Apprendre l'assembleur
(INTEL - DOS 16 bits)

Par Benoît-M

Ce document a été téléchargé depuis http://benoit-m.developpez.com/assembleur/tutoriel

Dernière modification du contenu le 03 janvier 2003

« Désormais, pour apprendre le français, il faudra SAVOIR le français ! » disait Coluche. De même, ce tutoriel s'adresse à tous ceux qui connaissent déjà les rudiments d'un langage évolué tel que le BASIC, le FORTRAN, le C ou le PASCAL et qui souhaitent apprendre l'assembleur. Une connaissance même sommaire d'un de ces langages suffit ! Aucune connaissance en programmation système n'est requise : le but de ce cours est justement d'en introduire les fondements.

Contrairement aux langages évolués, l'assembleur, ou « langage d'assemblage » est constitué d'instructions directement compréhensibles par le microprocesseur : c'est ce qu'on appelle un langage de *bas niveau*. Il est donc intimement lié au fonctionnement de la machine. C'est pourquoi il est relativement difficile à assimiler, en tout cas beaucoup plus que les langages de haut niveau.

Cela explique également pourquoi il existe au moins autant de langages d'assemblage que de modèles de microprocesseurs.

Avant d'apprendre l'assembleur INTEL 80x86, il est donc primordial de s'intéresser à quelques notions de base concernant par exemple la mémoire ou le microprocesseur. C'est là en effet que se trouvent les principales difficultés pour le débutant. Ne soyez pas rebuté par l'abstraction des concepts présentés dans les premiers paragraphes : il est normal que durant la lecture, beaucoup de choses ne soient pas claires dans votre esprit. Tout vous semblera beaucoup plus simple quand nous passerons à la pratique dans le langage assembleur.

Le microprocesseur peut fonctionner sous deux modes : le mode *réel* et le mode *protégé*. Le mode protégé permet d'accéder à 2^{32} octets de mémoire vive, alors que le mode réel ne peut en adresser que 2^{20} = 1 Mo. Nous ne traiterons dans ce cours que le mode réel. C'est celui qu'utilisent la plupart des programmes DOS.

<u>Convention</u> : toutes les adresses sont écrites en notation hexadécimale. Les autres nombres seront la plupart du temps représentés en base décimale. Dans le cas contraire, nous ajouterons la lettre 'h' après les chiffres.

PREMIERE PARTIE

NOTIONS DE BASE SUR LE FONCTIONNEMENT DE L'ORDINATEUR

I. L'ARITHMETIQUE SIGNEE

On appelle *arithmétique non signée* l'arithmétique dans laquelle tous les entiers sont positifs. En *arithmétique signée* au contraire, les nombres peuvent être soit positifs, soit négatifs. *Un nombre signé n'est donc pas forcément négatif.*

Les données informatiques se présentent sous la forme d'une succession de chiffres binaires, les *bits*. Nous supposerons que les systèmes de numération binaire et hexadécimal vous sont déjà familiers.

Il est en revanche fort possible que vous ne connaissiez pas la façon dont un ordinateur représente les nombres négatifs. Il existe deux conventions : la notation en *signe et valeur absolue* et la notation en *complément à 2*.

La première est extrêmement simple : le premier bit représente le *signe*, et les autres bits la *valeur absolue* du nombre. Le bit de signe vaut 1 si le nombre est strictement négatif, 0 sinon.

Par exemple, le nombre (–14) codé sur 8 bits s'écrit ainsi :

10001110

Cette convention n'est quasiment jamais utilisée en informatique. On lui préfère la représentation en complément à 2 dont le principe est le suivant :

- les nombres positifs sont codés de la même façon qu'en convention « *signe et valeur absolue* ».

- les nombres négatifs sont obtenus en inversant tous les bits, puis en ajoutant 1.

Exemple : le nombre 14 codé sur 8 bits est représenté ainsi :

00001110

et (–14) ainsi :

- inversion des bits : 11110001
- ajout d'une unité : 11110010
- résultat : **11110010**

Remarque : le résultat intermédiaire, 11110001, est appelé « *complément à 1* ».

Vous allez immédiatement comprendre l'avantage de cette représentation. Faisons la somme de 14 et de (–14), de la même façon que s'il s'agissait d'entiers positifs :

00001110 + 11110010 = 1**00000000**

Le résultat étant codé sur 8 bits, le 1 situé à gauche n'est pas pris en compte. On obtient donc 14 + (-14) = 0.

L'intérêt évident est que la différence de deux nombres peut se calculer avec le même algorithme que leur somme. Il suffit de transformer au préalable le nombre retranché en son opposé. Cette conversion est très simple et très rapide.

Au contraire, en représentation « *signe et valeur absolue* », on aurait eu besoin de nombreux algorithmes, car plusieurs cas se présentent.

Remarque : la représentation en complément à 2 revient en fait à écrire (-1) comme ceci :

11111111

(-2) comme cela :

11111110

(-3) comme cela :

11111101

etc…

Il y a ainsi une symétrie entre les nombres positifs et les nombres négatifs. Il en résulte que le bit le plus à gauche représente le signe, de la même façon qu'en notation « *signe et valeur absolue* ». Avant tout calcul, nous pouvons donc affirmer que le nombre **1**0010101 est négatif.

II. LA MEMOIRE VIVE

1. La segmentation de la mémoire

Votre PC est conçu pour gérer 1 Mo (soit 2^{20} octets) de mémoire vive en mode réel. Il faut donc 20 bits au minimum pour adresser toute la mémoire. Or en mode réel les bus d'adresses n'ont que 16 bits. Ils permettent donc d'adresser 2^{16} = 65536 octets = 64 Ko, ce qui est insuffisant !

Afin de pallier ce manque, on utilise deux nombres pour adresser un octet quelconque de la RAM. Le premier est appelé *adresse de segment*, le second *adresse d'offset*. Ils seront stockés séparément.

La mémoire est ainsi découpée en *segments* de 64 Ko chacun. Un segment est donc en quelque sorte un gros bloc de mémoire auquel on peut accéder grâce à une *adresse de segment* qui désigne son numéro. Par exemple, le premier segment est le segment 0000 (en hexa), le deuxième est le 0001, le quarante-deuxième est le 0029, etc... Chaque numéro est codé sur 16 bits, c'est-à-dire 4 chiffres hexa.

Pour accéder à un octet particulier dans un segment, il suffit de compter le *décalage* de cet octet par rapport au début du segment. Ce décalage est obligatoirement inférieur ou égal à 65535 : il tient bien sur 16 bits lui aussi. On appelle ce décalage « *offset* ».

L'adresse d'un octet se note XXXX:YYYY où XXXX est l'adresse de segment et YYYY est l'offset (tous deux en notation hexadécimale, bien sûr).

Par exemple, le dix-septième octet de la RAM (le numéro 16) est situé à l'adresse 0000:0010. De même, l'octet 0000:0100 est l'octet numéro 256. Nous en arrivons à la petite subtilité qu'il convient de bien saisir, sous peine de ne rien comprendre à certains programmes en assembleur.

On pourrait penser que l'octet qui se trouve à l'adresse 0001:0000 est le numéro 65536. Il n'en est rien. C'est l'octet numéro 16.

Eh oui ! Les segments ne sont pas situés gentiment les uns à la suite des autres. Ils sont fort mal élevés et n'attendent pas que les segments qui les précédent soient terminés avant de commencer ! Ils se marchent donc sur les pieds.

Autrement dit, le deuxième segment ne démarre pas à l'octet 65536 comme il devrait le faire s'il était bien sage, mais à l'octet 16 ! Le troisième démarre à l'octet 32 et ainsi de suite…

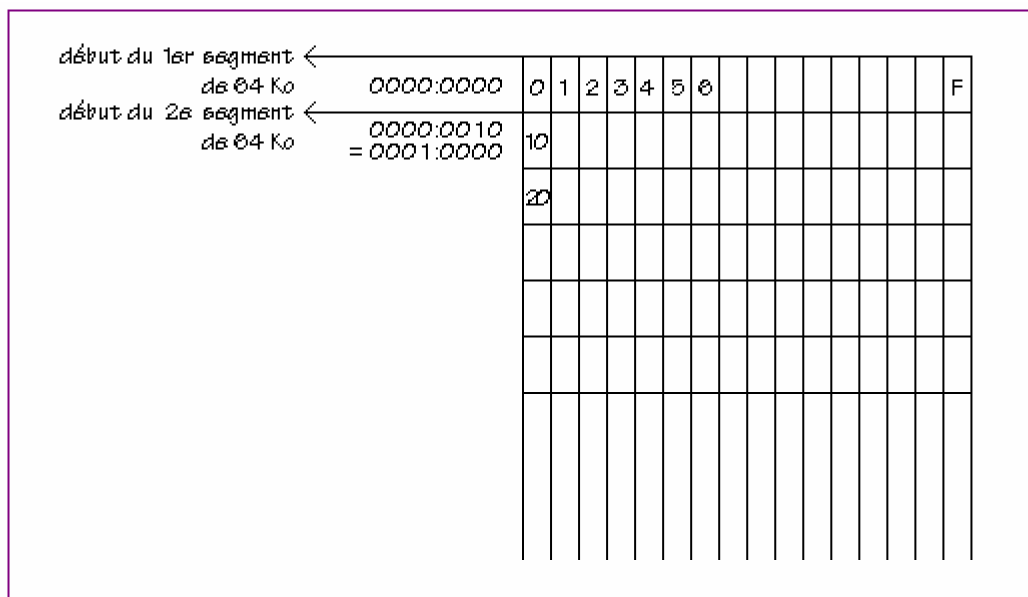
La notion de segment n'est pas tant physique que mathématique : elle sert à se repérer dans la RAM.

La conséquence immédiate de tout cela est qu'*un octet n'a pas une adresse unique.* Par exemple, l'octet numéro 66 peut être adressé par 0000:0042, mais aussi par 0001:0032, par 0002:0022, par 0003:0012 ou encore par 0004:0002. Toutes ces adresses sont équivalentes.

Voilà pour la subtilité. Si vous avez compris, vous devriez être capable de trouver facilement comment on calcule l'adresse effective d'un octet, c'est à dire sa position absolue dans la RAM.

Allez, un petit effort !

Voici la solution : si l'adresse de l'octet est A17C:022E, alors son adresse effective est A17C x 16 + 022E, soit A17C0 + 022E = A19EE. On a multiplié par 16, car le segment A17C débute à l'octet A17C x 16, puis on a simplement ajouté le décalage.
Au final, on a bien une adresse sur 20 bits puisqu'on obtient 5 chiffres hexa. Chaque petit bloc de 16 octets s'appelle un *paragraphe*.



La segmentation de la mémoire

Remarque importante : Il est souvent plus simple de considérer qu'un segment est un bloc *de taille quelconque* qui débute à une adresse effective multiple de 16 et qui permet, à l'aide de son *adresse de segment* et d'un *offset*, d'adresser le bloc entier (64 Ko au maximum). Cette définition est à notre avis celle qui a le plus de sens, et nous l'utiliserons tout au long de ce cours.

2. Structure d'un programme en mémoire

Lorsque l'utilisateur exécute un programme, celui-ci est d'abord chargé en mémoire par le système. Le DOS distingue deux modèles de programmes exécutables : les fichiers COM et les fichiers EXE.

La différence fondamentale est que les programmes COM ne peuvent pas utiliser plus d'un segment dans la mémoire. Leur taille est ainsi limitée à 64 Ko. Les programmes EXE ne sont quant à eux limités que par la mémoire disponible dans l'ordinateur.

a) les fichiers COM

Lorsqu'il charge un fichier COM, le DOS lui alloue toute la mémoire disponible. Si celle-ci est insuffisante, il le signale à l'utilisateur par un message et annule toute la procédure d'exécution. Dans le cas contraire, il crée le *PSP du programme* au début du bloc de mémoire réservé, et copie le programme à charger à la suite.

Mais qu'est-ce donc qu'un PSP ?

Pour simplifier, le PSP (« *Program Segment Prefix* ») est une zone de 256 (= 100h) octets qui contient des informations diverses au sujet du programme. C'est dans le PSP que se trouve la ligne de commande tapée par l'utilisateur. Par exemple, le PSP d'un programme appelé MONPROG, exécuté avec la commande "*MONPROG monfic.txt /S /H*", contiendra la chaîne de caractères suivante : " *monfic.txt /S /H*". Le programmeur a ainsi la possibilité d'accéder aux *paramètres*.

Voici à titre indicatif la structure simplifiée du PSP (ne vous souciez pas de ce que vous ne comprenez pas : pour l'instant, seules les deux dernières lignes nous intéressent vraiment) :

Offset	Description	Taille (octets)
00*h*	*Appel de l'int 20h*	2
02*h*	*Adresse du 1er segment qui se trouve au delà du prog.*	2
04*h*	*Réservé*	1
05*h*	*Far call de l'int 21h (inutilisé)*	5
0A*h*	*Vecteur de l'int 22h*	4
0E*h*	*Vecteur de l'int 23h*	4
12*h*	*Vecteur de l'int 24h*	4
16*h*	*Réservé*	22
2C*h*	*Segment du bloc d'environnement*	2
2E*h*	*Réservé*	82
80*h*	*Nombre de caractères dans la ligne de commande sans compter le code ASCII 13 (retour chariot)*	1
81*h*	*Ligne de commande (à partir du caractère espace qui suit le nom du programme) + code ASCII 13*	127

A présent que nous connaissons l'existence du PSP, il nous faut revenir sur un point important. Comme nous l'avons dit, un programme COM ne peut comporter qu'un seul segment, bien que le DOS lui réserve la totalité de la mémoire disponible. Ceci a deux conséquences. La première est que les adresses de segment sont inutiles dans le programme : les offsets seuls permettent d'adresser n'importe quel octet du segment. La seconde est *que le PSP fait partie de ce segment*, ce qui limite à *64 Ko – 256 octets* la taille maximale d'un fichier COM. Cela implique également que *le programme lui-même débute à l'offset 100h* et non à l'offset 0h.

b) les fichiers EXE

« *Pour un fichier EXE, tout est toujours un peu plus compliqué.* »
(Devise du programmeur débutant en assembleur)

Bien qu'il soit possible de n'utiliser qu'un seul segment à tout faire, la plupart des programmes EXE ont un segment réservé au *code* (c'est ainsi qu'on appelle les instructions du langage machine), un ou deux autres aux *données*, et un dernier à la *pile*.

La pile est une mémoire très spéciale qui sert comme son nom l'indique à *empiler* des données de 16 bits de façon temporaire. On peut ensuite retrouver ces données en les *dépilant.* Le « *dépilage* » se fait toujours dans l'ordre inverse de l'empilage.

Le PSP a lui aussi son propre segment. Le programme commence donc à l'offset 0h du segment de code et non à l'offset 100h.

Afin que le programme puisse être chargé et exécuté correctement, il faut que le système sache où commence et où s'arrête chacun de ces segments. A cet effet, les compilateurs créent un en-tête (ou « *header* ») au début de chaque fichier EXE. *Ce header ne sera pas copié en mémoire.* Son rôle est simplement d'indiquer au DOS (lors du chargement) la position relative de chaque segment dans le fichier.

3. Les cycles de lecture-écriture

Le code et les variables d'un programme se trouvent dans la mémoire vive. Pour accéder à une donnée en mémoire, le processeur place son adresse dans un *bus d'adresse*. Un *cycle de lecture* se met alors en place. Il consiste à retourner la donnée lue au processeur via le *bus de données*.

Pour l'écriture, l'adresse de la destination est transmise dans le bus d'adresse et la donnée à écrire est placée dans le bus de données.

