

# **MICROPROCESSEURS**

## **DE LA FAMILLE**

### **8086**

# Sommaire

<b>1</b>	<b>Structure d'un processeur En général</b>	<b>6</b>
1.1	L'unité de calcul	6
1.2	L'unité de control	7
<b>2</b>	<b>Le microprocesseur 8086</b>	<b>8</b>
2.1	La segmentation de la mémoire	9
2.2	Les registres du 8086	10
2.2.1	Les registres généraux	10
2.2.2	Les registres d'adressage (offset)	11
2.2.3	Les registres de segment	11
2.2.4	Format d'une adresse	12
2.2.5	Le registre d'état (flags)	13
2.3	Les modes d'adressage	14
2.4	Taille des échanges avec la mémoire	16
2.5	Les instructions du 8086	18
2.5.1	Les instructions de transfert	18
2.5.2	Les instructions Arithmétiques	19
2.5.3	Les instructions logiques	22
2.5.4	Les masques logiques :	23
2.5.5	Les instructions de décalage	24
2.5.6	Instructions agissant sur les indicateurs	25
2.5.7	Les instructions de contrôle de boucle	26
2.5.8	Les instructions de branchement	26
2.5.9	Instructions d'accès aux ports d'E/S	29
2.6	Ce qu'il ne faut pas faire	30
<b>3</b>	<b>L'assembleur NASM</b>	<b>33</b>
3.1	Les directives de NASM	33
3.2	Les pseudo instruction de NASM	34
3.3	Les expressions	34
<b>4</b>	<b>Les entrée sorties</b>	<b>35</b>
4.1.1	L'interruption 10h du BIOS	35
4.1.2	L'interruption 21h du DOS	37
4.2	Accès direct à la mémoire Vidéo	39
4.3	les temporisations	40
<b>5</b>	<b>Code machine des instructions</b>	<b>42</b>
5.1	Les codes REG, ADR et MOD	43
5.2	Tableau des codes binaires	43
<b>6</b>	<b>ANNEXE</b>	<b>48</b>
6.1	Instructions d'ajustement décimal	48
6.2	Les instructions de manipulation de chaînes	49
6.3	Instructions de transfert d'adresse	52
6.4	Instructions diverses	52

## Objectif du cours

Dans ce cours on va présenter le Microprocesseur 8086 de Intel, on va étudier son jeu d'instruction complet, on va apprendre à le programmer en assembleur et finir par étudier les codes machines.

- ▶ Pourquoi un cours sur les Microprocesseurs et l'assembleur ? : Parce que c'est la seule façon de comprendre comment fonctionne un ordinateur à l'intérieur. Il devient ainsi beaucoup plus facile de le programmer à l'aide d'autres langages plus évolués comme le Pascal, le C/C++, et les langages visuels.
- ▶ Pourquoi le 8086 d'Intel ? : Parce que la majeure partie des ordinateurs individuels utilisés de nos jours (2007) sont des PCs équipés de microprocesseurs Intel compatibles avec le 8086. C'est-à-dire que tout programme écrit pour tourner sur un 8086 peut être exécuté sur un Pentium 4. Ce qui signifie que si on maîtrise la programmation en assembleur du 8086, on a fait un **grand pas** vers la programmation de nos PC actuels que ce soit en assembleur ou à l'aide d'autres langages plus évolués comme le C/C++.
- ▶ Attention : Le 8086 est un microprocesseur qui était destiné à fonctionner dans des ordinateurs monotâches. C'est-à-dire qui ne peuvent exécuter qu'un seul programme à la fois. Il fonctionnait alors en mode **réel**, c.à.d que le programme en cours d'exécution peut accéder à n'importe quelle ressource de la machine y compris n'importe quelle zone mémoire. Avec les systèmes d'exploitation récents comme Windows ou Linux, les ordinateurs sont devenus multitâches c'est-à-dire que le processeur peut travailler sur plusieurs programmes à la fois. Il devient alors impératif de "réglementer" les accès à la mémoire afin qu'un programme ne puisse pas aller écrire dans une zone mémoire utilisée par un autre programme. Pour cela, Les processeurs actuels fonctionnent en mode **protégé**. Ils interagissent avec le système d'exploitation qui gère les ressources de la machine et évite les conflits entre les programmes qui s'exécutent simultanément.

Pas de panique, dans la plupart des cas, on peut exécuter les programmes destinés au 8086 sur un PC récent sans aucun problème.

## INTRODUCTION

Le "Job" d'un processeur est d'exécuter des **programmes**. Un programme est une suite **d'instructions** écrites une par ligne. Une instruction peut être plus ou moins sophistiquée selon le langage utilisé. Pour un langage de bas niveau comme l'assembleur, une instruction réalise une tâche élémentaire comme une addition par exemple. Avec un langage de haut niveau comme le C++, une instruction peut réaliser un ensemble de tâches qui nécessiterait plusieurs instructions en Assembleur. Il va falloir éclaircir un peu tout ça pour bien comprendre le fonctionnement d'une machine informatique.

Un processeur quel qu'il soit sait exécuter un ensemble bien défini de codes machines (jeux d'instructions). Chaque code machine est un nombre binaire de quelques octets, il correspond à une instruction élémentaire bien définie. Sur papier, on a pris l'habitude de les représenter en hexadécimal pour faciliter.

exécutable ↓	↓	Source ↓
Codes machine tels qu'ils seront présentés au processeur	Codes machine comme on a pris l'habitude de les représenter sur papier	Ecriture plus lisible dite Mnémonique ou Assembleur
10111101 01000001 00000000	BD 41 00	MOV BP,41h
10111110 01100101 01000001	BE 65 41	MOV SI,4165h
00000001 11011000	01 D8	ADD AX,BX
00111000 01000111 00000011	38 47 03	CMP [BX+3],AL

Par exemple, l'instruction (assembleur) *MOV BP,41h* qui signifie : placer le nombre 41h dans le registre BP est codée (en hexadécimal) par les trois octets *BD 41 00*, Dans la suite de ce cours, nous désignerons ces octets par : éléments d'instruction.

Quand on veut écrire un programme, on dispose d'un choix très important de langages de programmation différents les uns des autres : Assembleur, Basic/Qbasic, Pascal, C/C++, Visual Basic, Visual C++, Delphi, Java, ...

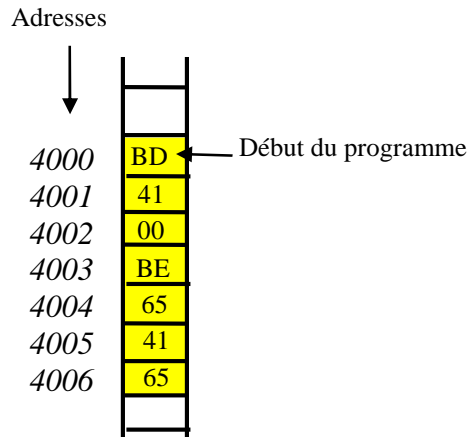
En fait, les lignes de programme que nous écrivons constituent ce qu'on appelle un programme **source** qui sera stocké dans un fichier texte dont l'extension dépend du langage choisi (test.pas pour le pascal, test.cpp pour le c++ etc ...)

Ces programmes sources sont compréhensibles par nous mais pas par le processeur. Pour que le processeur puisse les comprendre il faut les traduire (**compiler**) en langage machine qui est une suite de codes machine. Sur les PCs, se sont les fichiers avec l'extension .exe (test.exe). Chaque langage de programmation a son compilateur qui permet de transformer le programme source en un programme exécutable compréhensible par le processeur. Tous les exécutables se ressemblent et le processeur ne sait pas avec quel langage ils ont été écrits.

Avec un langage de haut niveau comme le C++, une instruction que nous écrivons peut être très sophistiquée. C'est le compilateur C++ qui la traduit en un ensemble d'instructions élémentaires compréhensible par le processeur.

L'intérêt du langage assembleur est que chaque instruction que nous écrivons correspond à une instruction élémentaire du processeur. C'est comme si on travaillait directement en langage machine. Ainsi, on sait exactement tout ce que fait le processeur lors de l'exécution d'un programme.

Quand on demande l'exécution d'un programme, celui-ci est chargé par le système d'exploitation (à partir du disque dur) dans une zone de la mémoire RAM. Celle-ci étant organisée en octets, chaque élément d'instruction est stocké dans une position mémoire. L'adresse (le numéro) de la case mémoire de début est communiquée au processeur pour qu'il commence l'exécution au bon endroit.



# 1 STRUCTURE D'UN PROCESSEUR EN GENERAL

Les différents constituants du Processeur peuvent être regroupés dans deux blocs principaux, l'unité de calcul et l'unité de control.

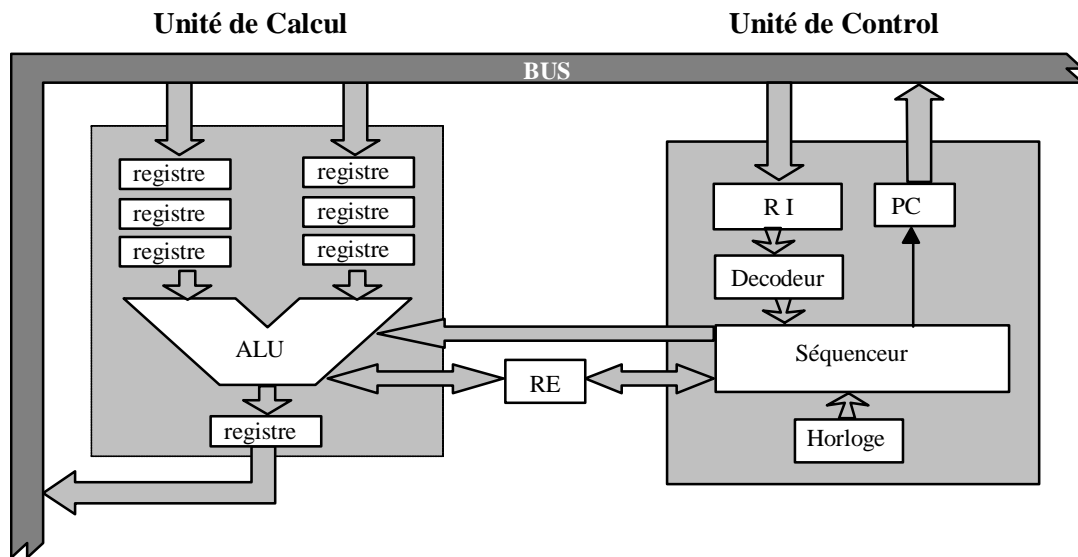


Fig. 1.1 : Architecture simplifiée d'un processeur

## 1.1 L'UNITE DE CALCUL

Elle est constituée de l'Unité Arithmétique et logique UAL et d'un certain nombre de registres

### L'ALU :

Elle est constituée d'un circuit logique combinatoire qui reçoit deux opérandes A ( $A_n \dots A_1 A_0$ ) et B ( $B_n \dots B_1 B_0$ ) et produit le résultat S ( $S_m \dots S_1 S_0$ ) selon l'indication appliquée sur l'entrée C ( $C_k \dots C_1 C_0$ ). Les opérations réalisées peuvent être soit arithmétiques,  $S=A+B$ ,  $S=A-B$ ,  $S=A \times B$  ... ou logiques  $S=A \text{ OU } B$ ,  $S=A \text{ ET } B$ ,  $S= A \text{ XOR } B$  ...

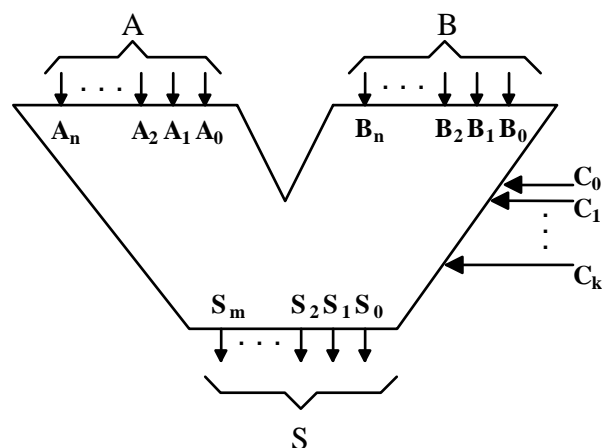


Fig. 1.2 : Unité arithmétique et logique

### Les registres :

Ce sont des mémoires élémentaires pouvant contenir chacun un opérande. Les registres peuvent être de 8, 16 ou 32 bits.

## 1.2 L'UNITE DE CONTROL

C'est l'unité de control qui supervise le déroulement de toutes les opérations au sein du Processeur. Elle est constituée principalement de :

### **l'horloge**

C'est l'horloge qui génère les signaux qui permettent le cadencement et la synchronisation de toutes les opérations. Attention, l'horloge n'est pas une montre au sens commun du terme, c'est juste un signal carré qui a une fréquence fixe (3 Ghz par exemple), a chaque coup (front) d'horloge, le microprocesseur (qui ne l'oublions pas n'est qu'un circuit électronique) réalise une tâche élémentaire. L'exécution d'une instruction nécessite plusieurs coups d'horloges.



Il existe des processeurs qui exécutent une instruction par coup d'horloge, ce n'est pas le cas du 8086.

### **Le compteur programme PC**

Le compteur programme (*PC : program counter*) est un registre (pointeur) qui contient l'adresse de la case mémoire où est stockée le prochain élément d'instruction qui devra être chargé dans le processeur pour être analysé et exécuté. Au début de l'exécution d'un programme, le PC est initialisé par le système d'exploitation à l'adresse mémoire où est stockée la première instruction du programme. Le compteur programme est incrémenté automatiquement chaque fois qu'un élément d'instruction est chargé dans le processeur

### **Le registre d'instruction RI**

C'est là où le CPU stocke l'instruction en cours d'exécution.

### **Le décodeur**

C'est lui qui va "décoder" l'instruction contenue dans RI et générer les signaux logiques correspondant et les communiquer au séquenceur.

### **Le séquenceur**

Il gère le séquençage des opérations et génère les signaux de commande qui vont activer tous les éléments qui participeront à l'exécution de l'instruction et spécialement l'ALU.

### **Le registre d'état**

Le registre d'état est formé de plusieurs bits appelés drapeaux ou indicateurs (*Flags*) qui sont positionnés par l'ALU après chaque opération. Par exemple l'indicateur Z indique quand il est positionné que le résultat de l'opération est égal à Zéro. L'indicateur C indique que l'opération a généré une retenue. Le bit N indique que le résultat est négatif ...

