBP-2**.

9.5 La valeur de retour d'une fonction

Dans le cas des fonctions, un registre est souvent employé pour retourner une valeur au programme appelant. La convention **C** consiste à placer cette valeur, suivant que sa taille soit de 8, 16 ou 32 bits, dans **AL**, **AX** ou **EAX** respectivement.

Bien évidemment, d'autres possibilités peuvent être envisagées.

9.6 Le retour de procédure

Avant de rendre le contrôle au programme appelant, une procédure doit laisser la pile dans le même état qu'au tout début de son exécution. Pour cela, si de la place a été réservée pour des variables locales, la valeur de **ESP** devra être réajustée. Ceci peut être très facilement réalisé à l'aide d'un **MOV** de **EBP** vers **ESP**. Finalement, il faudra restaurer la valeur de **EBP** par un **POP**.

Nous avons vu plus haut que le retour au programme appelant se faisait à l'aide de l'instruction **RET**. Même si la convention **C** exige du programme appelant de dépiler les paramètres après un appel de procédure, cette instruction peut aussi s'en occuper lorsqu'un immédiat (sur 16 bits) est fourni comme opérande. Dans ce cas, autant de bytes que précisé seront retirés du sommet de la pile. Par exemple, un **RET 4** aura pour effet de dépiler 4 bytes. Ceci peut être utile lorsque les programmes appelant et appelé sont tous deux écrits en assembleur.

9.7 Canevas général

En résumé, le schéma d'un appel à une procédure aura la forme suivante :

<pre> programme appelant : ; push des paramètres CALL nomDeProcédure ADD ESP, n : </pre>	<pre> programme appelé nomDeProcédure: PUSH EBP MOV EBP, ESP SUB ESP, m ; sauvegarde des ; registres utilisés : ; restauration des ; registres utilisés (dans ; l'ordre inverse) MOV ESP, EBP POP EBP RET </pre>
--	---

où *n* et *m* correspondent, respectivement, au nombre de bytes nécessaires pour les paramètres et pour les variables locales. Notons de plus que réajuster la valeur de `ESP` après un `CALL` à l'aide d'une addition est plus rapide que de faire les `POP` correspondants.

Comme pour le registre `EBP`, le programme appelant pourrait être en train d'utiliser d'autres registres au moment où il fait un appel de procédure. Afin de ne pas écraser les valeurs de ces registres et de ne pas perturber ainsi l'exécution de ce programme, il convient à la procédure appelée de sauvegarder les registres dont elle a besoin avant de les utiliser. Bien évidemment, ces mêmes registres devront être restaurés avant de rendre le contrôle au programme appelant (à l'exception bien sûr de l'éventuel registre utilisé pour renvoyer la valeur d'une fonction).

Juste à titre informatif, remarquons qu'à partir du processeur 80186 l'instruction `ENTER m, 0`, allouant *m* bytes pour des variables locales, équivaut à la suite d'instructions

```

PUSH EBP
MOV  EBP, ESP
SUB  ESP, m

```

Symétriquement, l'instruction `LEAVE` correspond à la suite d'instructions

```

MOV  ESP, EBP
POP  EBP

```

Avant de conclure, précisons que la zone de la pile correspondant à une procédure est appelée le *stack frame*.

Enfin, nous pouvons remarquer que le mécanisme d'appel à une interruption reste, dans l'essentiel, assez similaire au mécanisme d'appel à une procédure décrit ci-dessus. Le processeur 80386 offre pour cela deux instructions analogues au `CALL` et au `RET` : `INT` (*interrupt*) et `IRET` (*return from interrupt*), respectivement. Nous n'en dirons pas plus sur ce sujet.

9.8 L'édition des liens

Au cours des exercices nous verrons comment construire des programmes résultant d'un premier module écrit en assembleur et d'un second écrit en C++. Comme décrit dans le cours théorique, une *édition des liens* devra alors être réalisée entre ces deux modules afin de pouvoir les mettre en commun.

Pour que ceci puisse être effectué correctement, certaines informations devront être rajoutées au code source des deux modules. Dans le cas du module assembleur :

- les symboles définis dans le module C++ devront être déclarés avant leur utilisation dans le module assembleur à l'aide de la directive `EXTERN` ;
- les symboles utilisés dans le module C++ mais définis dans le module assembleur devront être déclarés dans ce dernier à l'aide de la directive `GLOBAL`.

Pour le module C++ :

- les symboles définis dans le module assembleur devront être déclarés avant leur utilisation dans le module C++ en utilisant `extern "C"` ;
- les définitions C++ des symboles utilisés dans le module assembleur devront être précédées de `extern "C"`.

