Chapitre 5

La programmation en assembleur du microprocesseur 8086

5.1 Généralités

Chaque microprocesseur reconnait un ensemble d'instructions appelé **jeu d'instructions** (Instruction Set) fixé par le constructeur. Pour les microprocesseurs classiques, le nombre d'instructions reconnues varie entre 75 et 150 (microprocesseurs **CISC** : Complex Instruction Set Computer). Il existe aussi des microprocesseurs dont le nombre d'instructions est très réduit (microprocesseurs **RISC** : Reduced Instruction Set Computer) : entre 10 et 30 instructions, permettant d'améliorer le temps d'exécution des programmes.
Une instruction est définie par son code opératoire, valeur numérique binaire difficile à manipuler par l'être humain. On utilise donc une **notation symbolique** pour représenter les instructions : les **mnémoniques**. Un programme constitué de mnémoniques est appelé **programme en assembleur**.
Les instructions peuvent être classées en groupes :

- instructions de transfert de données ;
- instructions arithmétiques ;
- instructions logiques ;
- instructions de branchement ...

5.2 Les instructions de transfert

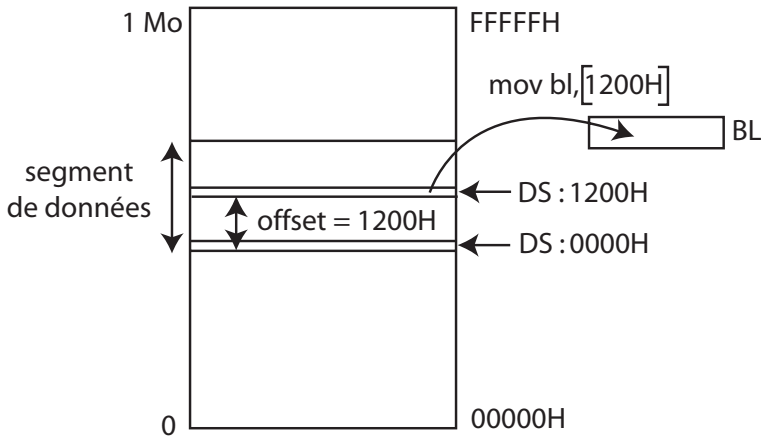
Elles permettent de déplacer des données d'une **source** vers une **destination** :

- registre vers mémoire ;
- registre vers registre ;
- mémoire vers registre.

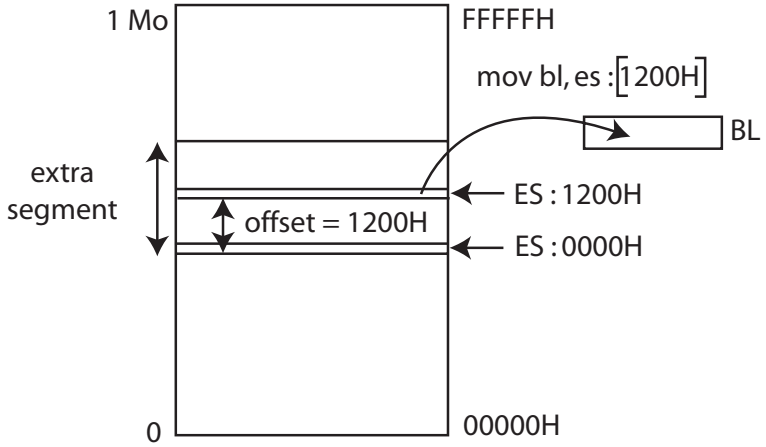
Remarque : le microprocesseur 8086 n'autorise pas les transferts de mémoire vers mémoire (pour ce faire, il faut passer par un registre intermédiaire).

Syntaxe : `MOV destination,source`
Remarque : MOV est l'abbréviation du verbe « to move » : déplacer.
Il existe différentes façons de spécifier l'adresse d'une case mémoire dans une instruction : ce sont les **modes d'adressage**.
Exemples de modes d'adressage simples :

- `mov ax,bx` : charge le contenu du registre BX dans le registre AX. Dans ce cas, le transfert se fait de registre à registre : **adressage par registre** ;
- `mov al,12H` : charge le registre AL avec la valeur 12H. La donnée est fournie immédiatement avec l'instruction : **adressage immédiat**.
- `mov bl,[1200H]` : transfère le contenu de la case mémoire d'adresse effective (offset) 1200H vers le registre BL. L'instruction comporte l'adresse de la case mémoire où se trouve la donnée : **adressage direct**. L'adresse effective représente l'offset de la case mémoire dans le segment de données (segment dont l'adresse est contenue dans le registre DS) : segment par défaut.