On dispose d'un jeu d'instructions conditionnées par l'état de différents drapeaux

## 2 LE MICROPROCESSEUR 8086

Disponible depuis 1987, le 8086 fut le premier microprocesseur 16 bits fabriqué par Intel. Parmi ses caractéristiques principales, on peut citer :

- ▶ Il se présente sous forme d'un boîtier de 40 broches alimenté par une alimentation unique de 5V.
- ▶ Il possède un bus multiplexé adresse/donnée de 20 bits.
- ▶ Le bus de donnée occupe 16 bits ce qui permet d'échanger des mots de 2 octets
- ▶ Le bus d'adresse occupe 20 bits ce qui permet d'adresser 1 M octets ( $2^{20}$ )
- ▶ Il est entièrement compatible avec le 8088, le jeu d'instruction est identique. La seule différence réside dans la taille du bus de données, celui du 8088 fait seulement 8 bits. Les programmes tourneront donc un peu plus lentement sur ce dernier puisqu'il doit échanger les mots de 16 bits en deux étapes.
- ▶ Tous les registres sont de 16 bits, mais pour garder la compatibilité avec le 8085/8088, certains registres sont découpés en deux et on peut accéder séparément à la partie haute et à la partie basse.

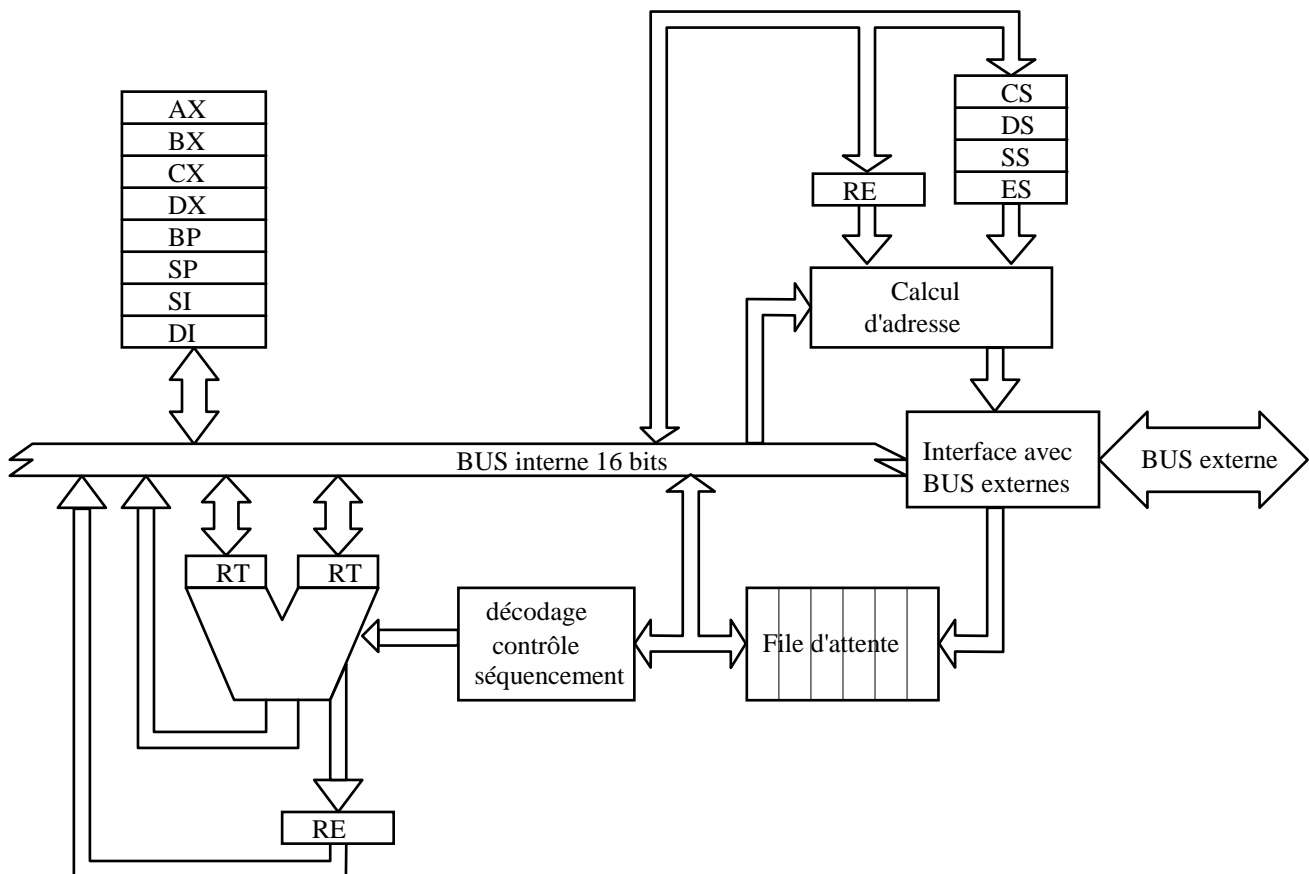


Fig. 2.1 : Synoptique fonctionnel du 8086



## 2.1 LA SEGMENTATION DE LA MEMOIRE

Le 8086 possède 20 bits d'adresse, il peut donc adresser  $2^{20}$  octets soit 1 Mo. L'adresse de la première case mémoire est 0000 0000 0000 0000 0000 celle de la dernière case est 1111 1111 1111 1111 1111 1111. Il me paraît inutile de justifier pourquoi à partir de cet instant, nous allons représenter les adresses en hexadécimal, et notre 8086 peut donc adresser 1 Mo allant de 00000 à FFFFF. Le problème qui se pose est comment représenter ces adresses au sein du  $\mu P$  puisque les registres ne font que 16 bits soit 4 digits au maximum en hexadécimal. La solution adoptée par Intel a été la suivante :

Puisque avec 16 bits on peut adresser  $2^{16}$  octets = 65535 octets = 64 ko, La mémoire totale adressable de 1 Mo est fractionnée en pages de 64 ko appelés segments. On utilise alors deux registres pour adresser une case mémoire donnée, Un registre pour adresser le segment qu'on appelle registre segment et un registre pour adresser à l'intérieur du segment qu'on désignera par registre d'adressage ou offset. Une adresse se présente toujours sous la forme **segment:offset**

A titre d'exemple, procédons au découpage de la mémoire en 16 segments qui ne se chevauche pas.

Segment	Adresse début	Adresse fin	Pointeur de segment
Segment 0	00000	0FFFF	00000
Segment 1	10000	1FFFF	10000
Segment 2	20000	2FFFF	20000
Segment 14	E0000	EFFFF	E0000
Segment 15	F0000	FFFFFF	F0000

Considérons la case mémoire d'adresse 20350, appelée adresse absolue ou adresse linéaire. Cette case mémoire se situe dans le segment 2, son adresse relative à ce segment est 350, on peut donc la référencer par le couple segment:offset = 20000:350,

Se pose maintenant le problème de la représentation de cette adresse au sein du CPU car les registres de 16 bits ne peuvent contenir que 4 digits. S'il n'y a aucun problème pour représenter 350 dans un registre d'offset, on ne peut pas représenter 20000 dans un registre segment. La solution adoptée par Intel est la suivante :

- ▶ Dans le registre segment, on écrit l'adresse segment sans le chiffre de faible poids
- ▶ Dans le registre d'adressage (d'offset) on écrit l'adresse relative dans le segment
- ▶ Pour calculer l'adresse absolue qui sera envoyée sur le bus d'adresse de 20 bits, le CPU procède à l'addition des deux registres après avoir décalé le registre segment d'un chiffre à gauche :

$$\begin{array}{r}
 \boxed{X} \boxed{X} \boxed{X} \boxed{X} \quad \mathbf{0} \quad \text{Segment} \\
 + \quad \boxed{X} \boxed{X} \boxed{X} \boxed{X} \quad \text{Offset} \\
 \hline
 \boxed{X} \boxed{X} \boxed{X} \boxed{X} \boxed{X} \quad \text{Adresse absolue}
 \end{array}$$

Fig. 2.2 : calcul d'adresse dans un 8086

Dans notre exemple, l'adresse de la case mémoire considérée devient 2000:350 soit :  
 Segment = 2000  
 Offset = 350

L'adresse absolue est calculée ainsi :

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 2 & 0 & 0 & 0 \\ \hline \end{array} & \text{Segment} \\
 + & \begin{array}{|c|c|c|c|} \hline & 3 & 5 & 0 \\ \hline \end{array} & \text{Offset} \\
 \hline
 \begin{array}{|c|c|c|c|c|} \hline 2 & 0 & 3 & 5 & 0 \\ \hline \end{array} & \text{Adresse absolue}
 \end{array}$$

Fig. 2.3 : exemple de calcul d'adresse

### Remarque :

Les zones réservées aux segments ne sont pas exclusives, elles peuvent se chevaucher. La seule règle à respecter lors du choix d'un segment est que le digit de plus faible poids soit nul. Nous pouvons donc commencer un segment tous les 16 octets.

Adresse absolue valide pour début de segment	segment
00000	0000
00010	0001
00020	0002
...	...

### Exercice 1)

1. Donner les adresses linéaires (absolues) des mémoires 3500:AB00, 0022:FFFF,
2. Proposer au moins deux adresses *segment:offset* différentes pour les mémoires d'adresse linéaire 10000, FFFFF, 00000

## 2.2 LES REGISTRES DU 8086

	Registres généraux	Registres d'adressage	Registres de segment	Registres de commande					
AX	<table border="1"><tr><td>AH</td><td>AL</td></tr></table>	AH	AL	<table border="1"><tr><td>SP</td></tr></table>	SP	<table border="1"><tr><td>CS</td></tr></table>	CS	<table border="1"><tr><td>IP</td></tr></table>	IP
AH	AL								
SP									
CS									
IP									
BX	<table border="1"><tr><td>BH</td><td>BL</td></tr></table>	BH	BL	<table border="1"><tr><td>BP</td></tr></table>	BP	<table border="1"><tr><td>DS</td></tr></table>	DS	<table border="1"><tr><td>FLAGS</td></tr></table>	FLAGS
BH	BL								
BP									
DS									
FLAGS									
CX	<table border="1"><tr><td>CH</td><td>CL</td></tr></table>	CH	CL	<table border="1"><tr><td>SI</td></tr></table>	SI	<table border="1"><tr><td>SS</td></tr></table>	SS		
CH	CL								
SI									
SS									
DX	<table border="1"><tr><td>DH</td><td>DL</td></tr></table>	DH	DL	<table border="1"><tr><td>DI</td></tr></table>	DI	<table border="1"><tr><td>ES</td></tr></table>	ES		
DH	DL								
DI									
ES									

Fig. 2.4 : les registres du 8086

Tous les registres et le bus interne du 8086 sont structurés en 16 bits.

Vu de l'utilisateur, le 8086 comprend 3 groupes de 4 registres de 16 bits, un registre d'état de 9 bits et un compteur programme de 16 bits non accessible par l'utilisateur.

### 2.2.1 Les registres généraux

Les registres généraux participent aux opérations arithmétiques et logiques ainsi qu'à l'adressage. Chaque demi-registre est accessible comme registre de 8 bits, le 8086 peut donc effectuer des opérations 8 bits d'une façon compatible avec le 8080.

**AX** : Accumulateur

- Usage général,
- Obligatoire pour la multiplication et la division,
- Ne peut pas servir pour l'adressage

**BX** : Base

- Usage général,
- Adressage, (Par défaut, son offset est relatif au segment DS)

**CX** : Comptage et calcul

- Usage général,
- Utilisé par certaines instruction comme compteur de répétition.
- Ne peut pas servir pour l'adressage

**DX** : Data

- Usage général,
- Dans la multiplication et la division 16 bits, il sert comme extension au registre AX pour contenir un nombre 32 bits,
- Ne peut pas servir pour l'adressage

### 2.2.2 Les registres d'adressage (offset)

Ces registres de 16 bits permettent l'adressage d'un opérande à l'intérieur d'un segment de 64 ko ( $2^{16}$  positions mémoires)

**SP** : Pointeur de Pile

- Utilisé pour l'accès à la pile. Pointe sur la tête de la pile.
- Par défaut, son offset est relatif à SS

**BP** : Pointeur de Base

- Adressage comme registre de base, (Par défaut, son offset est relatif à SS)
- Usage général

**SI** : Registre d'index (source)

- Adressage comme registre d'index, (Par défaut, son offset est relatif à DS)
- Certaines instruction de déplacement de données l'utilise comme index de l'opérande source. L'opérande destination étant indexé par DI
- Usage général

**DI** : Registre d'index (destination)

- Adressage comme registre d'index, (par défaut, son offset est relatif à DS)
- Certaines instruction de déplacement de données l'utilise comme index de l'opérande destination, l'opérande destination étant indexé par SI

### 2.2.3 Les registres de segment

Ces registres sont combinés avec les registres d'offset pour former les adresses. Une case mémoire est repérée par une adresse de la forme RS:RO. On place le registre segment au début d'une zone mémoire de 64Ko, ensuite on fait varier le registre d'offset qui précise l'adresse relative par rapport à cette position.

**CS : Code Segment**

Définit le début de la mémoire programme. Les adresses des différentes instructions du programme sont relatives à CS

**DS : Data Segment**

Début de la mémoire de données dans laquelle sont stockées toutes les données traitées par le programme

**SS : Stack Segment**

Début de la pile.

La pile est une zone mémoire gérée d'une façon particulière. Elle est organisée comme une pile d'assiettes. On pose et on retire les assiettes toujours sur le haut de la pile. Un seul registre d'adresse suffit donc pour la gérer, c'est le *stack pointer* SP. On dit que c'est une pile LIFO (Last IN, First Out).

Empiler une donnée : sauvegarder une donnée sur (le sommet) de la pile

Dépiler une donnée : retirer une donnée (du sommet) de la pile

**ES : Extra Segment**

Début d'un segment auxiliaire pour données

**2.2.4 Format d'une adresse**

Une adresse doit avoir la forme [Rs : Ro] avec les possibilités

Rien	Valeur
DS	BX
ES	BP
CS	SI
ES	DI

Si le registre segment n'est pas spécifié (cas rien), alors le processeur l'ajoute par défaut selon l'offset choisit :

Offset utilisé	Valeur	DI	SI	BX	BP
Registre Segment par défaut qui sera utilisé par le CPU		DS			SS

Tableau 2.1 : segment par défaut

Dans la suite de ce cours, On accèdera souvent à la mémoire en précisant seulement la partie offset de l'adresse. Par défaut on considère qu'on est dans le segment DATA

## 2.2.5 Le registre d'état (flags)



Six bits reflètent les résultats d'une opération arithmétique ou logique et 3 participent au control du processeur.

