# ~ ASSEMBLEUR 8086 ~

## ### LES REGISTRES GENERAUX (codE en 16bits)

Ces registres servent a stocker des valeurs :

**AX** – sert d'accumulateur et est principalement utilisE lors des operations arithmetique et logique
**BX** – utilise comme operande dans les calculs
**CX** – est utilisE comme compteur dans les structures iteratives (boucle)
**DX** – meme rôle que AX et notamment dans la division et la multiplication, intervient egalement dans les operations d'entrEes/sorties

NOTE: chacune de ces registres se divisent en deux parties (16bits) => H: high (8bits) | L: low (8bits)

---

## ### LES JEUX D'INSTRUCTION

## # RAPPEL NOTATION

| DEC | BIN | HEX |
|---|---|---|
| 0 | 0000 | 0000 (00h) |
| 1 | 0001 | 0001 (01h) |
| 2 | 0010 | 0002 (02h) |
| 3 | 0011 | 0003 (03h) |
| 4 | 0100 | 0004 (04h) |
| 5 | 0101 | 0005 (05h) |
| 6 | 0110 | 0006 (06h) |
| 7 | 0111 | 0007 (07h) |
| 8 | 1000 | 0008 (08h) |
| 9 | 1001 | 0009 (09h) |
| 10 | 1010 | 000A (0Ah) |
| 11 | 1011 | 000B (0Bh) |
| 12 | 1100 | 000C (0Ch) |
| 13 | 1101 | 000D (0Dh) |
| 14 | 1110 | 000E (0Eh) |
| 15 | 1111 | 000F (0Fh) |

---

## # AFFECTATION (mov)

mov x, y => affectation de y dans le registre x (x = y | ex: mov AX, 01h)

NOTE: si on ecrit AX tout entier, alors on utilise directement la capacitE mAXimale de ce registre (16bits)

## # READ & WRITE

**# caractere (write)**

```
mov ah, 02h
mov dx, '<caractere>'
int 21h ;interupteur
```

**# caractere (read)**

```
mov ah, 01h
int 21h
```

**# chaine de caractere (write)**

```
include emu8086.inc
print 'Hello word'
```

**# chaine de caractere (read)**

```
mov cx, 64 ;equiv 100(dec) => counter

INPUT:
mov ah, 01h
int 21h
cmp al, 00h ;00h equiv touche [ENTRER]
je END
loop INPUT

END:
```

NOTE:
mov ah, 01h => lecture avec echo
mov ah, 08h => lecture sans echo

---

## # ADDITION (add)

**[somme de 2 valeurs => AX + BX]**

```
mov AX, 03h ;affecter 03h dans le registre AX
mov BX, 01h ;affecter 01h dans le registre BX
add AX, BX ;additionner AX et BX puis affecter le resultat dans AX (AX = 04h)
```

**[somme de 3 valeurs => AX + BX + DX]**

```
mov AX, 02h
mov BX, 01h
mov CX, 00h ;il faut initialiser CX car on a besoin d'une valeur pour faire le 1er addition
mov DX, 05h
add CX, AX ;equiv a CX += AX (CX = 00h + 02h = 02h)
add CX, BX ;equiv a CX += BX (CX = 02h + 01h = 03h)
add CX, DX ;equiv a CX += DX (CX = 03h + 05h = 08h)
```

## # SOUSTRACTION (sub)

```
mov AX, 01h
mov BX, 03h
sub BX, AX ;BX -= AX (BX = 03h - 01h = 02h)
```

---

## # MULTIPLICATION (mul)

**[exemple 1] - multiplication de 2 valeurs**

```
mov AX, 05h
mov BX, 02h
mul BX ;equiv a AX *= BX (AX = 05h * 02h = A)
```

**[exemple 2] - carrE d'une valeur**

```
mov AX, 03h
mul AX ;equiv a AX *= AX (AX = 03h * 03h = 09h)
```

NOTE:
- Si on ecrit mul <registre> => le resultat est egale a la valeur du <registre> * valeur du AX
- le resultat d'une multiplication sera toujours affectE au registre AX

---

## # DIVISION (div)

```
mov AX, 0Ah
mov BX, 02h
div BX ;equiv AX /= BX (AX = AX / BX)
```

---

## # EMPILAGE et DEPILAGE (push/pop) - LIFO (Last In First Out)

push (empilage) sert a stocker temporairement la valeur d'un registre dans une pile (stack dans emu8086)
pop (depilage) sert a recuperer la derniere valeur de la pile et l'affecte dans un registre

**[exemple 1] - ancienne valeur du registre**

```
mov AX, 04h
push AX ;stocker la valeur de AX dans la pile "0100:FFFC => 04h"
mov AX, 01h ;ici AX vaut 01h
pop AX ;recuperer l'ancienne valeur de AX dans la pile (ici AX vaut 04h)
```

**[exemple 2] - permutation de 2 valeurs AX <=> BX**

```
mov AX, 06h
mov BX, 03h
push AX ;on insere la valeur de AX dans la pile
push BX ;on insere la valeur de BX dans la pile
```

pop AX ;on recupere la derniere valeur de la pile (Last In => BX) et l'affecte dans AX
pop BX ;on recupere le reste qui est AX et l'affecte dans BX

---

## # COMPARAISON/SAUT (cmp + jmp)

```
mov AX, 06h
mov BX, 09h
cmp AX, BX ;compare AX a BX
jl cas1 ;si AX < BX appeler l'etiquette 'cas1' qui affecte 01h a DX
jg cas2 ;si AX > BX appeler l'etiquette 'cas2' qui affecte 00h a DX

cas1:
mov DX, 01h
jmp END ;cette instruction permet d'ignorer le 'cas2' si 'cas1' est vraie

cas2:
mov DX, 00h

END:
```

