Septième partie

Programmation en langage d'asemblage

Chapitre 26

Initiation aux langages d'assemblage

Nous avons étudié les fonctionnalités d'un microprocesseur, illustrées par l'étude du `i8086` programmé en langage machine. Il faut bien avouer que cela devient un peu pénible pour écrire un long programme.

On utilise donc un langage symbolique très proche du langage machine, appelé **langage d'assemblage**. Les programmes en langage d'assemblage s'écrivent grâce à un éditeur de texte, comme dans le cas d'un langage évolué tel que le langage C. On doit ensuite traduire le programme tel qu'il est écrit, dit **programme source**, en un **programme objet** en langage machine, en utilisant un **assembleur**. On ne parle pas de *compilateur* comme dans le cas du langage C, car la traduction étant plus simple, les principes de traduction ne sont donc pas les mêmes.

26.1 Notion et mise en place de l'assembleur

Nous avons vu jusqu'ici les réactions du microprocesseur grâce à sa programmation en langage machine. C'est très intéressant mais trop rudimentaire pour travailler sur un programme conséquent. Nous avons vu, d'autre part, comment éditer un texte et conserver ce texte dans un fichier (texte) grâce à un éditeur de texte. Nous avons également vu comment programmer un ordinateur moderne grâce à un langage évolué, tel que le langage C.

Nous allons voir ici une façon intermédiaire de faire entre le langage machine et un langage évolué. On définit un **langage d'assemblage**, très proche du langage machine, que l'on peut conserver dans un fichier texte, comme le programme source d'un programme en langage évolué. On le traduit en langage machine grâce à un **assembleur**.

26.1.1 Notion générale

Introduction.- Comme pour un langage évolué, tel que le langage C, pour écrire en langage d'assemblage, on commence par écrire un **programme**, dit plus exactement **programme source**, dans un langage d'assemblage particulier.

On utilise ensuite un logiciel particulier, appelé un **assembleur**, pour traduire ce programme source en quelque chose de plus compréhensible par l'ordinateur, appelé **programme objet**, qui est un **fichier binaire** (c'est-à-dire qui ne donne rien de compréhensible avec un éditeur de texte).

Le programme source peut contenir un sous-programme ou plusieurs, sans contenir un programme complet. On utilise donc un autre logiciel, appelé **lieur** (*linker* en anglais), pour rassembler les divers programmes objet en un **programme exécutable**, c'est-à-dire que son appel sur la ligne de commande (dans le cas d'un interpréteur de commande textuel ; par double clic sur son icône dans le cas d'un interpréteur de commande graphique) déclenchera l'exécution de ce qui est décrit dans le programme (source). Le lieur a également pour tâche de compléter les adresses non résolues à cause de la modularité.

Diversité des langages d'assemblage.- De même qu'il existe de nombreux langages naturels, il existe de nombreux langages d'assemblage. Il en existe au moins un par microprocesseur.

Diversité des assembleurs.- L'assembleur doit traduire un programme source, écrit dans un langage d'assemblage donné, en un programme objet. Il y a donc au moins un compilateur par langage d'assemblage. Le programme objet dépend du système informatique sur lequel on se trouve, essentiellement du microprocesseur mais aussi, un peu, du système d'exploitation. Il n'est donc pas suffisant de chercher un assembleur, il faut un assembleur adapté à tel ou tel système informatique. De plus, puisque les assembleurs sont souvent des **progiciels**, il existe souvent plusieurs assembleurs pour un système informatique donné, qui se font concurrence (tout au moins dans la mesure où le marché est porteur).

26.1.2 Les assembleurs pour PC et MS-DOS

Nous allons nous initier au langage d'assemblage à travers le langage d'assemblage pour le microprocesseur `Intel 8086`, et même plus spécifiquement pour un ordinateur dit « compatible PC » muni du système d'exploitation MS-DOS.

Les divers assembleurs.- La société IBM a demandé à Microsoft de concevoir un assembleur pour le premier IBM PC, tout simplement appelé **ASM** (pour *ASseMbler*), très proche du langage machine. Elle commercialise ensuite une version avec plus de directives, appelée *macro assembleur* ou MASM. Microsoft distribua ensuite cet assembleur, dont l'acronyme est quelquefois revu en *Microsoft ASseMbler*. Cet assembleur est toujours vendu avec certaines suites de développement.

Devant le succès de son compilateur phare, le *Turbo-Pascal*, la société Borland a conçu également un assembleur, appelé tout naturellement *Turbo Assembler*, utilisant la même syntaxe que MASM, à quelques variantes près, et plus rapide. Il n'est plus vendu depuis le milieu des années 1990.

Devant le succès de Linux et le développement des logiciels libres, un assembleur libre, également compatible avec la syntaxe de MASM, appelé **NASM** (pour *Net ASseMbler*) est disponible.

Il existe une autre syntaxe, dite *syntaxe ATT*, pour les assembleurs. C'est celle utilisée dans l'assembleur *gas* (pour *GNU ASsembler*), celui utilisé avec Linux.

Notre choix.- Nous utiliserons ici la syntaxe Microsoft et l'un des trois premiers assembleurs cités.

26.1.3 Installation de MASM

Nous utiliserons la version 5.1 de l'assembleur de Microsoft pour nos exemples.

Obtention.- On peut se rendre sur le site :

`http://www.phatcode.net/`

cliquer sur '`Download/Compiler`' et récupérer MASM 5.1.

Mise en place.- On obtient un fichier `masm510.zip` dont on extrait le contenu dans un répertoire MASM (sous Windows). On place ce répertoire et son contenu à la racine de la clé USB contenant notre MS-DOS. On change la variable PATH dans le fichier '`autoexec.bat`' :

```
PATH C:\DOS;C:\MASM\DISK1;C:\MASM\DISK4
```

Le mieux est de créer un répertoire ASM dans lequel on placera tous nos programmes écrits en langage d'assemblage.

26.1.4 Un premier exemple de programme en langage d'assemblage

Introduction.- Écrivons un programme permettant d'afficher la lettre 'a' à l'écran. Nous allons voir que nous suivons à peu près les mêmes étapes que pour écrire un programme en langage évolué.

Première étape : écrire le programme source.- Le programme source s'écrit grâce à un éditeur de texte, n'importe lequel, par exemple EDIT sous MS-DOS. Écrivons donc le programme suivant :

```
; affiche.asm
;
; affiche la lettre 'a' a l'ecran
;
CODESEG  SEGMENT
         ASSUME CS:CODESEG
DEBUT:
         mov dl,'a'     ; place le code ascii de 'a' dans dl
         mov ah,2h      ; appel de la fonction ms-dos d'affichage
                        ; d'un caractere
         int 21h        ; affiche le caractere se trouvant en dl
         mov ax, 4c00h  ; revient a ms-dos
         int 21h
CODESEG  ENDS
         END DEBUT
```

sans essayer de comprendre pour l'instant.

Deuxième étape : sauvegarder le programme.- Enregistrons le programme ci-dessus, par exemple sous le nom **affiche.asm**.

L'extension « asm » est traditionnelle pour les programmes source d'un langage d'assemblage.

Ce programme source peut être écrit avec n'importe quel éditeur de texte, par exemple l'éditeur de texte EDIT de MS-DOS (en utilisant le menu déroulant accessible par Alt-F), mais également sous Windows ou sous Linux et être placé après coup dans le répertoire ASM de la clé USB.

Troisième étape : l'assemblage.- On fait appel au logiciel MASM :

```
C:\ASM> masm affiche.asm
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

Object filename [affiche.OBJ]:
Source listing  [NUL.LST]:
Cross reference [NUL.CRF]:

  5096 + 397341 Bytes symbol space free

  0 Warning Errors
  0 Severe  Errors

C:\ASM>
```

Si tout se passe bien un nouveau fichier, dit **fichier objet**, se trouve dans le même répertoire, avec un nom dépendant du compilateur, ici `affiche.obj`.

Quatrième étape : le lieur.- Dans le cas général il faut lier entre eux plusieurs programmes objet :

```
C:\ASM> link affiche.obj
Microsoft (R) Overlay Linker  Version 3.64
Copyright (C) Microsoft Corp 1981-1988.  All rights reserved.

Run File [affiche.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK: Warning L4021: no stack segment

C:\ASM>
```

Si tout se passe bien un nouveau fichier se trouve dans le répertoire, de nom ici `affiche.exe`.

Cinquième étape : exécution du programme.- Ce nouveau fichier est en général un exécutable et, en le lançant (de façon habituelle, c'est-à-dire en écrivant son nom sur la ligne de commande puis en appuyant sur la touche de retour), on devrait voir apparaître 'a' à l'écran. C'est un logiciel du système d'exploitation qui permet cette exécution, il s'agit du **chargeur** (*loader* en anglais) :

```
C:\ASM> affiche
a
C:\ASM>
```

le nom du fichier sans extension est suffisant.

26.2 Structure d'un programme source en langage d'assemblage

Nous venons de voir comment écrire un programme simple en langage d'assemblage et comment obtenir un programme exécutable à partir de celui-ci. Nous avons écrit ce premier programme en langage d'assemblage sans nécessairement en comprendre la syntaxe. Nous allons revenir sur ce programme et le commenter.

26.2.1 Instructions et directives d'assemblage

Regardons à nouveau le programme que nous avons pris en exemple :

```
; affiche.asm
;
; affiche la lettre 'a' a l'ecran
;
CODESEG  SEGMENT
         ASSUME CS:CODESEG
DEBUT:
         mov dl,'a'     ; place le code ascii de 'a' dans dl
         mov ah,2h      ; appel de la fonction ms-dos d'affichage
                        ; d'un caractere
         int 21h        ; affiche le caractere se trouvant en dl
         mov ax, 4c00h  ; revient a ms-dos
         int 21h
CODESEG  ENDS
         END DEBUT
```

Beaucoup de lignes du programme nous sont immédiatement compréhensibles au vu de ce que nous avons appris avec le langage symbolique d'*Intel*. Il s'agit des lignes médianes que nous avons écrites en minuscule ; seuls les *commentaires*, qui suivant les points virgule ';' sont nouveaux. Ces lignes correspondent à des *instructions*.

Une **instruction** *est une ligne en langage d'assemblage dont la traduction en langage machine est immédiate.*

On comprend également le rôle des lignes blanches ou des lignes uniquement constituées d'un commentaire. Les autres lignes, possédant des indications nouvelles pour nous, sont des *directives d'assemblage*.

Une **directive d'assemblage** *n'est pas la simple traduction sous forme mnémotechnique de code machine. Elle est là pour indiquer à l'assembleur comment traiter certaines choses.*

Il s'agit des lignes 5 à 7 et des deux dernières lignes de notre programme.

26.2.2 Segments d'un programme

La différence essentielle entre un programme en langage d'assemblage et un programme en langage machine, au-delà des directives d'assemblage, est certainement la manipulation explicite des segments

26.2.2.1 Les segments

Notion de mémoire segmentée.- Pour éviter le chevauchement des différentes parties (données et code), les concepteurs des systèmes d'exploitation multi-tâches en sont venus à attribuer des parties bien déterminées de la mémoire vive à chacune des fonctions : programme, données et pile. On parle alors de **segmentation de la mémoire**.

Cas du 8086.- Le microprocesseur 8086 n'est pas vraiment destiné aux systèmes d'exploitation multi-tâches ; la segmentation y a été introduite, comme nous l'avons vu, pour des raisons de compatibilité avec le microprocesseur précédent d'*Intel*, le i8080/85.

Techniquement, pour un microprocesseur `i8086`, un **segment** est une partie de la mémoire de 64 KiO commençant à une adresse physique divisible par 16 (donc une adresse qui se termine par `0h`). Une telle adresse divisible par 16 est appelée un **début de paragraphe** (en anglais *paragraph boundary*).

Programmes .com et .exe.- Il existe deux types de programmes pour le `i8086` : les programmes pouvant tourner également sur `i8080/85`, dont l'extension est traditionnellement '.com' (lire *point-com*), et les autres, dont l'extension est traditionnellement '.exe' (lire *point-exe*).

Les programmes `.com` doivent tenir entièrement dans un segment (puisque la mémoire adressable d'un `i8080` est de 64 KiO).

