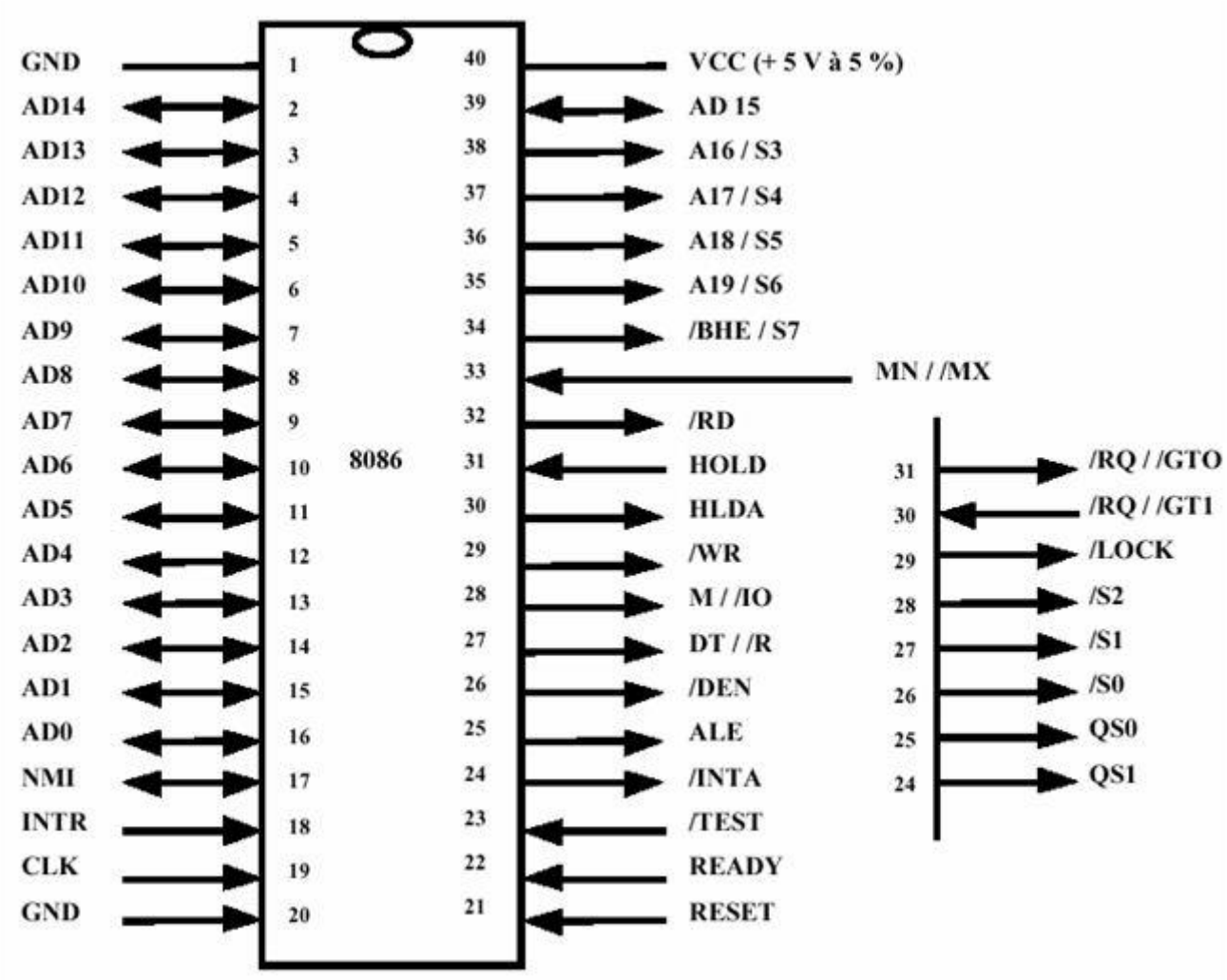
Le microprocesseur 8086 / 8088

I) Introduction :

Le processeur 8086 d'Intel est à la base des processeurs Pentium actuels. Les processeurs successifs (de PC) se sont en effet construits petit à petit en ajoutant à chaque processeurs des instructions et des fonctionnalités supplémentaires, mais en conservant à chaque fois les spécificités du processeur précédent. C'est cette façon d'adapter les processeurs à chaque étape qui permet qu'un ancien programme écrit pour un 8086 fonctionne toujours sur un nouvel ordinateur équipé d'un Pentium IV.

II) Architecture externe du 8086 :

Le 8086 est un circuit intégré de forme DIL de 40 pattes comme le montre la figure suivante :



Le 8086 (développé en 1978) est le premier microprocesseur de type x86.Il est équipé d'un bus de données de 16 bits et un bus d'adresses de 20 bits et fonctionne à des fréquences diverses selon plusieurs variantes : 5, 8 ou 10 MHz.

III) Architecture interne du 8086 :

Il existe deux unités internes distinctes : l'UE (Unité d'Exécution) et l'UIB (Unité d'Interfaçage avec le Bus). Le rôle de l'UIB est de récupérer et stocker les informations à traiter, et d'établir les transmissions avec les bus du système. L'UE exécute les instructions qui lui sont transmises par l'UIB. L'image ci-dessous résume les notions présentées ici. Le microprocesseur pris comme exemple est le 8086/8088. Les processeurs actuels de la famille x86 traitent les informations de la même façon.

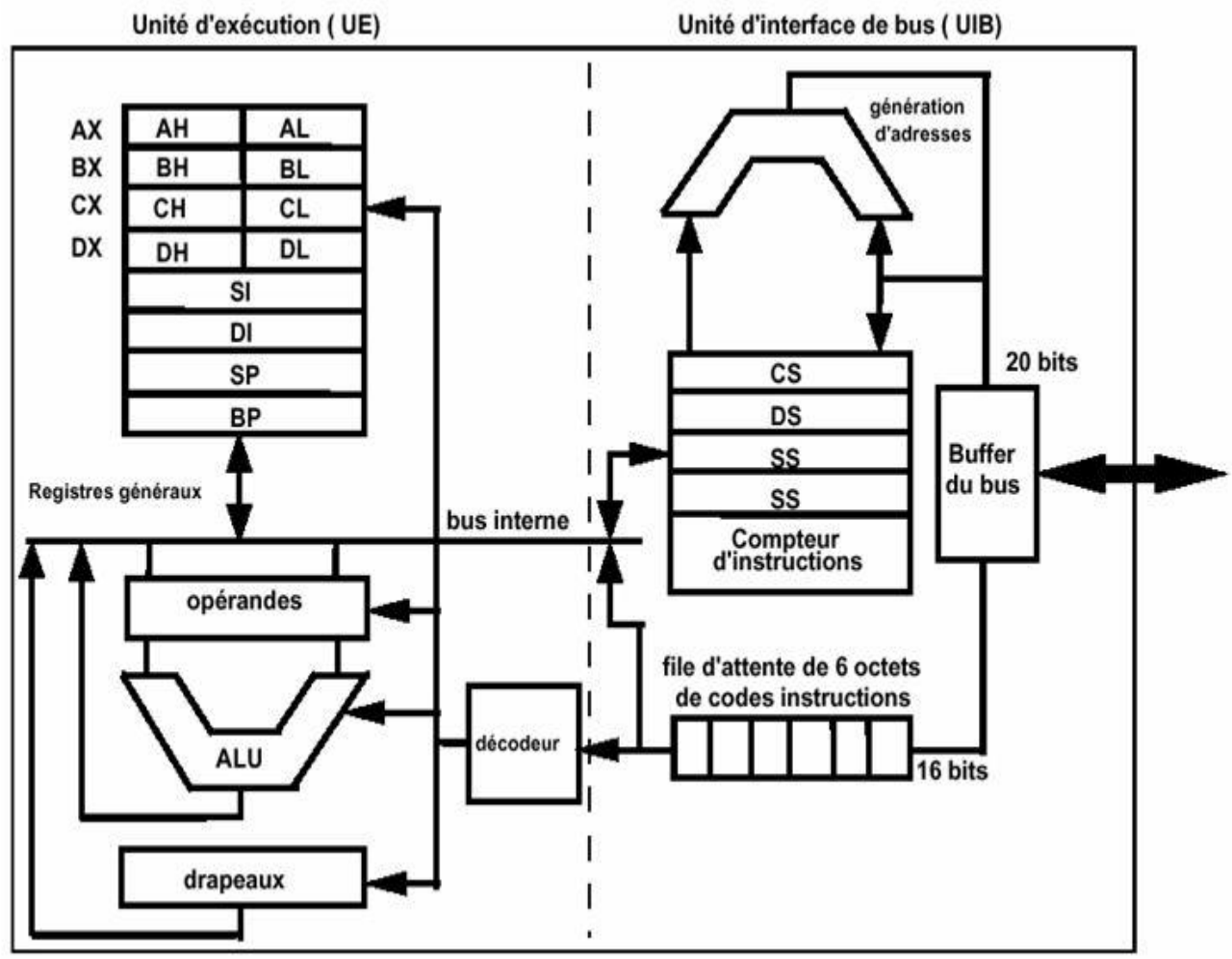
Nous pouvons à présent examiner plus en détail le traitement des instructions par ’UE et l'UIB. Avec le microprocesseur 8085, le traitement des instructions se passait comme suit :

-Extraction des instructions par l'UIB

- Exécution des instructions

- Extraction des nouvelles instructions

Lorsque l'exécution d'une instruction est terminée, l'UE reste inactif un court instant, pendant que l'UIB extrait l'instruction suivante. Pour remédier à ce temps d'attente, le prétraitement ou traitement pipeline a été introduit dans le 8086/8088. Pendant que l'UE exécute les informations qui lui sont transmises, l'instruction suivante est chargée dans l'UIB. Les instructions qui suivront sont placées dans une file d'attente. Lorsque l'UE a fini de traiter une instruction l'UIB lui transmet instantanément l'instruction suivante, et charge la troisième instruction en vue de la transmettre à l'UE. De cette façon, l'UE est continuellement en activité. Dans la figure suivante nous pouvons observer un schéma plus détaillé de l'UE et l'UIB. Nous y retrouvons les éléments dont il a été question précédemment.



Donc en conclusion on peut dire que le 8086/8088 se compose essentiellement de deux unités : la BIU qui fournit l'interface physique entre le microprocesseur et le monde extérieur et l'EU qui comporte essentiellement l'UAL de 16 bits qui manipule les registre généraux de 16 bits aussi .

Remarque :

La file d'attente d'instructions contient des informations qui attendent d'être traitées par l'UE. La file d'attente est parfois appelée capacité *de* traitement. Le microprocesseur 8086 est capable de mémoriser jusqu'à six octets. Les microprocesseurs actuels sont bien entendu équipés d'une file d'attente plus rapide et plus large, c'est à dire capable d'emmagasiner plus d'informations.

IV) Les registres du 8086/8088 :

IV-1) Introduction :

Le jeu de registres contient l'ensemble des registres du microprocesseur. Un registre est une petite partie de mémoire intégrée au microprocesseur, dans le but de recevoir des informations spécifiques, notamment des adresses et des données stockées durant l'exécution d'un programme. Il existe plusieurs types de registres. Certains d'entre eux sont

affectés à des opérations d'ordre général et sont accessibles au programmeur à tout moment. Nous disons alors qu'il s'agit de registres généraux. D'autres registres ont des rôles bien plus spécifiques et ne peuvent pas servir à un usage non spécialisé.

IV-2) Les registres généraux :

Les registres généraux peuvent être utilisés dans toutes les opérations arithmétiques et logiques que le programmeur insère dans le code assembleur. Un registre complet présente une grandeur de 16 bits. Comme le montre la figure 2, chaque registre est en réalité divisé en deux registres distincts de 8 bits. De cette façon, nous pouvons utiliser une partie du registre si nous désirons y stocker une valeur n'excédant pas 8 bits. Si, au contraire, la valeur que nous désirons y ranger excède 8 bits, nous utiliserons le registre complet, c'est à dire 16 bits. Nous verrons plus loin qu'il est possible de manipuler facilement les registres généraux.

Le programmeur dispose de 8 registres internes de 16 bits qu'on peut diviser en deux groupes comme le montre la figure 2 :

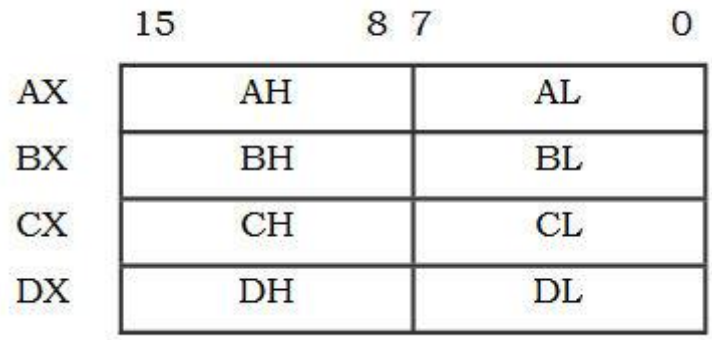
- *groupe de données* : formé par 4 registres de 16 bits (AX,BX,CX,et

DX) chaque registre peut être divisé en deux registres de 8 bits

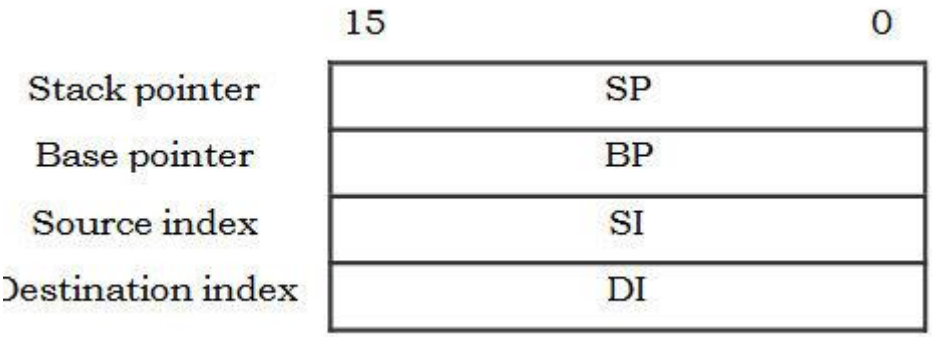
(AH, AL, BH, BL, CH, CL, DH et DL)

- *groupe de pointeur et indexe* : formé de 4 registres de 16 bits (SI, DI, SP, BP) et font généralement référence à un emplacement en mémoire.

Groupe de données :



Groupe de pointeur et indexe :



IV-2-1) Groupe de données :

Registre AX : (Accumulateur)

Toutes les opérations de transferts de données avec les entrées-sorties ainsi que le traitement des chaînes de caractères se font dans ce registre, de même les opérations arithmétiques et logiques.

Les conversions en BCD du résultat d'une opération arithmétique (addition, soustraction, multiplication et la division) se font dans ce registre.

Registre BX : (registre de base)

Il est utilisé pour l'adressage de données dans une zone mémoire différente de la zone code : en général il contient une adresse de décalage par rapport à une adresse de référence.). (Par exemple, l'adresse de début d'un tableau). De plus il peut servir pour la conversion d'un code à un autre.

Registre CX : (Le compteur)

Lors de l'exécution d'une boucle on a souvent recours à un compteur de boucles pour compter le nombre d'itérations, le registre CX a été fait pour servir comme compteur lors des instructions de boucle.

Remarque :

Le registre CL sert en tant que compteur pour les opérations de décalage et de rotation, dans ce cas il va compter le nombre de décalages (rotation) de bits à droite ou à gauche.

Registre DX :

On utilise le registre DX pour les opérations de multiplication et de division mais surtout pour contenir le numéro d'un port d'entrée/sortie pour adresser les interfaces d'E/S.

IV-2-2) Groupe de pointeur et indexe :

Ces registres sont plus spécialement adaptés au traitement des éléments dans la mémoire. Ils sont en général munis de propriétés d'incrémentation et de décrémentation.

Un cas particulier de pointeur est le pointeur de pile (Stack Pointer : SP). Ce registre permet de pointer la pile pour stocker des données ou des adresses selon le principe du "Dernier Entré Premier Sorti" ou "LIFO"

(Last In First Out).

L'idexe SI : (source indexe) :

Il permet de pointer la mémoire il forme en général un décalage (un offset) par rapport à une base fixe (le registre DS), il sert aussi pour les instructions de chaîne de caractères, en effet il pointe sur le caractère source

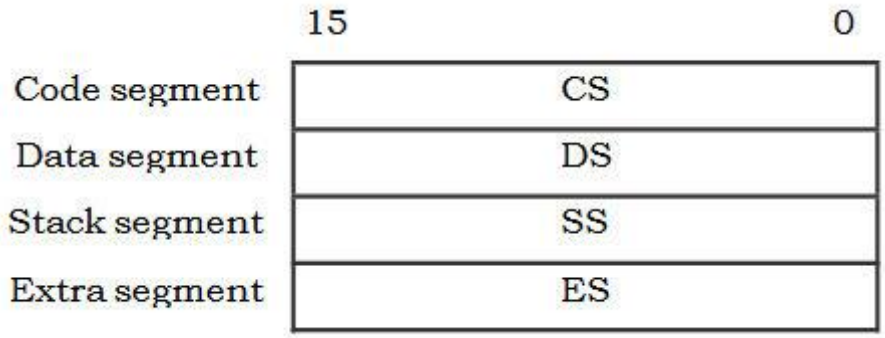
L'index DI : (Destination indexe) :

Il permet aussi de pointer la mémoire il présente un décalage par rapport à une base fixe (DS ou ES), il sert aussi pour les instructions de chaîne de caractères, il pointe alors sur la destination

Les pointeurs SP et BP : (Stack pointer et base pointer)

Ils pointent sur la zone pile (une zone mémoire qui stocke l'information avec le principe filo : voir plus loin), ils presentent un décalage par rapport à la base (le registre SS). Pour le registre BP il a un rôle proche de celui de BX, mais il est généralement utilisé avec le segment de pile.

IV -2- 3) Les registres segment:



Le 8086 a quatre registres segments de 16 bits chacun : CS (code segment, DS (Data segment), ES (Extra segment) et SS (stack segment), ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le début de chacun d'entre eux. Chaque segment de mémoire ne peut excéder les 65535 octets.

Le registre CS (code segment) :

Il pointe sur le segment qui contient les codes des instructions du programme en cours.

Remarque :

Si la taille du programme dépasse les 65535 octets alors on peut diviser le code sur plusieurs segments (chacun ne dépasse pas les 65535 octets) et pour basculer d'une partie à une autre du programme il suffit de changer la valeur du registre CS et de cette manière on résout le problème des programmes qui ont une taille supérieure à 65535 octets.

Le registre DS (Data segment) :

Le registre segment de données pointe sur le segment des variables globales du programme, bien évidemment la taille ne peut excéder 65535 octets (si on a des données qui dépassent cette limite, on utilise la même astuce citée dans la remarque précédente mais dans ce cas on change la valeur de DS).

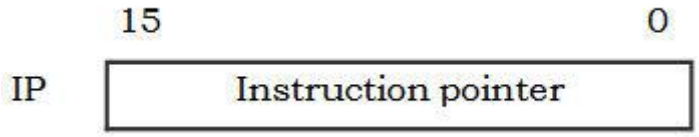
Le registre ES (Extra segment) :

Le registre de données supplémentaires ES est utilisé par le microprocesseur lorsque l'accès aux autres registres est devenu difficile ou impossible pour modifier des données, de même ce segment est utilisé pour le stockage des chaînes de caractères.

Le segment SS (Stack segment) :

Le registre SS pointe sur la pile : la pile est une zone mémoire ou on peut sauvegarder les registres ou les adresses ou les données pour les récupérer après l'exécution d'un sous-programme ou l'exécution d'un programme d'interruption, en général il est conseillée de ne pas changer le contenu de ce registre car on risque de perdre des informations très importantes (exemple les passages d'arguments entre le programme principal et le sous-programme).