NOTE:
- cmp = compare 2 registres (<registre1> est comparE au <registre2>)
- jmp = le jump (saut) permet d'executer une/plusieurs instruction(s) definie(s) a l'aide d'une etiquette
- jl = si la valeur du <registre1> est inferieur au <registre2>
- je = si la valeur du <registre1> est egale au <registre2>
- jg = si la valeur du <registre1> est superieur au <registre2>

---

## # PARITE

```
mov  AX, 02h ;en bin => 0010 (1er bit a droite => 0, donc pair)
test AX, 1 ;signifie tester le 1er (1) bit a droite
jz PAIR ;z est le flag du resultat null qui retourne 1 (vraie) si le resultat est 0 (AX est pair) et retourne 0 (fausse) si le resultat est 1 (AX est impair)
jmp IMPAIR

PAIR:
mov DX, 02h ;affecter 02h dans DX si AX est pair
jmp END

IMPAIR:
mov DX, 01h ;affecter 01h dans DX si AX est impair

END:
```

NOTE:
(voir section rappel notation pour verifier)
- Dans la valeur binaire d'un nombre entier, le 1er bit (a droite) de ce nombre est toujours egale a '0' si ce dernier est 'pair'
- Dans la valeur binaire d'un nombre entier, le 1er bit (a droite) de ce nombre est toujours egale a '1' si ce dernier est 'impair'

---

## # OR (test)

```
mov AH, 02h
mov AL, 01h
mov BH, 03h
mov BL, 06h

cmp AH, AL ;compare AH a AL
jg code ;if AH > AL jump to code
cmp BH, BL ;compare BH a BL
jg code ;if BH > BL jump to code

mov CL, 00h
add CL, AL ;CL += AL
add CL, BL ;CL += BL
jmp END

code:
mov DL, 00h
add DL, AL ;DL += AL
add DL, BL ;DL += BL

END:
```

TRAD: traduction en langage C

```
int AH = 02h;
int AL = 01h;
int BH = 03h;
int BL = 06h

if (AH > AL) || (BH > BL){
   int DL = AL + BL;
}else{
   int CL = AL + BL;
}
```

## # BOUCLE (loop)

**[exemple 1]**

```
mov CX, 08h ;CX est le nombre d'iteration (counter)
mov AX, 04h
mov BX, 07h

code: ;empiler la valeur de AX et BX 8fois
push AX
push BX
loop code
```

**[exemple 2]**

```
mov CX, 09h
mov BX, 02h

code:
add BX, 06h
loop code
```

**[exemple 3] - meme code que 'exemple 2' sans utiliser loop et CX**

```
mov AX, 00h ;sert de counter
mov BX, 00h

code:
add BX, 01h
inc AX ;equiv a AX++
cmp AX, 09h ;condition de poursuite = nbr d'iteration + 1
je END ;si AX = 09h alors fin de la boucle (appeler END:)
jmp code ;sinon appeler code:

END:
```

NOTE:
- inc <registre>: incrementation d'un <registre>
- dec <registre>: decrementation d'un <registre>

---

## # DEFINITION DE DONNEE et SEGMENTATION (DB & DW)

**=> la segmentation est une maniere de coder professionnelement dans le langage d'assemblage**

**data segment** ;permet de specifier les donnees a utiliser
var1 db <value> ;stocker <value> dans var1 | db signifie 'define byte' => permet un stockage de 1byte (8bits = 1octet)
var2 dw <value> :stocker <value> dans var2 | dw signifie 'define word' => permet un stockage de 2byte (16bits = 2octet)
ends

**stack segment** ;permet de definir un case memoire
db 128 dup(0) ;reservation d'une case memoire de 8bits * 128cases et defini la 1ere case a 0
ends

**code segment** ;le coprs du code
mov AX, data ;le registre AX sert d'intermediaire pour stocker les donnees dans 'data'
mov DS, AX ;les donnees dans 'data' ne peuvent par etre affectEs directement dans le DS (Data Segment), c'est pourquoi on utilise AX
mov AX, stack
mov SS, AX ; la meme chose pour le SS (Stack Segment)

mov AX, 00h ;reinitialiser la valeur de AX

mov AL, var1 ;affectation de var1 dans AL (8bits)
mov BX, var2 ;affectation de var2 dans BX

mov AX, 4C00h ;les 3 lignes suivantes permettent de terminer le programme plus proprement
int 21h
ends

---

## # TABLEAU

data segment
;[exemple1]: tableau defini
tab db 1,2,3,4
;[exemple2]: tableau non defini (5 cases memoires)
tab db 5 dup(0)
ends

stack segment ;facultatif
db 128 dup(0)
ends

code segment
mov AX, data
mov DS, AX
mov AX, stack
mov SS, AX

mov AX, 00h ;reinitialiser la valeur de AX

mov SI, 00h ;initialiser l'index
mov CX, 05h ;nombre d'iteration

;READ TABLE
INPUT:
mov AH, 01h ;permet la lecture

```
int 21h ;permet la lecture
mov tab[SI], AL ;la valeur lue sera dans AL (8bits) | affecter cette valeur dans le tableau
inc SI ;incrementation de l'indice SI
loop INPUT

mov SI, 00h ;reinitialiser l'index
mov CX, 05h ;nombre d'iteration

;WRITE TABLE
OUTPUT:
mov DL, tab[SI] ;stocker l'element du tableau dans DL (8bits)
mov AH, 02h ;permet l'ecriture
int 21h ;permet l'ecriture
inc SI
loop OUTPUT

mov AX, 4C00h
int 21h
ends
```