Types de segments.- On distingue quatre types de segments pour un programme '.exe' :

- les **segments de code** (en anglais *code segment*) pour les instructions en langage d'assemblage traduisibles immédiatement en langage machine ;
- les **segments des données** (en anglais *data segment*) pour les données (*sic*) ;
- le **segment de pile** (en anglais *stack segment*) pour le contenu temporaire d'informations ;
- les **segments supplémentaires** (en anglais *extra segment*) sont traditionnellement réservés aux données initialisées, le plus souvent des chaînes de caractères.

Taille et adresse d'un segment.- Les **segments logiques** peuvent être confondus physiquement : il suffit qu'ils aient la même adresse.

Un segment est limité à 64 KiO. Le code d'un programme, par exemple, peut cependant nécessiter plus de 64 KiO. Il faut alors définir plusieurs segments de code et on devra passer de l'un à l'autre en l'indiquant explicitement.

Rappels sur adresse physique et adresse logique.- Nous avons déjà vu que l'adresse physique d'un élément de mémoire du 8086 (qui exige 20 bits) est déterminée par deux nombres de 16 bits : le contenu d'un *registre de segment* et un *décalage*. Ces deux valeurs déterminent l'*adresse logique*, notée `S:D` si `S` est le registre de segment et `D` le décalage.

Pour obtenir l'adresse physique à partir de l'adresse logique, il suffit de multiplier le contenu du registre de segment par 16 (ou `10h`, il s'agit donc d'un simple décalage à gauche) et de lui ajouter le décalage.

Il faut faire attention qu'il existe plusieurs adresses logiques différentes correspondant à une même adresse physique.

26.2.2.2 Directives de segmentation (1) : définition des segments

Syntaxe.- Les directives `SEGMENT` et `ENDS` indiquent à l'assembleur le début et la fin du contenu d'un segment :

```
etiquette  SEGMENT [options]
           ; contenu du segment
etiquette  ENDS
```

Les étiquettes du début et de fin du segment doivent être identiques, et être un identificateur permis du langage d'assemblage.

Options.- La directive `SEGMENT` peut être suivie de trois types d'options, l'*alignement*, l'*association* et la *classe*, codés suivant le format :

```
nom SEGMENT align combine 'class'
```

Type d'alignement.- Le type d'alignement indique à quel type d'endroit le segment doit commencer. Il s'agit en général de `PARA` pour indiquer qu'il doit commencer en début de paragraphe, c'est-à-dire à une adresse divisible par 16, ou `10h`. Il s'agit d'ailleurs du type par défaut.

Type d'association.- Le type d'association indique s'il faut associer ce segment avec d'autres segments lors du liage. Les types d'association sont `STACK`, `COMMON`, `PUBLIC`, `AT` et `NONE`.
On définit, par exemple, le segment de pile de la façon suivante :

```
nom SEGMENT PARA STACK
```

de façon à n'obtenir qu'une seule pile pour le programme.

Type de classe.- Le type de classe, indiqué entre apostrophes verticales, est utilisé pour regrouper les segments associés lors du liage.

On peut, par exemple, utiliser `'code'` pour le segment de code (recommandé par Microsoft), `'data'` pour le segment de données et `'stack'` pour le segment de pile.

Pour reprendre l'exemple précédent, on écrira souvent :

```
nom SEGMENT PARA STACK 'stack'
```

26.2.2.3 Directives de segmentation (2) : détermination des segments

Notion.- Le nom que nous avons donné à notre seul segment nous désigne clairement sa fonction, il s'agit d'un segment de code, mais ceci n'est évidemment pas compréhensible par l'assembleur. Il faut donc une directive pour indiquer la fonction de chaque segment.

De plus il peut y avoir plusieurs segments d'un même type. Il faut donc, suivant l'endroit où on se trouve, indiquer clairement de quel segment on s'occupe à ce moment-là.

Syntaxe.- La directive `ASSUME` initialise les valeurs des registres `CS`, `DS`, `SS` et `ES` en leur attribuant le nom du segment correspondant grâce à l'opérateur ':', suivant le modèle suivant :

```
ASSUME SS:stackname, DS:datasegname, CS:codesegname
```

On peut aussi écrire :

```
ASSUME SS:stackname, DS:datasegname, CS:codesegname, ES:NOTHING
```

pour préciser qu'on n'utilise pas, dans le cas pris en exemple, de segment supplémentaire.

Cas des segments de données et de pile.- La directive `ASSUME` est suffisante pour déterminer les segments de code et les segments supplémentaires. Mais, puisque `DS` et `SS` n'acceptent pas la valeur d'un registre, il faut ajouter, par exemple, les lignes suivantes :

```
MOV AX,datasegname
MOV DS,AX
```

pour déterminer le segment de données.

Emplacement des segments.- Quels sont les emplacements réels (ou absolus) des segments en mémoire vive ? Ceci est le problème du *chargeur*, lié au système d'exploitation. Ces emplacements peuvent varier d'un chargement du programme à l'autre.

Bien entendu, on pourrait placer les segments à la main mais ce n'est pas ce qui est fait habituellement.

26.2.3 Présentation d'un programme d'assemblage

26.2.3.1 Commentaires

En langage d'assemblage tout ce qui suit le caractère point-virgule ';' jusqu'à la fin de la ligne est un **commentaire** dont l'assembleur ne tient pas compte. On aurait donc aussi bien pu écrire le programme précédent sous la forme suivante :

```
CODESEG  SEGMENT
         ASSUME CS:CODESEG
DEBUT:
         mov dl,'a'
         mov ah,2h
         int 21h
         mov ax,4c00h
         int 21h
CODESEG  ENDS
         END DEBUT
```

26.2.3.2 Indentations

Les indentations et les lignes blanches dans le programme ne sont là que pour le rendre plus compréhensible pour un être humain. Elles ne sont pas nécessaires pour l'assembleur. On aurait pu écrire le programme précédent sous la forme suivante :

```
CODESEG  SEGMENT
ASSUME CS:CODESEG
DEBUT:
mov dl,'a'
mov ah,2h
int 21h
mov ax,4c00h
int 21h
CODESEG  ENDS
         END DEBUT
```

26.2.3.3 Passage à la ligne

Contrairement à ce qui se passe pour un langage évolué, tel que le langage C, la mise en forme ne peut pas être tout à fait quelconque. Le passage à la ligne, par exemple, est obligatoire après chaque instruction. Nous avons vu que le point-virgule est le début d'un commentaire, celui-ci allant jusqu'à la fin de la ligne (il n'y a pas d'autre indicateur de fin de commentaire).

26.2.3.4 Minuscule et majuscule

L'assembleur MASM ne fait pas la différence entre la minuscule et la majuscule correspondante. Nous aurions pu écrire notre programme, par exemple :

```
CODESEG  SEGMENT
         ASSUME CS:CODESEG
DEBUT:
         MOV DL,'a'
         MOV AH,2H
         INT 21H
         MOV AX,4C00H
         INT 21H
CODESEG  ENDS
         END DEBUT
```

26.2.4 Programme et DOS

Un programme écrit en langage d'assemblage interagit avec le système d'exploitation. Il faut en particulier lui préciser quel est le point de départ du code et ce qu'il faut faire lorsque l'exécution du programme est terminée.

26.2.4.1 Point d'entrée du programme

Notion.- On ne veut pas nécessairement commencer par la première ligne de code. Nous en verrons des exemples naturels plus tard. Il faut donc indiquer au système d'exploitation, et plus exactement au chargeur, quelle doit être la première ligne de code à exécuter.

Syntaxe.- Un programme en langage d'assemblage se termine par la directive `END` de format :

```
END identificateur
```

où `identificateur` est l'une des étiquettes ou des noms de procédure utilisés dans le segment de code.

Programme sans point d'entrée.- Le point d'entrée est facultatif puisque certains programmes ne sont pas destinés à être exécuter. Il s'agit de portions de programme à lier ultérieurement.

26.2.4.2 Retour au Dos

Nous avons vu que notre programme se termine par l'appel à la fonction 4C00h de l'interruption 21h :

```
MOV AX,4C00H
INT 21H
```

Le gestionnaire de cette interruption indique comment revenir au DOS.

26.2.5 Exemples de programmes utilisant plusieurs segments

Donnons quelques exemples de programmes utilisant plusiers segments.

26.2.5.1 Programme avec segment de données

26.2.5.2 Programme avec segment de pile

26.2.5.3 Programme avec segment supplémentaire

26.2.5.4 Programme avec plusieurs segments de code

26.2.6 Structure d'un programme

Conformément à l'exemple que nous venons de voir et de commenter, un programme est composé :

- d'instructions, traductions immédiates sous forme mnémotechnique des instructions du langage machine ;

- de commentaires ;

- de directives d'assemblages.

La structure d'un programme en langage d'assemblage Microsoft `i8086` est la suivante :

```
page 60, 132
TITLE essai.asm squelette de programme
;----------------------------------------
STSEG SEGMENT
      DB 64 DUP (?)
STSEG ENDS
;----------------------------------------
DTSEG SEGMENT
; declaration des variables ici
DTSEG ENDS
;----------------------------------------
CDSEG SEGMENT
MAIN  PROC   FAR
      ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG
      MOV    AX,DTSEG
      MOV    DS,AX
      ;
      ; code proprement dit ici
      ;
      mov   ax,4c00h ;revient a ms-dos
      int   21h
MAIN  ENDP
CDSEG ENDS
      END   MAIN
```

26.2.7 Définition simplifiée des segments

La façon dont nous avons défini les segments dans notre exemple de programme est maintenant appelée la **définition complète des segments**. En effet, d'autres directives ont été introduites plus tard, ce qui donne lieu à la **définition simplifiée des segments**, disponible depuis la version 5.0 de MASM et la version 1 de TASM.

Exemple.- Le programme ci-dessus se réécrit de la façon suivante avec la définition simplifiée des segments :

```
; affiches.asm
; affiche la lettre 'a' a l'ecran
      .model small
      .stack 100h
      .data
DATA1 DB 52h
DATA2 DB ?
      .code
MAIN :
      mov dl,'a'   ;place le code ascii de 'a' dans dl
      mov ah,2h    ;appel a la fonction ms-dos d'affichage
                   ;d'un caractere
      int 21h      ;affiche le caractere qui se trouve en dl
      mov ax,4c00h ;revient a ms-dos
      int 21h
      end MAIN
```

Modèle de mémoire.- On commence par indiquer le **modèle de mémoire**, c'est-à-dire le nombre de segments physiques distincts utilisés.

Les options pour le modèle de mémoire sont SMALL, MEDIUM, COMPACT, LARGE et HUGE :

- **SMALL MODEL** : utilise 64 KiO de mémoire pour le code et 64 KiO pour les données. C'est le modèle de mémoire le plus souvent utilisé.
- **MEDIUM MODEL** : les données doivent tenir dans 64 KiO de mémoire mais le code peut occuper plus de 64 KiO.
- **COMPACT MODEL** : est l'opposé du modèle MEDIUM. Le code doit tenir dans 64 KiO de mémoire mais les données peuvent occuper plus de 64 KiO.
- **LARGE MODEL** : le code et les données peuvent occuper plus de 64 KiO mais aucun contenu de variable (y compris de tableau) ne peut dépasser 64 KiO.
- **HUGE MODEL** : le code et les données peuvent occuper plus de 64 KiO, y compris le contenu d'une seule variable.

Il existe un autre modèle de mémoire, le modèle **TINY**. La capacité mémoire totale utilisée pour le code et les données ne peut pas dépasser 64 Ko. Ce modèle est lié aux programmes '.com'. Il ne peut peut pas être utilisé avec la définition simplifiée des segments.

Indication du modèle.- Le modèle est indiqué grâce à la directive ".MODEL" avec l'un des modèles pour paramètre. Dans notre exemple nous avions :

```
.model small
```

Cette directive engendre automatiquement les directives `ASSUME` nécessaires.