IV-2-4) Le registre IP : (Le compteur de programme) :



Instruction Pointer ou Compteur de Programme, contient l'adresse de l'emplacement mémoire où se situe la prochaine instruction à exécuter. Autrement dit, il doit indiquer au processeur la prochaine instruction à exécuter. Le registre IP est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante.

II-2-5 : Le registre d'état (Flag) :

Le registre d'état FLAG sert à contenir l'état de certaines opérations effectuées par le processeur. Par exemple, quand le résultat d'une opération est trop grand pour être contenu dans le registre cible (celui qui doit contenir le résultat de l'opération), un bit spécifique du registre d'état (le bit OF) est mis à 1 pour indiquer le débordement.

Remarque : **Drapeaux (flags)**

Les drapeaux sont des indicateurs qui annoncent une condition particulière suite à une opération arithmétique ou logique.

Le registre d'état du 8086 est formé par les bits suivants :

15															0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

Remarque :

X : bit non utilise.

CF (*Carry Flag) :*

Retenue: cet indicateur et mis à 1 lorsque il y a une retenue du résultat à 8 ou 16 bits. il intervient dans les opérations d'additions (retenue) et de soustractions (borrow) sur des entiers naturels. Il est positionné en particulier par les instructions ADD, SUB et CMP (comparaison entre deux valeurs).

CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes. Exemples (sur 8 bits pour simplifier) :

```
    10010110          11011001
+   01010100      +   01010010
CF=0 11101010     CF=1 00101011
```

PF (*Parity Flag) :*

Parité : si le résultat de l'opération contient un nombre pair de 1 cet indicateur est mis à 1, sinon zéro.

AF (*Auxiliary Carry) :*

Demie retenue : Ce bit est égal à 1 si on a une retenue du quarter de poids faible dans le quartier de poids plus fort.

ZF (Zero Flag) :

Zéro : Cet indicateur est mis à 1 quand le résultat d'une opération est égal à zéro. Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), ZF=1 indique que les deux **opérandes étaient égaux. Sinon, ZF est positionné à 0.**

SF (*Sign Flag) :*

SF est positionné à 1 si le bit de poids fort du résultat d'une addition ou soustraction est 1 ; sinon SF=0. SF est utile lorsque l'on manipule des entiers signés, car le bit de poids fort donne alors le signe du résultat. Exemples (sur 8 bits) :

```
    10010110                11011001
  + 01010100              + 01010010
SF=1 11101010           SF=0 00101011
```

OF (*Overflow Flag) :*

Débordement : si on a un débordement arithmétique ce bit est mis à 1.c a d le résultat d'une opération excède la capacité de l'opérande (registre ou case mémoire), sinon il est à 0.

DF (*Direction Flag) :*

Auto Incrémentation/Décrémentation : utilisée pendant les instructions de chaîne de caractères pour auto incrémenté ou auto décrémenter le SI et le DI.

IF (*Interrupt Flag) :*

Masque d'interruption : pour masquer les interruptions venant de l'extérieur ce bit est mis à 0, dans le cas contraire le microprocesseur reconnaît l'interruption de l'extérieur.

TF (*Trap Flag) :*

Piége : pour que le microprocesseur exécute le programme pas à pas du.

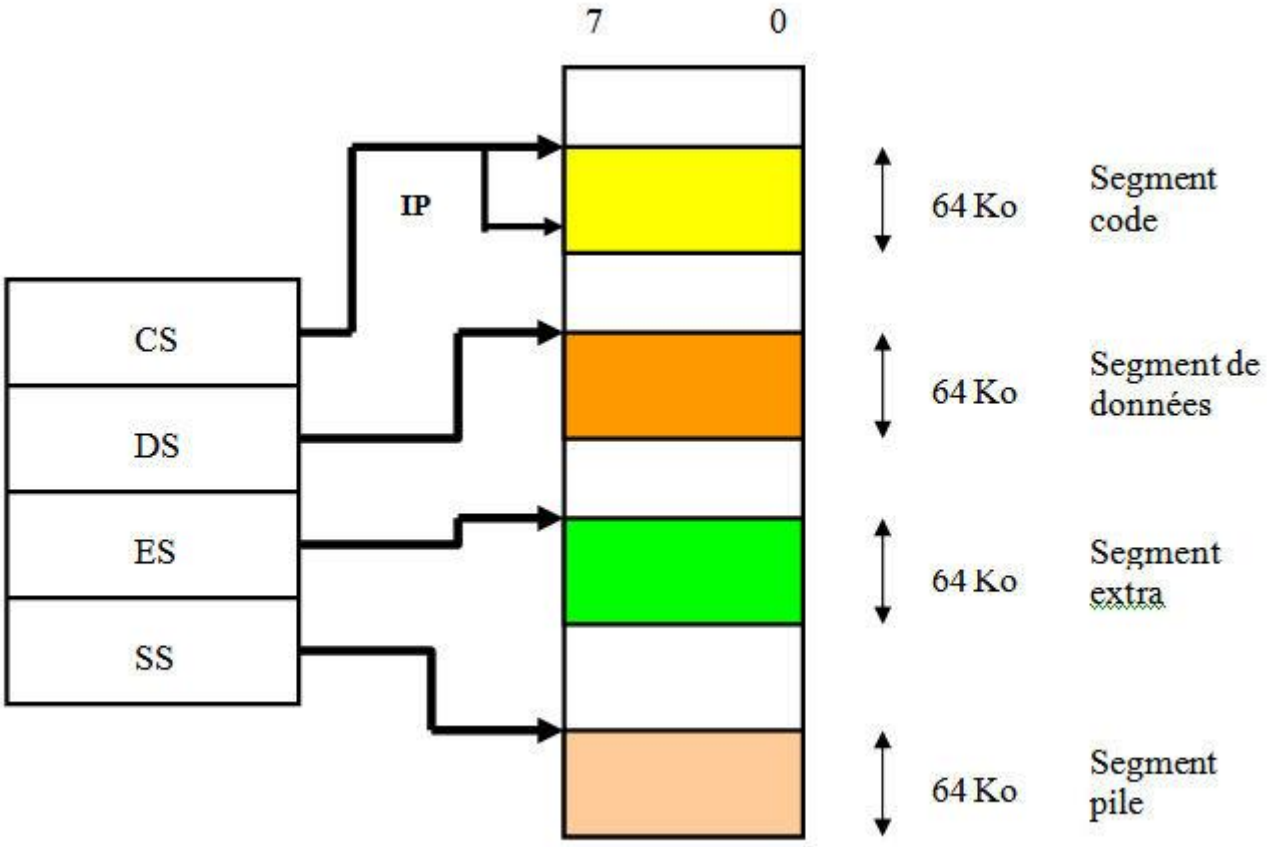
Remarque :

Les instructions de branchements conditionnels utilisent les *indicateur*s (drapeaux), qui sont des bits spéciaux positionnés par l'UAL après certaines opérations. Chaque indicateur est manipulé individuellement par des instructions spécifiques.

V) Gestion de la mémoire :

V-1) Introduction :

L'espace mémoire adressable (1 méga = 2^{20} bits du bus d'adresse) du 8086/8088 est divisé en quatre segment logiques allant jusqu'à 64 KOctets chacun . L'accès à ces espaces est direct et simultané, or Le compteur de programme est de 16 bits donc la possibilité d'adressage est de 2 16 = 64 Ko (Ce qui ne couvre pas la totalité de la mémoire), alors on utilise deux registres pour indiquer une adresse au processeur, Chaque segment débute à l'endroit spécifié par le registre segment. Le déplacement (offset) à l'intérieur de chaque segment se fait par un registre de décalage qui permet de trouver une information à l'intérieur du segment. Exemple la paire de registre CS:IP : pointe sur le code d'une instruction (CS registre segment et IP Déplacement)



V-2) Adresse physique (Segmentation de la mémoire) :

Nous abordons ici le problème de la segmentation de la mémoire. Nous venons de voir qu'en assembleur, les données étaient normalement regroupées dans une zone mémoire nommée *segment de donnée*s, tandis que les instructions étaient placées dans un segment *d'instruction*s (de même pour le segment pile et segment de données supplémentaires). Ce partage se fonde sur la notion plus générale de *segment de mémoir*e, qui est à la base du mécanisme de gestion des adresses par les processeurs 80x86. On a vu aussi que le registre IP, qui stocke l'adresse d'une instruction, fait lui aussi 16 bits. Or, avec 16 bits il n'est possible d'adresser que 2 16 = 64 Kilo octets. Le bus d'adresses du 8086 possède 20 bits. Cette adresse de 20 bits est formée par la juxtaposition d'un registre segment (16 bits de poids fort) et d'un déplacement *(offse*t, 16 bits de poids faible). Adresse physique= Base * 16+ offset

Le schéma de la figure suivante illustre la formation d'une adresse 20 bits à partir du segment et du déplacement sur 16 bits :

Bits	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
	Base 0 0 0 0
+	0 0 0 0 Offset
=	Adresse physique

Remarque :

On appellera *segment de mémoire* une zone mémoire adressable avec une valeur fixée du segment (les 16 bit de poids fort). Un segment a donc une taille maximale de 64 Ko.

Exemple :

Pour CS=1000 et IP = 2006

Adresse physique =

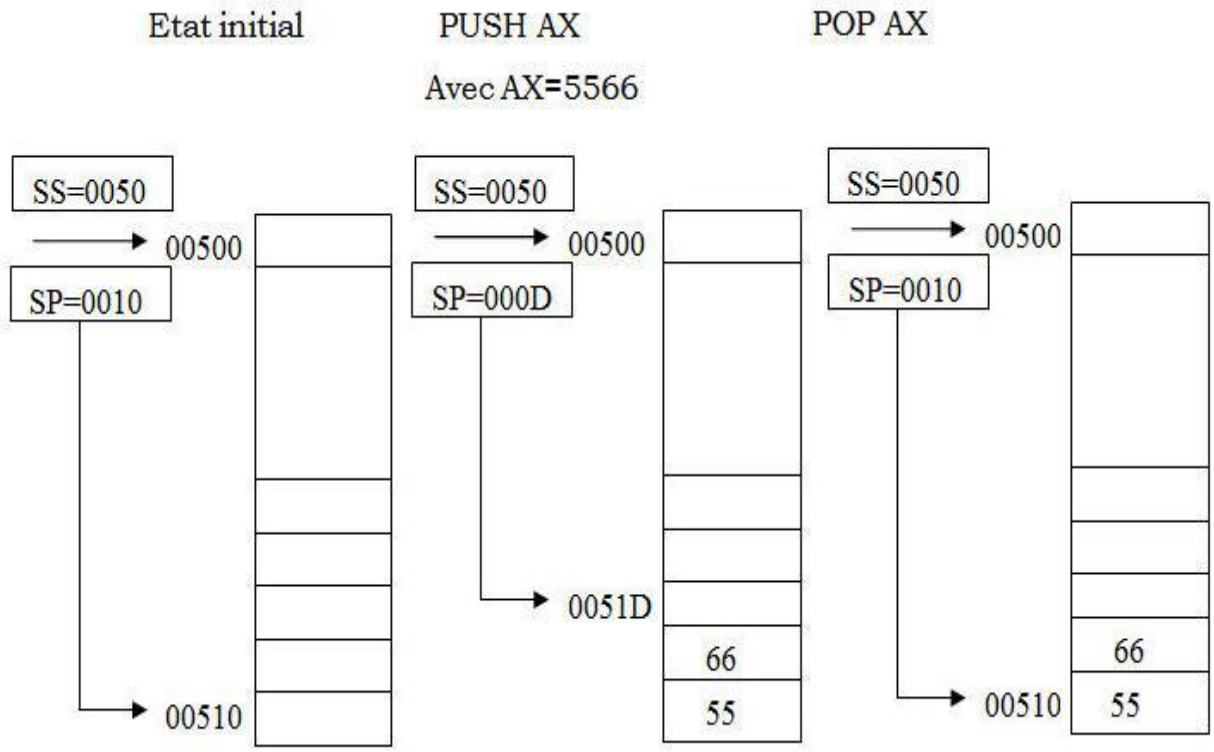
```
   10000
+  2006
=  12006
```

V-3) Implémentation de la pile :

Le pointeur de pile (en combinaison avec le segment de pile SS) pointe vers le dessus de la pile (TOS : top of stack) en mémoire. Une pile est un ensemble de données placées en mémoire de manière à ce que seulement la donnée du "dessus" soit disponible à un instant donné. Pour aller chercher la donnée sous celle du dessus par exemple, on doit d'abord enlever celle du dessus. Le rôle du pointeur de pile (et de la pile vers laquelle il pointe) est le suivant. Quand un processeur exécute une instruction, il est possible qu'il soit interrompu par une "Interruption" (c'est-à-dire un appel au processeur qui est prioritaire aux instructions du programme qu'il traite). Il doit alors arrêter de s'occuper de l'instruction qu'il traite présentement pour s'occuper de l'interruption. Quand l'interruption sera traitée, il retournera à l'instruction qu'il traitait quand il a été interrompu. Mais pour cela, il doit se rappeler de cette instruction ainsi que de l'état de certains registres au moment où il traitait l'instruction. Donc pour ne pas les perdre, il les placera temporairement dans une pile (à l'intérieur de la mémoire RAM par exemple) et pourra les récupérer une fois l'interruption traitée. Le pointeur de pile (SP) donne donc l'adresse en mémoire de cette pile temporaire.

Les *piles* offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs.

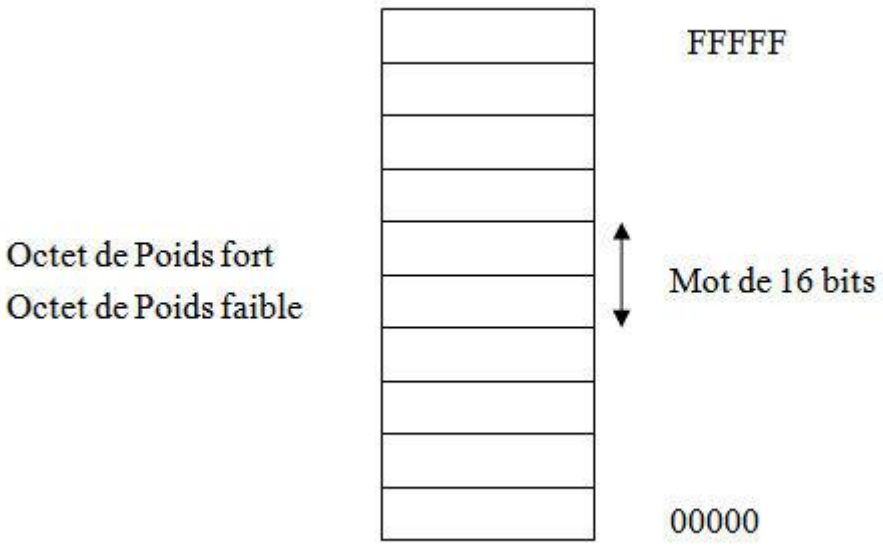
Le schéma suivant montre comment une valeur est stocker dans la pile (pushed) et comment elle est récupérée (poped) :



V-4) Organisation de la mémoire :

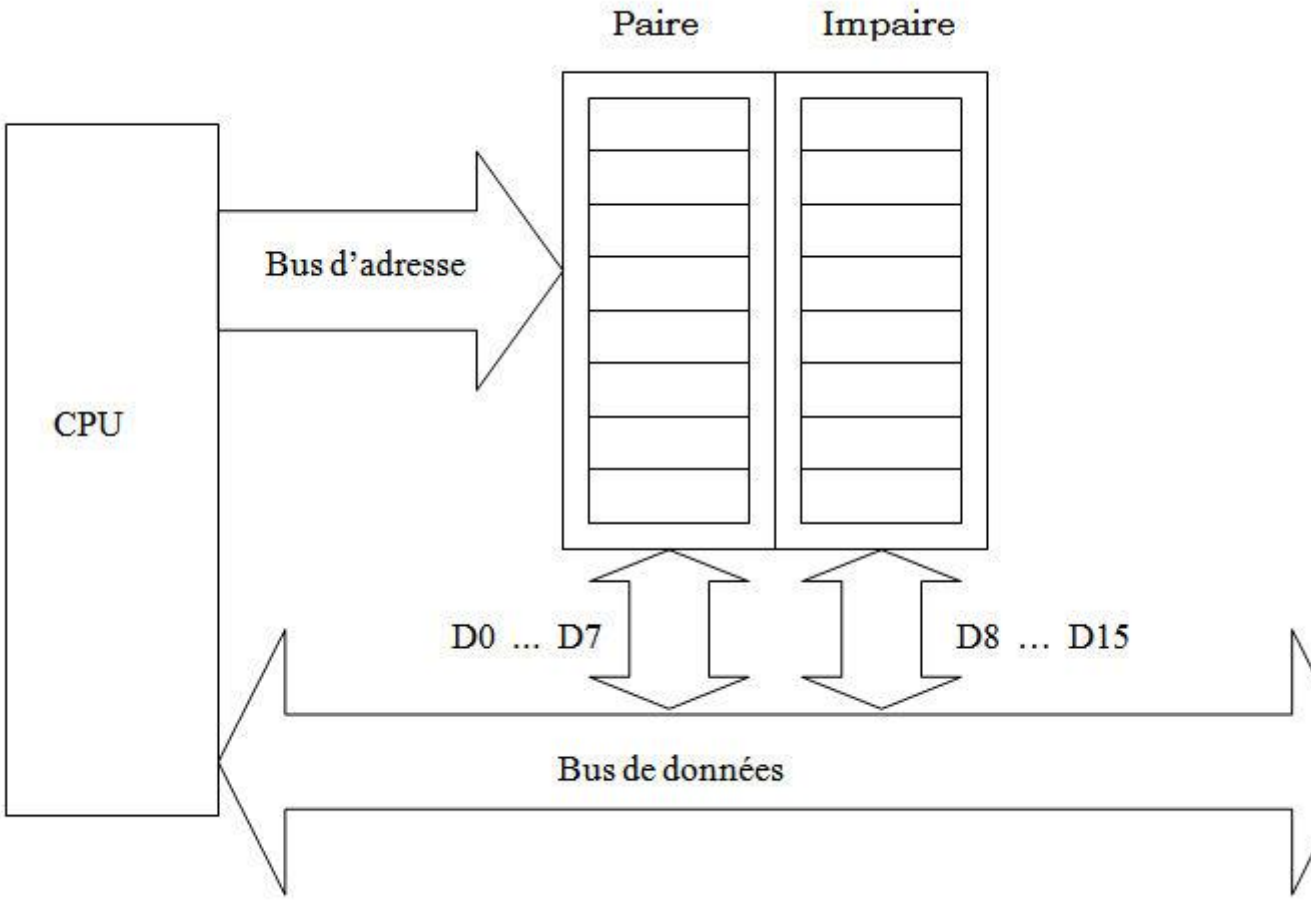
V-4-1) Organisation logique :

Logiquement la mémoire est organisée de cette manière :



V-4-2) Organisation physique :

Le microprocesseur 8086 est processeur 16 bits (bus de données de 16 bits), ce qui donne la possibilité à ce microprocesseur d'accéder en même temps à deux cases mémoires de 8 bits. En effet pour le 8086 la mémoire est organisée en deux Banks (un bank pair et un bank impair chacun de 512 KOctet) comme le montre la figure suivante :