Ouf !…

Nous aurons l'occasion de revenir sur la mémoire vive dans les autres chapitres pour apporter quelques précisions. Vous devriez mieux comprendre toutes ces notions en lisant la suite, car jusqu'ici nous n'avons pas encore parlé de la façon dont l'ordinateur se sert des adresses pour accéder à des données ou pour exécuter du code. C'est là qu'intervient le microprocesseur...

III. LE MICRO-PROCESSEUR – LES REGISTRES

1. Généralités sur les registres

Le microprocesseur est le cœur de l'ordinateur. C'est lui qui est chargé de reconnaître les instructions et de les exécuter (ou de les faire exécuter). Chaque instruction se présente sous la forme d'une suite de bits qu'on représente en notation hexadécimale (exemple : B44C, ou 1011010001001100 en binaire). Une instruction se compose d'un code opérateur (le plus souvent appelé « *opcode* ») qui désigne l'action à effectuer (B4 dans notre exemple) et d'un « *champ d'opérandes* » sur lesquelles porte l'opération (ici l'opérande est 4C).

Pour travailler, le microprocesseur utilise de petites zones où il peut stocker des données. Ces zones portent le nom de *registres* et sont très rapides d'accès puisqu'elles sont implantées dans le microprocesseur lui-même et non dans la mémoire vive.

Les registres des processeurs INTEL se classent en quatre catégories :

- *les registres **généraux** (16 bits)*
- *les registres **de segment** (16 bits)*
- *les registres **d'offset** (16 bits)*
- *le registre des **indicateurs** (16 bits)*

2. Les registres généraux

Ils ne sont pas réservés à un usage très précis, aussi les utilise-t-on pour manipuler des données diverses. Ce sont en quelque sorte des registres à tout faire. Chacun de ces quatre registres peut servir pour la plupart des opérations, mais ils ont tous une fonction principale qui les caractérise.

Nom	Fonction privilégiée
AX	*Accumulateur*
BX	*Base*
CX	*Compteur*
DX	*Données*

Le registre AX sert souvent de registre d'entrée-sortie : on lui donne des paramètres avant d'appeler une fonction ou une procédure. Il est également utilisé pour de nombreuses opérations arithmétiques, telles que la multiplication ou la division de nombres entiers. Il est appelé « *accumulateur* ».

Exemples d'utilisation :

- l'instruction "MOV AX, *1982*" place l'entier 1982 dans AX.
- "ADD AX, *1983*" ajoute à AX le nombre 1983 et place le résultat dans AX.

Le registre BX peut servir de *base*. Nous verrons plus tard ce que ce terme signifie.

Le registre CX est utilisé comme compteur dans les boucles. Par exemple, pour répéter 15 fois une instruction en assembleur, on peut mettre la valeur 15 dans CX, écrire l'instruction précédée d'une étiquette qui représente son adresse en mémoire, puis faire un LOOP à cette adresse. Lorsqu'il reconnaît l'instruction LOOP, le processeur « sait » que le nombre d'itérations à exécuter se trouve dans CX. Il se contente alors de décrémenter CX, de vérifier que CX est différent de 0 puis de faire un saut (« *jump* ») à l'étiquette mentionnée. Si CX vaut 0, le processeur ne fait pas de saut et passe à l'instruction suivante.

Exemple :

```
;(...)

mov cx, 15 ;mettre 15 dans CX

Toto_est_beau:

;écrire les instructions à répéter ici
;(...)

loop Toto_est_beau

;(...)
```

Le registre DX contient souvent l'adresse d'un tampon de données lorsqu'on appelle une fonction du DOS. Par exemple, pour écrire une chaîne de caractères à l'écran, il faut placer l'offset de cette chaîne dans DX avant d'appeler la fonction appropriée.

Chacun de ces quatre registres comporte 16 bits. On peut donc y stocker des nombres allant de 0 à 65535 en arithmétique non signée, et des nombres allant de –32768 à 32767 en arithmétique signée. Les 8 bits de poids fort d'un registre sont appelés « *partie haute* » et les 8 autres « *partie basse* ». Chaque registre est en fait constitué de deux « sous-registres » de 8 bits. La partie haute de AX s'appelle AH, sa partie basse AL. Il en va de même pour les trois autres.

En arithmétique non signée, il en découle la formule suivante :

AX = AH x 256 + AL

ainsi que leurs homologues :

BX = BH x 256 + BL

CX = CH x 256 + CL

DX = DH x 256 + DL

Il va de soi qu'en modifiant AL ou AH, on modifie également AX.

Pour stocker un nombre de 32 bits, on peut utiliser des paires de registres. Par exemple, DX:AX signifie DX x 65535 + AX en arithmétique non signée. Cette notation (DX:AX) n'est pas reconnue par l'assembleur (ni par la machine). Ne confondez pas cela avec les adresses de segment et d'offset.

Sur un PC relativement récent, AX, BX, CX, DX ne sont en fait que les parties basses de registres de 32 bits nommés EAX, EBX, ECX, EDX (« E » pour « Extended »). On a donc un moyen plus pratique de stocker les grands nombres. En fait, chaque registre (pas seulement les registres généraux) peut contenir 32 bits.

3. Les registres de segment

Ils sont au nombre de quatre :

Nom	Nom complet	Traduction
CS	*Code segment*	Segment de code
DS	*Data segment*	Segment de données
ES	*Extra segment*	Extra segment
SS	*Stack segment*	Segment de pile

Contrairement aux registres généraux, ces registres ne peuvent servir pour les opérations courantes : *ils ont un rôle très précis*. On ne peut d'ailleurs pas les utiliser aussi facilement que AX ou BX, et une petite modification de l'un d'eux peut suffire à « planter » le système. Eh oui ! L'assembleur, ce n'est pas Turbo Pascal ! Il n'y a aucune barrière de protection, si bien qu'une petite erreur peut planter le DOS. Mais rassurez-vous : tout se répare très bien en redémarrant l'ordinateur…

Dans le registre CS est stockée l'adresse de segment de la prochaine instruction à exécuter. La raison pour laquelle il ne faut surtout pas changer sa valeur directement est évidente. De toute façon, vous ne le pouvez pas. Le seul moyen viable de le faire est d'utiliser des instructions telles que des sauts (JMP) ou des appels (CALL) vers un autre segment. CS sera alors automatiquement actualisé par le processeur en fonction de l'adresse d'arrivée.

Le registre DS est quant à lui destiné à contenir l'adresse du segment des données du programme en cours. On peut le faire varier à condition de savoir exactement pourquoi on le fait. Par exemple, on peut avoir deux segments de données dans son programme et vouloir accéder au deuxième. Il faudra alors faire pointer DS vers ce segment.

ES est un registre qui sert à adresser le segment de son choix. On peut le changer aux mêmes conditions que DS. Par exemple, si on veut copier des données d'un segment vers un autre, on pourra faire pointer DS vers le premier et ES vers le second.

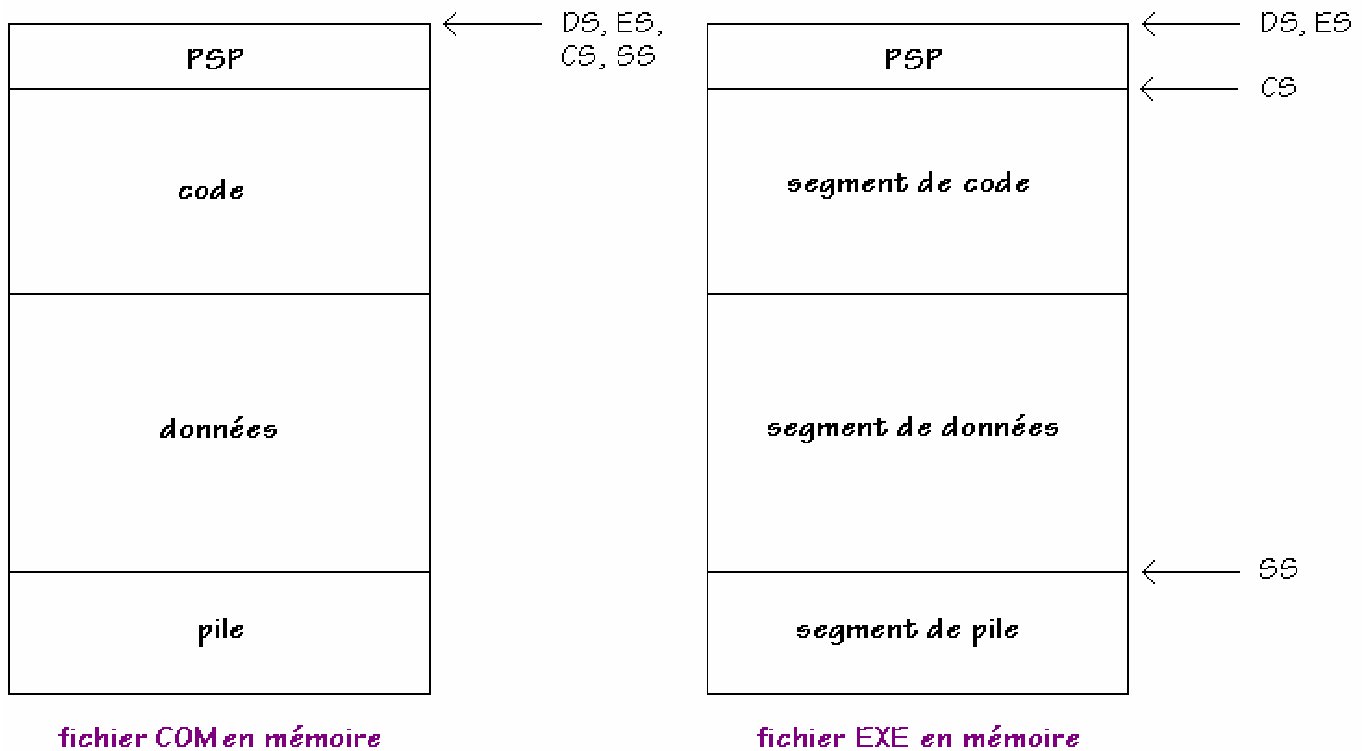
Le registre SS adresse le segment de pile. Il est rare qu'on doive y toucher car le programme n'a qu'une seule pile.

Intéressons-nous à présent aux valeurs que le DOS donne à ces registres lors du chargement en mémoire d'un fichier exécutable ! Elles diffèrent selon que le fichier est un programme COM ou EXE. Pour écrire un programme en assembleur, il est nécessaire de connaître ce tableau par cœur :

Registre	Valeur avant l'exécution	
	Fichier COM	Fichier EXE
CS	Adresse de l'unique segment, c'est à dire adresse de segment du PSP	Adresse du segment de code
DS	Adresse de l'unique segment, c'est à dire adresse de segment du PSP	Adresse de segment du PSP
ES	Adresse de l'unique segment, c'est à	Adresse de segment du PSP

	dire adresse de segment du PSP	
SS	Adresse de l'unique segment, c'est à dire adresse de segment du PSP	Adresse du segment de pile

Le schéma suivant montre mieux la situation :



Dans un fichier EXE, le *header* indique au DOS les adresses initiales de chaque segment *par rapport au début du programme* (puisque le compilateur n'a aucun moyen de connaître l'adresse à laquelle le programme sera chargé). Lors du chargement, le DOS ajoutera à ces valeurs l'*adresse d'implantation* pour obtenir ainsi les véritables adresses de segment. Dans le cas d'un fichier COM, tout est plus simple. Le programme ne comporte qu'un seul segment, donc il suffit tout bêtement au DOS de charger CS, DS, ES et SS avec l'adresse d'implantation.

Remarque : *Pourquoi DS et ES pointent-ils vers le PSP dans le cas d'un fichier EXE ?* Première raison : pour que le programmeur puisse accéder au PSP ! Deuxième raison : parce qu'un programme EXE peut comporter un nombre quelconque de segments de données. *C'est donc au programmeur d'initialiser ces registres*, s'il veut accéder à ses données.

4. Les registres d'offset

Les voici :

Nom	Nom complet	Traduction
IP	*Instruction pointer*	Pointeur d'instruction
SP	*Stack pointer*	Pointeur de pile
SI	*Source index*	Index de source
DI	*Destination index*	Index de destination

BP	*Base pointer*	Pointeur de base

Le registre IP désigne l'offset de la prochaine instruction à exécuter, par rapport au segment adressé par CS. La combinaison de ces deux registres (i.e. CS:IP) suffit donc à connaître l'adresse absolue de cette instruction. Le processeur peut alors aller la chercher en mémoire et l'exécuter. De plus, il actualise IP en l'incrémentant de la taille de l'instruction en octets. Tout comme CS, il est impossible de modifier IP directement.

Le registre SP désigne le sommet de la pile. Il faut bien comprendre le fonctionnement de la pile, aussi allons-nous insister sur ce point. La pile ne peut stocker que des *mots*. On appelle un mot (« *word* » en anglais) un nombre codé sur deux octets (soit 16 bits). Prenons un exemple simple : un programme COM. Le segment de pile (adressé par SS) et le segment de code ne font qu'un. Avant l'exécution, SP vaut FFFE. C'est l'offset de la dernière donnée de 16 bits empilée, par rapport à SS bien sûr. Pourquoi FFFE ? Tout simplement parce que la pile se remplit à l'envers, c'est-à-dire en partant de la fin du segment et en remontant vers le début. Le premier mot empilé se trouve à l'offset FFFE. Il tient sur deux octets : l'octet FFFE et l'octet FFFF. Mais comment se fait-il qu'un mot soit déjà empilé avant le début du programme ? Ce mot est un zéro que le DOS place sur la pile avant l'exécution de tout programme COM. Nous en verrons la raison plus tard.

A présent, que se passe-t-il si à un instant quelconque une instruction ordonne au processeur d'empiler un mot ? Eh bien le *stack pointer* sera *décrémenté* de 2 et le mot sera copié à l'endroit pointé par SP. Rappelez-vous que la pile se remplit à l'envers ! C'est pour cette raison que SP est décrémenté à chaque empilage et non pas incrémenté.

Un petit exemple pour rendre les choses plus concrètes :

PUSH AX

L'effet de cette instruction est d'empiler le mot contenu dans le registre AX. Autrement dit, SP est automatiquement décrémenté de 2, puis AX est copié à l'adresse SS:SP.

Lors du dépilage, le mot situé au sommet de la pile, c'est-à-dire le mot adressé par SS:SP, est transféré dans un registre quelconque choisi par le programmeur, après quoi le stack pointer est incrémenté de 2.

Exemple :

POP BX

Cette fois, on retire le dernier mot empilé pour le placer dans le registre BX. Evidemment, SP sera incrémenté de 2 aussitôt après.

La pile est extrêmement utile lorsqu'il s'agit de stocker provisoirement le contenu d'un registre qui doit être modifié.

Exemple :

```
;(...)

push ax ;sauvegarde le contenu de AX sur la pile

;ici se trouvent les instructions qui modifient AX

;(...)

pop ax ;restaure le contenu de AX

;(...)
```

Il est important de comprendre qu'on ne peut dépiler que le mot qui se trouve au sommet de la pile. Le premier mot empilé est le dernier qui sera dépilé. La pile doit être manipulée avec une extrême précaution. Un dépilage injustifié fait planter la machine presque systématiquement.

Les trois derniers registres sont beaucoup moins liés au fonctionnement interne du processeur. Ils sont mis à la disposition du programmeur qui peut les modifier à sa guise et les utiliser comme des registres généraux. Comme ces derniers cependant, ils ont une fonction qui leur est propre : servir d'*index* (SI et DI) ou de *base* (BP). Nous allons expliciter ces deux termes.