Définition des segments.- La définition simplifiée des segments utilise trois directives simples pour définir chacun des segments `CS`, `DS` et `SS`, à savoir ".CODE", ".DATA" et ".STACK" :

- Pour la pile, il suffit d'indiquer la capacité mémoire de celle-ci en KiO. Dans notre exemple nous avions :

  ```
  .stack 100h
  ```

  ou :

  ```
  .stack 64
  ```

- La déclaration des variables se fait suivant des règles que nous verrons plus tard entre la directive :

  ```
  .data
  ```

  et la directive :
  `.code [nom]`
- Le code proprement dit suit cette dernière directive.

Les noms de ces segments sont, par défaut, `STACK`, `_DATA` et `_TEXT`. On peut remplacer le nom par défaut du segment de code.

La taille par défaut de la pile est de 1 024 octets.

Forme d'un programme.- Un programme avec la définition simplifiéee des segments se présente donc sous la forme suivante :

```
; commentaires eventuels
      .model small
      .stack 100h
      .data
; declaration des variables ici

      .code
MAIN :
      ;
      code proprement dit ici
      ;
      mov ax,4c00h ;revient a ms-dos
      int 21h
      end MAIN
```

26.2.8 Définition des segments

26.2.8.1 Définition du segment de pile

La définition du segment de pile contient seulement une ligne :

```
DB 64 DUP (?)
```

Cette directive réserve ici 64 octets de mémoire pour la pile. Nous expliquerons plus en détail les directives 'DB', 'DUP' et '(?)' au moment de la déclaration des variables. Remarquons dès à présent que, sauf indication contraire, les nombres sont en base dix et non en base seize, contrairement à `debug`.

26.2.8.2 Définition du segment des données

Un peu de la même façon que pour la définition du segment de pile, nous déclarons ici deux variables. Les étiquettes 'DATA1' et 'DATA2' correspondent à leurs noms ; ce sont des identificateurs. Comme nous le verrons 'DB' est le type, il s'agit ici d'octets (*Define Byte*). La première variable est initialisée à la valeur `52` en hexadécimal alors que la seconde n'est pas initialisée.

26.2.8.3 Définition du segment de code

Procédures.- Le code est une suite de définition de **procédures**, c'est-à-dire de sous-programmes. Dans notre exemple, il y a une seule procédure, la procédure 'MAIN'.

Syntaxe.- Une procédure commence par une étiquette (qui est un identificateur non utilisé pour autre chose), suivie de la directive 'PROC' (pour *PROCedure*) et éventuellement d'options (comme ici de 'FAR') et se termine par la même étiquette suivie de 'ENDP' (pour *END Procedure*) :

```
procname  PROC [options]
          .
          .
          .
procname  ENDP
```

Point d'entrée du programme.- L'assembleur doit savoir où commencer, c'est-à-dire par quelle procédure commencer (qui fait éventuellement appel à d'autres). Ceci est indiqué par la dernière ligne du programme qui comprend la directive 'END' suivi de l'étiquette de la *procédure principale*.

Le système d'exploitation DOS exige que la procédure principale ait l'option 'FAR'.

Options.- Les deux options principales sont `FAR` (si le code de la procédure n'est pas situé dans le même segment que le code appelant) et `NEAR`. Par défaut l'option est `NEAR`.

26.3 Les fichiers utilitaires de l'assembleur

Nous avons vu que l'assembleur fournit un fichier `.obj` et un fichier `.exe`, le fichier exécutable étant le plus important pour nous. L'assembleur peut aussi fournir deux autres fichiers : un **listing** reprenant le programme source, sa traduction en langage machines et quelques autres renseignements et un **fichier de références croisées**.

26.3.1 Listing

26.3.1.1 Obtention

Obtention du listing.- Nous avons vu que MASM nous demande le nom du fichier de listing. Si on utilise l'option par défaut, 'NUL.LST', on n'obtient rien. Si on met `affiche.lst`, on obtient un nouveau fichier, portant ce nom.

Exemple.- Dans notre cas, le contenu de ce fichier est :

```
Microsoft (R) Macro Assembler Version 5.10                  2/3/13 16:02:26

                                                             Page     1-1


                            ; affiche.asm
                            ;
                            ; affiche la lettre 'a' a l'ecran
                            ;
 0000                       CODESEG  SEGMENT
                                     ASSUME CS:CODESEG
 0000                       DEBUT:
 0000  B2 61                         mov dl,'a'      ; place le code ascii de 'a' dans dl

 0002  B4 02                         mov ah,2h       ; appel de la fonction ms-dos d'affichage
                                                     ; d'un caractere

 0004  CD 21                         int 21h         ; affiche le caractere se trouvant en dl

 0006  B8 4C00                       mov ax, 4c00h   ; revient a ms-dos
 0009  CD 21                         int 21h
 000B                       CODESEG  ENDS
                                     END DEBUT
Microsoft (R) Macro Assembler Version 5.10                  2/3/13 16:02:26

                                                             Symbols-1


Segments and Groups:

                N a m e                 Length  Align   Combine Class

CODESEG  . . . . . . . . . . . .        000B    PARA    NONE

Symbols:

                N a m e                 Type    Value   Attr

DEBUT  . . . . . . . . . . . . .        L NEAR  0000    CODESEG

@CPU . . . . . . . . . . . . . .        TEXT  0101h
@FILENAME  . . . . . . . . . . .        TEXT  affiche
@VERSION . . . . . . . . . . . .        TEXT  510


     18 Source  Lines
     18 Total   Lines
      7 Symbols

  48026 + 397361 Bytes symbol space free
```

```
    0 Warning Errors
    0 Severe  Errors
```

Commentaires.- 1^o) Le listing est supposé être envoyé sur une imprimante (*lister* en anglais), d'où son nom et la présence d'un certain nombre de caractères d'échappement non nécessairement compréhensibles par un éditeur de textes : les tabulations et surtout le passage à la ligne sous la forme d'un `CTRL-L`. Ici il y a un tel passage à la ligne avant le deuxième `Microsoft`, dont nous n'avons pas tenu compte.

2^o) Chaque page commence par deux lignes de la forme :

```
Microsoft (R) Macro Assembler Version 5.10                  2/3/13 16:02:26

                                                            Page 1 - 1
```

indiquant le nom de l'assembleur, sa version, ainsi que la date et l'heure de l'assemblage pour la première ligne, le numéro de page pour la seconde ligne.

3^o) Il y a visiblement deux sortes de pages : les premières (une seule dans notre exemple) reprennent le code avec des indications supplémentaires, les secondes (pages dénommées `Symbols`) commentent les symboles utilisés en langage d'assemblage.

4^o) Les premières pages ont visiblement trois colonnes : la première colonne indique le décalage de l'adresse en mémoire vive dans le segment adéquat, la seconde colonne indique le contenu de cet emplacement en hexadécimal, la troisième colonne comprend le code source.

5^o) Nous avons déjà vu comment traduire les instructions en code machine. Il y a donc peu à dire sur le segment de code.

6^o) Aucune directive ne produit de code, ni même d'emplacement mémoire nouveau. C'est normal puisque ce sont des messages à l'assembleur.

7^o) Les pages concernant les symboles donnent des renseignements sur les segments, les procédures et les symboles.

8^o) On rappelle le nom de chaque segment, il n'y en a qu'un seul ici, sa longueur, la façon dont il doit être aligné et sa classe.

Les segments ne sont pas listés dans l'ordre de leur définition mais dans l'ordre alphabétique. De plus les noms sont systématiquement en majuscule.

9^o) Suit la **table des procédures**. Celle-ci est compréhensible : on donne chaque nom de procédure (dans l'ordre alphabétique), son type (ici `L NEAR` pour procédure proche), sa valeur (c'est-à-dire l'adresse par rapport au début du segment de code), son **attribut**, c'est-à-dire le segment dans lequel elle est définie.

26.3.1.2 Directives de listing

On peut utiliser deux directives pour formater le listing.

Caractéristiques de la page.- Au début du programme, la directive `PAGE` permet de déterminer le nombre maximum de lignes sur une page et le nombre maximum de caractères sur une ligne. Sa syntaxe est :

```
PAGE [longueur][,largeur]
```

Par exemple `PAGE 60,132` donne 60 lignes par page et 132 caractères par ligne.

Par défaut on a `PAGE 50,80`.

Si on veut forcer le listing à produire une nouvelle page à un endroit (du source) donné, on utilise tout simplement la directive `PAGE` (sans paramètres) à cet endroit.

TITRE.- Si on utilise :

```
TITLE text [comment]
```

ce qui suit `TITLE` sera écrit chaque deuxième ligne de page du listing.

26.4 Forme d'un programme en langage d'assemblage

26.4.1 Identificateurs

Notion.- Nous avons vu la notion d'**identificateur** lors de l'étude d'un langage évolué (le langage C, par exemple). Ces identificateurs nous servent pour nommer les variables, les sous-programmes, les constantes,...

Il n'y a pas d'identificateur en langage machine (il n'y a que des adresses) mais on a intérêt à en utiliser en langage d'assemblage.

Remarque.- Rappelons que le langage d'assemblage est une création totalement libre. Sa syntaxe dépend donc de l'assembleur qui est utilisé. Nous suivons ici les règles de Microsoft, concepteur de MASM, règles également suivies par TASM et NASM.

Alphabet.- L'*alphabet* du langage d'asssemblage Microsoft du `i8086` est constitué des vingt-six lettres de l'alphabet latin, en minuscule ou majuscule, des dix chiffres de '0' à '9' et des cinq caractères spéciaux suivants : le point d'interrogation '?', le point '.', le '@', le blanc souligné '_ ' et le dollar '$'. L'assembleur ne fait pas de différence entre une minuscule et la majuscule correspondante.

Identificateur.- Les *identificateurs* sont les mots de moins de trente-et-un caractères sur l'alphabet (247 depuis MASM 6.0) ci-dessus répondant aux règles suivantes :

- ce ne sont pas des mots rérvés ;
- le premier caractère n'est ni un chiffre, ni un point.

Mots réservés.- Les mots réservés sont les suivants :

```
Noms de registres :

AH AL AX BH BL BP BX CH CL CS CX DH DI DL DS DX EAX EBP EBX ECX
EDI EIP ES ESI FS GS IP SI SP SS

Mnémonymes d'instructions :

AAA         AAD         AAM         AAS
ADC         ADD         AND         ARPL
BOUND       BSF         BSR         BTn
CALL        CBW         CDQ         CLC
CLD         CLI         CLTS        CMC
CMP         CMPSn       CMPXCHG     CMPXCHG8B
CWDn        DAA         DAS         DEC
DIV         ENTER       ESC         HLT
IDIV        IMUL        IN          INC
INSn        INT         INTO        IRET
JA          JAE         JB          JBE
JCXZ        JE          JECXZ       JG
JGE         JL          JLE         JMP
JNA         JNAE        JNB         JNBE
JNE         JNG         JNGE        JNL
JNLE        JNO         JNP         JNS
```

```
JNZ         JO          JP          JPE
JPO         JS          JZ          LAHF
LAR         LDS         LEA         LEAVE
LES         LFS         LGDT        LGS
LIDT        LLDT        LMSW        LOCK
LODSn       LOOP        LOOPE       LOOPNEn
LOOPNZn     LOOPZ       LSL         LSS
LTR         MOV         MOVSn       MOVSX
MOVZX       MUL         NEG         NIL
NOP         NOT         OR          OUTn
POP         POPA        POPAD       POPF
POPFD       PUSH        PUSHAD      PUSHF
PUSHFD      RCL         RCR         REN
REP         REPE        REPNE       REPNZ
REPZ        RET         RETF        ROL
ROR         SAHF        SAL         SAR
SBB         SCASn       SETnn       SGDT
SHL         SHLD        SHR         SHRD
SIDT        SLDT        SMSW        STC
STD         STI         STOSn       STR
SUB         TEST        VERR        VERRW
WAIT        XADD        XCHG        XLAT
XOR

Directives :

$           *           +           -
.           /           =           ?
[           ]
ALIGN       ASSUME      BYTE        COMM
COMMENT     DB          DD          DF
DOSSEG      DQ          DS          DT
DW          DWORD       ELSE        END
ENDIF       ENDM        ENDP        ENDS
EQU         EVEN        EXITM       EXTRN
EXTERN      FWORD       GROUP       IF
IF1         IF2         IFB         IFDEF
IFDIF       IFE         IFIDN       IFNB
IFNDEF      INCLUDE     INCLUDELIB  IRP
IRPC        LABEL       LOCAL       MACRO
NAME        ORG         PAGE        PROC
PUBLIC      PURGE       QWORD       RECORD
REPT        REPTRD      SEGMENT     STRUC
SUBTTL      TBYTE       TITLE       TWORD
UNION       WORD
.186        .286        .286P       .287
.386        .386P       .387        .8086
.8087       .ALPHA      .CODE       .CONST
.CREF       .DATA       .DATA?      .ERR
.ERR1       .ERR2       .ERRB       .ERRDEF
```

```
.ERRDIF     .ERRE       .ERRIDN     .ERRNB
.ERRNDEF    .ERRNZ      .FARDATA    .FARDATA?
.LALL       .LFCOND     .LIST       .MODEL
.OUT        .RADIX      .SALL       .SEQ
.SFCOND     .STACK      .TFCOND     .TYPE
.XALL       .XCREF      .XLIST

Opérateurs :

AND         BYTE        COMMENT     CON
DUP         EQ          FAR         GE
GT          HIGH        LE          LENGTH
LINE        LOW         LT          MASK
MOD         NE          NEAR        NOT
NOTHING     OFFSET      OR          PTR
SEG         SHL         SHORT       SHR
SIZE        STACK       THIS        TYPE
WHILE       WIDTH       WORD        XOR

Identificateurs prédéfinis :

@Data       @Model
```

