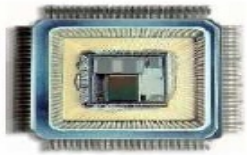
École Nationale des Sciences de l'Informatique

Module:

Architecture et Fonctionnement des Microprocesseurs

Chapitre 6:

Les Instructions de branchements ,les fonctions d'affichage et les sous programmes du Microprocesseur 8086



A.U. 2022-2023

Les instructions du µP 8086

Le 8086 offre 89 types d'instructions (89 mnémoniques) de base qu'on peut diviser en 6 groupes :

- Les instructions de transfert des données.
- Les instructions arithmétiques.
- Les instructions logiques.
- Les instructions de traitement de chaînes de données.
- Les instructions de rupture de séquence.
- Les instructions de contrôle de l'état du processeur.

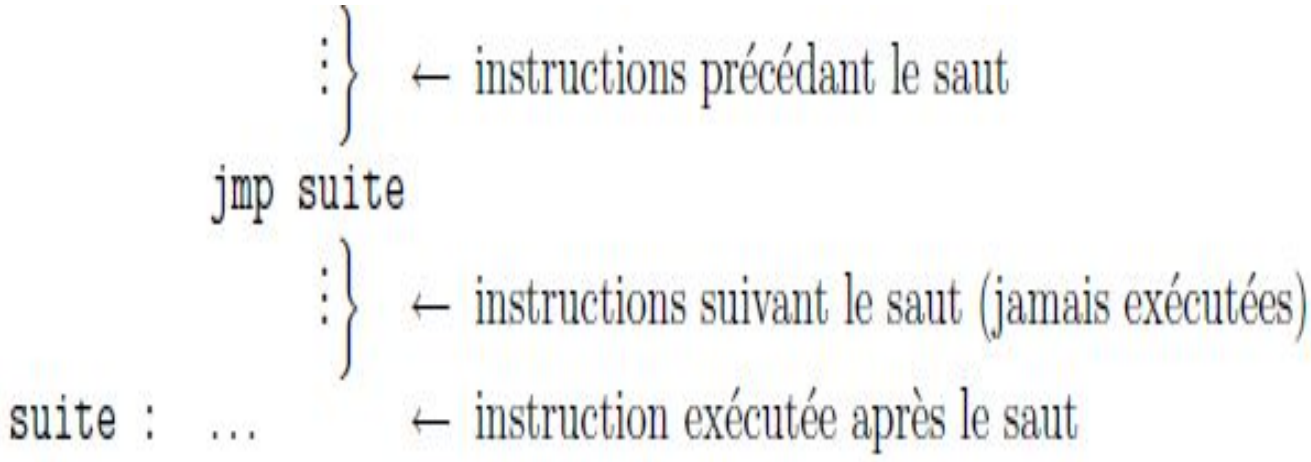
Les instructions de branchement

Les instructions de branchement ou saut permettent de modifier l'ordre d'exécution des instructions du programme en fonction de certaines conditions. Il existe 3 types de saut :

- ***saut inconditionnel ;***
- ***sauts conditionnels ;***
- ***appel de sous-programmes***.

• **Instruction de saut inconditionnel : *JMP label***

Cette instruction effectue un saut (jump) vers le label spécifié. Un label ou étiquette est une représentation symbolique d'une instruction en mémoire :



Les instructions de comparaison: CMP et TEST

Syntaxe:

CMP Destination,Source

Destination = registre / case mémoire

Source = registre / case mémoire / valeur

Exemples:

CMP AL,BL ***Set 'Zf=1' flag if AL = BL et Set 'Sf=1' flag if AL < BL.***

CMP BL,13 ***Set 'Zf=1' flag if BL = 13 et Set 'Sf=1 ' flag if BL < 13.***

CMP CL,[20] ***Set 'Zf=1' flag if CL = [20] et Set 'Sf=1' flag if CL < [20].***

Syntaxe:

TEST *(« Test for bit pattern ») Syntaxe :*

TEST Destination, Source

Description : Effectue un ***ET logique*** *bit à bit entre Destination et Source. Le résultat n'est pas conservé, donc Destination n'est pas modifié. Seuls les flags sont affectés.*

Cet opérateur est souvent utilisé pour tester certains bits de Destination. Indicateurs affectés : CF, OF, PF, SF, ZF

Remarque : l'instruction JMP ajoute au registre IP (pointeur d'instruction) le nombre d'octets (distance) qui sépare l'instruction de saut de sa destination. Pour un saut en arrière, la distance est négative (codée en complément à 2).

- **Instructions de sauts conditionnels : J*condition label***

Un saut conditionnel n'est exécuté que si une certaine condition est satisfaite, sinon l'exécution se poursuit séquentiellement à l'instruction suivante.

La condition du saut porte sur l'état de l'un des indicateurs d'état du microprocesseur :

CF : indicateur de retenue (carry) ;
PF : indicateur de parité;
ZF : indicateur de zéro ;
SF : indicateur de signe ;
OF : indicateur de dépassement (overflow).

instruction	nom	condition
`JZ label`	Jump if Zero	saut si ZF = 1
`JNZ label`	Jump if Not Zero	saut si ZF = 0
`JE label`	Jump if Equal	saut si ZF = 1
`JNE label`	Jump if Not Equal	saut si ZF = 0
`JC label`	Jump if Carry	saut si CF = 1
`JNC label`	Jump if Not Carry	saut si CF = 0
`JS label`	Jump if Sign	saut si SF = 1
`JNS label`	Jump if Not Sign	saut si SF = 0
`JO label`	Jump if Overflow	saut si OF = 1
`JNO label`	Jump if Not Overflow	saut si OF = 0
`JP label`	Jump if Parity	saut si PF = 1
`JNP label`	Jump if Not Parity	saut si PF = 0

Remarque : les indicateurs sont positionnés en fonction du résultat de la dernière opération.

Exemple :

```
                ⋮   ← instructions précédant le saut conditionnel
          jnz suite
                ⋮   ← instructions exécutées si la condition ZF = 0 est vérifiée
suite :   ...       ← instruction exécutée à la suite du saut
```

Remarque : il existe un autre type de saut conditionnel, les sauts arithmétiques. Ils suivent en général l'instruction de comparaison : ***CMP opérande1,opérande2***

condition	nombres signés	nombres non signés
=	JEQ label	JEQ label
>	JG label	JA label
=>	JGE label	JAE label
<	JL label	JB label
<=	JLE label	JBE label
≠	JNE label	JNE label

Exemple :

```
          cmp ax,bx
          jg superieur
          jl inferieur
superieur :  ...
               ⋮
inferieur :  ...
```

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

L'équivalent de quelques instructions du langage C en assembleur

- if then else

```
If ax =1
   bx = 10;
else {
       bx = 0;
       cx = 10;
     }
```

Assembleur

```
CMP AX, 1
      JNZ  Else

      MOV BX,10
       JMP ET2
Else:  MOV BX,0
       MOV CX,10
ET2 : Ret
```

instruction	nom	condition
JZ label	Jump if Zero	saut si ZF = 1
JNZ label	Jump if Not Zero	saut si ZF = 0
JE label	Jump if Equal	saut si ZF = 1
JNE label	Jump if Not Equal	saut si ZF = 0
JC label	Jump if Carry	saut si CF = 1
JNC label	Jump if Not Carry	saut si CF = 0
JS label	Jump if Sign	saut si SF = 1
JNS label	Jump if Not Sign	saut si SF = 0
JO label	Jump if Overflow	saut si OF = 1
JNO label	Jump if Not Overflow	saut si OF = 0
JP label	Jump if Parity	saut si PF = 1
JNP label	Jump if Not Parity	saut si PF = 0

