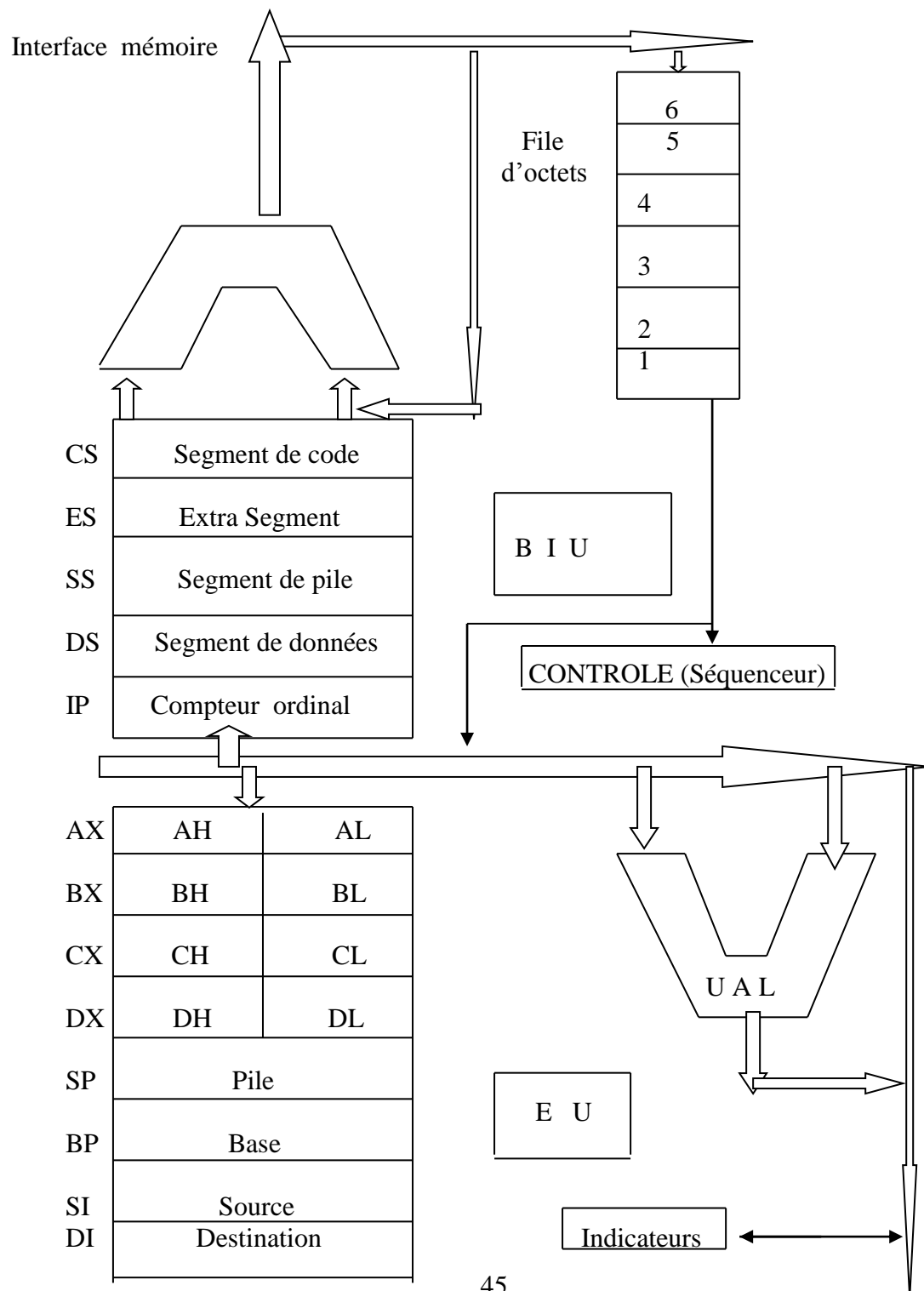


# ***PRESENTATION D'UNE MACHINE PEDAGOGIQUE AVEC SON LANGAGE D'ASSEMBLAGE***

## **PARTIE 1** PRESENTATION D'UNE MACHINE PEDAGOGIQUE

1) **ARCHITECTURE :** Diagramme fonctionnel du microprocesseur 8086 :



**Registres de base ou registres segment :**

CS ES SS DS IP

**Registres généraux :**

AX BX CX DX

**Registres pointeurs**

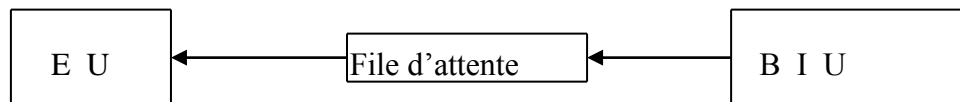
de base : SP BP

d'index : SI DI

**Registre d'état ou registre condition :** Indicateurs (16 bits)**File d'octets :** 6 octets**2) FONCTIONNEMENT DU MICROPROCESSEUR :**

Le micro processeur 8086 est constitué de :

- Unité d'exécution E U (Execution unit)
- Unité d'interface de bus (BIU)



Les deux unités EU et BIU fonctionnent en parallèle en mode pipeline ou anticipation (Pré recherche) : Lorsqu'une instruction est en cours d'exécution, le microprocesseur accède déjà à la suivante, c'est à dire pendant que l'EU exécute une instruction, la BIU appelle la suivante .

