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	*copier un registre dans un autre*
MOV R , M	*copier le contenu d'une case mémoire dans un registre*
MOV M , R	*copier un register dans une case mémoire*
MOV R , im	*copier une constante dans un registre*
MOV taille M , im	*copier une constante dans une case mémoire (taille = BYTE ou WORD)*

~~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 pontée par SP

PUSH R_{16}
PUSH word [adr]

~~PUSH im~~ ~~PUSH R_8~~

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_{16}
POP word M

~~POP R_8~~

L'exemple de la figure 2.4 illustre plusieurs situations :

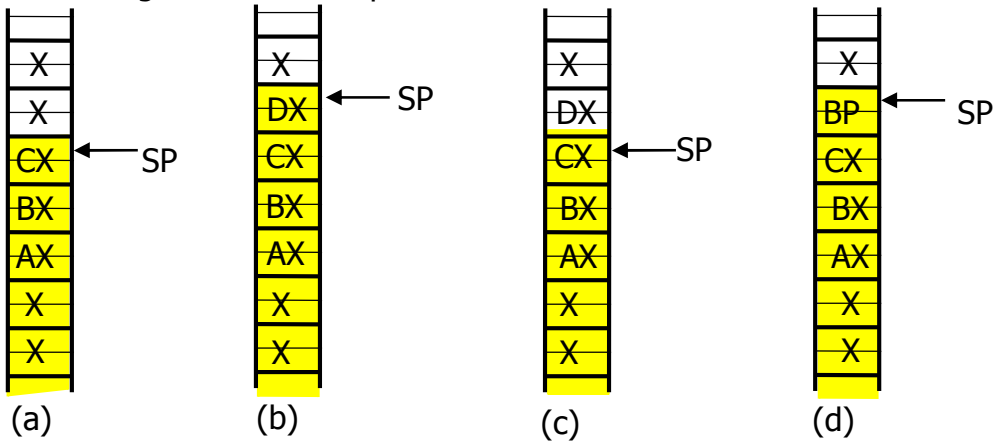


Fig. 2.5 : fonctionnement d'une pile

(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

Dépile 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 → 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 curry avec résultat dans l'opérande destination,

Od + Os + C → Od

INC Op Incrémente l'opérande Op

Op + 1 → 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]

4000h→	25h
	33h

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]

4000h→	FFh
	33h

► Soustraction

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

Od - Os → Od

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

Od - Os - C → Od

DEC Op Décrémente l'opérande Op
Op - 1 → Op

NEG Op Donne le complément à 2 de l'opérande Op : remplace Op par son négatif
Cà2(Op) → 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_8 → AL x Op_8 → AX
MUL Op_{16} → AX x Op_{16} → 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_8
AH	AL

DIV Op_8 ;AX / Op_8 , Quotient → AL , Reste → AH

DIV S_{16} ;DX:AX / S_{16} , Quotient → AX , Reste → DX

DX:AX	Op_{16}
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 → 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 ⊕ 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 AND 0 = 0 (0 = élément absorbant de AND)*
- *x AND 1 = x (1 = 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
-------------------
      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 OR 1 = 1 (1 = élément absorbant de OR)*
- *x OR 0 = x (0 = 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
-------------------
      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 XOR 1 = $\bar{X}$*
- *X XOR 0 = X (0 = 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
-------------------
      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

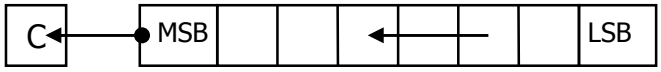
Dans les instructions de décalage, l'opérande k peut être soit une constante (immédiat) soit le registre CL :

```
INST  AX,1  ; décaler AX de 1 bit
INST  BL,4  ; décaler BL de 4 bits
INST  BX,CL ; décaler BX de CL bits
```

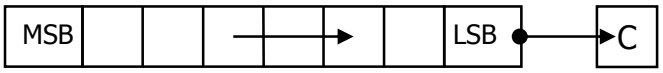
On peut aussi décaler le contenu d'une case mémoire mais il faut préciser la taille

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

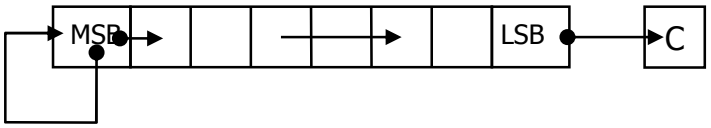
SHL R/M,k (*SHift Logical Left*) décalage logique à gauche de k bits
SAL R/M,k (*SHift Arithmé Left*) décalage arithmétique à gauche