Dans la mémoire, les octets se suivent et forment parfois des *chaînes de caractères*. Pour utiliser une chaîne, le programmeur doit pouvoir accéder facilement à tous ses octets, l'un après l'autre. Or pour effectuer une opération quelconque sur un octet, il faut connaître son adresse. Cette adresse doit en général être une *constante évaluable par le compilateur*. Pourquoi une constante ? Parce que l'adresse est une opérande comme les autres, elle se trouve immédiatement après l'opcode et doit donc avoir une valeur numérique fixe !

Prenons un exemple :

MOV AH, [*MonOctet*]

Pas de panique ! Cette instruction en assembleur signifie « *Mettre dans AH la valeur de l'octet adressé par le label MonOctet !* ». A la compilation, "*MonOctet*" sera remplacé par la valeur numérique qu'il représente et on obiendra alors une instruction en langage machine telle que :

8A260601

8A26 est l'opcode (hexa) de l'instruction "MOV AH, [*constante quelconque*]", et 0601 est l'offset de "*MonOctet*".

Il serait pourtant fastidieux, dans le cas d'une chaîne de 112 caractères, de traiter les octets avec 112 instructions dans lesquelles seule l'adresse changerait. Il faudrait pouvoir faire une boucle sur l'adresse, mais alors celle-ci ne serait plus une constante, d'où le problème.

Pour les constructeurs du microprocesseur, la seule solution était de créer de nouveaux opcodes pour chaque opération portant sur un octet en mémoire. Ces opcodes spéciaux feraient la même action que ceux dont ils seraient dérivés, mais l'adresse passée en paramètre serait alors considérée comme un *décalage par rapport à un registre spécial.* Il suffirait donc de faire varier ce registre, et le processeur y ajouterait automatiquement la valeur de l'opérande pour obtenir l'adresse réelle ! C'est à cela que servent SI, DI et BP.

Par exemple :

MOV AH, [*MonOctet* + DI]

sera codé :

8AA50601

8AA5 est l'opcode pour l'instruction "MOV AH, [*constante quelconque* + DI]".

Remarque : les registres SI et BP auraient tout aussi bien pu être employés, mais pas les registres généraux, **SAUF BX**. En effet, BX peut jouer exactement le même rôle que BP. N'oubliez pas que BX est appelé registre de « *base* », et que BP signifie « *Base Pointer*. » Nous verrons la différence entre une base et un index lorsque nous commencerons l'assembleur.

5. Le registre des indicateurs

Un programme doit pouvoir faire des choix en fonction des données dont il dispose. Pour cela, il lui faut par exemple comparer des nombres, examiner leur signe, découvrir si une erreur a été constatée, etc…

Il existe à cet effet de petits indicateurs, les *flags* qui sont des bits spéciaux ayant une signification très précise. De manière générale, *les flags fournissent des informations sur les résultats des opérations précédentes.*

Ils sont tous regroupés dans un registre : le *registre des indicateurs*. *Comprenez bien que chaque bit a un rôle qui lui est propre et que la valeur globale du registre ne signifie rien.* Le programmeur peut lire chacun de ces flags et parfois modifier leur valeur directement.

En mode réel, certains flags ne sont pas accessibles. Nous n'en parlerons pas. Nous ne commenterons que les flags couramment utilisés.

Nous verrons quelle utilisation on peut faire de ces indicateurs dans la troisième partie de ce tutoriel.

CF (« *Carry Flag* ») est l'indicateur de retenue. Il est positionné à 1 si et seulement si l'opération précédemment effectuée a produit une retenue. De nombreuses fonctions du DOS l'utilisent comme indicateur d'erreur : CF passe alors à 1 en cas de problème.

PF (« *Parity Flag* ») renseigne sur la parité du résultat. Il vaut 1 ssi ce dernier est pair.

AF (« *Auxiliary Carry Flag* ») est peu utilisé.

ZF (« *Zero Flag* ») passe à 1 ssi le résultat d'une opération est égal à zéro.

SF (« *Sign Flag* ») passe à 1 ssi le résultat d'une opération sur des nombres signés est négatif.

TF (« *Trap Flag* ») est utilisé pour le « *débuggage* » d'un programme. S'il vaut 1, une routine spéciale du débuggeur est appelée après l'exécution de chaque instruction par le processeur.

IF (« *Interrupt Flag* ») sert à empêcher les appels d'interruptions lorsqu'il est positionné à 1. Cependant, toutes les interruptions ne sont pas « *masquables* ».

DF (« *Direction Flag* ») est utilisé pour les opérations sur les chaînes de caractères. S'il vaut 1, celles-ci seront parcourues dans le sens des adresses décroissantes, sinon les adresses seront croissantes.

OF (« *Overflow Flag* ») indique qu'un débordement s'est produit, c'est-à-dire que la capacité de stockage a été dépassée. Il est utile en arithmétique *signée*. Avec des nombres non signés, il faut utiliser ZF et SF.

Remarque : Les notations CF, PF, AF, etc… ne sont pas reconnues par l'assembleur. Pour utiliser les flags, il existe des instructions spécifiques que nous décrirons plus tard.

IV. LES INTERRUPTIONS

1. Introduction

Le microprocesseur ne peut exécuter qu'une seule instruction à la fois. Pour connaître son adresse, il utilise le couple de registres CS:IP dont la valeur est incrémentée automatiquement. Par conséquent, le code du programme courant est exécuté *de manière linéaire*.

Imaginons cependant qu'un événement extérieur demande l'attention de l'ordinateur, par exemple la pression d'une touche du clavier. La machine doit pouvoir réagir *immédiatement*, sans attendre que le programme en cours d'exécution se termine. Pour cela, elle interrompt ce dernier pendant un bref instant, le temps de traiter l'événement survenu puis rend le contrôle au programme interrompu.

Une interruption n'est rien d'autre que l'appel d'une routine spéciale présente en mémoire appelée ISR (« Interrupt Service Routine »).

Les interruptions se divisent en trois catégories :

- les interruptions **électroniques**, par exemple : le clavier
- les interruptions du **BIOS**, par exemple : l'accès aux disques
- les interruptions du **DOS**, par exemple : l'accès aux systèmes de fichiers

Comment sont-elles déclenchées ?

Une interruption peut être déclenchée par votre matériel. C'est ce qui arrive lorsque vous appuyez sur une touche du clavier. Aucun logiciel n'intervient et le contrôle est passé directement à la routine qui gère le clavier : ce sont les *interruptions matérielles*.

Les *interruptions logicielles* sont quant à elles appelées par des *instructions* en langage machine au sein d'un programme. Leur importance est capitale. Rappelez-vous que contrairement au PASCAL ou au C, l'assembleur ne dispose pas de fonction préprogrammée. Chaque instruction doit être directement traduisible en langage machine.

Mais alors, comment fait-on pour écrire une chaîne de caractères à l'écran ? Ou bien pour lire un caractère entré au clavier ?

Eh bien on le fait de la même façon que le DOS lui-même ! On déclenche les interruptions appropriées à l'aide de l'instruction « INT » du langage machine. C'est donc une routine du DOS (ou parfois du BIOS) qui fera tout le travail. Les paramètres (ou leurs adresses) sont passés dans les registres.

Voici un petit exemple en assembleur qui écrit la lettre 'A' à l'écran :

```
;(...)

mov dl, 'A'
mov ah, 02
int 21h

;(...)
```

Examinons-le ligne par ligne :

- l'instruction "MOV DL, '*A*'" demande au processeur de mettre dans le registre DL le code ASCII de la lettre 'A', c'est-à-dire 65, ou 41h.

- "MOV AH, *02"* : mettre le nombre 2 dans AH.

- Enfin, la dernière instruction appelle l'interruption numéro *21h*. Il existe 256 interruptions. *Toutes sont notées en base hexadécimale.*

Explications :

Comme vous aurez très vite l'occasion de vous en rendre compte, l'interruption 21h est l'interruption du DOS par excellence. Elle permet d'appeler de nombreuses fonctions très diverses. Pour cela, il suffit de mentionner leur numéro dans le registre AH.

Il est très difficile de mémoriser le rôle de chaque interruption, et a fortiori de chaque fonction ou sous-fonction, d'autant plus d'elles sont désignées par des numéros hexadécimaux et qu'elles attendent des paramètres dans des registres précis. C'est pourquoi tout programmeur se doit d'avoir à sa disposition une liste des interruptions pour travailler. Celle de Ralph Brown est très connue, et vous la trouverez sur l'Internet.

Revenons à notre exemple. La fonction numéro 2 de l'interruption 21h sert à écrire un caractère à l'écran. Il faut pour cela écrire le code ASCII du caractère dans le registre DL et bien sûr placer le nombre 2 dans AH.

Une fois que AH et DL ont été ajustés, l'interruption 21h peut être appelée à l'aide de l'instruction INT.

D'autres interruptions ne remplissent qu'une seule tâche. Vous n'avez donc pas besoin de mettre un numéro de fonction dans AH. L'appel de l'interruption suffit.

2. La table des vecteurs d'interruptions

A chaque appel d'interruption, l'ordinateur doit pouvoir trouver l'adresse de l'ISR correspondante. Pour cela, il dispose de *la table des vecteurs d'interruptions* (TVI, ou IVT : « *Interrupt Vector Table* »). Cette table est implantée à l'adresse 0000:0000 c'est-à-dire au début de la RAM.

La table est organisée comme suit :

Adresse	Taille (octets)	Valeur
0000:0000	2	Adresse d'offset de l'interruption numéro 0
0000:0002	2	Adresse de segment de l'interruption numéro 0

0000:0004	2	Adresse d'offset de l'interruption numéro 1
0000:0006	2	Adresse de segment de l'interruption numéro 1
0000:0008	2	Adresse d'offset de l'interruption numéro 2
0000:000A	2	Adresse de segment de l'interruption numéro 2
0000:000C	2	Adresse d'offset de l'interruption numéro 3
0000:000E	2	Adresse de segment de l'interruption numéro 3
etc… etc… etc…		
0000:03FC	2	Adresse d'offset de l'interruption numéro 255
0000:03FE	2	Adresse de segment de l'interruption numéro 255

La raison pour laquelle les offsets précèdent les adresses de segment vient du codage en *« little endian »* utilisé par INTEL. Les processeurs de cette marque (contrairement à la plupart des autres, qui travaillent en « *big endian* ») ont une représentation des données en mémoire aussi curieuse qu'insupportable pour le programmeur : ils placent le poids fort après le poids faible. Ainsi, le mot 4A28h sera codé 284Ah. Puisque l'adresse de l'ISR tient sur 32 bits (16 + 16), elle est représentée comme un double mot (« *dword* »), donc l'adresse d'offset, qui est le mot de poids faible, se trouve au début. Eh oui, c'est pénible, mais il faudra s'y faire !

Remarque : pour calculer l'adresse de l'entrée dans la TVI correspondant à l'interruption numéro X, il suffit de multiplier X par 4.

Exemple : l'adresse de l'ISR numéro 21h est stockée à 0000:0084.

3. Sauvegarde de l'état des registres lors de l'appel

Un appel d'interruption obéit à certaines règles, car il est indispensable, une fois l'ISR exécutée, que le programme interrompu retrouve les registres dans le même état qu'ils étaient auparavant. Tout doit se passer comme si l'interruption n'avait jamais eu lieu. C'est pourquoi avant de faire un saut à l'ISR pointée par l'entrée correspondante dans la TVI, l'ordinateur sauvegarde tous les registres sur la pile courante. Il restaurera leur contenu avant de rendre le contrôle. Cette procédure est automatique. Il n'incombe pas au programmeur de prendre toutes ces précautions.

DEUXIEME PARTIE

PREMIER CONTACT AVEC LE LANGAGE ASSEMBLEUR

Cette partie a pour but de vous présenter l'architecture du langage assembleur à travers de courts exemples. Ne vous inquiétez pas si vous ne comprenez pas parfaitement certaines instructions : toutes seront détaillées dans la troisième partie. *L'essentiel est que vous compreniez grossièrement ce qu'on fait et pourquoi on le fait.* Les explications qui accompagnent ces exemples devraient y suffire. Vous pourrez ensuite apprendre le langage en lui-même en consultant la partie suivante.

<u>Remarques pratiques préliminaires</u> :

- *Les mots qui sont imprimés en italique sont ceux qui ne font pas partie du langage en lui-même. Ils sont choisis par le programmeur.*

- *Tous les programmes présentés ci-dessous suivent la syntaxe du compilateur TASM. Il existe de très légères différences d'un compilateur à l'autre.*

- *Le code peut être écrit en majuscules ou en minuscules. Chaque mot est séparé des autres par des espaces ou des tabulations.*

- *Les commentaires sont précédés du signe ';'.*

- *Les nombres doivent toujours commencer par un chiffre (entre 0 et 9), même les nombres hexadécimaux. Il faut donc écrire 0F2Ah et non pas F2Ah. Par contre, l'écriture 5CFh est correcte.*

- *Les apostrophes et les guillemets sont équivalents. Il est cependant plus commode de réserver l'usage des guillemets aux chaînes de plusieurs caractères et celui des apostrophes aux caractères isolés.*

- *Vous pouvez taper vos programmes avec l'éditeur de texte du DOS (EDIT.COM). Donnez-leur l'extension '.asm'.*

- *La compilation se fait en tapant "TASM /m9 MONPROG" si votre source s'appelle 'MONPROG.ASM'.*

  Le paramètre '/m9', facultatif, indique au compilateur qu'il devra effectuer 9 passes, c'est-à-dire qu'il examinera 9 fois le code source. Comme chaque examen se fait de manière linéaire, le compilateur trouve parfois des instructions qui font référence à des labels placés plus loin dans le programme. Il manque donc d'informations, mais il continue son examen jusqu'à la fin. Quand il a terminé, il recommence tout depuis le début pour résoudre les problèmes qui s'étaient posés.

  La compilation crée un fichier objet ('.obj'). Pour obtenir un fichier EXE, tapez "TLINK MONPROG". Pour un fichier COM, tapez "TLINK /tdc MONPROG".

  Après l'édition des liens (le « linkage »), vous pouvez supprimer les fichiers

MONPROG.obj' et 'MONPROG.map'.

- *Il est possible de créer un fichier BAT qui s'occupe de toutes ces étapes. Ouvrez l'éditeur EDIT du DOS et tapez un programme tel que celui-ci :*

```
@ECHO OFF
TASM /m9 %1.asm
IF NOT EXIST %1.obj GOTO FIN
TLINK %1.obj REM : ajouter ici /tdc pour obtenir un fichier COM
ERASE %1.map
ERASE %1.obj
:FIN
```

  Enregistrez-le et nommez-le MAKE.BAT. Vous pouvez compiler et linker en tapant "MAKE MONPROG". Pour compiler des fichiers COM, ajoutez le paramètre '/tdc' à la commande TLINK.

- *Pour pouvoir lancer TASM et TLINK quel que soit le dossier courant, changez la variable PATH ainsi :*

```
PATH = %path%;c:\MonChemin\DossierTASM
```

  Vous pouvez inclure cette ligne à la fin de votre fichier AUTOEXEC.BAT afin qu'elle soit exécutée à chaque démarrage.