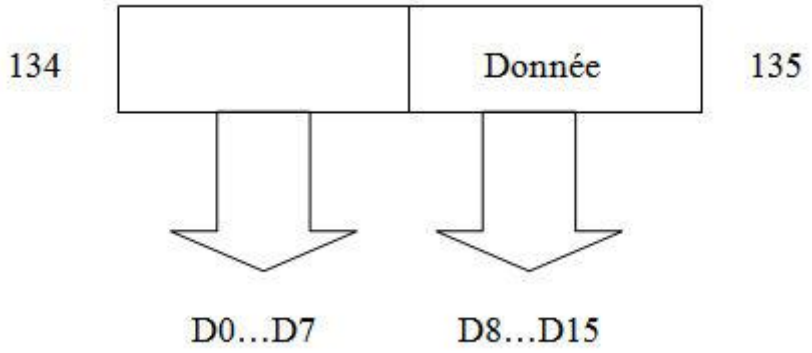


Les bits D0..D7 sont appelé partie base alors que les bits D8...D15 sont appelés partie haute. Le 8086 peut charger un octet (8 bits) ou un mot (16 bits) ou un double mot (32 bits) de la mémoire, en effet pour l'octet il suffit de donner l'adresse de ce dernier pour être chargé dans la CPU, pour le mot il suffit de donner l'adresse le 8086 cherche l'octet du poids faible à l'adresse donnée et l'octet du poids le plus fort à l'adresse qui suit , mais un problème apparaît lorsque on veut accéder à une case mémoire impaire tel que 135 par exemple , en effet :

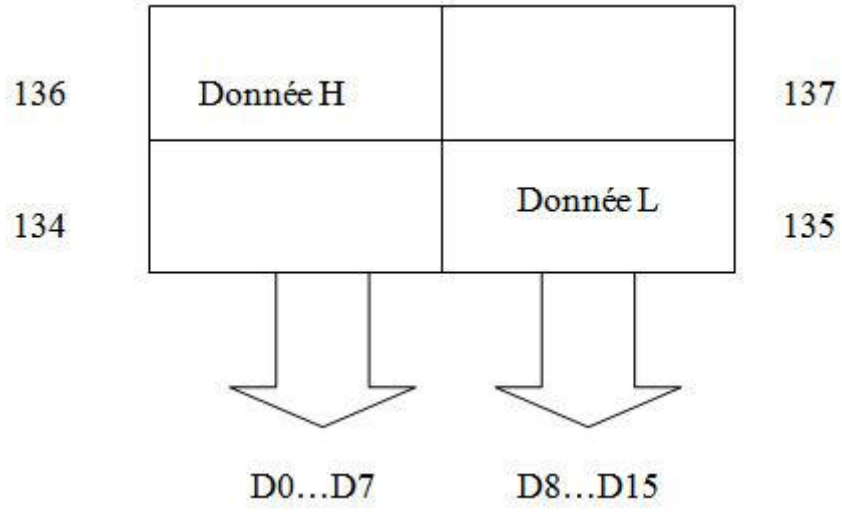
La figure suivante montre comment les cases sont rangées dans les deux banks :

Paire	Impaire
6	7
4	5
2	3
0	1

- Si on veut accéder par exemple à l'octet (8 bits) d'une adresse paire celle-ci sera directement transmise sur les lignes D0...D7 mais si on veut accéder à une adresse impaire tel que 135 par exemple, donc il faut aussi que la donnée serait charger sur les lignes D0.. D7, or en réalité et en regardant l'organisation de la mémoire e par la figure précédente on constate que la donnée sera transmise sur les lignes D8..D15 (adresse impaire) ce qui va obliger le microprocesseur de changer cette octet du poids haut au poids faible d'une manière automatique.



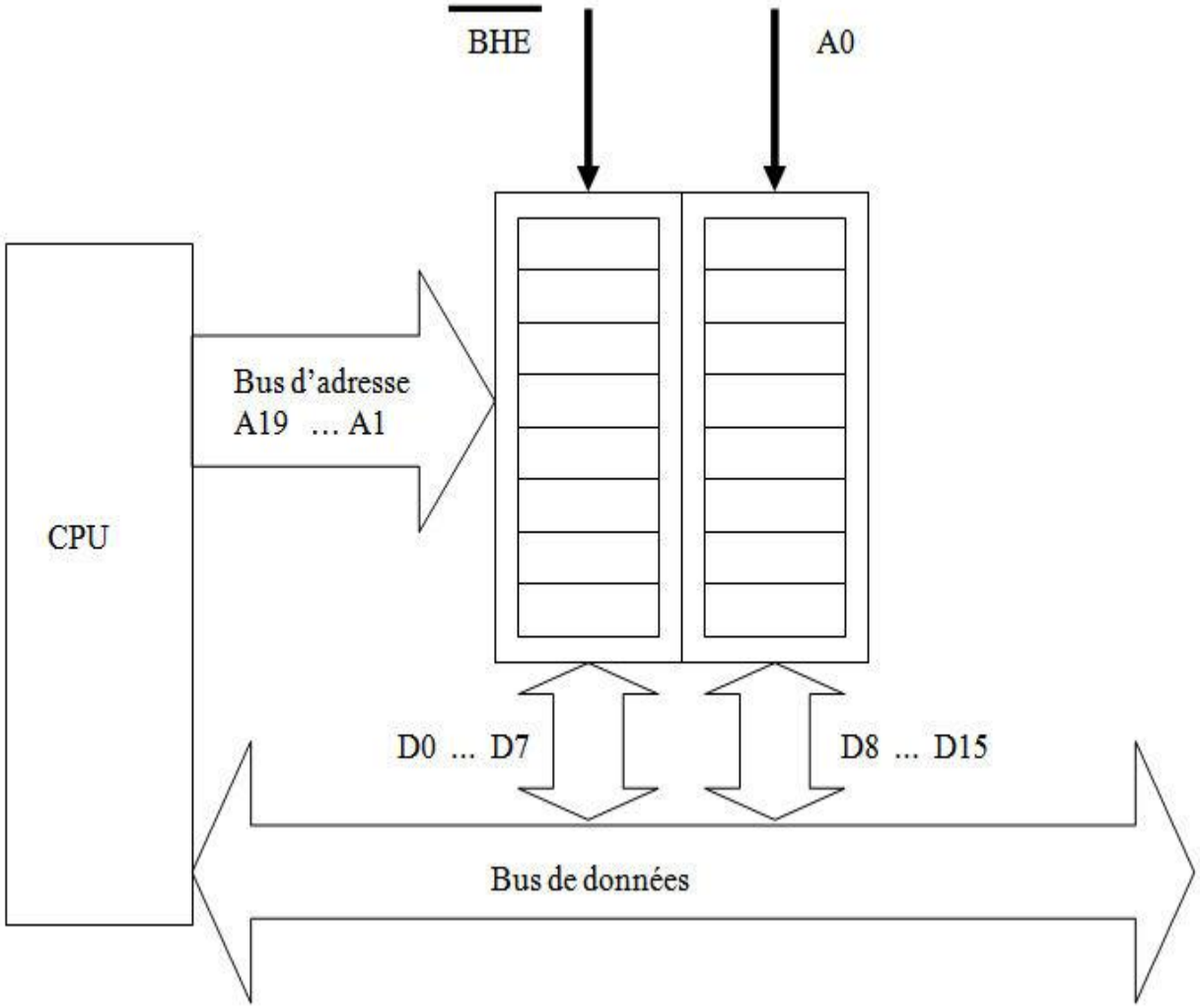
- Si on veut accéder à un mot : Si l'adresse est paire on n'aura pas de problème le pois faible sera chargé sur les lignes D0...D7 et le pois fort sera chargé sur les lignes D8...D15 donc l'accès à la mémoire se fait avec un seul cycle .mais si on veut accéder à un mot logé dans une case impaire tel que 135 par exemple alors il nous faut deux cycles pour charger la donnée en effet l'organisation de la mémoire nous donne la disposition suivante :



Donc le microprocesseur doit accéder à la mémoire en deux temps le premier pour chercher l'octet 135 et le deuxième pour chercher l'octet haut à partir de l'adresse 136.de plus il doit permuter ces deux octet pour avoir le poids faible sur les lignes D0...D7 et le poids fort sur les lignes D8...D15.

Remarque :

Pour sélectionner les banks pair et impair le microprocesseur utilise deux signaux (BHE et A0 : le premier bit du bus d'adresse) comme le montre la figure suivante :



Pour sélectionner le bank pair A0=0

Pour sélectionner le bank impair BHE = 0

Les interruptions du microprocesseur

I) Introduction :

Un système à base de microprocesseur peut comporter plusieurs périphériques tels que : un Ecran, une souris, une imprimante, un disque dure, un CNA (convertisseur numérique analogique), un CAN etc. ..

Pour dialoguer avec ces périphériques le microprocesseur a trois façons de communiquer avec ces derniers :

- En questionnant de façon continue le périphérique pour vérifier que des données peuvent être lues ou écrites (Polling).

- En l'interrompant lorsqu'un périphérique est prêt à lire ou écrire des données (interruption)

- En établissant une communication directe entre deux périphériques (DMA : Direct memory acces).

I-1/ Le polling :

Bien que le polling soit une façon simple d'accéder à un périphérique, le débit des données dans un périphérique est parfois beaucoup plus lent que la vitesse de fonctionnement d'un processeur, et le polling peut donc être très inefficace. On lui préfère en général la méthode par interruptions.

I-2) Le DMA :

En cas de transfert de données de grande quantité entre deux périphériques, il peut être plus efficace de laisser communiquer entre eux les deux périphériques plutôt que de solliciter le processeur. Ce type de communication avec les entrées/sorties s'appelle gestion de transfert par DMA, Le processeur indique au contrôleur de DMA quels sont les périphériques qui doivent communiquer, le nombre et éventuellement l'adresse des données à transférer, puis il initie le transfert. Le processeur n'est pas sollicité durant le transfert, et une interruption signale au processeur la fin du transfert.

I-3) L'interruption :

Il arrive fréquemment qu'un périphérique ou un programme nécessite une intervention du microprocesseur à cet effet il est possible de l'interrompre quelques instants, celui-ci va effectuer l'instruction demandée, est exécuter, une fois qu'elle rend le travail initial.

Ce mode d'échange est principalement basé sur l'initiative du périphérique. Grâce à ce mécanisme d'interruption, le dispositif périphérique prend l'initiative de l'échange.

Une interruption est un évènement qui provoque l'arrêt du programme en cours et provoque le branchement du microprocesseur à un sous- programme particulier dit de "traitement de l'interruption".

Remarque 1 : (ordre de priorité)

Lorsqu'un périphérique reçoit des données pour le processeur, il envoie au processeur un signal d'interruption. Si celui-ci peut être interrompu (à condition qu'il ne soit pas en train de communiquer avec un périphérique de plus haute priorité) , il stoppe la tâche en cours , sauve son état (ces registres) en mémoire (la zone pile) et appelle la routine correspondant au numéro d'interruption .

Remarque 2 : (plusieurs interruptions) :

Le fonctionnement des périphériques se fait en général d'une manière asynchrone, donc plusieurs interruptions peuvent être déclenchées en même temps, il existe souvent un contrôleur d'interruptions destiné à gérer les conflits entre interruptions

Le jeu d'instructions du 8086/8088

I) introduction :

On peut diviser les instructions du 8086/88 en 6 groupes comme suit :

- Instructions de transfert de données.
- Instructions arithmétiques.
- Instructions de bits (logiques).
- Instructions de sauts de programme.
- Instructions de chaîne de caractères.
- Instructions de contrôle de processus.
- Instructions d'interruptions.

II) Les instructions de transfert de données :

Ils sont divisés en 4 sous- groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Général	MOV	Transfert d'octets ou de mots
	PUSH	Chargement de la pile
	POP	Déchargement de la pile
	PUSHA	Chargement de tous les registres dans la
	POPA	pile Déchargement de tous les registres
	XCHG	dans la pile Echange d'octet ou de mot
	XLAT	Translation d'octet
Entrées-sorties	IN	Entrée de mot ou d'octet
	OUT	Sortie de mot ou d'octet
Adresses	LEA	Chargement de l'adresse effective
	LDS	Chargement du pointeur avec DS
	LES	Chargement du pointeur avec ES
Indicateurs	LAHF	Transfert des indicateurs dans AH
	SAHF	Rangement de AH dans les indicateurs
	PUSHP	Chargement des indicateurs dans la pile
	POPF	Déchargement des indicateurs de la pile.

II-1) Les instructions d'usage général :

II-1-1) MOV :

Elle permet de transférer les données (un octet ou un mot) d'un registre à un autre registre ou d'un registre à une case mémoire, sa syntaxe est comme suit :

Exemples :

```
MOV destination, source
MOV   AX, BX     ;  Transfert d'un registre de 16 bits vers un registre de
16 Bits
MOV   AH, CL     ;  Transfert d'un registre de 8 bits vers un registre de 8
bits
MOV   AX, Val1   ; Transfert du contenu d'une case mémoire 16 bits vers AX
MOV   Val2, AL   ; Transfert du contenu du AL vers une case mémoire  D'adresse
Val2
```

Remarques :

- Il est strictement interdit de transférer le contenu d'une case mémoire vers une autre case mémoire comme suit

```
MOV    Val1, Val2
```

Pour remédier à se problème on va effectuer cette opération sur deux étapes :

```
MOV AL, Val2
MOV Val1, AL
```

- On n'a pas le droit aussi de transférer un registre segment vers un autre registre segment sans passer par un autre registre :

```
MOV   DS, ES
```

On va passer comme la première instruction :

```
MOV   AX, ES
MOV   DS, DS
```

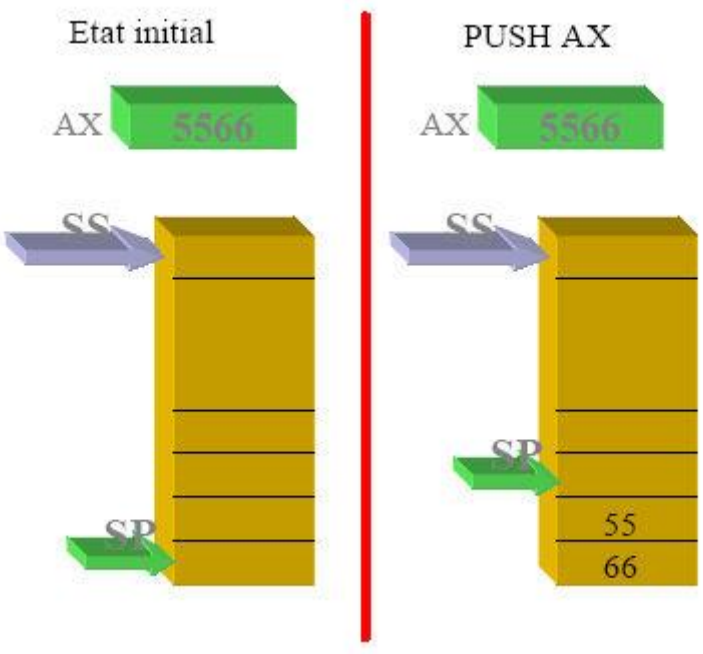
- Le CS n'est jamais utilisé comme registre destination.

II-1-2) PUSH :

Elle permet d'empiler les registres du CPU sur le haut de la pile

```
Syntaxe : PUSH  SOURCE
```

Exemple :

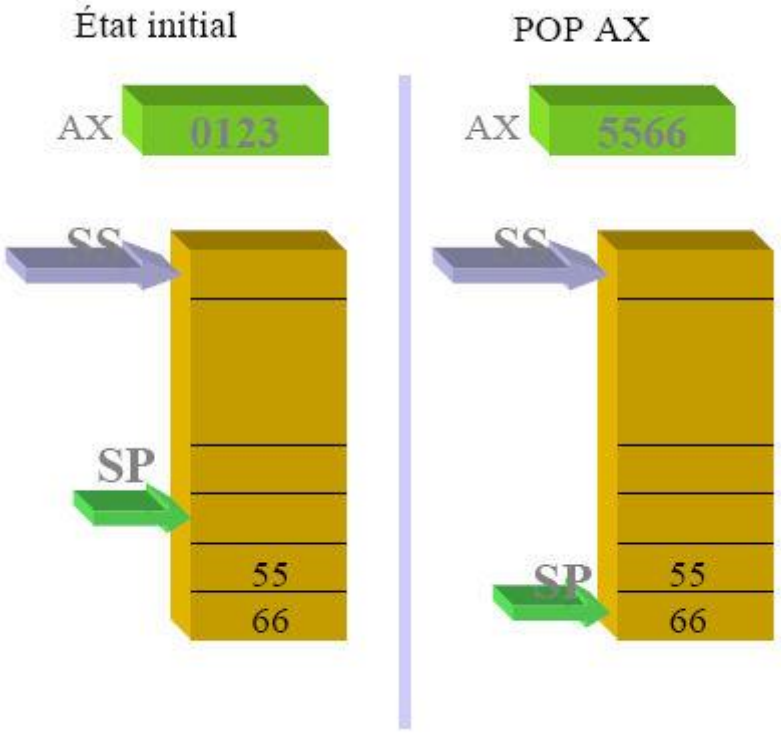


II-1-3) POP :

Elle permet de dépiler les registres du CPU sur le haut de la pile

```
Syntaxe : POP  destination
```

Exemple :



II-1-4) PUSHA :

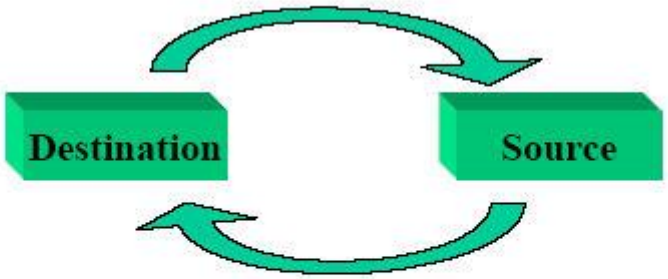
Cette instruction permet d'empiler la totalité des registres internes du microprocesseur sur la pile.

II-1-5) POPA :

Cette instruction permet de dépiler la totalité des registres internes du microprocesseur sur la pile.

II-1-6) XCHG :

Elle permet de commuter la source avec la destination comme suit :



XCHG AX,BX ; elle permute entre AX et BX
Exemple AX=0123 et BX = 5678
Après l'exécution de l'instruction on aura :
AX=5678 et BX = 0123

II-1-7) XLAT :

Cette instruction est utilisée pour convertir des données d'un code à un autre, en effet elle permet de placer dans l'accumulateur AL le contenu de la case mémoire adressée en adressage base+décalage (8 bits), la base étant le registre BX et le décalage étant AL lui même dans le segment DS

```
(AL)<--------[ (BX) + (AL) ]
```

Syntaxe : XLAT tab_source

Exemple :

conversion du code binaire 4 bits en un digit hexa codé en ASCII

```
Tab  db     '0123456789ABCDEF'
MOV AL,1110B        ; chargement de la valeur à convertir (07) MOV BX, OFFSET
TAB
                    ; pointé sur le tableau
XLAT                ; Al est chargé par le code ASCII de 'E'
```

II - 2) les instructions d'entrées-sorties :

II-2-1) IN) OUT:

Elle permet de récupérer des données d'un port (donc de la périphérie) ou restituer des données à un port, dans les deux cas s'il s'agit d'envoyer ou de recevoir un octet on utilise l'accumulateur AL, s'il s'agit d'envoyer ou de recevoir un mot on utilise l'accumulateur AX.

Syntaxe :

```
IN ACCUMULATEUR, DX
OUT DX, ACCUMULATEUR
```

Remarque :

DX : contient l'adresse du port.

ACCUMULATEUR : contient la donnée (à recevoir ou à emmètre).

II-3 / Les instructions de transfert d'adresses :

II-3-1) LEA (Load Effective Address):

Elle transfert l'adresse offset (décalage) d'une opérande mémoire dans un registre de 16 bits (pointeur ou index). Cette commande a le même rôle que l'instruction MOV avec offset mais elle est plus puissante car on peut utiliser avec elle toute technique d'adressage.