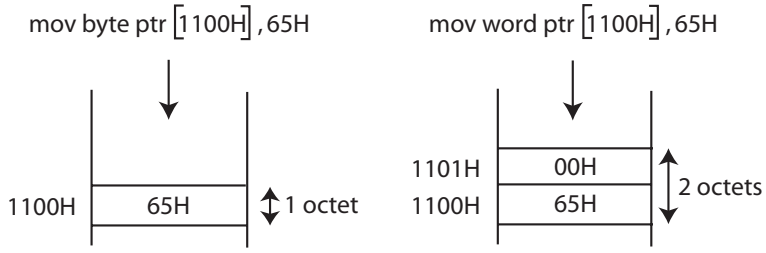


On peut changer le segment lors d'un adressage direct en ajoutant un **préfixe de segment**, exemple : `mov bl,es :[1200H]`. On parle alors de **forçage de segment**.



Remarque : dans le cas de l'adressage immédiat de la mémoire, il faut indiquer le **format** de la donnée : octet ou mot (2 octets) car le microprocesseur 8086 peut manipuler des données sur 8 bits ou 16 bits. Pour cela, on doit utiliser un **spécificateur de format** :

- `mov byte ptr [1100H],65H` : transfère la valeur 65H (sur 1 octet) dans la case mémoire d'offset 1100H ;
- `mov word ptr [1100H],65H` : transfère la valeur 0065H (sur 2 octets) dans les cases mémoire d'offset 1100H et 1101H.

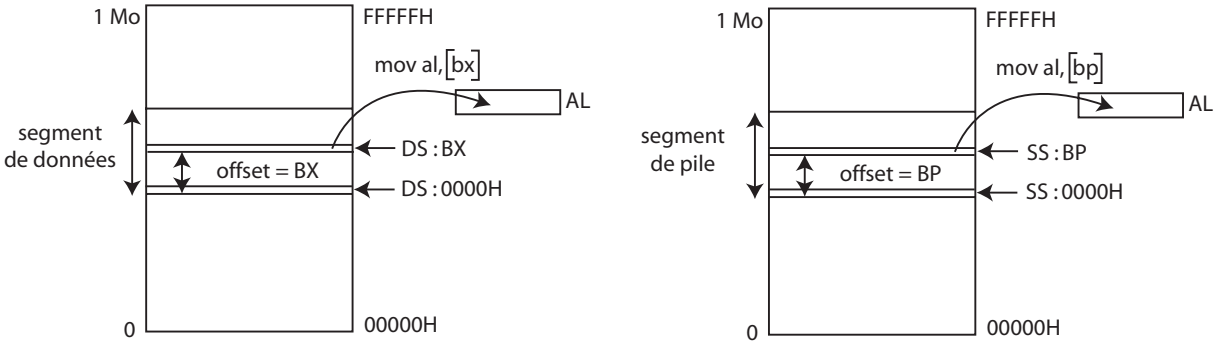


Remarque : les microprocesseurs Intel rangent l'octet de poids faible d'une donnée sur plusieurs octets à l'adresse la plus basse (format **Little Endian**).
Modes d'adressage évolués :

- **adressage basé** : l'offset est contenu dans un **registre de base** BX ou BP.
  Exemples :

  `mov al,[bx]` : transfère la donnée dont l'offset est contenu dans le registre de base BX vers le registre AL. Le segment associé par défaut au registre BX est le segment de données : on dit que l'adressage est **basé sur DS** ;

  `mov al,[bp]` : le segment par défaut associé au registre de base BP est le segment de pile. Dans ce cas, l'adressage est **basé sur SS**.



- **adressage indexé** : semblable à l'adressage basé, sauf que l'offset est contenu dans un registre d'index SI ou DI, associés par défaut au segment de données.
  Exemples :

  `mov al,[si]` : charge le registre AL avec le contenu de la case mémoire dont l'offset est contenu dans SI ;

`mov [di],bx` : charge les cases mémoire d'offset DI et DI + 1 avec le contenu du registre BX.