Les instructions appelées (Pré recherchées) sont mises en attente dans une file d'attente de la BIU . Cette file d'attente fonctionne selon le principe FIFO .

**NB :**

Les données sont transmises sur 16 bits pour le 8086 .

**3) CARACTERISTIQUES DU MICROPROCESSEUR :**

La gamme des microprocesseurs 8086 équipe les micro ordinateurs de type PC et compatibles.

Les premiers modèles de PC, commercialisés au début des années 1980 utilisaient le 8086 (Un microprocesseur de 16 bits). Les modèles suivants ont été utilisés successivement : Le 80 286, le 80 386, le 80 486 et le Pentium (80 586). Chacun de ces processeurs est plus puissant que les précédents (Augmentation de la fréquence d'Horloge, de la longueur du Bus : 32 bits d'adresse et de données et ajout de registres).

Chacun d'entre eux est compatible avec les modèles précédents, c'est à dire un programme écrit dans le langage machine du 286 peut s'exécuter sans modification sur un 486. Mais l'inverse n'est pas vrai, puisque chaque génération a ajouté des instructions nouvelles : On parle de compatibilité ascendante.

On distingue différents registres :

**■ Registres généraux (Registres de travail) :**

Ce sont des accumulateurs au sens large, mais généralement chaque registre est réservé à une fonction spécifique.

Ce sont 4 registres qui peuvent être séparés en 8 registres de 8 bits chacun (un octet), car un registre a une taille égale à 16 bits :

AX	=	AH	+	AL	(E/S, MUL, DIV ...)
BX	=	BH	+	BL	(Registre de base pour l'@ en zone mémoire)
CX	=	CH	+	CL	(Boucles, Comptage des données)
DX	=	DH	+	DL	(@ indirect des ports d'E/S, ...)

**Exemple :** Le registre AX



AH = Partie Haute (High : Forte) du registre AX = Octet gauche de AX

AL = Partie basse (Low : Faible) du registre AX = Octet droit de AX

#### ■ Registres pointeurs (base et index)

Ce sont 4 registres de 16 bits chacun.

SI = Index Source.

DI = Index Destination.

SP = Pointeur de Pile.

BP = Base de pile (Pointeur de base)

SI et DI sont deux registres d'index qui permettent de déplacer des données dans la MC.

Généralement, SI et DI impliquent l'utilisation de DS.

SP et BP sont associés à SS.

#### ■ Registres segments (Registres de base) :

CS = Code Segment (Programme)

DS = Data Segment (Données)

ES = Extra Segment (Données)

SS = Stack Segment (Pile)

Ce sont ces registres qui représentent le segment dans lequel on travaille ou auquel on veut accéder.

#### **NB :**

L'adresse physique de la mémoire se calculera à l'aide des registres de base combinés aux registres précédents. Le calcul de l'adresse est effectué par la machine, le programmeur ne s'occupant que du déplacement. Les registres de base peuvent être mis à jour soit automatiquement, soit maniés directement par le programmeur.

#### ■ Compteur ordinal :

IP = c'est un registre de 16 bits qui contient le déplacement de l'instruction courante vis à vis du code segment (CS), c'est à dire il contient le décalage servant à calculer l'adresse de la prochaine instruction dans le segment désigné par CS et est mis à jour par la BIU

#### **NB :**

L'adresse est exprimée sous la forme CS : IP

L'adresse physique se calcule de la manière suivante :  $10H * CS + IP$

## ■ Registre d'état (Indicateurs)

C'est un registre condition de taille égale à 16 bits , mis à jour par le système à chaque exécution d'instruction. Il détermine l'état actuel du microprocesseur et de son environnement.

C'est un mot d'état contenant les indicateurs :

15 ..... 0

R	R	R	R	OF	DF	IF	TF	S	Z	U	AC	U	P	U	C
---	---	---	---	----	----	----	----	---	---	---	----	---	---	---	---

- R = Bit réservé
- OF = Dépassement (Overflow) , indique un dépassement de capacité pour les opérations en arithmétique signée .
- DF = Direction d'incrémentation : 0 incrémentation et 1 décrémentation de l'index .
- IF = Interrupt : Autorise ou bien interdit les interruptions internes (0 autorise les Its et 1 masque les Its)
- TF = Trap : Si ce bit est positionné , alors une IT « TRAP » se produit à chaque instruction (= 1 déclenche une IT après la prochaine instruction, puis revient à zéro).
- S = Sign : Indique le signe du nombre (1 → négatif et 0 → positif).
- Z = Zéro : Indique si résultat est nul ou non (= 1 → Résultat nul).
- U = Bit indéfini .
- AC = Auxiliary carry : Employé dans les calculs décimaux.
- P = Parity : Parité : mis à 1 si l'octet de poids faible du résultat est pair.
- C = Carry : Retenue : Mis à 1 si retenue , à 0 autrement.