26.4.2 Syntaxe d'une instruction de langage d'assemblage

Syntaxe.- La forme générale d'une instruction du langage d'assemblage (Microsoft, rappelons-le) est constituée des quatre **champs** suivants :

```
[Étiquette :] Opération [Opérandes] [;Commentaire]
```

où les crochets '[' et ']' indiquent que le champ en question n'apparaît pas toujours. À part dans une ligne de pur commentaire, le champ `opération` est toujours présent.

Le nombre d'opérandes (zéro, un ou deux) dépend de l'opération. Lorsqu'il y a deux opérandes, ceux-ci sont séparés par une virgule ','. L'espace après la virgule n'est pas obligatoire mais il améliore quelquefois la lisibilité.

Il y a un maximum de 132 caractères par ligne (512 depuis MASM 6.0).

Exemples.- Écrivons quelques instructions :

```
L1 : cmp bx, cx ;compare bx   tous les champs sont presents
     add ax,25                operation et deux operandes
     inc bx                   operation et un operande
     ret                      operation seule
;commentaire                  commentaire seul
```

Remarque.- L'étiquette, l'opération, les opérandes et les commentaires peuvent commencer à n'importe quelle colonne mais, pour des raisons de lisibilité, toutes les étiquettes commencent en général à la même colonne. Il en est de même de toutes les opérations, de tous les premiers opérandes, de tous les second opérandes et de tous les commentaires.

26.4.3 Les nombres entiers en langage d'assemblage

Bases utilisées en langage d'assemblage.- Nous avons vu que, dans le langage symbolique de `debug`, on ne peut utiliser que la base seize. En langage d'assemblage, il est traditionnel d'utiliser les entiers exprimés en base dix (*numération décimale*) certes mais aussi en base deux (*numération binaire*, qui correspond au langage machine) ou en base seize (*numération hexadécimale*), bien entendu parce que cette dernière base est liée à la base deux en plus lisible pour nous.

Écriture des entiers.- Pour savoir dans quelle base est exprimé un entier on utilise les conventions suivantes en langage d'assemblage MASM :

- un entier en base dix est écrit de la façon habituelle, par exemple `100` ou `23` ; on peut lui ajouter le suffixe 'd' pour insister, par exemple `100d` ;

- un entier en base deux est écrit en le faisant suivre du symbole 'b' ou 'B' (évidemment pour *binaire*, ou *binary* en anglais), par exemple `100b` ou `101B` ;

- un entier en base seize est écrit en utilisant les chiffres successifs '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a' ou 'A' (pour dix), 'b' ou 'B' (pour onze), 'c' ou 'C' (pour douze), 'd' ou 'D' (pour treize), 'e' ou 'E' (pour quatorze) et 'f' ou 'F' (pour quinze) suivi d'un 'h' ou d'un 'H' (pour *hexadécimal*), par exemple `100h`, `23h` ou `1ah`.

Remarque.- Il y a ambiguïté entre le registre `ah` et l'entier hexadécimal `ah`. Pour lever celle-ci, on fait précéder l'entier du chiffre '0', en l'écrivant donc `0ah`. Plus généralement une constante entière commençant par un des chiffres 'A' à 'F' peut être confondue avec un identificateur. On convient donc de la faire précéder du chiffre '0'.

26.4.4 Initialisation d'un registre avec un caractère

Syntaxe.- On peut initialiser un registre avec un caractère. Pour cela, on utilise l'instruction `mov` mais `source` est un caractère ASCII placé entre apostrophes (apostrophes verticales, la même à droite et à gauche).

Exemple.- L'instruction :

```
mov ax, 'b'
```

permet de placer le caractère 'b' dans le registre `ax`.

Remarque.- En fait ce n'est pas le caractère 'b' qui est placé dans le registre mais un entier en binaire, son **code** ASCII. L'assembleur est chargé de cette traduction.

Quels sont les caractères permis ?.- Le **jeu de caractères** est défini par le code choisi. En ce qui concerne l'assembleur Microsoft, on utilise le *code ASCII*.

26.4.5 Accès à la mémoire vive grâce aux variables

Notion de variable.- Jusqu'à maintenant nous avons utilisé les registres, c'est-à-dire la mémoire du microprocesseur lui-même, pour conserver des données. Pour beaucoup d'applications, nous avons besoin de plus d'emplacement mémoire ; on utilise alors la mémoire vive.

Nous avons déjà vu comment accéder à la mémoire vive en utilisant des **adresses absolues** en langage machine. Cette façon de faire n'est pas toujours satisfaisante, car nous risquons d'accéder à un emplacement mémoire réservé pour autre chose, d'où l'intérêt d'utiliser des **variables** (dont l'emplacement mémoire exact est attribué par le système d'exploitation au moment du chargement du programme exécutable).

Les variables en langage d'assemblage.- Les variables sont définies par des directives. Les règles varient donc d'un assembleur à l'autre. Une fois de plus nous nous intéresserons ici aux directives Microsoft pour la déclaration des variables en langage d'assemblage pour le `i8086`, devenues un des standards de fait, suivi par MASM, TASM et NASM.

Emplacement de la déclaration d'une variable.- Nous avons déjà vu, lors de l'étude des segments, que les variables se déclarent dans le segment des données.

Nom d'une variable.- Toute variable à laquelle on fera référence possède un nom, qui est un identificateur non utilisé pour autre chose. Nous avons déjà vu les règles de formation des identificateurs.

Types des variables.- Il y a cinq types simples de variables, suivant la place occupée en mémoire :

- le type octet **DB** (pour *Define Byte*) qui occupe un octet ;
- le type mot **DW** (pour *Define Word*) qui occupe deux octets ;
- le type mot double **DD** (pour *Define Doubleword*) qui occupe quatre octets ;
- le type mot quadruple **DQ** (pour *Define Quadword*) qui occupe huit octets ;
- le type dix octets **DT** (pour *Define Ten bytes*) qui occupe dix octets.

MASM 6.0 a aussi introduit les termes `BYTE`, `WORD`, `DWORD`, `QWORD` et `TWORD` respectivement pour ces directives.

Déclaration d'une variable.- La syntaxe de la déclaration d'une variable est la suivante :

```
NOM TYPE VALEUR Commentaire
```

où `NOM` est le nom de la variable, `TYPE` l'un des cinq types ci-dessus et `VALEUR` la valeur initiale de la variable.

Initialisation des variables lors de la déclaration.- Si on ne veut pas initialiser la variable lors de la déclaration, il suffit de mettre le symbole ' ?'. Sinon on peut mettre soit un entier en décimal (suivi éventuellement de 'D' ou 'd'), en binaire (suivi de 'B' ou 'b'), en octal (suivi de 'Q' ou 'q') ou en hexadécimal (suivi de 'H' ou 'h'), soit un mot placé entre apostrophes verticales ou entre guillemets verticaux.

Bien entendu la valeur doit être compatible avec le type (donc la taille de la mémoire).

On a par exemple :

```
DATA1  DB ?       ; variable non initialisee
DATA2  DB 23      ; decimal
DATA3  DB 35d     ; decimal
DATA4  DB 100101b ; binaire
DATA5  DB 15h     ; hexadecimal
DATA6  DB 0A8h    ; hexadecimal
DATA7  DB 'a'     ; caractere
DATA8  DW 961     ; decimal
DATA9  DW 'ah'    ; caracteres
DATA10 DD 2314    ; decimal
DATA11 DD "L'e"   ; interet des apostrophes
```

Puisque les lettres 'D' et 'B' sont à la fois des chiffres hexadécimaux et des indications de base de numération, MASM 6.0 a introduit les lettres `T` (pour *Ten*) et `Y` (pour *binarY*) pour les bases dix et deux, respectivement.

Tableaux.- On peut aussi définir des variables de type tableau.

- 1^o) La première méthode consiste à initialiser une variable d'un type simple avec un contenu plus grand que celui qui est permis *a priori* :

```
DATA1 DB '2438Bonjour'
```

éventuellement en utilisant des virgules si les formats sont différents :

```
DATA2 DB 12, 23, 54h, 1010b, 'Bonjour'
```

On est toutefois limité par la longueur de la ligne.

- 2) Pour remplir le tableau avec un certain nombre de fois la même valeur d'octet, on peut utiliser la directive **DUP** (pour *DUPlicate*) :

```
DATA DB 7 DUP('a')
```

qui déclare un tableau de 7 octets, chaque octet étant initialisé par la lettre 'a'.

La façon la plus propre de déclarer un tableau (d'octets) est évidemment la suivante :

```
TAB DB 500 DUP(?)
```

qui réserve de la place pour un tableau de 500 octets, non initialisés.

- 3^o) La référence au premier élément d'un tableau, en fait celui de numéro 0, se fait grâce au nom de ce tableau. Par exemple :

```
mov al, DATA1
```

placera le caractère '2' dans `al`.

On utilise le symbole '+' plus une constante pour les autres valeurs. Par exemple :

```
mov al, DATA1+2
```

placera le caractère '3' dans `al`.

Adresse absolue d'une variable.- En langage machine, il faut spécifier l'adresse absolue d'une 'variable', en fait le décalage. Nous avons remarqué que cela peut être dangereux et qu'il vaut mieux laisser faire le système d'exploitation. C'est ce qui se passe en général avec un langage d'assemblage.

Il arrive cependant que l'on veuille une adresse particulière (cas de la manipulation des périphériques, par exemple). On peut alors spécifier l'adresse absolue en utilisant la directive **ORG** (pour *ORiGin*). Par exemple :

```
ORG 101h
DATA DB 'a'
```

placera (le code ASCII de) la lettre 'a' à l'adresse `101h`.

Connaissance de l'adresse absolue.- En général on ne connaît pas l'adresse absolue d'une variable. Si on a vraiment besoin de l'adresse (en fait le décalage du premier octet) d'une variable, on peut utiliser la directive **OFFSET**. Par exemple :

```
mov si, offset DATA1
```

place l'adresse (d'offset) de la variable `DATA1` dans le registre `SI`.

26.4.6 Les constantes en langage d'assemblage

Notion de constante en langage d'assemblage.- En langage évolué une **constante** est une variable, qui occupe un emplacement de la mémoire vive, dont on ne peut pas changer la valeur. Ceci n'est pas le cas pour un langage d'assemblage.

Dans un langage d'assemblage une **constante** est un identificateur auquel on attribue une valeur. Lors de l'assemblage, toute occurrence de cet identificateur est remplacée par cette valeur. Une constante n'occupe donc pas d'emplacement mémoire lors de l'exécution.

Emplacement de la déclaration des constantes.- Comme il s'agit d'une directive, la déclaration d'une constante peut se trouver n'importe où, à l'intérieur du segment des données, ou dans un autre segment ou à l'extérieur de tout segment mais elle doit précéder l'utilisation de la constante.

