Assembleur i8086

Philippe Preux
IUT Informatique du Littoral

Année universitaire 95 – 96

Avertissement

Ce document décrit le langage d'assemblage étudié et utilisé dans le cadre des TP d'architecture. Il est avant tout à considérer comme un manuel de référence. Il ne s'agit pas d'en effectuer une lecture séquentielle du début à la fin. Les chapitres 1, 2 et 4 présentent les principes de base. Les chapitres 3 et 5 sont à utiliser en cas de besoin. Le chapitre 6 présente des études de cas montrant comment écrire un programme en assembleur, depuis la spécification du problème, jusqu'à son exécution.

Une règle d'or (valable pour la programmation en assembleur mais aussi dans n'importe quel langage et même en toute généralité en informatique) :

L'ordinateur ne se trompe jamais : il ne fait que ce qu'on lui demande de faire.

Aussi, la programmation en assembleur étant quelque peu subtile, il s'agit de réfléchir avant de commencer à taper un programme, de continuer à réfléchir pendant qu'on le tape, à poursuivre l'effort pendant qu'on l'assemble et que l'on corrige les erreurs de syntaxe, et de ne pas se laisser aller quand enfin, on en arrive à son exécution qui donnera rarement le résultat attendu du premier coup avant une bonne période d'entraînement. Pendant tout ce temps, rappelez-vous la règle d'or et gardez votre calme. Ça ne sert à rien de s'énerver : l'ordinateur a tout son temps et c'est lui qui « décide » si votre programme marche ou pas !

On se reportera au document intitulé « Méthodologie de programmation en assembleur » où est décortiquée la manière de concevoir et d'écrire des programmes en assembleur.

Afin de simplifier l'exposé qui suit, nous nous en sommes tenus à la programmation dans le modèle mémoire dit `SMALL` (ou `TINY`). Pour la programmation dans des modèles mémoires plus étendus, on se référera au manuel de l'assembleur.

Chapitre 1

Introduction

Dans ce premier chapitre, nous introduisons rapidement les notions de processeur, mémoire, adresse et registre et décrivons le codage des valeurs numériques.

1.1 Un processeur, en deux mots

Sans entrer dans les détails qui seront vus en cours, nous présentons le minimum à savoir sur ce qu'est un processeur.

Un processeur constitue le cœur de tout ordinateur : il exécute les instructions qui composent les programmes que nous lui demandons d'exécuter. Les instructions sont stockées en mémoire (en dehors du processeur). Ces instructions (dont l'ensemble compose le *langage d'assemblage*, ou *assembleur*) sont très simples mais n'en permettent pas moins, en les combinant, de réaliser n'importe quelle opération programmable.

Pour exécuter un programme, le processeur lit les instructions en mémoire, une par une. Il connait à tout instant l'adresse (l'endroit dans la mémoire) à laquelle se trouve la prochaine instruction à exécuter car il mémorise cette adresse dans son *compteur ordinal*.

Les instructions agissent sur des données qui sont situées soit en mémoire, soit dans des registres du processeur. Un registre est un élément de mémorisation interne au processeur et contenant une valeur. Les registres sont en nombre très limité, 14 en l'occurence pour le 8086. Pour accéder une donnée en mémoire, il faut spécifier son adresse. Pour accéder une donnée dans un registre, il faut spécifier son nom (chaque registre possède un nom qui est une chaîne de caractères).

Le 8086 est un processeur 16 bits, c'est-à-dire qu'il traite des données codées sur 16 bits.

Pour terminer, on donne quelques définitions :

- un *bit* est une valeur binaire qui, par convention, peut prendre la valeur 0 ou 1 ;
- un *octet* est une donnée codée sur 8 bits ;
- un *mot* est une donnée codée sur 16 bits ;
- un *Koctet* est un ensemble de 1024 octets.

1.2 La mémoire

æ

La mémoire est une séquence d'octets, tous numérotés de manière unique par un entier compris entre 0 et $N-1$. On dit alors que la capacité de la mémoire est de N octets. On a toujours N qui est une puissance de 2 ($N = 2^m$). On appelle *adresse effective* (AE) d'un octet en mémoire le numéro qui lui est associé (cf. fig 1.1).

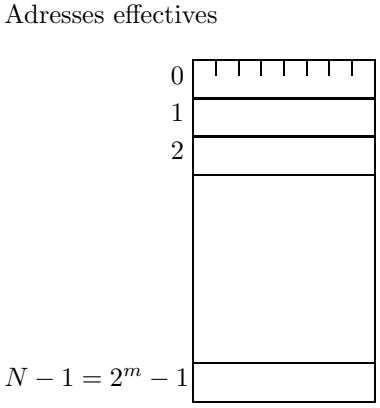


FIG. 1.1 – *Une mémoire de N octets.*

Pour le 8086, et donc pour nous programmeurs du 8086, un octet est désigné par un couple

(numéro de segment, déplacement dans le segment)

qui constituent une *adresse segmentée* (AS).

Un *segment* est un ensemble de 64 Koctets consécutifs. Le *déplacement* (ou offset) spécifie un octet particulier dans un segment (cf. fig. 1.2). Segment et déplacement sont codés sur 16 bits et peuvent donc prendre une valeur comprise entre 0 et 65535. On a une relation entre adresses effectives et couple (segment, déplacement) sous la forme de l'équation :

$$\text{adresse effective} = 16 \times \text{ segment} + \text{déplacement}$$

æ

Une conséquence de la relation énoncée plus haut entre adresse effective et adresse segmentée est qu'un octet d'adresse effective donnée peut être accédé de plusieurs manières. Plus précisément, à chaque AE correspond $2^{12} = 4096$ AS différentes (cf. fig. 1.3).

æ

Lors de son exécution, un programme utilise plusieurs segments mémoire (cf. fig. 1.4) :

- un segment contient les instructions du programme (le *segment de code*) ;
- un segment contient les données du programme (le *segment de données*) ;
- un segment contient la pile du programme (le *segment de pile*). Ce segment très important sera décrit au chapitre 4.

Ces trois segments peuvent se situer n'importe où en mémoire et même se recouvrir partiellement ou totalement. Leur attribution en mémoire est généralement réalisée automatiquement par le système d'exploitation sans que le programmeur ait à s'en soucier. En respectant un certain protocole (suivi dans ce manuel), on est certain de référencer le bon segment sans se poser de questions trop compliquées.

1.3 Les registres du 8086

æ

Le 8086 possède 14 registres de 16 bits (cf. fig. 1.5). Ce sont :

ax registre d'usage général contenant des données. Les 8 bits de poids faible se nomment `al` et les 8 bits de poids fort se nomment `ah`.

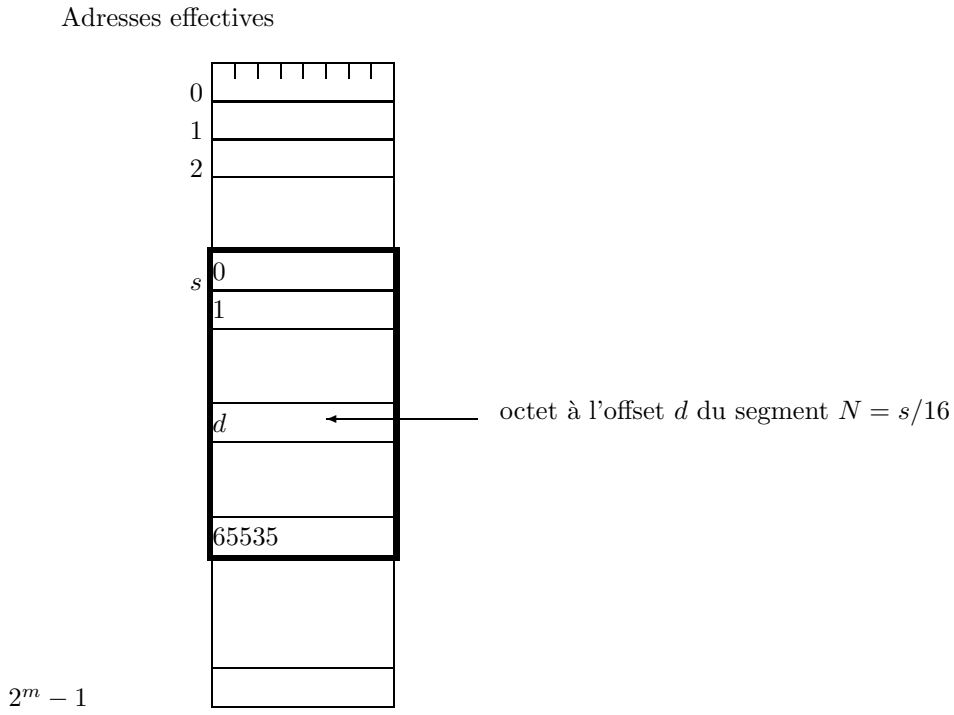


FIG. 1.2 – *Adressage segmenté. En traits gras, on a représenté le segment commençant à l'adresse effective s dont le numéro est donc* $N = \frac{s}{16}$*. s doit être un multiple de 16 pour être le* 1^{er} *octet d'un segment.*

bx registre d'usage général contenant des données. Comme `ax`, `bx` se décompose en `bl` et `bh`.

cx registre d'usage général contenant des données. Comme `ax`, `cx` se décompose en `cl` et `ch`.

dx registre d'usage général contenant des données. Comme `ax`, `dx` se décompose en `dl` et `dh`.

si registre d'usage général contenant généralement le déplacement dans un segment d'une donnée.

di registre d'usage général contenant généralement le déplacement dans un segment d'une donnée.

bp registre utilisé pour adresser des données dans la pile (cf. section 4.4).

sp registre pointeur de pile. Ce registre sera décrit plus loin (cf. section 4.2.4).

ip registre pointeur d'instruction (*compteur ordinal*). Ce registre indique la prochaine instruction à exécuter.

flags registre d'indicateurs de l'état du processeur. Certains bits de ce registre portent des noms. Ce sont tous des indicateurs binaires :

- **O** le bit d'*overflow* est positionné par la plupart des instructions arithmétiques pour indiquer s'il y a eut un débordement de capacité lors du calcul (un nombre trop grand ou trop petit)
- **D** bit de direction. Il sera décrit plus loin (cf. section 3.3.6)
- **S** le bit de signe est positionné par la plupart des instructions arithmétiques pour indiquer le signe du résultat (positif ou négatif – cf. section 1.4.2)
- **Z** le bit de zéro est positionné par la plupart des instructions arithmétiques pour indiquer que le résultat du calcul est 0

C le bit de *carry* (retenue) est positionné par la plupart des instructions arithmétiques pour indiquer si le calcul a engendré une retenue qui devra être reportée sur les calculs suivants (cf. section 3)

A le bit dit *auxiliary carry* (retenue auxiliaire) est positionné par la plupart des instructions arithmétiques pour indiquer une retenue entre bits de poids faible et bits de poids forts d'un octet, d'un mot ou d'un double mot (cf. section 3)

P le bit de parité est positionné par la plupart des instructions arithmétiques. Il indique si les 8 bits de poids faible du résultat comportent un nombre pair de 1.

Le 8086 comporte également des registres 16 bits pour contenir des numéros de segment :

cs *code segment* segment contenant le programme en cours d'exécution

ds *data segment* segment contenant les données

es registre segment auxiliaire pour adresser des données

ss *stack segment* segment contenant la pile

Les valeurs des registres `cs`, `ds` et `ss` sont automatiquement initialisées par le système d'exploitation au lancement du programme. Dès lors, ces segments sont implicites, c'est-à-dire que si l'on désire accéder à une donnée en mémoire, il suffit de spécifier son offset sans avoir à se soucier du segment.

1.4 Le codage des nombres

Les instructions du 8086 manipulent des valeurs numériques entières codées sur 1 ou 2 octets.
Nous décrivons ici le codage des nombres entiers. Nous distinguons quatre types de codage :

- non signé pour des nombres entiers forcément positifs ;
- signé pour des nombres entiers positifs ou négatifs.
- décimal compacté, ou décimal codé binaire
- décimal non compacté

C'est au programmeur de décider du codage qu'il utilise et à écrire son programme en conséquence. L'ordinateur ne sait pas quel est le codage utilisé. Il exécute simplement des instructions.

Nous décrivons ces codages et quelques notions qui leur sont liées.

Nous supposerons dans ce qui suit que les données sont codées sur un octet (8 bits). Tout ce qui est dit peut-être adapté immédiatement pour des données qui seraient codées sur un mot, un double mot ou un nombre quelconque de bits.

1.4.1 Représentation non signée des nombres entiers

Avec l bits, il est possible de coder 2^l valeurs différentes. Si $l = 8$, on peut donc coder 256 valeurs différentes. On peut établir le codage suivant, sur 8 bits :

Valeur	Codage
0	00000000
1	00000001
2	00000010
3	00000011
...	...
84	01010100
...	...
254	11111110
255	11111111

Ce codage permet de représenter tout nombre entier entre 0 et 255 (0 et $2^l - 1$) avec 8 bits (l bits). Du fait que l'on ne représente que des nombres positifs, ce codage est qualifié de *non signé*.

1.4.2 Représentation signée des nombres entiers

Si l'on veut pouvoir représenter des nombres positifs et négatifs, l'idée naturelle consiste, plutôt qu'à coder des nombres compris entre 0 et 255, à représenter des nombres entre -128 et +127, ce qui représente toujours 256 valeurs différentes à coder avec 8 bits.

Valeur	Codage
-128	10000000
-127	10000001
...	...
-45	11010011
...	...
-1	11111111
0	00000000
1	00000001
2	00000010
...	...
84	01010100
...	...
126	01111110
127	01111111

Ce codage se nomme *codage par complément à deux*. L'obtention du codage par complément à deux d'un nombre est indiqué plus bas.

On constate, ceci est très important pour que le processeur traite indifféremment des nombres en codage signé ou non, que le fait de faire une opération (par exemple ajouter 1 à une valeur) fonctionne quel que soit le codage utilisé. Faire la somme au niveau du codage binaire ou au niveau des nombres représentés donne le même résultat.

Prenons un exemple :

25	→	`00011001`
+ 36	→	`00100100`
61	←	`00111101`

Dans le codage signé, on constate que le bit de poids fort vaut 1 pour tous les nombres négatifs, 0 pour tous les nombres positifs. Ce bit indique donc le signe du nombre codé. Aussi, ce bit se nomme le *bit de signe*.

Il faut bien prendre garde que si le bit de signe vaut 0, la valeur représentée est la même en codage signé ou non signé. Par contre, si le bit de signe vaut 1, la valeur codée est supérieure à 127 en codage non signé, inférieure à 0 en codage signé.

1.4.3 Codage décimal

En codage décimal[1], un chiffre décimal (compris donc entre 0 et 9) est codé sur 4 bits. 4 bits codant en principe 16 valeurs différentes, seules 10 de ces 16 combinaisons sont effectivement utilisées en codage décimal, les autres n'ayant alors pas de sens. Le codage est le suivant :

1. on parle aussi de *binaire codé décimal*

Valeur	Codage
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

On distingue deux types de représentation, compactée ou non. Un chiffre nécessitant 4 bits pour son codage, on peut représenter deux chiffres par octet. Ce codage porte le nom de *codage décimal compacté*. On peut également ne mettre qu'un seul chiffre par octet, les 4 bits de poids fort de l'octet étant inutilisés. On qualifie ce codage de *décimal non compacté*.

On reverra plus loin ces codages (cf. 3.3.6).

Le codage décimal est très utile pour passer de la représentation codée d'un nombre dans sa représentation en chaîne de caractères ASCII.

Par exemple, 19 se code :

décimal compacté	`00011001`
décimal non compacté	`00000001`
	`00001001`

Deux octets sont nécessaires dans la représentation non compactée.

1.4.4 Complément à deux d'un nombre

Nous indiquons ici comment obtenir le complément à deux d'un nombre. Cette méthode donne la représentation binaire signée d'un nombre négatif. L'algorithme est simple. Soit à calculer la représentation du nombre $-n$ (où n est un nombre positif). On effectue les opérations suivantes :

- écrire n sous forme binaire
- en complémenter tous les bits ($0 \rightarrow 1$, $1 \rightarrow 0$)
- ajouter la valeur 1 à ce nombre binaire.

Nous présentons cet algorithme sur un exemple. Soit à calculer le complément à deux de -23 :

- la représentation binaire sur 8 bits de 23 est `00010111`
- par complément des bits, on obtient : `11101000`
- en ajoutant 1, on obtient : `11101001`

Le complément à deux de `00010111` est donc `11101001`. La représentation binaire signée de -23 est donc `11101001`.

1.4.5 Extension du signe

Étendre le signe d'une donnée consiste, lors du transfert de la valeur d'un octet dans un mot à recopier le bit de signe de l'octet sur tous les bits de l'octet de poids fort du mot. Par exemple, l'extension du signe de `10011110` en mot donne `1111111110011110`.

1.5 Notation

Dans ce document, nous utiliserons la notation suivante :

- `10` indique la valeur numérique 10
- `AX` indique la valeur contenue dans le registre `AX`
- `(10)` indique la valeur qui se trouve à l'offset 10 dans le segment de données
- `(SI)` indique la valeur qui se trouve à l'offset dont la valeur se trouve dans le registre `SI`
- `(DS:SI)` indique la valeur qui se trouve à l'offset contenu dans le registre `SI` et le segment dans le registre `DS`

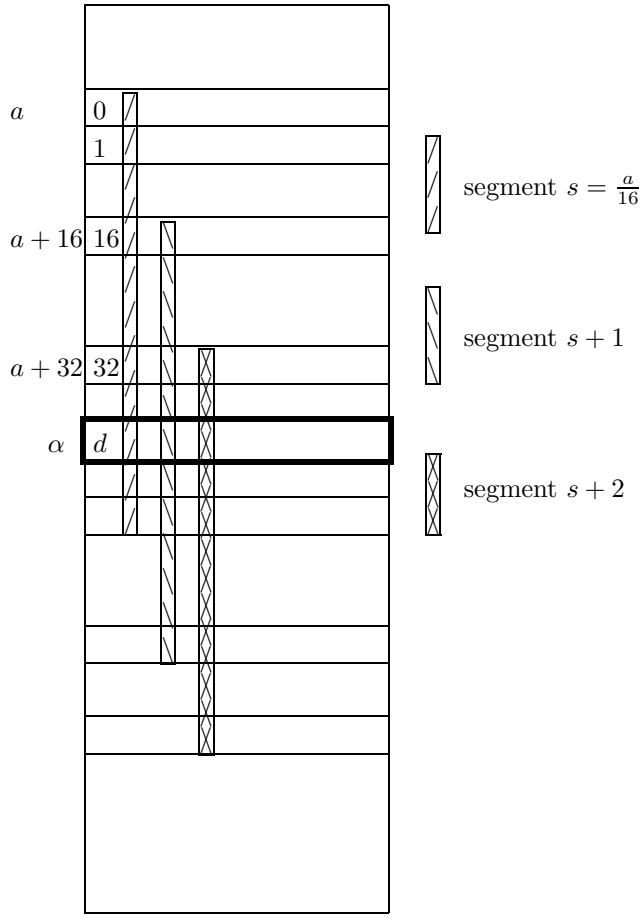


FIG. 1.3 – *Les segments : on a représenté quelques segments pour montrer leur chevauchement et le fait qu'un octet-mémoire possède plusieurs adresses segmentées. Par exemple, l'octet d'adresse effective* α *peut être adressé de multiples façons, selon que l'on considère qu'il se trouve dans le segment* s, $s+1$, $s+2$, ... *Dans chacun de ces cas, l'adresse segmentée de l'octet* α *est (* s, d *), (* $s+1$, $d-16$ *), (* $s+2$, $d-32$ *), ... Notons que chaque octet-mémoire appartient à* $2^{12} = 4096$ *segments différents.*