- *Vous pouvez débugger vos programmes (par exemple les exécuter instruction par instruction en observant les changements induits dans la RAM et dans les registres,...) avec le logiciel Turbo Debugger 16/32 bits qui est très pratique et très performant. A défaut, vous pouvez vous rabattre sur l'archaïque DEBUG.COM (il se trouve dans votre dossier de commandes DOS). Mais alors bon courage !*

Le seul travail du compilateur (et du linkeur) est de convertir chacune de vos instructions en son équivalent en langage machine. *Le programme compilé présentera donc exactement la même structure et la même linéarité que votre code source.*

I. PREMIER EXEMPLE : LES FICHIERS COM

Voici un petit programme COM qui écrit le message « Bonjour, monde ! » à l'écran.

```
.386

code segment use16

assume cs:code, ds:code, ss:code

org 100h ;offsets décalés de 100h = 256

debut :

mov ah, 09h ;fonction n°9 : écrire une chaîne à l'écran
mov dx, offset message ;mettre l'offset de la chaîne dans DX
int 21h ;écrire la chaîne à l'écran en appelant l'interruption 21h

ret ;rendre la main au DOS

message db "Bonjour, monde !", '$' ;définition du message

code ends

end debut
```

Ce code source commence par des *directives*. Une directive est une information que le programmeur fournit au compilateur. *Elle n'est pas transformée en une instruction en langage machine.* Elle n'ajoute donc aucun octet au programme compilé.

La directive ".386" indique au compilateur que le programme est destiné à tourner sur des processeurs INTEL de modèle 386 (ou supérieur). Cela nous autorise à utiliser certaines instructions qui ne sont pas disponibles sur les modèles antérieurs, comme PUSHA ou POPA. Dans cet exemple, cette directive aurait très bien pu être omise.

La ligne

code segment use16

sert à déclarer un segment que l'on appelle "*code*". On aurait tout aussi bien pu le nommer "*marteau*" ou "*voiture*". Ce sera le segment de notre programme. N'oubliez pas qu'un fichier COM ne peut comporter qu'un seul segment. Cette ligne ne sera pas compilée : elle ne sert qu'à indiquer au compilateur le début d'un segment.

Le mot "use16" indique que les adresses de segment et d'offset sont codées sur 16 bits et non sur 8 bits. Vous devez systématiquement l'écrire.

La directive

assume cs:*code*, ds:*code*, ss:*code*

informe le compilateur que tout au long du programme, CS, DS et SS pointeront de façon privilégiée vers le segment "*code*", ce qui est évident d'ailleurs puisque c'est le seul segment… Vous comprendrez le sens exact de "assume" dans la troisième partie du cours.

Enfin, les mots

org *100h*

signifient qu'il faudra ajouter 100h (soit 256) à tous les offsets. Pourquoi ? Souvenez-vous de la structure d'un programme COM en mémoire. Si vous n'écrivez pas cette ligne, TASM considérera que le programme débute à l'offset 0000. Or, lors de l'exécution, le DOS le chargera après le PSP, c'est-à-dire à l'adresse 100h. C'est pourquoi il est nécessaire de recalculer les offsets : l'offset 0000 deviendra 0100. *Cette directive est en quelque sorte le trait caractéristique des fichiers COM.*

Nous en arrivons au programme proprement dit. Il commence par un label :

debut :

Lui non plus n'est pas compilé. Il ne sert qu'à représenter l'adresse de l'instruction qui le suit, c'est-à-dire :

mov ah, *09h*

Cette ligne demande au processeur de charger la valeur 9 dans le registre AH. C'est le numéro de la fonction de l'interruption 21h qui écrit une chaîne de caractères à l'écran. L'offset de cette chaîne est attendu dans DX. D'où la ligne suivante :

mov dx, offset *message*

Le mot-clé "offset" sert à extraire l'offset du label "*message*" qui représente quant à lui l'adresse du message à écrire (il contient donc une adresse de segment ET un offset). L'adresse de segment de la chaîne doit être transmise dans DS. Mais il est inutile de changer ce registre, car il pointe déjà vers notre segment.

Il nous reste à appeler l'interruption 21h :

int *21h*

et à rendre la main au DOS :

ret

Lorsque nous aborderons les procédures, vous comprendrez mieux le sens de ce mot et ce qu'il fait exactement. Pour l'instant, sachez simplement que seul un fichier COM peut se terminer avec cette instruction.

Nous arrivons à la ligne :

message db "*Bonjour, monde !*", '*$*'

Il s'agit d'une définition de données. Le mot "db" (« *define byte* ») signifie que le compilateur devra écrire les octets qui suivent tels qu'ils sont dans notre code source. Il va donc écrire le code ASCII du 'B', puis celui du 'o', etc… Il terminera en écrivant le code ASCII du signe '$'.

C’est ainsi que la fonction 9 de l’interruption 21h reconnaît la fin de la chaîne à écrire. Si vous oubliez ce signe, elle écrira tous les octets de la RAM jusqu’à ce qu’elle tombe par hasard sur lui.

Le mot “*message*” placé en début de ligne est un label de données. Il représente *l’adresse* du code ASCII du ‘B’. *Remarquez qu’il n’y a pas de caractère ‘:’ après un label de données.*

La ligne

code ends

indique la fin du segment “*code*”, et enfin

end *debut*

informe le compilateur que le fichier est fini, tout comme le “END.” du PASCAL. Le nom du label “*debut*” est mentionné : ce sera le point d’entrée de notre programme. C’est vers lui que pointera CS:IP avant l’exécution.

Remarque : vous n’êtes pas tenu de rendre aux registres la valeur qu’ils avaient au début de votre programme. De toute façon, avant de charger un programme, le DOS sauvegarde le contenu de tous les registres puis met le contenu des registres généraux (ainsi que SI, DI et BP) à zéro. Il les restaurera quand vous lui rendrez la main.

II. DEUXIEME EXEMPLE : LES FICHIERS EXE

Le programme suivant est un programme EXE qui fait exactement la même chose que l'exemple précédent.

```
.386

code segment use16

assume cs:code, ds:data, ss:pile

debut :

mov ax, data ;charger ds avec
mov ds, ax ;l'adresse du segment data

mov ah, 09h
mov dx, offset message
int 21h

mov ah, 4ch ;4C = numéro de fonction de l'int. 21h qui sert à quitter
int 21h

code ends



data segment use16

message db "Bonjour, monde !", '$'

data ends



pile segment stack

remplissage db 256 DUP (?)

pile ends



end debut
```

Examinons-le !

Tout d'abord, la directive

org *100h*

a disparu.

En effet, lors de l'exécution, CS pointera vers notre segment de code et non pas vers le PSP. Il est donc inutile de décaler les offsets de 256.

Remarquons que notre programme dispose de deux segments supplémentaires :

- Le segment "*data*" est destiné à contenir les données, c'est-à-dire les variables.

- Le segment "*pile*" sera notre segment de pile.

Notre directive "assume" devient donc :

assume cs:*code*, ds:*data*, ss:*pile*

Ainsi, le compilateur est informé que DS pointera vers le segment "*data*" et SS vers "*pile*".

Les deux lignes suivantes,

```
mov ax, data
mov ds, ax
```

servent à initialiser le registre DS. Celui-ci pointe vers le PSP au début du programme mais nous voulons le faire pointer vers notre segment de données appelé "*data*". Cela est nécessaire puisque la fonction 9 de l'interruption 21h attend l'adresse de la chaîne dans le couple DS:DX et que notre message se trouve dans le segment de données.

La première instruction charge l'adresse du segment "*data*" dans AX. La seconde transfère cette valeur de AX dans DS.

Mais pourquoi diable utiliser AX comme intermédiaire ? Après tout, on pourrait écrire :

```
mov ds, data
```

Eh bien non ! *Pour la simple raison que DS est un registre de segment et qu'en tant que tel on ne peut pas lui charger de valeur immédiate.*

On appelle « *valeur immédiate* » toute constante tapée directement dans l'instruction elle-même.

Exemples de chargement de valeurs immédiates :

mov ax, *135* *;charge 135 dans AX*

mov bx, offset *message* *;charge l'offset de message dans BX*

mov bx, offset *fin* – offset *debut* *;charge le nombre d'octets entre fin et debut dans BX*

mov es, *10* *;instruction illicite car ES est un registre de segment !*

Remarque : une autre possibilité aurait été d'écrire : "PUSH AX" (empiler AX) puis "POP DS" (dépiler le dernier nombre empilé et le placer dans DS).

Les trois lignes suivantes :

```
mov ah, 09h
mov dx, offset message
int 21h
```

n'ont pas changé.

Il nous faut également terminer le programme par un appel de la fonction 4ch de l'interruption 21h. C'est ainsi que se terminent les programmes EXE.

```
mov ah, 4ch
int 21h
```

Remarque : on aurait également pu écrire :

```
mov ax, 4c00h
int 21h
```

La seule différence est que AL est mis à zéro, ce qui indique au programme à qui on rend le contrôle (ici le DOS) que notre programme s'est terminé normalement. Les fichiers COM peuvent également utiliser la fonction 4ch.

Le segment de code se termine :

```
code ends
```

et le segment de données commence à sa suite :

```
data segment use16

message db "Bonjour, monde !", '$'

data ends
```

Il nous reste à écrire le segment de pile. Dans ce programme, il n'était pas absolument indispensable de le séparer du segment de code. Mais c'est une bonne habitude de le faire.

```
pile segment stack

remplissage db 256 DUP (?)

pile ends
```

Le mot-clé "stack" indique que ce segment est le segment de pile.

Les mots "db *256* DUP (?)" déclarent 256 octets non initialisés. C'est la « matière » de notre pile.

Sachez que tout appel d'interruption se traduit par l'empilage des flags et de CS:IP. Il est donc indispensable d'avoir une pile, même si celle-ci peut éventuellement partager le même segment que le code, comme dans un fichier COM. Mais dans un programme EXE, il vaut mieux réserver un segment à la pile. Les 256 octets que nous déclarons ici indiquent seulement que la pile contient 256 octets. Ainsi, au début de l'exécution, SS:SP pointera vers la fin de ces octets. Nous verrons ce que signifie le point d'interrogation dans la troisième partie.

La fin du fichier et le point d'entrée sont signalés par :

end *debut*

Et voilà ! Vous avez découvert l'allure d'un programme en assembleur. Nous pouvons donc à présent nous pencher sur l'étude du langage.

TROISIEME PARTIE

LE LANGAGE ASSEMBLEUR

I. DEFINITION DE DONNEES ET ADRESSAGE

1. Les définitions de données

Les mots “db” (« *define byte* »), “dw” (« *define word* »), ”dd” (« *define double word* ») permettent de déclarer et d’initialiser une variable. Il faut bien comprendre que leur seule action est d’écrire les données dans l’exécutable à l’endroit même où elle se trouvent dans le code source.

Pour accéder à ces données (appelées « *variables* »), il suffit de connaître leur adresse. Pour cela, on peut les faire précéder d’un *label de données*.

Exemple :

TOTO dw 1982

Dans la ligne précédente, TOTO représente l’adresse du nombre 1982 qui est défini juste après. Comme ce nombre est codé sur deux octets (word = 16 bits), c’est le premier octet qui est adressé par TOTO.

Le compilateur se contentera d’écrire le nombre 1982 à l’endroit où se trouve la déclaration.

Remarque : en C, on écrirait ceci :

int TOTO = 1982;

En BASIC ou en PASCAL, cette définition n’a pas d’équivalent, puisque les variables ne peuvent être initialisées lors de leur déclaration. Il est donc obligatoire d’ajouter une ligne de code pour le faire.

Dans la partie précédente, nous avons eu à définir le message qui serait affiché à l’écran. Voici la ligne que nous avons écrite :

message db “*Bonjour, monde !*”, ‘*$*’

Lorsqu’il rencontre une chaîne de caractères, le compilateur écrit *les codes ASCII* de tous les caractères, les uns à la suite des autres.

Voici une autre manière d’écrire cette définition de données. Le résultat (i.e. le programme compilé) est EXACTEMENT LE MÊME :

message db *42h*, *111*, ‘*nj*’

db ‘*our, mo*’

db ‘*nde !$*’

Pour définir plusieurs fois à la suite les mêmes données, on utilise “DUP” (« *duplicate* ») de la manière suivante :

TOTO db *100* dup(“*TOTO EST BEAU*”)

Cela revient à écrire :

TOTO db “*TOTO EST BEAU*”

TOTO db “*TOTO EST BEAU*”

TOTO db “*TOTO EST BEAU*”

;… etc (100 fois) …

Il est fréquent que de nombreuses données n’aient pas besoin d’être initialisées à une valeur précise. Dans ce cas, on les regroupe à la fin du programme et on les remplace par le caractère ‘?’. Les adresses des labels seront calculées de la même façon mais aucune données ne sera écrite dans le fichier (et a fortiori dans la RAM). On dit que l’on met ses variables sur le « *tas* » (« *heap* » en anglais). La seule différence avec une définition classique est qu’au début de l’exécution, nos variables n’auront pas de valeur définie. Leurs valeurs seront « aléatoires » en ce sens qu’on ne peut les connaître au moment de la compilation.

Exemple :

```
;(...)

TOTO db 165 ;ce nombre sera écrit...
VAR1 dw ? ;tous ces octets ne seront
VAR2 dw ?, ? ;pas écrits dans le fichier
VAR3 db 15 dup(?) ;donc pas dans la RAM.

code ends ;fin du segment

end start ;fin du fichier
```

Remarque : Si elles ne sont pas regroupées en fin de programme, le compilateur sera obligé d'écrire les données dans le fichier afin de ne pas fausser les adresses des variables (ou du code) qui suivent. Il remplira alors les points d’interrogation avec des zéros.

Pour indiquer qu’un chiffre est noté en base hexadécimale, on lui ajoute la lettre ‘h’. La lettre ‘b’ signifie que le chiffre est codé en binaire, la lettre 'o' en octal et la lettre ‘d’ en base décimale (base par défaut).

2. L’adressage

a) l’adressage immédiat

On appelle « *adressage immédiat* » l’adressage qui ne fait intervenir que des constantes.