Exemple :

LEA BX, TAB_VAL (c'est équivalent à MOV BX, offset TAB_VAL)

II-3-2) LDS) LES :

Cette instruction permet de charger le segment et l'offset d'une adresse

Exemple :

Au lieu de faire :

```
MOV   BX, offset tab_val
MOV   AX , Seg tab_val
MOV  DS , AX
```

On remplace ces trois instructions par une seule :

```
LDS BX , tab_val ;elle   charge   automatiquement l'offset de tab_val dans
le registre BX
                 ;et le segment dans le registre DS .
```

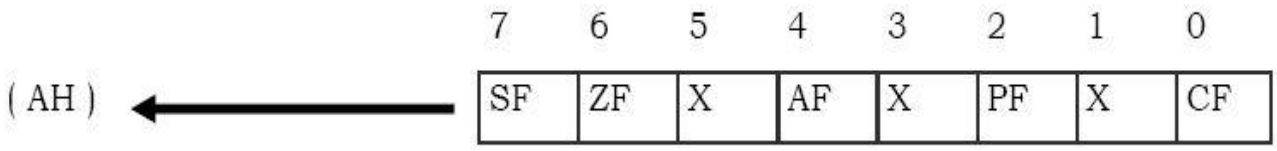
Remarque :

Pour l'instruction LES : le segment est ES.

II-4) Les instructions d'indicateur :

II-4-1) LAHF) SAHF :

LAHF : Load AH from Flags : place l'octet de poids faible du registre d'état (FLAGS) dans le registre AH comme suit :



SAHF : Store AH into Flags : Place le contenu de AH dans l'octet de poids faible du registre d'état (FLAGS).

II-4-2) PUSHF) POPF:

PUSHF : Permet d'empiler la totalité du registre d'état (FLAGS)

POPF : Permet de dépiler le registre d'état (FLAGS).

III) Instructions arithmétiques :

Les instructions arithmétiques peuvent manipuler quatre types de nombres :

- Les nombres binaires non signés
- Les nombres binaires signés.
- Les nombres décimaux codés binaires (DCB), non signés.
- Les nombres DCB non condensés, non signés.

Les instructions arithmétiques sont divisées en quatre sous-groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Addition	ADD	Addition sur un octet ou un mot
	ADC	Addition sur un octet ou un mot avec retenue
	INC	Incrémentation de 1
	AAA	Ajustement ASCII
	DAA	Ajustement décimal
Soustraction	SUB	Soustraction sur un octet ou un mot
	SBB	Soustraction sur un octet (mot) avec retenue
	DEC	Décrémentation de 1
	NEG	Mètre un octet ou un mot en négatif
	CMP	Comparaison d'octet ou mot
	AAS	Ajustement ASCII
	DAS	Ajustement décimal
Multiplication	MUL	Multiplication d'octet ou de mot **non signée**
	IMUL	Multiplication d'octet ou de mot **signée**
	AAM	Ajustement ASCII
Division	DIV	Division d'octet ou de mot **non signée**
	IDIV	Division d'octet ou de mot **signée**
	AAD	Ajustement ASCII
	CBW	Conversion d'un octet en un mot
	CWD	Conversion d'un mot en double mots

III-1) Addition :

III-1-1) ADD: (Addition)

```
Syntaxe :    ADD   Destination, source
```

Elle permet d'additionner le contenu de la source (octet ou un mot) avec celui de la destination le résultat est mis dans la destination

```
  Destination <---------- Destination  +  source
```

Exemples :

```
ADD AX, BX  ; AX = AX + BX (addition sur 16 bits) ADD  AL,BH
            ; AL = AL + BH (addition sur 8 bits )
ADD AL, [SI]  ; AL = AL + le contenu de la case mémoire
```

```
              ; pointé par SI
ADD [DI], AL ; le contenu de la case mémoire pointé par DI
             ; Est additionnée avec AL, le résultat est mis
             ; dans la case mémoire pointé par DI
```

Remarques :

- Ici on a presque les mêmes restrictions de l'instruction MOV c.a.d on n'a pas le droit d'additionner deux cases mémoires sans utiliser un registre de données.

Exemples :

```
ADD Tab1 , Tab2
```

sera remplacé par :

```
MOV AX , Tab2
ADD Tab1 , AX
```

- De même une valeur immédiate ne peut être une destination.

III-1-2) ADC : (Addition avec retenue)

```
Syntaxe :    ADC   Destination, source
```

Elle permet d'additionner le contenu de la source (octet ou un mot) avec celui de la destination et la retenue (CF) le résultat est mis dans la destination

```
  Destination  <------------   Destination  +  source + retenue
```

Exemples :

```
ADC AX,BX   ; AX = AX + BX + CF(addition sur 16 bits )
ADC AL,BH   ; AL = AL + BH + CF(addition sur 8 bits )
ADC AL,[SI] ; AL = AL + le contenu de la case mémoire pointé par SI + CF
ADC [DI],AL ; le contenu de la case mémoire pointé par DI
            ; est additionné avec AL + CF , le résultat est
            ; mis dans  la case mémoire pointé par DI
```

Remarque :

Les restrictions de l'instruction ADD sont valables pour l'instruction ADC.

III-1-3) INC : (Incrémentation)

```
Syntaxe :    INC    Destination
```

Elle permet d'incrémenter le contenu de la destination

```
  Destination <----------  Destination  +  1
```

Exemples :

```
INC AX     ; AX = AX +  1 (incrémentation sur 16  bits).
INC AL     ; AL = AL +1 (incrémentation sur 8 bits).
INC [SI]   ; [SI] = [SI] + 1 le contenu de la case mémoire pointé  par SI
sera incrémenter
```

Remarque :

On ne peut pas incrémenter une valeur immédiate.

III-1-4) AAA) DAA : (ASCII) DECIMAL Adjust for Addition)

L'addition de deux nombres BCD génére parfois un résultat qui n'est pas un nombre en BCD d'ou il faut faire des corrections sur ces nombres pour avoir un résultat cohérent. Cette instruction examine le quarte bas de AL et vérifie s'il est conforme ou non :

- Si oui (elle met AF et CF à zéro pour information) efface la quarter haut de AL

- Si non :

  - Elle ajoute 6 à AL
  - Ajoute 1 à AH
  - Efface le quarter haut de AL
  - Met AF et CF à 1 (pour information)

Exemples : on veut faire l'addition en BCD de 73 + 88

73		0111 0011
	+	
88		1000 1000
163		1111 1011

On remarque que ni 1111 ni 1011 est un nombre BCD donc on va ajouter 6

au premier quarte d'ou l'opération devient :

	1111 1011		
+	0110		
	=	10000 0001	
	+	0110	
	=	0110 0001	(63)

L'octet le plus haut :

```
      00000001
+     00000000
-------------------
=     00000001          (01)
```

d'ou les résultat 163

III-2) Soustraction :

III-2-1) SUB : (Soustraction)

```
Syntaxe :    SUB   Destination, source
```

Elle permet de soustraire la destination de la source (octet ou un mot) le résultat est mis dans la destination

```
  Destination <---------  Destination  --  source
```

Exemples :

```
SUB AX,BX    ; AX = AX - BX (Soustraction sur 16 bits )
SUB  AL,BH   ; AL = AL - BH ( Soustraction sur 8 bits )
SUB AL,[SI]  ; AL = AL - le contenu de la case mémoire pointé par SI
SUB [DI],AL  ; le contenu de la case mémoire pointé par DI
             ; est soustraite de AL , le résultat est mis
             ; dans la case mémoire pointé par DI
```

Remarques :

- On a les mêmes restrictions de l'instruction ADD.

III-2-2) SBB : (Soustraction avec retenue)

```
Syntaxe :    SBB   Destination, source
```

Elle permet de soustraire la destination de la source et la retenue (octet ou un mot) le résultat est mis dans la destination

```
 Destination <--------   Destination  --  source -- retenue
```

Exemples :

```
SBB AX,BX    ; AX = AX-BX - CF (Soustraction sur 16 bits )
SBB  AL,BH   ; AL = AL - BH - CF( Soustraction sur 8 bits )
SBB AL,[SI]  ; AL = AL - le contenu de la case mémoire
             ; pointé  par SI - CF
SBB [DI],AL  ; le contenu de la case mémoire pointé par DI
             ; est  soustraite avec AL - CF, le résultat est
             ; mis dans  la case mémoire pointé par DI
```

Remarques :

- On a les mêmes restrictions de l'instruction ADD.

III-2-3) DEC : (Décrémentation)

```
Syntaxe :    DEC   Destination
```

Elle permet de décrémenter le contenu de la destination

```
  Destination   <----------- Destination  -  1
```

Exemples :

```
DEC AX    ; AX = AX -  1 (décrémentation sur 16  bits).
DEC  AL   ; AL = AL -1 (décrémentation sur 8 bits).
DEC [SI]  ; [SI] = [SI] - 1 le contenu de la case mémoire
          ; pointé  par SI sera décrémenter
```

Remarque :

On ne peut pas décrémenter une valeur immédiate.

III-2-4) NEG : (Négatif)

```
Syntaxe :    NEG   Destination
```

Elle soustrait l'opérande destination (octet ou mot) de 0 le résultat est stocker dans la destination, donc avec cette opération on réalise le complément à deux d'un nombre

```
  Destination  <----------- 0 - Destination
```

Exemples :

```
NEC AX           ; AX = 0 - AX
NEC AL           ; AL = 0 - AL
NEC [SI]         ; [SI] = 0 - [SI]
```

Remarque :

Les indicateurs affectés par cette opération sont : AF, CF, OF, PF, SF, ZF

III-2-5) CMP : (Comparaison)

```
Syntaxe :    CMP  Destination   , Source
```

Elle soustrait la source de la destination , qui peut être un octet ou un mot , le résultat n'est pas mis dans la destination , en effet cette instruction touche uniquement les indicateurs pour être tester avec une autre instruction ultérieure de saut conditionnel

Les indicateurs susceptibles d'être touché sont : AF, CF, OF, PF, SF, ZF

Donc cette instruction va nous permettre de comparer deux nombres comme le montre le tableau suivant :

	Opérande non signé				Opérande signé			
	OF	SF	ZF	CF	OF	SF	ZF	CF
Source < destination	-	-	0	0	0/1	0	0	-
Source = destination	-	-	1	0	0	0	1	-
Source > destination	-	-	0	1	0/1	1	0	-

III-2-6) AAS) DAS: (ASCII) DECIMAL Adjust for Substraction)

Elle est identique à l'instruction AAA/DAA mais l'ajustement se fait en BCD pour la soustraction.

III-3) La multiplication :

III-3-1) MUL : (Multiplication pour les nombres non signés)

MUL effectue une multiplication non signée de l 'opérande source avec l'accumulateur :

```
Syntaxe : MUL   Source
```

- -Si la source est un octet alors elle sera multipliée par l'accumulateur AL le résultat sur 16 bits sera stocké dans le registre AX.
- Si la source est un mot alors elle sera multipliée avec l'accumulateur AX le résultat de 32 bits sera stocké dans la paire des registres AX et DX

Remarque :

Cette multiplication traite les données en tant que nombres non signés

Donc on aura :

```
(AX)       <-------------   (AL) X  Source (octet)
(AX)(DX)   <------------    (AX) X Source (mots)
```

En conclusion :

Multiplication	Opérande 1	Opérande 2	Résultat
Octet x Octet	AL	Registre ou memoire	AX
Mots x Mots	AX	Registre ou memoire	DX AX
Mots x Octet	AL= Octet, AH=0	Registre ou memoire	DX AX

III-3-2) IMUL : (Multiplication pour les nombres signés)

MUL effectue une multiplication signée de l 'opérande source avec l'accumulateur :

```
Syntaxe :    IMUL   Source
```

- Si la source est un octet alors elle sera multiplieéepar l'accumulateur
- AL le résultat sur 16 bits sera stocké dans le registre AX .
- Si la source est un mot alors elle sera multiplié avec l'accumulateur AX le résultat de 32 bits sera stocké dans la paire des registres AX et DX

Remarque :

Cette multiplication traite les données en tant que nombres signés

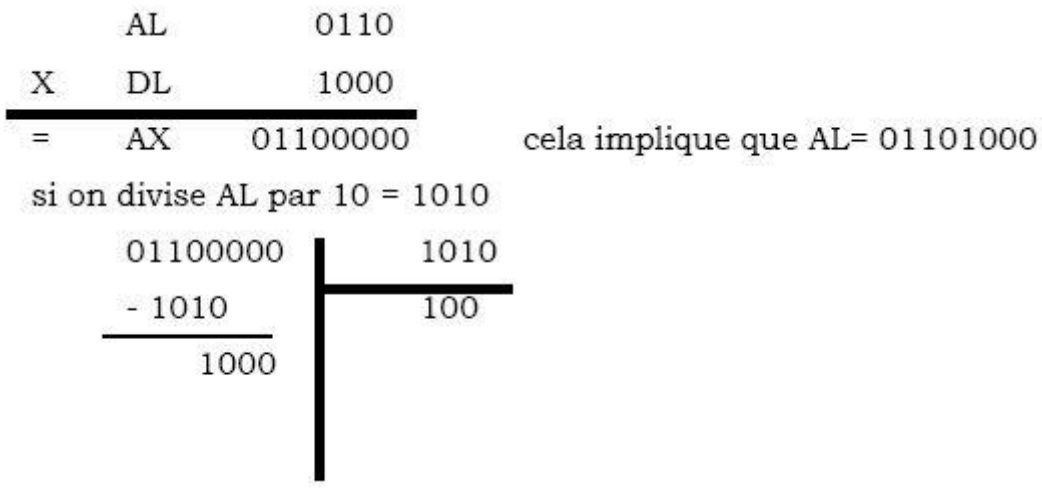
III-3-3) AAM: (ASCII Adjust for Multiplication)

Comme AAA et AAS cette instruction va nous permettre de corriger le résultat d'une multiplication de deux nombres en BCD, pour corriger le résultat de l'instruction AAM divise AL par 10.

Exemple :

```
MOV AL , 6
MOV DL , 8
MUL DL
AAM
```

Si on décortique cette instruction on aura :



III-4) La division :

III-4-1) DIV : (Division des nombres non signés)

```
Syntaxe :   DIV  Source
```

Elle effectue une division non signée de l'accumulateur par l'opérande source :

Exemples :

- Si l'opérande est un octet : alors on récupère le quotient dans le registre AL et le reste dans le registre AH.
- Si l'opérande est un mot : alors on récupère le quotient dans le registre AX et le reste dans le registre DX

A/

```
MOV AH,00h
MOV AL,33H
MOV DL,25H
DIV DL    ; Cela implique que AH=    et   AL =
```

B/

```
MOV AX,500H
MOV CX,200H
DIV CX     ;Cela implique que    AX=         et   AL =
```

III-4-2) IDIV : (Division des nombres signés)

```
Syntaxe :    IDIV  Source
```

Elle effectue une division signée de l'accumulateur par l'opérande source :

- Si l'opérande est un octet : alors on récupère le quotient dans le registre AL et le reste dans le registre AH.
- Si l'opérande est un mot : alors on récupère le quotient dans le registre AX et le reste dans le registre DX

Exemples:

A/

```
MOV  AH,00h
MOV  AL,-33H
MOV DL,25H
IDIV  DL  ;   Cela implique que AH=   et   AL =
```

B/

```
MOV AX,-500H
MOV CX,200H
IDIV  CX   ;  Cela implique que AX=   et   AL =
```

III-4-3) AAD : (ASCII Adjust for Division)

Pour corriger le résultat elle va multiplier le contenu de AH par 10 et l'ajoute à celui de AL

Remarque :

Pour cette instruction il faut faire l'ajustement avant l'instruction de division.

III-4-4) CBW (convert byte to word)

Cette instruction permet de doubler la taille de l'opérande signé

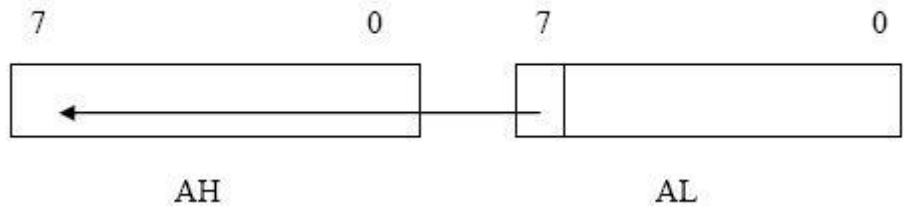
```
  Octet ------------>  Mots
```

Remarque :

CBW reproduit le bit 7 (bits de signe) de AL dans AH jusqu'à remplissage de ce dernier.

Exemple :

```
MOV AL, +96        ; AL=0110 0000
CBW                ; AH=0000 0000 et AL=0110 0000
```



III-4-5) CWD (convert Word to Double)

Cette instruction permet de doubler la taille de l'opérande signé

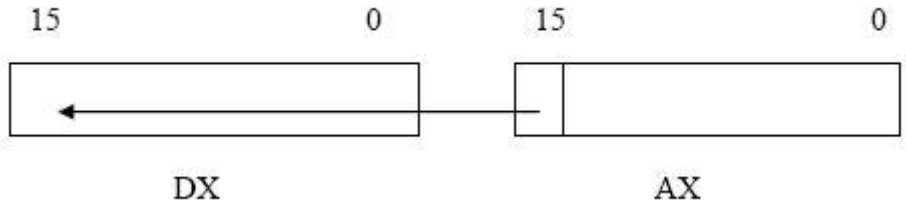
```
  Word    -------- >    Double
```

Remarque :

CWD reproduit le bit 15 (bits de signe) de AX dans DX jusqu'à remplissage de ce dernier.

Exemple :

```
MOV  AX, +260     ; AX=0000 0001 0000 0100
CWD               ; DX=0000H, AX=0104H
```



IV) Les instructions logiques (de bits) :

Ils sont divisés en trois sous-groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Logique	NOT	Inversion logique sur un octet ou un mot
	AND	Et logique
	OR	Ou logique
	XOR	Ou exclusif
	TEST	Et logique sans résultat, affecte uniquement les indicateurs du registre des flags.
Décalages	SHL	Décalage logique à gauche
	SAL	Décalage arithmétique à gauche
	SHR	Décalage logique à droite
	SAR	Décalage arithmétique à droite
Rotation	ROL	Rotation à gauche
	ROR	Rotation à droite
	RCL	Rotation à gauche à travers le bit de retenue
	RCR	Rotation à droite à travers le bit de retenue

IV-1) Les instructions logiques :

IV-1-1) NOT : (Négation)

Elle réalise la complémentation à 1 d'un nombre

```
Syntaxe :       NOT   Destination
```

Exemple :

```
MOV AX, 500        ; AX = 0000 0101 0000 0000
NOT AX             ; AX = 1111 1010 1111 1111
```

IV-1-2) AND : (Et logique)

```
Syntaxe :    AND   Destination, source
```

Elle permet de faire un ET logique entre la destination et la source (octet ou un mot) le résultat est mis dans la destination

```
  Destination    <----------   Destination  .  source
```

Exemples :

```
MOV AX , 503H      ; AX = 0000 0101 0000 0011
AND AX , 0201H     ;       0000 0101 0000 0011
                   ;  AND  0000 0010 0000 0001
                   ;     = 0000 0000 0000 0001
AND AX,BX          ; AX = AX . BX (Et logique entre AX et BX)
AND  AL,BH         ; AL = AL . BH (ET logique sur 8 bits)
AND AL,[SI]        ; AL = AL AND le contenu de la case mémoire
                   ; pointé  par SI
AND [DI],AL        ; ET logique entre la case mémoire pointé par
                   ; DI et  AL , le résultat est mis dans la case
                   ; mémoire pointé par DI
```

IV-1-3) OR : (OU logique)

```
Syntaxe :    OR     Destination, source
```

Elle permet de faire un OU logique entre la destination et la source (octet ou un mot) le résultat est mis dans la destination

```
   Destination   <----------- Destination  +  source
```

Exemples :

```
MOV AX , 503H        ; AX = 0000 0101 0000 0011
OR AX , 0201H        ;          0000 0101 0000 0011
                     ;    OR    0000 0010 0000 0001
                     ;    =     0000 0111 0000 0011
OR AX,BX             ; AX = AX + BX ( OU logique entre AX et BX )
OR  AL,BH            ; AL = AL + BH ( OU logique sur 8 bits )
OR AL,[SI]           ; AL = AL OU le contenu de la case mémoire
                     ; pointé  par SI
OR [DI],AL           ; OR logique entre la case mémoire pointé par
                     ; DI et  AL, le résultat est mis dans la case
                     ; mémoire pointé par DI
```

IV-1-4) XOR : (OU exclusif)

```
Syntaxe :    XOR  Destination, source
```

Elle permet de faire un OU exclusif logique entre la destination et la source (octet ou un mot) le résultat est mis dans la destination

```
  Destination   <------------ Destination  +  source
```

Exemples :

```
MOV AX , 503H   ; AX = 0000 0101 0000 0011
XOR AX , 0201H  ;          0000 0101 0000 0011
                ;   XOR    0000 0010 0000 0001
```

```
                ;     =     0000 0011 0000 0010
XOR AX,BX       ; AX = AX + BX (OU exclusif  entre AX et BX )
XOR  AL,BH      ; AL = AL + BH ( OU exclusif sur 8 bits )
XOR AL,[SI]     ; AL = AL OU exclusif le contenu de la case
                ; Mémoire pointé  par SI
XOR [DI],AL     ; XOR logique entre la case mémoire pointé par
                ; DI et  AL, le résultat est mis dans la case
                ; mémoire pointé par DI
```

IV-1-5) TEST :

```
Syntaxe :    TEST   Destination, source
```

Elle permet de faire un ET logique entre la destination et la source (octet ou un mot) mais la destination ne sera pas touchée en effet cette instruction ne touche que les indicateurs.