Déclaration.- On utilise la directive **EQU** (pour *EQUate*). Par exemple :

```
TVA EQU 20
```

Ceci signifie que l'assembleur remplacera chaque occurrence de 'TVA' par 20.

La valeur peut être une expression (constante) :

```
TOTALVENTES DW 0
...
TV EQU TOTALVENTES
```

Cas de chaînes de caractères.- MASM 6.0 a introduit la directive **TEXTEQU** pour le cas des chaînes de caractères :

```
nom TEXTEQU <texte>
```

26.4.7 Les structures de contrôle

Nous avons vu les structures de contrôle disponibles sur le microprocesseur `i8086` en étudiant son langage machine. Il s'agit ici de voir comment les utiliser dans un langage d'assemblage, en particulier en remplaçant une adresse absolue par une *étiquette*.

26.4.7.1 Saut inconditionnel

Nous avons vu la notion de *saut inconditionnel* lors de l'étude du langage machine. Nous avons remarqué alors la difficulté que constitue le calcul de l'adresse. Nous allons voir que remplacer une adresse absolue par une étiquette nous facilite grandement la vie.

Syntaxe.- La forme d'un saut inconditionnel est :

```
- - - - - - - - - - - -
etiquette :
- - - - - - - - - - - -
           jmp etiquette
- - - - - - - - - - - -
```

où `etiquette` est une **étiquette**, c'est-à-dire un identificateur non déjà utilisé pour autre chose.

Intérêt des étiquettes.- Une étiquette permet de repérer l'instruction à laquelle on veut aller, puisque nous ne pouvons plus manipuler les adresses absolues.

Exemple.- Écrivons un programme en langage d'assemblage qui itère un programme vu précédemment, à savoir qui permet de saisir une lettre et affiche cette lettre (elle apparaît donc deux fois) et recommence indéfiniment. On ne pourra interrompre ce programme que de l'une des façons indiquées ci-dessus.

On peut utiliser par exemple :

```
; saut.asm
; saisit une lettre au clavier, l'affiche, passe a la
; ligne suivante et recommence jusqu'a CTRL-C
         .model small
         .stack 100h
         .code
start :
; affiche '?'
         mov dl, '?'
         mov ah, 2h
         int 21h
; saisit le caractere
         mov ah, 1h
         int 21h
; affiche le caractere
         mov dl, al
         mov ah, 2h
         int 21h
; passe a la ligne
         mov dl, 13
```

```
        mov ah, 2h
        int 21h
        mov dl, 10
        mov ah, 2h
        int 21h
; recommence
        jmp start
; revient a ms-dos (mais sans interet ici)
        mov ax, 4c00h
        int 21h
        end start
```

26.4.7.2 Sauts conditionnels

Nous avons déjà vu ce qu'est un saut conditionnel, les divers sauts conditionnels et les mnémonymes associés à ceux-ci lors de l'étude du langage machine. Nous allons voir ici comment les mettre en place en langage d'assemblage.

Mise en place en langage d'assemblage.- La seule différence ici est que l'on utilise une étiquette au lieu d'une adresse absolue, ce qui évite d'avoir à calculer celle-ci.

Exemple de boucle pour.- Écrivons un programme qui affiche les caractères de code ascii compris entre 33 et 255.

```
; ascii.asm
; affiche les caracteres ascii de 33 a 255
              .model small
              .stack 100h
              .code
start :
              mov bl, 255
              mov cl, 32
debut :
              inc cl
              mov dl, cl
              mov ah, 2h
              int 21h
              cmp bl, cl
              jnz debut
; revient a ms-dos
              mov ax, 4c00h
              int 21h
              end start
```

Exemple de test.- Écrivons un programme permettant de saisir un caractère puis affichant 'A' s'il s'agit d'un 'a'.

```
; a.asm
; saisit un caractere et affiche 'A' si c'est un 'a'
            .model small
            .stack 100h
            .code
start :
; affiche '?'
            mov dl, '?'
            mov ah, 2h
            int 21h
; saisit le caractere
            mov ah, 1h
            int 21h
; compare avec 'a'
            cmp al, 'a'
; va a la fin si ce n'est pas un 'a'
            jne fin
; affiche 'A' sinon
            mov dl, 'A'
            mov ah, 2h
            int 21h
fin :
;revient a ms-dos
            mov ax, 4c00h
            int 21h
            end start
```

Exemple d'alternative.- Écrivons un programme qui permet de saisir un caractère, affiche 'Y' s'il s'agit d'un 'b' et affiche 'N' sinon.

```
; b.asm
; saisit un caractere, affiche 'Y' si c'est un 'b'
; et 'N' sinon
            .model small
            .stack 100h
            .code
start :
; affiche '?'
            mov dl, '?'
            mov ah, 2h
            int 21h
; saisit le caractere
            mov ah, 1h
            int 21h
; compare avec 'b'
            cmp al, 'b'
; va a SINON si ce n'est pas un 'b'
            jne SINON
; affiche 'Y' sinon
            mov dl, 'Y'
            mov ah, 2h
            int 21h
; va a la fin
            jmp fin
; affiche 'N' sinon
SINON :
            mov dl, 'N'
            mov ah, 2h
            int 21h
fin :
; revient a ms-dos
            mov ax, 4c00h
            int 21h
            end start
```

Exemple de boucle non contrôlée.- Écrivons un programme permettant de saisir un caractère, affichant un espace, affichant une seconde fois ce caractère (la première fois étant due à l'écho), allant à la ligne et recommençant jusqu'à ce que le caractère saisi soit '#'.

```
; tantque.asm
; saisit un caractere, l'affiche jusqu'a '#'
            .model small
            .stack 100h
            .code
START :
; affiche '?'
            mov dl, '?'
            mov ah, 2h
            int 21h
; saisit le caractere
            mov ah, 1h
            int 21h
; compare avec '#'
            cmp al, '#'
; va a FIN si c'est un '#'
            je FIN
; sauvegarde du caractere
            mov bl, al
; affiche un espace
            mov dl, ' '
            mov ah, 2h
            int 21h
; affiche le caractere
            mov dl, bl
            mov ah, 2h
            int 21h
; va a la ligne
            mov dl, 13
            mov ah, 2h
            int 21h
            mov dl, 10
            mov ah, 2h
            int 21h
; revient au debut
            jmp START
FIN :
; revient a ms-dos
            mov ax, 4c00h
            int 21h
            end start
```

26.5 Les sous-programmes en langage d'assemblage

Nous avons vu la notion de sous-programme lors de l'étude d'un langage évolué, le langage C. Nous avons vu comment mettre en place un tel sous-programme en langage machine.

Un microprocesseur utilisant la notion de segment, tel que `i8086`, complique un peu la mise en place puisqu'il faut distinguer les sous-programmes se trouvant dans le même segment de code de ceux qui se trouvent dans un autre segment.

26.5.1 Sous-programme intra-segmentaire

26.5.1.1 Calque du langage machine

Introduction.- Voyons comment mettre en place en place un sous-programme situé dans le même segment, ce qui n'est donc possible que pour de petits sous-programmes.

Nom d'un sous-programme.- Un sous-programme porte un **nom**, qui est un identificateur (non utilisé pour autre chose, comme d'habitude).

Passage des paramètres.- Contrairement aux sous-programmes des langages évolués, un sous-programme en langage d'assemblage n'a pas d'arguments. Autrement dit tous les paramètres sont passés comme variable globale, en utilisant les registres, la pile ou des emplacements de la mémoire vive.

Définition du sous-programme.- Un sous-programme est défini par l'étiquette formée de son nom et de deux points ':', suivi d'une portion de code, suivi de la directive `ret` (pour *return* ou *retour*).

La définition se place soit après le *programme principal* (sinon la portion de programme serait exécutéee une fois), c'est-à-dire entre le retour à MS-DOS et la fin du programme, soit il faut placer un saut inconditionnel avant le sous-programme.

Appel d'un sous-programme.- Pour appeler un sous-programme, c'est-à-dire faire exécuter la portion de programme qui en constitue le corps, il suffit d'utiliser l'instruction `call` suivi du nom du sous-programme.

Un exemple.- Écrivons à nouveau le programme consistant à saisir un caractère et à l'afficher à l'écran mais en utilisant, cette fois-ci, deux sous-programmes : un sous-programme `getc` de saisie d'un caractère et un sous-programme `putc` d'affichage d'un caractère.

```
; call1.asm
; saisit une lettre au clavier et l'affiche a l'ecran
         .model small
         .stack 100h
         .code
; programme principal
start :
         call getc    ;appel de la fonction de saisie
         mov dl,al    ;copie le code ascii de ce caractere dans dl
         call putc    ;appel de la fonction d'affichage
         mov ax,4c00h ;revient a ms-dos
         int 21h
; sous-programmes
putc :
         mov ah,2h
         int 21h
         ret
getc :
         mov ah,1h
         int 21h
         ret
         end start
```

26.5.1.2 Directives de sous-programmes

Syntaxe.- La syntaxe que nous avons donnée ci-dessus suit de très près ce qui se fait en langage machine. Il vaut mieux utiliser les directives suivantes :

```
NOM PROC NEAR
; le code
NOM ENDP
```

le paramètre `NEAR` étant optionnel, puisqu'il s'agit du paramètre par défaut. On utilise le paramètre `FAR` pour un sous-programme inter-segmentaire. Le programme principal est toujours inter-segmentaire pour le système d'exploitation MS-DOS.

Exemple. Réécrivons le programme précédent en utilisant ces directives :

```
; call2.asm
; saisit une lettre au clavier et l'affiche a l'ecran \\
         .model small
         .stack 100h
         .code
; programme principal
start    PROC   FAR
         call   getc          ;appel de la fonction de saisie
         mov    dl,al         ;copie le code ascii de ce
                              ;caractere dans dl
         call   putc          ;appel de la fonction d'affichage
         mov    ax,4c00h      ;revient a ms-dos
         int    21h
start    ENDP
; sous-programmes
putc     PROC
         mov    ah,2h
         int    21h
         ret
putc     ENDP
getc     PROC
         mov ah,1h
         int 21h
         ret
getc     ENDP
         end start
```

26.5.2 Sous-programme inter-segmentaire

Lorsque l'instruction `call` et le sous-programme qu'elle appelle se trouvent dans le même segment, on parle d'**appel proche** (en anglais *near call*). Lorsque l'instruction `call` et le sous-programme qu'on appelle se trouvent dans des segments différents, on parle d'**appel lointain** (en anglais *far call*).

Lors de l'appel d'un sous-programme inter-segmentaire, il faut sauvegarder les registres `CS` et `IP` puis, à la fin du sous-programme, récupérer les anciennes valeurs de ces deux registres.

26.5.2.1 Première syntaxe

Mise en place.- La définition du sous-programme se fait de la même façon que pour la première syntaxe d'un sous-programme proche, à cela près qu'il se termine par l'instruction `RETF` (pour *RETurn Far*) au lieu de l'instruction `RET`.

Lors de l'appel de ce sous-programme, il faudra indiquer explicitement que l'on fait appel à un sous-programme lointain par l'instruction :

```
call far ptr sub
```

si le nom du sous-programme est, par exemple, `sub`.

Exemple.-

Remarque.- Comment le sous-programme est-il placé dans un autre segment ? Grâce à PROC FAR ? avec debug ?

à terminer

26.5.2.2 Deuxième syntaxe

NOM PROC FAR
NOM ENDP

à terminer

26.5.3 Utilisation de la pile pour conserver les registres

Introduction.- L'utilisation des sous-programmes pose deux problèmes en ce qui concerne les registres. On utilise des registres pour passer les paramètres, ce qui réduit le nombre de registres pour stocker les autres informations. De plus on modifie, dans le corps du sous-programme, les valeurs des registres ; cela peut avoir une influence subtile sur le programme sans que l'on s'en aperçoive facilement, surtout si on n'a pas écrit soi-même le sous-programme.

Par exemple, dans la portion de programme suivante :

```
; exemple a eviter
mov  dx,11
mov  al,65
call putc
add  dx,2
```

la valeur du registre `dx` a été changée par le sous-programme `putc` alors que l'on s'attend certainement à ce qu'il contienne la valeur 11, même après exécution du sous-programme.