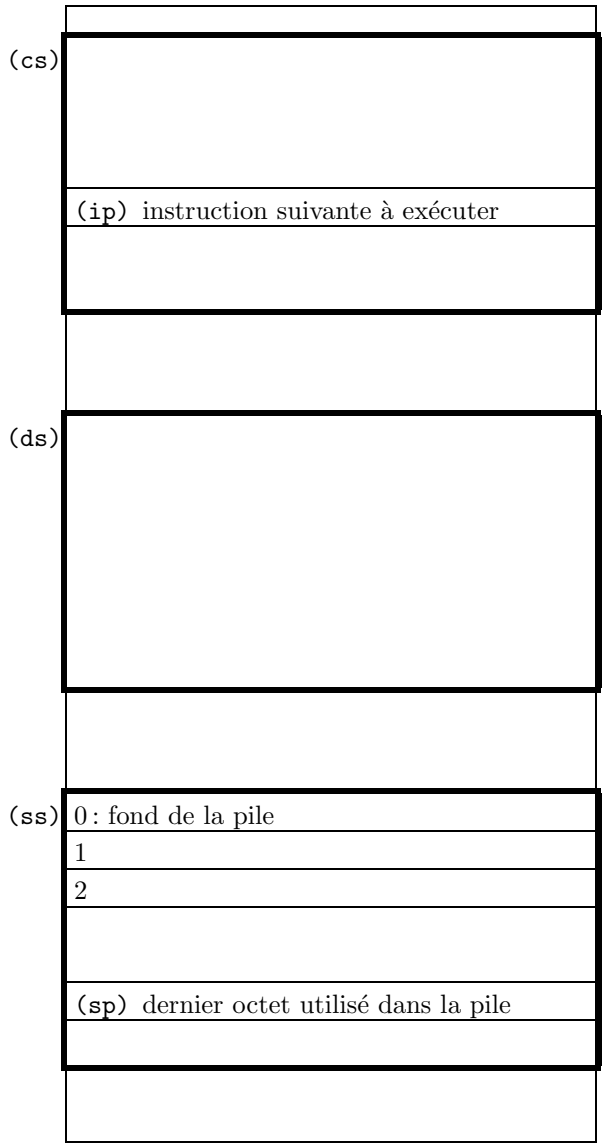


FIG. 1.4 – *Structure de la mémoire. On a représenté en traits gras les segments de code, de données et de pile qui sont créés lors du déclenchement de l'exécution d'un programme (de haut en bas – en réalité, ils peuvent bien entendu être situés n'importe où en mémoire). Chaque segment est référencé par un registre de segment. Les différents segments peuvent se recouvrir partiellement ou complétement.*

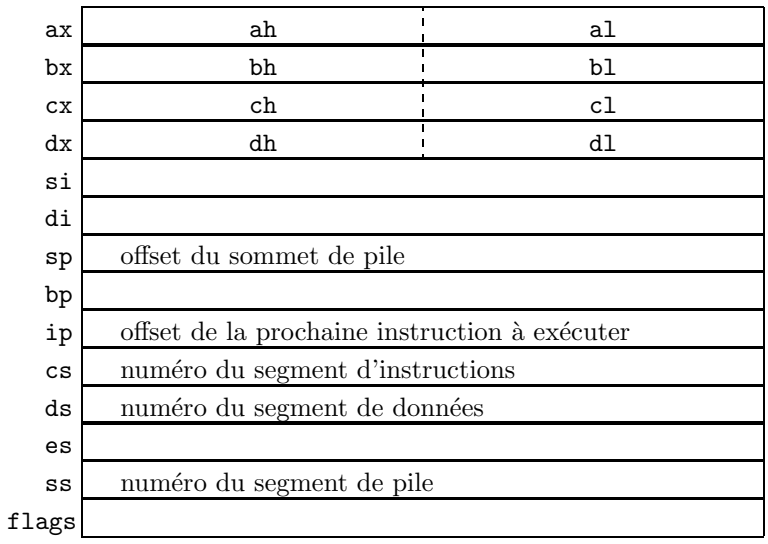


FIG. 1.5 – *Les registres du 8086 et leur utilisation. Chaque registre contient 16 bits.*

Chapitre 2

Anatomie d'un programme en assembleur

Un programme en assembleur a une forme bien particulière. Chaque ligne d'un source assembleur comporte une instruction. Chaque ligne est composée de champs. De gauche à droite, on a :

- le champ étiquette, qui peut être vide
- le champ mnémonique (le nom de l'instruction)
- le champ opérande (les arguments de l'instruction), qui peut être vide
- le champ commentaire, qui peut être vide

Une étiquette est un identificateur composé de lettres, chiffres et de caractères `$`, `%`, `_` et `?` Quelques exemples d'étiquettes valides : `boucle`, `fin_de_tant_que`, ...

Un mnémonique est, généralement, composé uniquement de lettres. Quelques mnémoniques que nous retrouverons souvent : `MOV`, `CMP`, `LOOP`, ... On pourra indifféremment écrire les mnémoniques avec des lettres minuscules ou majuscules.

Les opérandes d'une instruction indiquent les données à traiter, soit sous la forme de valeurs constantes, soit en spécifiant l'adresse mémoire (l'emplacement en mémoire) où se trouve la donnée, soit le nom d'un registre contenant la donnée ou contenant l'adresse mémoire de la donnée.

Un commentaire commence par un caractère `;` et se termine en fin de ligne.

Bien que la syntaxe soit assez souple, nous respecterons toujours les règles suivantes :

- une instruction par ligne
- le mnémonique de l'instruction commence à la colonne 10
- une définition d'étiquette commence en colonne 1
- les opérandes des instructions commencent en colonne 20
- les commentaires en fin de ligne commencent en colonne 35
- les blocs de commentaires sont alignés sur les mnémoniques et commencent donc en colonne 10 (c'est-à-dire, le `;` de début de commentaire se trouve en colonne 10)

2.1 Exemple

```
        ; Forme générale d'un fichier source en assembleur 8086
        ; Nom du fichier  :    modele.asm
        ; Date d'ecriture :    15 Nov. 1993
        ; Objet           :    indiquer ici à quoi sert le
        ;                      programme, comment on l'utilise, ...

        .MODEL    small          ; définit le modèle mémoire
        .STACK    200h           ; taille de la pile

        .DATA                    ; définition des données du programme
donnee1 DB        ....
        ....
        ....
        ....

        .CODE                    ; les instructions composant le programme
        ....
        ....
        ....
        ....
        mov       ah,4ch         ;
        int       21h            ; terminaison du programme
        ....
        ....
        END                      ; fin du source
```

Un programme en assembleur commencera donc, par convention, par un bloc de commentaire indiquant l'auteur du programme, la date de réalisation du programme ainsi que son utilité et utilisation.

On trouve ensuite quelques lignes spécifiant des informations qui seront détaillées plus tard.

On trouve ensuite une zone de définition des données qui débute par la pseudo-instruction[1] `.DATA`.

On trouve ensuite les instructions du programme, lesquelles débutent par une pseudo-instruction `.CODE`. Parmi ces instructions, on trouvera généralement la séquence indiquée qui termine l'exécution du programme. Enfin, la dernière ligne du programme est la pseudo-instruction `END` qui indique la fin du source.

On notera que la plupart des instructions ont leur premier opérande qui indique une donnée et l'endroit où il faut ranger le résultat (donnée et résultat sont stockés au même endroit), le deuxième qui indique une donnée. Par exemple, dans le squelette de programme ci-dessus, dans l'instruction `mov ah, 4ch`, `ah` est la destination de l'instruction, `4ch` la donnée source.

2.1.1 Spécifier une donnée en mémoire (une variable)

Il y a 16 manières de spécifier le déplacement dans un segment mémoire :

- `étiquette`
- `[bx]`
- `[si]`
- `[di]`
- `bx+étiquette`

1. une *pseudo-instruction*, ou *directive d'assemblage*, est une commande en langage d'assemblage qui n'est pas une instruction du processeur. Elle est seulement utilisée pour donner des informations à l'assembleur.

- `bx+si+étiquette`
- `bx+di+étiquette`
- `[bx+si]`
- `[bx+di]`
- `[bp+si]`
- `[bp+di]`
- `bp+étiquette`
- `bp+si+étiquette`
- `bp+di+étiquette`
- `si+étiquette`
- `di+étiquette`

`étiquette` indique l'étiquette d'une donnée définie dans la section `.data` du programme (*cf.* plus bas). où `déplacement` est une constante codée sur 16 bits.

Notons au passage que l'on peut récupérer la valeur des adresses de début de segment de données et de code en utilisant les symboles `@data` et `@code` respectivement.

Il faut noter que toutes les références à des opérandes en mémoire sont implicitement situées dans le segment dont le numéro est contenu dans le registre `DS`. Lorsque cela n'est pas le cas, cela sera mentionné explicitement.

2.1.2 Les constantes

Les constantes numériques

On peut spécifier les constantes numériques dans les bases 2, 8, 10 et 16. Une constante numérique commence toujours par un chiffre. A priori, une constante est notée en base 10. Pour l'exprimer en base 2, une constante ne doit comporter que des chiffres 0 et 1 et être terminée par la lettre `b`. Pour l'exprimer en base 8, une constante ne doit comporter que des chiffres 0, 1, 2, 3, 4, 5, 6 ou 7 et se terminer par la lettre `o`. Pour l'exprimer en base 16, une constante ne doit comporter que des chiffres de 0 à 9 ou des lettres de `a` à `f` pour les valeur hexadécimales correspondant aux nombres décimaux 10, 11, 12, 13, 14 et 15 et se terminer par la lettre `h`.

Par exemple :

- `100110b` est une constante binaire dont la valeur décimale est 38
- `37o` est une constante octale dont la valeur décimale est 31
- `4ch` est une constante hexadécimale dont la valeur décimale est 76

Puisqu'une constante numérique doit toujours commencer par un chiffre, une constante hexadécimale dont le premier caractère est une lettre sera précédée d'un `0`. Ainsi, la valeur `c3` sera notée `0c3h` dans un source assembleur.

Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entre `'`. Si on veut mettre un caractère `'` dans une constante, il suffit de la doubler. Exemples :

- `'hello world'`
- `'l''arbre'`

Déclaration d'une constante

La directive `EQU` associe une valeur à un symbole qui pourra être ensuite utilisé dans le programme à la place de la constante qu'elle définit.

```
nom EQU constante
```

Par exemple, la commande :

```
TROIS    EQU       3
```

définit une constante qui s'appelle `TROIS`. dont la valeur est 3. Une fois cette déclaration effectuée, toute occurence de l'identificateur `TROIS` sera remplacée par la valeur indiquée.

2.1.3 Les déclarations de variables

L'espace mémoire est utilisé pour stocker des constantes (chaînes de caractères, valeurs numériques, ...) ou des variables. Avant d'employer une variable, il faut préalablement la déclarer (comme dans tout langage). Déclarer une variable (et ceci est vrai aussi dans les langages de haut niveau comme Pascal, mais c'est un peu caché) revient toujours à réserver un emplacement en mémoire pour y stocker la valeur de cette variable.

La directive DB

```
[nom] DB constante [, constante]
```

Réserve et initialise un octet. `nom` est un symbole permettant d'accéder à cet octet [2]. Par exemple, les 4 lignes suivantes :

```
OCTET          DB    36
DEUX_OCTETS    DB    43, 26
LETTRE         DB    'e'
CHAINE         DB    'hello world !', 13, 10, '$'
TABLEAU        DB    10 dup (0)
```

définissent 5 symboles comme suit :

- `NB_OCTET` référence une donnée codée sur un octet dont la valeur sera initialisée à 36
- `DEUX_OCTETS` référence une donnée codée sur un octet dont la valeur sera initialisée à 43, l'octet suivant étant initialisé avec 26. On peut considérer que `DEUX_OCTETS` référence un tableau de 2 octets ;
- `LETTRE` référence une donnée dont la valeur sera initialisée avec le code ASCII du caractère 'e' ;
- `CHAINE` référence une chaîne de caractères composée des caractères 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', ' ', '!', des caractères de code 13 (retour en début de ligne) et 10 (aller à la ligne suivante) et enfin du caractère '$' [3].
- `TABLEAU` référence un tableau de 10 octets dont la valeur initiale des éléments est 0

On constate donc que la valeur initiale d'un symbole peut être exprimée de différentes manières :

- une valeur
- plusieurs valeurs séparées par des virgules

2. la notation `[xxx]` indique que `xxx` est facultatif

3. les caractères de code 13 et 10 placés en fin de chaîne entraînent le passage à la ligne après l'affichage du message ; le caractère '$' indique la fin de la chaîne de caractères. En général, toute chaîne de caractères devant être affichée à l'écran se terminera par un caractère '$'.

– le code d'un caractère

– une chaîne de caractères, chacun des caractères initialisant un octet

On pourra généralement utiliser l'une de ces formes dans toute instruction, du moment que cela a un sens. On verra également d'autres manières de spécifier des valeurs par la suite. Celles-ci sont tout aussi valides pour n'importe quelle instruction, du moment qu'elles ont un sens.

La directive DW

```
[nom] DW constante [, constante]
```

Réserve et initialise un mot (16 bits). `nom` est un symbole permettant d'accéder à ce mot. Ainsi, les lignes suivantes définissent :

```
MOT     DW      1559
MOT2    DW      ((45 * 23) / 9 + 23)
TMOT    DW      20 DUP (0)
```

– le symbole `MOT` qui référence une une donnée codée sur un mot et qui sera initialisée avec la valeur 1559 ;

– le symbole `MOT2` qui référence une donnée codée sur un mot et qui est initialisée avec la valeur 138 ;

– le symbole `TMOT` qui référence un tableau de 20 mots initialisés avec la valeur 0

On voit donc sur cet exemple qu'une valeur constante peut être spécifiée avec une expression numérique, du moment que celle-ci est calculable lors de l'assemblage. Sans entrer dans les détails ici, on peut indiquer que la valeur d'une expression est calculable à l'assemblage si elle ne comporte que des valeurs numériques ou des constantes définies par des pseudo-instructions `EQU` ou une adresse. A ce propos, il est très important de bien noter la différence entre un symbole défini par une directive `EQU` et un symbole défini par une directive `DB` ou `DW`. Si nous considérons les exemples précédents :

– `TROIS` est une constante dont la valeur est 3 ;

– `OCTET`, `MOT`, `DEUX_OCTETS`, ... sont des constantes dont la valeur est l'adresse à laquelle seront stockées les valeurs 36, 1559, 43, 26, ...

Par analogie avec un programme Pascal, nous pouvons dire que `TROIS` est une constante, alors que les autres symboles (`OCTET`, `MOT`, ...) sont des variables qui seront initialisées avec les valeurs indiquées. Au cours de l'exécution du programme, la valeur se trouvant à l'adresse `OCTET` pourra être modifiée.

Notons que les octets composant un mot sont stockés de la manière suivante : le poids faible est mis dans le premier octet, le poids fort dans le deuxième octet. Ainsi, si on déclare :

```
M       DW      1234h
```

la valeur `34h` est mise à l'adresse `M` et la valeur `12h` à l'adresse `M+1`.

2.2 L'assemblage

Une fois un programme source assembleur saisi sous éditeur de textes dans un fichier dont l'extension est `ASM`, il faut le transformer en un programme exécutable. Pour un langage de haut niveau comme Pascal, cette phase se nomme la *compilation*. Pour un programme écrit en assembleur, cette transformation se nomme *assemblage*. Pour cela, on utilise la commande `TASM2` avec, en argument, le nom du fichier à assembler. Des erreurs de syntaxe seront éventuellement détectées qu'il faudra corriger avant de continuer. Une fois que l'assemblage a eu lieu sans détection d'erreur, on obtient un fichier de même nom que le fichier source et avec une extension `OBJ`. Il faut alors réaliser une *édition de liens*. Pour cela, on utilise la commande `TLINK2`

avec, en argument, le nom du fichier, sans extension. A l'issue de ce processus, on obtient un fichier portant le même nom que le fichier source et d'extension `EXE`. Ce fichier est exécutable.

Il est bon de noter que dans le programme exécutable, les instructions ont été codées. Ainsi, chaque instruction est codée par un octet ou une succession d'octets consécutifs en mémoire (segment de code). Dans le codage d'une instruction, on trouve le *code-opération* qui indique le mnémonique (1 ou 2 octets) suivi du ou des opérandes (valeur immédiate, registre, adresse).

FIG. 2.1 – *Le codage d'une instruction*

Chapitre 3

Les instructions du 8086

Dans ce chapitre, nous présentons une par une les instructions du 8086.

Modèle de description d'une instruction

Pour toutes les instructions possédant plusieurs modes d'adressage des opérandes, on trouvera un tableau présentant ces modes de la forme suivante ;

`xxx`	`operande1`, `operande2`, ...	commentaire
...		
...		

Dans la colonne de gauche, on trouve le mnémonique de l'instruction. Dans la colonne suivante, on trouve le mode d'adressage des opérandes. La colonne de droite absente dans certaines tables, donnent quelques informations complémentaires.

Les modes d'adressage sont indiqués de la manière suivante :

- `AL`, `AH`, ... un registre particulier
- `registre` un registre 8 ou 16 bits parmi `AL`, `AH`, `AX`, `BL`, `BH`, `BX`, `CL`, `CH`, `CX`, `DL`, `DH`, `DX`, `SI` ou `DI`.
- `variable` une étiquette de donnée définie dans la section `.data` du programme
- `registre/variable` un registre ou une étiquette de donnée
- `constante` une constante
- `etiquette` une étiquette correspondant à une instruction, donc définie dans la section `.code` du programme

La plupart des instructions modifient les indicateurs du registre `FLAGS`. Un tableau indique l'effet de l'instruction couramment décrite sur ces indicateurs. Il a la forme suivante ;

O	D	I	T	S	Z	A	P	C

La première ligne indique le nom des bits intéressants de `FLAGS`. La deuxième ligne (vide ici) indique l'effet de l'instruction sur un bit en particulier. On note :

* le bit est modifié en fonction du résultat de l'exécution de l'instruction

? le bit a une valeur indéfinie après l'exécution de l'instruction

1 le bit est mis à 1

0 le bit est mis à 0

Une case vide indique que l'indicateur n'est pas modifié par l'instruction.

3.1 Les instructions arithmétiques et logiques

3.1.1 Les instructions arithmétiques

Comme dans de nombreux processeurs, le 8086 possède des instructions $+$, $-$, $\times$ et $\div$ qui traitent des données entières codées sur un octet ou un mot. Pour faire des opérations sur des données plus complexes (des nombres flottants par exemple), on devra les programmer.

`ADD` **addition sans retenue**

```
ADD  registre/variable, registre
ADD  registre, registre/variable
ADD  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

L'instruction `ADD` effectue l'addition du contenu du registre source au registre destination, sans report de retenue, soit ;

destination ← source + destination

La retenue est positionnée en fonction du résultat de l'opération.

Par exemple, si les instructions suivantes sont exécutées ;

```
mov   ax, a9h
add   ax, 72h
```

alors le registre `AX` contient la valeur `1bh`, le bit `C` est positionné à la valeur 1, la retenue auxiliaire `A` est mise à 0.

Si nous considérons les deux instructions suivantes ;

```
mov   ax, 09h
add   ax, 3ah
```

alors le registre `AX` contient la valeur `43h`, le bit `C` est mis à 0, la retenue auxiliaire `A` est mise à 1.

`SUB` **soustraction sans retenue**

```
SUB  registre/variable, registre
SUB  registre, registre/variable
SUB  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

L'instruction `SUB` effectue la soustraction du contenu du registre source au registre destination, sans report de retenue, soit ;

destination ← destination - source

La retenue est positionnée en fonction du résultat de l'opération.

Par exemple, si les instructions suivantes sont exécutées ;

```
mov  ax, 39h
sub  ax, 18h
```

le registre `AX` contient ensuite la valeur 21h. Les bits `Z`, `S` et `C` du registre `FLAGS` sont mis à 0 car le résultat n'est pas nul, son signe est positif et aucune retenue n'est générée.