## 1) GENERALITES:

Pour écrire un programme, l'utilisateur a souvent le choix entre plusieurs langages (FORTRAN, PASCAL, C, ADA, Assembleur ...).

Par contre, l'ordinateur comprend seulement son propre langage, le langage machine.

En programmation, on utilise le terme langage pour indiquer un ensemble d'instructions et de règles syntaxiques permettant l'écriture de ce qu'on appelle le code source. Mais la machine n'est capable d'exécuter que des programmes écrits en code machine (Code objet). D'où la nécessité de traduire le code source en code objet avant l'exécution : Ce travail de traduction se fait automatiquement à l'aide de traducteurs, tels les assembleurs et les compilateurs.

On peut distinguer plusieurs niveaux de langages de programmation, les niveaux supérieurs s'approchant du langage de l'utilisateur, les niveaux inférieurs s'adaptant mieux aux caractéristiques des machines.

### POURQUOI L'ASSEMBLEUR ?

L'exécution d'un programme consiste à « donner » à la machine une séquence d'instructions directement interprétables par elle. Obligatoirement, les premiers programmes étaient écrits en binaire. C'était une tâche difficile et exposée aux erreurs : C'était l'époque des langages machine. Par la suite et pour faciliter le travail, les programmes ont été écrits en donnant directement les noms (abrégiés) des opérations : On les a appelés les codes mnémoniques (ADD, SUB, MUL, DIV, MOV, ...). Les adresses des instructions et des variables pouvaient être données sous forme symbolique. Pour pouvoir utiliser effectivement tous ces symboles, il fallait trouver le moyen de les convertir en langage machine : Ce fut réalisé par un programme, l'assembleur (Langage d'assemblage) qui est toujours utilisé, car c'est le seul langage qui permet d'exploiter au maximum les ressources de la machine.

Bien que remplacé par les langages évolués et relégué au deuxième plan, le langage d'assemblage reste un langage important. Il est encore utilisé aujourd'hui dans certains cas par des spécialistes :

- lorsqu'on veut tirer profit de l'architecture de la machine.
- Lorsqu'on veut un diagnostic d'erreurs : Pour détecter certaines erreurs, on est obligé d'examiner le contenu de la mémoire.

#### NB :

- Le langage d'assemblage est propre à chaque type de machine.
- Il permet d'accéder à toutes les ressources et aux facilités de traitement de la machine.
- Le langage d'assemblage est un langage relativement compliqué, quel est donc l'intérêt de l'apprendre ? Son principal avantage est sa vitesse d'exécution (Sans comparaison avec le langage PASCAL, et même encore, il est plus rapide que le langage C). En plus, même si vous ne l'utilisez pas beaucoup, le fait de le connaître vous permettra de mieux programmer dans les langages de haut de niveau.

D'autre part, puisque l'assembleur est un utilitaire qui n'est pas interactif, le programme que l'on désire traduire en langage machine (on dit assembler) doit être placé dans un fichier texte (avec l'extension .ASM sous DOS), et la saisie du programme source au clavier nécessite un programme appelé éditeur de texte.

Ensuite 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 objet 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 liens (même si l'on en a qu'un seul). L'éditeur de liens « fabrique » un fichier exécutable, avec l'extension .EXE.



Ce qu'il faut faire

Avec qui le faire

**Zone opération :** (Code opération)

Elle permet à la machine de savoir quelle opération elle doit réaliser , c'est à dire quels éléments elle doit mettre en œuvre.

**Zone adresse :**

Elle ne contient pas la plupart du temps la donnée elle même, mais son adresse, c'est à dire l'emplacement de la case mémoire où est rangée cette donnée.

**NB :**

La partie de l'instruction appelée zone @ données peut être divisé en champs et peut aller jusqu'à 4 champs : On parle alors d'instructions à N adresses ( $N = 0, 1, 2, 3, 4$ ). On dit alors d'une machine qu'elle est à N adresses, si la plupart de ses instructions sont à N adresses.

### 3) PRESENTATION D'UN LANGAGE D'ASSEMBLAGE : M A S M

**NB :**

MASM est le langage machine, produit par Microsoft le plus répandu pour les micro ordinateurs.

#### 3.1) Structure générale du programme source :

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

Les données sont déclarées par des DIRECTIVES (Pseudo- Instructions), c'est à dire des mots clefs spéciaux que comprend l'assembleur.

Les directives qui déclarent des données sont regroupées dans le SEGMENT de DONNEES qui est délimité par les directives SEGMENT et ENDS.