Ici, `"C"` indique que c'est la convention C qui est utilisée.

Annexe A

Effets des instructions sur les flags

Le tableau ci-dessous reprend les instructions modifiant l'un des quatre flags que nous avons rencontré dans ce syllabus, à savoir, le *overflow flag*, le *sign flag*, le *zero flag* et le *carry flag*.

Un ✓ indique que le flag correspondant est modifié de manière cohérente avec le fonctionnement de l'instruction. Les chiffres 0 et 1 indiquent, respectivement, que l'instruction met à 0 ou à 1 le flag considéré. Un signe ? indique que l'instruction rend le contenu du flag correspondant indéfini. Enfin, l'absence de symbole représente que la valeur du flag considéré reste inchangée après l'exécution de l'instruction.

Instruction	OF	SF	ZF	CF
ADC	✓	✓	✓	✓
ADD	✓	✓	✓	✓
AND	0	✓	✓	0
BSF			✓	
BSR			✓	
BT				✓
BTC				✓
BTR				✓
BTS				✓
CLC				0
CMC				✓
CMP	✓	✓	✓	✓
DEC	✓	✓	✓	
DIV	?	?	?	?
IDIV	?	?	?	?
IMUL	✓	?	?	✓
INC	✓	✓	✓	
IRET	✓	✓	✓	✓

Instruction	OF	SF	ZF	CF
MUL	✓	?	?	✓
NEG	✓	✓	✓	✓
OR	0	✓	✓	0
RCL	✓			✓
RCR	✓			✓
ROL	✓			✓
ROR	✓			✓
SAL	✓	✓	✓	✓
SAR	✓	✓	✓	✓
SHL	✓	✓	✓	✓
SHLD	?	✓	✓	✓
SHR	✓	✓	✓	✓
SHRD	?	✓	✓	✓
SBB	✓	✓	✓	✓
STC				1
SUB	✓	✓	✓	✓
TEST	0	✓	✓	0
XOR	0	✓	✓	0

Index d'instructions

ADC, 40
ADD, 39, 40
AND, 29, 31, 45
BSF, 34
BSR, 34
BTC, 35
BTR, 35
BTS, 35
BT, 35
CALL, 21, 52, 55
CBW, 44
CDQ, 44
CLC, 33
CMP, 45
CWDE, 44
CWD, 44
DEC, 41
DIV, 43, 44
ENTER, 55
IDIV, 43
IMUL, 42
INC, 40, 41
INT, 55
IRET, 55
JAE, 47
JA, 47
JBE, 47
JB, 47
JCXZ, 47
JC, 47
JECXZ, 47
JE, 46, 47
JGE, 47
JG, 47
JLE, 47
JL, 47
JMP, 46, 52
JNAE, 47
JNA, 47
JNBE, 47
JNB, 47
JNC, 47
JNE, 47
JNGE, 47
JNG, 47
JNLE, 47
JNL, 47
JNO, 47
JNP, 47
JNS, 47
JNZ, 47
JO, 47
JPE, 47
JPO, 47
JP, 47
JS, 47
JZ, 47
LEAVE, 55
LEA, 43
LOOPE, 49
LOOPNE, 49
LOOPNZ, 49
LOOPZ, 49
LOOP, 49
MOVSX, 24
MOVZX, 23, 24
MOV, 15, 16, 23, 25–29, 35, 39, 54
MUL, 41–43
NEG, 44
NOT, 29–31
OR, 29, 30
POPAD, 52
POPAW, 52
POPA, 21, 52
POPPD, 52
POPPW, 52
POPF, 52
POP, 51, 54, 55

PUSHAD, 51, 52
PUSHAW, 52
PUSHA, 21, 52
PUSHFD, 52
PUSHFW, 52
PUSHF, 52
PUSH, 51, 53
RCL, 34
RCR, 34
RETF, 52
RETN, 52
RET, 52, 54, 55
ROL, 34
ROR, 34
SAL, 32
SAR, 32
SBB, 41
SETNZ, 49
SHLD, 35, 36
SHL, 32, 43
SHRD, 35, 36
SHR, 32
STC, 33
SUB, 40
TEST, 45
XCHG, 24
XOR, 29

Bibliographie

- [1] Raymond Devillers. Fonctionnement des ordinateurs, aspects matériels et logiciels (syllabus). Université Libre de Bruxelles, octobre 1991.
- [2] William B. Giles. *Assembly language programming for the Intel 80XXX family*. Prentice-Hall, 1991.
- [3] The NASM development team. *NASM — The netwide assembler (version 0.98.35)*, 2002.
- [4] Yves Roggeman. Langage d'assemblage (syllabus). Université Libre de Bruxelles, septembre 1993.