Remarque : une valeur constante peut éventuellement être ajoutée aux registres de base ou d'index pour obtenir l'offset. Exemple :

```
mov [si+100H],ax
```

qui peut aussi s'écrire

```
mov [si][100H],ax
```

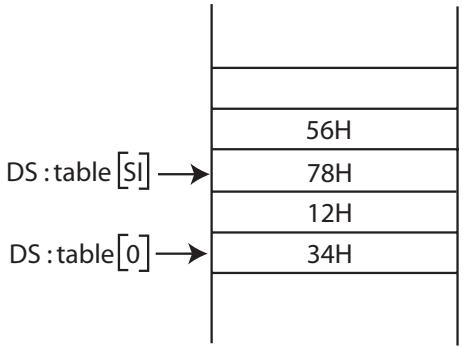
ou encore

```
mov 100H[si],ax
```

Les modes d'adressage basés ou indexés permettent la manipulation de tableaux rangés en mémoire. Exemple :

```
mov si,0
mov word ptr table[si],1234H
mov si,2
mov word ptr table[si],5678H
```

Dans cet exemple, `table` représente l'offset du premier élément du tableau et le registre SI joue le rôle d'indice de tableau :



- **adressage basé et indexé** : l'offset est obtenu en faisant la somme d'un registre de base, d'un registre d'index et d'une valeur constante. Exemple :

  ```
  mov ah,[bx+si+100H]
  ```

  Ce mode d'adressage permet l'adressage de structures de données complexes : matrices, enregistrements, ... Exemple :

  ```
  mov bx,10
  mov si,15
  mov byte ptr matrice[bx][si],12H
  ```

  Dans cet exemple, BX et SI jouent respectivement le rôle d'indices de ligne et de colonne dans la matrice.

5.3 Les instructions arithmétiques

Les instructions arithmétiques de base sont l'**addition**, la **soustraction**, la **multiplication** et la **division** qui incluent diverses variantes. Plusieurs modes d'adressage sont possibles.

Addition : `ADD opérande1,opérande2`
L'opération effectuée est : `opérande1` ← `opérande1` + `opérande2`.
Exemples :

- `add ah,[1100H]` : ajoute le contenu de la case mémoire d'offset 1100H à l'accumulateur AH (adressage direct) ;
- `add ah,[bx]` : ajoute le contenu de la case mémoire pointée par BX à l'accumulateur AH (adressage basé) ;
- `add byte ptr [1200H],05H` : ajoute la valeur 05H au contenu de la case mémoire d'offset 1200H (adressage immédiat).

Soustraction : `SUB opérande1,opérande2`
L'opération effectuée est : `opérande1` ← `opérande1` − `opérande2`.

Multiplication : `MUL opérande`, où `opérande` est un registre ou une case mémoire.
Cette instruction effectue la multiplication du contenu de AL par un opérande sur 1 octet ou du contenu de AX par un opérande sur 2 octets. Le résultat est placé dans AX si les données à multiplier sont sur 1 octet (résultat sur 16 bits), dans (DX,AX) si elles sont sur 2 octets (résultat sur 32 bits).
Exemples :

- ```
  mov al,51
  mov bl,32
  mul bl
  ```
  → $\mathrm{AX} = 51 \times 32$
- ```
  mov ax,4253
  mov bx,1689
  mul bx
  ```
  → $(\mathrm{DX}, \mathrm{AX}) = 4253 \times 1689$
- ```
  mov al,43
  mov byte ptr [1200H],28
  mul byte ptr [1200H]
  ```
  → $\mathrm{AX} = 43 \times 28$
- ```
  mov ax,1234
  mov word ptr [1200H],5678
  mul word ptr [1200H]
  ```
  → $(\mathrm{DX}, \mathrm{AX}) = 1234 \times 5678$

Division : `DIV opérande`, où `opérande` est un registre ou une case mémoire.
Cette instruction effectue la division du contenu de AX par un opérande sur 1 octet ou le contenu de (DX,AX) par un opérande sur 2 octets. Résultat : si l'opérande est sur 1 octet,

alors AL = quotient et AH = reste ; si l'opérande est sur 2 octets, alors AX = quotient et DX = reste.
Exemples :

- ```
  mov ax,35
  mov bl,10
  div bl
  ```
  → AL = 3 (quotient) et AH = 5 (reste)
- ```
  mov dx,0
  mov ax,1234
  mov bx,10
  div bx
  ```
  → AX = 123 (quotient) et DX = 4 (reste)

Autres instructions arithmétiques :

- ADC : addition avec retenue ;
- SBB : soustraction avec retenue ;
- INC : incrémentation d'une unité ;
- DEC : décrémentation d'une unité ;
- IMUL : multiplication signée ;
- IDIV : division signée.