Les instructions sont placées dans un autre segment : SEGMENT de CODE.

La directive ASSUME est toujours présente. Elle permet l'attribution des registres.

La première instruction du programme (Dans le segment d'instructions) doit toujours être repérée par une ETIQUETTE (Label)

Le programme doit se terminer par la directive END suivie par le nom de l'étiquette de la première instruction (Ceci permet d'indiquer à l'éditeur de liens quelle est la première instruction à exécuter lorsqu'on lance le programme).

Les points virgules (;) indiquent des commentaires.

**NB :**

La différence entre une INSTRUCTION et une DIRECTIVE (Pseudo instruction) réside dans le fait que pour cette dernière, il n'y a aucun code généré.

PILE

SEGMENT

PARA STACK ' PILE '

	.	; PARA : Paragraphe
	.	; STACK PILE : utile pour le LINK.
	.	; Zone non obligatoire réservée au système.
		; (Zone de travail du système)
PILE	ENDS	; Fin de la pile.
DATA	SEGMENT	; DATA est le nom du segment de Données.
	.	; Ce segment contient les directives de déclaration
	.	; des données.
	.	
DATA	ENDS	; Fin du segment de données.
CODE	SEGMENT	; CODE est le nom du segment d'instructions
DEBUT	...	; Première instruction étiquetée par DEBUT.
	.	; Suite d'instructions.
	.	
	.	
DEBUT	ENDP	
CODE	ENDS	; Fin du segment de code.
	END	DEBUT ; Fin du programme dont la première
		; instruction est étiquetée par DEBUT.

### 3.2) DIRECTIVES : (Pseudo-Instructions)

**EQU**

(Equal)



a) **Sémantique :**

Elle assigne une constante à un identificateur qui ne pourra plus être modifiée au cours de l'exécution du programme.

b) **Exemples :**

VRAI	EQU	1	;	VRAI est l'identificateur et vaut 1
CINQ	EQU	2*2+1	;	Évalué lors de l'assemblage.

c) **NB :**

- Les constantes peuvent être définies dans différentes bases, en ajoutant un caractère le signalant. Par défaut, l'assembleur assume que la constante est une valeur décimale.

**Exemples :**

00101b	Constante binaire
0FFFh	Constante hexadécimale
1024d	Constante décimale
1024	Constante décimale
'a'	Constante de type caractère
'ABCD'	Constante de type chaîne de caractères

- Les constantes hexadécimales qui commencent par une lettre, doivent être précédées d'un zéro, sinon l'assembleur considère qu'il s'agit d'un identificateur.

**DB**

( DEFINE BYTE )

a) **Sémantique :**

Elle définit (et initialise) un ou plusieurs octets.

Elle est aussi utilisée pour définir une variable de type caractère ou chaîne de caractères. Dans les deux cas, on utilise les apostrophes comme délimiteurs et se termine par \$.

b) **Exemples :**

STATUS	DB	0	;	Définition d'une variable
STATUS				
VECT	DB	4,7,5	;	et sa mise à 0.
			;	Définition d'un vecteur VECT de 3
			;	éléments : VECT[1]=4 ; VECT[2]=7
			;	et VECT[3]=5
SURFACE	DB	?	;	Définition d'une variable SURFACE
			;	non initialisée
MESS	DB	' Le ciel est bleu '		
M	DB	07,' Ceci émet un BIP '		
SALUT	DB	' BONJOUR '		

c) **NB :**

Codes fréquemment utilisés :

13	ou (0DH)	Retour chariot (Carriage return) :
		Retourne le curseur au début de la ligne.
10	ou (0AH)	Nouvelle ligne (Line feed) :
		Passage à la ligne suivante.
27		ECHAPPE (ESCAPE)
08		RECU à gauche (BACKSPACE)
09		TAB (TABULATION)
07		BEL Emission d'un BIP sonore

DW

( DEFINE WORD )

**a) Sémantique :**

Elle définit (et initialise éventuellement) un ou plusieurs mots .

**b) Exemple :**

POIDS	DW	100	;	Définition d'une variable POIDS = 100
V	DW	12,54,30	;	Définition d'un vecteur à 3 éléments .
X	DW	?	;	Définition d'une variable X non initialisée.

DD

( DOUBLE DATA )

**a) Sémantique :**

Elle définit (et initialise éventuellement) un ou plusieurs double mots.

**b) Exemple :**

ADR	DD	0FFAE17C4H	;	Définition d'une adresse
			;	= FFAE17C4H
TAB	DD	504C2H, 30H, 0F4CD9H	;	Définition d'un tableau
			;	de 3 adresses

**c) NB :**

L'ordre de 2 mots d'un double mot en MC est :

ADR	=	FFAE	174C
		Mot de poids faible	Mot de poids fort

DUP

( DUPLIQUER )

**a) Sémantique :**

Elle signale à l'assembleur de dupliquer l'instruction. Ainsi, l'assembleur répète les valeurs () autant de fois que le nombre indiqué après la pseudo instruction DB ou DW (C'est à dire avant DUP).