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

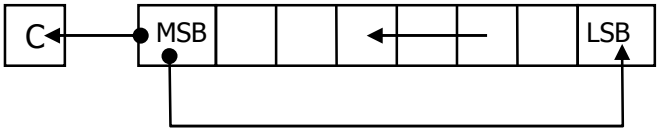


SAR R/M,k (*SHift Arithmetic right*) décalage arithmétique à droite

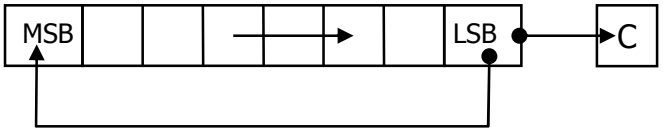


Les décalages arithmétiques permettent de conserver le signe. Ils sont utilisés pour effectuer des opérations arithmétiques comme des multiplications et des divisions par 2.

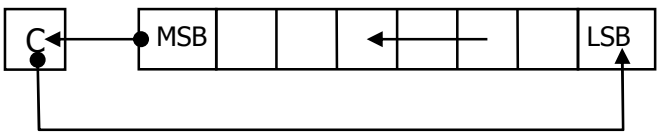
ROL R/M,k (*Rotate Left*) Rotation à gauche



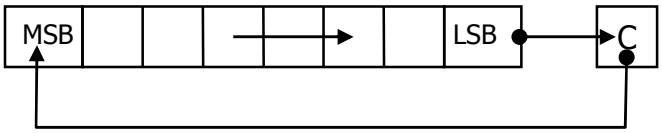
ROR R/M,k (*Rotate Right*) Rotation à droite



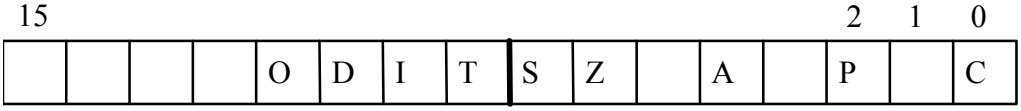
RCL R/M,k (*Rotate Through CF Left*) Rotation à gauche à travers le Carry



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

Après l'opération, on a : AH : S Z A P C

SAHF

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

PUSHF

Empile le registre d'état,

POPF

Dépile 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). Quant 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.

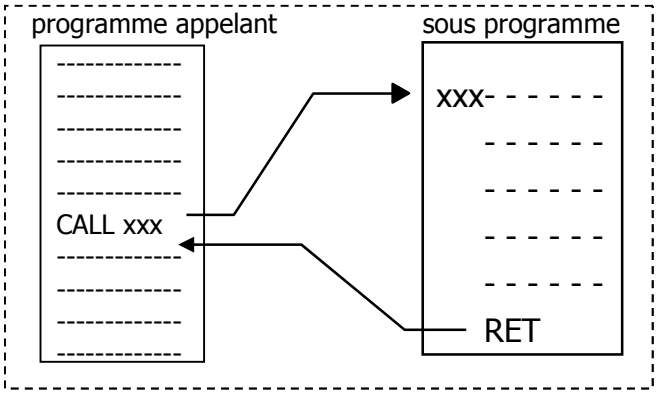
▶ Branchements inconditionnels

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 Grater*) aller à la ligne xyz si plus grand (signé). C'est-à-dire si (S ⊕ O) + Z = 1

JGE xyz (*Jump if Grater or Equal*) aller à la ligne xyz si plus grand ou égal (signé). C'est-à-dire si S ⊕ O = 0

JL xyz (*Jump if Less*) aller à la ligne xyz si plus petit (signé). C'est-à-dire si S ⊕ 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 ⊕ 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 dan 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] dan 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 pou 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 peux 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_8~~

- 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~~

A signifie Accumulateur qui peut être AL ou AX selon la taille de l'opération 8 bits ou 16 bits
X est l'extension de l'accumulateur. (8 bits : X =AH→ X:A = AH:AL = AX) (16 bits : X = DX→ X:A = DX:AX)