Considérons cette partie de programme qui stocke le nombre 125h dans une variable appelée TOTO, lui ajoute 15 puis charge le résultat dans AX :

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

;(...)

mov word ptr ds:[TOTO], 125h
add word ptr ds:[TOTO], 15
mov ax, ds:[TOTO]

;(...)

ret

TOTO dw 0
;(...)

code ends

end debut
```

La ligne

mov word ptr ds:[*TOTO*], *125h*

charge le nombre 125h dans le mot de la RAM adressé par DS et l'offset de "TOTO". La ligne suivante ajoute 15 au contenu de ce mot.

L'expression "word ptr" devant l'adresse, obligatoire ici, indique la taille de la variable dans laquelle doit être stocké le nombre 125h.

Si on avait mis "dword ptr", ce nombre aurait été codé sur 32 bits (00000125h) au lieu de 16 bits (0125h) : on aurait donc écrasé les deux octets qui suivent la variable. Si on avait mis "byte ptr", la compilation aurait été impossible car un octet ne peut contenir un nombre supérieur à FFh.

Le compilateur n'a en effet aucun moyen de connaître cette taille. La variable "*TOTO*" n'a pas de taille (malgré le mot "dw") : ce n'est en fait qu'un pointeur vers le premier octet du *word* qu'elle représente.

En revanche, la troisième instruction ne fait pas apparaître l'expression "word ptr". Cette fois-ci elle est facultative du fait que la destination (AX = 16 bits) impose la taille.

On peut évidemment remplacer "*TOTO*" par une constante numérique :

mov word ptr ds:[*004Ch*], *125h*

Le mot 125h sera alors écrit à l'adresse DS:4C.

Une question se pose à présent : *que se passe-t-il si le programmeur n'écrit pas le registre de segment dans l'adresse ?*

C'est là qu'intervient la directive "assume". Le compilateur va devoir trouver lui-même quel registre l'utilisateur a voulu sous-entendre.

Si on a utilisé un label, alors le segment sera celui dans lequel est déclaré le label. Mais le compilateur veut un REGISTRE de segment. Il va donc prendre celui qui est censé pointer vers le bon segment et pour le savoir, il examine la directive assume.

Voilà pourquoi cette dernière peut nous épargner d'écrire pour chaque variable l'expression "ds:".

Comprenez bien que cette directive ne sert à rien d'autre qu'à cela et qu'en aucune façon elle ne force les registres de segment à prendre quelque valeur que ce soit.

b) l'adressage indexé et/ou basé

Comme nous l'avons déjà expliqué, il est possible d'utiliser des registres de *base* ou d'*index* pour adresser un octet.

On appelle « *base* » les registres BX et BP et « *index* » les registres SI et DI. La différence est que la base est censée être fixe tandis que l'index varie automatiquement lorsqu'on utilise certaines opérations, telles que MOVSB.

Remarque : BX, BP, DI et SI peuvent tous être utilisés comme des registres généraux.

Voici les adressages possibles:

- ***Constante seule (= adressage immédiat) :***

Exemples :

MOV AH, byte ptr [*TOTO*]

MOV BX, word ptr ds:[*1045h*]

MOV ES:[*10* + *TOTO* x 2], BL

- ***Base seule :***

Exemple :

MOV dword ptr ds:[BP], *15*

- ***Index seul :***

Exemple :

MOV dword ptr es:[DI], *142*

➉ ***Base + Constante :***

Exemples :

MOV byte ptr ds:[BP + *1*], *12*

MOV ds:[BX + (*TOTO – BOBO*)/*10*], AX

➉ ***Index + Constante :***

Exemples :

MOV CX, [*TOTO* + DI]

MOV AL, ds:[*1* + SI]

➉ ***Base + Index :***

Exemple :

MOV ds:[BP + SI], AH

➉ ***Base + Index + Constante :***

Exemple :

MOV AX, word ptr [*TOTO* + BX + DI + *1*]

Remarque : si la constante n'est pas un label, il est parfois impératif de spécifier le registre de segment ! Tout dépend du contexte…

II. SAUTS INCONDITIONNELS, PROCEDURES ET MACROS

1. Les sauts inconditionnels

Ce paragraphe répond à la question : « *Mais comment fait-on un GOTO en assembleur* ? ».

L'instruction équivalente est JMP qui signifie « *jump* ».

La syntaxe est

JMP *MonLabel*

La référence à l'étiquette *MonLabel* sera remplacée lors de la compilation par la *distance* (signée) en octets qui sépare l'instruction qui suit immédiatement le JMP de l'instruction adressée par *MonLabel*. En pratique, le JMP s'utilise exactement comme un GOTO en BASIC.

Cette instruction permet de faire un *saut inconditionnel* : le programme fera ce saut quoiqu'il arrive.

Exemple :

```
;(...)

jmp short COUCOU

;tout ceci ne sera pas exécuté
;(...)
;(...)

COUCOU:

;le saut amène l'exécution ici
;(...)
;(...)

end start
```

Le mot "short" ajouté après "jmp" indique au compilateur que le label "*COUCOU*" se trouve à une distance (signée) qui peut être stockée sur un seul octet. Le compilateur ne le sait pas encore lorsqu'il essaie de compiler l'instruction de saut car le label se trouve en deçà de cette instruction. C'est pourquoi il prévoit deux octets pour écrire le saut, au cas où l'adresse d'arrivée soit éloignée de plus de 128 octets. Lorsqu'il effectue une deuxième « passe », s'il s'aperçoit qu'un seul octet aurait suffi il remplit alors l'octet inutile avec l'instruction NOP (« *No Operation* ») qui ne fait rien du tout. Le programme marchera parfaitement (encore heureux !), mais cela gaspille un octet. Le programmeur a la possibilité d'aider le compilateur en ajoutant le mot "short", ainsi un seul octet est prévu pour la distance de saut. Si le label est placé plus haut que l'instruction de saut, il est inutile de l'écrire.

Remarque : Ce mot peut également être utilisé avec les instructions de saut conditionnel que nous étudierons plus loin.

2. Les procédures

a) appels de procédures

Ceux qui ont déjà programmé en BASIC connaissent les instructions GOSUB et RETURN. Leurs équivalents en assembleur sont CALL et RET.

➓L'instruction **CALL** sert à appeler une procédure :

Syntaxe

CALL *MonLabel*

Action : empile l'offset de l'instruction suivante puis fait un simple saut à l'adresse représentée par *MonLabel*.

➓**RET** (ou RETN) permet de retourner à l'instruction qui suit immédiatement le CALL :

Syntaxe :

RET

Action : dépile l'adresse de retour et la met dans IP. Le programme continue donc à l'adresse qui suit le CALL.

Remarque : Rappelez-vous qu'il est possible de terminer un programme COM avec l'instruction RET. Puisque le DOS empile un zéro de deux octets au chargement du programme, IP prendra la valeur 0000 lorsque le processeur exécutera cette instruction : il pointera donc vers le début du PSP. Or le PSP commence toujours par les deux octets suivants : CDh 20h (ce qui s'écrit INT 20h en assembleur). L'interruption 20h, qui permet de terminer un programme COM, sera donc appelée et le contrôle sera rendu au DOS.

Voici un exemple d'utilisation des procédures aussi simple que possible : ce programme COM appelle 12 fois une procédure qui écrit un message à l'écran et rend la main au DOS. Il n'est d'aucune utilité et n'est pas optimisé du tout.

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

mov cx, 12 ;nombre d'itérations

boucle :

call ecrit_message ;appel de procédure
loop boucle ;décrémenter CX et sauter à «  boucle » si CX<>0

ret ;terminer le programme en exécutant l'int. 20h

ecrit_message :

mov ah, 09h
mov dx, offset message
int 21h

ret

message db "Bonjour, monde !", 10, 13, '$'

code ends

end debut
```

Remarque : Les codes ASCII 10 et 13 représentent respectivement la fin de ligne et le retour chariot. Grâce à eux, on revient à la ligne chaque fois qu'on a écrit le message.

Ce programme fonctionne parfaitement. Toutefois, les conventions d'écriture veulent que les procédures soient écrites comme suit :

```
ecrit_message proc near

mov ah, 09h
mov dx, offset message
int 21h

ret

ecrit_message endp
```

Le programme compilé est toujours le même, mais le code est plus lisible puisqu'on distingue

les procédures des simples labels. Le mot “near” signifie que l’adresse de cette procédure est réduite à un *offset*. Il n’y a pas d’adresse de segment. La procédure ne pourra donc être appelée que de l’intérieur même du segment.

Le mot qui a le sens contraire de “near” est “far”. Les « *call far* » sont assez peu utilisés.

b) le passage des paramètres

Il existe deux moyens de passer des paramètres à une procédure : les registres et la pile.

➉ ***Le passage par registres*** :

Il consiste à transmettre les paramètres dans des registres convenus à l’avance, par exemple AX et BX. Les deux gros avantages de cette méthode sont sa simplicité et la vitesse d’exécution. L’inconvénient majeur est le nombre limité de registres, d’autant plus qu’au moment de l’appel, certains d’entre eux ne seront peut-être pas disponibles.

➉ ***Le passage par la pile*** :

C’est la méthode qu’utilisent les langages de haut niveau tels que le C ou le PASCAL. Avant l’appel, on empile les paramètres un à un. La procédure appelée se chargera de les lire.

Pour cela on utilise le registre BP de la manière suivante : on transfère la valeur de SP dans BP à l’aide de l’instruction “MOV BP, SP”. L’adresse de retour (qui sera dépilée quand le processeur rencontrera l'instruction “RET”) se trouve à l’adresse SS:[BP], le dernier paramètre est adressé par SS:[BP + 2], l’avant-dernier par SS:[BP + 4], etc… On peut ainsi lire n’importe quel paramètre sans le dépiler.

Attention cependant ! A la fin de la procédure, il faudra tout de même rendre à SP la valeur qu’il avait avant l’empilage des paramètres. Pour cela, on fait suivre l’instruction RET du nombre de paramètres, *multiplié par 2* (car chaque paramètre comporte deux octets).

Exemple :

```
;(...)

push AX ;empiler le paramètre 1
push BX ;puis le paramètre 2
push CX ;puis le paramètre 3
;NB : on peut écrire ces 3 instructions d'un seul coup : "PUSH AX BX CX"
call foto

;(...)
;(...)

foto proc near
mov bp, sp
;lecture des paramètres...
;(...)
ret 6
foto endp

;(...)
```

Soyez toujours très vigilant avec les appels de procédures : pensez que l'adresse de retour sera dépilée lorsque le programme rencontrera l'instruction RET. Il serait donc suicidaire de laisser une donnée empilée avant d'appeler cette instruction.

3. Les macros

Etant donné que certaines instructions se répètent constamment dans un programme, l'écriture de macrofonctions (ou *macros*) est un moyen pratique de rendre votre code source plus lisible.

Il est possible de choisir pour certaines suites d'instructions un nom qui les représente. Lorsque le compilateur (en fait, le préprocesseur) rencontrera ce nom dans votre code source, il le remplacera par les lignes de code qu'il désigne. Ces lignes forment une « *macro* ».

Les macros, à la différence des procédures, n'ont aucune signification pour la machine. Seul le compilateur comprend leur signification. Elles ne sont qu'un artifice mis à la disposition du programmeur pour clarifier son programme. Lorsque le compilateur rencontre le nom d'une macro dans votre code, il le remplace par le code de la macro. *Tout se passe exactement comme si vous aviez tapé vous-même ce code à la place du nom de la macro.*

Ainsi, si vous appelez quinze fois une macro dans votre programme, le compilateur écrira quinze fois le code de cette macro. C'est toute la différence avec les procédures qui ne sont écrites qu'une seule fois mais peuvent être appelées aussi souvent qu'on veut à l'aide d'un CALL.

Voici comment écrire une macro : l'exemple suivant sert à écrire un message à l'écran.

```
ecrit_texte   macro text?

local text, fin ;labels locaux

push ax dx ;sauvegarde de AX et DX

mov ah, 09h
mov dx, offset text
int 21h

pop dx ax ;restaurer DX et AX

jmp short fin ;pour ne pas exécuter les données qui suivent !

text db text?, '$'

fin:

endm
```

Le code précédent peut être écrit n'importe où dans votre programme, à condition qu'il se trouve avant tout appel de cette macro. Afin d'éviter les ennuis, il est fortement conseillé de réunir vos macros au début du code, avant toute autre ligne. De toute façon, il ne sera pas compilé à l'endroit où vous l'avez écrit mais aux endroits où se trouvent les appels de macros.

Le mot "*text?*" est un « *paramètre* ». Le point d'interrogation n'est pas requis ; nous l'avons mis pour indiquer qu'il s'agit d'un paramètre et non d'une variable. Une macro peut utiliser plusieurs paramètres séparés par des virgules.

Pour appeler cette macro, il vous suffit d'écrire la ligne

ecrit_texte "Coucou ! Ceci est un essai !"

Le compilateur se chargera alors de la remplacer par les instructions comprises entre la première et la dernière ligne de cet exemple, en prenant le soin de remplacer le mot "*text?*" par le message fourni en paramètre.

En assembleur, chaque label doit avoir un nom unique. Or, si une macro est appelée plusieurs fois, les mêmes noms seront utilisés. Il faut donc la plupart du temps inclure la directive LOCAL qui forcera le compilateur à changer le nom des labels à chaque appel de la macro. Attention : cette directive doit suivre immédiatement la déclaration de la macro.

Supposons à présent que l'on veuille écrire à l'écran le message « Je suis bien content » et revenir à la ligne à l'aide de notre macro *ecrit_texte*.

La syntaxe suivante :

ecrit_texte "Coucou ! Ceci est un essai !", 10, 13

est incorrecte, car le compilateur croirait que l'on a écrit trois paramètres ! Il faut alors entourer notre unique paramètre par les signes '<' et '>' :

ecrit_texte <"Coucou ! Ceci est un essai !", 10, 13>

4. La directive EQU

La directive EQU a un rôle voisin de celui des macros. Elle permet de remplacer un simple mot par d'autres plus complexes. Son intérêt est qu'elle peut être invoquée en plein milieu d'une ligne.

Quelques exemples :

Longueur EQU (*fin – debut*)

Message EQU "*Bonjour messieurs ! Comment allez-vous ?*", '*$*'

Version EQU 2

Quitter EQU ret

Quitter2 EQU int 20h

Mettre_dans_AH EQU mov ah,

Interruption_21h EQU int 21h

Un programme qui contient de telles directives peut se terminer ainsi :

Mettre_dans_AH 4Ch

Interruption_21h

Ou ainsi (si c'est un COM) :

Quitter2

5. L'inclusion de fichiers

Il est possible d'accéder à des procédures, des macros ou des définitions EQU qui se trouvent dans d'autres fichiers. Cela permet de se constituer des librairies de macros ou de procédures que l'on peut réutiliser d'un programme à l'autre.

Pour inclure le fichier TOTO.LIB, écrivez au début de votre code source :

```
If1
include TOTO.LIB
endif
```

La condition "if1" indique au compilateur que l'inclusion ne doit s'effectuer que lors de la première passe.
Le fichier TOTO.LIB est un fichier texte tout à fait banal qui contient des lignes de code en assembleur.

III. LES PRINCIPALES INSTRUCTIONS DU LANGAGE MACHINE

Remarques préliminaires :

- *Le principe du langage assembleur est de remplacer chaque opcode hexadécimal par un mot facile à retenir. Ce mot est appelé mnémonique. Par exemple, "INT" est le mnémonique associé à l'opcode CDh. Chaque fois que le compilateur rencontrera ce mot, il le remplacera par l'octet CDh et écrira ensuite l'opérande (ici : le numéro de l'interruption) en hexadécimal.*

- *Cette liste récapitule les instructions que nous connaissons déjà et en présente de nouvelles. Elle n'est pas exhaustive mais vous sera amplement suffisante pour la plupart de vos programmes.*

- *Certaines instructions, comme PUSHA, ne sont disponibles que pour des modèles de processeurs plus évolués que le 8086, par exemple le 286. N'oubliez pas la directive .386 si vous les utilisez.*

1. L'instruction NOP (« *No Operation* »)

Syntaxe : NOP

Description : Ne fait rien ! Mais alors RIEN ! Que dalle ! Niet !

2. L'instruction MOV (« *Move* »)

Syntaxe : MOV *Destination*, *Source*

Description : Copie le contenu de *Source* dans *Destination*.

Mouvements autorisés : MOV Registre général, Registre quelconque

MOV *Mémoire*, Registre quelconque

MOV Registre général, *Mémoire*

MOV Registre général, *Constante*

MOV *Mémoire*, *Constante*

MOV Registre de segment, Registre général

Remarques : *Source* et *Destination* doivent avoir la même taille. On ne peut charger dans un registre de segment que le contenu d'un registre général (SI, DI et BP sont considérés ici comme des registres généraux).

Exemples : MOV AX, *5*

MOV ES, DX

MOV AL, [*Variable1*] *;Copie un octet car AL contient 8 bits*

MOV [*Variable2*], DS *;Copie un word car DS contient 16 bits*

MOV word ptr [*Variable3*], *12* *;Ici, on spécifie que la variable est un word*

3. L'instruction XCHG (« *Exchange* »)

Syntaxe : XCHG *Destination*, *Source*

Description : Echange les contenus de *Source* et de *Destination*.