**b) Exemples :**

TAB	DB	5 DUP (?)	;	Définition d'un tableau TAB de 5 de
			;	5 éléments non initialisés
Y	DB	100 DUP (2)	;	Initialisation des éléments de Y à 2
MAT	DB	10 DUP (0,1)	;	Définition d'une matrice MAT de
			;	20 éléments initialisés comme suit :
			;	0,1 , 0,1 , ... , 0,1
			;	un , deux , ... , dix
Z	DW	20 DUP (?)	;	20 éléments non initialisés

c) **NB :**

- On déclare les variables à l'aide des DIRECTIVES.
- L'assembleur attribue à chaque variable une adresse.
- Dans un programme, on repère les variables grâce à leurs noms qui sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre ( De même pour les étiquettes).
- Lors de la déclaration, on peut affecter une valeur initiale à une variable.
- Les valeurs initiales peuvent être données en Hexadécimal.

### 3.3) INSTRUCTIONS :

#### ADDITION

a) **Syntaxe :**

ADD                      Destination , Source

b) **Sémantique :**

Elle additionne deux valeurs numériques sur 8 ou 16 bits. Le résultat reste dans Destination qui peut être une cellule mémoire ou bien un registre.

( Destination      ←      Destination + Source )

c) **Exemples :**

ADD	AX , BX	;	AX	←	AX + BX
ADD	BX , 1	;	BX	←	BX + 1
ADD	ID , 2	;	ID	←	ID + 2

d) **NB :**

ADD	MEM1 , MEM2	;	INTERDITE
-----	-------------	---	-----------

## ADC

a) **Syntaxe :**

ADC	Destination , Source
-----	----------------------

b) **Sémantique :**

Elle addition avec retenue (Carry).

Elle effectue la même opération que ADD et ajoute ensuite la valeur de l'indicateur CF au résultat obtenu.

(Cette instruction réalise donc une opération sur 32 bits).

c) **N.B :**

Destination peut être une cellule mémoire ou un registre.

## SOUSTRACTION

a) **Syntaxe :**

SUB	Destination , Source
-----	----------------------

b) **Sémantique :**

Elle soustrait deux valeurs numériques sur 8 ou 16 bits. Le résultat est placé dans Destination qui peut être un registre ou bien une cellule mémoire.

( Destination ← Destination - Source )

c) **Exemples :**

SUB	AX , BX	;	AX	←	AX - BX
SUB	BX , 1	;	BX	←	BX - 1
SUB	IDENT , 2	;	IDENT	←	IDENT - 2
		;			(Sur 16 bits)

d) **NB :**

SUB	ID1 , ID2	;	INTERDITE
-----	-----------	---	-----------

## SBB

a) **Syntaxe :**

SBB	Destination , Source
-----	----------------------

b) **Sémantique :**

Elle soustraction avec retenue (Carry).

Effectue la même opération que SUB et soustrait ensuite la valeur de l'indicateur CF au résultat obtenu. (Opération sur 32 bits).

c) **NB :**

Le résultat est dans Destination qui peut être un registre ou une cellule mémoire.

## MULTIPLICATION

a) **Syntaxe :**

MUL	Multiplicateur
-----	----------------

b) **Sémantique :**  
Elle réalise une opération de multiplication sur 8 ou 16 bits sur des nombres non :

**Sur 8 bits :** Octet \* AL  $\longrightarrow$  AX (Résultat dans AX sur 16 bits)  
**Sur 16 bits :** MOT \* AX  $\longrightarrow$  AX , DX (Résultat sur 32 bits)

c) **Exemples :**

MUL	NBRE	
MUL	2	; INTERDITE

d) **NB :**  
Les deux formes possibles sont :  
 MUL      Mem      ou bien      MUL      Registre

## IMUL

a) **Syntaxe :**

	IMUL	Multiplicateur
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		
49		
50		
51		
52		
53		
54		
55		
56		
57		
58		
59		
60		
61		
62		
63		
64		
65		
66		
67		
68		
69		
70		
71		
72		
73		
74		
75		
76		
77		
78		
79		
80		
81		
82		
83		
84		
85		
86		
87		
88		
89		
90		
91		
92		
93		
94		
95		
96		
97		
98		
99		
100		

b) **Sémantique :**  
Elle est identique à MUL sauf qu'elle manipule des nombres signés.