Si nous considérons les deux instructions suivantes ;

```
mov  ax, 26h
sub  ax, 59h
```

le registre `AX` contient ensuite la valeur `cdh`. Le bit `Z` est mis à zéro. Les bits `C`, `A` et `S` sont mis à 1.

`IMUL`
`MUL` **Les multiplications en assembleur**

IMUL registre/variable

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

Les deux instructions `IMUL` et `MUL` effectuent des multiplications.

L'instruction `IMUL` effectue la multiplication d'opérandes signés. L'instruction `MUL` effectue la multiplication d'opérandes non signés.

Les indicateurs de retenue (`C`) et de débordement (`O`) sont mis à un si le résultat ne peut pas être stockée dans l'opérande destination.

Une addition ou une soustraction de données codées sur n bits donne un résultat sur au plus $n+1$ bits. Le bit supplémentaire est la retenue et est stocké dans le bit `C` de `flags`. Par contre, une multiplication de deux données de n bits donne un résultat sur $2n$ bits.

Dans leur première forme qui prend une donnée 8 bits en opérande (donc le résultat est sur 16 bits), les instructions `mul` et `imul` effectuent le produit de la valeur contenue dans le registre `al` avec la valeur de l'opérande fourni. Le résultat est placé dans le registre `ax`.

Dans leur deuxième forme qui prend une donnée 16 bits en opérande (donc le résultat est sur 32 bits), les instructions `mul` et `imul` effectuent le produit de la valeur contenue dans le registre `ax` avec la valeur de l'opérande fourni. Le résultat est placé dans la paire de registres `dx` et `ax`. `dx` contient le poids fort du résultat, `ax` le poids faible.

Exemple de multiplication sur 8 bits : soient les instructions suivantes :

```
mov   al, 4
mov   ah, 25
imul  ah
```

À l'issue de l'exécution de ces 3 instructions, le registre `AH` contient la valeur 100, produit de 4 par 25.

Pour bien comprendre la différence entre les instructions `imul` et `mul`, regardons les exemples suivants :

```
mov  bx, 435
mov  ax, 2372
imul bx
```

À l'issue de l'exécution de ces 3 instructions, `ax` contient la valeur `be8c` et `dx` la valeur `f`, soit la valeur hexadécimale `fbe8c`, c'est-à-dire 1031820, le produit de 435 par 2372. Les deux données étant positives, le résultat est le même que l'on utilise l'instruction `imul` ou l'instruction `mul`.

Considérons maintenant la séquence d'instructions :

```
mov  bx, -435
mov  ax, 2372
imul bx
```

À l'issue de leur exécution, `ax` contient la valeur `4174` et `dx` contient la valeur `fff0`, soit la valeur -1031820. Si l'on remplace `imul` par `mul`, le résultat n'a pas de sens.

IDIV
DIV **Les divisions en assembleur**

IDIV registre/variable

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Les deux instructions `DIV` et `IDIV` réalisent les opérations de division et de calcul de reste. `DIV` l'effectue sur des données non signées, `IDIV` sur des données signées.

Dans tous les cas, le dividende est implicite. Le diviseur est fourni en opérande. Le résultat se compose du quotient et du reste de la division. Le reste est toujours inférieur au diviseur. On a le choix entre :

- la division d'une donnée 16 bits stockée dans `AX` par une donnée 8 bits qui fournit un quotient dans `AL` et un reste dans `AH` sur 8 bits.
- la division d'une donnée 32 bits stockée dans la paire de registres `DX` (poids fort) et `AX` (poids faible) par une donnée 16 bits qui fournit un quotient dans `AX` et un reste dans `DX` sur 16 bits.

Soient les quelques lignes d'assembleur suivantes :

```
mov  ax, 37
mov  dx, 5
div  dl
```

Après leur exécution, le registre `AX` contient la valeur 7, quotient de 37 par 5 et le registre `DX` contient le reste de la division, 2.

Pour les deux instructions, si le diviseur de la division est nul, un message `Division par zero` sera affiché automatiquement.

Dans le cas d'une division signée `IDIV`, le reste a le même signe que le dividende et sa valeur absolue est toujours inférieure au diviseur.

Pour bien comprendre le fonctionnement de ces deux instructions, prenons l'exemple suivant :

```
mov  bx, 435
mov  ax, 2372
div  bx
```

À l'issue de l'exécution de cette séquence d'instructions, le registre `ax` contiendra la valeur 5 qui est le quotient de 2372 par 435, et le registre `dx` vaudra 197 qui est le reste de la division. Si on remplace `div` par `idiv`, le résultat est inchangé puisque le diviseur et le dividende sont tous deux positifs.

Si l'on considère maintenant la séquence :

```
mov  bx, -435
mov  ax, 2372
idiv bx
```

on aura ensuite `ax` qui contient la valeur -5 (soir `fffb` en hexadécimal) et `dx` la valeur 197. Si l'on remplace `idiv` par `div`, le résultat n'a pas de sens.

3.1.2 Incrémentation, décrémentation

Nous voyons ici deux types d'instructions très fréquemment utilisées et qui sont en fait des cas particuliers des instructions d'addition et de soustraction, l'*incrémentation* et la *décrémentation*. Incrémenter signifie « ajouter 1 », alors que décrémenter signifie « retirer 1 ». Notons cependant que l'on utilise souvent les termes incrémenter et décrémenter même si les quantités ajoutées ou retirées sont différentes de 1, en général lorsque la variable modifiée est un compteur.

`DEC` **décrémentation**

DEC registre/variable

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

`DEC` soustrait 1 au contenu de l'opérande, sans modifier l'indicateur de retenue.

Soient les instructions assembleur suivantes ;

```
mov  al, 01h
dec  al
```

`AL` contient ensuite la valeur 00. Le bit `Z` est mis à 1, les bits `O`, `P`, `A` et `S` mis à 0.

`INC` **incrémentation**

INC registre/variable

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

`INC` ajoute 1 au contenu de l'opérande, sans modifier l'indicateur de retenue.

Soient les instructions assembleur suivantes ;

```
mov  al, 3fh
inc  al
```

`AL` contient ensuite la valeur `40h`. Le bit `Z` est mis à 0 (le résultat de l'incrémentation n'est pas nul), l'indicateur de retenue auxiliaire bit `A` est mis à 1 (passage de retenue entre les bits 3 et 4 lors de l'incrémentation), les bits bit `O` et bit `S` mis à 0 (pas de débordement de capacité, le signe du résultat est positif).

3.1.3 Opposé d'un nombre

`NEG` **Négation par complément à 2**

NEG registre/variable

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

`NEG` transforme la valeur d'un registre ou d'un opérande mémoire en son complément à deux. Ainsi, après exécution des instructions ;

```
mov  ax, 35
neg  ax
```

le registre `AX` contient la valeur -35.

3.1.4 Les instructions booléennes et logiques

Ces instructions disponibles sur tous les processeurs travaillent sur les données au niveau des bits (et non sur des valeurs numériques comme les instructions vues jusqu'à présent).

`AND` **et-logique**

```
AND  registre/variable, registre
AND  regisre, registre/variable
AND  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

`AND` réalise un et-logique bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

Considérons les deux lignes suivantes :

```
mov   al, 36h
and   al, 5ch
```

Le registre `AL` contient ensuite la valeur 14h obtenue de la manière suivante ;

36h	0011	0110
$\wedge$ 5ch	0101	1100
14h	0001	0100

Les bits `S` et `Z` sont mis à zéro et le bit `P` à 1.

`OR` **ou-logique**

```
OR  registre/variable, registre
OR  regisre, registre/variable
OR  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

`OR` réalise un ou-logique bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

Considérons les deux lignes suivantes :

```
mov   al, 36h
or    al, 5ch
```

Le registre `AL` contient ensuite la valeur `7eh` obtenue de la manière suivante ;

36h	0011	0110
$\vee$ 5ch	0101	1100
7eh	0111	1110

Les bits `S` et `Z` sont mis à zéro et le bit `P` à 1.

`XOR` **ou-exclusif**

```
XOR  registre/variable, registre
XOR  regisre, registre/variable
XOR  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

`XOR` réalise un ou-exclusif bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

Considérons les deux lignes suivantes :

```
mov   al, 36h
and   al, 5ch
```

Le registre `AL` contient ensuite la valeur `6ah` obtenue de la manière suivante ;

36h	0011	0110
$\oplus$ 5ch	0101	1100
6ah	0110	1010

Les bits `S` et `Z` sont mis à zéro et le bit `P` à 1.

`NOT` **Négation logique**

```
NOT  registre/variable
```

O	D	I	T	S	Z	A	P	C

`NOT` transforme la valeur d'un registre ou d'un opérande mémoire en son complément logique bit à bit.

Considérons les deux lignes suivantes :

```
mov   al, 36h
not   al
```

Le registre `AL` contient ensuite la valeur `c9h` obtenue de la manière suivante ;

$\neg$ 36h	0011	0110
c9h	1100	1001

Les bits `S` et `P` sont mis à 1, le bit `Z` à 0.

3.1.5 Les instructions de décalage et rotation

Nous décrivons ici des opérations classiques des langages d'assemblage qui se nomment *décalages* et *rotations*. On les rencontre couramment car leur utilisation simplifie grandement certains traitements.

`RCL`
`ROL`
`RCR`
`ROR` **Les rotations en assembleur**

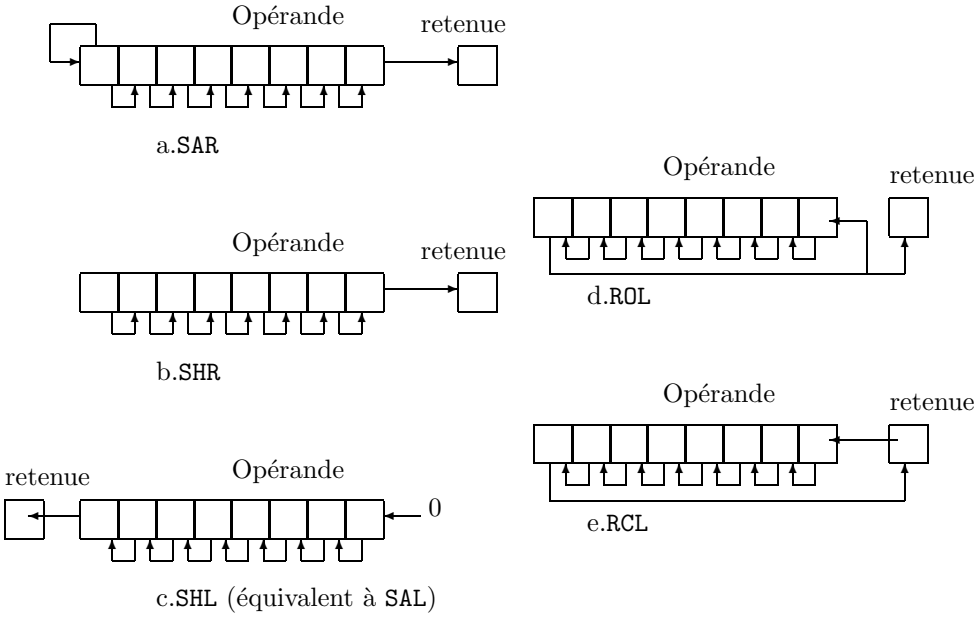


FIG. 3.1 – *Décalages et rotations en assembleur*

Les rotations sont des opérations logiques binaires fréquemment utilisées. Elles considèrent un opérande (octet ou mot) comme un tore dont elles décalent les bits. Lors du décalage, un bit déborde d'un côté, à gauche ou à droite, selon le sens de la rotation. Selon le cas, quelques détails diffèrent ;

`RCL` le bit de poids fort est mis dans l'indicateur de retenue `C`, la valeur de cet indicateur étant préalablement mise dans le bit de poids faible (cf. figure 3.1.e)

`ROL` le bit de poids fort est mis dans l'indicateur de retenue `C` et dans le bit de poids faible de l'opérande. L'ancienne valeur de l'indicateur de retenue n'est pas utilisée (cf. figure 3.1.d)

æ

Les opérandes des instructions `RCL`, `RCR`, `ROL` et `ROR`. étant les mêmes, nous n'en présentons qu'une, l'instruction `RCL`.

```
RCL  registre/variable, 1
RCL  registre/variable, CL
```

O	D	I	T	S	Z	A	P	C
*								*

`RCL` effectue une rotation à gauche de l'opérande destination indiqué, 1 ou `CL` fois, en prenant en compte le contenu de l'indicateur de retenue. Le bit de poids fort de l'opérande destination est mis dans la retenue. Le contenu de la retenue est mis dans le bit de poids faible de l'opérande destination.

`RCR` effectue une rotation à droite de l'opérande destination indiqué, 1 ou `CL` fois, en prenant en compte le contenu de l'indicateur de retenue. Le bit de poids faible de l'opérande destination est mis dans la retenue. Le contenu de la retenue est mis dans le bit de poids fort de l'opérande destination.

`ROL` effectue une rotation à gauche de l'opérande destination indiqué, 1 ou `CL` fois, sans prendre en compte le contenu de l'indicateur de retenue. Le bit de poids fort est mis dans l'indicateur de retenue lors de la rotation ainsi que dans le bit de poids faible de l'opérande.

`ROR` effectue une rotation à droite de l'opérande destination indiqué, 1 ou `CL` fois, sans prendre en compte le contenu de l'indicateur de retenue. Le bit de poids faible est mis dans l'indicateur de retenue lors de la rotation ainsi que dans le bit de poids fort de l'opérande.

Considérons les deux lignes suivantes :

```
mov   al, 16h
mov   cl, 3
xxx   al, cl
```

où `xxx` est une instruction de rotation. Selon le choix de cette instruction, nous obtenons les résultats suivants ;

xxx	rcl	rcr	rol	ror
al	b0h	85h	b0h	85h
C	0	1	0	1

SAL
SHL
SAR
SHR **Les décalages en assembleur**

Une opération de décalage consiste simplement à décaler tous les bits d'une donnée. Contrairement aux rotations qui considèrent une donnée (un octet ou un mot) comme un tore, un décalage considère la donnée comme une file ; ainsi, le bit qui « déborde » de la retenue est perdu.

Les opérandes des instructions `SAL`, `SAR`, `SHL` et `SHR` étant les mêmes, nous ne présentons qu'une seule instruction, `SAL`.

```
SAL  registre/variable, 1
SAL  registre/variable, CL
```

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

`SAL` et `SHL` sont des synonymes et peuvent être utilisées l'une pour l'autre indifféremment. `SAL` effectue un décalage vers la gauche, sauvegardant le bit de poids fort dans l'indicateur de retenue et mettant un 0 dans le bit de poids faible (cf. figure 3.1.c).

`SHR` effectue un décalage vers la droite. Le bit de poids faible est mis dans la retenue. Un 0 est mis dans le bit de poids fort de la donnée (cf. figure 3.1.b).

`SAR` effectue un décalage vers la droite, sauvegardant le bit de poids faible dans l'indicateur de retenue. Par ailleurs (et c'est là toute la différence avec l'instruction précédente), le bit de poids fort est conservé (cf. figure 3.1.a). Le bit de poids fort de la donnée initiale est donc dupliqué.

Considérons les deux lignes suivantes :

```
mov   al, 16h
mov   cl, 3
sal   al, cl
```

Le registre AL contient ensuite la valeur b0h. En effet, 16h s'écrit 00010110 en binaire. Si on décale cette valeur de trois positions vers la gauche, on obtient 10110000, soit b0h Le bit C est mis à 0.

Considérons les deux lignes suivantes :

```
mov   al, 36h
mov   cl, 3
sar   al, cl
```

Le registre AL contient ensuite la valeur 05h. Le bit C est mis à 1.

La différence entre les instructions SAR et SAL n'apparait que si le bit de poids fort de la donnée vaut 1. Le tableau suivant donne, pour deux valeurs de la donnée (mise dans le registre AL), l'effet des différents décalages.

al	f0	70
sar al, 1	f8	38
shr al, 1	78	38

Il faut noter que pour un nombre, un décalage d'une position vers la gauche correspond à une multiplication de ce nombre par 2 et qu'un décalage d'une position vers la droite correspond à une division entière par 2. En généralisant, un décalage de l positions vers la gauche correspond à une multiplication par 2^l et un décalage de l positions vers la droite correspond à une division par 2^l. Il faut bien noter, et c'est ce qui justifie l'existence des deux instructions sar et shr et leur subtile différence, que sar effectue une division sur un nombre en représentation signée tandis que shr effectue une division sur un nombre en représentation non signée.

Bien entendu, les opérations de décalage étant connues comme des opérations logiques binaires et non des opérations arithmétiques, on prendra garde à l'interprétation des indicateurs de débordement ou de retenue que l'on fera à l'issue d'une instruction de décalage.

3.1.6 Les instructions de transfert

MOV transfert d'une valeur

Ces instructions réalisent des transferts de données entre 2 adresses mémoire, 2 registres, ou entre un registre et la mémoire. Ceci correspond donc à l'affectation des langages de haut niveau A:= B.

```
MOV  registre/variable, registre
MOV  registre, registre/variable
MOV  registre, registre de segment
MOV  registre de segment, registre
MOV  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C

registre de segment indique l'un des registres cs, ds, es ou ss.

L'instruction MOV effectue le transfert d'une donnée vers un registre ou une adresse mémoire.

Transfert de la valeur d'un registre de segment vers un autre registre de segment

Aucun mode d'adressage de la commande MOV ne permet de transférer la valeur d'un registre de segment dans un autre registre de segment. Nous disposons uniquement de la possibilité de transférer la valeur d'un registre de segment vers un registre de données et de la possibilité de transférer la valeur d'un registre de données vers un registre de segment. Aussi, pour atteindre notre but, nous devrons effectuer le transfert en deux étapes, en utilisant un registre de données intermédiaire :

– transférer la valeur du registre de segment source dans un registre de données sur 16 bits

– transférer la valeur de ce registre de données dans le registre de segment destination

Par exemple, supposons que nous voulions transférer le contenu du registre `DS` dans le registre `ES`. Nous pouvons l'écrire sous la forme suivante :

```
mov  ax, ds
mov  es, ax
```

Remarquons qu'au cours de cette opération, la valeur du registre `AX` a été perdue lors de l'exécution de la première instruction. Ceci peut poser un problème qui pourra être résolu en utilisant les techniques présentées à la section 4.3.2.

`LEA` Chargement de l'offset d'une donnée

```
LEA  registre/variable, étiquette
```

O	D	I	T	S	Z	A	P	C

Ctte instruction charge l'offset de la donnée référencée dans le deuxième opérande qui est, en général, une donnée déclarée dans le segment de données.

`XCHG` Échange de valeurs

```
XCHG  registre/variable, registre
XCHG  registre, registre/variable
```

O	D	I	T	S	Z	A	P	C

L'instruction `XCHG` échange le contenu de deux emplacements mémoire ou registres. Supposons que le registre `AX` contienne la valeur 20 et le registre `DX` la valeur 10, l'exécution de l'instruction :

```
xchg  ax, dx
```

entraînera : `AX` contient la valeur 10 et `DX` la valeur 20.

3.2 Les tests en assembleur

En assembleur, il n'existe pas de tests comme dans les langages de haut niveau tel Pascal. Cependant, il est bien entendu possible de réaliser des tests. On utilise pour cela les bits du registre `FLAGS` comme condition de test et une instruction de branchement conditionnelle (saut si certains bits du registre `FLAGS` valent 0 ou 1) pour déclencher la partie `alors` ou la partie `sinon` du test.

3.2.1 Principe général

Les bits du registre `FLAGS` sont positionnés par les instructions que nous avons déjà vues (instructions arithmétiques, logiques, ...). Ils sont également positionnés par des instructions spécialement conçues pour réaliser des tests et qui n'ont d'autres effets que de positionner les bits de `FLAGS` en fonction de certaines conditions sur leurs opérandes. Ces instructions sont `CMP` et `TEST`.