Mouvements autorisés : XCHG Registre général, Registre général

XCHG Registre général, *Mémoire*

XCHG *Mémoire*, Registre général

4. L'instruction JMP (« *Jump* »)

Syntaxe : JMP *MonLabel*

Description : Saute à l'instruction pointée par *MonLabel*.

5. L'opérateur CMP (« *Compare* »)

Syntaxe : CMP *Destination*, *Source*

Description : Cet opérateur sert à comparer deux nombres : *Source* et *Destination*. *C'est le registre des indicateurs qui contient les résultats de la comparaison.* Ni *Source* ni *Destination* ne sont modifiés.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Remarque : Cet opérateur effectue en fait une soustraction mais contrairement à SUB, le résultat n'est pas sauvegardé.

Le programme doit pouvoir réagir en fonction des résultats de la comparaison. Pour cela, on utilise les *sauts conditionnels* (voir ci-dessous).

6. Les instructions de saut conditionnel

Les sauts conditionnels sont terriblement importants car ils permettent au programme de faire des choix en fonction des données.

Un saut conditionnel n'est effectué qu'à certaines conditions portant sur les flags (par exemple : CF = 1 ou ZF = 0).

Certains mnémoniques de sauts conditionnels sont totalement équivalents, c'est-à-dire qu'ils représentent le même opcode hexadécimal. C'est pour aider le programmeur qu'ils existent parfois sous plusieurs formes.

a) les sauts de comparaison

- **JE** (« *Jump if Equal* ») fait un saut au label spécifié si et seulement si ZF = 1. Rappelez-vous que ce flag est à 1 si et seulement si le résultat de l'opération précédente vaut zéro. Comme CMP réalise une soustraction, on utilise généralement JE pour savoir si deux nombres sont égaux.

Exemple :

```
;(...)

cmp ah, bh ;comparaison de AH et BH (soustraction)
je egal ;saut ssi AH = BH (le processeur regarde ZF pour le savoir)

different :

;(...)

egal :

;(...)
```

Mnémonique équivalent : **JZ** (« *Jump if Zero* »)

- **JG** (« *Jump if Greater* ») fait un saut au label spécifié si et seulement si ZF = 0 et SF = OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est supérieur à un autre.

Exemple :

```
;(...)

cmp ah, bh ;comparaison
jg AH_superieur_BH ;saut ssi AH > BH

AH_inferieur_ou_egal_BH :

;(...)

AH_superieur_BH :

;(...)
```

Mnémonique équivalent : **JNLE** (« *Jump if Not Less Or Equal* »)

- **JGE** (« *Jump if Greater or Equal* ») fait un saut au label spécifié si et seulement si SF = OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est supérieur ou égal à un autre.

Mnémonique équivalent : **JNL** (« *Jump if Not Less* »)

- **JL** (« *Jump if Less* ») fait un saut au label spécifié si et seulement si SF <> OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est inférieur à un autre.

Mnémonique équivalent : **JNGE** (« *Jump if Not Greater Or Equal* »)

- **JLE** (« *Jump if Less Or Equal* ») fait un saut au label spécifié si et seulement si SF <> OF ou ZF = 1. On l'utilise en arithmétique *signée* pour savoir si un nombre est inférieur ou égal à un autre.

Mnémonique équivalent : **JNG** (« *Jump if Not Greater* »)

- **JA** (« *Jump if Above* ») fait un saut au label spécifié si et seulement si ZF = 0 et CF = 0. On l'utilise en arithmétique *non signée* pour savoir si un nombre est supérieur à un autre.

Mnémonique équivalent : **JNBE** (« *Jump if Not Below Or Equal* »)

- **JAE** (« *Jump if Above or Equal* ») fait un saut au label spécifié si et seulement si CF = 0. On l'utilise en arithmétique *non signée* pour savoir si un nombre est supérieur ou égal à un autre.

Mnémonique équivalent : **JNB** (« *Jump if Not Below* »)

- **JB** (« *Jump if Below* ») fait un saut au label spécifié si et seulement si CF = 1. On l'utilise en arithmétique *non signée* pour savoir si un nombre est inférieur à un autre.

Mnémonique équivalent : **JNAE** (« *Jump if Not Above Or Equal* »)

- **JBE** (« *Jump if Below or Equal* ») fait un saut au label spécifié si et seulement si CF = 1 ou ZF = 1. On l'utilise en arithmétique *non signée* pour savoir si un nombre est inférieur ou égal à un autre.

Mnémonique équivalent : **JNA** (« *Jump if Not Above* »)

b) les sauts de test sur les flags

Ces instructions testent un flag unique et exécutent ou non le saut selon la valeur de ce flag.

- **JC** (« *Jump if Carry* ») fait un saut au label spécifié si et seulement si CF = 1.

Remarques : Ce mnémonique correspond au même opcode que JB. Il est souvent employé pour vérifier que l'appel d'une interruption n'a pas déclenché d'erreur.

Exemple :

```
;(...)

mov ah, 3eh
int 21h

jc short ErreurSurvenue

; sinon : pas d'erreur...
;(...)
;(...)

ErreurSurvenue :

;(...)
```

Si l'appel de la fonction 3Eh de l'interruption 21H se solde par une erreur, alors la CF vaudra 1 et le saut sera accompli. Dans le cas opposé, l'exécution continuera normalement de manière linéaire.

- **JNC** (« *Jump if not Carry* ») fait un saut au label spécifié si et seulement si CF = 0.
- **JZ** (« *Jump if Zero* ») fait un saut au label spécifié si et seulement si ZF = 1. Ce mnémonique correspond au même opcode que JE.
- **JNZ** (« *Jump if not Zero* ») fait un saut au label spécifié si et seulement si ZF = 0. Ce mnémonique correspond au même opcode que JNE.
- **JS** (« *Jump if Sign* ») fait un saut au label spécifié si et seulement si SF = 1.
- **JNS** (« *Jump if not Sign* ») fait un saut au label spécifié si et seulement si SF = 0.
- **JO** (« *Jump if Overflow* ») fait un saut au label spécifié si et seulement si OF = 1.
- **JNO** (« *Jump if not Overflow* ») fait un saut au label spécifié si et seulement si OF = 0.
- **JP** (« *Jump if Parity* ») fait un saut au label spécifié si et seulement si PF = 1.
- **JNP** (« *Jump if not Parity* ») fait un saut au label spécifié si et seulement si PF = 0.

c) le saut de test sur le registre CX

JCXZ (« *Jump if CX = Zero* ») fait un saut au label spécifié si et seulement si CX = 0.

7. Les instructions arithmétiques

a) l'instruction INC (« *Increment* »)

Syntaxe : INC *Destination*

Description : Incrémente *Destination*.

Indicateurs affectés : AF, OF, PF, SF, ZF

Exemple : INC CL

b) l'instruction ADD (« *Addition* »)

Syntaxe : ADD *Destination*, *Source*

Description : Ajoute *Source* à *Destination*

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Exemple : ADD byte ptr [*VARIABLE* + DI], *5*

c) l'instruction ADC (« *Add with Carry* »)

Syntaxe : ADC *Destination*, *Source*

Description : Ajoute (*Source* + CF) à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

d) l'instruction DEC (« *Decrement* »)

Syntaxe : DEC *Destination*

Description : Décrémente *Destination*.

Indicateurs affectés : AF, OF, PF, SF, ZF

e) l'instruction SUB (« *Substract* »)

Syntaxe : SUB *Destination*, *Source*

Description : Soustrait *Source* à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

f) l'instruction SBB (« *Substract with Borrow* »)

Syntaxe : SBB *Destination*, *Source*

Description : Soustrait (*Source* + CF) à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

g) l'instruction MUL (« *Multiply* »)

Syntaxe : MUL *Source*

Description : Effectue une multiplication d'entiers *non signés*.

- Si *Source* est un octet : AL est multiplié par *Source* et le résultat est placé dans AX.
- Si *Source* est un mot : AX est multiplié par *Source* et le résultat est placé dans DX:AX.
- Si *Source* est un double mot : EAX est multiplié par *Source* et le résultat est placé dans EDX:EAX.

Indicateurs affectés : CF, OF

Remarque : *Source ne peut être une valeur immédiate.*

Exemples : MUL CX

MUL byte ptr [*TOTO*]

h) l'instruction IMUL (« *Integer Multiply* »)

Syntaxe : IMUL *Source*

IMUL *Destination*, *Source*

IMUL *Destination*, *Source*, *Valeur*

Description : Effectue une multiplication d'entiers *signés*.

IMUL *Source* :

- Si *Source* est un octet : AL est multiplié par *Source* et le résultat est placé dans AX.
- Si *Source* est un mot : AX est multiplié par *Source* et le résultat est placé dans DX:AX.
- Si *Source* est un double mot : EAX est multiplié par *Source* et le résultat est placé dans EDX:EAX.

IMUL *Destination, Source* : Multiplie *Destination* par *Source* et place le résultat dans *Destination*.

IMUL *Destination, Source, Valeur* : Multiplie *Source* par *Valeur* et place le résultat dans *Destination*.

Indicateurs affectés : CF, OF

i) l'instruction DIV (« *Divide* »)

Syntaxe : DIV *Source*

Description : Effectue une division euclidienne d'entiers *non signés*.

- Si *Source* est un octet : AX est divisé par *Source*, le quotient est placé dans AL et le reste dans AH.
- Si *Source* est un mot : DX:AX est divisé par *Source*, le quotient est placé dans AX et le reste dans DX.

- Si *Source* est un double mot : EDX:EAX est divisé par *Source*, le quotient est placé dans EAX et le reste dans EDX.

Indicateurs affectés : AF, OF, PF, SF, ZF

Remarque : *Source ne peut être une valeur immédiate.*

j) l'instruction IDIV (« *Integer Divide* »)

Syntaxe : IDIV *Source*

Description : Effectue une division euclidienne d'entiers *signés*.

- Si *Source* est un octet : AX est divisé par *Source*, le quotient est placé dans AL et le reste dans AH.
- Si *Source* est un mot : DX:AX est divisé par *Source*, le quotient est placé dans AX et le reste dans DX.
- Si *Source* est un double mot : EDX:EAX est divisé par *Source*, le quotient est placé dans EAX et le reste dans EDX.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

k) l'instruction NEG (« *Negation* »)

Syntaxe : NEG *Destination*

Description : Forme le complément à 2 de *Destination*, i.e. prend l'opposé de *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Remarque : A ne pas confondre avec NOT, qui forme le complément à 1.

8. Les instructions logiques

a) l'instruction NOT (« *Logical NOT* »)

Syntaxe : NOT *Destination*

Description : Effectue un NON logique bit à bit sur *Destination* (i.e. chaque bit de *Destination* est inversé).

b) l'instruction OR (« *Logical OR* »)

Syntaxe : OR *Destination*, *Source*

Description : Effectue un OU logique inclusif bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Remarque : Afin d'optimiser la taille et les performances du programme, on peut utiliser l'instruction "OR AX, AX" à la place de "CMP AX, 0". En effet, un OU bit à bit entre deux nombres identiques ne modifie pas *Destination* et est exécuté « infiniment » plus rapidement qu'une soustraction. Comme les flags sont affectés, les sauts conditionnels sont possibles.

c) l'instruction AND (« *Logical AND* »)

Syntaxe : AND *Destination*, *Source*

Description : Effectue un ET logique bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

d) l'instruction TEST (« *Test for bit pattern* »)

Syntaxe : TEST *Destination*, *Source*

Description : Effectue un ET logique bit à bit entre *Destination* et *Source*. Le résultat n'est pas conservé, donc *Destination* n'est pas modifié. Seuls les flags sont affectés.

Cet opérateur est souvent utilisé pour tester certains bits de *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Exemple :

```
;(...)

test ax, 00000010b
jnz bit2_vaut_1 ;saut ssi le 2e bit = 1

; sinon : le 2e bit de AX vaut 0...
;(...)

bit2_vaut_1:

;(...)
```

e) l'instruction XOR (« *Exclusive logical OR* »)

Syntaxe : XOR *Destination*, *Source*

Description : Effectue un OU logique exclusif bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Remarque : Pour remettre un registre à zéro, il est préférable de faire "XOR AX, AX" que "MOV AX, 0". En effet, le résultat est le même mais la taille et surtout la vitesse d'exécution de

l'instruction sont très largement optimisées.

f) l'instruction SHL (« *Shift logical Left* »)

Syntaxe : SHL *Destination*, *Source*

Description : Décale les bits de *Destination* de *Source* positions vers la gauche. Les bits les plus à droite sont remplacés par des zéros.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Exemple : SHL AX, *4 ;permet de multiplier par 16 de façon infiniment plus rapide que MUL*

Mnémonique équivalent : SAL (« *Shift Arithmetical Left* »)

g) l'instruction SHR (« *Shift logical Right* »)

Syntaxe : SHR *Destination*, *Source*

Description : Décale les bits de *Destination* de *Source* positions vers la droite. Les bits les plus à gauche sont remplacés par des zéros.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Mnémonique équivalent : SAR (« *Shift Arithmetical Right* »)

h) l'instruction ROL (« *Rotate Left* »)

Syntaxe : ROL *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la gauche. Le dernier bit à être sorti à gauche et à être rentré à droite est placé dans CF. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

i) l'instruction ROR (« *Rotate Right* »)

Syntaxe : ROR *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la droite. Le dernier bit à être sorti à droite et à être rentré à gauche est placé dans CF. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

j) l'instruction RCL (« *Rotate through Carry Left* »)

Syntaxe : RCL *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la gauche. CF est utilisé comme intermédiaire : chaque bit qui sort à gauche est placé dans CF, et le

contenu de CF est ensuite réinséré à droite. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

k) l'instruction RCR (« *Rotate through Carry Right* »)

Syntaxe : RCR *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la droite. CF est utilisé comme intermédiaire : chaque bit qui sort à droite est placé dans CF, et le contenu de CF est ensuite réinséré à gauche. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

9. Les instructions de manipulation des flags

a) l'instruction CLC (« *Clear Carry flag* »)

Syntaxe : CLC

Description : Met CF à 0.

Indicateurs affectés : CF

b) l'instruction STC (« *Set Carry flag* »)

Syntaxe : STC

Description : Met CF à 1.

Indicateurs affectés : CF

c) l'instruction CLD (« *Clear Direction flag* »)

Syntaxe : CLD

Description : Met DF à 0.

Indicateurs affectés : DF

d) l'instruction STD (« *Set Direction flag* »)

Syntaxe : STD

Description : Met DF à 1.

Indicateurs affectés : DF

e) l'instruction CLI (« *Clear Interrupt flag* »)

Syntaxe : CLI

Description : Met IF à 0.

Indicateurs affectés : IF

f) l'instruction STI (« *Set Interrupt flag* »)

Syntaxe : STI

Description : Met IF à 1.

Indicateurs affectés : IF

g) l'instruction CMC (« *Complement Carry flag* »)

Syntaxe : CMC

Description : Inverse CF.

Indicateurs affectés : CF

h) l'instruction LAHF (« *Load AH from Flags* »)

Syntaxe : LAHF

Description : Charge dans AH l'octet de poids faible du registre des indicateurs.

i) l'instruction SAHF (« *Store AH into Flags* »)

Syntaxe : SAHF

Description : Stocke les bits de AH dans le registre des indicateurs.

10. Les instructions de gestion de la pile

a) l'instruction PUSH (« *Push Word onto Stack* »)

Syntaxe : PUSH *Source*

Description : Empile le mot *Source*. SP est décrémenté de 2.