## DIVISION

a) **Syntax :**

DIV	Diviseur
-----	----------

**b) Sémantique :**  
Elle réalise une opération de division sur 8 ou 16 bits sur des nombres non signés :

<b>Sur 8 bits :</b>	AX / Octet	→	Quotient = AL Reste = AH
<b>Sur 16 bits :</b>	DX , AX / Mot	→	Quotient = AX Reste = DX

c) **Exemples :**

NBRE	DB	5	;	si AX = 14 alors on	14 / 5
...			;	Quotient	AL = 2
DIV	NBRE		;	Reste	AH = 4
DIV	2		;	INTERDITE	

d) **NB :**  
Les deux formes possibles sont :  
DIV          Mem  
DIV          Reg

## IDIV

a) **Syntaxe :**

IDIV	Diviseur
------	----------

b) **Sémantique :**  
Elle est identique à DIV sauf qu'elle manipule des nombres signés.

## MOV

a) **Syntaxe :**

MOV                      Destination , Source

b) **Sémantique :**

Elle déplace des données de Source à Destination

## INC

a) **Syntaxe :**

INC                      Destination

b) **Sémantique :**

Incrémentation de Destination  
( Destination ← Destination + 1 )

## DEC

a) **Syntaxe :**

DEC                      Destination

b) **Sémantique :**

Décrémentation de Destination  
( Destination ← Destination - 1 )

## NEG

a) **Syntaxe :**

NEG                      Reg  
NEG                      Mem

b) **Sémantique :**

Elle calcule le complément à 2 de l'opérande contenu dans un registre ou dans une cellule mémoire.

c) **Exemples :**

Supposons que AX contient 5

NEG      AX                      ;      AX ← - 5  
NEG      AX                      ;      AX ← 5

## AND

a) **Syntaxe :**

AND                      Reg , Mem

b) **Sémantique :**

ET Logique sur les deux opérandes spécifiés.

## OU

a) **Syntaxe :**

- OR                      Reg , Mem
- b) **Sémantique :**  
Ou Logique            sur les deux opérandes spécifiés.

## OU Exclusif

- a) **Syntaxe :**  
XOR                      Reg , Mem
- b) **Sémantique :**  
OU Exclusif            sur les deux opérandes.

## NOT

- a) **Syntaxe :**  
NOT                      Reg  
NOT                      Mem
- b) **Sémantique :**  
NON Logique            sur les deux opérandes.

## COMPARAISON

- a) **Syntaxe :**  
CMP                      Destination , Source
- b) **Sémantique :**  
CMP compare les deux opérandes spécifiés et est suivie par une instruction de branchement.  
Le résultat de la comparaison est indiqué par les indicateurs.  
( Elle soustrait deux valeurs numériques, n'en modifie aucun, et ne retient pas le résultat, mais positionne les indicateurs )  
Destination - Source            :  
CF = 1    implique   Destination < Source    ou bien    ZF = 0  
CF = 0    implique   Destination = Source    ou bien    ZF = 1  
CF = 0    implique   Destination > Source    ou bien    ZF = 0

- c) **Exemple :**  
CMP            AL , 87  
JE                OK  
...

## INTERRUPTION

- a) **Syntaxe :**  
INT                      Type d'IT
- b) **Sémantique :**  
L'interruption INT permet d'interrompre le programme où elle se trouve pour faire appel à une procédure relative au type d'interruption qui peut être :  
Soit de type fonction DOS (INT 21H),  
Soit de type gestion du clavier (INT 16H),  
Soit de type gestion de l'écran (INT 10H),

...

**c) Exemple :**

INT            21H  
(Mettre dans AH le numéro de la fonction appelée)

## CHARGEMENT (Load Effectif Adress)

**a) Syntaxe :**

LEA            Destination , Source

**b) Sémantique :**

Chargement de l'adresse effective .

**c) Exemple :**

LEA            BX , [ DI ] ; équivalente à : MOV    BX , DI

LEA            BX , IDT ; équivalente à : MOV    BX , OFFSET IDT

## BRANCHEMENTS

**Branchement inconditionnel :**

JMP            Cible

**Branchement conditionnel :**

JGE            Cible ( ou JNL            Cible )  
Saut si  $\geq$  ... ( ou saut si  $<$  ... )

JNE            Cible ( ou JNZ            Cible )  
Saut si non égal à ... ( ou Saut si non nul ou non égal à Zéro )

JNGE            Cible ( ou JL            Cible )  
Saut si non  $\geq$  ... ( ou saut si  $<$  ... )

JE            Cible ( ou JZ            Cible )  
Saut si  $=$  ... équivalente à si Saut ZF = 1 ...  
ou  
Saut si résultat NUL (Zéro) équivalente à Saut si ZF = 1 ....  
JS            Cible ( ou JNS            Cible )  
Saut si signe... équivalente à Saut si SF = 1  
ou  
Saut si Non Signe ... équivalente à Saut si SF = 0

JO            Cible ( ou JNO            Cible )  
Saut si Overflow équivalente à Saut si OF = 1  
ou



Saut si Not Overflow ... équivalente à Saut si OF = 0

## BOUCLES

a) **Syntaxe :**

LOOP Cible

b) **Sémantique :**

Décrémente CX

**SI** CX  $\neq$  0 **alors aller à** Cible

LOOPE Cible

LOOPZ Cible

Identique chacune à LOOP Cible :

Décrémente CX

**Si** ZF = 1 **et** CX  $\neq$  0 **alors aller à** Cible

LOOPNE Cible

Identique à LOOP

Décrémente CX

**Si** ZF = 0 **et** CX  $\neq$  0 **alors aller à** Cible

## DECALAGES

### SAL

a) **Syntaxe :**

SAL Destination , Compte

b) **Sémantique :**

Décalage arithmétique à GAUCHE.