Exemples :

```
MOV AX, 503H    ; AX = 0000 0101 0000 0011
TEST AX, 0201H  ;  0000 0101 0000 0011
```

Elle va effectuer un ET logique entre le premier nombre et le second sans toucher les deux mais elle va affecter uniquement les indicateurs (Flags)

IV-2) Les instructions de décalages :

SHL : décalage logique à gauche :

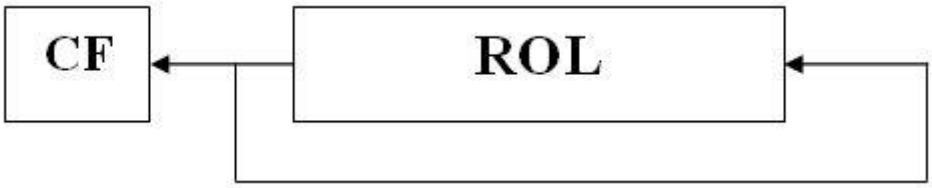
SHR : décalage logique à droite :

SAL : décalage arithmétique à gauche :

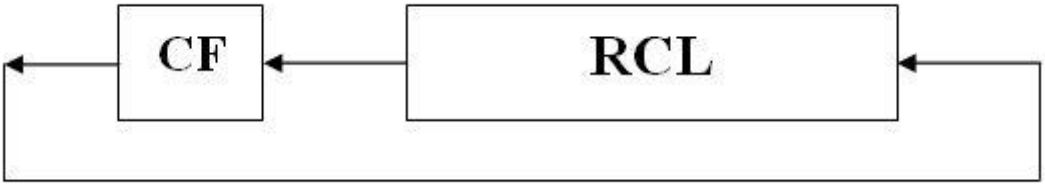
SAR : décalage arithmétique à gauche :

IV-3) Les instructions de rotations :

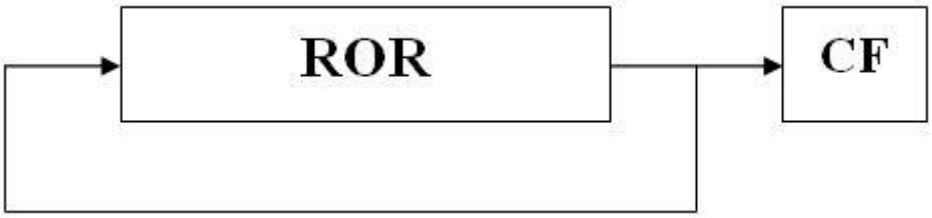
ROL : Rotation à gauche :



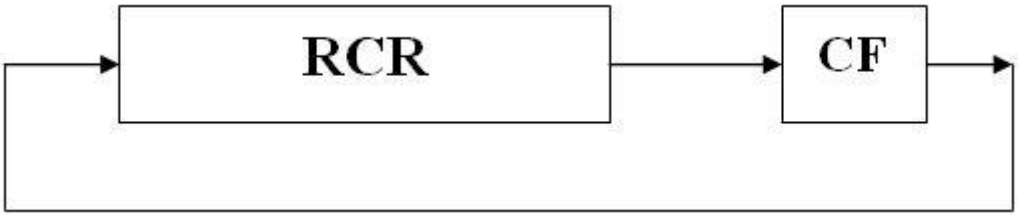
RCL : Rotation a travers la retenue à gauche :



ROR : Rotation à droite :



RCR : Rotation à travers la retenue à droite :



Syntaxe des instructions de rotation et de décalage :

```
ROR  destination, compteur
```

Exemple :

```
ROR AX,1
ROL AL,1
```

Si on veut faire quatre rotations de suite on a deux solutions :

```
ROR AL,1
ROR AL,1
ROR AL,1
ROR AL,1
```

Ou encore :

```
MOV CL,4
ROR AL,CL
```

Remarque :

Les instructions de rotations et de décalages logiques ne tiennent pas compte du bit de signe donc elles travaillent avec les nombres non signés.

Les instructions de rotations et de décalages arithmétiques préservent le bit de signe donc elles sont réservées aux nombres signés.

V) Instructions de sauts de programme :

Elles permettent de faire des sauts dans l'exécution d'un programme (rupture de séquence)

Remarque :

Ces instructions n'affectent pas les Flags. Dans cette catégorie on trouve toutes les instructions de branchement, de boucle et d'interruption après un branchement, le tableau suivant donne ces instructions :

Type	Nom	Fonction
Branchements inconditionnels	CALL	Appel à un sous programme
	RET	Retour d'un sous programme
	JMP	Saut
Branchements conditionnels (arithmétique non signée)	JA/JNBE	Si supérieur / Si non inférieur ou non égal
	JAE/JNB	Si supérieur ou égal/ Si non inférieur
	JB/JNAE	Si inférieur/si non supérieur ni égal
	JBE/JNA	Si inférieur ou égal/si non supérieur.
Branchements conditionnels (arithmétique signée)	JG/JNLE	Si plus grand/si pas inférieur ni égal
	JGE/JNL	Si plus grand ou égal/Si pas inférieur
	JL/JNGE	Si moins que/Si pas plus grand ni égal
	JLE/JNG	Si moins que ou égal/Si pas plus grand
Branchement conditionnels (flags)	JC	Si retenue
	JE/JZ	Si égal/Si zéro
	JNC	Si pas de retenue
	JNE/JNZ	Si non égal / Non zéro
	JNO	Si pas de débordement
	JNP/JPO	Si pas de parité/ Si parité impaire
	JNS	Si pas de signe
	JO	Si débordement
	JP/JPE	Si parité / Si parité paire
	JS	Si signe (négatif)
Boucles	LOOP	Boucle
	LOOPE/LOOPZ	Boucle si égal/Si zéro
	LOOPNE/LOOPNZ	Boucle si différent/si diff 0
	JCXZ	Branchement si CX=0
Interruptions	INT	Interruption
	INTO	Interruption si débordement
	IRET	Retour d'interruption.

V-1) Branchement inconditionnel

V-1-1) CALL : notion de procédure :

La notion de procédure en assembleur correspond à celle de fonction en langage C, ou de sous-programme dans d'autres langages.

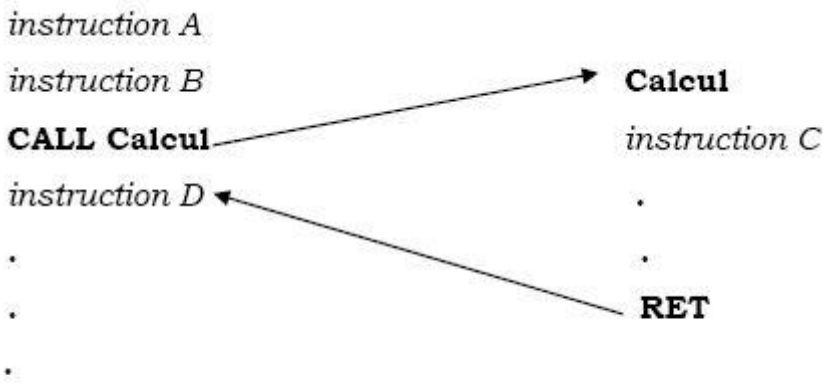


FIG. – Appel d'une procédure.

La procédure est nommée calcul. Après l'instruction B, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D.

Une procédure est une suite d'instructions effectuant une action précise, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme.

Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

L'exécution d'une procédure est déclenchée par un programme *appelant*. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite.

Instructions CALL et RET

L'appel d'une procédure est effectué par l'instruction CALL.

CALL *adresse_debut_procedure*

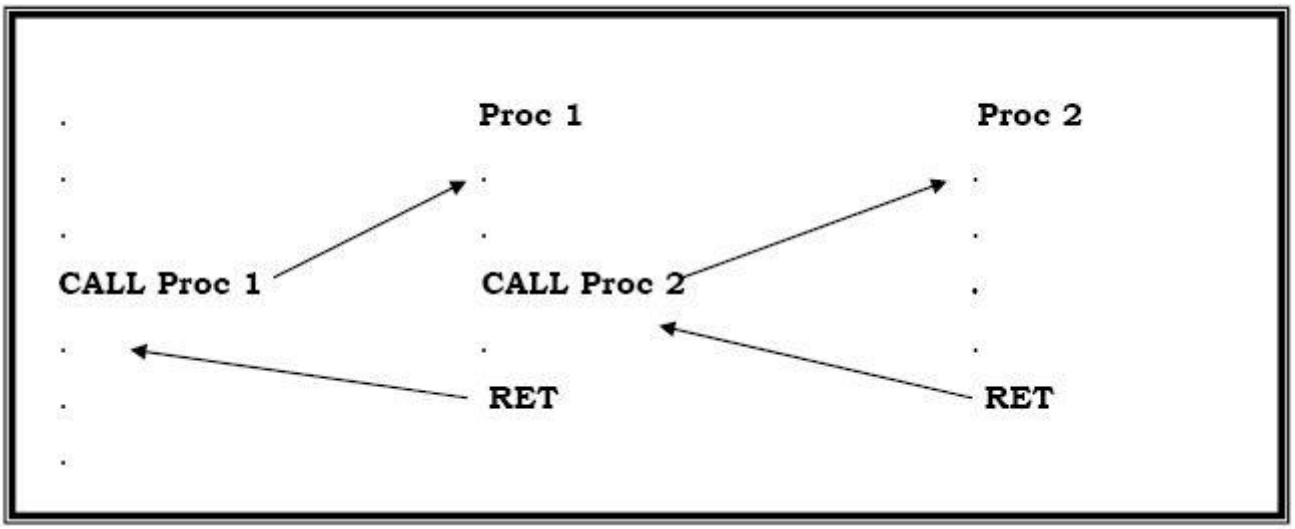
L'adresse est sur 16 bits, la procédure est donc dans le même segment d'instructions. CALL est une nouvelle instruction de branchement inconditionnel. La fin d'une procédure est marquée par l'instruction RET :

V-1-2) RET :

RET ne prend pas d'argument ; le processeur passe à l'instruction placée immédiatement après le CALL.

RET est aussi une instruction de branchement : le registre IP est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL. Comment le processeur retrouve-t-il cette valeur ? Le

problème est compliqué par le fait que l'on peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure suivante :



L'*adresse de retour*, utilisée par RET, est en fait sauvegardée sur la pile par l'instruction CALL. Lorsque le processeur exécute l'instruction RET, il dépile l'adresse sur la pile (comme POP), et la range dans IP.

L'instruction CALL effectue donc les opérations :

- Empiler la valeur de IP. A ce moment, IP pointe sur l'instruction qui suit le CALL.
- Placer dans IP l'adresse de la première instruction de la procédure (donnée en argument).

Et l'instruction RET :

- Dépiler une valeur et la ranger dans IP.

Remarque 1 :

Si la procédure appartient au même segment que le programme principal elle est dite de type NEAR sinon elle est dite de type FAR, la différence entre eux c'est que dans le premier cas le processeur doit empiler une seule valeur dans la pile c'est le registre IP mais dans le deuxième cas il faut empiler le registre IP ainsi que le registre segment CS et bien sur il les dépiler pendant le retour de la procédure.

Remarque 2 : Passage de paramètres

En général, une procédure effectue un traitement sur des données

(*paramètres*) qui sont fournies par le programme appelant, et produit un résultat qui est transmis à ce programme. Plusieurs stratégies peuvent être employées :

1. *Passage par registre* : les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).

2. *Passage par la pile* : les valeurs des paramètres sont empilées. La procédure lit la pile.

Exemple avec passage par registre

On va écrire une procédure (SOMME) qui calcule la somme de 2 nombres naturels de 16 bits.

Convenons que les entiers sont passés par les registres AX et BX, et que le résultat sera placé dans le registre AX.

La procédure s'écrit alors très simplement : SOMME PROC NEAR

```
ADD AX, BX      ; AX <- AX + BX
RET SOMME ENDP
```

et son appel, par exemple pour ajouter 6 à la variable Truc :

```
MOV AX, 6
MOV BX, Truc
CALL SOMME
MOV Truc, AX
```

Exemple avec passage par la pile

Cette technique met en œuvre un nouveau registre, BP (*Base Pointer*), qui permet de lire des valeurs sur la pile sans les dépiler ni modifier SP. Le registre BP permet un mode d'adressage indirect spécial, de la forme :

```
MOV AX, [BP+6]
```

Cette instruction charge le contenu du mot mémoire d'adresse BP+6 dans

AX. Ainsi, on lira le sommet de la pile avec :

```
MOV BP, SP          ; BP pointe sur le sommet
MOV AX, [BP]        ; lit sans dépiler
```

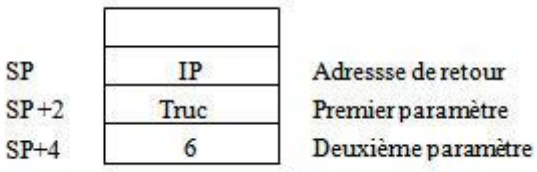
Et le mot suivant avec :

```
MOV AX, [BP+2]      ; 2 car 2 octets par mot de pile.
```

L'appel de la procédure (SOMME2) avec passage par la pile est :

```
PUSH 6
PUSH Truc
CALL SOMME2
```

La procédure SOMME2 va lire la pile pour obtenir la valeur des paramètres. Pour cela, il faut bien comprendre quel est le contenu de la pile après le CALL :



Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP

empilée par CALL). Chaque élément de la pile occupe deux octets. La procédure SOMME2 s'écrit donc :

```
SOMME2 PROC near              ; AX <- arg1 + arg2
MOV BP, SP                    ; adresse sommet pile
MOV AX, [BP+2]                ; charge argument 1
ADD AX, [BP+4]                ; ajoute argument 2
RET
SOMME2 ENDP
```

La valeur de retour est laissée dans AX.

La solution avec passage par la pile parait plus lourde sur cet exemple simple. Cependant, elle est beaucoup plus souple dans le cas général que le passage par registre. Il est très facile par exemple d'ajouter deux paramètres supplémentaires sur la pile. Une procédure bien écrite modifie le moins de registres possible. En général, l'accumulateur est utilisé pour transmettre le résultat et est donc modifié. Les autres registres utilisés par la procédure seront normalement sauvegardés sur la pile. Voici une autre version de SOMME2 qui ne modifie pas la valeur contenue par BP avant l'appel :

```
SOMME2 PROC near                  ; AX <- arg1 + arg2
PUSH BP                           ; sauvegarde BP
MOV BP, SP                        ; adresse sommet pile
MOV AX, [BP+4]                    ; charge argument 1
ADD AX, [BP+6]                    ; ajoute argument 2
POP BP                            ; restaure ancien BP
RET
SOMME2 ENDP
```

Noter que les index des arguments (BP+4 et BP+6) sont modifiés car on a ajouté une valeur au sommet de la pile.

V-1-3) JMP : (Saut inconditionnel)

Syntaxe :

```
JMP cible
```

Si le JMP est de type NEAR alors IP = IP + Déplacement

Si le JMP est de type FAR alors CS et IP sont remplacé par les nouvelles valeurs obtenues à partir de l'instruction.

JMP transfert, sans condition, la commande à l'emplacement de destination. L'opérande Cible peut être obtenu à partir de l'instruction elle- même (JMP direct) ou à partir de la mémoire ou à partir d'un registre indiqué par l'instruction.

V-2 saut conditionnel :

V-2-1) JC : (Si retenue)

Si CF=1 alors IP = IP + déplacement

V-2-2) JE/JZ :(Si égal/Si zéro)

Si ZF=1 alors IP = IP + déplacement

V-2-3) JNC :(Si pas de retenue)

Si CF=0 alors IP = IP + déplacement

V-2-4) JNE/JNZ :(Si non égal) Non zéro)

Si ZF=0 alors IP = IP + déplacement

V-2-5) JNO :(Si pas de débordement)

Si OF=0 alors IP = IP + déplacement

V-2-6) JNP/JPO :(Si pas de parité/ Si parité impaire)

Si PF=0 alors IP = IP + déplacement

V-2-7) JNS :(Si pas de signe)

Si SF=0 alors IP = IP + déplacement

V-2-8) JO :(Si débordement)

Si OF=0 alors IP = IP + déplacement

V-2-9) JP/JPE:(Si parité) Si parité paire)

Si PF=1 alors IP = IP + déplacement

V-2-10) JS :(Si signe (négatif))

Si SF=1 alors IP = IP + déplacement

V-3) Les instructions de boucle :

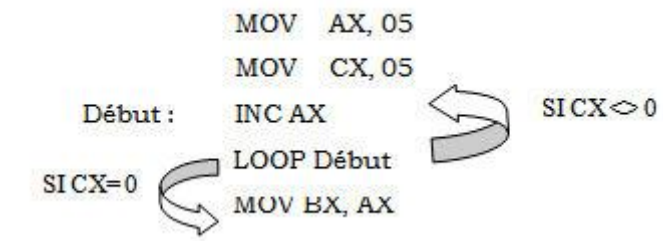
V-3-1 /LOOP : (boucle) :

Elle décrémente le contenu de CX de 1.

Si CX est différente de zéro alors IP = IP + déplacement

Si CX = 0 l'instruction suivante est exécutée.