- La boucle FOR

- For (k=0; k<=10; k++)
- bx = bx + k;

Assembleur

```
        MOV BX,0
        MOV CX,0 ;cx=k
For:    CMP CX,10
        JA  Endfor
        ADD BX,CX
        INC CX
        JMP  For
Endfor: hlt
```

condition	nombres signés	nombres non signés
$=$	JEQ label	JEQ label
$>$	JG label	JA label
$<$	JL label	JB label
$\neq$	JNE label	JNE label

Exemple :

```
                cmp ax,bx
                jg superieur
                jl inferieur
superieur :     ...
                 ⋮
inferieur :     ...
```

instruction	nom	condition
JZ label	Jump if Zero	saut si ZF = 1
JNZ label	Jump if Not Zero	saut si ZF = 0
JE label	Jump if Equal	saut si ZF = 1
JNE label	Jump if Not Equal	saut si ZF = 0
JC label	Jump if Carry	saut si CF = 1
JNC label	Jump if Not Carry	saut si CF = 0
JS label	Jump if Sign	saut si SF = 1
JNS label	Jump if Not Sign	saut si SF = 0
JO label	Jump if Overflow	saut si OF = 1
JNO label	Jump if Not Overflow	saut si OF = 0
JP label	Jump if Parity	saut si PF = 1
JNP label	Jump if Not Parity	saut si PF = 0

- WHILE

- bx = 5
- while (bx >0)
  bx = bx -1;

Assembleur

```
        MOV BX,5
while:  CMP BX,0
        JLE Endwhile
        DEC BX
        JMP while
Endwhile: hlt
```

condition	nombres signés	nombres non signés
=	JEQ label	JEQ label
>	JG label	JA label
<	JL label	JB label
≠	JNE label	JNE label

Exemple :

```
                cmp ax,bx
                jg superieur
                jl inferieur
superieur :     ...
                  ⋮
inferieur :     ...
```

instruction	nom	condition
JZ label	Jump if Zero	saut si ZF = 1
JNZ label	Jump if Not Zero	saut si ZF = 0
JE label	Jump if Equal	saut si ZF = 1
JNE label	Jump if Not Equal	saut si ZF = 0
JC label	Jump if Carry	saut si CF = 1
JNC label	Jump if Not Carry	saut si CF = 0
JS label	Jump if Sign	saut si SF = 1
JNS label	Jump if Not Sign	saut si SF = 0
JO label	Jump if Overflow	saut si OF = 1
JNO label	Jump if Not Overflow	saut si OF = 0
JP label	Jump if Parity	saut si PF = 1
JNP label	Jump if Not Parity	saut si PF = 0

- SWITCH

- switch (n) {
  case 1: ....; break;
  case 2: .....; break;
  default: .....;
  }

Assembleur

```
        CMP  n,1
        JNE case2
        .............
        JMP endswitch
case2:  CMP n,2
        JNE  default
        ..........
        JMP endswitch
default:    ............
endswitch: ............
```

Écrire le code de l'instruction

```
if (a>b) && (c <= d)
        {
          ..............
      }
```

En assembleur

```
if:   cmp a, b
        jle  endif

      cmp c, d
        jg  endif
        ..............
endif:
```

Exercice: coder en assembleur les instructions suivantes:

1.
```
if (a >b) || (c > d))
   {
        }
```

2.
```
for (i=1; i < 10; i++)
  {
      }
```

```
cmp a, b
 jle  case1
  case2: ......
   .......
       jmp endif
case1 : cmp c, d
     jle  endif
      jmp case2

     endif:hlt
```

2)

```
 Mov cx,1
for:cmp cx, 10
     JAE
     jmp enfor
      ET1: inc  cx
       jmp for
      enfor:hlt
```

condition	nombres signés	nombres non signés
=	JEQ label	JEQ label
>	JG label	JA label
<	JL label	JB label
≠	JNE label	JNE label

Exemple :

```
                cmp ax,bx
                jg superieur
                jl inferieur
superieur :     ...
                    ⋮
inferieur :     ...
```

instruction	nom	condition
JZ label	Jump if Zero	saut si ZF = 1
JNZ label	Jump if Not Zero	saut si ZF = 0
JE label	Jump if Equal	saut si ZF = 1
JNE label	Jump if Not Equal	saut si ZF = 0
JC label	Jump if Carry	saut si CF = 1
JNC label	Jump if Not Carry	saut si CF = 0
JS label	Jump if Sign	saut si SF = 1
JNS label	Jump if Not Sign	saut si SF = 0
JO label	Jump if Overflow	saut si OF = 1
JNO label	Jump if Not Overflow	saut si OF = 0
JP label	Jump if Parity	saut si PF = 1
JNP label	Jump if Not Parity	saut si PF = 0

Exemple

```
jmp debut
  TAB1 db 1 dup(05h,30h,50h,5fh)
debut:
 mov bl,30h
  mov cx,4
 mov si,0
Xor dl,dl
      ET1:mov al,TAB1[SI]
       cmp al,bl
       jg sup
       jl inf
       je  egal
                    sup:add dl,TAB1[si]
                  inc si
          loop ET1
          jmp fin
          inf:  sub dl,TAB1[si]

        inc si
          loop ET1
                egal:or dl,TAB1[si]

        inc si
          loop ET1
                fin:  ret
```

Les instructions de branchement après cmp op1,op2

Signées :

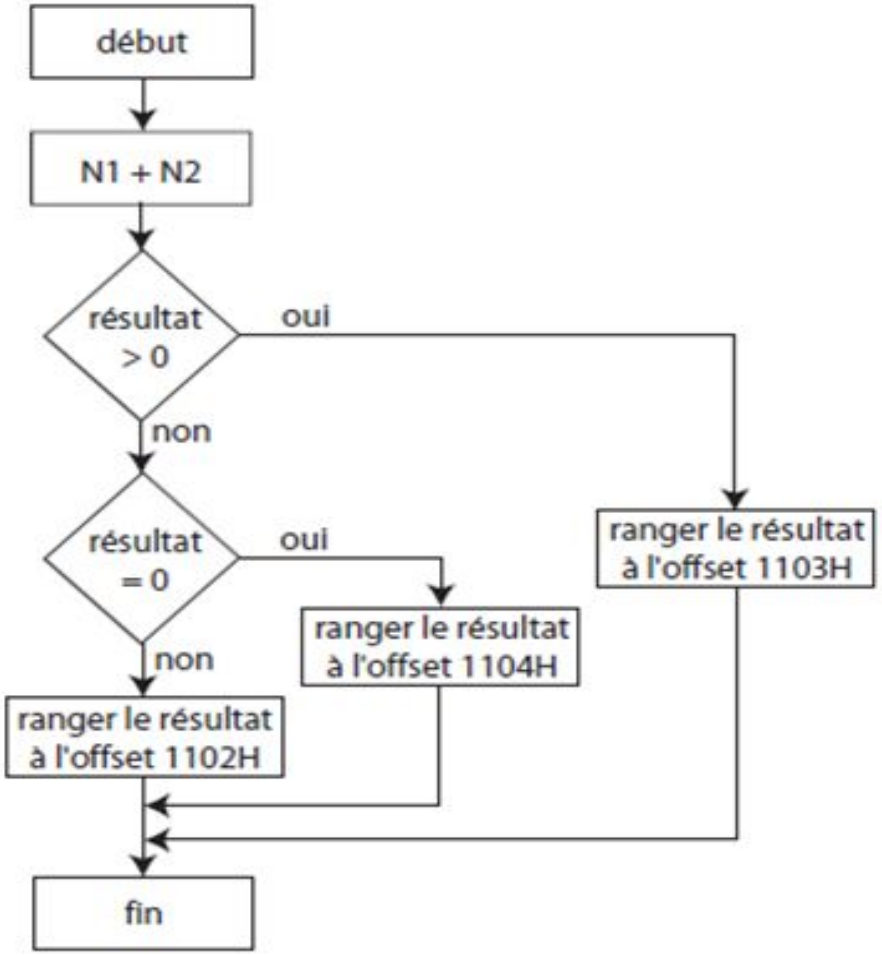
je *op*	branchement à l'adresse *op* si *op1* = *op2*
jne *op*	branchement à l'adresse *op* si *op1* ≠ *op2*
jl *op* (**jnge**)	branchement à l'adresse *op* si *op1* < *op2*
jle *op* (**jng**)	branchement à l'adresse *op* si *op1* ≤ *op2*
jg *op* (**jnle**)	branchement à l'adresse *op* si *op1* > *op2*
jge *op* (**jnl**)	branchement à l'adresse *op* si *op1* ≥ *op2*