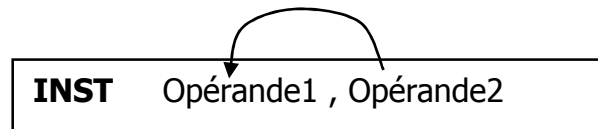
- ▶ **C** : (Carry) indique le dépassement de capacité de 1 sur une opération 8 bits ou 16 bits. Ce flag peut être utilisé par des instructions de saut conditionnel, des calculs arithmétique en chaîne ou dans des opération de rotation.
- ▶ **P** : (Parité) indique que le nombre de 1 est un nombre pair. Ce flag est utilisé avec certains sauts conditionnels.
- ▶ **A** : (retenue **A**rithmétique) indique une retenue sur les 4 bits (digit) de poids faible. Par exemple quand la somme des 2 digits de poids faible dépasse F (15)
- ▶ **Z** : (Zéro) Indique que le résultat d'une opération arithmétique ou logique est nul. Il est utilisé dans plusieurs instructions de sauts conditionnels.
- ▶ **S** : (Signe) reproduit le bit de poids fort d'une quantité signée sur 8 bits ou sur 16 bits. L'arithmétique signée fonctionne en complément à 2. S=0 : positif, S=1 : négatif. Ce flag sert lors de sauts conditionnels.
- ▶ **T** : (Trap) met le CPU en mode pas à pas pour faciliter la recherche des défauts d'exécution.
- ▶ **I** : (Interruption) autorise ou non la reconnaissance des interruptions  
 I = 0 → Interruptions autorisées  
 I = 1 → Interruptions non autorisées
- ▶ **D** : (Direction) fixe la direction de l'auto-inc/décrémentation de SI et DI lors des instruction de manipulation de chaînes.  
 D = 0 → Incrémentation des index  
 D = 1 → décrémentation des index
- ▶ **O** : (*Overflow*) indique un dépassement de capacité quand on travaille avec des nombres signés. Comme par exemple si la somme de 2 nombres positifs donne un nombre négatif ou inversement. (40h + 40h = 80h et O=1)

## 2.3 LES MODES D'ADRESSAGE

Dans la suite on utilisera les abréviations suivantes :

INST : instruction,  
 R : Registre quelconque,  
 Rseg : Registre Segment  
 Roff : Registre d'offset  
 Adr : Adresse  
 [ adr ] : contenu Mémoire  
 Off : Offset de l'adresse  
 Im : donnée (constante)  
 Dep : déplacement (constante)  
 Op : Opérande  
 Os : Opérande source  
 Od : opérande destination

La structure la plus générale d'une instruction est la suivante :



L'opération est réalisée entre les 2 opérandes et le résultat est toujours récupéré dans l'opérande de gauche.

Il y a aussi des instructions qui agissent sur un seul opérande

Les opérandes peuvent être des registres, des constantes ou le contenu de cases mémoire, on appelle ça le mode d'adressage

### ➔ Adressage registre (R)

L'opération se fait sur un ou 2 registres

INST R , R

INST R

*Exemples :*

INC AX : incrémenter le registre AX

MOV AX, BX : Copier le contenu de BX dans AX

### ➔ Adressage Immédiat (IM)

Un des opérande est une constante (valeur) :

INST R , im

INST taille [adr] , im

*Exemples :*

MOV AX, 243 : charger le registre AX par le nombre décimal 243

ADD AX, 243h : additionner le registre AX avec le nombre hexadécimal 243

MOV AX, 0xA243 : Quand le chiffre de gauche du nombre hexadécimal est une lettre, il est préférable d'utiliser le préfix 0x pour l'hexadécimal

MOV AL, 'a' : Charger le registre AL par le code ASCII du caractère 'a'  
 MOV AX, 'a' : Charger le registre AH par 00 et le registre AL par le code ASCII du caractère 'a'  
 MOV AX,'ab' : Charger AH par 'a' et AL par 'b'

### ➔ Adressage direct (DA)

Un des deux opérands se trouve en **mémoire**. L'adresse de la case mémoire ou plus précisément son Offset est précisé directement dans l'instruction. L'adresse Rseg:Off doit être placée entre [ ], si le segment n'est pas précisé, DS est pris par défaut,

INST R , [adr]  
 INST [adr] , R  
 INST taille [adr] , im

*Exemples :*

MOV AX,[243] : Copier le contenu de la mémoire d'adresse DS:243 dans AX  
 MOV [123],AX : Copier le contenu de AX dans la mémoire d'adresse DS:123  
 MOV AX, [SS:243] : Copier le contenu de la mémoire SS:243 dans AX

### ➔ Adressage indirect (IR)

Un des deux opérands se trouve en **mémoire**. L'offset de l'adresse n'est pas précisé directement dans l'instruction, il se trouve dans l'un des 4 registres d'offset BX, BP, SI ou DI et c'est le registre qui sera précisé dans l'instruction : [Rseg : Roff]. Si Rseg n'est pas spécifié, le segment par défaut sera utilisé (voir Tableau 2.1)

INST R , [Rseg : Roff]  
 INST [Rseg : Roff] , R  
 INST taille [Rseg : Roff] , im

*Exemples :*

MOV AX, [BX] ; Charger AX par le contenu de la mémoire d'adresse DS:BX  
 MOV AX, [BP] ; Charger AX par le contenu de la mémoire d'adresse SS:BP  
 MOV AX, [SI] ; Charger AX par le contenu de la mémoire d'adresse DS:SI  
 MOV AX, [DI] ; Charger AX par le contenu de la mémoire d'adresse DS:DI  
 MOV AX, [ES:BP] ; Charger AX par le contenu de la mémoire d'adresse ES:BP

L'adressage indirect est divisé en 3 catégories selon le registre d'offset utilisé. On distingue ainsi, l'adressage Basé, l'adressage indexé et l'adressage basé indexé,

### ▶ Adressage Basé (BA)

L'offset se trouve dans l'un des deux registres de base BX ou BP. On peut préciser un déplacement qui sera ajouté au contenu de Roff pour déterminer l'offset,

INST R , [Rseg : Rb+dep]  
 INST [Rseg : Rb+dep] , R  
 INST taille [Rseg : Rb+dep] , im

*Exemples :*

MOV AX, [BX] : Charger AX par le contenu de la mémoire d'adresse **DS**:BX  
 MOV AX, [BX+5] : Charger AX par le contenu de la mémoire d'adresse **DS**:BX+5  
 MOV AX, [BP-200] : Charger AX par le contenu de la mémoire d'adresse **SS**:BP-200  
 MOV AX, [ES:BP] : Charger AX par le contenu de la mémoire d'adresse **ES**:BP

### → Adressage Indexé (X)

L'offset se trouve dans l'un des deux registres d'index SI ou DI. On peut préciser un déplacement qui sera ajouté au contenu de Ri pour déterminer l'offset,

INST R , [Rseg : Ri+dep]  
 INST [Rseg : Ri+dep] , R  
 INST taille [Rseg : Ri+dep] , im

*Exemples :*

MOV AX, [SI] ; Charger AX par le contenu de la mémoire d'adresse **DS**:SI  
 MOV AX, [SI+500] ; Charger AX par la mémoire d'adresse **DS**:SI+500  
 MOV AX, [DI-8] ; Charger AX par la mémoire d'adresse **DS**:DI-8  
 MOV AX, [ES:SI+4] ; Charger AX par la mémoire d'adresse **ES**:SI+4

### → Adressage Basé Indexé (BXI)

L'offset de l'adresse de l'opérande est la somme d'un registre de base, d'un registre d'index et d'un déplacement optionnel.

Si Rseg n'est pas spécifié, le segment par défaut du registre **de base** est utilisé :

INST R , [Rseg : Rb+Ri+dep]  
 INST [Rseg : Rb+Ri+dep] , R  
 INST taille [Rseg : Rb+Ri+dep] , im

*Exemples :*

MOV AX,[BX+SI] ; AX est chargé par la mémoire d'adresse **DS**:BX+SI  
 MOV AX,[BX+DI+5] ; AX est chargé par la mémoire d'adresse **DS**:BX+DI+5  
 MOV AX,[BP+SI-8] ; AX est chargé par la mémoire d'adresse **SS**:BP+SI-8  
 MOV AX,[BP+DI] ; AX est chargé par la mémoire d'adresse **SS**:BP+DI

## 2.4 TAILLE DES ECHANGES AVEC LA MEMOIRE

La mémoire est organisée en octets.

Quand on fait une instruction entre un registre et une donnée qui se trouve en mémoire, c'est le registre qui détermine la taille de l'opération.

Si le registre est un registre simple (8 bits), l'opération se fera avec une seule case mémoire.

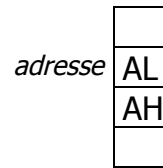
MOV [adresse], AL donne → 

AL



Si le registre est un registre double (2 octets), l'opération se fera avec deux cases mémoires

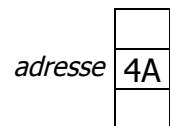
MOV [adresse], AX donne →



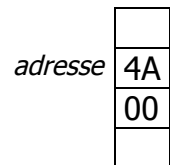
Remarquer que c'est la partie basse du registre qui est traitée en premier, et ceci dans les deux sens

Quand on fait une opération entre une **constante** et une **case mémoire**, il y a ambiguïté, le processeur ne sait pas s'il faut considérer la constante sur 8 bits ou sur 16 bits. Il faut utiliser les préfixes BYTE et WORD pour préciser le nombre d'octets à écrire :

MOV BYTE [adresse],4Ah ; On écrit 4A dans la position adresse



MOV WORD [adresse],4Ah ; On écrit 004A, 4A → adresse, et 00 → adresse+1



**Exercice 2) (ram.asm) : tracer le programme ci-dessous**

```

mov bx,4000h      ; adresse
mov ax,2233h
mov [bx],ax
mov word [bx+2],4455h
mov byte [bx+4],66
mov byte [bx+5],77

```

*Afficher la ram en hexadécimal à partir de la position 4000H*

*Afficher la ram en décimal à partir de la position 4000H*

## 2.5 LES INSTRUCTIONS DU 8086

### 2.5.1 Les instructions de transfert

#### MOV Od , Os

Copie l'opérande Source dans l'opérande Destination

MOV R1 , R2	<i>copier un registre dans un autre</i>
MOV R , M	<i>copier le contenu d'une case mémoire dans un registre</i>
MOV M , R	<i>copier un registre dans une case mémoire</i>
MOV R , im	<i>copier une constante dans un registre</i>
MOV taille M , im	<i>copier une constante dans une case mémoire (taille = BYTE ou WORD)</i>

~~MOV M , M~~

#### PUSH Op

Empiler l'opérande Op (Op doit être un opérande 16 bits)

- Décrémente SP de 2
- Copie Op dans la mémoire pointée par SP

PUSH R<sub>16</sub>  
PUSH word [adr]

~~PUSH im~~

~~PUSH R<sub>8</sub>~~

#### POP Op

Dépiler dans l'opérande Op (Op doit être un opérande 16 bits)

- Copie les deux cases mémoire pointée par SP dans l'opérande Op
- Incrémente SP de 2

POP R<sub>16</sub>  
POP word M

~~POP R<sub>8</sub>~~

L'exemple de la figure 2.4 illustre plusieurs situations :

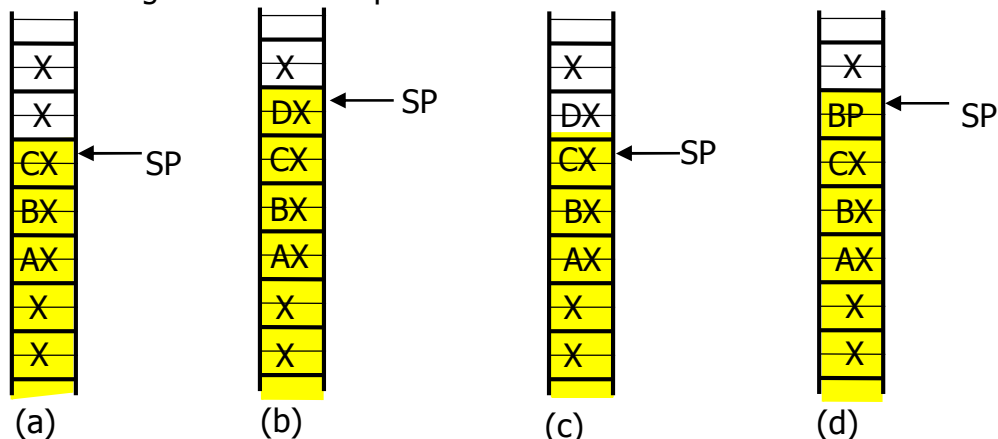


Fig. 2.5 : fonctionnement d'une pile

- (a) Situation de la pile après empilement des registres AX, BX et CX
- (b) Situation de la pile après empilement du registre DX
- (c) Situation de la pile après un dépilement. On remarquera que DX reste dans la mémoire mais s'il ne fait plus partie de la pile. La pile s'arrête à son sommet qui est pointé par le pointeur de pile.
- (d) Situation de la pile après empilement du registre BP. On remarque que la valeur BP écrase la valeur DX dans la mémoire.

**Exercice 3)** : (pile.asm) : tracer le programme ci-dessous. La valeur initiale de SP est quelconque

```

mov ax,2233h
push ax
mov ax,4455h
push ax
mov ax,6677h
push ax
mov bp,sp
mov al,[bp+3]
mov bx,[bp+1]

```

### PUSHA

Empile tous les registres généraux et d'adressage dans l'ordre suivant :  
*AX, CX, DX, BX, SP, BP, SI, DI*

### POPA

Dépille tous les registres généraux et d'adressage dans l'ordre inverse de PUSHA afin que chaque registre retrouve sa valeur. La valeur dépilée de SP est ignorée

*Remarque* : Les deux instructions PUSHA et POPA ne sont pas reconnues par le 8086. Elles ont été introduites à partir du 80186. Nous avons jugé bon de les introduire dans ce cours car elles sont très utiles et comme nous travaillons sur des Pentium, on peut les utiliser sans problème.

### XCHG OD, OS

Echange l'opérande Source avec l'opérande Destination. Impossible sur segment.

```

XCHG R1, R2
XCHG [adr], R
XCHG R, [adr]

```

~~XCHG [ ], [ ]~~

## 2.5.2 Les instructions Arithmétiques

Le 8086 permet d'effectuer les Quatre opérations arithmétiques de base, l'addition, la soustraction, la multiplication et la division. Les opérations peuvent s'effectuer sur des nombres de 8 bits ou de 16 bits signés ou non signés. Les nombres signés sont représentés en complément à 2. Des instructions d'ajustement décimal permette de faire

des calculs en décimal (BCD).