Remarques : *Source* ne peut être une valeur immédiate. Il est possible d'abréger votre code source en écrivant par exemple "PUSH AX BX BP". Le compilateur écrira alors trois fois l'instruction PUSH du langage machine. Il est possible également d'empiler des doubles mots.

b) l'instruction POP (« *Pop Word off Stack* »)

Syntaxe : POP *Destination*

Description : Dépile le mot qui se trouve au sommet de la pile et le place dans *Destination*. SP

est incrémenté de 2.

c) l'instruction PUSHF (« *Push Flags onto Stack* »)

Syntaxe : PUSHF

Description : Empile le registre des indicateurs. SP est décrémenté de 2.

Remarque : PUSHFD empile le registre des indicateurs codé sur 32 bits.

d) l'instruction POPF (« *Pop Flags off Stack* »)

Syntaxe : POPF

Description : Dépile le mot qui se trouve au sommet de la pile et le place dans le registre des indicateurs. SP est incrémenté de 2.

Indicateurs affectés : Tous

Remarque : POPFD est utilisé pour un registre des indicateurs codé sur 32 bits.

e) l'instruction PUSHA (« *Push All registers onto Stack* »)

Syntaxe : PUSHA

Description : Empile AX, BX, CX, DX, BP, SI, DI et SP.

Remarque : PUSHAD est utilisé pour des registres de 32 bits.

f) l'instruction POPA (« *Pop All registers off Stack* »)

Syntaxe : POPA

Description : Restaure AX, BX, CX, DX, BP, SI, DI et SP à partir de la pile.

Remarque : POPAD est utilisé pour des registres de 32 bits.

11. Les instructions de gestion des chaînes d'octets

a) l'instruction MOVSB (« *Move String Byte* »)

Syntaxe : MOVSB

Description : Copie l'octet adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés, sinon ils sont décrémentés.

Remarque : Pour copier plusieurs octets, faire REP MOVSB (« *Repeat Move String Byte* »). Le nombre d'octets à copier doit être transmis dans CX de même que pour un LOOP.

Exemple :

```
;(...)

mov ax, ds ;mettre ds
mov es, ax ; dans es

mov si, offset depart ;source

mov di, offset destination ;destination

mov cx, fin – depart ;nb d'octets

cld ;vers la droite

rep movsb ;copier !

;(...)
```

b) l'instruction SCASB (« *Scan String Byte* »)

Syntaxe : SCASB

Description : Compare l'octet adressé par ES:DI avec AL. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté, sinon il est décrémenté.

Remarques : Pour comparer plusieurs octets, faire "REP SCASB" ou "REPE SCASB" (« *Repeat until Egal* »), ou encore "REPZ SCASB" (« *Repeat until Zero* »). Ces trois préfixes sont équivalents.
Le nombre d'octets à comparer doit être transmis dans CX. La boucle ainsi créée s'arrête si CX = 0 ou si le caractère pointé par ES:DI est le même que celui contenu dans AL (i.e. si ZF = 1). On peut ainsi rechercher un caractère dans une chaîne.
Pour répéter au contraire la comparaison *jusqu'à ce que* ZF = 0, c'est-à-dire jusqu'à ce que AL et le caractère adressé par ES:DI *diffèrent*, utiliser REPNE ou REPNZ.

Exemple :

```
;(...)

mov ax, ds ;mettre ds
mov es, ax ; dans es

mov di, offset chaine ;destination

mov cx, fin – chaine ;longueur

cld ;vers la droite

mov al, 0

rep scasb ;chercher 0

;ES:DI pointe maintenant vers l'octet qui suit le premier 0 trouvé (si un 0 a
été trouvé...)

;(...)
```

c) l'instruction LODSB (« *Load String Byte* »)

Syntaxe : LODSB

Description : Charge dans AL l'octet adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté, sinon il est décrémenté.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour MOVSB.

d) l'instruction STOSB (« *Store String Byte* »)

Syntaxe : STOSB

Description : Stocke le contenu de AL dans l'octet adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté, sinon il est décrémenté.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

e) l'instruction CMPSB (« *Compare String Byte* »)

Syntaxe : CMPSB

Description : Compare l'octet adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés, sinon ils sont décrémentés.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

12. Les instructions de gestion des chaînes de mots

Ce sont les mêmes que les précédentes, hormis qu'elles traitent des mots et non des octets et que leur nom se termine par un 'W' au lieu du 'B'.

a) l'instruction MOVSW (« *Move String Word* »)

Syntaxe : MOVSW

Description : Copie le mot adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés de 2, sinon ils sont décrémentés de 2.

Remarque : Pour copier plusieurs mots, faire "REP MOVSW". Le nombre de mots à copier doit être transmis dans CX.

b) l'instruction SCASW (« *Scan String Word* »)

Syntaxe : SCASW

Description : Compare le mot adressé par ES:DI avec AX. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

c) l'instruction LODSW (« *Load String Word* »)

Syntaxe : LODSW

Description : Charge dans AX le mot adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

d) l'instruction STOSW (« *Store String Word* »)

Syntaxe : STOSW

Description : Stocke le contenu de AX dans le mot adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour STOSB.

e) l'instruction CMPSW (« *Compare String Word* »)

Syntaxe : CMPSW

Description : Compare le mot adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés de 2, sinon ils sont décrémentés de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour CMPSB.

13. Les instructions de gestion des chaînes de doubles mots

Elles traitent des doubles mots et leur nom se termine par un 'D'.

a) l'instruction MOVSD (« *Move String Double Word* »)

Syntaxe : MOVSD

Description : Copie le double mot adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés de 4, sinon ils sont décrémentés de 4.

Remarque : Pour copier plusieurs doubles mots, faire "REP MOVSD". Le nombre de doubles mots à copier doit être transmis dans CX.

b) l'instruction SCASD (« *Scan String Double Word* »)

Syntaxe : SCASD

Description : Compare le double mot adressé par ES:DI avec EAX. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

c) l'instruction LODSD (« *Load String Double Word* »)

Syntaxe : LODSD

Description : Charge dans EAX le double mot adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

d) l'instruction STOSD (« *Store String Double Word* »)

Syntaxe : STOSD

Description : Stocke le contenu de EAX dans le double mot adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour STOSB.

e) l'instruction CMPSD (« *Compare String Double Word* »)

Syntaxe : CMPSD

Description : Compare le double mot adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés de 4, sinon ils sont décrémentés de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour CMPSB.

14. Les instructions d'appel

a) l'instruction CALL (« *Procedure Call* »)

Syntaxe : CALL *MaProc*

Description : Appel de procédure. Si *MaProc* se trouve dans un segment extérieur, le processeur empile CS. Ensuite, dans tous les cas, il empile IP et fait un saut à l'étiquette *MaProc*.

b) l'instruction RET (ou RETN)

Syntaxe : RET

Description : Retour de procédure se trouvant à l'intérieur du segment (NEAR). Un mot est dépilé et placé dans IP. Le contrôle retourne donc à la procédure appelante.

c) l'instruction RETF

Syntaxe : RETF

Description : Retour de procédure se trouvant à l'extérieur du segment (FAR). Deux mots sont dépilés et placés dans CS:IP. Le contrôle retourne donc à la procédure appelante.

15. Les instructions de boucle

a) l'instruction LOOP

Syntaxe : LOOP *MonLabel*

Description : Décrémente CX, puis, si CX <> 0, fait un saut à *MonLabel*.

b) l'instruction LOOPE (« *Loop while Equal* »)

Syntaxe : LOOPE *MonLabel*

Description : Décrémente CX, puis, si CX <> 0 et ZF = 1, fait un saut à *MonLabel*.

Mnémonique équivalent : LOOPZ

c) l'instruction LOOPNE (« *Loop while not Equal* »)

Syntaxe : LOOPNE *MonLabel*

Description : Décrémente CX, puis, si CX <> 0 et ZF = 0, fait un saut à *MonLabel*.

Mnémonique équivalent : LOOPNZ

16. Les instructions d'adressage

a) l'instruction LEA (« *Load effective address* »)

Syntaxe : LEA *Destination*, *Source*

Description : Charge l'offset de la source dans le registre *Destination*.

Exemple : LEA BP, word ptr [BP + *TOTO*]

b) l'instruction LDS (« *Load pointer using DS* »)

Syntaxe : LDS *Destination*, *Source*

Description : Transfère dans DS:*Destination* le contenu de la mémoire adressée par *Source*.

c) l'instruction LES (« *Load pointer using ES* »)

Syntaxe : LES *Destination*, *Source*

Description : Transfère dans ES:*Destination* le contenu de la mémoire adressée par *Source*.

17. Les instructions de conversion arithmétique

Remarque préliminaire : Nous n'expliciterons pas toutes ces instructions.

a) l'instruction AAA (« *ASCII Adjust for Addition* »)

b) l'instruction AAD (« *ASCII Adjust for Division* »)

c) l'instruction AAM (« *ASCII Adjust for Multiplication* »)

d) l'instruction AAS (« *ASCII Adjust for Substraction* »)

e) l'instruction CBW (« *Convert Byte to Word* »)

Syntaxe : CBW

Description : Convertit l'octet *signé* stocké dans AL en un mot (signé) stocké dans AX. Ainsi, si AL est négatif, AH sera rempli de 1 binaires, sinon, AH sera mis à 0.

f) l'instruction CWD (« *Convert Word to Double Word* »)

Syntaxe : CWD

Description : Convertit le mot *signé* stocké dans AX en un double mot (signé) stocké dans DX:AX. Ainsi, si AX est négatif, DX sera rempli de 1 binaires, sinon DX sera mis à 0.

g) l'instruction DAA (« *Decimal Adjust for Addition* »)

h) l'instruction DAS (« *Decimal Adjust for Substraction* »)

i) l'instruction MOVSX (« *Move with Sign Extend* »)

Syntaxe : MOVSX *Destination*, *Source*

Description : Déplace le contenu *signé* d'un registre de 8 bits dans un registre de 16 bits, ou bien déplace le contenu signé d'un registre de 16 bits dans un registre de 32 bits. Si *Source* est

négatif, la partie haute de *Destination* sera remplie de 1 binaires, sinon elle sera remplie de 0.

j) l'instruction MOVZX (« *Move with Zero Extend* »)

Syntaxe : MOVZX *Destination*, *Source*

Description : Déplace le contenu *non signé* d'un registre de 8 bits dans un registre de 16 bits, ou bien déplace le contenu signé d'un registre de 16 bits dans un registre de 32 bits. La partie haute de *Destination* sera donc mise à 0.

k) l'instruction XLAT (« *Translate* »)

18. Les instructions d'entrée-sortie

a) l'instruction IN (« *Input from port* »)

Syntaxe : IN *Destination*, *Port*

Description : Charge un octet ou un mot depuis un port d'entrée-sortie dans AL ou AX. *Port* peut être DX ou bien une constante de 8 bits.

b) l'instruction OUT (« *Output to port* »)

Syntaxe : OUT *Port*, *Source*

Description : Ecrit dans Port la valeur contenue dans Source. Source ne peut être que AL ou AX. Port est une constante ou bien DX.

QUATRIEME PARTIE

LES INTERRUPTIONS DU DOS RELATIVES AUX FICHIERS

Remarque préliminaire : Cette partie présente quelques unes des interruptions du DOS qui servent à manier les fichiers d'un disque. Nous la concluerons par l'étude d'un exemple de programme qui crypte un fichier choisi par l'utilisateur.

I. LECTURE ET ECRITURE DE FICHIERS AVEC LES HANDLES

Sous DOS, il existe deux méthodes pour accéder aux fichiers et pour chacune d'elles un lot d'interruptions spécifiques. La première est la méthode des FCB (« *File contol block* »). Elle est rarement utilisée, aussi ne l'aborderons nous pas. Nous étudierons le principe de la méthode des « *handles* ».

Pour lire ou écrire des données dans un fichier, il est nécessaire de l'*ouvrir*, c'est-à-dire de le charger en mémoire. Quand toutes les opérations de lecture et d'écriture auront été effectuées, le fichier devra être *refermé* afin d'enregistrer les éventuelles dernières modifications et surtout de libérer la mémoire occupée.

1. Ouverture d'un fichier

On ouvre un fichier en appelant la fonction 3dh de l'interruption 21h. Celle-ci attend comme paramètre dans DS:DX *l'adresse d'une chaîne de caractères* qui contient le chemin d'accès au fichier sur un disque, par exemple "*C:\MonDoss\MonFic.txt*".

Remarque : il n'est pas indispensable de mentionner le chemin d'accès complet : par défaut, le fichier sera cherché à partir du dossier courant.

Remarque importante : La chaîne doit être impérativement suivie de l'octet 00h qui sert à marquer sa fin.

Il nous faut également spécifier le *mode d'accès* en écrivant dans AL un 0 (si on veut ouvrir le fichier en *lecture seule*), un 1 (si on veut l'ouvrir en *écriture seule*) ou un 2 (*lecture* ET *écriture*).

Si l'interruption échoue, le flag CF sera mis à 1 sans que le fichier soit ouvert. Dans le cas contraire, CF est mis 0 et le registre AX contient un petit nombre entier (par exemple 5) appelé « *handle* » (ce qui signifie « *poignée* ») du fichier. *Ce handle représente le fichier. C'est lui qu'il faudra désormais invoquer pour effectuer des opérations de lecture ou d'écriture, et non pas le chemin d'accès.*
En effet, les chemins d'accès ne sont donc plus d'aucune utilité puisque les fichiers sont ouverts dans la mémoire vive.

2. Lecture dans un fichier

Une fois le fichier ouvert, on peut le lire avec la fonction 3fh. Il suffit de mentionner le handle dans BX, le nombre d'octets à lire dans CX, et l'adresse d'un *buffer* dans DS:DX. Au cas où vous ne sauriez pas ce qu'est un *buffer* (ou *tampon*), sachez que c'est simplement une variable (généralement une chaîne de caractères) destinée à *recevoir des données* (ou à

en fournir). Dans notre cas, le buffer va recevoir les octets lus dans le fichier.

Après l'appel, AX contient le nombre d'octets qui ont été effectivement lus (il peut être inférieur à la taille demandée si le fichier n'est pas assez long). En cas de problème, CF sera mis à 1.

3. Ecriture dans un fichier

Pour écrire des données, on procède de même avec la fonction 40h. Les paramètres sont les mêmes que pour la fonction 3fh. Le *buffer* contient cette fois les octets à *écrire*. Après l'appel, le nombre d'octets qui ont été effectivement écrits est stocké dans AX (il sera être inférieur à la taille spécifiée si le disque est plein).

Les données sont écrites sur le disque dès que le tampon (dans la mémoire vive) est plein.

4. Existence d'un pointeur de fichier

Une question se pose cependant : *à quel endroit du fichier les données sont-elles lues (ou écrites) ?*

Réponse : quand un fichier est ouvert, un pointeur spécial pointant vers le début du fichier est créé. La première opération de lecture (ou d'écriture) se fera donc *au début du fichier*. Mais entre chaque opération, le pointeur est incrémenté de la taille des données que l'on a lues (ou écrites). La deuxième opération se fera donc sur les octets qui suivent ceux de la première.

Remarque : Il est possible de modifier directement le pointeur de fichier : voyez pour cela la fonction 42h...

5. Fermeture d'un fichier

Pour terminer, le fichier doit être refermé. Les modifications éventuellement apportées et non enregistrées seront écrites sur le disque, et le *handle* sera libéré. C'est la fonction 3eh qui se charge de tout cela. Elle attend simplement le handle du fichier dans BX. Et comme d'habitude, CF vaut 1 après l'appel si des erreurs ont été rencontées.