Non signées :

je *op*	branchement à l'adresse *op* si *op1* = *op2*
jne *op*	branchement à l'adresse *op* si *op1* ≠ *op2*
jb *op* (**jnae**)	branchement à l'adresse *op* si *op1* < *op2*
jbe *op* (**jna**)	branchement à l'adresse *op* si *op1* ≤ *op2*
ja *op* (**jnbe**)	branchement à l'adresse *op* si *op1* > *op2*
jae *op* (**jnb**)	branchement à l'adresse *op* si *op1* ≥ *op2*

Exemple d'application des instructions de sauts conditionnels : on veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul.

Organigramme :



Exemple d'application des instructions de sauts conditionnels : on veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul.

```
mov [1100h],10110101b
mov [1101h],01111010b
mov al,[1100h]
add al,[1101h]
js negatif
jz null
mov [1102h],al
jmp fin

null:mov [1104h],al
jmp fin

negatif: mov [1103h],al
fin: ret
```

Les instructions de comparaison et de branchement

cmp *op1,op2*	calcul de *op1* – *op2* et de ZF,CF et OF
jmp *op*	branchement inconditionnel à l'adresse *op*
jz *op*	branchement à l'adresse *op* si ZF=1
jnz *op*	branchement à l'adresse *op* si ZF=0
jo *op*	branchement à l'adresse *op* si OF=1
jno *op*	branchement à l'adresse *op* si OF=0
js *op*	branchement à l'adresse *op* si SF=1
jns *op*	branchement à l'adresse *op* si SF=0
jc *op*	branchement à l'adresse *op* si CF=1
jnc *op*	branchement à l'adresse *op* si CF=0
jp *op*	branchement à l'adresse *op* si PF=1
jnp *op*	branchement à l'adresse *op* si PF=0

Ecrire un programme, en langage assembleur 8086, qui permet de compter les nombres nuls dans un tableau d'octets mémoire de longueur 100h et débutant à l'adresse [200h], le résultat sera placé à l'adresse [400h].

```
jmp debut
  TAB1 db 1 dup(05h,33h,00h,45h,0xEA,00h)
debut:
mov cx,6
 mov si,0
 mov bx,0
      ET1:mov al,TAB1[SI]
       cmp al,00h
       jz null
       inc si
       loop ET1  ; dex cx cmp cx avec 0 si dif de
zero elle fait saut vers et1
       jmp fin
            null: inc bx
          inc si
          loop ET1
         mov [400h],bx
                fin:  ret
```

Ecrire un programme qui permet de déterminer le maximum dans un tableau d'octets mémoire de longueur 100h et débutant à l'adresse [200h], le résultat sera placé à l'adresse [400h].

```
  jmp debut
tab db 10h, 5h, 3h ,2h, 1h, 0h, 0XFh, 5h, 12h,10h
maxi db 0
adrmaxi db ?
debut:
mov cx, 10
mov si, 0
boucle: mov al, tab[si]
cmp al, maxi
jb etiqu
mov al, tab[si]
mov adrmaxi, si
mov maxi, al
etiqu:   inc si

loop boucle
ret
```

Interruptions de saisie et d'affichage

Pour réaliser les opérations standards (affichage, saisie), on utilise les fonctions pré-écrites suivantes:

Saisie d'un caratère: mov AH, 1 ; fonction no. 1
(avec écho) int 21h ; résultat est mis dans AL

Saisie d'un caractère mov AH, 7 ; fonction no. 7
(sans écho) int 21h ; résultat dans AL

Affichage d'un caratère: mov DL, "A"; caractère A est transfére dans DL
mov AH, 2; fonction no. 2
int 21h ; appel au DOS

Affichage d'une chaine de caractères:

```
mov  DX, offset  chaine; ou bien Lea DX, offset chaine
mov AH, 09; fonction no. 9
int 21h;
```

Arrêt de programme: hlt ou bien int 21h ou bien hlt;

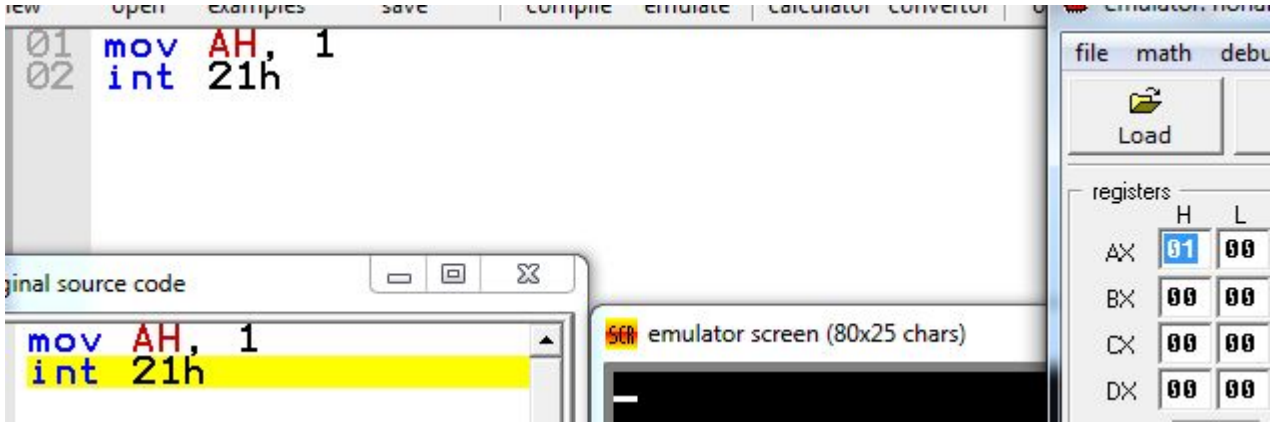
À mettre à la fin de chaque fin programme; c'est l'équivalent du return (0) en C. Ces instructions ont pour effet de retourner au DOS