5.4 Les instructions logiques

Ce sont des instructions qui permettent de manipuler des données au niveau des bits. Les opérations logiques de base sont :

- ET ;
- OU ;
- OU exclusif ;
- complément à 1 ;
- complément à 2 ;
- décalages et rotations.

Les différents modes d'adressage sont disponibles.

ET logique : `AND opérande1,opérande2`
L'opération effectuée est : `opérande1` ← `opérande1` ET `opérande2`.
Exemple :

```
mov al,10010110B
mov bl,11001101B      →
and al, bl
```

AL =	1	0	0	1	0	1	1	0
BL =	1	1	0	0	1	1	0	1
AL =	1	0	0	0	0	1	0	0

Application : **masquage** de bits pour mettre à zéro certains bits dans un mot.
Exemple : masquage des bits 0, 1, 6 et 7 dans un octet :

7	6	5	4	3	2	1	0	
0	**1**	0	1	0	1	**1**	**1**	
0	**0**	**1**	**1**	**1**	**1**	**0**	**0**	← **masque**
0	**0**	0	1	0	1	**0**	**0**	

OU logique : `OR opérande1,opérande2`
L'opération effectuée est : `opérande1` ← `opérande1` OU `opérande2`.
Application : mise à 1 d'un ou plusieurs bits dans un mot.
Exemple : dans le mot 10110001B on veut mettre à 1 les bits 1 et 3 sans modifier les autres bits.

7	6	5	4	3	2	1	0	
1	0	1	1	**0**	0	**0**	1	
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	← **masque**
1	0	1	1	**1**	0	**1**	1	

Les instructions correspondantes peuvent s'écrire :

```
mov ah,10110001B
or ah,00001010B
```

Complément à 1 : `NOT opérande`
L'opération effectuée est : `opérande` ← $\overline{\texttt{opérande}}$.
Exemple :

```
mov al,10010001B
not al
```
→ $\text{AL} = \overline{10010001}\text{B} = 01101110\text{B}$

Complément à 2 : `NEG opérande`
L'opération effectuée est : `opérande` ← $\overline{\texttt{opérande}} + 1$.
Exemple :

```
mov al,25
mov bl,12
neg bl
add al,bl
```
→ $\text{AL} = 25 + (-12) = 13$

OU exclusif : `XOR opérande1,opérande2`
L'opération effectuée est : `opérande1` ← `opérande1` $\oplus$ `opérande2`.
Exemple : mise à zéro d'un registre :

```
mov al,25
xor al,al
```
→ $\text{AL} = 0$

Instructions de décalages et de rotations : ces instructions déplacent d'un certain nombre de positions les bits d'un mot vers la gauche ou vers la droite.
Dans les décalages, les bits qui sont déplacés sont remplacés par des zéros. Il y a les décalages logiques (opérations non signées) et les décalages arithmétiques (opérations signées).

Dans les rotations, les bits déplacés dans un sens sont réinjectés de l'autre côté du mot.

Décalage logique vers la droite (Shift Right) : `SHR opérande,n`

Cette instruction décale l'opérande de n positions vers la droite.

Exemple :

```
mov al,11001011B
shr al,1
```

→ entrée d'un 0 à la place du bit de poids fort ; le bit sortant passe à travers l'indicateur de retenue CF.

Remarque : si le nombre de bits à décaler est supérieur à 1, ce nombre doit être placé dans le registre CL ou CX.

Exemple : décalage de AL de trois positions vers la droite :

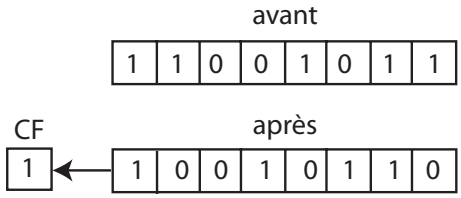
```
mov cl,3
shr al,cl
```

Décalage logique vers la gauche (Shift Left) : `SHL opérande,n`

Cette instruction décale l'opérande de n positions vers la droite.

Exemple :

```
mov al,11001011B
shl al,1
```



→ entrée d'un 0 à la place du bit de poids faible ; le bit sortant passe à travers l'indicateur de retenue CF.