Une fois les indicateurs positionnés, une instruction dite de *saut conditionnel* teste un bit ou une combinaison de bits de `FLAGS` et, en fonction du résultat :

- effectue une rupture de séquence (un *saut*) vers un endroit précis dans le code où l'exécution se poursuit normalement.
- continue en séquence si le test ne donne pas un résultat positif.

Nous présentons l'instruction `CMP`, les instructions de sauts conditionnels et inconditionnels puis présentons sur des exemples le codage des tests.

3.2.2 Les instructions de comparaison

`CMP` **comparaison**

C'est l'instruction la plus utilisée pour positionner les indicateurs avant d'effectuer une instruction de saut conditionnel.

```
CMP  registre/variable, registre
CMP  registre, registre/variable
CMP  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

`CMP` permet de comparer deux valeurs. Pour cela `CMP` soustrait le second opérande du premier, sans cependant modifier l'opérande destination, mais en positionnant les indicateurs en fonction du résultat. Ainsi, si le résultat de la soustraction est nul, donc l'indicateur `Z` a été positionné à 1, cela signifie que les deux valeurs comparées sont égales. En utilisant des raisonnements du même genre, on peut savoir si les deux valeurs sont différentes, ordonnées strictement ou non.

À l'issue de l'exécution des deux instructions suivantes :

```
mov  al, 23
cmp  al, 34
```

le registre `AL` n'est pas modifié par l'exécution de l'instruction `CMP` et contient toujours la valeur affectée auparavant 23. L'indicateur de retenue `C` est positionné à 1, ce qui indique que le deuxième opérande de l'instruction `CMP` est supérieur à la valeur du premier opérande. L'indicateur de zéro `Z` valant 0, cela indique que les deux données sont différentes (sinon, la soustraction d'un nombre à lui-même donne 0, donc le bit `Z` est positionné à 1). Le bit de signe `S` est également mis à 1 car $29 - 34$ est un nombre négatif.

`Jxx` **Les instructions de saut conditionnel**

Toutes les instructions de saut conditionnel prennent le même type d'opérande (attention, bien distinguer la valeur zéro 0 du bit `O` du registre d'état) :

`JA`	`étiquette`	saut si supérieur (`C` =0 et `Z` =0)
`JAE`	`étiquette`	saut si supérieur ou égal (`C` =0)
`JB`	`étiquette`	saut si inférieur (`C` =1)
`JBE`	`étiquette`	saut si inférieur ou égal (`C` =1 ou `Z` =1)
`JC`	`étiquette`	saut si retenue (`C` =1)
`JCXZ`	`étiquette`	saut si `CX` vaut 0
`JE`	`étiquette`	saut si égal (`Z` =1)
`JG`	`étiquette`	saut si supérieur (`Z` =0 ou `S` =`O`)
`JGE`	`étiquette`	saut si supérieur ou égal (`S` =`O`)
`JL`	`étiquette`	saut si inférieur (`S` ≠`O`)
`JLE`	`étiquette`	saut si inférieur ou égal (`Z` =1 ou `S` ≠`O`)
`JNC`	`étiquette`	saut si pas de retenue (`C` =0)
`JNE`	`étiquette`	saut si non égal (`Z` =0)
`JNO`	`étiquette`	saut si pas de débordement (`O` =0)
`JNP`	`étiquette`	saut si pas de parité (`P` =0)
`JNS`	`étiquette`	saut si pas de signe (`S` =0)
`JO`	`étiquette`	saut si débordement (`O` =1)
`JP`	`étiquette`	saut si parité (`P` =1)
`JS`	`étiquette`	saut si signe (`S` =1)

O	D	I	T	S	Z	A	P	C

Toutes ces instructions fonctionnent selon le schéma suivant. Quand la condition est vraie, un saut est effectué à l'instruction située à l'étiquette spécifiée en opérande.

Notons que les tests d'ordre (inférieur, supérieur, ...) se comprennent de la manière suivante. Supposons que nous comparions deux données par une instruction `CMP` et que nous exécutions ensuite une instruction `JG`, c'est-à-dire « saut si plus grand ». Il y aura alors saut si la valeur du deuxième opérande de l'instruction `CMP` est supérieur à la valeur du premier opérande.

L'étiquette référencée dans l'instruction de saut conditionnel ne doit pas se trouver trop loin de l'instruction de saut. Sinon, une erreur d'assemblage est déclenchée et le message :

`Relative jump out of range by xxx bytes`

est affiché. Pour remédier à cette erreur, il faut procéder de la manière suivante. Supposons que l'instruction :

```
    je     etiquette
```

provoque l'erreur en question. Il faut transformer cette instruction dans la séquence d'instructions :

```
    jne    nouvelle-etiquette
    jmp    etiquette
nouvelle-etiquette:
```

C'est-à-dire :

1. remplacer l'instruction de saut conditionnel par l'instruction de saut correspondant à la condition opposée (ici, saut si le bit `Z` vaut 1 est remplacé par saut si le bit `Z` vaut 0). L'opérande de cette nouvelle instruction est une nouvelle étiquette.
2. faire suivre l'instruction de saut conditionnel par une instruction de saut inconditionnel `jmp` (voir 3.2.2) dont l'opérande est l'étiquette de l'instruction de saut conditionnel originale
3. déclarer ensuite la nouvelle étiquette après l'instruction `jmp`

JMP : saut inconditionnel

JMP étiquette

O	D	I	T	S	Z	A	P	C

L'instruction JMP effectue un saut inconditionnel à l'étiquette spécifiée. Contrairement à un saut conditionnel, le saut est toujours effectué, le registre FLAGS n'intervenant en rien dans cette opération.

3.2.3 Exemples de codage de tests en assembleur

Généralement parlant, un test dans un langage évolué a la forme suivante :

```
SI (condition vraie) ALORS
    action-alors
SINON
    action-sinon
FIN_SI
```

Si la valeur de l'expression condition est vraie, les instructions composant la partie action-alors sont exécutées. Sinon, les instructions de la partie action-sinon sont exécutées.

En assembleur, ce type de construction est réalisé à l'aide du squelette de programme suivant :

```
     calcul de la condition
     Jcc   SINONn
     action-alors
     ...
     JMP   FSIn
SINONn:
     action-sinon
     ...
FSIn:
     ...
```

où Jcc dénote l'une des instructions de saut conditionnel vues plus haut.

Nous notons les points suivants :

- le calcul de la condition positionne les indicateurs du registre FLAGS
- en fonction de ce positionnement, un saut conditionnel est réalisé par l'instruction Jcc (où cc représente la condition à tester – voir les différentes possibilités dans la section traitant des instructions de saut conditionnel) vers la partie sinon du test. Si la condition est vérifiée, aucun saut n'est réalisé et le programme poursuit son exécution en séquence avec les instructions de la partie action-alors
- une fois les instructions de la partie alors exécutées, un saut doit être réalisé (instruction JMP FSIn) pour atteindre la fin de la structure de test afin de ne pas également exécuter la partie sinon
- une étiquette (SINONn) est déclarée indiquant le début de la partie action-sinon où le branchement conditionnel doit être exécuté pour déclencher l'exécution des instructions de la partie sinon du test
- une fois la partie sinon exécutée, on continue en séquence avec les instructions situées après le test (c'est-à-dire, après l'étiquette FSIn)

Pour des raisons de clarté, nous utiliserons un format spécial pour les étiquettes codant un test. Une étiquette `SINON` indique le début de la séquence `action-sinon` et une étiquette `FSI` indique la fin de la structure de test. Ces étiquettes seront numérotées, les tests étant numérotés dans le programme de 1 à n, par convention.

Afin d'illustrer le propos, nous donnons un exemple de codage d'un test. Nous voulons effectuer la division d'un nombre par un autre. La division n'est possible que si le diviseur est non nul. Aussi, nous devons tester si l'opérande diviseur est nul avant d'exécuter la division. Si le diviseur est nul, nous désirons que le résultat du traitement soit -1. Sinon, le résultat est le quotient des deux nombres.

```
        .data
val     db      ....            ; mettre ici la valeur du diviseur
        .code
        ...
        mov     ax, ...         ; valeur du dividende
        cmp     val, 0
        je      SINON1          ; diviseur nul => saut
        mov     bl, [val]
        div     bl
        jmp     FSI1
SINON1: mov     ax, -1          ; diviseur nul => resultat vaut -1
FSI1:   ...
        ...
```

Notons que ce programme n'est pas complet. Il vise simplement à montrer le codage d'un test.

3.3 Les boucles en assembleur

Comme pour les tests, il n'existe pas de structure générale en assembleur pour coder une boucle. Cependant, à l'aide des instructions vues précédemment pour réaliser des tests, on peut coder n'importe quel type de boucle. On verra également que des instructions simplifient grandement le codage des boucles `pour`.

3.3.1 Principe général

Le principe de la réalisation des boucles `tant-que` ou `jusqu'a` est assez particulier. Une boucle est essentiellement composée de deux éléments :

- une condition de sortie qui indique quand la boucle doit être interrompue, qu'il faut sortir de la boucle et continuer l'exécution des instructions en séquence
- un corps de boucle qui spécifie l'action à réaliser pendant que la condition de sortie n'est pas vérifiée, à chaque itération de la boucle

La condition de sortie va donc être codée d'une manière ressemblant aux tests. Un saut conditionnel testera cette condition et entraînera, quand la condition est vérifiée, la sortie de la boucle.

Le corps de la boucle devra pour sa part « boucler », c'est-à-dire, une fois exécuté, entraîner le re-calcul de la condition de sortie et la prise de décision quant à la poursuite ou non des itérations.

Nous indiquons maintenant comment est réalisé le codage d'une boucle `tant-que` et d'une boucle `répéter`.

3.3.2 Boucles `tant-que`

Le squelette d'une boucle `tant-que`, dans un langage de haut niveau, est le suivant :

```
TANT-QUE (condition) FAIRE
    action
FIN_TQ
```

Cela va se traduire en assembleur sous la forme suivante :

```
TQn:  calcul de la condition
      Jcc   FTQn
      action
      ...
      JMP   TQn
FTQn:
      ...
```

où `Jcc` dénote l'une des instructions de saut conditionnel vues plus haut.

Nous notons les points suivants :

- une étiquette `TQ` indique le début de la boucle
- la boucle commence par une évaluation de la condition de sortie qui positionne les indicateurs du registre `FLAGS`
- en fonction de la valeur des indicateurs (donc du résultat du test de sortie), un saut conditionnel est effectué en fin de boucle (`Jcc FTQn`), pour quitter la boucle le moment venu
- on trouve ensuite le corps de la boucle, terminé par une instruction de saut incondtionnel vers le début de la boucle (`JMP TQn`) qui permettra de ré-évaluer la condition d'arrêt après chaque itération
- une étiquette `FTQ` indiquant la fin de la boucle

Pour des raisons de clarté, nous utiliserons un format spécial pour les étiquettes codant une boucle `TANT-QUE`. Une étiquette `TQ` indique le début de la structure de boucle et une étiquette `FTQ` indique la fin de la structure de boucle. Ces étiquettes seront numérotées, les boucles étant numérotées dans le programme de 1 à n, par convention.

3.3.3 Boucles `répéter`

Le squelette d'une boucle `REPETER` dans un langage de haut niveau est le suivant :

```
REPETER
    action
JUSQUA (condition vraie)
```

Cela se traduit de la manière suivante en assembleur :

```
REPETERn:
     action
     ...
     calcul de la condition
     Jcc   REPETERn
```

Notons les points suivants :

- une étiquette `REPETER` repère le début de la boucle
- on trouve ensuite le corps de la boucle
- à l'issue de l'exécution du corps, on trouve l'évaluation du test d'arrêt qui positionne les indicateurs du registre `FLAGS`
- une instruction de saut conditionnel effectue un branchement pour continuer les itérations si la condition d'arrêt n'est pas vérifiée

Pour des raisons de clarté, nous utiliserons un format spécial pour les étiquettes codant une boucle `REPETER`. Une étiquette `REPETER` indique le début de la boucle. Ces étiquettes seront numérotées, les boucles `REPETER` étant numérotées dans le programme de 1 à n, par convention.

3.3.4 Les boucles pour

Une boucle `pour` est généralement codée à l'aide d'une instruction de la famille `LOOP`. Pour les boucles traitant des chaînes de données (caractères, mots), voir la section 3.3.6.

`LOOP` **boucles contrôlées par un compteur**

`LOOP`	`étiquette`	saut court si le compteur est différent de 0
`LOOPE`	`étiquette`	saut court si compteur différent de 0 et `Z` =1
`LOOPNE`	`étiquette`	saut court si compteur différent de 0 et `Z` =0

O	D	I	T	S	Z	A	P	C

`LOOP` fonctionne avec le registre `CX` qui joue le rôle de compteur de boucles. `LOOP` décrémente le compteur sans modifier aucun des indicateurs. Si le compteur est différent de 0, un saut à l'étiquette opérande de l'instruction `LOOP` est réalisé. Le squelette d'une boucle `pour` s'écrit de la manière suivante :

```
POUR indice := 1 A bs FAIRE
    action
FAIT
```

Cependant, en assembleur, seules existent des boucles ayant un indice variant d'une certaine valeur `bs` à 1, en décrémentant sa valeur à chaque itération. Aussi, la boucle précédente doit initialement être transformée dans une boucle du genre [1] :

```
POUR indice := bs A 1, pas := -1 FAIRE
    action
FAIT
```

Cette boucle se traduit en assembleur de la manière suivante :

```
       mov   cx, bs
POURn: ...
       action
       loop  POURn
```

En exemple, nous écrivons un programme qui affiche dix fois la chaîne de caractères 'hello world'.

```
        .data
msg:    db    'hello world', 13, 10, '$'
        .code
        mov   ax, @data   ; initialisation de la valeur
        mov   ds, ax      ;    de ds
        mov   cx, 10      ; initialisation du compteur de boucles
        ;
        ; corps de boucle
        ;
boucle: mov   ah, 9       ; affichage du message
        lea   dx, msg
        int   21h
        loop  boucle      : contrôle de la boucle
        ...
```

1. on prendra garde en effectuant cette transformation. Voir à ce propos le document « méthodologie de programmation en assembleur »

Le début initialise le registre DS pour qu'il contienne le numéro du segment de données et le compteur de boucles. Le corps de la boucle contient alors un appel à la routine affichant un message à l'écran (cf. section 5.1.2).

Il est très courant de parcourir un tableau dans une boucle tout en voulant sortir de la boucle lorsqu'une condition est vérifiée (parce que l'on cherche une valeur dans le tableau possédant une propriété particulière par exemple). Notons que ce type de construction n'existe pas en Pascal. Des instructions sont disponibles en assembleur pour réaliser ce genre de boucle, les instructions LOOPE et LOOPNE (les instructions LOOPZ et LOOPNZ étant des synonymes). Comme pour les boucles LOOP, le registre CX joue le rôle de compteur de boucle. Par ailleurs, à chaque itération, une condition (la valeur du bit Z du registre FLAGS) est testée. Pour l'instruction LOOPE, il y a saut à l'étiquette spécifiée en opérande tant que le compteur est non nul et que l'indicateur Z vaut 1. Pour l'instruction LOOPNE, il y a saut à l'étiquette spécifiée en opérande tant que le compteur est non nul et que l'indicateur Z vaut 0.

Exemple

Nous présentons un exemple de boucle ainsi que les conventions que nous adoptons pour le codage d'une boucle pour. Nous recherchons dans un tableau d'entiers l'élément d'indice le plus faible qui soit multiple de 7.

```
          .data
tableau   db      34, 72, 48, 32, 56, 12, 8, 9, 45, 63, 80
          .code
          mov     ax, @data
          mov     ds, ax
          mov     cx, 11  ; nombre d'éléments dans la table
          mov     dl, 7   ; valeur du diviseur
          mov     bx, 0   ; indice de l'élément considéré
BOUCLE1:
          mov     ah, 0
          mov     al, BYTE PTR table+bx
          div     dl      ; division de l'élément courant par 7
          inc     bx      ; mise à jour de l'indice
          cmp     ah, 0   ; ah contient le reste de la division
          loopne  BOUCLE1 ; saut si non multiple et non terminé
```

À l'issue de l'exécution de ces instructions, le registre BX contient l'indice du premier élément du tableau table ayant une valeur multiple de 7.

On notera les points suivants :

- une étiquette BOUCLE1 indique le début de la boucle ;
- on trouve ensuite le corps de la boucle ;
- une instruction (ici CMP) positionne les indicateurs du registre FLAGS pour éventuellement sortir de la boucle avant d'avoir effectué toutes les itérations (quand une valeur multiple de 7 a été trouvée, on sort immédiatement) ;
- l'instruction LOOPNE décide s'il faut boucler en fonction de la position des indicateurs du registre FLAGS et de la valeur du compteur de boucle (le registre CX) qui est automatiquement décrémenté.

Comme pour les autres structures, on notera que les étiquettes BOUCLE sont numérotées de 1 à n.

3.3.5 Instructions diverses

Nous décrivons dans cette section quelques instructions qui se révèlent souvent utiles pour le codage des tests ou des boucles.

CLC

CLD
CMC
STC
STD **Armement des indicateurs**

Nous décrivons ici quelques instructions ne prenant aucun opérande et agissant chacune sur un bit particulier du registre d'indicateurs `FLAGS`.

`CLC` mise à zéro de l'indicateur de retenue `C`

`CLD` mise à zéro de l'indicateur de direction `D`

`CMC` complémente l'indicateur de retenue `C`

`STC` mise à 1 de l'indicateur de retenue `C`

`STD` mise à 1 de l'indicateur de direction `D`

3.3.6 Le traitement des chaînes de données

Nous présentons dans cette section des instructions spécialement conçues pour le traitement de séquences de données, que ces données soient des caractères (des octets) ou des mots. Chacune effectue le traitement d'un élément de chaîne. Il y a cinq types d'instruction :

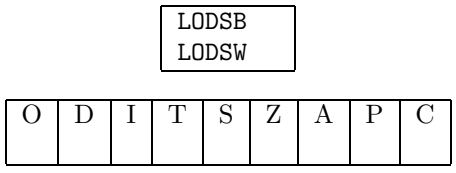
- `LODS` : chargement d'un élément de chaîne depuis la mémoire dans un registre
- `STOS` : écriture d'un élément de chaîne en mémoire
- `MOVS` : transfert d'un élément entre deux chaînes
- `CMPS` : comparaison entre deux éléments de chaîne
- `SCAS` : comparaison entre une valeur et un élément de chaîne

Toutes ces instructions sont conçues pour être aisément insérées dans une boucle qui traitera l'ensemble des éléments de la chaîne. Classiquement, lors d'une itération, un élément d'une chaîne est référencé par son index dans la chaîne. Pour pouvoir itérer l'instruction, il faut donc incrémenté cet index après chaque traitement. Pour nous aider, les instructions de traitement de chaînes incrémentent automatiquement l'index. Par ailleurs, ces instructions traitent des chaînes d'octets, ou de mots. Aussi, selon le type de données traitées, 1 ou 2 doit être ajouté à la valeur de l'index. C'est effectivement de cette manière qu'agissent ces instructions.

Notons également que pour traiter les chaînes de leur fin vers leur début plutôt que dans le sens contraire, il suffit de soustraire 1 ou 2 à la valeur de l'index. Pour cela, l'indicateur `D` du registre `FLAGS` indique la direction du traitement. Si `D` vaut 0, le traitement va du début vers la fin de la chaîne et la valeur 1 ou 2 est ajoutée à l'index. Si `D` vaut 1, le traitement va dans le sens contraire et la valeur 1 ou 2 est soustraite à l'index. Notons que le bit `D` est positionné par les instructions `STD` et `CLD`. Remarque importante : les indicateurs de `FLAGS` ne sont pas modifiés par les opérations modifiant la valeur des index.

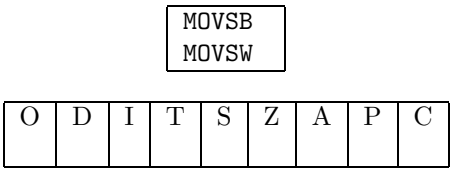
Dans les instructions de traitement de chaînes de caractères, le registre `SI` pointe implicitement vers la chaîne source, le registre `DI` vers la chaîne destination.

LODS **chargement d'un élément de chaîne**



LODSB charge le registre AL (respectivement, AX pour l'instruction LODSW) avec l'octet (respectivement le mot) pointé par le registre SI. Une fois le chargement effectué, il est automatiquement ajouté 1 (respectivement 2) à SI si l'indicateur D vaut 0, ou retiré 1 (respectivement 2) si l'indicateur D vaut 1.

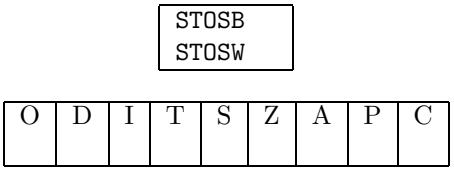
MOVS transfert d'un élément entre deux chaînes



MOVSB copie l'octet (respectivement le mot pour l'instruction MOVSW) pointé par SI vers l'octet (respectivement le mot) pointé par ES:DI. Le segment destination est forcément ES. Par contre, le segment de la source qui est DS par défaut peut être redéfini.

Une fois la copie réalisée, SI et DI sont automatiquement modifiés. Si l'indicateur de direction vaut 0, 1 (respectivement 2) est ajouté à SI et DI. Si l'indicateur de direction vaut 1, 1 (respectivement 2) est retiré à SI et DI.

STOS écriture en mémoire d'un élément d'une chaîne



STOSB écrit en mémoire l'octet (respectivement le mot pour l'instruction STOSW) se trouvant dans le registre AL (respectivement dans le registre AX) en mémoire à l'adresse pointée par le registre DI. Le segment destination est forcément ES.

Une fois la copie réalisée, DI est automatiquement modifié. Si l'indicateur de direction vaut 0, 1 (respectivement 2) est ajouté à DI. Si l'indicateur de direction vaut 1, 1 (respectivement 2) est retiré à DI.

Nous donnons un premier exemple qui consiste à multiplier par 2 tous les éléments d'un tableau d'entiers

```
        .data
TABLEAU DW      10, 20, 30, 40, 50, 60, 70, 80, 90, 100
RES     DW      10 DUP (0)
        .code
        mov     ax, @data
        mov     ds, ax
        mov     es, ax
        lea     si, TABLEAU
        lea     di, RES
        mov     cx, 10
BOUCLE1: lodsw
        mov     bl, 2
        mul     bl
        stosw
        loop    BOUCLE1
        ...
```

Un deuxième exemple illustre la transformation d'une table de données codées en décimal non compacté en ASCII. Cette opération est simplement obtenue par ou-logique avec la valeur `30h` (le code ASCII des chiffres 0, 1, ... 9 est `30h`, `31h`, ... `39h`).