### ► Addition :

**ADD Od , Os** Additionne l'opérande source et l'opérande destination avec résultat dans l'opérande destination,

$$Od + Os \rightarrow Od$$

ADD AX,123                      ADD AX,BX                      ADD [123],AX                      ADD BX,[SI]

**ADC Od , Os** Additionne l'opérande source, l'opérande destination et le carry avec résultat dans l'opérande destination,

$$Od + Os + C \rightarrow Od$$

**INC Op** Incrémente l'opérande Op

$$Op + 1 \rightarrow Op$$

Attention, l'indicateur C n'est pas positionné quand il y a débordement, C'est l'indicateur Z qui permet de détecter le débordement

Pour incrémenter une case mémoire, il faut préciser la taille :

INC byte [ ]

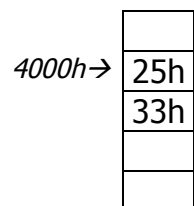
INC word [ ]

### **Exercice 4) :**

En partant à chaque fois de la situation illustrée à droite. Quelle est la situation après chacune des instructions suivantes

INC byte [4000h]

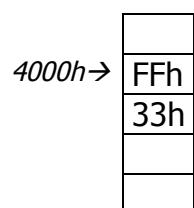
INC word [4000h]



En partant à chaque fois de la situation illustrée à droite. Quelle est la situation après chacune des instructions suivantes

INC byte [4000h]

INC word [4000h]



### ► Soustraction

**SUB Od , Os** Soustrait l'opérande source et l'opérande destination avec résultat dans l'opérande destination.

$$Od - Os \rightarrow Od$$

**SBB Od , Os** Soustrait l'opérande source et le carry de l'opérande destination avec résultat dans l'opérande destination.

$$Od - Os - C \rightarrow Od$$

**DEC Op** Décrémente l'opérande Op  
 $Op - 1 \rightarrow Op$

**NEG Op** Donne le complément à 2 de l'opérande Op : remplace Op par son négatif  
 $\text{Cà2}(Op) \rightarrow Op$

**CMP Od , Os** Compare (soustrait) les opérandes Os et Od et positionne les drapeaux en fonction du résultat. L'opérande Od n'est pas modifié

### ► Multiplication

**MUL Op** instruction à un seul opérande. Elle effectue une multiplication non signée entre l'accumulateur (AL ou AX) et l'opérande Op. Le résultat de taille double est stocké dans l'accumulateur et son extension (AH:AL ou DX:AX).

MUL Op<sub>8</sub>            →    AL x Op<sub>8</sub>    →    AX  
 MUL Op<sub>16</sub>           →    AX x Op<sub>16</sub> →    DX:AX

- L'opérande Op ne peut pas être une donnée, c'est soit un registre soit une position mémoire, dans ce dernier cas, il faut préciser la taille (byte ou word)

MUL BL            ;    AL x BL    →    AX  
 MUL CX           ;    AX x CX    →    DX:AX  
 MUL byte [BX] ;    AL x (octet pointé par BX) →    AX  
 MUL word [BX] ;    AX x (word pointé par BX) →    DX :AX

~~MUL im~~

~~MUL AX,R~~

~~MUL AX, im~~

- Les drapeaux C et O sont positionnés si la partie haute du résultat est non nulle. La partie haute est AH pour la multiplication 8 bits et DX pour la multiplication 16 bits

**Exercice 5) :** (*mul.asm*) Tracer le programme ci-dessous en indiquant à chaque fois la valeur des indicateurs C et O

```
mov al,64h
mov bl,2
mul bl
mov al,64h
mov cl,3
mul cl
```

**IMUL Op** (*Integer Multiply*) Identique à MUL excepté qu'une multiplication signée est effectuée.

**Exercice 6)** (*imul.asm*) : différence entre MUL et IMUL  
 Tracer le programme ci-dessous

```
MOV    al,11111111b
mov    bl,00000010b
mul    bl
MOV    al,11111111b
mov    bl,00000010b
imul   bl
```

## ► Division

**DIV Op** Effectue la division  $AX/Op_8$  ou  $(DX|AX)/Op_{16}$  selon la taille de Op qui doit être soit un registre soit une mémoire. Dans le dernier cas il faut préciser la taille de l'opérande, exemple : *DIV byte [adresse]* ou *DIV word [adresse]*.

AX		Op <sub>8</sub>
AH		AL

DIV Op<sub>8</sub> ;AX / Op<sub>8</sub> , Quotient → AL , Reste → AH

DIV S<sub>16</sub> ;DX:AX / S<sub>16</sub> , Quotient → AX , Reste → DX

DX:AX		Op <sub>16</sub>
DX		AX

- S ne peut pas être une donnée (immédiat) ~~DIV 125h~~
- Après la division L'état des indicateurs est indéfini
- La division par 0 déclenche une erreur

**IDIV Op** Identique à DIV mais effectue une division signée

**CBW** (*Convert Byte to Word*) Effectue une extension de AL dans AH. On écrit le contenu de AL dans AX en respectant le signe

Si AL contient un nombre positif, On complète par des 0 pour obtenir la représentation sur 16 bits.

Si AL contient un nombre négatif, On complète par des 1 pour obtenir la représentation sur 16 bits.

+5 = 0000 0101 ⇒ 0000 0000 0000 0101

5 = 1111 1011 ⇒ 1111 1111 1111 1011

**CWD** (*Convert Word to Double Word*) effectue une extension de AX dans DX en respectant le signe. On écrit AX dans le registre 32 bits obtenu en collant DX et AX DX | AX

### 2.5.3 Les instructions logiques

**NOT Op** Complément à 1 de l'opérande Op

**AND Od , Os** ET logique  
Od ET Os → Od

**OR Od , Os** OU logique  
Od OU Os → Od

**XOR Od , Os** OU exclusif logique  
 Od OUX Os  $\rightarrow$  Od

**TEST Od , Os** Similaire à AND mais ne retourne pas de résultat dans Od, seuls les indicateurs sont positionnés

**Exercice 7)** : (logic.asm) Tracer en binaire le programme ci-dessous

```
MOV    AX,125h
AND    AL,AH
```

**Exercice 8)** : (xor.asm) Programme qui fait  $AX \oplus BX$  sans utiliser l'instruction XOR

### 2.5.4 Les masques logiques :

Le 8086 ne possède pas d'instructions permettant d'agir sur un seul bit. Les masques logiques sont des astuces qui permettent d'utiliser les instructions logiques vues ci-dessus pour agir sur un bit spécifique d'un octet out d'un word

#### Forcer un bit à 0 :

Pour forcer un bit à 0 sans modifier les autres bits, on utilise l'opérateur logique AND et ces propriétés :

- $x \text{ AND } 0 = 0$  ( $0 = \text{élément absorbant de AND}$ )
- $x \text{ AND } 1 = x$  ( $1 = \text{élément neutre de AND}$ )

On fait un AND avec une valeur contenant des 0 en face des bits qu'il faut forcer à 0 et des 1 en face des bits qu'il ne faut pas changer

	XXXX	XXXX
AND	1110	1101
<hr/>		
	xxx0	xx0x

#### Forcer un bit à 1 :

Pour forcer un bit à 1 sans modifier les autres bits, on utilise l'opérateur logique OR et ces propriétés :

- $x \text{ OR } 1 = 1$  ( $1 = \text{élément absorbant de OR}$ )
- $x \text{ OR } 0 = x$  ( $0 = \text{élément neutre de OR}$ )

On fait un OR avec une valeur contenant des 1 en face des bits qu'il faut forcer à 1 et des 0 en face des bits qu'il ne faut pas changer

	XXXX	XXXX
OR	0010	0000
<hr/>		
	xx1x	xxxx

#### Inverser un bit :

Pour inverser la valeur d'un bit sans modifier les autres bits, on utilise l'opérateur logique XOR et ces propriétés :

- $X \text{ XOR } 1 = \bar{X}$
- $X \text{ XOR } 0 = X$  ( $0 = \text{élément neutre de XOR}$ )

Donc, on fait un XOR avec une valeur contenant des 1 en face des bits qu'il faut inverser et des 0 en face des bits qu'il ne faut pas changer

	XXXX	XXXX
XOR	0010	0000
<hr/>		
	xx $\bar{x}$ x	xxxx

**Exercice 9)** : (masque.asm) Tracer le programme ci-dessous

```
MOV    AX,1A25h
AND    AX,F0FFh
```

### 2.5.5 Les instructions de décalage

Dans les instructions de décalage, l'opérande k peut être soit une constante (immédiat) soit le registre CL :

INST AX,1 ; décaler AX de 1 bit

INST BL,4 ; décaler BL de 4 bits

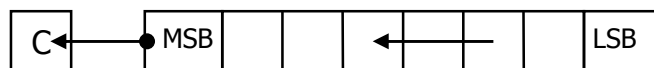
INST BX,CL ; décaler BX de CL bits

On peut aussi décaler le contenu d'une case mémoire mais il faut préciser la taille

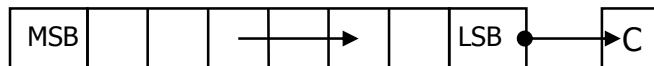
INST byte [BX],1 ; décaler une fois le contenu de la case mémoire d'adresse BX

**SHL R/M,k** (*Shift Logical Left*) décalage logique à gauche de k bits

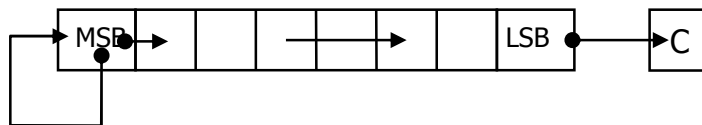
**SAL R/M,k** (*Shift Arithmé Left*) décalage arithmétique à gauche



**SHR R/M,k** (*Shift Logical right*) décalage logique à droite

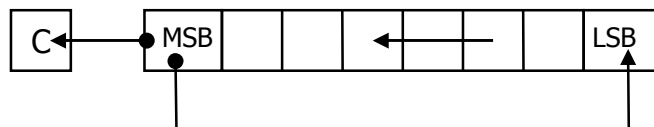


**SAR R/M,k** (*Shift Arithmetic right*) décalage arithmétique à droite

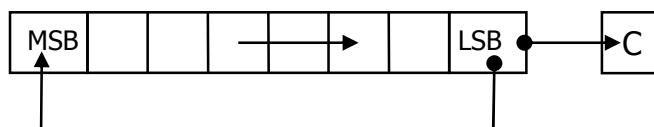


Les décalages arithmétiques permettent de conserver le signe. Ils sont utilisés pour effectuer des opérations arithmétiques comme des multiplications et des divisions par 2.

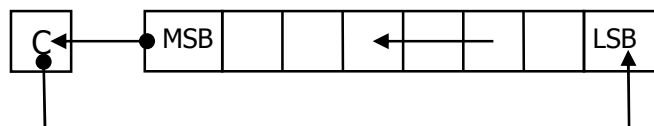
**ROL R/M,k** (*Rotate Left*) Rotation à gauche



**ROR R/M,k** (*Rotate Right*) Rotation à droite

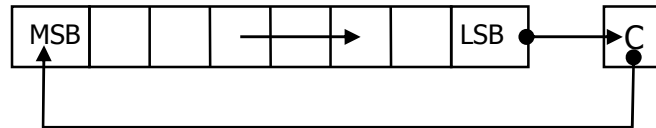


**RCL R/M,k** (*Rotate Through CF Left*) Rotation à gauche à travers le Carry

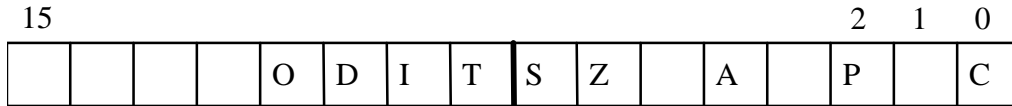




**RCR R/M,k** (*Rotate Through CF Right*) Rotation à droite à travers le Carry



**2.5.6 Instructions agissant sur les indicateurs**



**CLC** (*Clear Carry*) positionne le drapeau C à 0

**STC** (*Set Carry*) positionne le drapeau C à 1

**CMC** Complémente le drapeau C

**CLD** Positionne le Drapeau D à 0

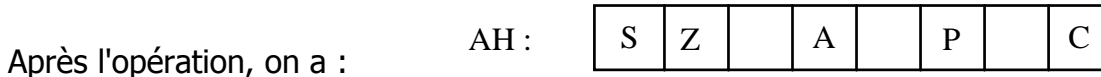
**STD** Positionne le Drapeau D à

**CLI** Positionne le Drapeau I à 0

**STI** Positionne le Drapeau I à 1

**LAHF**

Copier l'octet bas du registre d'état dans AH



**SAHF**

Opération inverse de LAHF : Transfert AH dans l'octet bas du registre d'état

**PUSHF**

Empile le registre d'état,

**POPF**

Dépille le registre d'état,

**Exercice 10):**

1. programme qui positionne l'indicateur P à 0 sans modifier les autres indicateurs
2. programme qui positionne l'indicateur O à 1 sans modifier les autres indicateurs

## 2.5.7 Les instructions de contrôle de boucle

**LOOP xyz** L'instruction **loop** fonctionne automatiquement avec le registre CX (compteur). Quand le processeur rencontre une instruction loop, il décrémente le registre CX. Si le résultat n'est pas encore nul, il reboucle à la ligne portant l'étiquette xyz, sinon il continue le programme à la ligne suivante

```

MOV  CX,une valeur
ici: XXX  xx,yy
      XXX  xx,yy
      .....
      XXX  xx,yy
      XXX  xx,yy
      loop ici
      XXX  xx,yy

```

### **Remarque sur l'étiquette :**

L'étiquette est une chaîne quelconque qui permet de repérer une ligne. Le caractère ':' à la fin de l'étiquette n'est obligatoire que si l'étiquette est seule sur la ligne

**LOOPZ xyz** (*Loop While Zero*) Décrémente le registre CX (aucun flag n'est positionné) on reboucle vers la ligne xyz tant que CX est différent de zéro et le flag Z est égal à 1. La condition supplémentaire sur Z, donne la possibilité de quitter une boucle avant que CX ne soit égal à zéro.

**LOOPNZ xyz** Décrémente le registre CX et reboucle vers la ligne xyz tant que CX est différent de zéro et le flag Z est égal à 0. Fonctionne de la même façon que loopz,

**JCXZ xyz** branchement à la ligne xyz si CX = 0 indépendamment de l'indicateur Z

**Exercice 11)** : (boucle.asm) Programme qui ajoute la valeur 3 au contenu des 100 cases mémoire du segment DATA dont l'adresse (offset) commence à 4000h

**Exercice 12)** : (boucle2.asm) Programme qui multiplie par 3 les 100 words contenu dans le segment DATA à partir de l'offset 4000h. On suppose que les résultats tiennent dans 16 bits (< 65536)

## 2.5.8 Les instructions de branchement

3 types de branchement sont possibles :

- ▶ Branchements inconditionnels
- ▶ Branchements conditionnels
- ▶ Appel de fonction ou d'interruptions

Tous ces transferts provoquent la poursuite de l'exécution du programme à partir d'une nouvelle position du code. Les transferts conditionnels se font dans une marge de -128 à +127 octets à partir de la position de transfert.

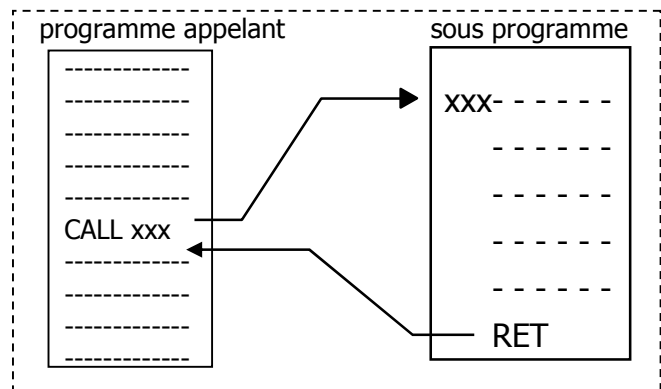
### ► Branchements inconditionnels

**JMP xyz** Provoque un saut sans condition à la ligne portant l'étiquette xyz.

**CALL xyz** Appel d'une procédure (sous programme) qui commence à la ligne xyz. La position de l'instruction suivant le CALL est empilée pour assurer une poursuite correcte après l'exécution du sous programme.