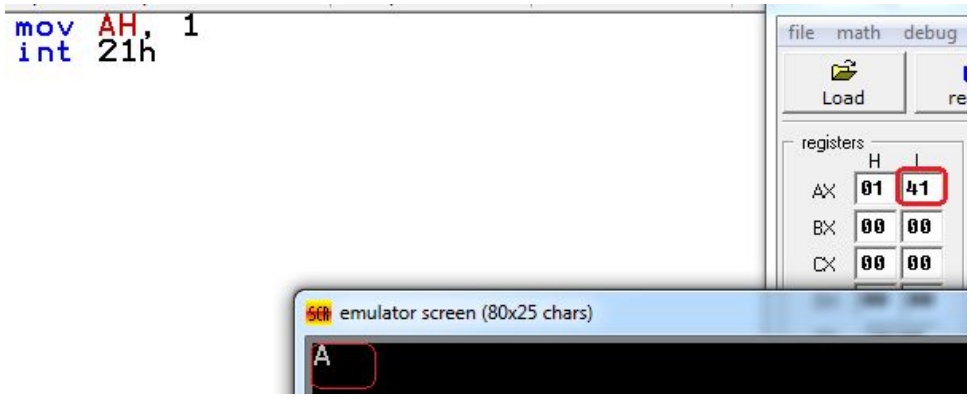
Saisie d’un caratère: **Saisie du caractére avec affichage**

DU code ASCI I qui sera mis dans AL

mov AH, 1;

int 21h ;





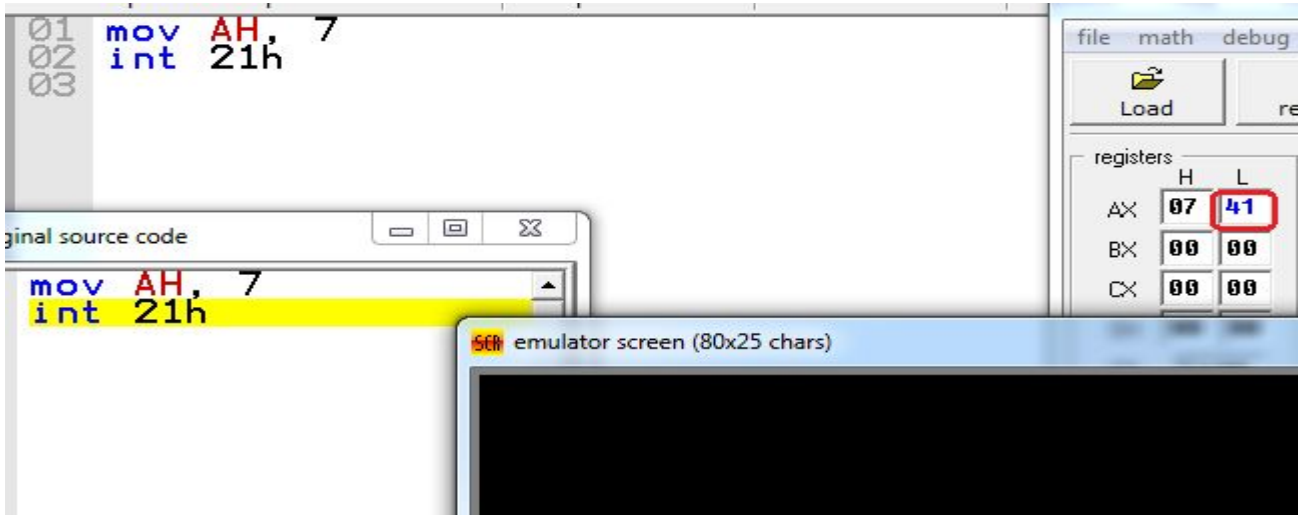
Saisie d'un caractère **Saisie du caractére sans affichage**

code ASCI I du caractére est mis dans AL

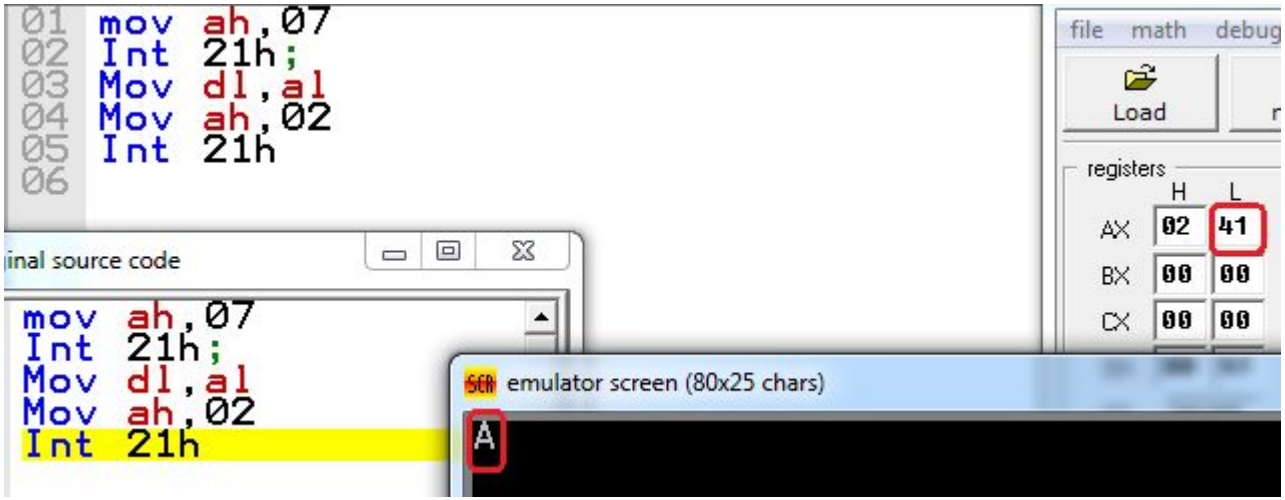
mov AH, 7

int 21h

On va saisir la lettre A



Pour saisir et afficher A



Pour réaliser les opérations standards (affichage, saisie), le système d'exploitation (ici DOS) fournit les fonctions pré-écrites suivantes:

Affichage d'un caratère:

mov DL, "A"; ***caractère A est transfére dans DL***

mov AH, 2; ***fonction no. 2***

int 21h ;

```
mov DL, "A"
 mov AH, 2
int 21h
```

exemple

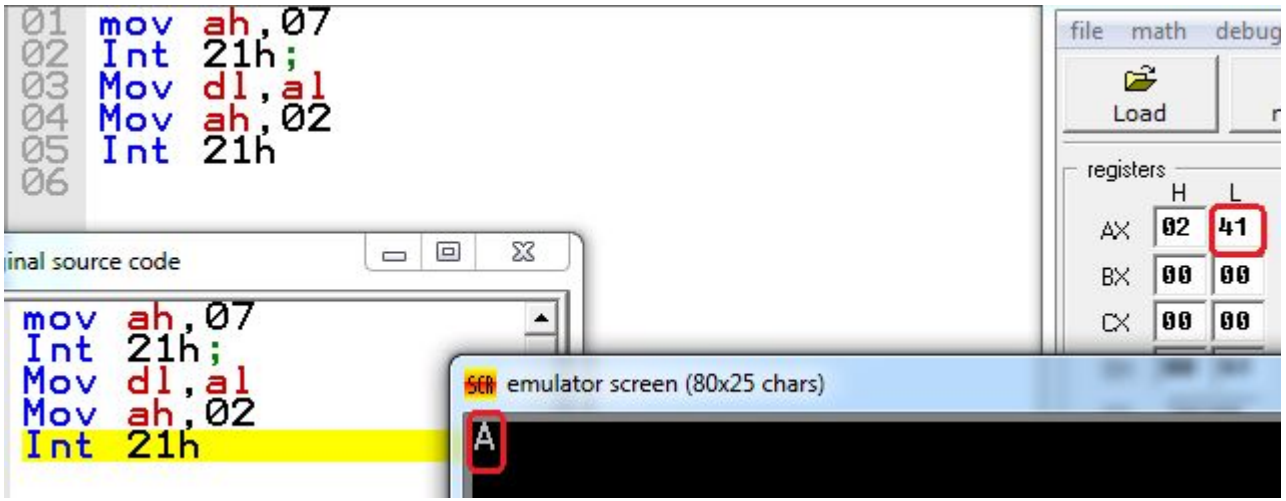
mov ah,07

Int 21h;