```
          .data
TABLEAU   DW      1, 2, 3, 4, 5, 6, 7, 8, 9
RES       DW      9 DUP (0)
          .code
          mov     ax, @data
          mov     ds, ax
          mov     es, ax
          lea     si, TABLEAU
          lea     di, RES
          mov     cx, 9
BOUCLE1:  lodsw
          or      al, 30h
          stosw
          loop    BOUCLE1
          ...
```

`CMPS` **comparaison de chaînes**

CMPSB CMPSW

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

`CMPSB` compare l'octet (respectivement le mot pour l'instruction `CMPSW`) pointé par `SI` à l'octet (respectivement le mot) pointé par `ES:DI`. Le segment destination est forcément `ES`. Par contre, le segment de la source qui est `DS` par défaut peut être redéfini.

Une fois la copie réalisée, `SI` et `DI` sont automatiquement modifiés. Si l'indicateur de direction vaut 0, 1 (respectivement 2) est ajouté à `SI` et `DI`. Si l'indicateur de direction vaut 1, 1 (respectivement 2) est retiré à `SI` et `DI`.

`SCAS` **comparaison d'un élément de chaîne avec une valeur**

SCASB SCASW

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

`SCASB` compare la valeur de l'octet (respectivement le mot pour l'instruction `SCASW`) contenu dans le registre `AL` (respectivement `AX`) à l'octet (respectivement le mot) pointé par `ES:DI`. La donnée en mémoire est toujours référencée via le registre de segment `ES`.

Une fois la comparaison réalisée, `DI` est automatiquement modifié. Si l'indicateur de direction vaut 0, 1 (respectivement 2) est ajouté à `DI`. Si l'indicateur de direction vaut 1, 1 (respectivement 2) est retiré à `DI`.

`REP`
`REPE`
`REPZ`
`REPNE`
`REPNZ` **répéter une instruction de traitement de chaînes de données**

Nous décrivons ici une instruction qui permet d'itérer sur toute une chaîne les opérations élémentaires vues précédemment. Cette instruction préfixe en fait l'opération à itérer. Elle donne tout son intérêt aux instructions `LODS`, `STOS`, `MOVS`, `CMPS`, `SCAS`.

`REP` ne peut préfixer que les instructions `LODS`, `STOS`, `MOVS`. Un compteur (registre `CX`) indique le nombre d'itérations à effectuer.

Les préfixes `REPE` et `REPNE` ne peuvent préfixer que les instructions `CMPS` et `SCAS`. Pour les préfixes `REPE` et `REPNE`, un compteur (registre `CX`) indique le nombre maximal d'itérations à effectuer. Par ailleurs, l'instruction élémentaire est répétée tant qu'une condition (égalité ou non-égalité entre les opérandes de `CMPS` ou `SCAS`) n'est pas remplie.

```
REP   MOVSB
REP   MOVSW
REP   LODSB
REP   LODSW
REP   STOSB
REP   STOSW
REPE  CMPSB
REPE  CMPSW
REPE  SCASB
REPE  SCASW
REPNE CMPSB
REPNE CMPSW
REPNE SCASB
REPNE SCASW
```

O	D	I	T	S	Z	A	P	C
					*			

- `REP MOVS` copie `CX` éléments de la chaîne pointée par `SI` vers la chaîne pointée par `ES:DI`
- `REP LODS` charge `CX` éléments de la chaîne pointée par `SI` dans `AL` (ou `AX`)
- `REP STOS` écrit `CX` éléments de la chaîne pointée par `ES:DI` avec la valeur contenue dans `AL` (ou `AX`)
- `REPE CMPS` compare au plus `CX` éléments de la chaîne pointée par `ES:DI` avec ceux de la chaîne pointée par `SI`. Les itérations sont poursuivies tant que les éléments des deux chaînes sont égaux et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction `REPE CMPS` est terminée
- `REPE SCAS` compare au plus `CX` éléments de la chaîne pointée par `ES:DI` avec la valeur du registre `AL`, ou `AX` selon le cas. Les itérations sont poursuivies tant que les éléments de la chaîne sont égaux à la valeur du registre et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction `REPE SCAS` est terminée
- `REPNE CMPS` compare au plus `CX` éléments de la chaîne pointée par `ES:DI` avec ceux de la chaîne pointée par `SI`. Les itérations sont poursuivies tant que les éléments des deux chaînes sont différents et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction `REPNE CMPS` est terminée
- `REPNE SCAS` compare au plus `CX` éléments de la chaîne pointée par `ES:DI` avec la valeur du registre `AL`, ou `AX`, selon le cas. Les itérations sont poursuivies tant que les éléments de la chaîne sont différents de la valeur du registre et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction `REPNE SCAS` est terminée

Notons que `REPZ` est un synonyme de `REPE` ; `REPNZ` est un synonyme de `REPNE`.

Pour résumer, à chaque itération, les actions suivantes sont exécutées :

1. tester le compteur (`CX`). S'il est à zéro, sortir de l'instruction de répétition et déclencher l'exécution de la suivante. Sinon :

2. exécuter l'opération sur la chaîne
3. décrémenter le compteur
4. si l'opération est `SCAS` ou `CMPS`, tester l'indicateur de zéro. Si la condition d'itération n'est pas remplie, poursuivre avec l'instruction qui suit le `REP`. Sinon :
5. recommencer

Exemple 1

Copie d'un tableau de caractères dans un autre tableau.

```
        .data
CHAINE  db      'hello world'     ; chaîne source
RES     db      11 dup (?)        ; chaîne cible
        .code
        mov     ax, @data
        mov     ds, ax
        mov     es, ax
        lea     di, RES           ; offset chaîne cible
        lea     si, TABLE         ; offset chaîne source
        mov     cx, 11            ; longueur de la chaîne
        rep movsb                 ; copie
        ...
```

Exemple 2

Recherche d'un caractère dans une chaîne de caractères en utilisant l'instruction `SCAS`

```
        .data
CHAINE  db      'hello world'
        .code
        mov     ax, @data
        mov     es, ax
        mov     al, 'w'
        lea     di, CHAINE
        mov     cx, 11
        repne scasb
        ...
```

À l'issue de l'exécution de cette séquence d'instructions, le registre `DI` pointe sur le caractère qui se trouve après le 'w' dans la chaîne de caractère, c'est-à-dire 'o'. Puisque l'offset de `CHAINE` vaut 0, `di` vaut 7 à l'issue de l'exécution de `repne scasb`.

Exemple 3

Recherche d'une chaîne de caractères dans une autre. À l'issue de cette recherche, le registre `AX` indiquera le résultat et vaudra 1 si la chaîne est trouvée, 0 sinon.

```
        .data
CH1     db      'hello world'
CH2     db      'rl'
        .code
        mov     ax, @data
        mov     ds, ax
        mov     es, ax
```

```
        lea     si, CH1
        mov     ax, 1
        mov     bx, 10
TQ1:    lea     di, CH2
        mov     cx, 2
        repe cmpsb
        je      FTQ1      ; sous-chaîne trouvée
        dec     bx
        jne     TQ1
        mov     ax, 0
FTQ1:   ...
```

3.4 Autres instructions

Nous regroupons ici quelques instructions qui peuvent être utiles.

3.4.1 Instructions arithmétiques

ADC **Addition avec retenue**

```
ADC  registre/variable, registre
ADC  registre, registre/variable
ADC  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

L'instruction `ADC` effectue l'addition du contenu du registre source au registre destination avec report de retenue, soit ;

$$\text{destination} \leftarrow \text{source} + \text{destination} + \text{retenue}$$

La retenue `C` est positionnée en fonction du résultat de l'opération.

Pour expliquer le positionnement de la retenue auxiliaire `A`, supposons que nous additionnions des octets. Le bit `A` indique si une retenue a lieu entre les quatre bits de poids faible et les quatre bits de poids fort. Ainsi, si on additionne les données hexadécimales `45` et `3e`, il y a une retenue lors de l'addition des deux demi-octets de poids faible (`5` + `e` est supérieur à `f`, donc il y a une retenue). Aussi, le bit de retenue auxiliaire est positionné à 1.

Nous proposons un exemple d'addition sur 32 bits. Nous voulons additionner une donnée se trouvant dans les registres `AX` et `DX` à une donnée se trouvant dans les registres `BX` et `CX` et ranger le résultat dans les registres `AX` et `DX` (`AX` et `BX` contiennent les poids faibles, `DX` et `CX` les poids forts). Le code est alors le suivant ;

```
add  ax, bx
adc  dx, cx
```

La première instruction calcule le poids faible du résultat en additionnant simplement les poids faibles des deux opérandes. La deuxième instruction calcule alors le poids fort du résultat en additionnant les poids forts des deux opérandes et l'éventuelle retenue générée par l'addition des poids faibles. Pour tenir compte de cette retenue, on a utilisé l'instruction `ADC`.

SBB **Soustraction avec retenue**

```
SBB  registre/variable, registre
SBB  registre, registre/variable
SBB  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

L'instruction `SBB` effectue la soustraction du contenu du registre source au registre destination avec report de retenue, soit :

destination ← destination - source - retenue

La retenue est positionnée en fonction du résultat de l'opération.

Nous proposons un exemple de soustraction sur 32 bits. Nous voulons soustraire une donnée se trouvant dans les registres `BX` et `CX` d'une donnée se trouvant dans les registres `AX` et `DX` et ranger le résultat dans les registres `AX` et `DX` (`AX` et `BX` contiennent les poids faibles, `DX` et `CX` les poids forts). Le code est alors le suivant ;

```
sub  ax, bx
sbb  dx, cx
```

La première instruction calcule le poids faible du résultat en soustrayant simplement les poids faibles des deux opérandes. La deuxième instruction calcule alors le poids fort du résultat en soustrayant les poids forts des deux opérandes en tenant compte de l'éventuelle retenue générée par la soustraction des poids faibles. Pour tenir compte de cette retenue, on a utilisé l'instruction `SBB`.

`TEST` **test logique**

```
TEST  registre/variable, registre
TEST  registre/variable, constante
```

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Cette instruction effectue un et-logique entre ses deux opérandes. Cependant, contrairement à l'instruction `AND`, elle ne modifie pas l'opérande destination. Elle se contente de positionner les indicateurs du registre `FLAGS`.

Par exemple, l'instruction `test` peut être utile pour tester le signe d'un nombre (voir la description de l'instruction `jz` dans le paragraphe suivant) :

```
.....
test   al, 80h   ; test du bit de signe
jz     negatif
.....
```

3.4.2 Les instructions d'ajustement décimal

Les instructions décrites dans cette section sont utiles pour effectuer des calculs sur des nombres codés en BCD. Elles réalisent un ajustement du contenu du registre `AL` ou `AX` pour la réalisation d'opérations arithmétiques. Elles ne prennent aucun opérande.

Les quatre instructions en question sont les suivantes ;

`AAA` ajustement ASCII de `AL` après addition `ADD`

`AAD` ajustement ASCII de `AX` avant division `DIV`

`AAM` ajustement ASCII de `AX` après multiplication `MUL`

`AAS` ajustement ASCII de `AL` après soustraction `SUB`

Ces quatre instructions positionnent les indicateurs du registre `FLAGS` de la manière suivante :

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

Nous reprenons le rôle de chacune.

`AAA` ajustement après addition

L'instruction `AAA` ne doit être exécutée qu'après avoir effectué une addition `ADD`. Elle transforme alors le contenu du registre `AL` en un nombre décimal non compacté (c'est-à-dire que le poids faible doit être compris entre 0 et 9). Si besoin (valeur de `AL` supérieure à 9), la valeur 1 est ajoutée au contenu de `AH`.

Ainsi, si nous voulons réaliser l'addition de 36 et 45 en représentation décimale non compactée, on peut écrire ;

```
mov   ax, 0306h
add   ax, 0405h
```

qui affectera la valeur `070b` au registre `AX`, ce qui n'est pas une valeur correcte en représentation décimale non compactée. Si on exécute maintenant l'instruction `AAA`, le registre `AX` contiendra ensuite la valeur `0801`. 81 est bien la somme de 36 et 45.

`AAD` ajustement avant division

Cette instruction doit être exécutée avant de réaliser la division de deux opérandes octets en représentation décimale non compactée de manière à ce que la division donne le bon résultat. Pour cela, elle transforme la valeur non compactée contenue dans `AX` en une valeur binaire.

Ainsi, si nous désirons diviser 37 par 5, on peut écrire :

```
mov    dx,5
mov    ax,0307h
aad
div    dl
```

Le registre `ah` reçoit la valeur 2, reste de la division. Le registre `al` reçoit la valeur 5, quotient de la division.

`AAM` ajustement après multiplication

Cette instruction doit être exécutée après une multiplication de deux opérandes octets en représentation décimale non compactée afin d'obtenir le résultat de la multiplication sous forme décimale non compactée également.

Ainsi, si nous désirons multiplier 7 par 5, on peut écrire :

```
mov    dx,5
mov    ax,7
mul    dl
aam
```

À l'issue de l'exécution de cette séquence d'instructions, `ax` contient la valeur `0305h`. On notera qu'avant l'exécution de l'instruction `AAM`, `ax` contient la valeur `23h`, soit 35. Notez bien la différence entre 35 et `0305h`.

`AAS` ajustement après soustraction

Cette instruction doit être exécutée après une soustraction `SUB` de deux opérandes en représentation décimale non compactée afin d'obtenir le résultat de la soustraction en représentation décimale non compactée.

Ainsi, si nous désirons retirer 7 de 6, on peut écrire :

```
mov    al,6
mov    dl,7
sub    al,dl
aas
```

À l'issue de l'exécution de cette séquence d'instructions, `al` contient la valeur 9 (et `ax` contient la valeur `FF09h`). On notera qu'avant l'exécution de l'instruction `AAS`, `ax` contient la valeur `00ffh`.

Instructions d'ajustement décimal en représentation décimale compactée

Les deux instructions suivantes permettent d'effectuer un ajustement pour la réalisation d'additions ou de soustractions en représentation décimale compactée. Aucune instruction n'est disponible pour l'ajustement pour multiplication ou division du fait de la complexité de la tâche et de la rareté de son utilisation.

`DAA` ajustement décimal de `AL` après addition `ADD`

`DAS` ajustement décimal de `AL` après soustraction

Ces deux instructions positionnent les indicateurs du registre `FLAGS` de la manière suivante ;

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

Supposons que nous voulions additionner les nombres décimaux 35 et 26 en représentation décimale compactée. Nous pourrions écrire ;

```
mov  al, 35h
add  al, 26h
daa
```

Notons que pour indiquer à l'assembleur que l'on veut représenter en décimal compacté les nombres 35 et 26, il faut les écrire comme s'ils étaient des constantes hexadécimales. Ainsi, considérant la première instruction, 5 est mis dans les 4 bits de poids faible de `AL`, 3 est mis dans les 4 bits de poids fort de `AL`.

À l'issue de l'exécution de ces trois instructions, `AL` contient la valeur `61h`. 61 est bien le résultat de la somme de 35 et 26.

Pour mieux comprendre le mécanisme, notons qu'avant l'exécution de l'instruction `DAA`, le registre `AL` contient la valeur `5bh`. En effet, le processeur ne sait pas (et n'a pas à savoir) que l'on a codé les nombres sous forme décimale compactée ; c'est simplement au programmeur de le savoir. Aussi, le 8086 exécute simplement dans un premier temps l'addition de deux données hexadécimales `35h` et `26h` puis ajuste ce résultat pour obtenir la représentation décimale compactée (via l'instruction `DAA`).

Prenons un deuxième exemple pour comprendre le fonctionnement du bit de retenue et de débordement. Soient les trois lignes ;