**RET** Retour de sous programme. L'exécution du programme continue à la position récupérée dans la pile.

**INT n** appel à l'interruption logicielle n° n



### ► Branchements conditionnels

Les branchements conditionnels sont conditionnés par l'état des indicateurs (drapeaux) qui sont eux même positionnés par les instructions précédentes.

Dans la suite nous allons utiliser la terminologie :

- **supérieur** ou **inférieur** pour les nombres non signés
- **plus petit** ou **plus grand** pour les nombres signés
- + pour l'opérateur logique OU

**JE/JZ xyz** (*Jump if Equal or Zero*) Aller à la ligne xyz si résultat nul ou si égalité. C'est-à-dire si Z=1

**JNE/JNZ xyz** (*Jump if Not Equal or Not Zero*) Aller à la ligne xyz si résultat non nul ou si différent. C'est-à-dire si Z=0

**JA xyz** (*Jump if Above*) aller à la ligne xyz si supérieur (non signé). C'est-à-dire si C + Z = 0

**JAE xyz** (*Jump if Above or Equal*) aller à la ligne xyz si supérieur ou égal (non signé). C'est-à-dire si C = 0

**JB xyz** (*Jump if Bellow*) Branche si inférieur (non signé). C'est-à-dire si C = 1

**JBE xyz** (*Jump if Bellow or Equal*) aller à la ligne xyz si inférieur ou égal (non signé). C'est-à-dire si C + Z = 1

**JC xyz** (*Jump if CARRY*) aller à la ligne xyz s'il y a retenu. C'est-à-dire si C = 1

- JNC xyz** (*Jump if No Carry*) aller à la ligne xyz s'il n'y a pas de retenu. C'est-à-dire si  $C = 0$
- JG xyz** (*Jump if Greater*) aller à la ligne xyz si plus grand (signé). C'est-à-dire si  $(S \oplus O) + Z = 1$
- JGE xyz** (*Jump if Greater or Equal*) aller à la ligne xyz si plus grand ou égal (signé). C'est-à-dire si  $S \oplus O = 0$
- JL xyz** (*Jump if Less*) aller à la ligne xyz si plus petit (signé). C'est-à-dire si  $S \oplus O = 1$
- JLE xyz** (*Jump if Less or Equal*) aller à la ligne xyz si plus petit ou égal (signé). C'est-à-dire si  $(S \oplus O) + Z = 1$
- JO xyz** (*Jump if Overflow*) aller à la ligne xyz si dépassement. C'est-à-dire si  $O = 1$
- JNO xyz** (*Jump if No Overflow*) aller à la ligne xyz s'il n'y a pas de dépassement  $O = 0$
- JP/JPE xyz** (*Jump if Parity or Parity Even*) aller à la ligne xyz si parité paire. C'est-à-dire si  $P = 1$
- JNP/JPO xyz** (*Jump if No Parity or if Parity Odd*) aller à la ligne xyz si parité impaire. C'est-à-dire si  $P = 0$
- JS xyz** (*Jump if Sign*) aller à la ligne xyz si signe négatif. C'est-à-dire si  $S = 1$
- JNS xyz** (*Jump if No Sign*) aller à la ligne xyz si signe positif. C'est-à-dire si  $S = 0$

**Exercice 13) :**

Ecrire la suite d'instructions pour réaliser les étapes suivantes :

1. Mettre 1 dans AX
2. incrémenter AX
3. si  $AX < 200$  recommencer au point 2
4. sinon copier AX dans BX

**Exercice 14) :**

Ecrire la suite d'instructions pour réaliser les étapes suivantes :

1. copier le contenu de la case mémoire [1230h] dans CX
2. Comparer CX à 200
  - a. si  $<$  incrémenter CX et recommencer au point 2
  - b. si  $>$  décrémenter CX et recommencer au point 2
  - c. si  $=$  copier CX dans AX et continuer le programme

**Exercice 15) :** (cherche.asm)

Programme qui cherche la valeur 65 dans le RAM à partir de la position 4000h. Une

fois trouvée, placer son adresse dans le registre AX

**Exercice 16)** : (boucle3.asm) Programme qui divise par 3 les 100 octets contenus dans les 100 cases mémoires commençant à l'adresse 4000h. Ne pas utiliser l'instruction **loop**

### 2.5.9 Instructions d'accès aux ports d'E/S

**IN** AL/AX, port lire un port d'entrée sortie.

**IN** AL, port ; charge le contenu du port d'adresse *port* dans AL

**IN** AX, port ; charge le contenu du port d'adresse *port* dans AL et le contenu du port d'adresse *port+1* dans AH

Si l'adresse *port* tient sur 8 bits, elle peut être précisée immédiatement, sinon il faut passer par le registre DX

```
IN AL, 4Dh          MOV DX, 378h
                   IN AL, DX
```

**OUT** port, AL/AX Ecriture dans un port d'E/S

L'adressage du port se fait de la même façon que pour IN

Out *port*, AX ; écrit AL dans *port* et AH dans *port+1*

## 2.6 CE QU'IL NE FAUT PAS FAIRE

Voici une liste non exhaustive de quelques erreurs à éviter

- ▶ Une opération ne peut pas se faire entre deux cases mémoire, il faut que ça passe par un registre. On ne peut pas avoir des instructions du style :

~~MOV [245],[200]  
ADD [BX],[BP]~~      ~~INST [ ], [ ]~~

- ▶ Bien que le registre SP soit un registre d'adressage, il ne peut être utilisé directement pour accéder à la pile, on peut toutefois le copier dans un registre valide pour l'adressage (BX, BP, SI, DI) et utiliser ensuite ce dernier :

~~MOV AX,[SP]~~      MOV BP,SP  
MOV AX,[BP]

- ▶ On ne peut pas faire des opérations directement sur un registre segment, il faut passer par un autre registre. On ne peut pas non plus copier un registre segment dans un autre

~~MOV ES,02F7H~~      MOV BX,02F7h  
MOV ES,BX

~~MOV ES,DS~~      MOV AX,DS  
MOV ES,AX

~~INC ES~~      MOV BX,ES  
INC BX  
MOV ES,BX

~~ADD ES,2~~      MOV BX,ES  
ADD BX,2  
MOV ES,BX

- ▶ On ne peut pas utiliser directement une adresse segment dans une instruction, il faut passer par un registre segment.

~~MOV [2A84h : 55],AX~~      MOV BX,2A84h  
MOV ES,BX  
MOV [ES : 55] , AX

- ▶ On ne peut pas faire une multiplication ou une division sur une donnée (immédiat)

~~MUL im  
DIV im~~

- ▶ On ne peut pas empiler/dépiler un opérande 8 bits

~~PUSH R/M<sub>8</sub>~~

- ▶ Le segment et l'offset sont séparé par le caractère ":" et non ","

~~[DS , BX]~~

[DS : BX]

- ▶ On ne peut pas utiliser les opérateurs + - x sur des registre ou des mémoires

~~MOV AX, BX+2~~

~~MOV AX, DX x 2~~





## 3 L'ASSEMBLEUR NASM

La syntaxe des mnémoniques que nous avons utilisée jusqu'à présent est la syntaxe de l'assembleur NASM. Rappelons que le rôle de l'assembleur est de convertir le programme source écrit en mnémonique en codes machines compréhensible par le processeur.

### 3.1 LES DIRECTIVES DE NASM

Les directives ne sont pas des instructions du 8086, elles sont destinées à l'assembleur qui les utilise pour savoir de quelle manière il doit travailler.

#### ► **BITS**

Cette directive indique à NASM s'il doit produire un code 16 bits ou un code 32 bits. Elle est utilisée sous la forme : `BITS 16` ou `BITS 32`

#### ► **ORG**

Cette directive indique à l'assembleur l'adresse à partir de laquelle le programme sera implanté dans la RAM. Pour les programmes ".com", cette adresse doit être 100h

#### ► **SEGMENT**

Cette directive précise dans quel segment, les instructions ou les données qui la suivent seront assemblées :

##### **SEGMENT .text**

Les instructions ou les données initialisées (par `db` ou `dw`) qui suivent cette 'déclaration' seront placés dans le segment programme (Code segment)

##### **SEGMENT .data**

Les données (initialisées) déclarées après cette directive sont placées dans le segment de données (Data segment)

##### **SEGMENT .bss**

Les données déclarées après cette directive seront placées dans le segment de données mais cette partie est réservée à la déclaration des variables non initialisées.

#### ► **%INCLUDE**

Comme en langage C, cette directive permet d'inclure un fichier source avant la compilation.

#### ► **EQU** : Définition de constante

```
mille EQU 1000
```

Chaque fois que l'assembleur rencontrera la chaîne *mille*, il la remplacera par le nombre 1000 (tout simplement☺).

#### ► **%DEFINE**

La directive *%define* permet de définir une constante un peut de la même façon que la directive *EQU*. La différence subtile est que *%define* définit une macros qu'on peut redéfinir dans la suite du programme ce qui n'est pas possible avec *EQU*.

### 3.2 LES PSEUDO INSTRUCTION DE NASM

Les pseudo instructions ne sont pas des instructions du 8086. Elles ne seront donc pas traduites en langage machine lors de l'assemblage. NASM les interprète et réalise les fonctions correspondantes. Les pseudo instructions les plus courantes servent à la déclaration des variables initialisées ou non initialisées, définition de constantes et la répétition des instructions :

- ▶ **db** : (*define byte*) définit une variable initialisée de 1 octet
- ▶ **dw** : (*define word*) définit une variable initialisée de 2 octets
- ▶ **resb** : réserve un octet pour une variable non initialisée
- ▶ **resw** : réserve un mot de 2 octets pour une variable non initialisée

### 3.3 LES EXPRESSIONS

NASM peut évaluer des expressions entre constantes. Les opérateur reconnus sont :

- +, -, \*** : addition, soustraction et multiplication
- /** : division non signée
- //** : division signée
- %** : modulo non signé
- %%** : modulo signé
- ~** : complément logique
- &** : ET logique
- |** : OU logique
- ^** : XOR logique
- <<** : décalage à gauche
- >>** : décalage à droite

```
x equ 0F00h
mov ax, (2*x+6)/2
mov ax, x << 4
mov ax, (x >> 8)+(x << 4)
mov ax, x & x >> 4; >> est prioritaire sur &
mov ax, x | x >> 4; >> est prioritaire sur |
```

Attention, les expressions ne peuvent être utilisés qu'avec des constantes. On ne peut pas avoir des choses du genre : `MOV AX, DX+2`

## 4 LES ENTREE SORTIES

Pour faire des entrées sorties (essentiellement avec l'écran et le clavier), on passe par des interruptions du BIOS ou du DOS. Nous n'allons voir ici que ce dont nous avons besoin.

### 4.1.1 L'interruption 10h du BIOS

Le BIOS est de relativement bas niveau et dépend fortement de la machine. L'interruption 10h peut effectuer beaucoup de fonctions différentes, le numéro de la fonction désirée doit être placé dans AH avant l'appel de l'interruption. Nous ne parlerons ici que de quelques fonctions.

#### ► Fonction 00

Cette fonction permet de choisir un mode texte ou un mode graphique. En changeant de mode, on peut effacer l'écran, ce qui fait que l'on peut appeler cette fonction pour effacer l'écran et rester dans le même mode.

Paramètres :

AH = 00

AL	mode	Résolution texte	dimensions caractère	Résolution graphique	Couleurs	pages	Segment
00h	T	40x25	9x16	360x400	16	8	B800
01h	T	40x25	9x16	360x400	16	8	B800
02h	T	80x25	9x16	720x400	16	8	B800
03h	T	80x25	9x16	720x400	16	8	B800
04h	G	40x25	8x8	320x200	4	.	B800
05h	G	40x25	8x8	320x200	4	.	B800
06h	G	80x25	8x8	640x200	2	.	B800
07h	T	80x25	9x16	720x400	mono	.	B000
0Dh	G	40x25	8x8	320x200	16	8	A000
0Eh	G	80x25	8x8	640x200	16	4	A000
0Fh	G	80x25	8x14	640x350	mono	2	A000
10h	G	80x25	8x14	640x350	16	.	A000
11h	G	80x30	8x16	640x480	mono	.	A000
12h	G	80x30	8x16	640x480	16	.	A000
13h	G	40x25	8x8	320x200	256	.	A000

Tableau 4.1 : modes écran

- Pour les modes texte, on peut doubler le nombre de ligne en chargeant un jeu de caractère de hauteur 8 pixels. Voir fonction 11
- Pour ne pas effacer l'écran, placer le bit 7 de AL à 1 (Ajouter 80h)

#### ► Fonction 09

Cette fonction permet d'écrire un caractère

- Permet les répétitions,
- Gère la couleur en mode texte et en mode graphique,
- Ne gère pas le curseur.

Paramètres : AH = 09h

AL = caractère à écrire

BH = page écran

BL = attribut de couleur (RVB :111=Blanc, 000=Noir)

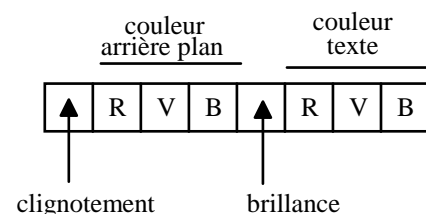


Fig. 4.1 : couleur en mode texte

CX = nombre de fois

Les caractères spéciaux ne sont pas reconnus (le 7 ne fait pas bip). Le bit 7 de la couleur fait un ou exclusif en mode graphique et un clignotement (uniquement en mode plein écran) en mode texte.

En mode graphique, l'attribut de couleur ne concerne que le caractère ou le pixel, il n'agit pas sur la couleur de l'arrière plan. Ceci est valable pour les autres fonctions qui gèrent la couleur.

### ► **Fonction 0Eh**

Cette fonction permet d'écrire un caractère,

- Fonctionne en mode graphique,
- Gère le curseur
- Gère la couleur seulement en mode graphique. Seule la couleur du caractère est gérée, la couleur du fond n'est pas gérée.

paramètres :     AH = 0Eh  
                   AL = code ascii du caractère à écrire  
                   BL = couleur du caractère (mode graphique uniquement).

Les caractères spéciaux sont reconnus :

- 10 (LF : *Line Feed* ) descend le curseur d'une ligne
- 13 (CR : *Carriage Return* ) ramène le curseur en début de lignes
- 08 (BS : *Back Space* ) ramène le curseur d'une position à gauche
- 07 (BEL) fait bip

```
mov  ah,0Eh  ; affiche le caractère A à la position courante du curseur
mov  al,'A'
int  10h
```

### ► **Fonction 02**

Cette fonction permet de positionner le curseur où on le désire, dans la page courante ou dans une page cachée.