Exemple :

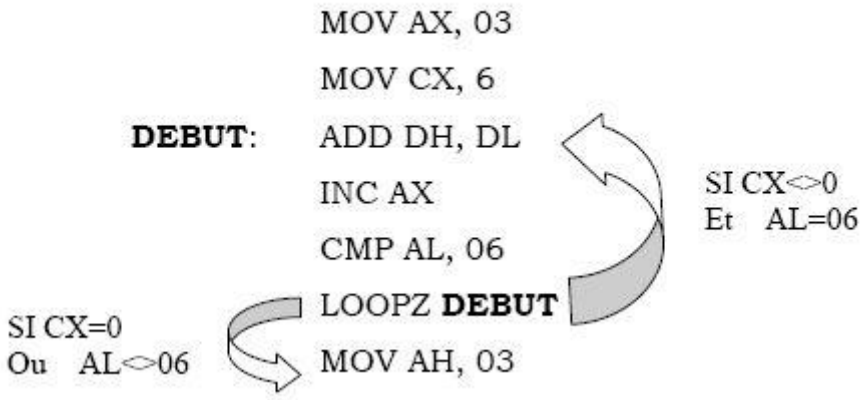


L'exécution de l'instruction MOV BX, AX sera faite après l'exécution de la boucle 5 fois.

V-3-2) LOOPE) LOOPZ : (boucle si égale ou si égale à zéro) : Le registre CX est décrémenter de 1 automatiquement

Si CX est différent de zéro et ZF=1 alors IP = IP + déplacement

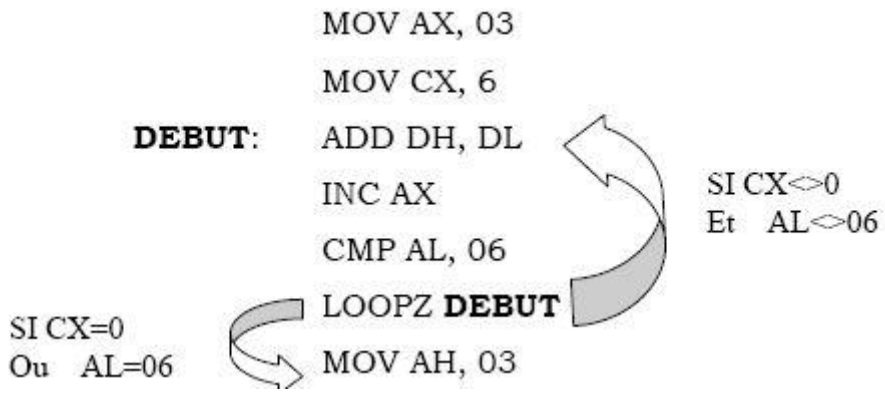
Exemple :



V-3-3) LOOPNE) LOOPNZ : (boucle si égale ou si égale à zéro) : Le registre CX est décrémenter de 1 automatiquement

Si CX est différent de zéro et ZF=0 alors IP = IP + déplacement

Exemple :



VI) Les instructions de chaînes de caractères :

Les instructions de chaînes des caractères sont au nombre de 14 comme le montre le tableau suivant :

Nom	Fonction
REP	Préfixe de répétition
REPE/REPZ	Répétition tant qu'égal à zéro
REPNE/REPNZ	Répétition tant que différent de zéro
MOVS	Déplacement de chaîne
MOVSB/MOVSW	Déplacement de chaîne
CMPS	Comparaison de chaînes
INS	Entrée (de port d'E/S)
OUTS	Sortie (vers un port d'E/S)
SCAS	Balayage d'une chaîne
LODS	Chargement de chaîne
STOS	Rangement de chaînes

Elles permettent de travailler sur des blocs d'octets ou de mots allant jusqu'à 64 Koctet.

Remarque :

Ces blocs peuvent être des valeurs numériques ou alphanumériques.

VI -1) Les préfixes de répétitions :

VI-1-1) REP :

Ces instructions sont utilisées avec les instructions de chaînes de caractères pour assurer la répétition de l'instruction si on veut appliquer l'instruction sur un ensemble d'informations.

REP décrément automatiquement CX est test est ce qu'il est égal à zéro ou non. Si CX = 0 REP s'arrêt

VI-1-2) REPE) REPZ :

Pour REPE/REPZ : c'est la même chose que REP c'est-à-dire elle décrément automatiquement le registre CX mais elle peut sortir de la boucle si ZF<>0

VI-1-2) REPNE) REPNZ :

Pour REPNE/REPNZ : c'est la même chose que REP c'est-à-dire elle décrément automatiquement le registre CX mais elle peut sortir de la boucle si ZF=0

VI-2/ Les instructions MOVE-STRING :

Elle déplace un élément du segment de données pointé par

DS : SI vers le segment Extra pointé par ES : DI

Remarque :

Si l'élément à transférer est un octet on utilise : MOVB Si l'élément à transférer est un Mot on utilise : MOVW

Mais dans les deux cas on n'utilise que d'opérande.

SI et DI sont ensuite incrémentés de 1 (si DF=0) ou décrémentés de 1 (si DF=1) d'une manière automatique.

Exemple :

```
Donnee SEGMENT
Mess_Sour db 'bonjour iset de nabeul'     ; message source
Donnee ENDS
Extra SEGMENT
Mes_Des db 22 dup (0)                    ; message destination
Extra ENDS
Code SEGMENT
Assume CS : Code, DS : Donnee, ES : extra
PROG PROC
MOV AX,Donnee MOV DS, AX MOV AX,Extra MOV ES, AX
LEA SI, Mess_Sour                 ; pointé le message source
LEA DI, Mess_Des                  ; pointé le message destination
MOV CX, 22                        ; nombre de caractère à transfèrer
CLD                               ; incrémentation automatique du SI et DI
REP MOVSB                          ; transfert avec le préfixe REP MOV AX,
4C00H
                                  ; Retour au DOS
INT 21H
PROG ENDP
CODE ENDS
```

```
END PROG
```

VI-3/ Les instructions COMPARE-STRING :

Comparaison de chaîne : elle soustrait l'octet ou mot de destination (pointé par DI) de l'octet ou mot source (pointé par SI).CMPS affecte les indicateurs mais ne change pas les opérandes.

Si CMPS est utilisé avec le préfixe de répétition REPE/REPZ, elle est interprétée comme « comparer tant que la chaîne n'est pas finie (CX <>0) et que les éléments à comparer ne sont pas égaux (ZF=1)

Si CMPS est utilisé avec le préfixe de répétition REPNE/REPNZ, elle est interprétée comme « comparer tant que la chaîne n'est pas finie (CX <>0) et que les éléments à comparer ne sont pas égaux (ZF=0)

Remarque :

On ne peut pas utilisé le préfixe REP avec l'instruction CMPS car cela revient à comparer uniquement les deux derniers éléments des deux chaînes.

Exemple :

trouver le premier caractère différent entre les deux chaînes.

```
Donnee SEGMENT
Mess_1 db 'bonjour iset de nabeul' Mess_2 db 'bonsoir iset de nabeul'
Donnee ENDS
Code SEGMENT
Assume CS : Code, DS : Donnee
PROG PROC
MOV AX,Donnee
MOV DS, AX
LEA SI, Mess_Sour  ; pointé le message source
LEA DI, Mess_Des   ; pointé le message destination
MO V CX, 22        ; nombre de caractère àcomparer
CLD                ; incrémentation automatique du SI et DI
REP CMPSB          ; transfert avec le préfixe
REP JCXZ  Tito     ; les deux chaînes sont identiques
MOV al, [SI-1]     ; on met dans AL le caractère
                   ; Différent
Tito : MOV AX, 4C00H  ; Retour au DOS INT 21H
PROG ENDP
CODE ENDS
END PROG
```

VI-4) Les instructions SCAN STRING :

Syntaxe :

```
SCAS chaine_destination
SCASB
SCASW
```

SCAS soustrait l'élément de la chaîne de destination (octet ou mot) adressé par DI dans le segment extra du contenu de AL (un octet) ou de AX (un mot) et agit sur les indicateurs. Ni la chaîne destination ni l'accumulateur ne change de valeur.

Exemple **:**

recherche de la lettre ‘a' dans une chaîne. EXTRA SEGMENT

```
Mess_Des db ‘bonjour iset de nabeul' EXTRA ENDS
Code SEGMENT
Assume CS : Code, ES : EXTRA
PROG PROC MOV AX, EXTRA MOV ES, AX
LEA DI, Mess_Des                  ; pointé le message destination
MOV CX, 22                        ; nombre de caractère à comparer
CLD                               ; incrémentation automatique du DI
MOV AL,'a'
REPNZ SCASB                       ; transfert avec le préfixe REP JCXZ
Tito  ; les deux chaînes sont identiques
Call oui_le_a_existe          ; on met dans AL le caractère
; Différent
Tito : MOV AX, 4C00H              ; Retour au DOS
INT 21H
PROG ENDP
CODE ENDS
END PROG
```

VI-5) Les instructions LOAD STRING (LODS) et STORE STRING (STOS) :

VI-4-1) LODS :

Syntaxe :

```
LODS chaine_source
LODSB
LODSW
```

LODS transfert l'élément de chaîne (octet ou mot) adressé par SI au registre AL ou AX et remet à jour SI pour qu'il pointe vers l'élément suivant de la chaîne.

VI-4-2) STOS :

Syntaxe :

```
STOS chaine_destination
STOSB
STOSW
```

STOS transfert un octet ou un mot du registre AL ou AX vers l'élément de chaîne adressé par DI et modifie DI pour qu'il pointe vers l'emplacement suivant de la chaîne.

Exemple :

transfert d'une chaîne source vers une chaîne destination en utilisant LODS et STOS.

```
Donnee SEGMENT
Mess_Sour db 'bonjour iset de nabeul'     ; message source
Donnee ENDS
Extra SEGMENT
Mes_Des db 22 dup (0)                     ; message destination
Extra ENDS
Code SEGMENT
Assume CS : Code, DS : Donnee, ES : extra
PROG PROC
MOV AX,Donnee
MOV DS, AX
MOV AX,Extra
MOV ES, AX
LEA SI, Mess_Sour                 ; pointé le message source
LEA DI, Mess_Des                  ; pointé le message destination
MOV CX, 22                        ; nombre de caractère à transfèrer
CLD                               ; incrémentation automatique du SI et DI
DEBUT :     LODSB                 ; transfert avec le préfixe REP STOSB
LOOP DEBUT
MOV AX, 4C00H                     ; Retour au DOS INT 21H
PROG ENDP
CODE ENDS
END PROG
```

VII) Les instructions de commande du processeur :

Ces instructions agissent sur le processeur et ses indicateurs (Flags) ils sont en nombre de 12 comme le montre le tableau suivant

Type	Nom	Fonction
Indicateur (FLAGS)	STC	Met à 1 la retenue CF
	CLC	MET à 0 la retenue CF
	CMC	Complémente la retenue
	STD	Met à 1 la direction DF
	CLD	Met à 0 la direction DF
	STI	Met à 1 l'autorisation d'interruption
	CLI	Met à 0 l'autorisation d'interruption
Synchronisation	HLT	Halte jusqu'à interruption ou RESET
	WAIT	Attente jusqu'à broche TEST passe à 0
	ESC	Pour un coprocesseur
	LOCK	Verrouillage des bus pendant la prochaine instructions
Sans opération	NOP	Pas d'opération

VII-1) Indicateurs :

VII-1-1/ STD :

Met CF à 1 ; les registres d'indexation SI et/ou DI sont alors automatiquement décrémenter par les instructions de chaîne de caractère.

VII-1-2) STI :

Met IF à 1, permettant ainsi au CPU de reconnaître des demandes d'interruption masquables apparaissant sur la ligne d'entrée INTR.

VII-2) Synchronisation :

VII-2-1) HALT :

Maintient le processeur dans un état d'attente d'un RESET ou d'une interruption externe non masquable ou masquable (avec IF=1).

VII-2-2) WAIT :

Met le CPU en état d'attente tant que sa ligne de TEST n'est pas active. En effet toutes les cinq périodes d'horloge le CPU vérifie est ce que cette entrée est active ou non, si elle est active le processus exécute l'instruction suivante à WAIT.

VII-2-3) ESC :

L'instruction Escape fournit un mécanisme par lequel des coprocesseurs peuvent recevoir leurs instructions à partir de la suite d'instructions du 8086.

VII-2-4) LOCK :

Elle utilise dans les systèmes Multiprocesseur en effet elle permet le verrouillage du bus vis-à-vis des autres processeurs.

VII-3 Sans opération :

VII-3-1) NOP (No operation) :

Le CPU ne fait rien on peut s'en servir pour créer des temporisations.

Exemple :

```
Tempo : MOV CX, 7FFFH            ; Effectuer une temporisation
Temp1: PUSH CX                       ; avec deux boucles imbriqués
MOV CX,7FFFH
Temp2: NOP NOP NOP NOP
LOOP Temp2
POP CX
LOOP Temp1
RET
```

Programmation en assembleur

I) Introduction :

Lorsque l'on doit lire ou écrire un programme en langage machine, il est difficile d'utiliser la notation hexadécimale. On écrit les programmes à l'aide des instructions en mnémonique comme MOV, ADD, etc. Les concepteurs de processeurs, comme Intel, fournissent toujours une documentation avec les codes des instructions de leurs processeurs, et les symboles correspondantes.

L'assembleur est un utilitaire qui n'est pas interactif, (contrairement à l'utilitaire comme *debu*g : voir plus loin dans le cours). Le programme que l'on désire traduire en langage machine (on dit *assembl*er) doit être placé dans un fichier texte (avec l'extension .ASM sous DOS).

La saisie du programme source au clavier nécessite un programme appelé *éditeur de text*e.

L'opération d'assemblage traduit chaque instruction du programme source en une instruction machine. Le résultat de l'assemblage est enregistré dans un fichier avec l'extension .OBJ *(fichier objet*).

Le fichier .OBJ n'est pas directement exécutable. En effet, il arrive fréquemment que l'on construise un programme exécutable à partir de plusieurs fichiers sources. Il faut (relier) les fichiers objets à l'aide d'un utilitaire nommé *éditeur de lien* (même si l'on a qu'un seul). L'éditeur de liens fabrique un fichier exécutable, avec l'extension .EXE.

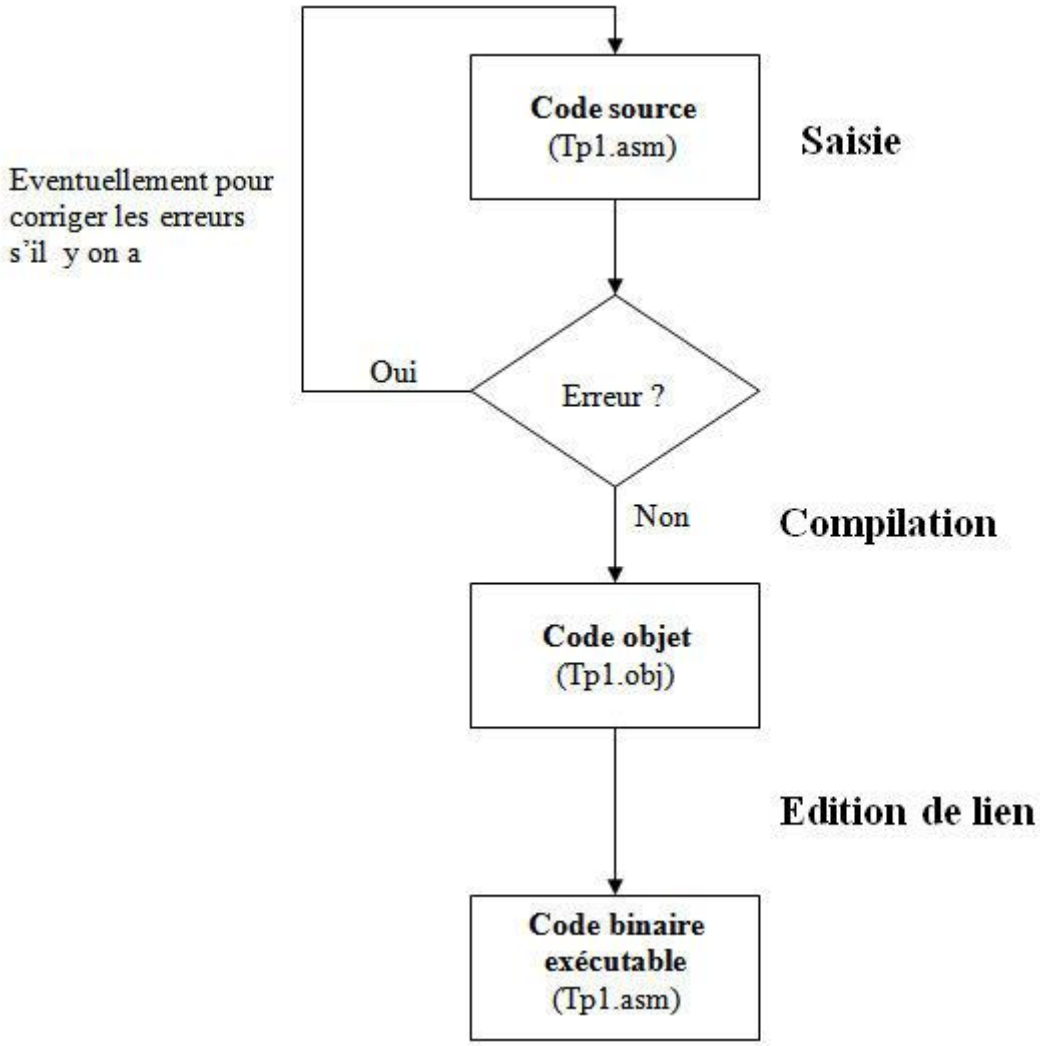
Le fichier .EXE est directement exécutable. Un utilitaire spécial du système d'exploitation (DOS ici), le *chargeur* est responsable de la lecture du fichier exécutable, de son implantation en mémoire principale, puis du lancement du programme.

Donc en conclusion pour assembler un programme on doit passer par les phases suivantes :

- Saisie du code source avec un éditeur de texte.

- Compiler le programme avec un compilateur.

- Editer les liens pour avoir un programme exécutable. Les trois phases sont schématisées par la figure suivante :



Remarque 1 :

- On ne peut passer du code source vers le code objet que si le programme source ne présente aucune erreur.
- La saisie se fait par des logiciels qui s'appellent éditeurs de texte, donc on peut utiliser n'importe quel éditeur de textes (tel que EDLINE sous MSDOS de Microsoft) sauf les éditeurs sous Windows car ces éditeurs ajoutent dans le fichier des informations (la taille des caractères, la police utilisée, la couleur etc...) que l'assembleur ne peut pas comprendre .Pour utiliser les éditeurs sous Windows il est conseiller d'enregistrer les fichiers sous forme RTF.

Editeur de lien :

- permet de lier plusieurs codes objets en un seul exécutable.
- permet d'inclure des fonctions prédéfinies dans des bibliothèques.

Plusieurs logiciels permettent le passage entre les trois phases présentée dans la figure précédente on peut citer : MASM (Microsoft Assembler : avec LINK comme éditeur de lien), TASM (Turbo assembler : avec TLINK comme éditeur de lien) et NASM etc ...

Remarque 2 :

On peut générer à partir d'un fichier objet d'autres formes de fichier pour des systèmes autres que l'ordinateur (compatible IBM). Les formes les plus connues sont INTEL HEX, ASCII HEX etc ...

Remarque 3 :

L'assembleur est utilisé pour être plus près de la machine, pour savoir exactement les instructions générées (pour contrôler ou optimiser une opération) On retrouve l'assembleur dans :

- la programmation des systèmes de base des machines (le pilotage du clavier, de l'écran, etc...),

- certaines parties du système d'exploitation,

- le pilotage de nouveaux périphériques (imprimantes, scanners, etc..

- l'accès aux ressources du système,

L'avantage donc de l'assembleur est de générer des programmes efficaces et rapides (à l'exécution) par contre ses inconvénients : développement et mise au point long.

II) Le fichier source (code source) :

Comme tout programme, un programme écrit en assembleur (programme source) comprend des définitions, des données et des instructions, qui s'écrivent chacune sur une ligne de texte.