Sauvegarde des registres.- On a donc intérêt, dans tout sous-programme, à sauvegarder tous les registres, sauf ceux qui contiendront les résultats, puis à les restituer à la fin. Pour cela on les place sur la pile au début du sous-programme et on les récupère à la fin.

Remarquons que l'on peut sans problème sauvegarder les registres qui contiennent les données, puisque les valeurs placées dans la pile demeurent aussi dans les registres.

Le sous-programme `putc`, par exemple, sera correctement écrit de la façon suivante :

```
putc:           ; affiche le caractere de al
      push ax   ; sauvegarde les registres
      push bx
      push cx
      push dx
      mov dl,al ; sous-programme proprement dit
      mov ah,2h
      int 21h
      pop dx    ; restaure les registres
      pop cx
      pop bx
      pop ax
      ret
```

Remarques.- 1°) Il faut récupérer les valeurs des registres dans l'ordre inverse de celui dans lequel elles ont été sauvegardées.

- 2°) Il est important de désempiler tous les éléments mis sur la pile dans le corps du sous-programme. Sinon, à la fin du sous-programme, l'adresse de retour ne sera pas correcte, ce qui conduira à un comportement imprévisible.

26.6 Les chaînes de caractères

Une **chaîne de caractères** (en anglais *string*) est une liste de caractères traitée comme un tout.

26.6.1 Déclaration des chaînes de caractères

Nous avons déjà vu que, dans le langage d'assemblage Microsoft, les variables chaînes de caractères sont déclarées comme des tableaux de caractères, en utilisant la directive `db`. Par exemple "abc" peut être déclaré de l'une des façons suivantes :

```
mot1 db 97, 98, 99      ; codes ASCII individuels
mot2 db 'a', 'b', 'c'   ; caracteres individuels
mot3 db 'abc'           ; constante chaine de caracteres
mot4 db "abc"           ; pas de difference entre ' et "
```

On peut aussi déclarer une telle variable sans l'initialiser :

```
mot db 5 dup (?)
```

26.6.2 Affichage d'une chaîne de caractère

Introduction.- Nous avons déjà vu une première façon d'afficher une chaîne de caractères, qui consiste à l'afficher caractère par caractère. En fait il existe une interruption du DOS permettant d'afficher une chaîne de caractères, à condition que celle-ci soit sous un format donné, c'est-à-dire qu'elle se termine par le caractère '$'.

En effet il faut bien un délimiteur pour indiquer quand il faut s'arrêter. Le choix de celui-ci est un peu arbitraire : on choisit un caractère peu utilisé.

La fonction 9h.- Pour afficher une chaîne de caractères en utilisant la fonction `9h` de l'interruption `21h`, il faut que la chaîne de caractères se termine par le caractère '$', qui lui-même ne sera pas affiché. Il faut placer l'adresse du premier caractère dans le registre `dx` et faire appel à cette fonction.

Exemple.- Écrivons un programme permettant d'afficher "Bonjour" et d'aller à la ligne.

```
; bonjour.asm
       .model small
       .stack 100h
       .data
message db 'Bonjour', 13, 10, '$'
       .code
start:
       mov ax,@data
       mov ds,ax
       mov dx,offset message ; copie l'adresse de message dans dx
       mov ah,9h             ; fonction d'affichage de mot
       int 21h               ; appel de l'interruption ms-dos
       mov ax, 4c00h
       int 21h
end    start
```

Remarques.- 1^o) Il ne faut surtout pas oublier le caractère qui termine la chaîne de caractères. Si on l'oublie, tous les caractères de la mémoire seront affichés jusqu'à ce que l'on rencontre un signe dollar, à supposer qu'il y en ait un.

- 2^o) Un inconvénient avec la fonction `9h` est qu'on ne peut pas afficher le signe dollar.

26.6.3 Entrée-sortie des chaînes de caractères

Nous avons vu deux façons d'afficher une chaîne de caractères mais aucune pour saisir une telle chaîne. Nous allons voir deux sous-programmes permettant de saisir et d'afficher une chaîne de caractères.

Nous suivrons la convention du langage C pour terminer une chaîne de caractères, à savoir par le caractère nul '0', plutôt que la convention de MS-DOS. Celle-ci a pour intérêt de ne pas utiliser un caractère affichable comme terminateur.

26.6.3.1 Saisie d'une chaîne de caractères

Principe.- Utilisons un sous-programme `gets`. On place l'adresse du tampon de la chaîne de caractères à saisir dans le registre `ax` et on appelle ce sous-programme. Cette adresse est celle du premier caractère. On utilise le registre `bx` comme index. On commence par initialiser `bx` par la valeur de `ax`. On saisit un caractère que l'on place à l'adresse indiquée par `bx` et on incrémente `bx`. On recommence jusqu'à ce qu'on rencontre `CR`; on place alors '0' au lieu de `CR` et on s'arrête.

Le programme.- Le sous-programme est donné ci-dessous dans un exemple de test qui demande un prénom, puis qui affiche "Bonjour" suivi de ce prénom. Nous ajoutons, uniquement pour cet exemple, deux lignes de code au sous-programme `gets` de façon à ce que la chaîne de caractères saisie se termine également par un dollar, de façon à pouvoir utiliser la fonction d'affichage de MS-DOS.

```
; gets.asm   test de gets
        .model small
        .stack 256
        .data
prenom  db 30 dup (?)
prompt1 db 'Quel est votre prenom ? ','$'
prompt2 db 13, 10, 'Bonjour ', '$'
        .code
start:
        mov ax,@data
        mov ds,ax
        mov ax,offset prompt1   ; affiche le prompteur
        call puts
        mov ax,offset prenom    ; saisie du prenom
        call gets
        mov ax,offset prompt2   ; affiche bonjour
        call puts
        mov ax,offset prenom    ; affiche le prenom
        call puts
        mov ax, 4c00h
        int 21h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; sous-programmes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
gets:             ; lit une chaine de caracteres au clavier
                  ; terminee par CR, et la
                  ; sauvegarde a l'adresse indiquee par ax
      push ax     ; sauvegarde les registres
```

```
      push bx
      push cx
      push dx
      mov bx, ax
 gets_nouveau :
      call getc   ; lit un caractere dans al
      cmp al, 13  ; est-ce CR ?
      je gets_fin ; si oui aller a la fin
                  ; si non le placer
      mov byte ptr [bx], al
      inc bx
      jmp gets_nouveau
 gets_fin :
      mov byte ptr [bx],0    ; termine la chaine avec 0
      inc bx
      mov byte ptr [bx],'$'  ; pour utiliser ici le puts du dos
      pop dx      ; restaure les registres
      pop cx
      pop bx
      pop ax
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
puts:
      push ax
      push bx
      push cx
      push dx
      mov dx,ax
      mov ah,9h
      int 21h
      pop dx
      pop cx
      pop bx
      pop ax
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getc:
      push bx
      push cx
      push dx
      mov ah,1h
      int 21h
      pop dx
      pop cx
      pop bx
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      end start
```

26.6.3.2 Affichage d'une chaîne de caractères

Principe.- Utilisons un sous-programme `puts`. On place l'adresse de la chaîne de caractères à afficher dans le registre `ax` et on appelle ce sous-programme. Cette adresse est celle du premier caractère de la chaîne. On utilise le registre `bx` comme index, comme dans le sous-programme précédent. On commence par initialiser `bx` par la valeur de `ax`. On affiche le caractère placé à l'adresse indiquée par `bx` et on incrémente `bx`. On continue jusqu'à ce qu'on trouve la valeur nulle.

Le programme.- Le sous-programme est donné ci-dessous dans un programme de test effectuant la même chose que le programme de test ci-dessus. Remarquons qu'ici nous avons enlevé les deux lignes de code non indispensables dans le sous-programme `gets`.

```
; puts.asm   test de puts
        .model small
        .stack 256
        .data
prenom  db 30 dup (?)
prompt1 db 'Quel est votre prenom ? ', 0
prompt2 db 13, 10, 'Bonjour ', 0
        .code
start:
        mov ax,@data
        mov ds,ax
        mov ax,offset prompt1   ; affiche le prompteur
        call puts
        mov ax,offset prenom    ; saisie du prenom
        call gets
        mov ax,offset prompt2   ; affiche bonjour
        call puts
        mov ax,offset prenom    ; affiche le prenom
        call puts
        mov ax, 4c00h
        int 21h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; sous-programmes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
puts:                           ; affiche une chaine de caracteres terminee
                                ; par 0 dont l'adresse est dans ax
        push ax                 ; sauvegarde les registres
        push bx
        push cx
        push dx
        mov bx,ax               ; initialise bx avec l'adresse
puts_nouveau:
        mov al,byte ptr [bx]    ; considere un caractere de la chaine
        cmp al,0                ; est-ce la fin ?
        je puts_fin             ; si oui aller a la fin
        call putc               ; sinon afficher le caractere
        inc bx
        jmp puts_nouveau
```

```
puts_fin:
        pop dx                  ; restaure les registres
        pop cx
        pop bx
        pop ax
        ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
gets:                           ; lit une chaine de caracteres au clavier
                                ; terminee par CR, et la
                                ; sauvegarde a l'adresse indiquee par ax
      push ax                   ; sauvegarde les registres
      push bx
      push cx
      push dx
      mov bx, ax
 gets_nouveau :
      call getc                 ; lit un caractere dans al
      cmp al,13                 ; est-ce CR ?
      je gets_fin               ; si oui aller a la fin
                                ; si non le placer
      mov byte ptr [bx],al
      inc bx
      jmp gets_nouveau
 gets_fin :
      mov byte ptr [bx],0       ; termine la chaine avec 0
      pop dx                    ; restaure les registres
      pop cx
      pop bx
      pop ax
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putc:                           ; affiche le caractere situe en al
      push ax
      push bx
      push cx
      push dx
      mov dl, al
      mov ah, 2h
      int 21h
      pop dx
      pop cx
      pop bx
      pop ax
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getc:
      push bx
      push cx
      push dx
      mov ah, 1h
```

```
      int 21h
      pop dx
      pop cx
      pop bx
      ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      end start
```

26.7 Entrées-sorties des entiers

Il existe des instructions d'entrée-sortie des entiers dans tous les langages de programmation évolués. Nous n'en avons pas vu pour le langage machine du `i80x86`. Il n'en existe pas ; ce langage, comme la plupart des langages modernes, est orienté mot et non orienté nombre.

Pour saisir ou afficher un entier, nous devons effectuer l'opération correspondante sur un mot puis traduire ce mot en un entier.

26.7.1 Affichage d'un entier

Principe.- Écrivons un sous-programme `putn` permettant d'afficher à l'écran un entier placé dans le registre `ax`.

On effectue la division euclidienne de l'entier par 10 : le reste est le chiffre de droite et on recommence avec le quotient s'il est non nul. Puisque nous obtenons les chiffres dans l'ordre inverse de celui de l'affichage, nous commençons par les sauvegarder sur la pile. Pour pouvoir déterminer le premier chiffre lorsque nous lirons la pile, on commence par placer une valeur sentinelle sur celle-ci, par exemple le caractère nul (à ne pas confondre avec le chiffre 0).