***** : *Positionné selon le résultat*
? : *Modifié, résultat non défini*

Transfert		O	S	Z	A	P	C
MOV D , S	D ← S						
PUSH R/M	empile R/M (16 bits)						
POP R/M	dépile R/M (16 bits)						
XCHG D , S	D ⇆ S						
XLAT	AL ← [BX+AL]						
LEA R_{16} , M	R_{16} ← offset de M						
LDS R_{16} , [a]	R_{16} ← [a], DS←[a+2]						
LES R_{16} , [a]	R_{16} ← [a], ES←[a+2]						
Arithmétique							
ADD D , S	D ← S + D	*	*	*	*	*	*
ADC D , S	D ← S + D + C	*	*	*	*	*	*
INC D	D ← D + 1	*	*	*	*	*	!
SUB D,S	D ← D - S	*	*	*	*	*	*
SBB D,S	D ← D – S - C	*	*	*	*	*	*
NEG D	D ← -D	*	*	*	*	*	*
CMP D,S	D - S	*	*	*	*	*	*
DEC D	D ← D - 1	*	*	*	*	*	*
MUL S	X:A ←A x S	*	?	?	?	?	*
IMUL S	X:A ←A x S (signée)	*	?	?	?	?	*
DIV S	A←(X:A) / S, X←Rst	?	?	?	?	?	?
IDIV S	division signée	?	?	?	?	?	?
CBW	Byte(AL) to word(AX) (signé)						
CWD	Word(AX) to Dword (DX|AX) (signé)						
DAA	Dec. adj. after ADD	?	*	*	*	*	*
AAA	ascii adj after ADD	?	?	?	*	?	*
DAS	dec adj after SUB	?	*	*	*	*	*
AAS	ascii adj after SUB	?	?	?	*	?	*
AAM	ascii adj after MUL	?	*	*	?	*	?
AAD	ascii adj before DIV	?	*	*	?	*	?
Logique							
NOT R/M	R/M ← R/M						
AND D , S	D ← D AND S	0	*	*	?	*	0
OR D , S	D ← D OR S	0	*	*	?	*	0
XOR D , S	D ← D XOR S	0	*	*	?	*	0
TEST D , S	D AND S	0	*	*	?	*	0
SHL/SHL R/M		*	*	*	?	*	*
SHR R/M		*	*	*	?	*	*
SAR R/M		*	*	*	?	*	*
ROL R/M							*
ROR R/M							*
RCL R/M							*
RCR R/M							*
Chaînes							
MOVSb/w	[ES:DI] ←[DS:SI],						
SCASb/w	AL/AX – [ES:DI];	*	*	*	*	*	*
LODSb/w	AL/AX ← [DS:SI]						
CMPSb/w	[ES:DI] - [DS:SI],	*	*	*	*	*	*
STOSb/w	AL/AX → [ES:DI]	*	*	*	*	*	*

Branchement		O	S	Z	A	P	C
JMP a	branche à l'adresse a						
JE/JZ a	saut si = (Z levé)						
JNE/JNZ a	saut si ≠ (Z baissé)						
JA a	saut si > (non signé)						
JAE a	saut si ≥ (non signé)						
JB a	saut si < (non signé)						
JBE a	saut si ≤ (non signé)						
JG a	saut si > (signé)						
JGE a	saut si ≥ (signé)						
JL a	saut si < (signé)						
JLE a	saut si ≤ (signé)						
JC a	saut si C = 1						
JNC a	saut si C = 0						
JO a	saut si O = 1						
JNC a	saut si C = 0						
JP/JPE a	saut si P = 1						
JNP/JPO a	saut si P = 0						
JS a	saut si S = 1						
JNS a	saut si S = 0						
CALL a	saut à sous programme						
RET	retour de sous programme						
LOOP A	dec CX, saut si CX ≠ 0						
LOOPZ a	dec CX, saut si :CX ≠ 0 et Z = 1						
LOOPNZ a	dec CX, saut si :CX ≠ 0 et Z = 0						
JCXZ a	saut si CX = 0 (∀ Z)						
action sur Indicateurs							
LAHF	AH ← RE_L						
SAHF	RE_L ← AH		*	*	*	*	*
PUSHF	empile RE						
POPF	dépile RE	*	*	*	*	*	*
CLC	Clear Carry flag						0
STC	Set Carry flag						1
CMC	complémente Carry						*
CLD	Clear Direction flag						
STD	Set Direction flag						
CLI	Clear Interrupt flag						
STI	Set Interrupt flag						
Divers							
INT n	déclenche l'interrupt n						
INTO n	interrupt si Overflow						
IRET	Retour d'interruption	*	*	*	*	*	*
HALT	entre en mode vail						
WAIT	entre en attente						
NOP	ne fait rien						
IN al/ax,port	Lit un port d' E/S						
out port,al/ax	écrit dans un port						