```
mov  al, 63h
add  al, 82h
daa
```

L'addition décimale de 63 et 82 donne 145, donc un nombre qui ne peut être représenté en décimal compacté sur un octet puisqu'il est plus grand que 100. Aussi, comme pour les opérations en binaire, le résultat contient le poids faible du résultat, c'est-à-dire 45, et l'indicateur de retenue `C` ainsi que l'indicateur de débordement sont positionnés à 1 par l'instruction `DAA`.

Notons que c'est lors de l'instruction `DAA` qu'a lieu le débordement de capacité et qu'apparait une retenue. En effet, l'addition donne le résultat hexadécimal `e5h` qui n'est pas trop grand et ne provoque aucun débordement et aucune retenue.

3.4.3 Autres

`LAHF`
`SAHF` **Instructions de transfert entre `FLAGS` et `AH`**

L'instruction `LAHF` copie dans `AH` la valeur du registre `FLAGS`.

L'instruction `SAHF` copie le contenu de `AH` dans le registre `FLAGS`. Seuls les cinq bits `S`, `Z`, `A`, `P` et `C` sont modifiés lors de cette opération quelle que soit la valeur des 3 bits de poids fort de `AH`.

Chapitre 4

Les sous-programmes

4.1 Principe général

Les sous-programmes jouent, en assembleur, le rôle des procédures et des fonctions dans les langages de haut niveau. Elles structurent les programmes en donnant un nom à un traitement et en réduisant la taille des programmes qui utilisent plusieurs fois la même séquence de code. Dans la suite de ce chapitre, on décrit tout d'abord le principe de fonctionnement des sous-programmes en assembleur. Nous décrivons ensuite les mécanismes à mettre en jeu pour écrire des sous-programmes paramétrés ou renvoyant une valeur.

4.2 Synoptique de l'appel d'un sous-programme

Appeler un sous-programme consiste à effectuer une rupture de séquence, poursuivre l'exécution du programme et reprendre ensuite l'exécution là où on a préalablement rompu la séquence. C'est exactement la même notion que dans les langages de haut niveau où l'appel d'une procédure suspend l'exécution de la séquence d'instructions en cours, exécute le corps de la procédure et reprend ensuite l'exécution de la séquence suspendue auparavant. Notons que l'on peut à nouveau appeler une procédure dans une procédure ou qu'une procédure peut s'appeler elle-même.

Dans un langage de haut niveau, on n'a aucun souci à se faire pour que cela fonctionne bien, tout le mécanisme étant pris en charge par le compilateur qui génère la séquence d'instructions en assembleur pour que tout fonctionne comme prévu. En assembleur, on doit comprendre le mécanisme pour pouvoir en tirer le maximum et également pouvoir passer des paramètres aux sous-programmes.

Dans ce qui suit, nous nommerons *appelant* le programme ou le sous-programme qui appelle un sous-programme, et *appelé* le sous-programme qui est appelé par l'appelant.

À la base, deux instructions sont nécessaires :

- une instruction dite *d'appel de sous-programme* qui va rompre la séquence d'instructions en cours d'exécution et donner le contrôle au sous-programme à appeler ;
- une instruction dite de *retour de sous-programme* qui, placée à la fin du sous-programme, indique au processeur qu'il faut reprendre l'exécution du programme interrompu, là où il s'est interrompu, c'est-à-dire exécuter les instructions qui suivent l'instruction d'appel du sous-programme.

4.2.1 L'appel d'un sous-programme

Un sous-programme est une séquence d'instructions. Pour le repérer, on peut utiliser l'adresse de sa première instruction. Pour spécifier dans l'instruction d'appel de sous-programme le sous-programme à appeler, on va donc utiliser son adresse (comme on utilise le nom d'une procédure pour l'appeler dans un langage de haut niveau : en assembleur, le nom d'une procédure, c'est son adresse).

4.2.2 Retour à l'appelant

Un sous-programme peut-être appelé à différents endroits dans un programme. Aussi, si son adresse est toujours la même, l'adresse de retour (là où l'exécution du programme doit reprendre après exécution du sous-programme) n'est pas toujours la même. Aussi, dans l'instruction de retour de sous-programme, il est impossible d'indiquer où il faut continuer l'exécution. Aussi, il est clair que le processeur doit "se rappeler" de l'endroit où à été réalisé l'appel pour pouvoir continuer l'exécution du programme à cet endroit, une fois le sous-programme exécuté. Le registre `IP` contient à tout instant l'adresse de l'instruction suivante à exécuter. Lorsque le processeur exécute l'instruction d'appel d'un sous-programme, `IP` contient donc l'adresse de l'instruction où devra reprendre l'exécution de l'appelant (cette adresse se nomme *adresse de retour*). Il suffit donc de mémoriser à l'instant où l'on exécute l'instruction d'appel de sous-programme la valeur de ce registre. En restaurant la valeur du registre `IP` à l'issue de l'exécution du sous-programme, l'exécution du programme reprendra naturellement, là où il faut, dans l'appelant.

On pourrait utiliser un registre pour sauvegarder cette valeur du registre `IP`. Cependant, si le sous-programme appelé appelle lui-même un autre sous-programme, où va-t-on sauvegarder la valeur de `IP` pour pouvoir reprendre son exécution, puisque le registre de sauvegarde est déjà utilisé. Deux registres de sauvegarde ne ferait pas non plus l'affaire : rien n'empêche un sous-programme qui appelle un sous-programme d'appeler un sous-programme, ... autant de fois que l'on veut. Pour résoudre ce problème, nous utilisons une structure de données fondamentale en informatique, une *pile*. Pour comprendre les piles en informatique, il suffit de comprendre le fonctionnement d'une pile d'assiettes dans une cuisine. Celle-ci fonctionne de la manière suivante :

- on place une une assiette au sommet de la pile
- on prend l'assiette qui se trouve en sommet de la pile

En informatique, c'est la même chose pour les données et les piles de données :

- on place les données au sommet de la pile
- on récupère les données au sommet de la pile

En informatique, nous pouvons utiliser une pile pour y stocker n'importe quel objet. L'intérêt d'une pile est qu'elle constitue une structure où l'accès à un objet est uniquement réalisé sur le dernier objet empilé (comme pour une pile d'assiettes, on prend la dernière assiette posée sur la pile). Revenons, à nos sous-programmes. Quand un appel de sous-programme est effectué, on doit sauvegarder l'adresse de retour. Plaçons-la sur une pile. Si ce sous-programme fait lui-même appel à un autre sous-programme, plaçons également l'adresse de retour sur la pile (c'est-à-dire, au-dessus de l'adresse de retour empilée précédemment). Quand ce sous-programme termine son exécution, consultons le sommet de la pile : elle contient l'adresse de retour de sous-programme. Dépilons-la et continuons l'exécution du programme à cette adresse. Quand ce sous-programme termine également son exécution, consultons à nouveau le sommet de la pile ; elle contient l'adresse de retour de ce sous-programme. Si nous la dépilons et que nous reprenons l'exécution du programme à cette adresse, nous serons revenus dans le premier programme appelant. Nous pouvons maintenant itérer le processus et avoir des sous-programmes qui appellent des sous-programmes qui appellent des sous-programmes ... Le retour de l'appelé au sous-programme appelant se passera alors sans aucune difficulté en utilisant la pile contenant les adresses de retour (nous nommerons cette pile, la *pile des adresses de retour*).

4.2.3 Résumé

Nous pouvons résumer succinctement les actions à réaliser lors de l'appel et lors du retour d'appel d'un sous-programme. Supposons que nous ayons une pile de retour. L'appel d'un sous-programme est réalisé simplement en empilant la valeur courante de `IP` et en stockant dans `IP` l'adresse d'appel du sous-programme. Le retour de sous-programme est réalisé simplement en dépilant le sommet de la pile et en plaçant cette valeur dépilée dans le registre `IP`.

4.2.4 La pile d'appel de sous-programme

La pile de retour

Nous avons indiqué que l'utilisation d'une pile de retour permet de résoudre le problème des appels de sous-programmes. Cependant, nous n'avons pas encore vu comment on réalise une pile en assembleur. C'est en fait fort semblable à l'implantation d'une pile dans un langage évolué. Par ailleurs, certaines instructions et certains registres sont prévus pour gérer la pile de retour en assembleur.

La pile elle-même est une zone mémoire (une espèce de tableau) où le processeur range les données à sauvegarder. Le seul problème à résoudre est alors de savoir où se trouve son sommet. En effet, si l'on voit clairement où se trouve le sommet d'une pile d'assiettes dans une cuisine, il est difficile de dire que l'on voit ce sommet pour une pile informatique. Aussi, nous avons besoin d'un indicateur qui va nous « montrer » ce sommet. Les éléments de la pile étant donc situés dans une zone mémoire, les éléments étant placés les uns à la suite des autres, il nous suffit d'avoir un registre qui indique l'adresse du sommet de la pile. L'élément en-dessous du sommet de la pile se trouve alors dans l'emplacement mémoire voisin, ... Ce registre se nomme le *pointeur de sommet de pile*. C'est le registre `SP` (`SP` signifie *stack pointer*, c'est à dire « pointeur de pile »). Quand on empile une donnée, ce pointeur de pile doit être mis à jour pour indiquer le nouveau sommet de pile (car « la pile monte »). Quand on dépile une donnée, ce pointeur doit également être mis à jour (car « la pile descend »). Etant donnée l'importance de ces opérations, les instructions d'appel et de retour de sous-programme modifient automatiquement le pointeur de pile pour que les appels et les retours de sous-programmes fonctionnent correctement.

Classiquement, les piles sont implantées avec leur base au-dessus de leur sommet, c'est-à-dire que l'adresse du sommet de pile est inférieure à l'adresse de leur base. Aussi, quand une donnée est empilée, le pointeur de sommet de pile est décrémenté, alors que lorsqu'une donnée est dépilée, le pointeur de sommet de pile est incrémenté.

Retour sur les instructions d'appel et de retour de sous-programmes

Ayant décrit l'implantation de la pile de retour, nous pouvons maintenant décrire complétement le fonctionnement des instructions d'appel et de retour de sous-programmes.

Instruction d'appel de sous-programme

À l'appel d'un sous-programme, les opérations suivantes sont automatiquement réalisées :

- `SP` est décrémenté
- (`SP`) ← adresse de retour (le contenu du registre `IP`)
- `IP` ← opérande de l'instruction d'appel de sous-programme

Instruction de retour d'un sous-programme

- `IP` ← (`SP`)
- `SP` est incrémenté

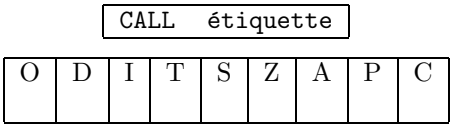
4.3 Appel de sous-programme sans passage de paramètre

4.3.1 Appels et retours de sous-programmes

Nous nous intéressons ici au cas le plus simple d'appel de sous-programme, le cas où le sous-programme ne prend pas de paramètre en entrée. Il suffit alors d'utiliser une instruction `CALL` pour appeler le sous-programme et une instruction `RET` pour effectuer le retour à l'appelant. Nous décrivons ces deux instructions.

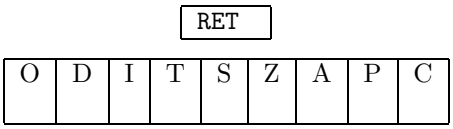
`CALL` : Appel d'un sous-programme

Nous décrivons ici l'instruction d'appel d'un sous-programme.



Les appels de sous-programme entraînent la sauvegarde dans la pile du contenu du registre `IP` uniquement. La valeur du registre `IP` est mise à jour avec la valeur de l'opérande fournie à l'instruction `CALL` pour que l'exécution du programme se poursuive en début de sous-programme. La valeur du registre pointeur de pile `SP` est bien entendu mise à jour après empilement de la valeur de `IP`. Le contenu du registre `CS` n'est pas modifié.

`RET` : retour d'un sous-programme



`RET` rend le contrôle au sous-programme appelant. Pour cela, le sommet de pile est dépilé dans le registre `ip`. L'adresse de retour doit donc nécessairement se trouver en sommet de pile.

Squelette d'un sous-programme

Nous donnons ci-dessous le squelette d'un programme en assembleur :

```
nom_sbr   proc
          ....
               corps du sous-programme
          ....
          ret
nom_sbr   endp
```

Nous notons les points suivants :

- le sous-programme est « déclaré » par la pseudo-instruction `proc`, son nom étant indiqué dans le champ étiquette.
- on trouve ensuite les instructions du sous-programme
- l'instruction `RET` indique le retour à l'appelant
- la pseudo-instruction `endp` indique la fin du sous-programme. On indique le nom du sous-programme dans le champ étiquette

4.3.2 Sauvegarde et restauration de contexte

Un sous-programme est libre de modifier le contenu de tous les registres. Cependant, l'appelant peut avoir stocké des valeurs dans des registres qu'il aimerait ne pas voir modifié durant l'exécution d'un sous-programme qu'il appelle.

Pour cela, la solution consiste à sauvegarder la valeur des registres en question. Afin que tout fonctionne bien et afin d'avoir un mécanisme standard, cette sauvegarde est effectuée sur la pile. On peut réaliser ces opérations à l'aide des instructions déjà présentées. Ainsi, sauvegarder la valeur du registre `AX` sur la pile peut se faire de la manière suivante :

```
sub    sp, 2
mov    [sp], ax
```

La première instruction place la valeur du registre `AX` au sommet de la pile. La deuxième fait pointer le sommet de pile « au-dessus » (ou « au-dessous », selon le point de vue adopté).

Cependant, ces opérations de sauvegarde/restauration du contenu de registres sont tellement courantes que des instructions spéciales ont été ajoutées pour les effectuer. L'instruction `PUSH` place la valeur spécifiée en opérande sur la pile. L'instruction `POP` retire le contenu du sommet de la pile et le place dans l'opérande du `POP`.

Il est important de noter que par convention, il est sous la responsabilité de l'appelé de sauvegarder la valeur des registres qu'il utilise et de restaurer leur valeur à la fin de l'exécution du sous-programme. L'appelant n'a pas à se soucier de ces sauvegardes.

Nous présentons maintenant les instructions `PUSH` et `POP`.

`PUSH` **sauvegarde sur la pile d'une valeur**

```
PUSH  registre/variable
```

O	D	I	T	S	Z	A	P	C

`registre 16 bits` dénote l'un des registres `ax`, `bx`, `cx`, `dx`, `si` ou `di`.

L'instruction `PUSH` place au sommet de la pile la valeur de l'opérande qui est donc nécessairement le contenu d'un registre. La valeur du pointeur de sommet de pile `SP` est diminuée de 2.

`PUSHF` **sauvegarde sur la pile de la valeur du registre d'indicateurs**

```
PUSHF
```

O	D	I	T	S	Z	A	P	C

`PUSHF` empile le registre d'indicateurs `FLAGS`. La valeur du pointeur de sommet de pile `SP` est ensuite diminuée de 2.

`POP` **dépilement d'une valeur**

```
POP  registre/variable
```

O	D	I	T	S	Z	A	P	C

`POP` dépile une valeur de la pile de retour et place la valeur ainsi dépilée dans l'opérande. La valeur du pointeur de sommet de pile `SP` est augmentée de 2.

`POPF` **dépilement de la valeur du registre d'indicateurs**

```
POPF
```

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

`POPF` dépile la valeur de `FLAGS`. La valeur du pointeur de sommet de pile est augmentée de 2.

4.4 Passage de paramètres et variables locales

Un sous-programme prend souvent des paramètres en entrée dès qu'il est capable de réaliser des actions un peu complexes. Il est également courant qu'un sous-programme ait besoin de variables locales. Enfin, il est également courant qu'un sous-programme ait des paramètres en sortie (résultat) ou renvoie une valeur (comme une fonction dans un langage de haut niveau). Nous traitons de tous ces mécanismes dans cette section.

Comme toujours, pour autoriser une flexibilité maximale, tous ces mécanismes vont être implantés à l'aide de la pile. Les paramètres seront empilés par l'appelant juste avant l'appel. L'appelé réservera, si nécessaire, un espace dans la pile pour y mettre la valeur de ses variables locales. Quant au renvoi d'une valeur, il se fera généralement par un registre. Nous décrivons maintenant dans le détail chacun de ces mécanismes.

4.4.1 Passage de paramètres par registre

Le mode de passage de paramètres le plus simple consiste à passer la valeur dans un registre : l'appelant charge un registre avec la valeur à passer ; l'appelé récupère la valeur dans ce même registre.

Appelant

```
...
mov   ax, 8
call  facto
...
```

Appelé

```
facto proc
      ...        ; parametre dans ax
      ...
      ret
facto endp
```

Nous donnons un exemple de sous-programme calculant la factorielle d'un nombre dont la valeur est passée en entrée dans le registre `AX`.

Appelant

```
...
mov   ax, 8
call  facto
...
```

Appelé

```
facto proc
      push  cx
      mov   cx, ax
      mov   ax, 1
boucle1:
      mul   ax, cx
      loop  boucle1
      pop   cx
      ret
facto endp
```

4.4.2 Passage de paramètres dans la pile

Le passage de paramètres par la pile permet de tout faire. C'est de cette manière que tous les langages de haut niveau passent leur paramètres aux procédures et fonctions. La maîtrise de cette technique est indispensable pour bien programmer en assembleur. Pour cela, un peu de rigueur quant à la manipulation de la pile de retour est nécessaire. Le processeur utilise implicitement la pile pour y empiler l'adresse de retour des sous-programmes. Il y a donc une contrainte à respecter lorsque l'on veut utiliser la pile :

Le microprocesseur doit impérativement trouver l'adresse de retour d'un sous-programme au sommet de la pile lorsqu'il exécute une instruction `RET`.

Si on respecte cette convention, on peut faire tout ce que l'on veut et cela fonctionnera.

Par convention, nous devons décider d'un protocole commun aux sous-programmes appelants et appelés quant à la localisation des données sur la pile. Nous savons que le processeur empile la valeur du registre `IP` quand il exécute une instruction `CALL`. Le sous-programme appelant est responsable du passage des paramètres, c'est-à-dire de leur empilage. Aussi, nous réaliserons le passage de paramètres de la manière suivante : l'appelant empile (par des instructions `PUSH`) la valeur des paramètres puis exécute l'appel du sous-programme (instruction `CALL`). La pile a alors la forme indiquée à la figure 4.1.

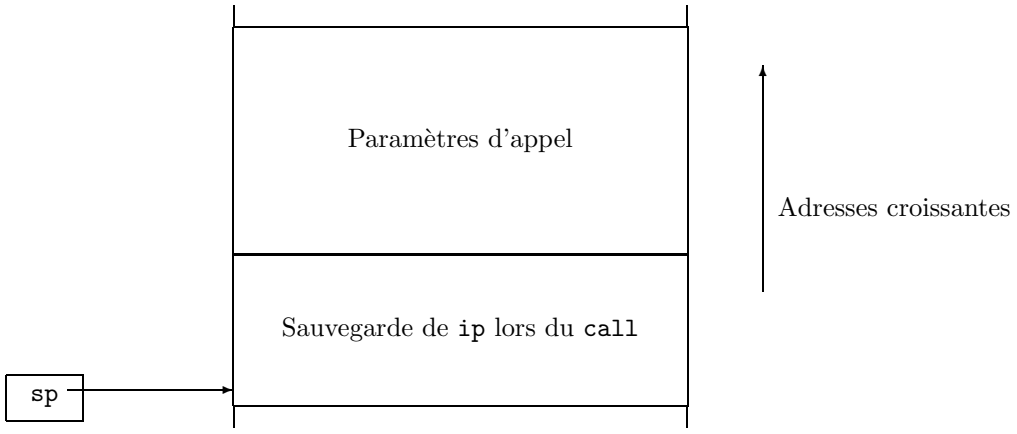


FIG. 4.1 – *Pile après exécution de l'instruction* `call`.

æ

Pour l'appelé, la convention est d'utiliser le registre `BP` pour accéder à la valeur des paramètres. `BP` doit donc être initialisé au début de l'exécution de l'appelé. Aussi, sa valeur doit être également sauvegardée. Cette sauvegarde doit être réalisée par l'appelé.

Pour résumer, nous donnons le squelette d'un appel de sous-programme avec passage de paramètres. Supposons que nous passions trois paramètres, les valeurs des registres `AX`, `BX` et `CX`.

Appelant