Mov dl,al

Mov ah,02

Int 21h



Arrêt de programme: **mov AX, 4C00h;**

int 21h;

À mettre à la fin de chaque fin programme; c'est l'équivalent du return (0) en C. Ces instructions ont pour effet de retourner au DOS

Affichage matrice

```
 MOV Dl,"5"   ;
mov ah,2
int 21h

mov dl,1Fh
mov ah,2
int 21h

mov dl,"6"
mov ah,2
int 21h

mov dl,1Fh
mov ah,2
int 21h

MOV Dl,"7"   ;
mov ah,2
int 21h

MOV Dl,0Ah   ;
mov ah,2
int 21h
MOV Dl,0Ah   ;
mov ah,2
int 21h
MOV Dl,0Dh   ;
mov ah,2
int 21h
```

Exemples

Affichage d'une chaine de caractères:

mov DX, offset chaine; ***adresse de caractère de la chaîne de caractére***

mov AH, 09h; ***fonction no. 9***

int 21h;

```
mov  DX, offset  message
mov ah,09h
int 21h
message db " Bonjour IRRM $"
ret
```

```
mov ah, 09h
mov dx, offset message
int 21h
    ret
message db "Bonjour LFI1 $ "
end
```

Exemple1 : afficher en binaire la suite :10110110b

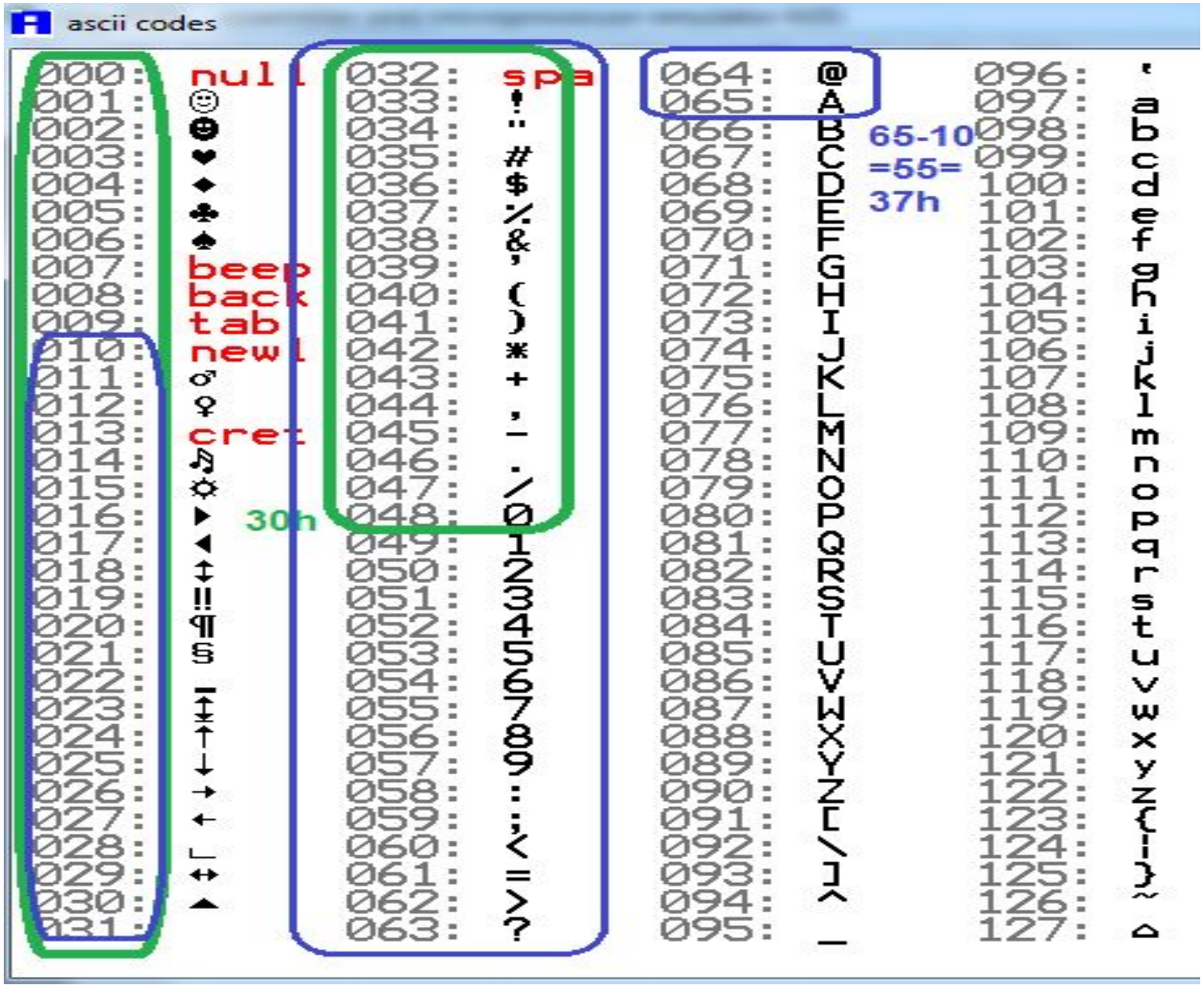
```
mov bl,10110110b
mov cx,8
boucle:shl bl,1
 jc affich1
 mov dl,'0'
 mov ah,2
int 21h
loop boucle
jmp fin
affich1: mov dl,'1'
mov ah,2
int 21h
 loop boucle
fin:ret
```

Exemple2: afficher les lettres alphabets Majuscule

```
mov dl,"A“
 mov cx,26
encore: mov ah,2
 int 21h
 inc dl
 loop encore
 ret
```

Exemple3: afficher les lettres alphabets Miniscule

```
mov dl,"a"
 mov cx,26
encore: mov ah,2
 int 21h
 inc dl
 loop encore
 ret
```


ascii codes
000: null
001: ☺
002: ☻
003: ♥
004: ♦
005: ♣
006: ♠
007: beep
008: back
009: tab
010: newl
011: ♂
012: ♀
013: cret
014: ♫
015: ☼
016: ►
017: ◄
018: ↕
019: ‼
020: ¶
021: §
022: ▬
023: ↨
024: ↑
025: ↓
026: →
027: ←
028: ∟
029: ↔
030: ▲
031:
032: spa
033: !
034: "
035: #
036: $
037: %
038: &
039: '
040: (
041:)
042: *
043: +
044: ,
045: -
046: .
047: /
048: 0
049: 1
050: 2
051: 3
052: 4
053: 5
054: 6
055: 7
056: 8
057: 9
058: :
059: ;
060: <
061: =
062: >
063: ?
30h
064: @
065: A
066: B
067: C
068: D
069: E
070: F
071: G
072: H
073: I
074: J
075: K
076: L
077: M
078: N
079: O
080: P
081: Q
082: R
083: S
084: T
085: U
086: V
087: W
088: X
089: Y
090: Z
091: [
092: \
093:]
094: ^
095: _
65-10
=55=
37h
096: `
097: a
098: b
099: c
100: d
101: e
102: f
103: g
104: h
105: i
106: j
107: k
108: l
109: m
110: n
111: o
112: p
113: q
114: r
115: s
116: t
117: u
118: v
119: w
120: x
121: y
122: z
123: {
124: |
125: }
126: ~
127: ⌂