Les données sont déclarées par des *directive*s, mots clefs spéciaux que comprend l'assembleur (donc ils sont destinés à l'assembleur. Les instructions (sont destinées au microprocesseur)

II-1) Les instructions :

La syntaxe des instructions est comme suit :

{Label :} Mnémonique {opérande} { ; commentaire}

- Le champ opérande est un champ optionnel selon l'instruction (parfois l'instruction nécessite une opérande et parfois non).

- Le champ commentaire: champ sans signification syntaxique et sémantique pour l'assembleur , il est optionnel mais très intéressant lorsque on programme en assembleur, en effet les instructions en assembleur sont des instructions élémentaires donc dans un programme le nombre d'instructions et assez élevé (par exemple pour utiliser des fonctions tels que COS ou SIN il faut réaliser ça en utilisant des opérations arithmétiques et

logiques de base) donc contrairement au langage évolué de programmation dans les programmes source on va trouver plus d'instructions.

Ce qui va rendre la compréhension des programmes assez délicate et difficile. Pour cette raison lorsque on programme en assembleur il vaut mieux mettre des commentaires pour que le programme soit lisible pour les utilisateurs.

Remarque :

Les commentaires sont mis en général dans les passages délicats.

- Le champ Label (étiquette) est destiné pour marquer une ligne qui sera la cible d'une instruction de saut ou de branchement. Une label peut être formée par 31 caractère alphanumérique ({A.. Z} {a.. z} {0.. 9} {.?@_$}) au maximum .Les noms des registres ainsi que la représentation mnémonique des instructions et les directives (voir plus loin) ne peuvent être utilisées comme Label. Le champ Label doit se terminer par ' : ' et ne peut commencer par un chiffre. De même il n'y a pas de distinction entre minuscules et majuscules.

Exemple :

```
ET1 : MOV  AX , 500H   ; mettre la valeur 500 dans le registre AX
```

II-2) Les directives :

Pour programmer en assembleur, on doit utiliser, en plus des instructions assembleur, des directives ou pseudo-instructions : 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. Donc les directives sont des déclarations qui vont guider l'assembleur.

Une directive est utilisée par exemple pour créer de l'espace mémoire pour des variables, pour définir des constantes, etc...

Pour déclarer une directive il faut utiliser la syntaxe suivante :

{Nom} Directive {opérande} { ; commentaire}

- Le champ opérande dépend de la directive
- Le champ commentaire a les mêmes propriétés vues dans le paragraphe précèdent.
- Le champ Nom indique le nom des variables : c'est un champ optionnel (selon la directive).

II-3 /Exemple de directives :

III- 3-1) Les directives de données :

III-3- 1-1 / EQU :

Syntaxe : Nom EQU Expression

Exemples :

```
VAL  EQU   50    ; assigne la valeur 50 au nom VAL
ET1   EQU   VAL* 5 + 1 ; assigne une expression calculer a VAL
```

III-3- 1-2) DB/DW/DD/DF/DP/DQ/DT:

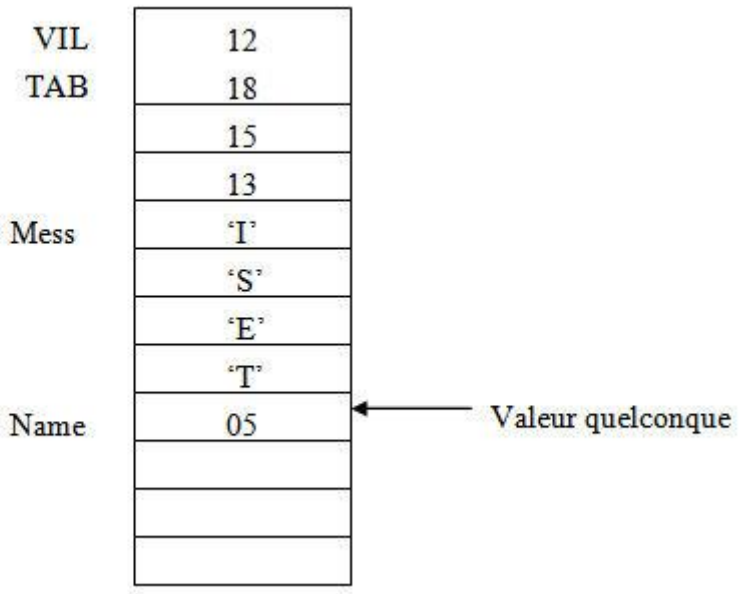
Ces directives sont utilisées pour déclarer les variables : L'assembleur attribue à chaque variable une adresse. Dans le programme, on repère les variables grâce à leurs noms. Les noms des variables sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre. Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les caractères @, et _. Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

a°) **DB (Define byte):** définit une variable de 8 bits : c a d elle réserve un espace mémoire d'un octet : donc les valeurs qu'on peut stocker pour cette directive sont comprises entre 0 et 255 (pour les nombres non signés) et de -128 jusqu'à 127 pour les nombres signés .

Syntaxe : Nom DB Expression

Exemple :

Vil DB 12H ; Définit une variable (un octet) de valeur Initiale 12.
Tab DB 18H, 15H, 13H ; définit un tableau de 3 cases ;(3 octet) Qui démarre à partir de l'adresse TAB.
Mess DB 'ISET' ; définit aussi un tableau mais les valeurs de chaque case ; n'est autre que le code ascii de chaque lettre.
Name DB ? ; définit une variable 8 bits de valeur initiale quelconque.

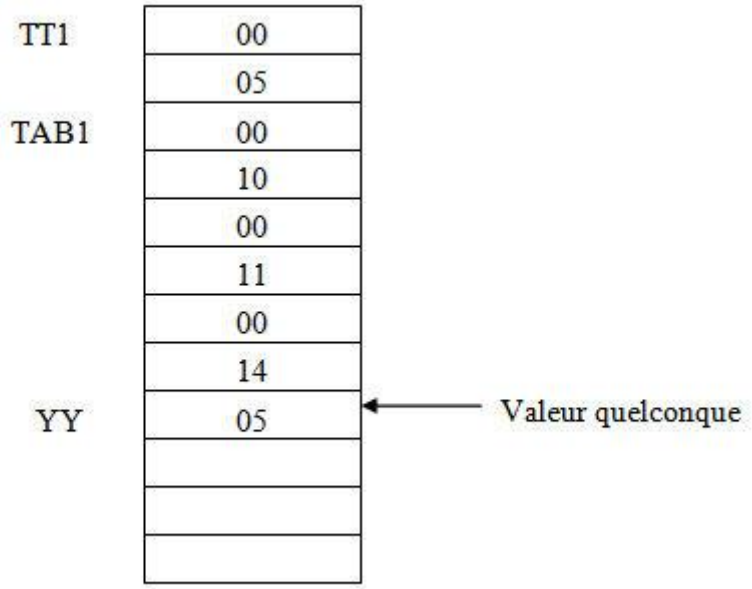


b°) DW (define word) : définit une variable de 16 bits : c a d elle réserve un espace mémoire d'un mot : donc les valeurs qu'on peut stocker pour cette directive sont comprises entre 0 et 65535 (pour les nombres non signés) et de -32768 jusqu'à 32767 pour les nombres signés .

Syntaxe : Nom DW Expression

Exemple :

```
TT1  DW     500H ; réserve deux cases mémoire (un mot) a partir de l'adresse
TT1.
TAB1 DW    10H,11H,14H ; réserve u  tableau de 6 cases chaque valeur sera
mise sur deux cases
YY     DW    ?; réserve un mot dans la mémoire de valeur initial quelconque.
```



c°) DD : (Define Double) : réserve un espace mémoire de 32 bits (4 cases mémoire ou 4 octets) :

Syntaxe : nom DD expression

Exemple :

```
 ff DD   15500000H
```

e°/ Directive dup

Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive *du*p:

```
tab DB 100 dup (15) ; 100 octets valant 15
y DW 10 dup (?) ; 10 mots de 16 bits non initialises
```

f°/ Les direcitive Word PTR et Byte PTR :

Dans certains cas, l'adressage indirect est ambigu. Par exemple, si l'on

écrit :

```
MOV [BX], 0 ;          range 0 a l'adresse spécifiée par BX
```

L'assembleur ne sait pas si l'instruction concerne 1, 2 ou 4 octets

consécutifs. Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

```
MOV byte ptr [BX], val ; concerne 1 octet
MOV word ptr [BX], val ; concerne 1 mot de 2 octets
```

III- 3-2) Les directives de segment :

La directive SEGMENT contrôle la relation entre la génération du code objet et la gestion des segments logiques ainsi générés.

L'instruction SEGMENT sert à :

- contrôler le placement du code objet dans des segments spécifiques.

- Associer les symboles représentant des adresses à un segment en considérant leur valeur comme un déplacement par rapport au début du segment.

- spécifier des directives pour l'editeur de lien (nom du segment, champs d'opérande de l'instruction SEGMENT déterminant le traitement du segment par l'éditeur de liens); ces informations sont passées telles quelles.

Syntaxe de la directive SEGMENT : Nom SEGMENT opérande

.

.

Nom ENDS

Le nom champ étiquette 'Nom' sert a :

- indiquer le nom du segment.

- établir une relation évidente entre des paires d'instructions

SEGMENT et ENDS. Alternance entre segments :

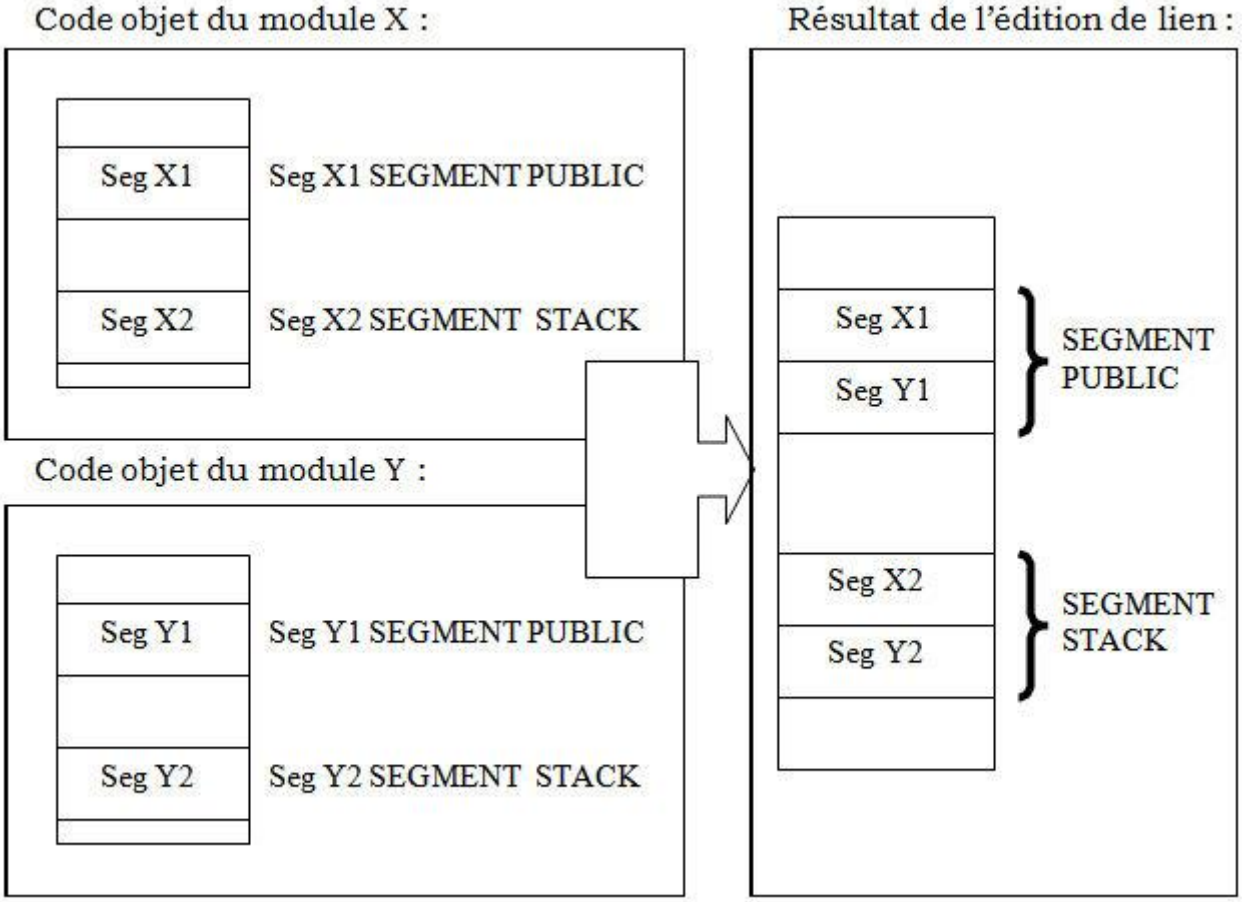
Un programme peut alterner entre différents segments pour y générer un code:

* L'instruction SEGMENT permet de re-ouvrir un segment déjà existant

(donc, SEGMENT soit crée un nouveau segment, soit ouvrir un segment en vue d'y ajouter du code supplémentaire).

Remarque :

Il ne faut pas oublier l'instruction ENDS avant une telle opération, elle permet de (temporairement) clore l'ancien segment.



Les opérandes de l'instruction SEGMENT déterminent la manière dont l'éditeur de liens traitera le segment :

Nom SEGMENT { COMMON | PUBLIC | STACK | MEMORY | AT adr } { BYTE | WORD | PARA | PAGE } TITO

COMMON :

Tous les segments avec l'étiquette *(classe*) seront placés à la même adresse de base (dans un bloc contigu) ; des zones du type (COMMON) avec différents noms ((classe)) seront placés l'un derrière l'autre.

PUBLIC :

Tous les segments avec ce qualificatif seront regroupés dans un seul segment résultant, l'un derrière l'autre.

STACK :

Un seul segment avec ce qualificatif est accepté, il est destiné à la gestion de la pile.

MEMORY :

Le premier segment portant ce qualificatif sera placé à une position de mémoire en dessus de tout autre segment; s'il y a d'avantage de segments de ce genre, ils seront traités comme les segments du type (COMMON).

AT :

Les étiquettes définies dans un tel segment sont définies comme étant relatives à la valeur ((adr)) 16) * 16.

e°/ Alignement de l'adresse de base d'un segment

Il est possible de contrôler la manière dont l'éditeur de liens détermine l'adresse ou sera placé un segment: on choisit l'alignement du segment (c.a.d de son premier byte).

Mot clé	Alignement sur	Adresse modulo
BYTE	Frontière de BYTES	1
WORD	Frontière de MOTS	2
PARA	Frontière de Paragraphes	16
PAGE	Frontière de Pages	256

IV) Structure d'un programme source :

```
PROGRAM  Exemple1
Pile SEGMENT STACK ; On met les directives pour réserver de l'espace mémoire.
PILE ENDS
Data SEGMENT ; On met les directives de données pour réserver de la mémoire
             ; Pour les variables qui seront utilisées dans le programme.
Data ENDS
Extra SEGMENT ; On met les directives pour déclarer ;
              ;les variables (les chaînes de Caractères).
```

```
Extra ENDS
Code SEGMENT
ASSUME   cs : code, ds, data, es : pile :ss :pile
PROG
         Mov AX,Data
         Mov DS,AX
         Mov AX,Extra
         Mov ES,AX
         Mov AX,pile
         Mov SS,AX   ; mettre les instructions du programme
Code ENDS END PROG
```

Comme tout programme, un programme écrit en assembleur comprend des définitions de données et des instructions, qui s'écrivent chacune sur une ligne de texte.

Les données sont déclarées par des directives, Les directives qui déclarent des données sont regroupées dans les segments de données, qui sont délimités par les directives *SEGMENT* et *END*S.

Les instructions sont placées dans un autre segment, le segment de code. La ligne :

Code SEGMENT

Sert à déclarer le segment code. On aurait aussi bien pu le nommer

(iset) ou *(microprocesseur*). Ce sera le segment de notre programme. Cette ligne ne sera pas compilée : elle ne sert qu'à indiquer au compilateur le début d'un segment.

La ligne :

PROG

La première instruction du programme (dans le segment d'instruction) doit toujours être repérée par une étiquette (dans notre cas : PROG). Le fichier doit se terminer par la directive *END* avec le nom de l'étiquette de la première instruction (ceci permet d'indiquer à l'éditeur de liens qu'elle est la première instruction à exécuter lorsque l'on lance le programme).

Comme nous l'avons vu, les directives *SEGMENT* et *ENDS* permettent de définir les segments de codes et de données. La directive *ASSUME* permet d'indiquer à l'assembleur quel est le segment de données et celui de codes

(etc...), afin qu'il génère des adresses correctes. Enfin, le programme doit commencer, avant toute référence au segment de données, par initialiser le registre segment DS (même chose pour : ES et SS), de la façon suivante :

```
MOV AX, Data
MOV DS, AX
```

Remarque:

On n'est pas tenu de rendre aux registres la valeur qu'ils avaient au début de notre programme. En effet, 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 il prend la main.

V) 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.

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 *paramètre*s.

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 segmen*t, 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 :

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.

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.

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 coeur :

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

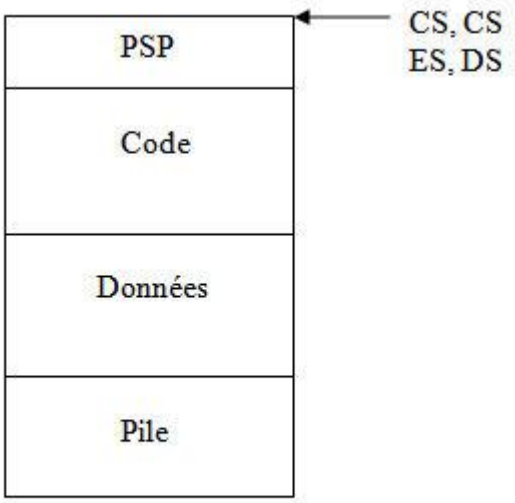
Dans un fichier EXE, le header indique au DOS les adresses initiales de chaque segment par rapport au début du programme (puisque le compilateur n'a aucun moyen de connaître l'adresse à laquelle le programme sera chargé). Lors du chargement, le DOS ajoutera à ces valeurs l'adresse d'implantation pour 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.

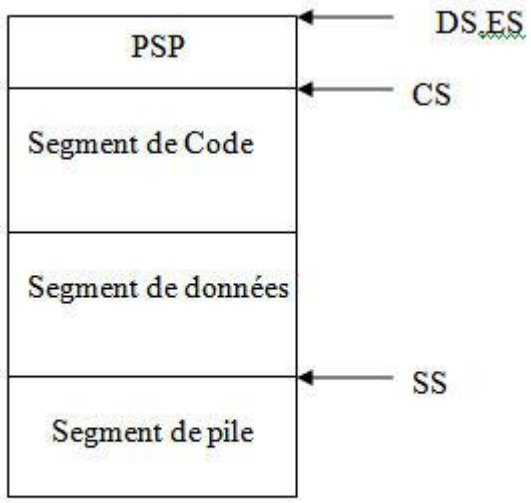
Remarque : *Pourquoi DS et ES pointent-ils vers le PSP dans le cas d'un fichier*

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.



Structure d'un programme .COM



Structure d'un programme .EXE

c°/ Structure d'un fichier .com

```
Code segment
Assume cs : code, ds : code, es : code, ss : code
Org 100h   ;  le programme debut à partir de 100h
Debut :
.....
.....                  ; mettre les instructions
.....
.....
Code ends
End debut
```

d°/ Structure d'un fichier .exe

Voir paragraphe IV /

Les directives de procédure :

Déclaration d'une procédure

L'assembleur possède quelques directives facilitant la déclaration de procédures. On déclare une procédure dans le segment d'instruction comme suit :

```
Calcul    PROC near                    ; procedure nommé Calcul
               ...                              ; instructions
              RET                            ; dernière instruction
Calcul     ENDP                          ; fin de la procédure
```

Le mot clef PROC commence la définition d'une procédure, near indiquant qu'il s'agit d'une procédure située dans le même segment d'instructions que le programme appelant.