```
...
push  ax
push  bx
push  cx
call  sbr
add   sp, 6
...
```

Appelé

```
sbr   proc
      push  bp
      mov   bp, sp
      ...
      ...   ...[bp+4]...
      ...   ...[bp+6]...
      ...   ...[bp+8]...
      ...
      ...
      pop   bp
      ret
sbr   endp
```

Nous notons les points suivants :

– dans l'appelant, les valeurs des paramètres sont empilées avant l'appel du sous-programme

- dans l'appelé (donc, la valeur du compteur ordinal `IP` à été empilée sur 2 octets), la valeur du registre `BP` est empilée

- la valeur du registre `BP` est initialisée avec la valeur du sommet de pile contenue dans `SP`

- on peut accéder maintenant à la valeur des paramètres via le registre `BP`. Ainsi,

  - `[bp+4]` accède au dernier paramètre empilé (donc la valeur du registre `CX`)
  - `[bp+6]` accède au paramètre empilé avant `CX` (donc la valeur du registre `BX`)
  - `[bp+8]` accède au paramètre empilé avant `BX` (donc la valeur du registre `AX`)

- en fin de sous-programme appelé, on doit restaurer la valeur du registre `BP` pour que l'appelant n'ait pas de problème. C'est le rôle de l'instruction `POP`

- après l'exécution du `POP`, le sommet de pile contient l'adresse de retour ; un `RET` peut donc être exécuté en toute sécurité

- de retour dans l'appelant, la zone mémoire utilisée pour stocker des paramètres doit être libérée. Pour cela, on modifie simplement la valeur du registre pointeur de sommet de pile `SP` en lui ajoutant la valeur 6 qui est le nombre d'octets occupés sur la pile par les valeurs empilées (2 pour `AX`, 2 pour `BX` et 2 pour `CX`). On aurait pu également libérer la pile avec des instructions `POP`. Cependant, l'habitude veut que l'on écrive les programmes comme il a été indiqué ici. Pai ailleurs, utiliser des instructions `POP` modifie la valeur des registres, ce qui peut amener des effets indésirables

Nous donnons un exemple d'appel de sous-programme : un sous-programme qui calcule le PGCD de deux nombres passés en paramètre.

L'algorithme utilisé est le suivant : soit à calculer le PGCD de x et y :

- Tant-que x $\neq$ y Faire

  - Si $x > y$ Alors $x \leftarrow x - y$
  - Sinon Si $x < y$ Alors $y \leftarrow y - x$

- Fait

Sortant de la boucle, x (ou y : en sortie de boucle, les deux variables contiennent la même valeur) contient la valeur du PGCD.

Appelant

```
...
mov   ax, 453
push  ax  ; empile x
mov   ax, 36
push  ax  ; empile y
call  pgcd
add   sp, 4
...
```

Appelé

```
pgcd  proc
      push  bp
      mov   bp, sp
      push  ax          ; sauvegarde de
      push  bx          ;    ax et bx
      mov   ax, [bp+4]  ; charge y dans ax
      mov   bx, [bp+6]  ; charge x dans bx
tq1:  cmp   ax, bx
      jg    sup         ; saut si y>x
      je    ftq1        ; sortir si x=y
      sub   ax, bx
      jmp   tq1
sup:  sub   bx, ax
      jmp   tq1
ftq1: call  affiche_nombre
      pop   bx
      pop   ax
      pop   bp
      ret
pgcd  endp
```

On suppose que l'on dispose du sous-programme `affiche_nombre` qui affiche le résultatun nombre à l'écran.

4.4.3 Variables locales

Nous abordons maintenant le problème des variables locales. Celui-ci va également être résolu en utilisant la pile de retour. Les locales sont, par définition, uniquement connues du sous-programme où elles sont déclarées. Aussi, la gestion des locales doit-elle être entièrement réalisée dans l'appelé. Pour cela, la solution la plus simple consiste, une fois sauvegardée la valeur des registres en début de sous-programme, à réserver de la place dans la pile pour y mettre la valeur des locales.

Cela donne la configuration de la figure 4.2.

æ

Nous donnons le squelette d'un appel de sous-programme avec passage de paramètres et définition de variables locales. Supposons que nous passions deux paramètres, les valeurs des registres `AX`, `BX`, que nous devions sauvegarder la valeur des registres `DI` et `SI` et que nous ayons besoin de 10 octets pour stocker des variables locales.

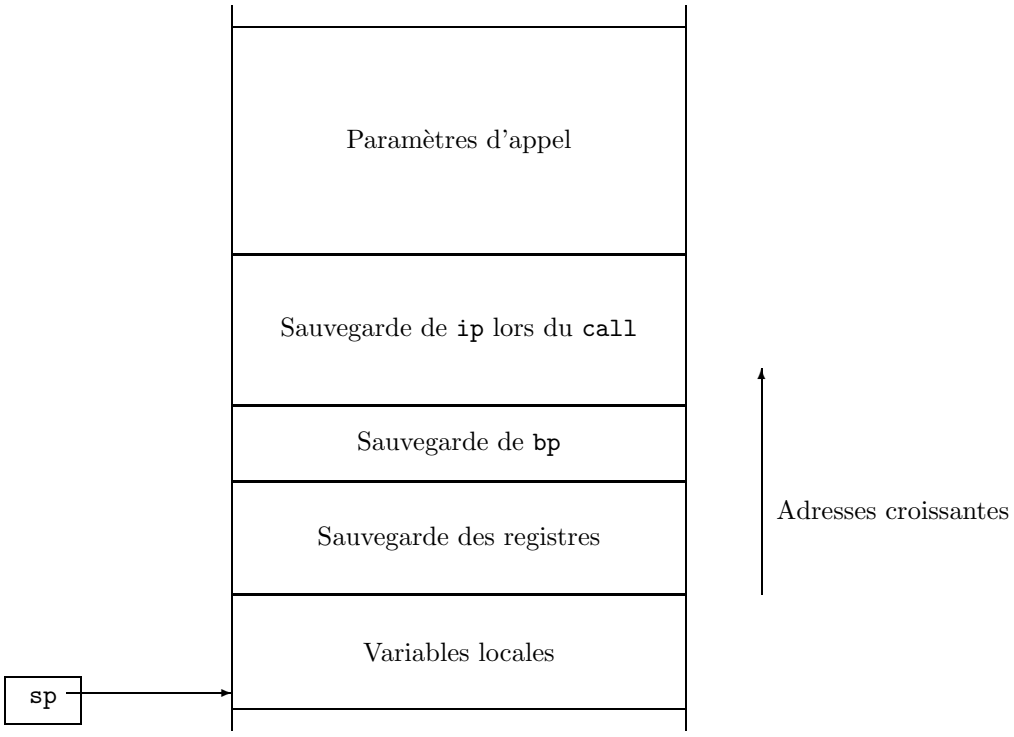


FIG. 4.2 – *Pile après réservation d'espace pour les variables locales.*

Appelant

```
...
push  ax
push  bx
call  sbr
add   sp, 6
...
```

Appelé

```
sbr   proc
      push  bp
      mov   bp, sp
      push  di
      push  si
      sub   sp, 10
      ...
      ...   ...[bp+4]...
      ...   ...[bp+6]...
      ...
      ...   ...[bp-6]...
      ...
      add   sp, 10
      pop   si
      pop   di
      pop   bp
      ret
sbr   endp
```

Le sous-programme appelant ressemble énormément au cas présenté plus haut de passage de paramètres

et n'appelle donc aucun commentaire particulier. Pour l'appelé, nous notons les points suivants :

- la sauvegarde (comme auparavant) du registre `BP`
- l'initialisation du registre `BP` pour qu'il pointe au sommet de la pile
- la sauvegarde de la valeur des registres `DI` et `SI`. Notons que `BP` pointe sur la valeur de la sauvegarde du registre `DI`
- la réservation de 10 octets sur la pile (par l'instruction `sub sp, 10`) qui pourront être manipulés librement dans le sous-programme et dont le rôle est de contenir la valeur des variables locales
- l'accès aux variables locales se fera dès lors par un adressage du genre `[bp-6]` ... `[bp-15]`, selon la taille des données. Notons, c'est très important, que le programmeur doit définir lui-même l'utilisation de ces 10 octets.
- en fin d'appelé, on libère la zone de pile utilisée pour stocker la valeur des variables locales (instruction `ADD`)
- on restaure ensuite la valeur d'entrée des registres `SI`, `DI` et `BP`.

4.4.4 Retour d'une valeur par un sous-programme : les fonctions en assembleur

Dans un langage de haut niveau, une fonction est une procédure qui retourne une valeur. En assembleur, une fonction sera un sous-programme qui retournera une valeur. Par convention, la valeur renvoyée est placée dans un registre.

Par exemple, le sous-programme suivant calcule la valeur de la factorielle du nombre passé en paramètre dans la pile et renvoie cette valeur dans le registre `AX` :

Appelant

```
...
mov   ax, 8
push  ax
call  facto
...
```

Appelé

```
facto proc
      push  bp
      mov   bp, sp
      push  cx
      mov   cx, [bp+4]
      mov   ax, 1
boucle1:
      mul   cx
      loop  boucle1
      pop   cx
      pop   bp
      ret
facto endp
```

Chapitre 5

Les interruptions du DOS

Nous présentons ici très succinctement quelques interruptions du BIOS qui nous serons utiles pendant les TPs pour, par exemple, afficher un message à l'écran ou saisir un caractère au clavier. D'une manière générale, toutes les opérations d'entrées/sorties sont réalisées par interruptions BIOS.

Le mécanisme d'interruptions sera décrit en détail en cours.

Sachons simplement ici qu'une interruption est une espèce de sous-programme pré-existant dans la machine. L'appel se fait via l'instruction `int 21h`. Le registre `ah` contient un numéro qui référence la fonctionnalité que l'on veut utiliser (9 pour afficher une chaîne de caractères, 1 pour saisir la frappe d'un caractère au clavier, ...)

5.1 Affichages de caractères à l'écran

5.1.1 Afficher un caractère à l'écran

```
      mov    dl, 'a'
      mov    ah, 2
      int    21h
```

affiche le caractère `a` à l'écran.

5.1.2 Afficher un message à l'écran

```
      .data
msg   db     'hello world', 13, 10, '$'
      .code
      mov    ax, @data
      mov    ds, ax
      mov    ah, 9
      lea    dx, msg
      int    21h
      mov    ah, 4ch
      int    21h
      end
```

affiche le message `hello world` à l'écran. La chaîne de caractères doit impérativement se terminer par un caractère `'$'`. Les caractères 13 et 10 sont les codes ASCII des 2 caractères constituant un retour-chariot.

5.2 Saisir une frappe au clavier

5.2.1 Saisie avec écho

Saisie d'un caractère

```
      mov    ah, 1
      int    21h
```

renvoie dans le registre `al` le code du caractère lu au clavier.

Saisie d'une chaîne de caractères

```
      .data
msg   db     80 dup (?)
      .code
      mov    ax, @data
      mov    ds, ax
      mov    bx, 0
      mov    cx, 80
      lea    dx, msg
      mov    ah, 3fh
      int    21h
```

saisit une chaîne de caractères au clavier d'au plus `cx` caractères et la stocke dans la variable `msg` qui est déclarée comme un tableau de 80 octets (soit 80 caractères). Le nombre de caractères lus est placé dans le registre `ax`. Notons que le retour-chariot est stocké sous forme de deux caractères successifs : le premier de code 13, le second de code 10.

5.2.2 Saisie sans écho

```
      mov    ah, 8
      int    21h
```

lit un caractère au clavier et le renvoie dans le registre `al`. Ce caractère n'est pas affiché à l'écran.

5.3 L'accès aux fichiers

Nous présentons ici les interruptions à utiliser pour effectuer les traitements de base sur les fichiers. On se reportera à 6.3 pour un exemple complet de traitement de fichiers.

5.3.1 Principe

En assembleur, les fichiers sont traités d'une manière assez similaire à un langage évolué comme Pascal. Pour pouvoir accéder aux données d'un fichier, que ce soit en lecture ou en écriture, il faut préalablement l'ouvrir. À l'ouverture, le programme récupère une *poignée*. Ensuite, toutes les opérations sur ce fichier seront effectuées en spécifiant cette poignée. À un instant donné, plusieurs fichiers peuvent avoir été ouverts par un programme. Chacun a une poignée différente laquelle permet au programme de les distinguer.

5.3.2 Création d'un fichier

Avant l'appel

- `cx` contient la valeur 0 ;
- `ds:dx` pointent sur le nom de fichier. C'est une chaîne de caractères terminée par un caractère ASCII de code 0.

Après l'appel

Le bit `C` du registre `flags` vaut 1 si une erreur est survenue.

L'appel a le format suivant :

```
mov  ah, 3ch
int  21h
```

5.3.3 Ouverture d'un fichier

Avant l'appel

- `al` contient le mode d'accès : 0 pour lecture, 1 pour l'écriture, 2 pour les deux ;
- `ds:dx` pointent sur le nom de fichier. C'est une chaîne de caractères terminée par un caractère ASCII de code 0.

Après l'appel

Le bit `C` du registre `flags` vaut 1 si une erreur est survenue. Sinon, le registre `ax` contient la poignée sur le fichier ouvert.

L'appel a le format suivant :

```
mov  ah, 3dh
int  21h
```

5.3.4 Fermeture d'un fichier

Avant l'appel

- `bx` contient la poignée du fichier à fermer

Après l'appel

Le bit `C` du registre `flags` vaut 1 si une erreur est survenue. Sinon, le fichier est fermé. Il ne peut plus être accédé via sa poignée. Si on veut l'accéder à nouveau, il faudra le ré-ouvrir au préalable.

L'appel a le format suivant :

```
mov  ah, 3eh
int  21h
```

5.3.5 Lecture dans un fichier

Avant l'appel	Après l'appel
– `bx` contient la poignée du fichier ; – `cx` contient le nombre de caractères à lire ; – `ds:dx` contient l'adresse du tampon où il faut stocker les caractères lus.	Le bit `C` du registre `flags` vaut 1 si une erreur est survenue. Sinon, le registre `ax` contient le nombre de caractères lus. En principe, c'est le nombre de caractères qui devait être lus à moins que la fin de fichier ne soit atteinte.

On peut utiliser cet appel pour lire une chaîne de caractères tapés au clavier. Pour cela, il faut que `bx` contienne la valeur 0 en entrée. La lecture s'effectuera sur au plus `cx` caractères et au plus jusqu'à la frappe d'un retour-chariot.

L'appel a le format suivant :

```
mov  ah, 3fh
int  21h
```

5.3.6 Écriture dans un fichier

Avant l'appel	Après l'appel
– `bx` contient la poignée du fichier ; – `cx` contient le nombre de caractères à écrire ; – `ds:dx` pointent sur le tampon contenant les caractères à écrire.	Le bit `C` du registre `flags` vaut 1 si une erreur est survenue. Sinon, le registre `ax` contient le nombre de caractères écrits.

L'appel a alors le format suivant :

```
mov  ah, 40h
int  21h
```

5.4 Lire l'heure courante

La fonction `2ch` de l'interruption `21h` lit l'heure courante, telle qu'elle est stockée dans l'ordinateur (s'il n'est pas à l'heure, cela ne donnera pas la bonne heure).

```
mov    ah, 2ch
int    21h
```

Au retour de l'appel, les registres contiennent les information suivantes :

`ch` heures
`cl` minutes
`dh` secondes
`dl` centièmes de secondes

Voir la section 6.4 pour un exemple d'utilisation de cette interruption.

5.5 Lire la date courante

La fonction `2ah` de l'interruption `21h` lit la date courante, telle qu'elle est stockée dans l'ordinateur (comme pour l'heure, si la date n'est pas bonne, cela ne donnera pas la bonne date).

```
mov    ah, 2ah
int    21h
```

Au retour de l'appel, les registres contiennent les information suivantes :

- `al` jour de la semaine codé (0 : dimanche, 1 : lundi, ...)
- `cx` année
- `dh` mois
- `dl` jour

Chapitre 6

Études de cas

6.1 Cas 1 : lire une suite de caractères au clavier

Problème à résoudre

Écrire un programme qui lit des caractères frappés au clavier et les affichent à l'écran tant que l'utilisateur n'appuie pas sur `<esc>`.

Principe

Faire une boucle :

```
repeter
  lire (c)
  si (c # ESC) alors
    affiche (c)
  fin_si
jusque c = ESC
```

En accord avec ce qui est dit sur les boucles `repeter` dans la méthodologie de programmation en assembleur, on doit tout d'abord transformer cette boucle en :

```
repeter:
  lire (c)
  si c # ESC alors
    afficher (c)
  fin_si
  si c # ESC alors
    jmp repeter
  fin_si
```

Puisque les deux `si` répondent à la même condition, on peut encore écrire la boucle sous la forme suivante :

```
repeter:
  lire (c)
  si c # ESC alors
    afficher (c)
    jmp repeter
  fin_si
```

Ce que nous pouvons traduire en assembleur :

Programme

```
        .MODEL SMALL
        .DATA
        .CODE
        mov    ax, @data
        mov    ds, ax
boucle: mov    ah, 8
        int    21h
        cmp    al, 27 ; 27 est le code ASCII de la touche <Esc>
        je     fin
        mov    dl, al
        mov    ah, 2
        int    21h
        jmp    boucle
fin:    mov    ah,4ch
        int    21h
        END
```

6.2 Cas 2 : afficher un nombre en base 2

Problème à résoudre

Afficher la valeur en binaire d'un nombre.

Principe

Afficher l'un après l'autre les bits du nombre en effectuant une boucle du genre :

```
pour b := 1 a 16 faire
  si (valeur.bit[b] = 0) alors
    afficher ('0')
  sinon afficher ('1')
fait
```

Pour accéder aux bits du nombre, plusieurs techniques sont envisageables. La plus simple consiste à décaler vers la gauche les bits du nombre (instruction `SHL` ou `ROL`) en plaçant le bit qui "déborde" dans le `C` du registre d'état. En testant ensuite ce bit `C` (avec une instruction `JC` ou `JNC`), on sait s'il faut afficher un `'0'` ou un `'1'`. Remarquons qu'en utilisant une instruction `ROL`, une fois les 16 décalages effectués, la valeur a été restaurée.

Programme

```
        .MODEL SMALL
        .STACK 100h
        .DATA
valeur  dw     52612
        .CODE
        mov    ax, @data
        mov    ds, ax
        mov    bx, [valeur]
        mov    cx, 16
boucle: rol    bx, 1
```

```
        jc      un
        mov     dl, '0'
        jmp     print
un:     mov     dl, '1'
print:  mov     ah, 2
        int     21h
        loop    boucle
fin:    mov     ah,4ch
        int     21h
        END
```

affiche `1100110110000100` qui est bien la représentation binaire de 52612.

6.3 Cas 3 : Afficher le contenu d'un fichier

Nous montrons sur un exemple l'utilisation des interruptions d'accès aux fichiers.
Problème à résoudre

Afficher le contenu d'un fichier à l'écran.

Principe

Le principe de l'algorithme est le suivant :

- ouverture du fichier
- répéter
  - lire les σ caractères suivants du fichier et les mettre dans une zone mémoire prévue pour cet usage (un *tampon*). Notons que cette action lit effectivement ν caractères. En général, $\nu = \sigma$. Si la fin de fichier est atteinte pendant la lecture, on a lu les $\nu < \sigma$ derniers caractères du fichier.
  - afficher ν caractères du tampon à l'écran
- jusqu'à ce que $\nu \neq \sigma$
- fermer le fichier

Programme

```
        .MODEL SMALL
        .STACK 100H
        .DATA
Filename  DB    'toto', 0
Buffer    db    100 dup (?), '$'
err1_msg  db    'Erreur a l''ouverture', 13, 10, '$'
err2_msg  db    'Erreur a la lecture', 13, 10, '$'
        .CODE
        ;
        ; initialisations
        ;
        mov   ax,@data
        mov   ds,ax
        ;
        ; ouverture et test d'existence du fichier
        ;
        mov   ah,3dh  ; ouverture du fichier
        mov   al,0    ; accès en lecture
        lea   dx, Filename
        int   21h
        jc    err1    ; le fichier n'existe pas -> erreur
        ;
        ; préparation de la boucle
        ;
```

```
          mov   bx, ax  ; sauvegarde de la poignée
          lea   dx, Buffer
          ;
boucle:   ;
          ; lecture des 100 caractères suivants du fichier
          ;
          mov   cx, 100
          mov   ah, 3fh ; lecture du fichier
          int   21h
          jc    err2
          ;
          ; affichage a l'ecran des caractères lus
          ;
          mov   cx, ax
          mov   si, ax
          mov   si+Buffer, '$'
          mov   ah,9
          int   21h
          cmp   cx, 100
          je    boucle
          ;
          ; fin du traitement
          :
          mov   ah, 3eh ; fermeture du fichier
          int   21h
          jmp   fin
          ;
          ; traitements des erreurs
          ;
err1:     ;
          ; le fichier ne peut etre ouvert
          ;
          lea   dx, err1_msg
          jmp   affiche_msg
err2:     ;
          ; erreur lors de l'accès au contenu du fichier
          ;
          lea   dx, err2_msg
          jmp   affiche_msg
affiche_msg:
          mov   ah,9
          int   21h
          jmp   fin
fin:      mov   ah,4Ch
          int   21h
          END
```

On a supposé que le fichier se nomme `toto`.

6.4 Cas 4 : afficher l'heure courante

Problème à résoudre

Afficher l'heure courante à l'écran sous une forme lisible.

Principe

Il suffit de lire l'heure courante via l'interruption BIOS *ad hoc* et d'afficher le contenu des registres d'une manière compréhensible pour un être humain.

Programme