Paramètres :     AH = 02h  
                   BH = numéro de la page  
                   DH = ligne (ordonnée)  
                   DL = colonne (abscisse)

En mode 25x80 les coordonnées vont de (0,0) à (24,79).

### **Exercice 17) (affcar.asm)**

Programme qui place l'écran en mode texte (3) et affiche le caractère A en vert sur bleu à la position (l=4,c=10)

### **Exercice 18) (couleurs.asm)**

Programme qui place l'écran en mode texte (3) et affiche les 16 premiers caractères de l'alphabet, chacun sur une ligne, chacun répété 40 fois et chacun avec une couleur différente sur fond noir (on commence avec la couleur 0 et on incrémente).

**► Fonction 05**

Cette fonction permet de sélectionner la page active de l'affichage.

Paramètres :     AH = 05h  
                  AL = numéro de la page

**► Fonction 11h, sous fonction 12h**

Cette fonction permet de charger le jeu de caractère de hauteur 8 pixels pour avoir un écran de 50 lignes. (Cette fonction doit être appelée après la fonction 00)

Paramètres  
AX = 1112h  
BL = 30h (08h semble marcher aussi)

**Exercice 19) (diag.asm)**

Programme qui place l'écran en mode 50 lignes et affiche ensuite l'alphabet (A ... Z) en Diagonal

**► Fonction 0Ch : allumer un pixel**

AH = 0Ch  
BH = 0 (numéro de page)  
AL = couleur du pixel  
    si bit 7 = 1 on trace en mode XOR sauf en mode 256 couleur  
CX = x (abscisse)  
DX = y (ordonnée)

**Exercice 20) (pixel.asm)**

Programme qui place l'écran en mode graphique 640x480, 16 couleur et allume le pixel de coordonnées (200,300) en jaune (14)

**4.1.2 L'interruption 21h du DOS**

Normalement le DOS est de relativement haut niveau et ne dépend pas de la machine. Il fait souvent appel au bios qui fonctionne à un niveau plus proche de la machine. L'interruption 21h peut réaliser plusieurs fonctions différentes. Nous ne citerons ici que celles que nous utiliserons.

**► Fonction 02**

Cette fonction permet d'écrire un caractère. Le caractère est envoyé vers la sortie standard, l'écriture peut donc être redirigée dans un fichier.

Paramètres :     AH = 02h  
                  DL = Caractère à écrire

**► Fonction 09**

Cette fonction permet en un seul appel, d'écrire une suite de caractères.

Paramètres :     AH = 09h  
                  DX = Adresse de la chaîne de caractères

La chaîne doit être terminée par le caractère \$

**Remarque :** Cette interruption retourne \$ dans le registre AL et ceci même si la documentation officielle affirme le contraire. Donc attention, si vous avez quelque chose dans AL avant d'appeler cette interruption, ce sera perdu

### Exemple : (phrase.asm)

```

;*****
; affiche une phrase ... l'aide de int21_fct09
;*****

BITS 16
ORG 0x0100

SEGMENT .data
txt      db  'MON PREMIER PROGRAMME NASM$'

SEGMENT .text

        MOV  AH,9      ; fonction 9 de int21
        MOV  DX,txt    ; adresse du début de la phrase
        INT  21h      ; écrit la phrase
        MOV  AX,4C00h  ; fin programme
        int  21h

```

### Remarques :

- Pour revenir à la ligne à la fin de la chaîne : 'Bonjour',10,13,','\$'
- Si la chaîne contient apostrophe : 'Ecole d'ingénieurs' → 'Ecole d',39,'ingénieurs\$'

### ► Fonction 07

Cette fonction permet de lire un caractère du clavier sans qu'il n'y ait d'écho à l'écran.

Paramètre passé : AH = 07

Paramètre retourné : AL = caractère lu

Les touches fonction retourne 2 caractères, d'abord un octet nul, puis le code étendu de la touche, il faut donc faire 2 appels consécutifs de la fonction 07.

### Exercice 21) (getchar.asm)

Programme qui :

- Affiche l'invité 'Veuillez taper un caractère'
- Attend l'entrée d'un caractère au clavier
- Affiche sur la ligne suivante 'Voici le caractère tapé ' suivi du caractère

### ► Fonction 0Bh


Cette fonction permet de savoir si un caractère est disponible dans la mémoire tampon du clavier. Elle est l'équivalente de la fonction kbhit (du C) ou de Keypressed (du Pascal). Il ne faut pas oublier de vider le buffer par une lecture à l'aide de la fonction 07 ou 08, sinon on risque d'avoir des surprises à la prochaine lecture du clavier.

Paramètre passé : AH = 0B

Paramètre retourné : AL = 0 ⇒ aucun caractère n'a été tapé

AL = 255 (-1) ⇒ au moins un caractère a été tapé

### ► Fonction 0Ah

Permet de saisir une chaîne de caractère au clavier. La saisie s'arrête quand on tape la touche , le caractère CR (13) est mémorisé avec la chaîne

Paramètres :

DX : adresse du *buffer* (zone mémoire tampon) où seront stockés la longueur de la chaîne ainsi que la chaîne saisie

[DX] : longueur max. avant d'appeler la fonction, il faut placer dans le premier octet du *buffer* la longueur max à ne pas dépasser, il faut compter le CR.

Une fois la saisie terminée, la fonction place dans le deuxième octet du *buffer* le nombre de caractère effectivement saisi. La chaîne saisie est placée tous de suite derrière.

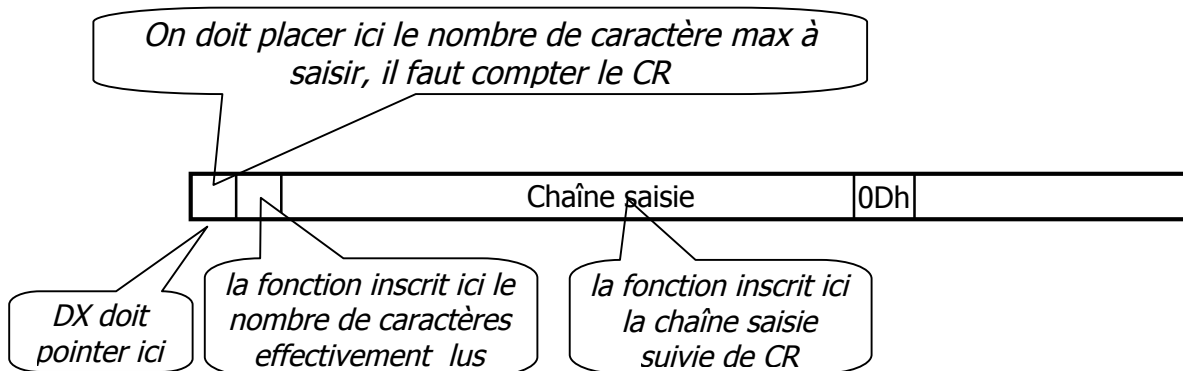


Tableau 4.2 : illustration de la fonction 0AH de int 21h

### **Exercice 22)** : (gets.asm)

Programme qui permet de saisir une chaîne de moins de 20 caractères et l'affiche ensuite en diagonale

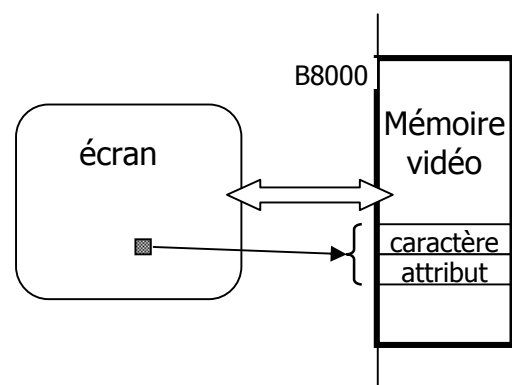
## 4.2 ACCES DIRECT A LA MEMOIRE VIDEO

La mémoire vidéo est une zone mémoire qui constitue une image de l'écran. Si on écrit quelque chose dans cette mémoire, elle apparaît à l'écran.

En mode texte, à chaque position de l'écran, correspondent deux positions (octets) de la mémoire vidéo. Le premier octet correspond au caractère affiché, le deuxième correspond à son attribut de couleur. La première paire d'octets représente le caractère en haut à gauche de l'écran. Pour le codage de la couleur, voir int 10h, fonction 09.

La mémoire écran commence à l'adresse B800h correspondant à l'adresse Segment:Offset = B800:0000

Si l'écran est configuré en mode 80 caractères par ligne. Chaque ligne correspond à 160 octets dans la mémoire vidéo. Pour écrire un "A" en rouge sur noir à la colonne 20 de la ligne 10, il faudra écrire 'A'=65=41h (code ascii de A) à la position  $10 \times 160 + 20 \times 2 = 1640$  et 04 dans la position suivante. La ligne 0 débute à la position mémoire 0, la ligne 1



à la position 160..., la ligne 10 à la position 1600 ...

Quand on programme en assembleur sur un PC, la modification de la valeur du segment DS peut donner des résultats inattendus. Nous utiliserons donc le registre segment ES pour accéder à la mémoire vidéo.

### 4.3 LES TEMPORISATIONS

On peut faire une temporisation en répétant plusieurs fois des instructions qui ne font rien. Cette méthode a l'inconvénient de dépendre de l'horloge système et ne donnera pas le même effet sur des machines différentes.

Une autre méthode consisterait à se servir du balayage de l'écran qui est à peu près le même sur toutes les machines. L'écran d'un ordinateur est balayé ligne par ligne, de haut en bas, par un spot d'électrons. Le balayage de tout l'écran dure à peu près 1/60 s. quand le spot arrive au coin bas-droite, il remonte directement au point haut-gauche pour recommencer. Pendant le balayage de l'écran, le bit 3 (4<sup>ème</sup>) du port 03DAh est égal 0, pendant le retour 'vertical' du spot, ce bit est placé à 1. On peut donc essayer de faire une temporisation de l'ordre de 1/70 s en surveillant le retour du spot. Dans ce qui suit, le bit 3 du port 03DAh sera désigné par 'spot'.

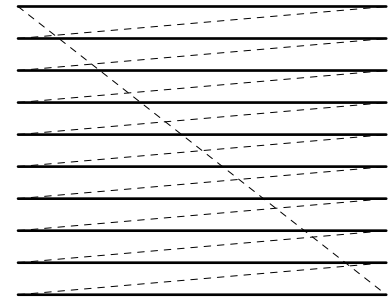
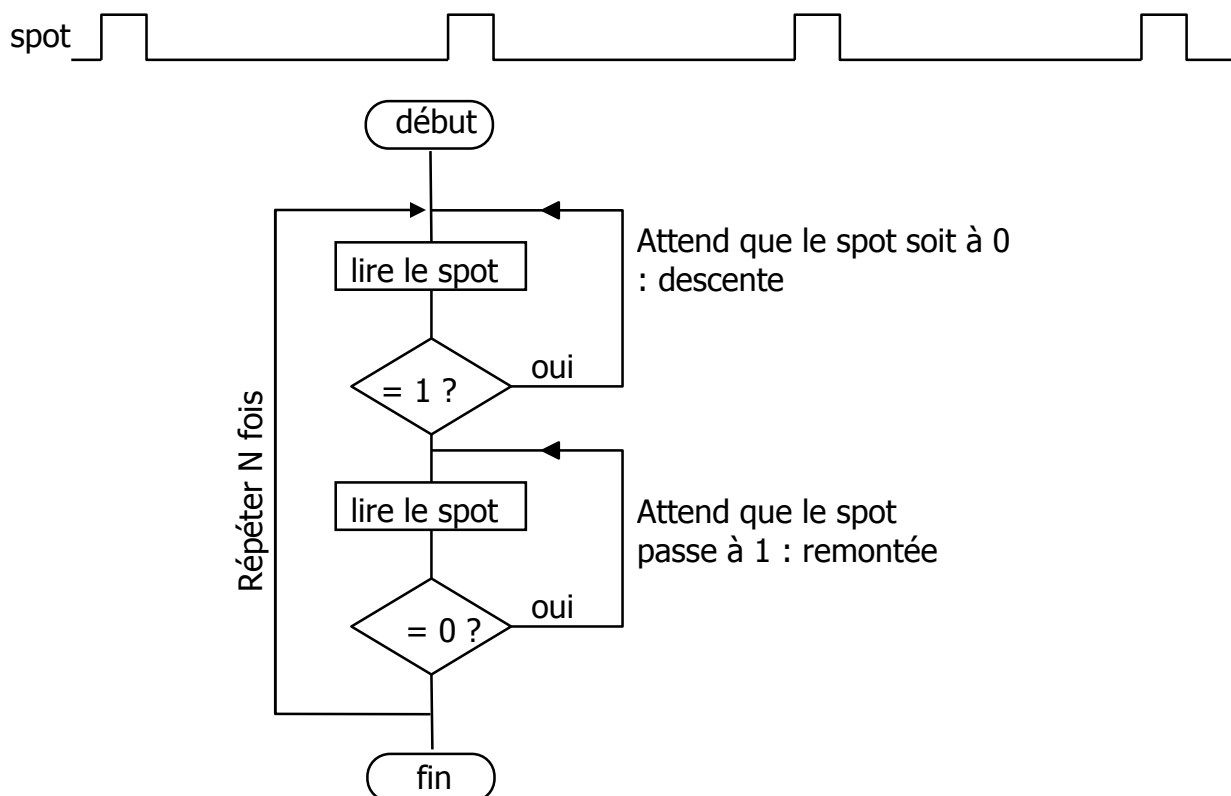


Fig. 4.2 : organigramme de la temporisation

Un appel unique à cette temporisation n'est pas intéressant car il peut contenir une erreur importante dépendant de la position du spot dans l'écran au moment de l'appel. Mais si on la répète plusieurs fois pour faire des temporisations plus longues, l'erreur est minimisée car c'est seulement le premier appel qui ne donne pas une temporisation précise. Si on la répète N fois on obtient une temporisation de l'ordre de  $N \times 1/70$  s





**Aide mémoire :**

Quelques interruptions									
		AH	AL	BH	BL	CX	DH	DL	commentaire
INT 10h	Mode écran	00	mode						
	Ecrit char	09	car	page	coul t/g	rep			!Curs !CarSpec
	Ecrt char	0A	car	page		rep			!Curs !CarSpec
	Ecrt char	0E	car		Coul g				Curs CarSpec
	Pos Curs	02		page			ligne	col	
	Choisir page	05	page						
	Allumer pixel	0Ch	couleur	page		x		y	
	50 lignes	11h	12h		30h				
INT 21h	Ecrt char	02						car	^C^B -> int 23h
	Ecrt chaîne	09					adresse		`\$'= fin chaîne
	Lit car !écho	07	Car lu						
	Lit chaîne	0Ah					adresse		Voir cours
	kbhit	0B	0 : non FF : oui						

Pseudo instructions et directives		
DB	Définir byte	X db 123
DW	Définir word	X dw 1234h
RESB	Réserver byte	X resb 2
RESW	Réserver word	X resw 5
EQU	Définir constante	Port equ 0x378
BITS	Générer code de n bits	BITS 16
ORG	Adresse début programme	ORG 0x100 (.com)
SEGMENT	Segment de saisie	SEGMENT .text
%DEFINE	Définir une constante	%define x 1000
%INCLUDE	Inclure un programme source	%include `nasmlib.asm`

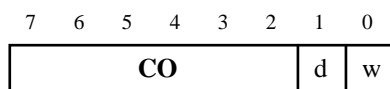
Opérateur arithmétiques et logique du préprocesseur de NASM		
+ , - , *	arithmétique	mov ax, (2*x+3)/6
/	Division non signée	
//	Division signée	
%	Modulo non signé	
%%	Modulo signé	
~	Complément binaire	
&	ET binaire	
	OU binaire	
^	XOR binaire	
<< >>	Décalage gauche et droite	
' '	code ascii	'A' = 65

## 5 CODE MACHINE DES INSTRUCTIONS

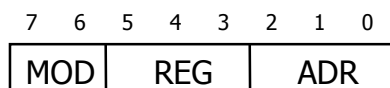
Une instruction peut comporter de 1 à 7 octets dont 1 ou 2 octets pour coder l'opération, les autres servent à définir les opérandes.