Destination contient la valeur à décaler.

Compte contient le nombre de fois qu'il faut décaler (On peut mettre Compte dans CL)

c) **Exemple :**

```
MOV    CL, 4
MOV    AH, 0FH ; AH ← 00001111
SAL    AH, CL  ; AH ← 11110000
```

### SAR

a) **Syntaxe :**

SAR Destination , Source

b) **Sémantique :**

Décalage arithmétique à DROITE.

c) **Exemple :**

```
MOV    AH, 0FH ; AH ← 00001111
MOV    CL, 4
SAR    AH, CL  ; AH ← 00000000
```

d) **NB :**

Ces cas sont INTERDITS :

SAR	REG , 3
SAL	REG , 3

## SHL

a) **Syntaxe :**

SHL Destination , Source

b) **Sémantique :**

Décalage Logique GAUCHE.

c) **Exemple :**

MOV	AH , 80H	;	AH	←	10000000
SHL	AH , 1	;	AH	←	00000000

## SHR

a) **Syntaxe :**

SHR Destination , Source

b) **Sémantique :**

Décalage Logique DROIT

c) **Exemple :**

MOV	AH , 1	;	AH	←	00000001
SHR	AH , 1	;	AH	←	00000000

d) **NB :**

Ces cas sont INTERDITS :

SHL	REG , 2
SHR	REG , 2

## ECHANGE

a) **Syntaxe :**

XCHG Destination , Source

b) **Sémantique :**

Elle échange le contenu des deux opérandes Destination et Source.

## 4) PROCEDURES :

4.1) **Déclaration de Procédure :**

Le corps de la procédure suit juste l'instruction étiquetée ENDP du segment de code.

CALCUL PROC NEAR ; Procédure nommée CALCUL

	.	;	NEAR	indique que la procédure
	.	;	CALCUL	est située dans le même
	.	;		segment que l'appelant.
	RET	;		Dernière instruction (Retour)
CALCUL	ENDP	;		Fin de la procédure CALCUL

#### 4.2) Appel de procédure

#### 4.3) :

CALL CALCUL ; CALL nom de la procédure

#### 4.4) Passage de paramètres :

-- REGISTRES  
-- PILE

#### Exemple :

SOMME	PROC	NEAR
	ADD	AX , BX
	RET	
SOMME	ENDP	

; Passage de paramètres par registres :

MOV	AX , 6
MOV	BC , TRUC
CALL	SOMME
MOV	TRUC , AX

## 5) PILE :

#### 5.1) Notion de pile :

Une pile est une zone de mémoire et un pointeur qui conserve l'adresse du sommet de pile.

La pile est souvent utilisée pour sauvegarder temporairement les contenus des registres.

#### 5.2) Les instructions :

**PUSH** Registre : Empile le contenu du registre dans la pile.  
**POP** Registre : Dépile la valeur en haut de pile et la range dans le registre.

**Exemple :**

On suppose que AX et BX contiennent des valeurs :

```

PUSH    AX           ; Empilement des valeurs contenues
PUSH    BX           ; dans AX et BX
MOV     BX, TRUC     ; Utilisation des registres
ADD     AX, BX       ; AX et BX
MOV     TRUC, BX
POP     BX           ; Récupération des anciennes
POP     AX           ; valeurs de AX et de BX

```

**5.3) Les registres SP et SS :**

SS (Stack Segment) : registre segment contient l'adresse du segment de pile courant.

SP (Stack Pointer) : Contient le déplacement du sommet de la pile.

**NB :**

L'instruction **PUSH** effectue les opérations :

```

SP      ← SP - 2
[ SP ] ← Valeur du registre

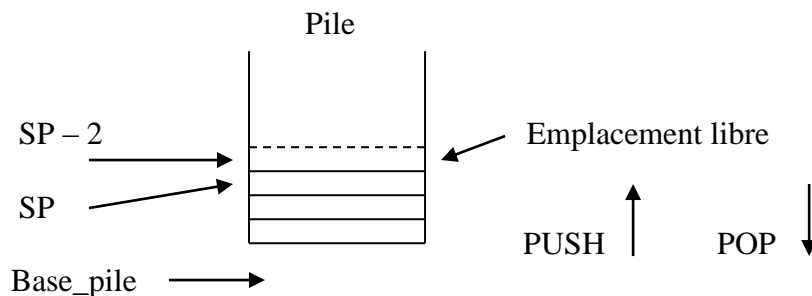
```

L'instruction **POP** effectue les opérations :

```

Registre (Destination) ← [ SP ]
SP      ← SP + 2

```



**5.4) Déclaration d'une pile :**

Pour utiliser une pile en assembleur, il faut déclarer un segment de pile et y réserver un espace suffisant. Ensuite, il est nécessaire d'initialiser les registres SS et SP.

