TP : Initiation au langage assembleur x86

I. LA MEMOIRE VIVE

1. La segmentation de la mémoire

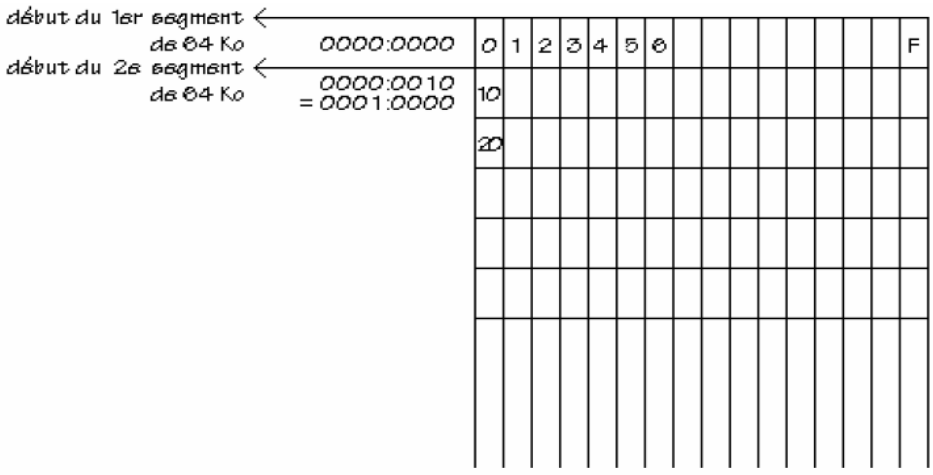
Le microprocesseur peut fonctionner sous deux modes : le mode réel et le mode protégé. Le mode protégé permet d'accéder à 2^{32} octets de mémoire vive, alors que le mode réel ne peut en adresser que 2^{20} = 1 Mo. Nous ne traiterons dans ce TP que le mode réel.
Un PC est conçu pour gérer 1 Mo (soit 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.

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 les uns à la suite des autres. Ils n'attendent pas que les segments qui les précédent soient terminés avant de commencer ! Ils se chevauchent.
Autrement dit, le deuxième segment ne démarre pas à l'octet 65536, mais à l'octet 16 ! Le troisième démarre à l'octet 32 et ainsi de suite…



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.

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 segment de mémoire réservé, et copie le programme à charger à la suite.
Le PSP (Program Segment Prefix) est une zone de 256 (= 100h) octets qui contient des informations diverses au sujet du programme. Par exemple, c'est dans le PSP que se trouvent les paramètres de la commande tapée par l'utilisateur.
Dans un segment alloué à un programme COM, le programme lui-même débute à l'offset 100h et non à l'offset 0h.

b) les fichiers EXE
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, 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.

II. LES REGISTRES

1. Généralités

Une instruction se compose d'un code opérateur (souvent appelé « opcode ») qui désigne l'action à effectuer (B4 par exemple) et d'un champ d'opérandes sur lesquelles porte l'opération (par exemple 4C).
Les registres des processeurs Intel x86 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)

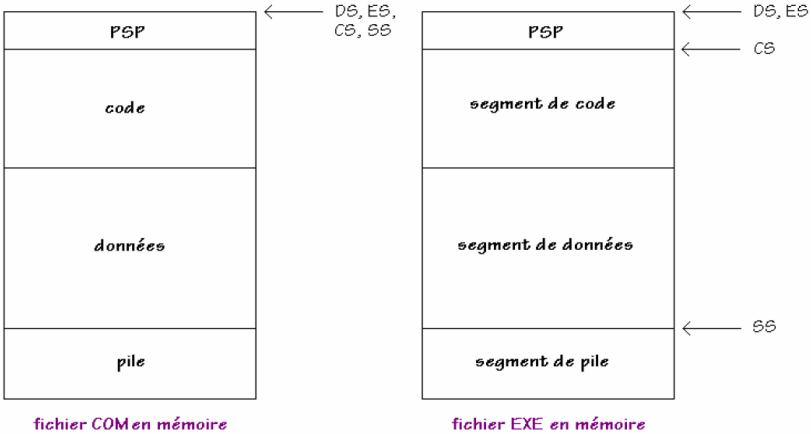
2. Les registres généraux

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.
AX : Accumulateur
BX : Base
CX : Compteur
DX : Données
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.

3. Les registres de segment

Contrairement aux registres généraux, ces registres ne peuvent pas servir pour les opérations courantes : ils ont un rôle très précis. Ils sont au nombre de quatre :
CS (Code segment) : Segment de code
DS (Data segment) : Segment de données
ES (Extra segment) : Extra segment
SS (Stack segment) : Segment de pile

Dans le registre CS est stockée l'adresse de segment de la prochaine instruction à exécuter.
Le registre DS est quant à lui destiné à contenir l'adresse du segment des données du programme en cours.
ES est un registre qui sert à adresser le segment de son choix. 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.



Dans le cas d'un fichier COM, le programme ne comporte qu'un seul segment, donc il suffit au DOS de charger CS, DS, ES et SS avec l'adresse d'implantation.
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

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.
Le registre SP désigne le sommet de la pile. La pile ne peut stocker que des mots (16 bits). La pile est extrêmement utile lorsqu'il s'agit de stocker provisoirement le contenu d'un registre qui doit être modifié.

Les trois derniers registres sont beaucoup moins liés au fonctionnement interne du processeur.
Les registres SI, DI et BP peuvent être utilisés 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).