Dans le cas le plus général, l'instruction se fait entre un registre et une case mémoire. Dans le code machine de l'instruction, on trouvera le code de l'opération (CO), le code du registre utilisé (REG), le code du mode d'adressage utilisé (MOD) et le code permettant de déterminer l'adresse de la case mémoire (ADR):

- ▶ Le premier octet est un octet optionnel qui représente un préfix qui peut être un préfix de répétition ou un préfix de changement de segment.
- ▶ Le 2<sup>ème</sup> octet dit Code Opération se présente comme suit :



- **CO** : C'est le code proprement dit de l'instruction
  - **d** : désigne la destination du résultat
    - d = 0 → Résultat dans mémoire ou opération entre 2 registres
    - d = 1 → Résultat dans registre
  - **w** : Opération 8 bits ou 16 bits
    - w = 0 → 8 bits
    - w = 1 → 16 bits
- ▶ Le 3<sup>ème</sup> octet permet de définir les opérandes



- **MOD** : Ce champ de 2 bits nous informe sur le mode d'adressage : registre, directe ou la nature du déplacement dans les autres cas,
  - **REG** : Ce champ de 3 bits désigne le registre constituant un opérande
  - **ADR** : Ce champs de 3 bits précise l'adresse de l'autre opérande quand il s'agit d'une position mémoire.
  - Pour une opération entre deux registres **R ↔ R** :
    - REG = Registre source
    - ADR = Registre destination
- ▶ Les octets suivants concernent :
- Les déplacements sur 8 ou 16 bits utilisés dans le calcul d'adresse
  - Les données sur 8 ou 16 bits dans le cas de l'adressage immédiat
  - ...

## 5.1 LES CODES REG, ADR ET MOD

REG				ADR		MOD			
w = 1		w = 0		000	BX+SI+d	00	- Directe - Indirecte avec dep = 0		
000	AX	000	AL	001	BX+DI+d				
001	CX	001	CL	010	BP+SI+d	01	Indirect déplacement <b>court</b> : 8 bits, ≤127, 1 ou 2 chiffres hex		
010	DX	010	DL	011	BP+DI+d				
100	SP	100	AH	100	SI+d	10	Indirect dép. <b>long</b> : 16 bits, >128 , + de 2 chiffres hex		
101	BP	101	CH	101	DI+d				
110	SI	110	DH	110	- BP+d - Direct	11	R ← R ou R ← im Dans ce cas : ADR= code registre destination		
111	DI	111	BH	111	BX+d				
REG Segment									
00	01	10	11						
ES	CS	SS	DS						
				Préfix de changement de segment					
				ES	CS	SS	DS		
				26h	2Eh	36h	3Eh		

## 5.2 TABLEAU DES CODES BINAIRES

<b>AAA</b>	0011 0111			
<b>AAD</b>	1101 0101	0000 1010		
<b>AAM</b>	1101 0100	0000 1010		
<b>AAS</b>	0011 1111			
<b>ADC</b>				
<ul style="list-style-type: none"> <li>R/M ↔ R</li> <li>R/M ← im</li> <li>AL/AX ← im</li> </ul>	0001 00dw 1000 00sw 0001 010w	MOD REG ADR MOD 010 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>ADD</b>				
<ul style="list-style-type: none"> <li>R/M ↔ R</li> <li>R/M ← im (**)</li> <li>AL/AX ← im</li> </ul>	0000 00dw 1000 00sw 0000 010w	MOD REG ADR MOD 000 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>AND</b>				
<ul style="list-style-type: none"> <li>R/M ↔ R</li> <li>R/M ← im</li> <li>Ac ← im</li> </ul>	0010 00dw 1000 000w 0010 010w	MOD REG ADR MOD 100 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>BOUND</b>	0110 0010	MOD REG ADR	(Adr ou dep)	
<b>CALL</b>				
<ul style="list-style-type: none"> <li>intra segment direct</li> <li>intra segment indirect</li> <li>inter segment direct</li> <li>inter segment indirect</li> </ul>	1110 1000 1111 1111 1001 1010	adresse MOD 010 ADR Adresse	(Dep) Segment	
<b>CBW</b>	1001 1000			
<b>CLC</b>	1111 1000			
<b>CLD</b>	1111 1100			
<b>CLI</b>	1111 1010			
<b>CMC</b>	1111 0101			
<b>CMP</b>				
<ul style="list-style-type: none"> <li>R/M ↔ R</li> <li>R/M ← im</li> <li>AL/AX ← im</li> </ul>	0011 10dw 1000 00sw 0011 110w	MOD REG ADR MOD 111 ADR Donnée	(Adr ou dep) (Adr ou dep)	donnée

<b>CMPS</b>	1010 011w			
<b>CWD</b>	1001 1001			
<b>DAA</b>	0010 0111			
<b>DAS</b>	0010 1111			
<b>DEC</b> • R/M 8 bits • R 16 bits	1111 111w 01001 REG	MOD 001 ADR	(Adr ou dep)	
<b>DIV R/M</b>	1111 011w	MOD 110 ADR	(Adr ou dep)	
<b>ENTER</b>	1100 1000	donnée		
<b>EDC</b>				
<b>HLT</b>	1111 0100			
<b>IDIV R/M</b>	1111 011w	MOD 111 ADR	(Adr ou dep)	
<b>IMUL R/M</b>	1111 011w	MOD 101 ADR	(Adr ou dep)	
<b>IN</b> • Port défini • Port dans DX	1110 010w 1110 110w	Port		
<b>INC</b> • R/M 8 bits • R 16 bits	1111 111w 01000 REG	MOD 000 ADR	(Adr ou dep)	
<b>INS</b>	0110 110w			
<b>INT</b>	1100 1101	Num. interruption		
<b>INTO</b>	1100 1110			
<b>IRET</b>	1100 1111			
<b>JA</b>	0111 0111	DepRel_8		
<b>JAE</b>	0111 0011	DepRel_8		
<b>JB</b>	0111 0010	DepRel_8		
<b>JBE</b>	0111 0110	DepRel_8		
<b>JC</b>	1110 0010	DepRel_8		
<b>JCXZ</b>	1110 0011	DepRel_8		
<b>JE</b>	0111 0100	DepRel_8		
<b>JG</b>	0111 1111	DepRel_8		
<b>JGE</b>	0111 1101	DepRel_8		
<b>JL</b>	0111 1100	DepRel_8		
<b>JLE</b>	0111 1110	DepRel_8		
<b>JMP</b> • intra segment direct • intra segment direct court • intra segment indirect • inter segment direct • inter segment indirect	1110 1001 1110 1011 1111 1111 1110 1010 1111 1111	Dep_Relatif_16 Dep_Relatif_8. MOD 100 ADR Adr. branch.16 MOD 101 ADR	(jmp etiq) (jmp short etiq)  SegL	    SegH
<b>JNC</b>	0111 0011	DepRel_8		
<b>JNE</b>	0111 0101	DepRel_8		
<b>JNO</b>	0111 0001	DepRel_8		
<b>JNS</b>	0111 1001	DepRel_8		
<b>JNP</b>	0111 1011	DepRel_8		
<b>JO</b>	0111 0000	DepRel_8		
<b>JP</b>	0111 1010	DepRel_8		
<b>JS</b>	0111 1000	DepRel_8		
<b>LAHF</b>	1001 1111			
<b>LDS</b>	1100 0101	MOD REG ADR	(Adr ou dep)	

<b>LEA</b>	10001101	MOD REG ADR	(Adr ou dep)	
<b>LEAVE</b>	11001001			
<b>LES</b>	1100 0100	MOD REG ADR	(Adr ou dep)	
<b>LOCK</b>	1111 0000			
<b>LODS</b>	1010 110w			
<b>LOOP</b>	1110 0010	DepRel_8		
<b>LOOPZ</b>	1110 0001	DepRel_8		
<b>LOOPNZ</b>	1110 0000	DepRel_8		
<b>MOV</b> <ul style="list-style-type: none"> <li>• R/M <math>\leftrightarrow</math> R</li> <li>• m <math>\leftarrow</math> im</li> <li>• R <math>\leftarrow</math> im</li> <li>• AX/AL <math>\leftarrow</math> M_direct</li> <li>• M_direct <math>\leftarrow</math> AX/AL</li> <li>• Rseg <math>\leftarrow</math> R/M</li> <li>• R/M <math>\leftarrow</math> Rseg</li> </ul>	1000 10dw 1100 011w 1011 w REG 1010 000w 1010 001w 1000 1110 1000 1100	MOD REG ADR MOD 000 ADR donnée Adresse Adresse MOD 0 Rseg ADR MOD 0 Rseg ADR	(Adr ou dep) (Adr ou dep)   (Adr ou dep) (Adr ou dep)	donnée
<b>MOVS</b>	1010 010w			
<b>MUL R/M</b>	1111 011w	MOD 100 ADR	(Adr ou dep)	
<b>NEG</b>	1111 011w	MOD 011 ADR	(Adr ou dep)	
<b>NOP</b>	1001 0000			
<b>NOT</b>	1111 011w	MOD 010 ADR	(Adr ou dep)	
<b>OR</b> <ul style="list-style-type: none"> <li>• R/M <math>\leftrightarrow</math> R</li> <li>• R/M <math>\leftarrow</math> im</li> <li>• AX/AL <math>\leftarrow</math> im</li> </ul>	0000 10dw 1000 000w 0000 110w	MOD REG ADR MOD 001 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>OUT</b> <ul style="list-style-type: none"> <li>• port défini</li> <li>• port dans DX</li> </ul>	1110 011w 1110 111w	port		
<b>OUTS</b>	0110 111w			
<b>POP</b> <ul style="list-style-type: none"> <li>• M</li> <li>• R</li> <li>• Rseg</li> </ul>	1000 1111 0101 1REG 000REG111	MOD 000 ADR	(Adr ou dep)	
<b>POPA (*)</b>	0100 0001			
<b>POPF</b>	1001 1101			
<b>PUSH</b> <ul style="list-style-type: none"> <li>• M</li> <li>• R</li> <li>• Rseg</li> </ul>	1111 1111 01010 REG 000REG110	MOD 000 ADR	(Adr ou dep)	
<b>PUSHA (*)</b>	0110 0000			
<b>PUSHF</b>	1001 1100			
<b>RCL</b> <ul style="list-style-type: none"> <li>• R/M,1</li> <li>• R/M,CL</li> <li>• R/M,im8 (*)</li> </ul>	1101 000w 1101 001w 1100 000w	MOD 010 ADR MOD 010 ADR MOD 010 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>RCR</b> <ul style="list-style-type: none"> <li>• R/M,1</li> <li>• R/M,CL</li> <li>• R/M,im8 (*)</li> </ul>	1101 000w 1101 001w 1100 000w	MOD 011 ADR MOD 011 ADR MOD 011 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>REP/REPZ, REPNZ</b>	1111 001z			

<b>RET</b> • intra segment • inter segment	1100 0011 1100 1011			
<b>ROL</b> • R/M,1 • R/M,CL • R/M,im8 (*)	1101 000w 1101 001w 1100 000w	MOD 000 ADR MOD 000 ADR MOD 000 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>ROR</b> • R/M,1 • R/M,CL • R/M,im8 (*)	1101 000w 1101 001w 1100 000w	MOD 001 ADR MOD 001 ADR MOD 001 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>SAHF</b>	1001 1110			
<b>SAL/SHL</b> • R/M,1 • R/M,CL • R/M,im8 (*)	1101 000w 1101 001w 1100 000w	MOD 100 ADR MOD 100 ADR MOD 100 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>SAR</b> • R/M,1 • R/M,CL • R/M,im8 (*)	1101 000w 1101 001w 1100 000w	MOD 111 ADR MOD 111 ADR MOD 111 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>SBB</b> • R/M ↔ R • R/M ← im • AL/AX ← im (*)	0001 10dw 1000 00sw 0001 110w	MOD REG ADR MOD 011 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>SCAS</b>	1010 111w			
<b>SHR</b> • R/M,1 • R/M,CL • R/M,im8 (*)	1101 000w 1101 001w 1100 000w	MOD 101 ADR MOD 101 ADR MOD 101 ADR	(Adr ou dep) (Adr ou dep) (Adr ou dep)	Donnée_8
<b>STC</b>	1111 1001			
<b>STD</b>	1111 1101			
<b>STI</b>	1111 1011			
<b>STOS</b>	1010 101w			
<b>SUB</b> • R/M ↔ R • M ← im (**) • R ← im (**) • AL/AX ← im	0010 10dw 1000 00sw 1000 00sw 0010 110w	MOD REG ADR MOD 101 ADR 11 101 REG donnée	(Adr ou dep) (Adr ou dep) donnée	donnée
<b>TEST</b> • R/M ↔ R • R/M ← im • AL/AX ← im	1000 010w 1111 011w 1010 100w	MOD REG ADR MOD 000 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée
<b>WAIT</b>	1001 1011			
<b>XCHG</b> • R/M ↔ R • R ↔ AX	1000 011w 10010 REG	MOD REG ADR	(Adr ou dep)	
<b>XLAT</b>	1101 0111			
<b>XOR</b> • R/M ↔ R • R/M ← im • AL/AX ← im	0011 00dw 1000 000w 0011 010w	MOD REG ADR MOD 110 ADR donnée	(Adr ou dep) (Adr ou dep)	donnée

- ▶ Les champs entre ( ) sont présents dans le cas de l'adressage direct [aaaa] ou de l'adressage indirect avec déplacement [R+dep] ou [R<sub>b</sub> + R<sub>i</sub> + dep]
- ▶ Un champ adresse est toujours constitué de 2 octets : Adr<sub>L</sub> Adr<sub>H</sub>
- ▶ Un champ de donnée peut être de 1 ou de 2 octets selon l'instruction, D<sub>L</sub> suivie éventuellement de D<sub>H</sub>

(\*) Ces instructions ne tournent pas sur le 8086 mais sur les processeurs qui l'on suivi

(\*\*) s=1 dans le cas R/M<sub>16</sub> □ im<sub>8</sub>, une extension de signe 8 bits vers 16 bits est effectués sur la donnée immédiate avant l'opération.