**Exemple :**

```

SEG_pile    SEGMENT    STACK
              DW        100 DUP(?)
Base_pile   EQU        THIS WORD
SEG_pile    ENDS

```

Noter que le mot clé `STACK` indique à l'assembleur qu'il s'agit d'un segment de pile.

Afin d'initialiser `SP`, il faut récupérer l'adresse du bas de pile grâce à :

```
Base_pile    EQU    THIS WORD
```

## 6) **MACRO :**

Une **MACRO** est un symbole qui remplace un ensemble de lignes de codes dans un programme.

Elle se situe dans le code segment.

### 6.1) **Déclaration d'une MACRO :**

```
Nom          MACRO
               .
               .
               .
            ENDM
```

### 6.2) **APPEL d'une macro :**

A l'aide de son `Nom`.

**NB :**

La différence entre une procédure et une macro est que, lors de l'assemblage du programme, l'appel de la macro (`Nom`) va être remplacé par le corps de la macro (Le programme va donc s'allonger). Par contre, ce n'est pas le cas de la procédure (Elle existe une seule fois dans le programme).

## 7) **Les opérations d'E/S du langage MASM :**

**NB :**

Les opérations d'E/S sont réalisées par le biais du `DOS`, en faisant appel aux fonctions du `DOS` Grâce à l'interruption `21H` (`INT 21H`).

### a) **Lecture d'un caractère** (Avec affichage sur écran) :

```
MOV    AH, 1
INT    21H
```

Le résultat de la lecture se trouve dans `AL`.

### b) **Lecture d'un caractère** (Sans affichage sur écran) :

```
MOV    AH , 8
INT    21H
```

c) **Ecriture d'un caractère** (Affichage sur écran) :

Le caractère se trouve dans DL.

```
MOV    AH , 2
INT    21H
```

d) **Ecriture d'un caractère** (Sur imprimante) :

Le caractère se trouve dans DL.

```
MOV    AH , 5
INT    21H
```

e) **Lecture d'une chaîne de caractères** (A partir du clavier) :

```
MOV    AH , 0AH
LEA    DX , BUFFER
INT    21H
Ou bien
MOV    AH , 0AH
MOV    DX , OFFSET BUFFER
INT    21H
```

Dans les deux cas, le résultat de cette lecture se trouve dans un BUFFER qu'il faut réserver.

f) **Ecriture d'un message** (Sur écran) :

MESSAGE : Chaîne de caractères qui a été déclarée.

```
LEA    DX , MESSAGE
MOV    AH , 9
INT    21H
Ou bien
MOV    DX , OFFSET MESSAGE
MOV    AH , 9
INT    21H
```

g) **Conversion d'un nombre en ASCII :**

```
MOV    DL , NOMBRE
ADD    DL , 48
```

8) **Exemple d'un programme écrit en M A S M :**

```
CDS          SEGMENT          PARA STACK 'PILE'
              DB                256 DUP (0)
CDS          ENDS

LDS          SEGMENT
MESSAGE     DB                ' J'APPRENDS LE MASM ! $'
LDS          ENDS

LPS          SEGMENT
PHRASE     PROC                FAR
              ASSUME            CS : LPS      ; Affectation des
              ASSUME            DS : LDS      ; registres, obligatoires
              ASSUME            SS : CDS      ; dans un programme
```

	MOV	AX , LDS	; on ne peut pas charger
	MOV	DS , AX	; un registre segment
			; qu'à partir d'un autre
	LEA	DX , MESSAGE	
	MOV	AH , 9	
	INT	21H	
	MOV	AH , 4 CH	; Retour au DOS
	INT	21H	
PHRASE	ENDP		
LPS	ENDS		
	END	PHRASE	

### **Lancement d'un programme écrit en M A S M : ( Sous DOS )**

On suppose que le programme a pour nom : ESSAI

Taper :

M A S M      ESSAI.asm

LINK          ESSAI.obj

ESSAI.exe

### **NB :**

Quand on utilise une macro, le programme objet a la même taille et prend le même temps que le programme équivalent sans macro. On économise seulement du temps et de l'espace mémoire pour l'écriture et le stockage du programme source. Par contre, dans le cas d'une procédure, on économise de l'espace mémoire puisque la procédure ne se trouve qu'une seule fois en mémoire (même si elle est appelée plusieurs fois). Mais le temps d'exécution augmente légèrement car il faut sauvegarder l'état de la machine au début de l'exécution de la procédure et le restituer à la fin de son exécution .