Le programme.- Écrivons un programme complet permettant d'afficher l'entier 16.

```
; affichen.asm
; affiche un entier
         .model small
         .stack 256
 CR      equ 13d
 LF      equ 10d
         .data

 prompt db  'a = ', 0
         .code
 debut :
          mov ax,@data
          mov ds,ax
          mov dx,offset prompt
          call puts
          mov ax, 16
          call putn
          mov ax,4c00h
          int 21h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; definition des sous-programmes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putc:
; voir plus haut, non recopie ici
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
puts:
; idem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putn:
          push ax              ; sauvegarde les registres
```

```
          push bx
          push cx
          push dx
          mov dx,0
          push dx              ; place 0 sur la pile comme sentinelle
          mov cx,10            ; initialise cx a 10 pour la division
          cmp ax,0
          jge chiffres
          ; si le nombre est negatif
          neg ax               ; ax = - ax
          push ax              ; sauvegarde de ax
          mov al,'-'
          call putc            ; affiche le signe -
          pop ax               ; on recupere ax
          cmp ax,0
          jne chiffres
          mov al,'0'
          call putc            ; affiche '0'
          jmp fin_putn
chiffres:
          div cx               ; dx = ax % cx, ax = ax/cx
          add dx,'0'           ; convertit le chiffre en caractere
          push dx              ; sauvegarde le chiffre sur la pile
          mov dx,0             ; dx = 0
          cmp ax,0             ; termine ?
          jne chiffres
                               ; tous les chiffres sont maintenant sur la
                               ; pile
                               ; on va les afficher dans l'ordre invers
affiche:
          pop ax               ; on recupere un chiffre
          cmp ax,0             ; est-la valeur sentinelle ?
          je fin_putn          ; si oui on a fini
          call putc            ; sinon afficher le chiffre
          jmp affiche
fin_putn:
          pop dx               ; on restaure les registres
          pop cx
          pop bx
          pop ax
          ret
          end debut
```

26.7.2 Saisie d'un entier

Principe.- Écrivons un sous-programme `getn` permettant de saisir un entier et de le placer dans le registre `ax`. On saisit un mot caractère par caractère grâce au sous-programme `getc`. Si le premier caractère est le signe '-' alors l'entier est négatif. Les caractères suivants doivent être des chiffres. On utilise la **méthode de Hörner** pour obtenir (la valeur absolue de) l'entier : n est initialisé à 0 puis, à chaque fois qu'on lit un chiffre a, on a $n = 10 * n + a$. On s'arrête lorsqu'on saisit un passage à la ligne.

Le programme.- Écrivons un programme complet permettant de saisir deux entiers et d'afficher leur somme.

```
; sum.asm
; saisit deux entiers au clavier et affiche leur somme
         .model small
         .stack 256
 CR      equ 13d
 LF      equ 10d
         .data
 prompt1 db  'a = ', 0
 prompt2 db  CR, LF, 'b = ', 0
 result  db  CR, LF, 'a + b = ', 0
 a       dw  ?
 b       dw  ?
         .code
 debut:
          mov ax,@data
          mov ds,ax
          mov ax,offset prompt1
          call puts              ; affiche le premier prompteur
          call getn              ; lit le premier nombre
          mov a,ax               ; le place dans a
          mov ax,offset prompt2
          call puts              ; affiche le second prompteur
          call getn              ; lit le second nombre
          mov b,ax               ; le place dans b
          mov ax,offset result
          call puts              ; affiche le resultat
          mov ax,a               ; ax = a
          add ax,b               ; ax = ax + b
          call putn
          mov ax,4c00h
          int 21h
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; definition des sous-programmes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putc:
; non recopie ici
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getc:
; idem
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
puts:
; idem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putn:
; idem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getn:
          push bx              ; sauvegarde les registres sauf ax
          push cx
          push dx
          mov dx,1             ; initialise le signe a +
          mov bx,0             ; initialise le chiffre a 0
          mov cx,0             ; initialise le nombre a 0
          call getc            ; lit le premier caractere
          cmp al,'-'           ; est-ce le signe - ?
          jne signe
          mov dx,-1            ; si oui noter le signe
          call getc            ; lire le premier chiffre
signe :
          push dx              ; sauvegarder le signe
nouveau:
          cmp al,13            ; est-on a la fin
          je fin_read          ; si oui aller a la fin
                               ; sinon
          sub al,'0'           ; convertir en chiffre
          mov bl,al
          mov ax,10
          mul cx               ; ax = cx*10
          mov cx,ax
          add cx,bx            ; cx = cx*10 + chiffre
          call getc            ; lit un nouveau chiffre
          jmp  nouveau
fin_read:
          mov ax,cx            ; le nombre est place dans ax
          pop dx               ; le signe dans dx
          cmp dx,1             ; le signe est-il + ?
          je fin_getn
          neg ax               ; sinon ax = - ax
fin_getn:
          pop dx
          pop cx
          pop bx
          ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          end debut
```

26.8 La programmation modulaire

La **programmation modulaire** en langage d'assemblage a deux origines : la première est que l'on ne veut pas réécrire sans arrêt les mêmes sous-programmes, même s'il suffit pour cela d'un copier-coller ; la seconde est que l'on a intérêt à séparer un gros programmes en plusieurs fichiers. On veut donc concevoir un fichier de sous-programmes, ayant en général une certaine unité ; il est assemblé pour obtenir un fichier objet. On peut alors **lier** plusieurs tels fichiers pour obtenir un fichier exécutable. On parle de **module** pour chacun de ces fichiers indépendants.

26.8.1 Le problème de l'interface

Notion d'interface.- Un module utilise un certain nombre d'identificateurs, pour les variables, les étiquettes, les noms de sous-programes, etc.

Un des grands principes de la modularité est que ces identificateurs, par défaut, ne sont pas connus de l'extérieur, c'est-à-dire des autres modules. En effet chaque module est en général conçu par une personne, ou une équipe, indépendante, qui peut vouloir utiliser le même identificateur qu'une autre, mais avec un sens différent.

Lorsqu'on veut qu'un identificateur soit connu de l'extérieur du module, c'est-à-dire par d'autres modules, il faut l'indiquer explicitement à travers ce qu'on appelle l'**interface**.

En langage d'assemblage, l'interfaçage se réalise grâce aux directives `public` et `extern`, comme nous allons le voir.

La directive `public`.- On utilise cette directive pour qu'un identificateur représentant une constante, une variable ou le nom d'un sous-programme soit accessible par d'autres modules que celui dans lequel il est défini. Pour les constantes, il ne peut s'agir que de constantes de type entier.

Il suffit de placer :

```
PUBLIC nom
```

ou, pour plusieurs identificateurs :

```
PUBLIC nom1, nom2, nom3
```

quelque part dans le fichier source du module.

La directive `extern`.- Il ne suffit pas que le module permettant d'utiliser un de ses identificateurs à l'extérieur le permette, en le déclarant `public`. Il faut également que le module qui veut l'utiliser indique explicitement qu'il veut l'utiliser : en effet sinon, lorsqu'on assemblera le module utilisateur, cela déclenchera une erreur puisqu'un identificateur serait utilisé sans être déclaré.

De plus il faut indiquer la nature, autrement dit le type, de cet identificateur.

Pour cela, on utilise la directive :

```
EXTRN nom:type
```

dans le segment adéquat, à savoir les déclarations de sous-programme dans le segment de code, les déclarations de variable dans le segment des données et les déclarations de constantes n'importe où.

Les différents types.- Les types possibles sont les suivants :

Type	**Description**
ABS	une constante (entière) definie par EQU ou =
PROC	une procédure
NEAR	nom dans le même segment
FAR	nom dans un segment différent
BYTE	taille de 8 bits
WORD	taille de 16 bits
DWORD	taille de 32 bits
FWORD	taille de 48 bits
QWORD	taille de 64 bits
TBYTE	taille de 10 octets

Liens entre les deux modules.- Il ne suffit pas de dire que le premier module possède des identificateurs qui seront utilisés dans le second module, encore faut-il qu'il y ait un lien entre les deux modules. Cela se fera grâce à l'utilisation du **lieur**. Jusqu'à maintenant on a lié un seul fichier objet, maintenant on indiquera en paramètres plusieurs fichiers objets.

26.8.2 Un exemple

Sujet.- Réécrivons le programme qui demande deux entiers et affiche leur somme de façon modulaire.

Écriture du premier module.- Pour cela, dans un premier fichier source de nom `mod.asm`, définissons les constantes 'CR' et 'LF' ainsi que les sous-programmes :

```
; mod.asm
         .model small
 CR      equ 13d
 LF      equ 10d
public CR, LF
         .code
public putn, putc, getc, puts, getn
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putn proc
          push bx        ; sauvegarde les registres
          push cx
          push dx
          mov dx,0
          push dx        ; place 0 sur la pile comme sentinelle
          mov cx,10      ; initialise cx  10 pour la division
          cmp ax,0
          jge chiffres
          ; si le nombre est ngatif
          neg ax         ; ax = - ax
          push ax        ; sauvegarde de ax
          mov al,'-'
          call putc      ; affiche le signe -
          pop ax         ; on rcupre ax
 chiffres:
          div cx         ; dx = ax % cx, ax = ax/cx
          add dx,'0'     ; convertit le chiffre en caractre
          push dx        ; sauvegarde le chiffre sur la pile
          mov dx,0       ; dx = 0
          cmp ax,0       ; termine ?
          jne chiffres
                         ; tous les chiffres sont maintenant sur la
                         ; pile
                         ; on va les afficher dans l'ordre inverse
 affiche:
          pop ax         ; on recupere un chiffre
          cmp ax,0       ; est-la valeur sentinelle ? 0 != '0'
          je fin_affiche ; si oui on a fini
          call putc      ; sinon afficher le chiffre
          jmp affiche
 fin_affiche:
          pop dx         ; on restaure les registres
          pop cx
          pop bx
```

```
          ret
putn endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
putc proc
          push ax
          push bx
          push cx
          push dx
          mov dl,al
          mov ah,2h
          int 21h
          pop dx
          pop cx
          pop bx
          pop ax
          ret
putc endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getc proc
          push bx
          push cx
          push dx
          mov ah,1h
          int 21h
          pop dx
          pop cx
          pop bx
          ret
getc endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
puts proc
          push ax
          push bx
          push cx
          push dx
          mov dx,ax
          mov ah,9h
          int 21h
          pop dx
          pop cx
          pop bx
          pop ax
          ret
puts endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
getn proc
          push bx        ; sauvegarde les registres
          push cx
          push dx
          mov dx,1       ; initialise le signe  +
```

```
          mov bx,0        ; initialise le chiffre  0
          mov cx,0        ; initialise le nombre  0
          call getc       ; lit le premier caractre
          cmp al,'-'      ; est-ce le signe - ?
          jne signe
          mov dx,-1       ; si oui noter le signe
          call getc       ; lire le premier chiffre
 signe:
          push dx         ; sauvegarder le signe
 nouveau:
          cmp al,13       ; est-on la fin ?
          je fin_read     ; si oui aller a la fin
                          ; sinon
          sub al,'0'      ; convertir en chiffre
          mov bl,al
          mov ax,10
          mul cx          ; ax = cx*10
          mov cx, ax
          add cx, bx      ; cx = cx*10 + chiffre
          call getc       ; lit un nouveau chiffre
         jmp  nouveau
 fin_read:
          mov ax,cx       ; le nombre est place dans ax
          pop dx          ; le signe dans dx
          cmp dx,1        ; le signe est-il + ?
          je fin_getn
          neg ax          ; sinon ax = - ax
 fin_getn:
          pop dx
          pop cx
          pop bx
          ret
getn endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
         end
```

<u>Remarque</u>.- Le programme source se termine par la directive `end` sans indication d'étiquette. Ceci indique que ce fichier ne peut pas donner lieu à un exécutable à lui tout seul.

<u>Assemblage</u>.- On assemble ce programme de la façon habituelle :

```
c:>masm mod.asm
```

ce qui donne lieu à un fichier `mod.obj`.

Écriture du deuxième module.- On écrit ensuite un programme source `main.asm` faisant appel à ces procédures et à ces constantes :

```
; main.asm
; saisit deux entiers au clavier et affiche leur somme
         .model small
         .stack 256
         .data
 extrn CR:abs, LF:abs
 prompt1 db  'a = $'
 prompt2 db  CR, LF, 'b = $'
 result  db  CR, LF, 'a + b = $'
 a       dw  ?
 b       dw  ?
         .code
 extrn putn:proc, getn:proc, puts:proc
 main proc
          mov ax,@data
          mov ds,ax
          mov ax,offset prompt1
          call puts              ; affiche le premier prompteur
          call getn              ; lit le premier nombre
          mov a, ax              ; le place dans a
          mov ax, offset prompt2
          call puts              ; affiche le second prompteur
          call getn              ; lit le second nombre
          mov b, ax              ; le place dans b
          mov ax, offset result
          call puts              ; affiche le rsultat
          mov ax, a              ; ax = a
          add ax, b              ; ax = ax + b
          call putn
          mov ax,4c00h
          int 21h
main endp
         end main
```

Assembage du deuxième module.- On assemble ce programme :

```
masm main.asm
```

et on obtient un fichier `main.obj`.