Somme de deux nombre en hexa et affichage en binaire

```
 jmp start
 table dw 2 dup(0xFD33h,0xE355h)
 somme dw ?
   start: xor si,si
              mov cx,2

     ET1: mov bx, table[si]; si pointe
add somme,bx
add si,2
        loop ET1
 mov cx,16

boucle:shl somme,1
 jc affich1
 mov dl,'0'
 mov ah,2
int 21h
jmp retour
affich1: mov dl,'1'
mov ah,2
int 21h
retour: loop boucle
ret
```

```
jmp start
TAB dw 2 dup(0xFD33h,0xE355h)
somme dw ?
start:
mov cx,2
mov si,0

ET1:mov bx,TAB[SI]
add somme,bx   ;1110000010001000
add si ,2
loop et1
mov cx,16
boucle:shl somme,1
jc affich1
mov dl,'0'
mov ah,2
int 21h
loop boucle
jmp fin
affich1: mov dl,'1'
mov ah,2
int 21h
loop boucle
fin:ret
```

Affichage en hexadécimal

```
jmp debut
var db 0
debut: mov ax,0xA8h;
mov var,ax
mov bl,var
and bl,0xF0h
mov cl,4
shr bl,cl
cmp bl,9h
jg affich_quartetfort
add bl,30h
mov dl,bl
mov ah,2
int 21h
jmp ET1
affich_quartetfort:add bl,37h
mov dl,bl
mov ah,2
int 21h
ET1: mov bl,var
and bl,0x0Fh
cmp bl,9h
jg affich_quartetfaible
add bl,30h
mov dl,bl
mov ah,2
int 21h
jmp fin
affich_quartetfaible:add bl,37h
 mov dl,bl
 mov ah,2
int 21h
 fin:ret
```


jmp debut
var db 0
debut: mov ax,0xA8h;
mov var,al
mov bl,var
and bl,0xF0h
mov cl,4
shr bl,cl
cmp bl,9h
jg affich_quartetfort
add bl,30h
mov dl,bl
mov ah,2
int 21h
jmp ET1
affich_quartetfort:add bl,37h
mov dl,bl
mov ah,2
int 21h
ET1: mov bl,var
and bl,0x0Fh
cmp bl,9h
jg affich_quartetfaible
add bl,30h
mov dl,bl
mov ah,2
int 21h
jmp fin
affich_quartetfaible:add bl,37h
 mov dl,bl
 mov ah,2
int 21h
 fin:ret

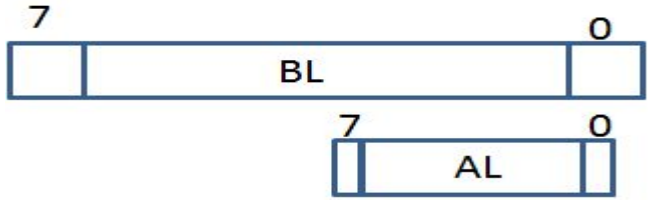
Affichage décimale

```
 jmp debut
    var db ?
debut: mov al,48;
mov var,al
mov dl,10
div dl
   mov var,ax
add al,30h
mov dl,al
mov ah,2
    int 21h
       mov ax,var
  add ah,30h
  mov dl,ah

mov ah,2

int 21h
```

```
jmp debut
   var db ?
debut: mov al,48;
mov var,al
mov dl,10
div dl
  mov var,ax
add al,30h
mov dl,al
mov ah,2
   int 21h
      mov ax,var
 add ah,30h
 mov dl,ah

mov ah,2
```

```
jmp debut
   var dw ?
debut: mov ax,5462;
mov var,ax
mov bx,1000
div bx
mov var,dx
mov dl,al
add dl,30h
mov ah,2
int 21h
mov dx,0
mov ax,var
mov bx,100
div bx
mov var,dx
mov dl,al
add dl,30h
mov ah,2
int 21h
mov dx,0
mov ax,var
mov bx,10
div bx
mov var,dx
mov dl,al
add dl,30h
mov ah,2
int 21h
mov dl,byte ptr var
add dl,30h
mov ah,2
int 21h
```

Avec affichage



Si le bit 7 de AL =1 on fait une

Série de 4 décalage à gauche de BL de bit

Si non

On fait un seul décalage de 1 bit à gauche de BX

```
MOV Bl,11011011b

mov al,01110101b

AND AL,10000000b

JZ ET1
Mov Cl,4
SHL Bl,Cl
mov cx,8

boucl:  shl  bl, 1
jc aff1
mov ah,2
mov dl,'0'
int 21h
jmp dec
aff1:       mov ah,2
mov dl,'1'
int 21h
```

```
dec:    loop boucl

JMP ET2
ET1: SHL Bl,1
mov cx,8

boucle:  shl  bl, 1
jc aff2
mov ah,2
mov dl,'0'
int 21h
jmp decr

aff2: mov ah,2
mov dl,'1'
int 21h
decr:loop boucle


ET2: Mov AX,4c00h
int 21h
```

```
bl=10101010
;si le bit 7=1 on faitune serie de 4 decalage de al vers gauche
;sinon un seul decalage vers la droire de bl

      mov bl,00101010b
      mov al,bl
      and bl,10000000b ;X0000000
      JZ dec_shr
            mov bl,al
        mov cl,4
        shl bl,cl;bl=A0
        mov cx,8
boucle:shl bl,1
 jc affich1
 mov dl,'0'
 mov ah,2
int 21h
loop boucle
jmp fin
affich1: mov dl,'1'
mov ah,2
int 21h
 loop boucle

        jmp fin

      dec_shr: mov bl,al
              shr bl,1
             mov cx,8
boucle1:shl bl,1
 jc affich2
 mov dl,'0'
 mov ah,2
int 21h
loop boucle
jmp fin
affich2: mov dl,'1'
mov ah,2
int 21h
 loop boucle1
                fin: ret
```

• **Appel de sous-programmes :** pour éviter la répétition d'une même séquence d'instructions plusieurs fois dans un programme, on rédige la séquence une seule fois en lui attribuant un nom et on l'appelle lorsqu'on en a besoin. Le programme appelant est *le programme principal*. La séquence appelée est *un sous-programme ou procédure*.

Ecriture d'un sous-programme :

```
nom_sp    PROC
            :   ← instructions du sous-programme
          ret   ← instruction de retour au programme principal
nom_sp    ENDP
```

Remarque : une procédure peut être de type NEAR si elle se trouve dans le même segment ou de type FAR si elle se trouve dans un autre segment.