5. Le registre des indicateurs

Les flags fournissent des informations sur les résultats des opérations précédentes. Ils sont tous regroupés dans un registre : le registre des indicateurs.
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.
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.
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.
Remarque : Les notations CF, PF, AF, etc… ne sont pas reconnues par l'assembleur. Pour utiliser les flags, il existe des instructions spécifiques.

III. 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 du processeur, 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 »).

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. Pour écrire une chaîne de caractères à l'écran ou pour lire un caractère entré au clavier par exemple, 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 le travail. Les paramètres (ou leurs adresses) sont passés dans les registres.

Certaines interruptions, comme la 21h, permettent d'appeler de nombreuses fonctions très diverses. Pour cela, il suffit de mentionner leur numéro dans le registre AH. Si la fonction a besoin de données, il faut mettre ces données dans le registre DX avant de lancer l'interruption.
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.

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

Il est indispensable, une fois l'ISR exécutée, que le programme interrompu retrouve les registres dans le même état qu'ils étaient auparavant. 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.

Exercices

Préliminaires :
-Le code peut être écrit en majuscules ou en minuscules.
-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'.
-La compilation crée un fichier objet ('.obj'). Pour obtenir un fichier COM, tapez "TLINK /tdc MONPROG".

1)Tapez le programme suivant (**sans les commentaires**) dans le fichier hello.asm, compilez-le et exécutez-le.

```
.386 ;indique au compilateur que le programme est destiné à tourner sur des processeurs INTEL de modèle >=386.
code segment use16 ;Cette directive déclare un segment que l'on appelle "code", dont les adresses (segment et offset) sont codées sur 16 bits.
assume cs:code, ds:code, ss:code ;Cette directive informe le compilateur que CS, DS et SS pointeront de façon privilégiée vers le segment "code"
org 100h ;offsets décalés de 100h
debut : ;label de début de programme
mov ah, 09h ;fonction : 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 tout le monde !",'$' ;définition du message
code ends ;indique la fin du segment "code"
end debut ;informe le compilateur que le fichier est fini
```

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.
"message" est un label de donnée. Il représente l'adresse du code ASCII de 'B'.
db (define byte) permet de déclarer et d'initialiser la variable "message". On pourrait aussi avoir dw (define word) ou dd (define double word).
'$' permet à la fonction de déterminer la fin de la chaîne à écrire.

a) Pourquoi demande-t-on que les offsets soient décalés de 100h (256) ?
b) Quels sont les numéros d'interruption et de fonction utilisées ici pour afficher des caractères à l'écran ?

2)Tapez le programme suivant (pile.asm) puis assemblez-le.

```
.386
code segment use16
org 100h
debut:
mov al,33
push al
ret
code ends
end debut
```

Lancez le debugger par la commande "td.exe pile".
Le panneau de gauche représente les instruction et leurs adresses. Celui de droite affiche le contenu des registre et l'état des flags ainsi que celui de la pile (en bas).
Tapez une fois sur la touche F7 pour exécuter la première instruction et notez les changements opérés dans le panneau de droite. Quels registres ont été modifiés. Pourquoi ?
Tapez une deuxième fois sur la touche F7. Quel registre a été modifié ? Notez sa valeur.
Modifiez le programme de manière à empiler le nombre 66 au dessus du nombre 33.
Relancez le debugger et expliquez quelles sont les particularités de la pile.
Lancez le debugger sur le programme hello et analysez son fonctionnement.

3) Ecrivez une procédure qui parmi deux entiers (passés en paramètres sur la pile) renvoie le plus grand dans AX. Consultez pour cela la liste des instructions plus loin dans ce document.

LECTURE ET ECRITURE DE FICHIERS

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.
Un buffer 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 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.

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

Récapitulation des différentes étapes :

3dh Ouvrir le fichier
- DS:DX : adresse d'une chaîne contenant le chemin d'accès
- AL : mode d'accès

3eh Fermer le fichier
- BX : handle

3fh Lire le fichier
- BX : handle
- CX : nombre d'octets
- DS:DX : adresse d'un buffer

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

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.

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 00000111b (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 :
00h Lettre du lecteur (0=courant, 1=A, 2=B, ...) sur lequel se trouve le fichier
01h Modèle de la recherche
0Ch Réservé
15h Attributs du fichier
16h Heure du fichier
18h Date du fichier

1Ah Taille du fichier
1Eh Nom du fichier avec l'extension
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

Résumé des fonctions :

4eh Trouver le premier fichier qui correspond aux caractéristiques spécifiées
- DS:DX : adresse d'une chaîne contenant le masque
- CX : attributs

4fh Trouver le prochain fichier

Exercices :

1) Ecrire le programme suivant : “trouver tous les fichiers ayant l'extension .ZZZ du dossier courant et les effacer”.
2) Ecrire un programme qui écrit à l'écran les noms des fichiers et dossiers du répertoire courant.

LES INSTRUCTIONS DU LANGAGE ASSEMBLEUR :

1. L’instruction NOP (« No Operation »)
Syntaxe : NOP
Description : Ne fait rien !

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.
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.
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.
-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

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.

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.

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 »)
b) l’instruction SCASW (« Scan String Word »)
c) l’instruction LODSW (« Load String Word »)
d) l’instruction STOSW (« Store String Word »)
e) l’instruction CMPSW (« Compare String Word »)

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 »)
b) l’instruction SCASD (« Scan String Double Word »)
c) l’instruction LODSD (« Load String Double Word »)
d) l’instruction STOSD (« Store String Double Word »)
e) l’instruction CMPSD (« Compare String Double Word »)

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

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.