Remarque : Attention lorsque vous laissez le fichier ouvert longtemps afin d'y ajouter progressivement des données ! Si le système plante, vous perdrez les données qui se trouvent dans le tampon à ce moment. C'est pourquoi il est conseillé de forcer régulièrement l'écriture sur le disque en refermant le fichier.

6. Conclusion

Le tableau suivant récapitule ces différentes étapes :

Fonction	Description	Paramètres
3d*h*	***Ouvrir le fichier***	***- DS:DX*** *: adresse d'une chaîne contenant le chemin d'accès* ***- AL*** *: mode d'accès*
3e*h*	***Fermer le fichier***	***- BX*** *: handle*
3f*h*	***Lire le fichier***	***- BX*** *: handle* ***- CX*** *: nombre d'octets* ***- DS:DX*** *: adresse d'un buffer*

40*h*	***Ecrire dans le fichier***	***- BX** : handle* ***- CX** : nombre d'octets* ***- DS:DX** : adresse d'un buffer*

II. LES FONCTIONS DE RECHERCHE DE FICHIERS

Pour rechercher un fichier (ou un dossier), on se sert des fonctions 4eh (« *Find First* ») et 4fh (« *Find Next* »).

Imaginons par exemple que nous voulions chercher dans le dossier courant tous les fichiers qui portent l'extension "*.com*" afin de les supprimer. Comment devons-nous nous y prendre ?

1. La fonction 4eh

La fonction 4eh sert à définir des critères de recherche et à trouver le *premier* fichier qui correspond à ces critères (s'il existe).

On doit passer dans DS:DX l'adresse de la chaîne de caractères qui contient le *masque de recherche* (dans notre exemple, ce masque est "**.com*"). Par défaut, les fichiers sont cherchés dans le dossier courant. Mais on peut évidemment spécifier un autre chemin dans le masque.

Remarque importante : afin que le DOS puisse connaître sa taille, le masque doit impérativement être terminé par l'octet 00h !

On écrit également dans CX les attributs des fichiers que l'on désire trouver. Si CX vaut 0, seuls les fichiers « normaux » pourront être trouvés. En fait, chaque bit de CL représente un attribut, comme le montre le tableau ci-dessous :

Bit	Signification
1	***Lecture seule***
2	***Fichier caché***
3	***Fichier système***
4	***Volume***
5	***Répertoire***
6	***Fichier***
7	*(Aucune...)*
8	*(Aucune...)*

Pour demander à la fonction 4eh de ne pas oublier les fichiers cachés, il suffit donc de charger CX avec la valeur 2 (bit numéro 2 = 1). De même, l'attribut **00000111**b (soit 7) nous permettra de trouver les fichiers en lecture seule, les fichiers cachés et les fichiers systèmes.

Remarque : Ne vous souciez pas trop des bits numéro 4 et 6. Laissez-les à 0.

Une fois que les paramètres ont été ajustés, on peut appeler la fonction 4eh. *Si aucun fichier n'a été trouvé, le flag CF est mis à 1. On doit donc fait un test sur CF pour savoir si la recherche peut continuer ou si elle doit s'arrêter.*

Si au contraire la fonction a trouvé un fichier, les caractéristiques de ce fichier (i.e. son nom, sa taille, ses attributs,...) sont inscrits dans une zone de la mémoire appelée *DTA* (« *Disk Transfer Area* »).

Mais où se trouve donc cette DTA et à quoi ressemble-t-elle ?

Réponse : par défaut, le DOS place la DTA dans le PSP de votre programme, à l'offset 80h.

Remarque : il vous est naturellement possible de la déplacer en faisant appel à la fonction 1ah.

Voici la structure de la DTA :

Adresse	Description	Taille (octets)
00*h*	*Lettre du lecteur (0=courant, 1=A, 2=B, ...) sur lequel se trouve le fichier*	1
01*h*	*Modèle de la recherche*	11
0C*h*	*Réservé*	9
15*h*	*Attributs du fichier*	1
16*h*	*Heure du fichier*	2
18*h*	*Date du fichier*	2
1A*h*	*Taille du fichier*	4
1E*h*	*Nom du fichier avec l'extension*	13

Puisque par défaut la DTA est située dans le PSP à l'offset 80h, le nom du fichier trouvé est écrit à l'offset 80h + 1eh = 9eh. De même, la taille se trouve à l'offset 9ah, etc...

2. La fonction 4fh

Jusqu'à présent, nous n'avons trouvé qu'un seul fichier ! Pour poursuivre la recherche, appelez la fonction 4fh sans écrire aucun paramètre dans les registres. Les caractéristiques de votre recherche ont été mémorisées dans la DTA : vous n'avez donc pas à les rappeler. De même que pour la fonction 4eh, CF est mis à 1 si aucun nouveau fichier n'a été trouvé. C'est le signe que vous pouvez arrêter votre recherche.

3. Conclusion

Résumons nous :

Fonction	Description	Paramètres
4e*h*	*Trouver le premier fichier qui correspond aux caractéristiques spécifiées*	***- DS:DX*** *: adresse d'une chaîne contenant le masque* ***- CX*** *: attributs*
4f*h*	*Trouver le prochain fichier*	*Aucun !*

A titre d'exemple, écrivons à présent le programme que nous évoquions tout à l'heure : "*trouver tous les fichiers COM du dossier courant et les effacer*".

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

mov ah, 4eh ;fonction 4eh: chercher le PREMIER fichier correspondant
mov dx, offset Masque ;mettre l'offset du masque dans DX
xor cx, cx ;attributs : nous ne cherchons que les fichiers normaux...

cherchons_le_fichier :

int 21h
jc recherche_terminee ;si CF=1, alors la recherche est terminée !

;à présent, les caractéristiques du fichier trouvé sont dans la DTA...

mov ah, 41h ;fonction servant à effacer un fichier
mov dx, 80h+1eh ;mettre dans DX l'adresse du nom du fichier trouvé
int 21h ;effacer le fichier !

;Remarque : ce programme lui-même sera effacé !!

mov ah, 4fh ;fonction 4fh : chercher le prochain fichier correspondant
jmp cherchons_le_fichier

recherche_terminee :

ret

Masque db "*.COM", 0 ;ne pas oublier le code ASCII 0 !

code ends

end debut
```

III. EXEMPLE DE PROGRAMME

1. Entrée bufferisée au clavier

L'exemple que nous proposons ici fait appel à une saisie au clavier. Comme nous n'en n'avons pas encore rencontré, ce paragraphe explique brièvement comment utiliser les fonctions 0ah et 0ch.

On peut se servir de la fonction 0ah pour lire une chaîne de caractères au clavier. Le seul paramètre à fournir est *l'adresse d'un buffer*. Comme il est souvent indispensable d'effacer le buffer avant la lecture, il est préférable de ne pas utiliser la fonction 0ah directement mais d'appeler la fonction 0ch. En effet, celle-ci commence par *effacer le buffer*, puis *appelle la fonction 0ah* (ou une fonction voisine dont le numéro doit être transmis dans AL).

Voici le code à écrire :

```
;(...)

mov ah, 0ch
mov al, 0ah ;la fonction 0ch doit appeller la 0ah après avoir vidé le buffer
mov dx, offset maChaine ;offset du buffer qui recevra les caractères entrés
int 21h

;(...)
```

La seule difficulté est dans la déclaration du buffer. Voici comment vous devez vous y prendre si vous considérez que l'utilisateur pourra entrer au plus *n* caractères :

```
;(...)

maChaine db n+1, ?, n dup(?), ? ;ou bien : maChaine db n+1, n+2 dup(?)

;(...)
```

Le premier octet de "*maChaine*" doit contenir le nombre d'octets maximal qui pourra être entré, plus 1. Pourquoi plus 1 ? Tout simplement parce que le DOS met à la fin des caractères tapés le code ASCII 13 (retour chariot).

Pour comprendre la suite de la déclaration, il faut savoir de quelle manière le DOS transmet les caractères qui ont été lus. Après la lecture, le deuxième octet du buffer contiendra le nombre exact d'octets lus (sans compter le 13 final). La chaîne proprement dite ne commencera donc qu'au troisième octet. Puisqu'un octet est utilisé pour donner au DOS le nombre maximal de caractères autorisé, un autre pour recevoir le nombre de caractères lus effectivement, et encore un autre pour recevoir le code ASCII 13, le buffer doit comporter trois octets de plus que la taille maximale de la chaîne. C'est pourquoi on écrit :

maChaine db *n+1*, ?, *n* dup(?), ?

2. Le programme

Le programme suivant demande à l'utilisateur d'entrer le nom d'un fichier se trouvant dans le dossier courant puis crypte ce fichier en appliquant un NOT logique sur chaque octet. Naturellement, il n'est pas optimisé du tout. Ce n'est qu'un exemple !

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

mov ah, 09h
mov dx, offset message
int 21h ;écrire le message à l'écran...

mov ah, 0ch ;fonction d'effacement du buffer et de saisie au clavier
mov al, 0ah ;saisie au clavier d'une chaîne du caractères
mov dx, offset buffer ;buffer où sera placée la chaîne
int 21h

xor bx, bx ;mettre BX à zéro
mov bl, byte ptr [buffer+1] ;mettre dans bl le nombre d'octets lus au clavier
mov byte ptr [buffer + 2 + bx], 0 ;écrire un 0 après le nom du fichier

mov ah, 3dh ;ouvrir le fichier de départ
mov dx, offset buffer + 2 ;nom du fichier de départ
mov al, 0 ;mode d'accès = lecture seule
int 21h
jc existe_pas ;si CF=1, le fichier demandé n'existe pas ou est inaccessible

mov bx, ax ;mettre le handle dans BX

mov ah, 3ch ;créer le fichier d'arrivée
mov dx, offset nomcrypte ;nom du fichier à créer
xor cx, cx ;attributs
int 21h

mov ah, 3dh ;ouvrir le fichier d'arrivée
mov dx, offset nomcrypte
mov al,1 ;mode d'accès = écriture seule
int 21h

mov word ptr [handle_arrivee], ax ;stocker le handle

lire_10000_octets :

mov ah, 3fh ;lire des données dans le fichier de départ
mov cx, 10000 ;nombre d'octets à lire
mov dx, offset donnees ;buffer pour recevoir les données lues
int 21h

or ax, ax ;comparer AX avec 0
jz fin_du_fichier ;si AX=0, on n'a pas lu d'octet donc le cryptage est fini !

mov cx, ax ;mettre le nombre d'octets lus dans CX (pour la boucle)
xor di, di ;mettre DI à 0 (pour la boucle)

octet_suivant:

not byte ptr ds:[donnees+di] ;crypter un octet
inc di ;faire pointer DI vers l'octet suivant en l'incrémentant
loop octet_suivant ;boucler autant de fois qu'on a lu d'octets

mov dx, offset donnees ;offset des données à écrire dans DX
mov cx, ax ;nombre d'octets à écrire dans CX
push ax bx ;sauvegarder AX (nb d'octets à écrire) et BX (handle de départ)
mov bx, word ptr [handle_arrivee] ;mettre dans BX le handle d'arrivée
mov ah, 40h ;fonction qui sert à écrire dans un fichier
int 21h
pop bx ax ;restaurer BX et AX

cmp ax, 10000 ;a-t-on lu, crypté et écrit moins de 10000 octets ?
jb fin_du_fichier ;oui ? Alors le cryptage est terminé !
jmp lire_10000_octets ;non ? Alors va crypter 10000 autres octets...

fin_du_fichier:

mov ah, 3eh ;fermer le fichier de départ
int 21h

mov bx, word ptr [handle_arrivee] ;handle du fichier d'arrivée dans BX
mov ah, 3eh ;fermer le fichier d'arrivée
int 21h

ret ;rendre la main au DOS

existe_pas :

mov ah, 09h
mov dx, offset erreur
int 21h ;afficher un message d'erreur quand le fichier est inaccessible
ret ;rendre la main au DOS


;-------------------données--------------------

message db 10, 13, "Entrez le nom du fichier à crypter", 10, 13
db "(sauf crypte.txt !)", 10, 13, '$'
erreur db 10, 13, "Ce fichier n'existe pas ou est inaccessible !"10, 13, '$'

nomcrypte db "crypte.txt", 0 ;ne pas oublier le 0 terminal !!
buffer db 13, 14 dup(?) ;taille maxi = 12 (8 pour le nom, 4 pour l'extension)
handle_arrivee dw ?
donnees db 10000 dup(?)

code ends

end debut
```

Suggestion : quand vous aurez compris ce petit programme, essayez donc de l'améliorer ! Par exemple, il faudrait que l'utilisateur puisse taper le nom du fichier à crypter directement comme paramètre du programme (i.e. “CRY MONFICH.EXE”). Voyez pour cela la structure du PSP telle que présentée dans la première partie. De plus, le programme devrait pouvoir trouver automatiquement le nom du fichier d'arrivée : ce serait le même que celui du fichier à crypter, mais avec une extension différente (par exemple “.cry”).
Vous pourriez également étendre les possibilités du logiciel en permettant à l'utilisateur de saisir non plus un simple nom de fichier, mais un masque complexe avec des chemins d'accès et des caractères “jokers” comme '*' et '?'. Vous aurez donc besoin des fonctions de recherche 4eh et 4fh. L'utilisateur doit de surcroît pouvoir choisir s'il veut crypter un fichier ou bien le décrypter...

CONCLUSION

Voilà. Nous espérons que vous avez compris l'essentiel de ce cours, à savoir la logique de la programmation en assembleur. Les connaissances que vous avez acquises en lisant la fin du cours devraient vous permettre de mieux saisir les premiers chapitres. C'est pourquoi nous vous invitons à relire ce tutoriel.

Certaines notions doivent encore vous paraître obscures. Maintenant que vous vous êtes familiarisé avec l'assembleur, commencez par écrire et tester quelques petits programmes COM inutiles et très simples. Vous verrez qu'au bout de quelques essais, tout ce que vous avez lu vous semblera très concret.

Lorsque vous aurez écrit vous-même quelques tout petits programmes qui fonctionnent, l'assimilation du langage deviendra alors très rapide, car l'assembleur est un langage très logique et très cohérent.

Nous vous conseillons d'écrire une librairie de petites macros ou procédures qui vous serviront dans tous vos programmes. Par exemple, vous pouvez faire une macro qui affiche un entier à l'écran, ou bien qui convertit une chaîne de caractères en un nombre entier.

Voici un exemple de macro qui renvoie dans AL le nombre de chiffres d'un entier non signé de deux octets passé dans AX :

```
nbdgt macro

local boucle, termine

pushf ;sauvegarder les flags sur la pile
push bx cx dx ;sauvegarder reg.
xor cx, cx ;i.e. MOV CX, 0

boucle:

xor dx, dx ;i.e. "MOV DX, 0". Indispensable car "DIV BX" utilise DX.
mov bx, 10
div bx ;division euclidienne de DX:AX par 10. Quotient = AX, reste = DX.
inc cx
or ax, ax ;i.e. "CMP AX, 0"
jnz boucle ;boucler si AX<>0. Sinon, on a atteint le dernier chiffre...

termine:

mov al, cl ;renvoyer le nb de chiffres dans AL
pop dx cx bx ;restaurer reg.
popf ;restaurer les flags

endm
```

N.B. : Pensez avant toute chose à vous munir d'une liste des interruptions du DOS. Vous ne pourrez rien faire si vous n'en avez pas !

Bon courage et bonne chance !