```
          ; Ce programme affiche l'heure courante
          .MODEL SMALL
          .STACK 100H
          .DATA
ch1       db    'Il est : $'
ch2       db    'h $'
ch3       db    'm $'
ch4       db    's, $'
          .CODE
          mov   ax,@data
          mov   ds,ax
          ;
```

```
          ; lecture de l'heure courante
          ;
          mov   ah,2ch
          int   21h
          ;
          ; maintenant, on a :
          ;   ch: heures
          ;   cl: minutes
          ;   dh: secondes
          ;   dl: 1/100 secondes
          ;
          ;
          ;   affiche les heures
          ;
          mov   ax, offset ch1
          call  aff_str
          mov   al,ch
          call  aff_nb
          ;
          ; affiche les minutes
          ;
          mov   ax, offset ch2
          call  aff_str
          mov   al,cl
          call  aff_nb
          ;
          ; affiche les secondes
          ;
          mov   ax, offset ch3
          call  aff_str
          mov   al,dh
          call  aff_nb
          ;
          ; affiche les centièmes de secondes
          ;
          mov   ax, offset ch4
          call  aff_str
          mov   al,dl
          call  aff_nb
          ;
fin:      mov   ah,4Ch
          int   21h
          ;
          ;
aff_nb    PROC
          ;
          ; le sous-programme aff_nb affiche la valeur du nombre
          ; code en hexadecimal se trouvant dans le registre al.
          ; Ce nombre doit avoir une valeur < 100.
          ;
          push  dx
          push  ax
          mov   ah, 0
          mov   dl, 10
          div   dl
          ; maintenant, al contient le chiffre des dizaines,
          ;  ah, le chiffre des unites
          mov   dx, ax
          or    dx, 3030h
          ; les deux chiffres ont ete transformes en code ASCII
          ; on affiche le chiffre des dizaines
          mov   ah,2
          int   21h
          ; on affiche le chiffre des unites
          xchg  dl, dh
          mov   ah,2
          int   21h
          pop   ax
          pop   dx
          ret
aff_nb    ENDP
          ;
          ;
aff_str   PROC
          ;
          ; le sous-programme aff_str affiche la chaine de caracteres
          ; dont l'offset se trouve dans le registre ax, dans le segment
          ; courant.
          ;
          push  dx
```

```
          push  ax
          mov   dx,ax
          mov   ah,9
          int   21h
          pop   ax
          pop   dx
          ret
aff_str   ENDP
          END
```

Annexe A

Introduction à l'utilisation du debugger

Nous indiquons ici en quelques mots l'utilisation du debugger. Un debugger est un outil qui permet d'exécuter instruction par instruction un programme tout en suivant en permanence l'évolution du contenu des registres, du segment de données, la pile, ... C'est un outil indispensable pour simplifier et accélérer la mise en point de programmes.

A.1 Lancement et présentation du debugger

Lancement du debugger

Après avoir obtenu un programme exécutable (que nous appellerons ici `prg.exe` à titre d'exemple), on lance la commande :

```
td2 prg
```

L'écran s'efface et affiche une interface avec une fenêtre dont on peut modifier la taille avec la souris. Une barre de menus est disponible en haut de l'écran. L'écran ressemble alors à ce qui est présenté à la figure A.1.

æ

Notons immédiatement que l'on quitte le debugger en tapant `Alt-X`.

Que peut-on faire avec un debugger ?

Sans entrer tout de suite dans les détails, il est important de comprendre avant tout ce que l'on peut attendre d'un debugger et le principe général de son utilisation.

Sans en attendre de miracles, un debugger aide (grandement) à répondre aux questions suivantes :

- pourquoi mon programme ne fonctionne-t-il pas ?
- pourquoi mon programme boucle-t-il ?
- où se plante mon programme ?

Pour cela, une fois dans le debugger, on peut placer des *points d'arrêt* dans le programme afin qu'il s'y arrête [1]. C'est-à-dire, on indique les instructions où l'on veut que le programme s'arrête. Une fois arrêté, on pourra consulter à loisir le contenu des registres, le contenu de la mémoire ou de la pile.

Nous détaillons maintenant le contenu de la fenêtre et présentons les fonctionnalités les plus utiles.

1. il est important de bien noter que si l'on ne place pas de points d'arrêt dans le programme, celui-ci va s'exécuter sans s'arrêter avant sa terminaison normale (donc, s'il boucle, il ne s'arrêtera pas plus sous le debugger que si on l'exécute sous DOS). Dans ce cas, exécuter le programme sous le debugger ou directement sous DOS ne change rien, si ce n'est que l'on pourra ensuite consulter le contenu des registres.

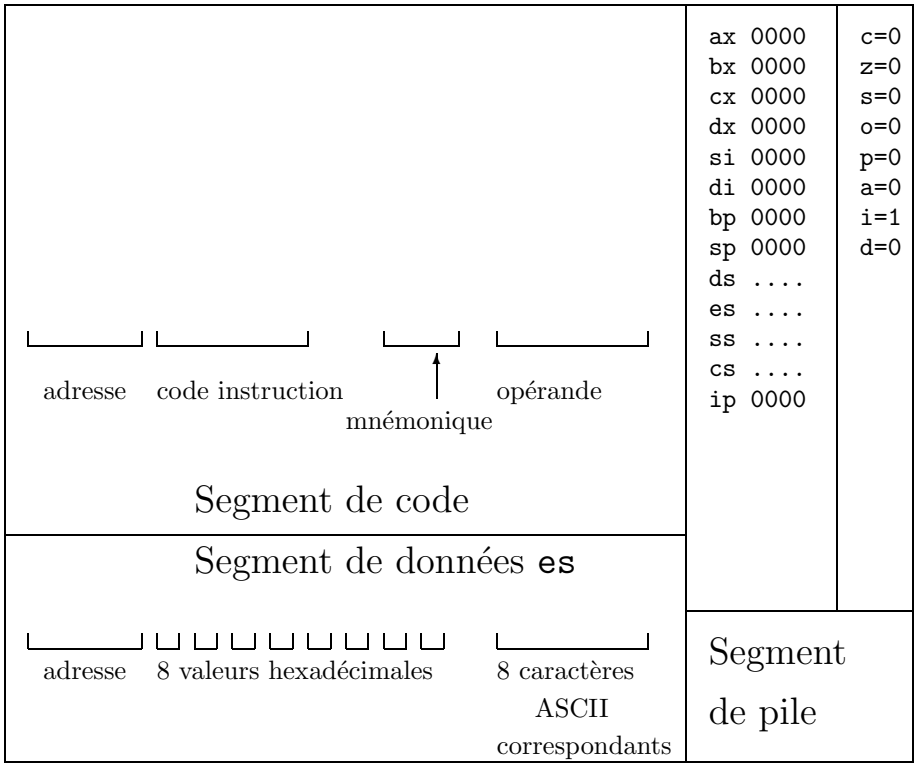


FIG. A.1 – *La fenêtre du débugger.*

A.1.1 La fenêtre de debuggage

Le menu `Run`

Parmi d'autres possibilités, ce menu vous propose :

- `Run` lance le programme. Si son exécution a déjà débutée, le programme poursuit son exécution là où il avait été arrêté. Sinon, il démarre à son début. Il ne s'arrête que lorsqu'il arrive à son terme ou sur le point d'arrêt (cf. plus loin) suivant, si l'on en a placé ;
- `Go to cursor` ayant cliqué sur une instruction, le programme débute ou poursuit son exécution jusqu'à atteindre cette instruction ;
- `Trace into` arrêté sur une instruction `call`, considère l'exécution de l'ensemble du sous-programme comme un pas d'exécution ;
- `Step over` exécute la prochaine instruction et s'arrête ;
- `Until return` exécute les instructions qui suivent jusque la fin du programme ou jusqu'à la prochaine instruction `ret` ;
- `Animate` déclenche l'exécution instructions par instructions. La vitesse d'exécution est réglable et est demandée à l'utilisateur ;
- `Prg Reset` remise à zéro de l'exécution du programme. Remet le debugger dans son état initial, comme si l'on n'avait pas encore lancé l'exécution du programme.

Les points d'arrêt

Un point d'arrêt est un endroit dans le programme où on indique que l'on veut que celui-ci arrête son exécution.

Il existe deux manières de mettre en place des points d'arrêt :

- soit implicitement en cliquant sur une instruction du programme et en lançant ensuite l'exécution du programme par un `Go to cursor`
- soit explicitement en cliquant sur une instruction du programme et en choisissant dans le menu `Breakpoints` l'option `At` d'y placer un point d'arrêt. On peut placer plusieurs points d'arrêt dans le programme. À son exécution (lancement par `Run`, `Go to cursor`, ...), le programme s'arrêtera automatiquement à chaque fois qu'il arrivera sur une instruction où on a placé un point d'arrêt. L'instruction n'est pas exécutée avant l'arrêt du programme. Elle l'est quand on continue l'exécution du programme.

A.2 Quelques opérations de base

A.2.1 Afficher le segment de données

Au lancement de `td`, le debugger ne connaît pas l'emplacement en mémoire du segment de données, le registre `ds` étant initialisé par le programme, en général à l'exécution des premières instructions du programme. Une fois `ds` initialisé, on peut visualiser le contenu du registre de segment dans une fenêtre en choisissant dans le menu `View` l'option `Dump`.

A.2.2 Afficher le contenu d'un segment mémoire quelconque

On peut visualiser le contenu de la mémoire à partir de n'importe quelle adresse en spécifiant un numéro de segment et un offset. Pour cela, ouvrir un "dump" mémoire en sélectionnant `Dump` dans le menu `View`. Ensuite, en cliquant dans la fenêtre avec le bouton droit de la souris, un menu apparait. Sélectionner `Goto...`. Le debugger demande l'adresse à partir de laquelle il doit afficher le contenu de la mémoire dans la fenêtre. On pourra alors taper `es:0`, `ds:0` ou `53DA:14`, selon ses besoins.

A.2.3 Modifier la valeur d'un registre, d'un indicateur, d'une donnée en mémoire

On peut facilement modifier la valeur d'un registre, d'un indicateur du registre d'état ou un octet en mémoire en cliquant dessus, en entrant sa nouvelle valeur et en validant.

A.3 Remarques

Il faut noter que certaines instructions n'apparaissent pas sous le debugger comme vous avez pu les taper dans votre programme source. Ainsi, comme nous l'avons vu, les instructions `sal` et `shl` sont synonymes. Aussi, quelle que soit celle que vous avez tapé dans le source de votre programme, c'est toujours `shl` qui sera affichée.

Par ailleurs, les instructions de décalages du genre `shr al,3` où le deuxième opérande est une valeur immédiate sont transformées en une séquence de 3 instructions `shr al,1`.

Les instructions du genre `push 23`, c'est-à-dire `push` d'une valeur immédiate, sont transformées en une séquence de plusieurs instructions. Par exemple, pour celle-ci, on obtiendra :

```
push  ax
push  bp
mov   word ptr [bp+2], 17
pop   bp
```

qui empile bien la valeur 23 (c'est-à-dire `17h`).

Annexe B

Paramètres de la ligne de commande et environnement

Lorsqu'on lance un programme, on peut spécifier des paramètres dans la ligne de commande (des noms de fichiers par exemple) quipourront être pris en compte par le programme durant son exécution. Par ailleurs, outre les paramètres de la ligne de commande, le système d'exploitation passe automatiquement au programme les variables d'environnement (c'est-à-dire, les variables positionnées par la commande MSDOS `set`). Dans la présente annexe, nous indiquons comment ces informations peuvent être accédées depuis un programme assembleur.

Tout programme en cours d'exécution possède une structure de données qui lui est propre est qui porte le doux nom de PSP, acronyme de *Program Segment Prefix*. Ce PSP est toujours composé de 256 octets. Parmi ceux-ci, on trouve :

- à l'offset `2ch` le numéro du segment où est situé l'environnement ;
- à partir de l'offset `81h`, la ligne de commande terminée par un caractère de code `0Dh` ;
- à l'offset `80h`, le nombre de caractères de la ligne de commande (sans compter le `0Dh`).

B.1 L'accès au PSP

L'interruption 21h renvoie le numéro du segment contenant le PSP si on place la valeur `62h` dans le registre `ah`. À son retour, on récupère le numéro du segment contenant le PSP dans le registre `bx`. Le PSP occupe les octets d'offset 0 à 255 de ce segment.

B.2 La ligne de commande

Nous indiquons sur des exemples la structure de la ligne de commande telle qu'elle est stockée dans le PSP. Si sous MS-DOS, on tape la commande :

```
prg a b c d e
```

on trouvera les informations suivantes dans le PSP :

```
80: 0A 20 65 20 66 20 67 20   a b c
88: 68 20 69 0D ........... d e
```

Il faut noter que les redirections ne font pas partie des paramètres de la commande et n'apparaissent pas dans le PSP. Ainsi, si on tape les commandes :

```
prg a b c d e > toto
prg a b c d e < titi > toto
```

```
prg a b c d e | more
```

on aura toujours le même contenu dans le PSP (celui donné plus haut).

B.3 L'environnement

L'environnement d'un programme est un ensemble de couples

< nom de variable, valeur >

Le nom d'une variable et sa valeur sont des chaînes de caractères. Dans le PSP, ils sont séparés par un signe `=` pour les distinguer. Chaque couple est terminé par un caractère ASCII de code `00`. La liste des couples se termine elle-aussi par un caractère de code `00`. (Donc, les deux derniers caractères de la zone mémoire où est stocké l'environnement ont tous deux la valeur `00`.)

Un exemple de contenu de la zone stockant l'environnement est donné ci-dessous :

```
00: 43 4f 4d 53 50 45 43 3d 43 3a 5c 43 4f 4d 4d 41 COMSPEC=C:\COMMA
10: 4e 44 2e 43 4f 4d 00 50 52 4f 4d 50 54 3d 00 54 ND.COM PROMPT= T
20: 45 4d 50 3d 43 3a 5c 54 45 4d 50 00 50 41 54 48 EMP=C:\TEMP PATH
30: 3d 43 3a 5c 44 4f 53 3b 43 3a 5c 42 41 54 3b 43 =C:\DOS;C:\BAT;C
40: 3a 5c 42 4f 52 4c 41 4e 44 43 5c 42 49 4e 3b 00 :\BORLANDC\BIN;
50: 00
```

Afin d'illustrer notre propos, nous donnons ci-dessous un programme qui affiche la première variable (son nom et sa valeur).

```
          .MODEL SMALL
          .STACK 100H
          .CODE
          mov   ah,62h     ; chargement du numero
          int   21h        ; du segment du PSP
          mov   ds,bx
          ; chargement de ds avec le numero du segment
          ; contenant l'environnement
          mov   ax,[ds:2ch]
          mov   ds,ax
          ; boucle de recherche de la fin de la
          ; premiere variable d'environnement. Elle
          ; est reperee par un octet de valeur nulle
          mov   si,0
boucle1:  inc   si
          cmp   BYTE PTR [si],0
          jne   boucle1
          ; si contient l'offset du caractere nul.
          ; On met un '$' en fin de chaine pour pouvoir
          ; l'afficher avec l'interruption classique
          ; et on appelle l'interruption en question
          mov   BYTE PTR [si],'$'
          mov   ah,9
          int   21h
          ; on remet un caractere nul la ou on avait mis
          ; un '$'. (Cela ne sert a rien ici puisque le
          ; programme s'arrete.)
          mov   BYTE PTR [si],0
          ; fin du programme
fin:      mov   ah,4Ch
          int   21h
          END
```

Annexe C

Du bon usage des registres

Mis à part quelques registres très spécifiques dont l'emploi est imposé par le processeur (par exemple, le registre d'état est forcément le registre `flags`, le numéro du segment de code se trouve forcément dans le registre `cs`, ...), la plupart des registres peuvent être utilisés dans diverses circonstances. Il peut être utile d'indiquer ici quelques conventions.

- `ax`, `bx`, `cx`, `dx` : registres de données. Ils peuvent généralement être utilisés librement. Seuls `ax` et `cx` ont parfois un usage réservé (instructions `mul`, `imul`, `div`, `idiv` et les différentes versions de `loop`) ;
- `si`, `di` : offset de données, utilisés par certaines instructions de traitement de chaînes ;
- `sp` : pointeur de sommet de pile. Prudence extrême lors de sa manipulation (cf. appel de sous-programmes 4) ;
- `bp` : souvent utilisé pour sauvegarder `sp` en cas de besoins (cf. appel de sous-programmes 4) ;
- `ip` : compteur ordinal. Ne doit pas être modifié ;
- `cs` : segment de code. Ne pas le modifier ;
- `ds` : segment de données : segment dans lequel sont en général référencées automatiquement les données ;
- `es` : segment de données utilisé spécifiquement par les instructions de traitement de chaînes en liaison avec le registre `di`;
- `ss` : segment de pile. Ne pas le modifier *a priori* ;
- `flags` : registre d'état.

Annexe D

Le codage ASCII

On donne le code ASCII des caractères usuels. L'abscisse indique le chiffre, l'ordonnée la dizaine (en hexadécimal). Aussi, le code ASCII du caractère : est `3a`.

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
2.	␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Annexe E

Les messages d'erreur

Nous indiquons ici les messages d'erreur les plus fréquents qui peuvent survenir lors de l'assemblage d'un fichier.

Un message d'erreur est affiché sous la forme :

```
**Error** nom_de_fichier  (numéro_de_ligne)  message
```

`Illegal instruction`

Utilisation d'un mnémonique inexistant. Pax exemple `move` à la place de `mov`.

`Illegal immediate`

Utilisation d'une constante à un endroit interdit. Par exemple : `mov  4, ax`.

`Illegal indexing mode`

Utilisation d'un mode d'adressage indexé inexistant. Par exemple : `mov al, [si+ax]`. Les modes existants sont indiqués à la section 2.1.1.

`Illegal number`

Erreur dans une constante numérique.

`Illegal use of constant`

Utilisation d'une constante là où cela est interdit. Par exemple : `mov  ax+4, 5`.

`Invalid operand(s) to instruction`

Opérande non accepté par l'instruction.

`Labels can't start with numeric characteres`

Une étiquette doit forcément commencer par une lettre.

`Model must be specified first`

La première ligne d'un source assembleur, en dehors des commentaires, doit être `.model  small`.

`Need address or register`

Oubli d'un opérande. Par exemple : `mov  ax,`.

`Relative Jump out of range by xxx bytes`

L'étiquette en opérande d'une instruction de saut conditionnel est trop éloignée. Voir la section 3.2.2 à ce propos.

`Reserved Word used as a symbol`

Vous utilisez un mot réservé comme étiquette. Changez son nom.

`Symbol already defined: x`

Vous définissez deux fois la même étiquette de la même manière.

`Symbol already different kind`
Vous définissez deux fois la même étiquette de manière différente.

`Too few operands to instruction`
Vous avez oublié un opérande (ou deux). Par exemple : `mov  ax`.

`Undefined symbol`
Utilisation d'une étiquette qui n'a pas été définie.

`Unexpected end of file (no END directive)`
Votre source ne se termine pas par une directive `END`.

`Unknown character`
Un caractère inattendu a été rencontré dans une instruction.

`Value out of range`
Valeur incorrecte. Par exemple : `db  1024` alors qu'un octet a une valeur forcément inférieure à 256.

Index

Table des matières