Exemple :

```
ss_prog1    PROC    NEAR
ss_prog2    PROC    FAR
```

Appel d'un sous-programme par le programme principal : ***CALL procédure***

```
          :   ← instructions précédant l'appel au sous-programme
call nom_sp   ← appel au sous-programme
          :   ← instructions exécutées après le retour au programme principal
```

CALL destination :

Appel de procédure dans le même segment **(NEAR)** ou dans un autre segment **(FAR).**

Pour un **CALL FAR** (Inter segment)

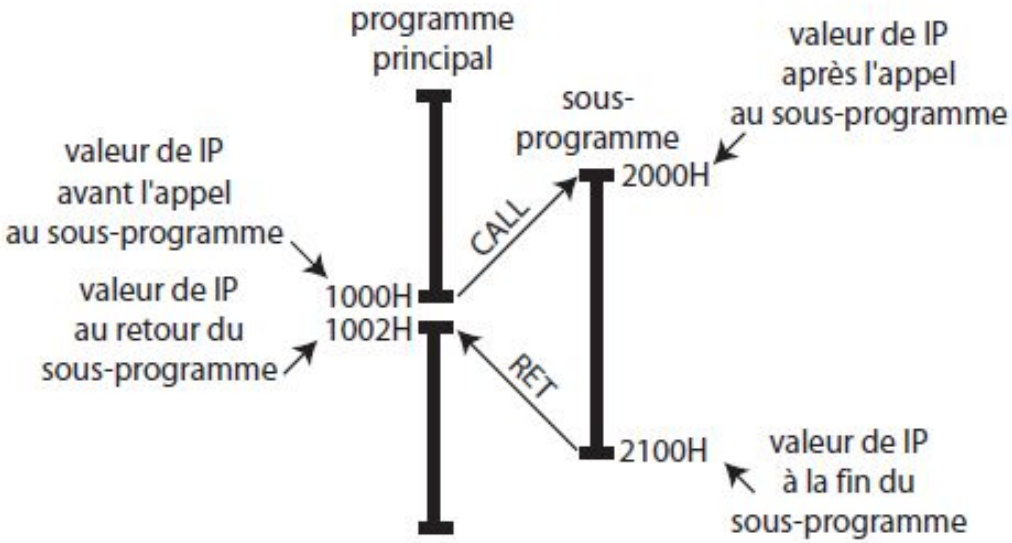
Sauvegarde de **CS** dans la pile **CS**←nouvelle valeur.

Sauvegarde de **IP** dans la pile **IP**←nouvelle valeur.

RET

Retour de procédure (NEAR : RET, FAR :RETF). Restauration à partir de la pile des anciennes valeurs de **CS** et **IP.**

Lors de l'exécution de l'instruction CALL, le pointeur d'instruction IP est chargé avec l'adresse de la première instruction du sous-programme. Lors du retour au programme appelant, l'instruction suivant le CALL doit être exécutée, càd que IP doit être rechargé avec l'adresse de cette instruction.



Avant de charger IP avec l'adresse du sous-programme, l'adresse de retour au programme principal, càd le contenu de IP, est sauvegardée dans une zone mémoire particulière appelée **pile.**
Lors de l'exécution de l'instruction RET, cette adresse est récupérée à partir de la pile et rechargée dans IP, ainsi le programme appelant peut se poursuivre.

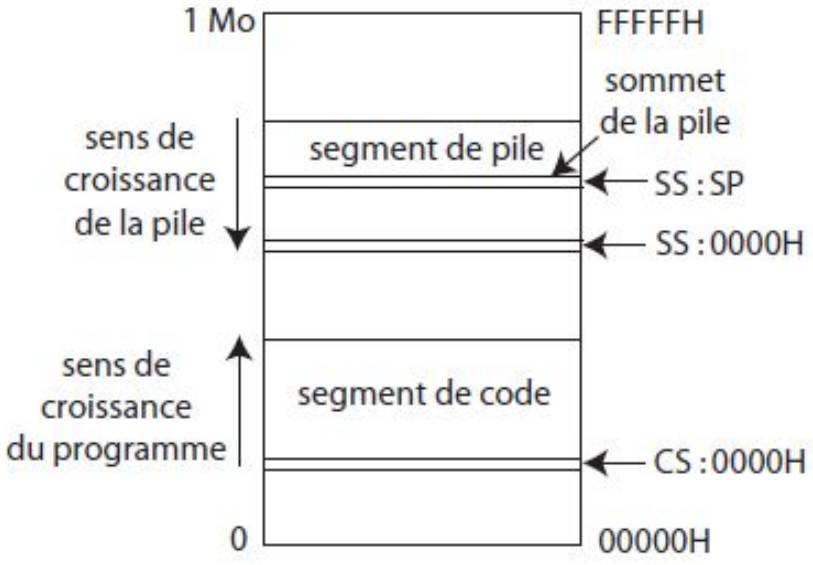
• **Fonctionnement de la pile :** la pile est une zone mémoire fonctionnant en mode LIFO (Last In First Out : dernier entré, premier sorti). Deux opérations sont possibles sur la pile :

- empiler une donnée : placer la donnée au sommet de la pile ;
- dépiler une donnée : lire la donnée se trouvant au sommet de la pile.

Le sommet de la pile est repéré par un registre appelé ***pointeur de pile (SP : Stack Pointer)*** qui contient l'adresse de la dernière donnée empilée.

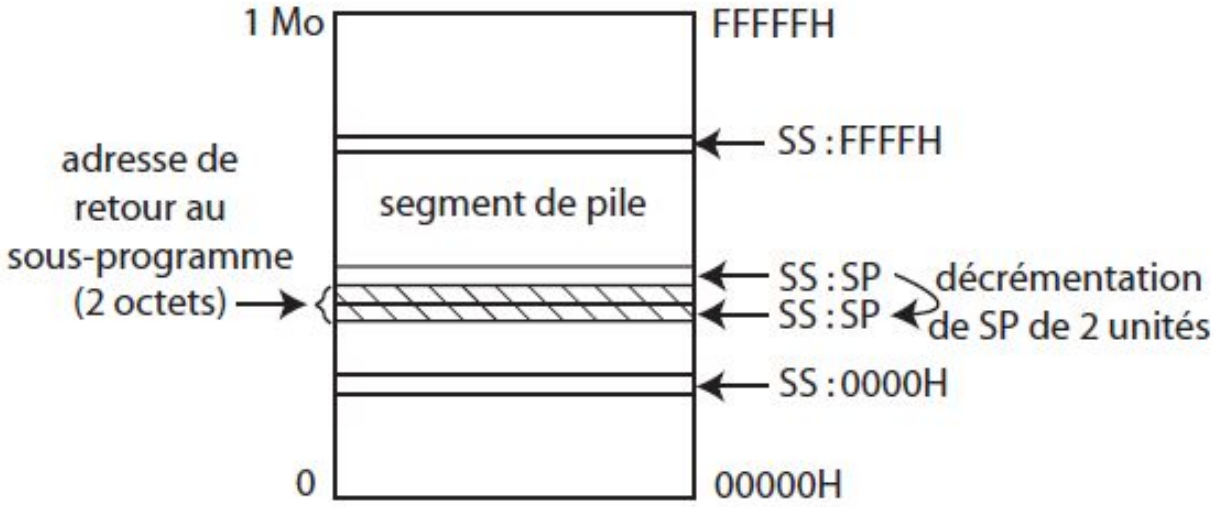
La pile est définie dans le segment de pile dont l'adresse de départ est contenue dans le registre SS (Stack Segment, registre de segment de pile).

Les données à empiler ou à dépiler sont uniquement des mots. La provenance ou la destination est un registre ou un emplacement mémoire. Le SP est décrémentée de 2 à chaque empilement et incrémentée de 2 à chaque dépilement.



Remarque : la pile et le programme croissent en sens inverse pour diminuer le risque de collision entre le code et la pile dans le cas où celle-ci est placée dans le même segment que le code (SS = CS).

Lors de l'appel à un sous-programme, l'adresse de retour au programme appelant (contenu de IP) est empilée et le pointeur de pile SP est automatiquement décrémenté. Au retour du sous-programme, le pointeur d'instruction IP est rechargé avec la valeur contenue sommet de la pile et SP est incrémenté.