L'appel s'écrit simplement : CALL Calcul

VI) Les modes d'adressage du 8086:

Les instructions et leurs opérandes (paramètres) sont stockées en mémoire principale. La taille totale d'une instruction (nombre de bits nécessaires pour la représenter en mémoire) dépend du type d'instruction et aussi du type d'opérande. Chaque instruction est toujours codée sur un nombre entier d'octets, afin de faciliter son décodage par le processeur.

Une instruction est composée de deux champs :

- le code opération, qui indique au processeur quelle instruction réaliser.

- le champ opérande qui contient la donnée, ou la référence à une donnée en mémoire (son adresse).

Les façons de désigner les opérandes constituent les "modes d'adressage". Selon la manière dont l'opérande (la donnée) est spécifié, c'est à dire selon le mode d'adressage de la donnée, une instruction sera codée par 1, 2, 3 ou 4 octets.

Le microprocesseur 8086possède 7 modes d'adressage :

- Mode d'adressage registre.

- Mode d'adressage immédiat.

- Mode d'adressage direct.

- Mode d'adressage registre indirect.

- Mode d'adressage relatif à une base.

- Mode d'adressage direct indexe.

- Mode d'adressage indexée.

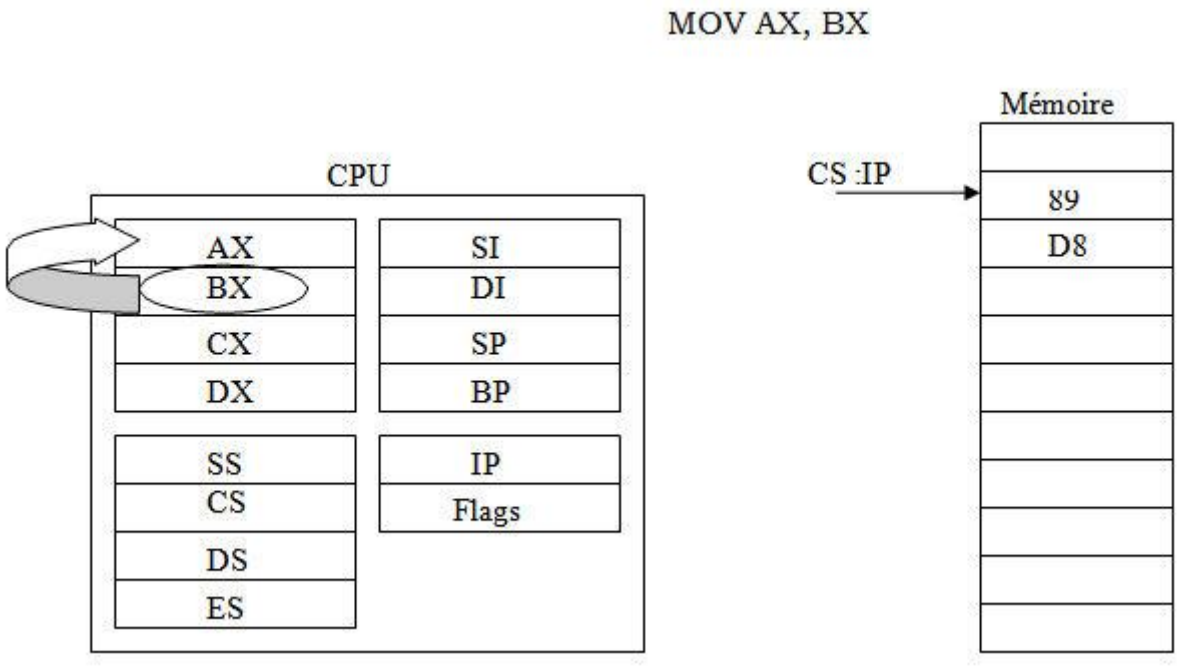
VI -1) Mode d'adressage registre :

Ce mode d'adressage concerne tout transfert ou toute opération, entre deux registres de même taille.

Dans ce mode l'opérande sera stockée dans un registre interne au microprocesseur.

Exemple :

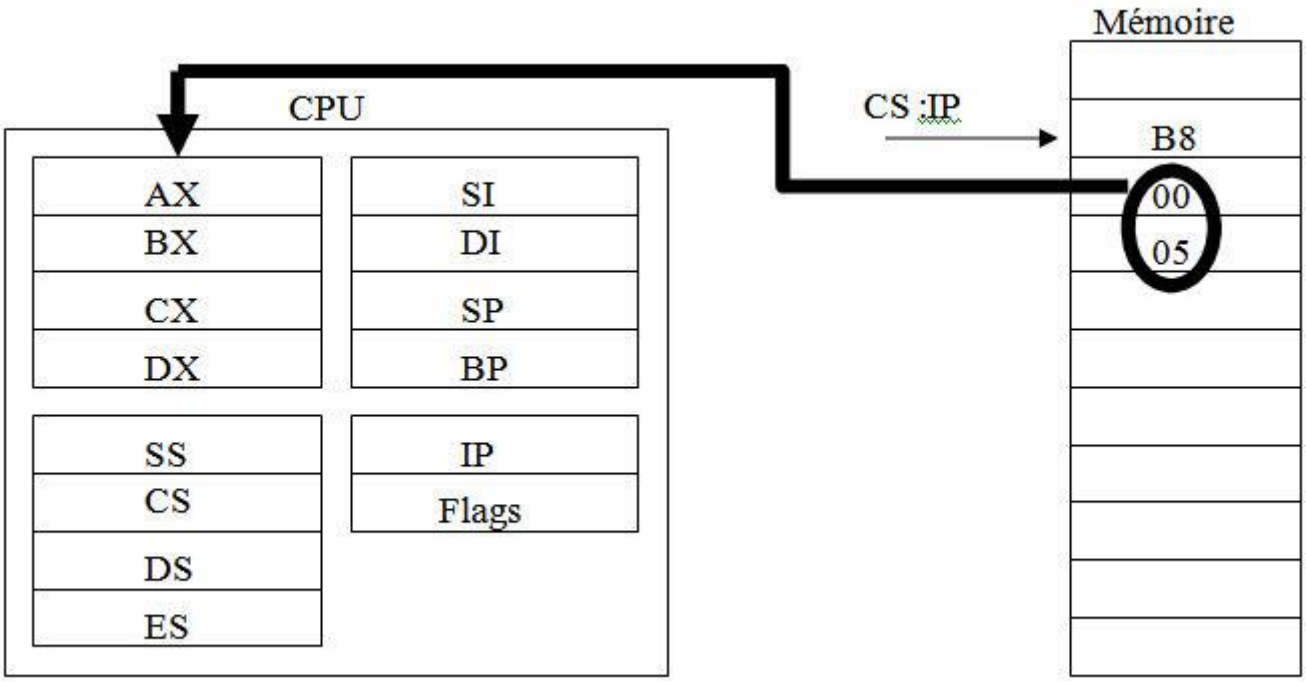
Mov AX, BX ; cela signifie que l'opérande stocker dans le registre BX sera transféré vers le registre AX. Quand on utilise l'adressage registre, le microprocesseur effectue toutes les opérations d'une façon interne. Donc dans ce mode il n'y a pas d'échange avec la mémoire, ce qui augmente la vitesse de traitement de l'opérande.



IV -2) Mode d'adressage immédiat :

Dans ce mode d'adressage l'opérande apparaît dans l'instruction elle- même, exemple :

```
MOV   AX,500H ; cela signifie que la valeur 500H sera stockée immédiatement
dans le registre AX
```



Remarque :

Pour les instructions telles que :

```
MOV    AX,-500H ;  le  signe  -  sera  propager  dans
                ;le  registre  jusqu'à remplissage de ce dernier.
```

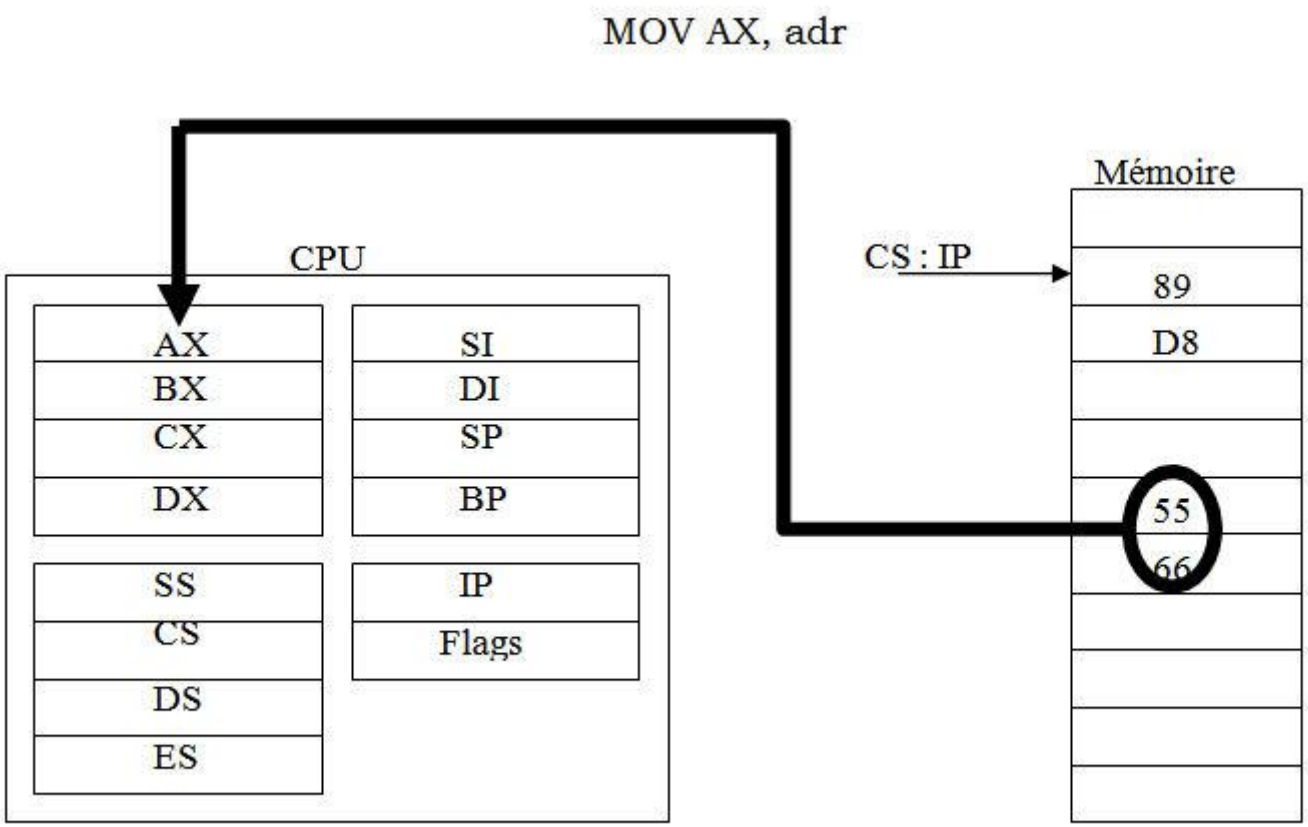
Exemple dans notre cas MOV AX,-500H donne AX =1111101100000000B MOV BL,-20H donne BL = 11100000B

VI -3) Mode d'adressage direct :

Dans ce mode on spécifie directement l'adresse de l'opérande dans l'instruction .exemple :

```
MOV AX,adr
```

La valeur adr est une constante (un déplacement) qui doit être ajouté au contenu du registre DS pour former l'adresse physique de 20 bits.



Remarque :

En général le déplacement est ajouté par défaut avec le registre segment DS pour former l'adresse physique de 20 bits, mais il faut signaler

qu'on peut utiliser ce mode d'adressage avec d'autres registres segment tel

que ES par exemple , seule la syntaxe en mnémonique de l'instruction change et devient :

```
MOV   AX, ES : adr
```

VI -4) Mode d'adressage registre indirect :

Dans ce mode d'adressage l'adresse de l'opérande est stockée dans un registre qu'il faut bien évidemment le charger au préalable par la bonne adresse. L'adresse de l'opérande sera stockée dans un registre de base (BX ou BP) ou un indexe (SI ou DI).

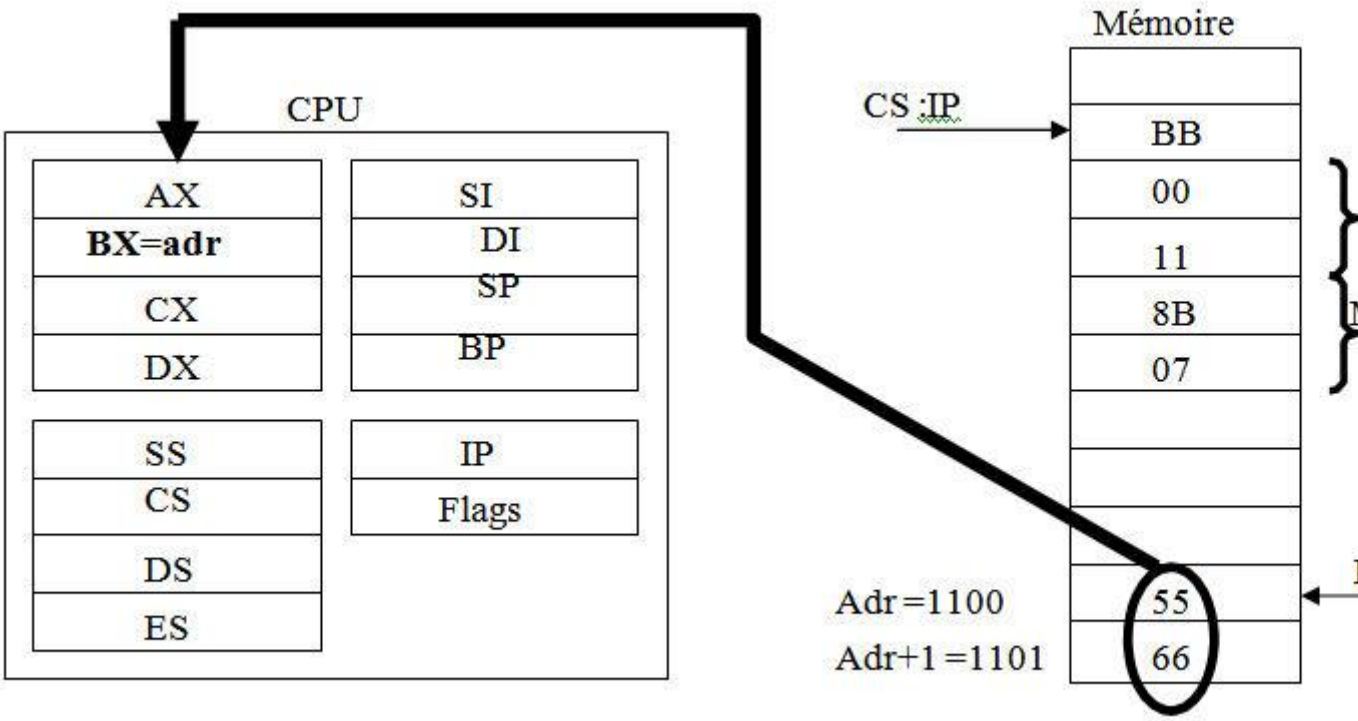
Exemple :

```
MOV  BX,offset adr
MOV  AX,[BX]
```

Le contenu de la case mémoire dont l'adresse se trouve dans le registre BX (c.a.d : Adr) est mis dans le registre AX

Remarque:

Le symbole [ ] design l'adressage indirect.



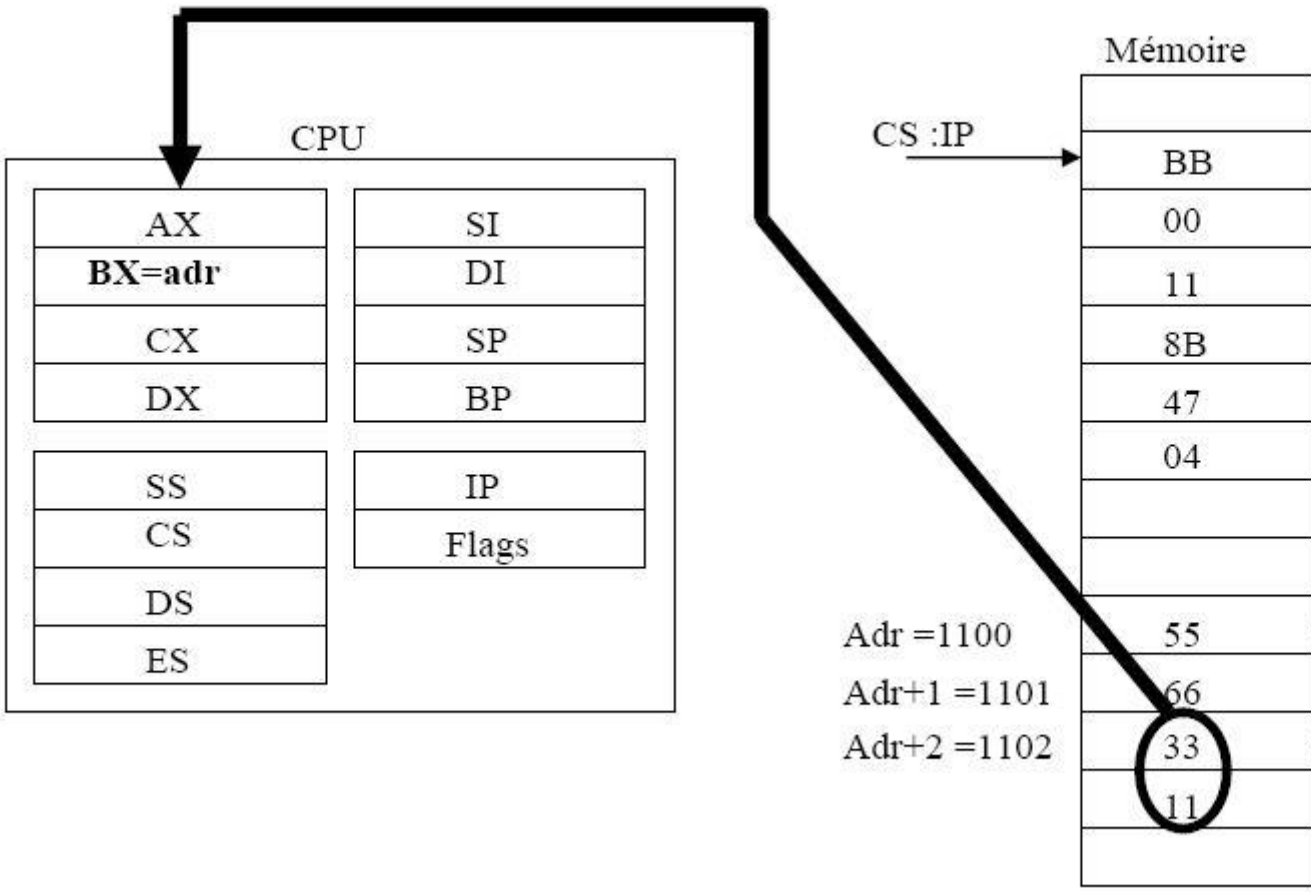
VI - 5) Mode d'adressage relatif à une base :

Dans ce mode d'adressage Le déplacement est déterminé par soi, le contenu de BX, soit le contenu de BP, auquel est éventuellement ajouté un décalage sur 8 ou 16 bits signé. DS et SS sont pris par défaut.

Exemple :

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

Cela signifie que dans le registre AX on va mettre le contenu de la case mémoire pointe par BX+2



Remarque :

Les syntaxes suivantes sont identiques :

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