Liage des deux modules.- On lie enfin les deux fichiers objet :

```
link main mod
```

Le nom de l'exécutable est alors demandé (le lieur hésite en effet entre le nom du premier fichier et celui du second). On répond par `main.exe`.

On fait exécuter évidemment le programme `main.exe` comme avant.

26.8.3 Les bibliothèques

26.8.3.1 Notions générales

Notion de bibliothèque.- Nous avons vu la notion de sous-programme et son utilité pour la programmation modulaire. Nous avons vu également comment écrire des sous-programmes dans un fichier autre que le fichier principal. On peut aller plus loin en créant des **bibliothèques** (en anglais *link library*) de sous-programmes déjà compilés. Il s'agit d'un fichier contenant des sous-programmes objet.

Intérêt des bibliothèques.- Imaginons que nous ayions un module pour les entiers, avec un certain nombre de constantes, variables et sous-programmes, et un module pour les chaînes de caractères. Nous pouvons placer ces deux modules dans une bibliothèque. L'intérêt est que nous avons un seul fichier bibliothèque. Lorsqu'on fait appel, dans un programme, à un sous-programme du module sur les entiers, seul ce module sera chargé et lié pour produire l'exécutable. Ainsi, bien que la bibliothèque puisse être de grande taille, seul l'essentiel sera lié. Bien entendu on peut écrire un module par sous-programme pour être sûr de ne lier que ce qui est vraiment utile, mais en général on recherche un compromis entre la taille et l'unité.

Outil de création de bibliothèques.- En général les assembleurs sont distribués avec un utilitaire permettant de créer les bibliothèques (*library manager* en anglais).

26.8.3.2 Cas de MASM

Introduction.- Nous allons voir comment créer une bibliothèque, puis comment l'utiliser, en utilisant MASM.

Création des modules.- Nous avons déjà vu comment créer les modules. On a un fichier source du schéma suivant, d'extension `.asm` :

`title` description du module (nom du fichier)
`.model small`
`public` liste d'identificateurs
déclaration des constantes
`.data`
variables
`.code`
procédures
`end`

Ensuite on assemble comme d'habitude, par exemple :

```
c:>masm mod.asm
```

pour obtenir un fichier `mod.obj`.

Création de la bibliothèque.- Si on veut créer une bibliothèque de nom `entier`, avec MASM on fera :

```
lib entier;
```

(remarquer le point-virgule). Cette bibliothèque ne contient encore rien mais il existe un fichier `entier.lib` de taille d'un peu plus d'un Ko.

Ajout d'un module à la bibliothèque.- Si on veut placer (ajouter) le module `mod.obj`, par exemple, dans la bibliothèque, avec MASM on écrit :

```
lib entier +mod
```

Évidemment la taille de la bibliothèque grossit. Les commandes possibles sont :

`+nom`	: ajoute un fichier objet
`-nom`	: enlève un fichier objet
`-+nom`	: remplace un fichier objet
`*nom`	: copie (extrait) un fichier objet
`-*nom`	: déplace (enlève et copie) un fichier objet

Utilisation d'une bibliothèque.- Les constantes, variables et sous-programmes se déclarent (avec `extrn`) dans le fichier source dans lequel on veut les utiliser comme lorsqu'ils sont dans un module séparé. On réalise la liaison lors de l'assemblage :

```
ml main.asm entier.lib
```

et on obtient, ici, un exécutable de nom `main.exe`.

Exemple.- Considérons à nouveau les deux fichiers sources `mod.asm` et `main.asm` de la section précédente (sur la programmation modulaire).
Assemblons le premier programme :

```
c:> masm mod.asm
```

donnant lieu à un fichier `mod.obj`.
Créons une bibliothèque `bib.lib` :

```
c:> lib bib;
```

et ajoutons-lui le module précédent :

```
c:> lib bib +mod
```

Assembons enfin notre programme principal :

```
c:> ml main.asm bib.lib
```

qui donne lieu à l'exécutable `main.exe`, dont on peut vérifier qu'il fonctionne.

26.9 Langage d'assemblage et langage évolué

Jusqu'à maintenant nous avons écrit des programmes soit dans un langage évolué (comme le langage C, Pascal,...), soit dans un langage d'assemblage. Nous avons déjà dit aussi que le langage machine est indispensable, mais à éviter autant que faire se peut, que le langage d'assemblage est plus compréhensible que le langage machine mais également à éviter et qu'il vaut mieux utiliser un langage évolué.

En pratique, de nos jours, presque tout est écrit en langage évolué, avec le quasi monopole du langage C. Cependant il faut écrire de petits modules en langage d'assemblage, soit pour accélérer certaines parties du code, soit pour la programmation système, certaines ressources n'étant pas accessibles à travers un langage évolué.

Il y a plusieurs façons de faire pour cela. Tout d'abord on peut avoir recours aux interruptions directement dans le langage évolué. Ensuite on peut avoir une partie du programme d'assemblage **en ligne** dans le programme en langage évolué, c'est-à-dire qu'il apparaît tel que dans le même fichier avec un mot clé (par exemple `asm`) pour l'introduire. Enfin on peut écrire deux parties de programme, l'un en langage évolué, l'autre en langage machine, et il faut **lier** ces deux parties.

26.9.1 Lier un programme C et un programme en langage d'assemblage

La façon de faire semble dépendre du compilateur utilisé.

26.9.1.1 Cas de Turbo C

Intéressons-nous au compilateur `Turbo C` de Borland et de ses descendants, à savoir en particulier `Borland C++`. Il faut de plus disposer de l'assembleur `Tasm`.

Un exemple.- Écrivons un programme demandant un numéro de ligne, un numéro de colonne et écrivant « nouvelle position du curseur » en débutant à la position correspondant à ces données.

Première étape : Écrivons la partie du programme en langage C :

```
/* curseurm.c */
#include <stdio.h>

extern set_curs(int, int);

void main(void)
    {
    int li, co;

    printf("Entrer ligne : ");
    scanf("%d", &li);

    printf("Entrer colonne : ");
    scanf("%d", &co);

    set_curs(li, co);
    printf("Nouvelle position du curseur\n");
    }
```

Deuxième étape : Écrivons la partie du programme en langage d'assemblage :

```
_DATA segment word 'DATA'
li        equ [bp+4]
co        equ [bp+6]
_DATA ends

_TEXT segment byte public 'CODE'
DGROUP GROUP _DATA
          assume cs:_TEXT, ds:DGROUP, ss:DGROUP

          PUBLIC _set_curs

_set_curs proc near
          push bp
          mov bp,sp

          mov ah, 02
          mov bx, 0
          mov dh, li
          mov dl, co
          int 10h

          pop bp
          ret
_set_curs endp
_TEXT     ends
          end
; curseur.asm
```

Remarquons, et nous y reviendrons plus longuement après, que le segment des données a le nom imposé `_DATA`, appartenant nécessairement à `DATA`, que le segment de code a le nom imposé `_TEXT`, appartenant nécessairement à `CODE`, que le nom de la procédure utilisée dans le programme C est le même précédé du caractère blanc souligné `_`. Nous expliquerons ensuite comment passer les paramètres.

Troisième étape : compilation, assemblage et lien. Dans le cas d'utilisation de Windows (ce qui est souvent le cas de nos jours), il faut se placer dans une fenêtre DOS ou en mode MS-DOS. Il faut que la variable d'environnement `PATH` permette l'utilisation des binaires de Borland C++ et de ceux de TASM.

Dans le répertoire contenant les deux fichiers ci-dessus, on utilise Borland C++ pour DOS en mode ligne de commande :

```
c:\ >bcc curseurm.c curseur.asm
```

On obtient alors un exécutable nommé `curseurm.exe` que l'on peut tester.

26.10 Bibliographie

[Dun-00] DUNTEMAN, Jeff, **Assembly Language Step-by-Step, Second Edition : Programming with DOS and Linux**, Wiley, 2000, XXV + 613 p. + CD-ROM.

[Un classique de l'introduction au langage d'assemblage, illustré par le langage du microprocesseur `80x86`. Le CD-ROM contient l'assembleur gratuit NASM, dans sa version pour DOS et pour Linux, qui est pris en exemple au lieu des traditionnels TASM et MASM, avec lesquels il est compatible.

Les trois premiers chapitres constituent une introduction très générale au langage d'assemblage et à l'hexadécimal. Le chapitre 4 est une vue générale de ce que fait un langages d'assemblage et les outils associés (système d'exploitation, éditeur de texte, assembleur, lieur, débogueur) puis une introduction à l'utilitaire `debug`. Le chapitre 5 explique comment mettre en place NASM et NASM-IDE (un environnement intégré pour NASM, également présent sur le CD-ROM). Le chapitre 6 porte sur les modèles de mémoire du 8086 et sur les registres. Le chapitre 7 porte sur l'instruction `mov` et sur la notion générale d'instruction. Le chapitre 8 donne un programme en langage d'assemblage et parle de la pile. Le chapitre 9 porte sur les sous-programmes et les macros, le chapitre 10 sur les branchements, le chapitre 11 sur les chaînes de caractères. Les deux derniers chapitres expliquent comment programmer en langage d'assemblage sous Linux.]

[Car-96] CARTHY, Joe, **An introduction to Assembly Language Programming and Computer Architecture**, International Thomson Publishing Company, 1996, XVI+367 p. + disk.

[La première partie, sur l'introduction à la programmation en langage d'assemblage, comprend un chapitre de rappel sur la programmation en langage évolué, illustrée par le langage C, un chapitre sur les concepts de langage d'assemblage, en commençant par le modèle matériel et illustrés par `i8086`, deux chapitres sur le langage d'assemblage du `i8086`. Dans la seconde partie, l'architecture des ordinateurs est décrite ainsi qu'une initiation au `i8086`, aux microprocesseurs de la famille `M6800` et les microprocesseurs RISC PowerPC 601 et Digital Alpha 21064. Un appendice donne les programmes pour `M6800`.

D'un point de vue pratique, commencez à la page 83 pour savoir comment écrire et compiler un programme avec MASM. Les fonctions essentielles d'un microprocesseur sont vues ; il n'y a rien sur `debug` ni sur les périphériques (clavier ou carte graphique), puisque c'est le microprocesseur seul qui est étudié.]

[Maz-98] MAZIDI, Muhammad Ali & MAZIDI, Janice Gillipsie, **The 80x86 IBM PC and Compatible Computers (Volumes I & II) : Assembly Language, Design, and Interfacing, Second Edition**, Prentice-Hall, 1998, XXXVIII + 984 p.

[Commencer par le bon appendice sur la programmtion en langage symbolique avec `debug`. Les exemples sont ensuite illustrés soit avec `debug`, soit avec MASM ou même l'interfaçage avec C/C++. Bonne étude du langage d'assemblage du `i8086`, des interruptions puis des autres circuits intégrés que l'on trouve sur la carte mère d'un PC ainsi que la façon d'interfacer un périphérique.]

[Irv-95] IRVINE, Kip R., **Assembly language for Intel-based computers**, Prentice-Hall, 1995, third edition 1998, XXIV + 676p. + CD-ROM.

[Le CD-ROM contient les exemples du livre et surtout l'assembleur de Microsoft MASM 6.11 avec une licence mono-utilisateur. Bonne description du langage d'assemblage, en utilisant `debug`, MASM ou TASM, des microprocesseurs de la famille `i8086` et des interruptions. N'aborde pas le graphisme, puisqu'il s'agit d'un périphérique.]

[Tho-86] THORNE, Michael, **Computer Organization and Assembly Language Programming : For IBM PCs and Compatibles**, Benjamin/Cummings, 1986, second edition 1990, XVIII + 697 p.

[Intéressant surtout par les exemples de programme en langage d'assemblage sous une forme dépouillée, sans trop de directives d'assemblage propres à MASM ou à TASM.]

Les références primaires sur la programmation en langage d'assemblage sont la description du microprocesseur intel, de son macro-assembleur, puis des assembleurs MASM et Turbo-assembler.