2. Les instructions de transfert pour la pile

- **Instruction PUSH : PUSH Op**
  - Empiler l'opérande Op (Op doit être un opérande de 16 bits)
    - Décrémente SP de 2
    - Copie Op dans la mémoire pointée par SP

PUSH R16 ; R16=Registre sur 16 bits

PUSH word [adr]

~~PUSH Val_Immédiate~~ ~~PUSH R8~~

- **Instruction POP : 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 R16

POP word M ~~POP R8~~

PUSH source : (la pile)←(source).

POP destination : (destination)←(la pile)

source	destination	Exemples
Registre de travail 16 bits Mémoire (mot) Registre de segment	**Registre de travail 16 bits Mémoire (mot) Registre de segment (sauf CS)**	PUSH AX; POP AX Push DS ; pop DS Push CS ; Push ES ; pop ES Push SS ; pop SS

Instruction PUSH : PUSH Op

- Empiler l'opérande Op (Op doit être un opérande de 16 bits)
  - Décrémente SP de 2
  - Copie Op dans la mémoire pointée par SP

PUSH R16 ; R16=Registre sur 16 bits

PUSH word [adr]

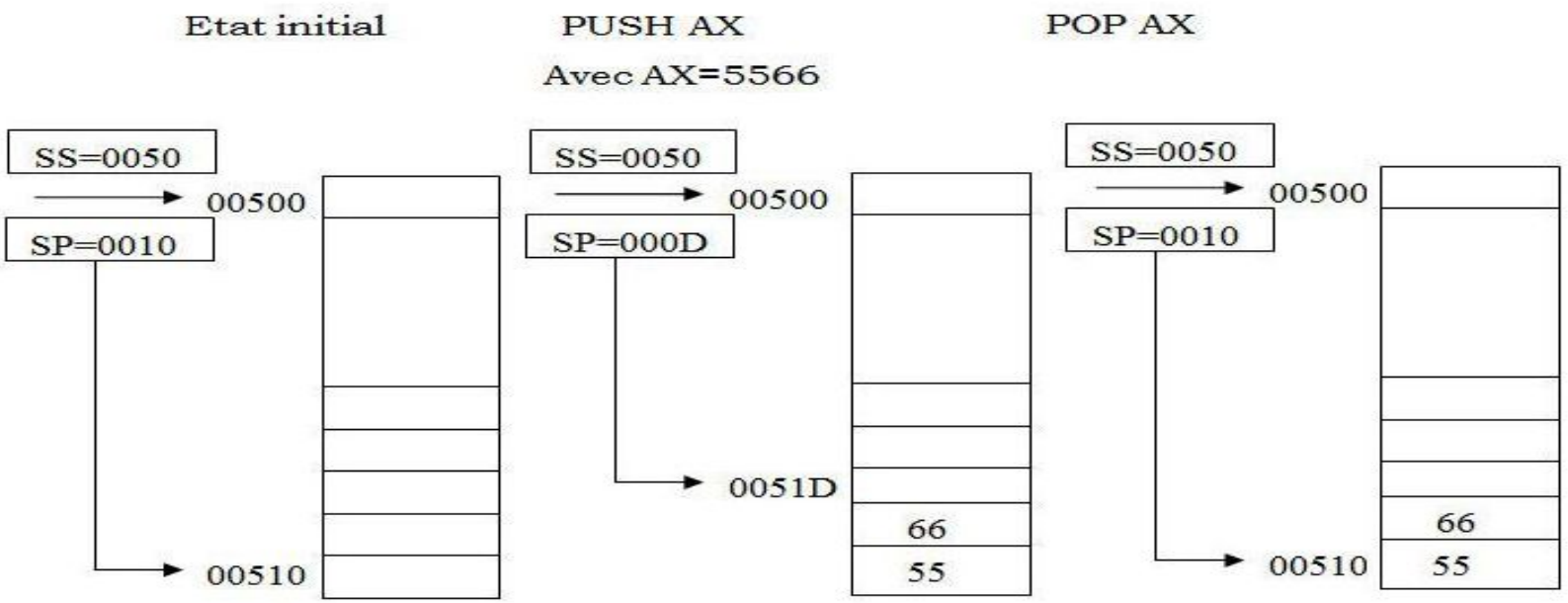
~~PUSH Val_Immédiate~~ ~~PUSH R8~~

Instruction POP : 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 R16

POP word M ~~POP R8~~



Exercice 1

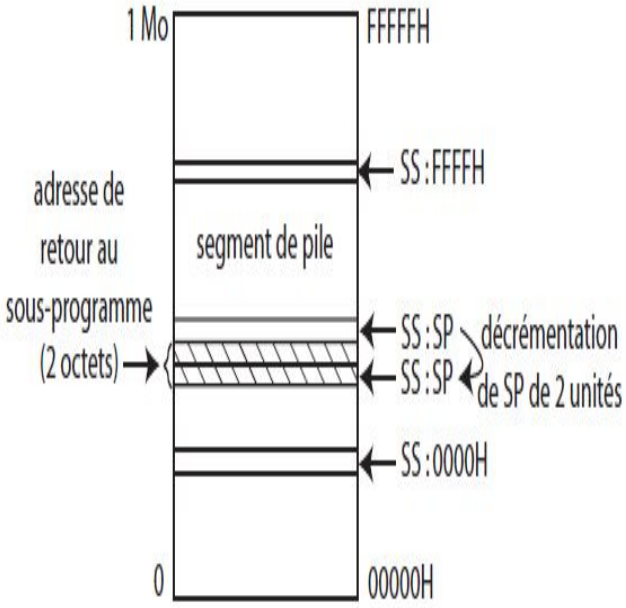
Déterminer le contenu des registres AX, BX, CX et DX dans chaque ligne de la séquence d'instructions ci-dessous sachant que ces registres contiennent initialement les valeurs suivantes :

AX=9897H BX=5678H CX=1BCEH DX=4532H

La pile contient une seule valeur = 1F8BH

```
mov AX,9897H
mov BX,5678H
mov CX,1BCEH
mov DX,4532H
OR BX,0xFFFFh
PUSH BX
XOR AX,AX
AND DX,0000H
MOV AX,000EH
MUL BL
POP CX
PUSH AX
POP DX
OR BX,0100H
```

BX=FFFFH
SS:SP=FFFFH
AX=0
DX=0
AX=000Eh
AX=0DF2h
CX=FFFFh
SS=0DF2h
DX=0DF2h
BX=FFFFh



MAX d'un tableau avec affichage décimale

```
jmp startTAB dw 1 dup (1325,1824,9760,2865)
var dw ?
VARMAX dw ?
start:
call findmax
call affich
findmax proc near
mov ax,TAB[0]
mov varmax,ax
mov si,1
mov cx,4
ET1:mov bx,TAB[si]
mov ax,varmax
cmp bx,ax
jz go
mov varmax,bx
go:
inc si
loop ET1
ret
findmax endp
affich proc near
   mov ax,VARMAX
   mov var,ax
   mov bx,1000
   div bx
   mov var,dx
   add ax,30h
   mov dl,al
   mov ah,2
   int 21h
   xor ax,ax
   xor dx,dx
 mov ax,var
   mov bx,100
   div bx
   mov var,dx
   add ax,30h
   mov dl,al
   mov ah,2
   int 21h
   xor ax,ax
   xor dx,dx
   mov ax,var
   mov bx,10
   div bx
   mov var,dx
   add ax,30h
   mov dl,al
   mov ah,2
   int 21h
   xor ax,ax
   xor dx,dx
   mov ax,var
   add ax,30h
   mov dl,al
   mov ah,2
   int 21h
   ret
affich endp
```