### Exemples :

```
mov ax , 3456h
R16 ← im16
1011 w REG donnée
1011 1 000 5634
B8 56 34
```

```
mov CL , [BP+SI]
R8 ← M
1000 10dw MOD REG ADR
1000 1010 00 001 010
8A 0A
```

```
Mov bx , 56h
R16 ← im16
1011 w REG donnée
1011 1 011 56 00
BB 56 00
```

```
Mov bl, 56h
R8 ← im8
1011 w REG donnée
1011 0 011 56
B3 56
```

```
Mov DX , [123h]
R16 ← M
1000 10dw MOD REG ADR adresse
1000 1011 00 010 110 2301
8B 16 23 01
```

```
Mov AX , BX
R16 ← R16
1000 10dw MOD REG ADR
1000 1001 11 011 000
89 D8
```

```
Mov [SI + 146h] , BL
M ← R8
1000 10dw MOD REG ADR deplong
1000 1000 10 011 100 4601
88 9C 46 01
```

```
Mov [BX+DI + 46h] , CX
M ← R16
1000 10dw MOD REG ADR depcourt
1000 1001 01 001 001 46
89 49 46
```

```
mov AX , [3456h]
AX ← Mdirect
1010 000w adresse
1010 0001 5634
A1 56 34
```

```
AND BL , 38h
R8 ← im
1000 000w MOD 100 ADR donée
1000 0000 11 100 011 38
80 E3 38
```

## 6 ANNEXE

### 6.1 INSTRUCTIONS D'AJUSTEMENT DECIMAL

#### DAA (*Decimal Adjust AL after addition*)

Instruction sans opérande qui agit sur le registre AL pour obtenir un résultat BCD après l'addition de deux nombres BCD (avec résultat dans AL). AL est un registre 8 bits, il ne peut représenter que 2 chiffres BCD, le résultat de l'addition ne doit pas dépasser 99.

Algorithme : Après l'addition, si le drapeau A est positionné ou si les 4 bits de poids faible de AL représente un nombre supérieur à 9 alors le processeur ajoute 6 à AL.

AL + BL		AL	A		AL
03 + 04	→	07	0	DAA →	07
05 + 07	→	0C	0	DAA →	12
39 + 28	→	61	1	DAA →	67

ADD AL , BL  
DAA

#### AAA (*ASCII Adjust after Addition*)

Instruction similaire à DAA sauf qu'ici, il s'agit de la représentation BCD étendue pour laquelle chaque chiffre est codé sur 8 bits (*unpacked BCD form*). Ceci facilite la conversion vers l'ASCII, d'où le nom de l'instruction.

Algorithme : Si l'addition des deux chiffres dépasse 9, l'instruction AAA recopie le chiffre des unités dans AL, incrémente AH. Si AH était = 0, on obtient les dizaines dans AH. Les drapeaux C et A sont positionnés.

Ce qui (d'une façon plus informatique) peut être représenté par :

Si (LSD(AL) > 9 OU A = 1)

AL ← (AL + 6) AND 0Fh

AL ← AH + 1

C ← 1

A ← 1

Sinon

C ← 0

A ← 0

FinSI

MOV AH, 5  
ADD AL , BL  
AAA

AL + BL		AL	A		AH	AL	C	A
03 + 04	→	07	0	AAA →	05	07	0	0
05 + 07	→	0C	0	AAA →	06	02	1	1
09 + 08	→	11	1	AAA →	06	07	1	1

#### DAS (*Decimal Adjust AL after Substraction*).

Ajustement décimal de AL après soustraction de deux opérandes BCD pour que le résultat soit aussi en BCD. (voir DAA)

#### AAS (*ASCII Adjust after Substraction*)

Ajustement décimal de AL après soustraction de deux opérandes BCD au format étendu (1 chiffre par 8 bits) pour que le résultat soit de même type. (Voir AAA)

#### AAM (*ASCII Adjust AX after multiply*) Ajustement décimal après multiplication de deux opérandes BCD au format étendu pour que le résultat soit de



même type. L'opération est faite implicitement sur le registre AX. (voir AAA)

### **AAD** (*ASCII Adjust AX before Division*)

Si AX contient un nombre BCD étendu (1 chiffre par 8 bits), cette instruction effectue un ajustement décimal inverse c.a.d. une conversion BCD vers binaire. Cette instruction est utile avant une division, car cette dernière se fait en binaire. Après la division, le quotient présent dans AL peut être converti en BCD à l'aide de l'instruction DAA.

Si AX contient la représentation BCD<sub>L</sub> du nombre décimal 98, soit AX=0000100100001000. Si on désire faire une division, il faut d'abord convertir AX en binaire (AAD), réaliser la division (DIV) et convertir ensuite le résultat en BCD (DAA) :

- AAD donne la représentation de 98 en binaire AX = 0000 0000 0110 0010

- DIV BL (avec BL=8) donne AH=02 = reste, et AL = 0C = quotient

Si on veut que le quotient soit représenté en BCD, on peut utiliser l'instruction DAA et on obtient AL = 0001 0010 = 12 (BCD)

## **6.2 LES INSTRUCTIONS DE MANIPULATION DE CHAINES**

Les instructions de manipulation de chaînes sont au nombre de 5 :

- ▶ **MOVS, LODS, STOS** sont des instructions de transfert, elles peuvent être répétées à l'aide du préfixe **REP**
- ▶ CMPS et SCAS sont des instructions de comparaisons, elles peuvent être répétées à l'aide du préfixe **REPZ**

Chacune des instructions existe en deux versions, l'une se fait entre deux octets **XXXXB**, l'autre se fait entre 2 words **XXXXW**

Ces instructions sont utilisées sans opérandes. Les opérations se font implicitement entre 2 opérandes pointés par les registres d'index SI et DI. L'opérande pointé par SI constitue l'opérande source, celui pointé par DI constitue l'opérande destination.

L'opérande source est pris par défaut dans le segment DATA à moins que l'on précise dans l'instruction un préfixe de changement de segment qui oblige le processeur à prendre l'opérande source dans un autres segment.

L'opérande destination est toujours situé dans le segment EXTRA, on ne peut pas le modifier.

[DS:SI] ↔ [ES:DI]

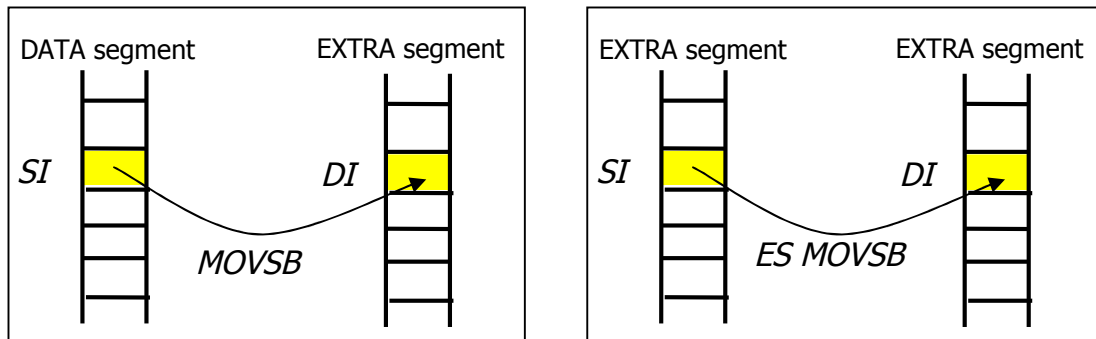
Après chaque opération SI et DI sont automatiquement incrémentés ou décrémentés selon la valeur de l'indicateur D :

- D = 0 → SI et DI sont incrémentés, (voir instruction CLD)
- D = 1 → SI et DI sont décrémentés. (Voir instruction STD)

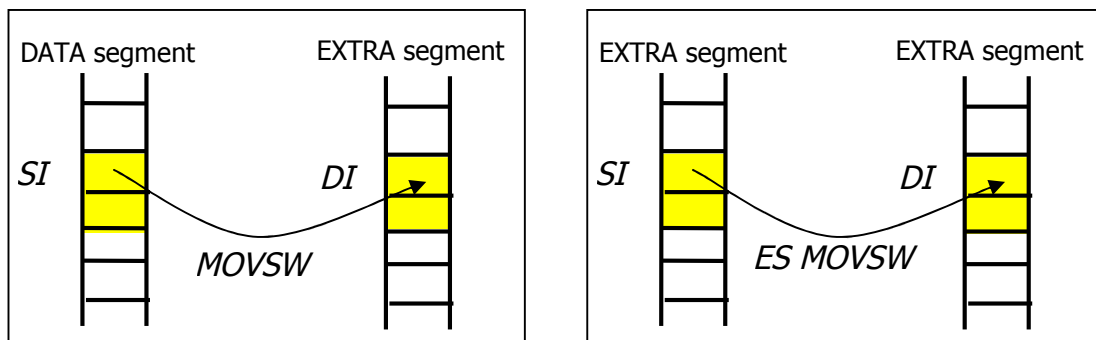
SI et DI sont incrémentés de 1 ou de 2 selon que l'opération s'effectue sur un octet (byte) ou sur un mot de 16 bits (word).

→ **MOVS : Copie l'opérande source dans l'opérande destination**

**MOVSB :** Copie un **octet** depuis la case source [DS:SI] vers la case destination [ES:DI] puis auto inc/décrémente les registre SI et DI



**MOVSW :** Transfert deux octets depuis la source vers la destination puis auto inc/décrémente de 2 les registre SI et DI



Grâce au préfixe de répétition REP, les instructions MOVSB et MOVSW sont répétées CX fois ce qui permet de copier une zone mémoire dans une autre.

**Exercice 23)**

Programme qui recopie 500 octets de la position 4000h du DATA segment vers la position 6000h du EXTRA segment

**Exercice 24)**

Programme qui recopie 500 octets de la position 4000h du EXTRA segment vers la position 6000h du même segment

**Exercice 25)**

Programme qui recopie 500 octets de la position 4000h du DATA segment vers la position 6000h du même segment

**→ LODS**

**LODSB** : Copie l'octet source dans AL puis inc/decr le registre SI.

**LODSW** : Copie le mot source dans AX puis inc/decr de 2 le registre SI

La répétition de cette instruction est sans intérêt.

**→ STOS**

**STOSB** : Copie AL dans l'octet destination et inc/decr le registre DI.

**STOSW** : Copie AX dans le mot destination et inc/dec de 2 le registres DI.

Avec le préfixe de répétitions REP, Cette instruction permet d'initialiser une chaîne (une zone mémoire) avec le même caractère.

**→ CMPS**

**CMPSB** : Compare l'octet source avec l'octet destination, positionne les indicateurs puis inc/decrémente les registres SI et DI.

**CMPSW** : Compare le mot source avec mot destination, positionne les indicateurs puis inc/decrémente de 2 les registres SI et DI

- Avec le préfixe de répétition **REPZ** (*repeat while Z*), cette instruction est répétée CX fois tant que Z=1
- Avec le préfixe de répétition **REPNZ** (*repeat while not Z*), cette instruction est répétée CX fois tant que Z=0

La répétition de cette instruction permet par exemple la comparaison de deux chaînes de caractères

**Exercice 26)** (compstr.asm)

Donner le programme qui compare deux chaînes de 30 caractères situés respectivement aux adresses 4000h et 6000h et positionne AL comme suit :

AL = 0 si égalité, AL = -1 si différents

**→ SCAS**

**SCASB** : Compare AL avec l'octet destination, positionne les indicateurs puis inc/decrémente le registre DI.

**SCASW** : Compare AX avec le mot destination, positionne les indicateurs puis inc/dec le registre DI de 2

- Avec le préfixe de répétition **REPZ** (*repeat while Z*), cette instruction est répétée CX fois tant que Z=1
- Avec le préfixe de répétition **REPNZ** (*repeat while not Z*), cette instruction est répétée CX fois tant que Z=0

Avec répétition cette opération permet (par exemple) de chercher l'occurrence d'une

valeur dans une chaîne

### **Exercice 27)** (findchar.asm)

Donner le programme qui cherche le caractère 'X' dans la chaîne de 30 caractères située à l'adresse 4000h du data segment.

- trouvé → DI = adresse
- non trouvé → DI = -1

## **6.3 INSTRUCTIONS DE TRANSFERT D'ADRESSE**

### **LEA R<sub>16</sub>, M**

(*Load Effective Adress*) Transfert l'offset de l'adresse de M dans le registre R<sub>16</sub>.

LEA AX, [BX+124] ; Copier dans AX la valeur de BX + 124

Remarquer que l'on serait tenté d'utiliser la syntaxe suivante :

MOV AX, BX+124 ;incorrect car pour réaliser une addition il faut utiliser l'instruction ADD

### **LDS R<sub>16</sub>, [adr]** (*Load pointer into DS*)

Le mot pointé par adr est recopié dans R<sub>16</sub> et le mot suivant est recopié dans DS

LDS BX,[200h]

30h	200h
31h	
32h	
33h	
34h	
35h	

Après l'instruction ci-dessus on aura :

BX = 3130h et DS = 3332h

Cette instruction doit être utilisée avec beaucoup de précaution, car on changeant la valeur de DS, les données déclarées dans le DATA segment ne seront plus accessibles.

### **LES R<sub>16</sub>,[adr]** (*Load pointer into ES*)

Similaire à LDS, ici le 2<sup>ème</sup> mot est chargé dans ES

## **6.4 INSTRUCTIONS DIVERSES**

### **XLAT**

Instruction sans opérande qui recopie dans AL le contenu de la case mémoire pointée par BX+AL

Peut servir dans des opérations de transcodage en faisant correspondre au contenu de AL une valeur préalablement rangée dans un tableau. BX doit être initialisé pour pointer sur le début de la table et AL sert de registre d'index par rapport à BX pour localiser l'élément désiré dans la table.

La figure ci-contre montre une table contenant les codes ASCII de quelques chiffres. Pour déterminer le code ASCII d'un chiffre, il suffit de placer le chiffre dans AL et d'exécuter l'instruction XLAT

(0)	48	← 2FA3
(1)	49	
(2)	50	
(3)	51	
(4)	52	
(5)	53	

```
MOV BX,2FA3h  
MOV AL,3  
XLAT
```

Le code ASCII de du chiffre 3 se trouve maintenant dan AL

- HALT** Met le processeur en mode veille. On peut sortir de cet état soit par une interruption externe autorisée soit par un RESET.
- WAIT** Met le processeur en mode WAIT. On peut sortir de cet état quand la broche test du processeur passe au niveau 1. Cette instruction sert à synchroniser le processeur sur des événements externes. Si une interruption externe autorisée intervient lors du mode WAIT, elle est exécutée puis le processeur revient en mode WAIT.