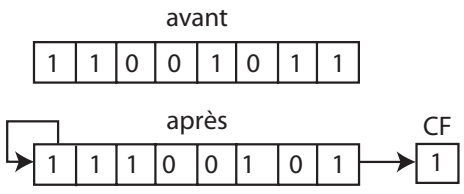
Même remarque que précédemment si le nombre de positions à décaler est supérieur à 1.

Décalage arithmétique vers la droite : `SAR opérande,n`

Ce décalage conserve le bit de signe bien que celui-ci soit décalé.

Exemple :

```
mov al,11001011B
sar al,1
```



→ le bit de signe est **réinjecté**.

Décalage arithmétique vers la gauche : `SAR opérande,n`
Identique au décalage logique vers la gauche.

Applications des instructions de décalage :

- cadrage à droite d'un groupe de bits.
  Exemple : on veut avoir la valeur du quartet de poids fort du registre AL :

```
mov al,11001011B
mov cl,4              →      AL = 0000 1100 B
shr al,cl
```

- test de l'état d'un bit dans un mot.
  Exemple : on veut déterminer l'état du bit 5 de AL :

```
mov cl,6                        mov cl,3
shr al,cl        ou             shl al,cl
```

  → avec un décalage de 6 positions vers la droite ou 4 positions vers la gauche, le bit 5 de AL est transféré dans l'indicateur de retenue CF. Il suffit donc de tester cet indicateur.

- multiplication ou division par une puissance de 2 : un décalage à droite revient à faire une division par 2 et un décalage à gauche, une multiplication par 2.
  Exemple :

```
mov al,48
mov cl,3        →      AL = 48/2^3 = 6
shr al,cl
```

Rotation à droite (Rotate Right) : `ROR opérande,n`
Cette instruction décale l'opérande de n positions vers la droite et réinjecte par la gauche les bits sortant.

Exemple :

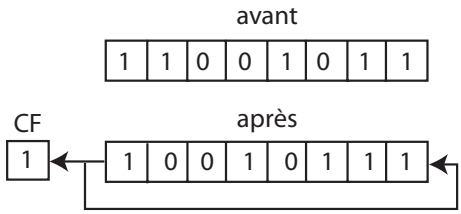
```
mov al,11001011B
ror al,1
```

→ réinjection du bit sortant qui est copié dans l'indicateur de retenue CF.

Rotation à gauche (Rotate Left) : `ROL opérande,n`
Cette instruction décale l'opérande de n positions vers la gauche et réinjecte par la droite les bits sortant.

Exemple :

```
mov al,11001011B
rol al,1
```



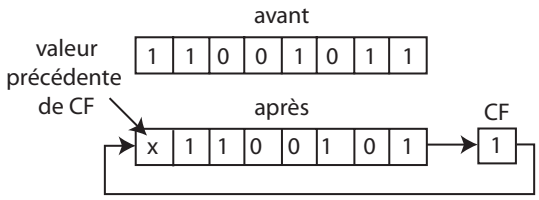
→ réinjection du bit sortant qui est copié dans l'indicateur de retenue CF.

Rotation à droite avec passage par l'indicateur de retenue (Rotate Right through Carry) : `RCR opérande,n`

Cette instruction décale l'opérande de n positions vers la droite en passant par l'indicateur de retenue CF.

Exemple :

```
mov al,11001011B
rcr al,1
```



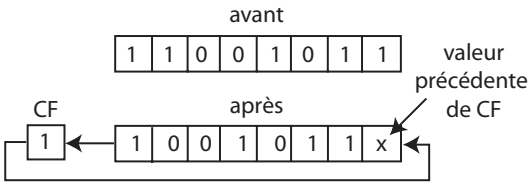
→ le bit sortant par la droite est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la gauche.

Rotation à gauche avec passage par l'indicateur de retenue (Rotate Left through Carry) : `RCL opérande,n`

Cette instruction décale l'opérande de n positions vers la gauche en passant par l'indicateur de retenue CF.

Exemple :

```
mov al,11001011B
rcl al,1
```



→ le bit sortant par la gauche est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la droite.

5.5 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 :

```
                    ⋮   ← instructions précédant le saut
              jmp suite
                    ⋮   ← instructions suivant le saut (jamais exécutées)
suite :       ...       ← instruction exécutée après le saut
```

Exemple :

```
boucle : inc ax
         dec bx          →      boucle infinie
         jmp boucle
```

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 (ou plusieurs) des indicateurs d'état (flags) du microprocesseur :

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`

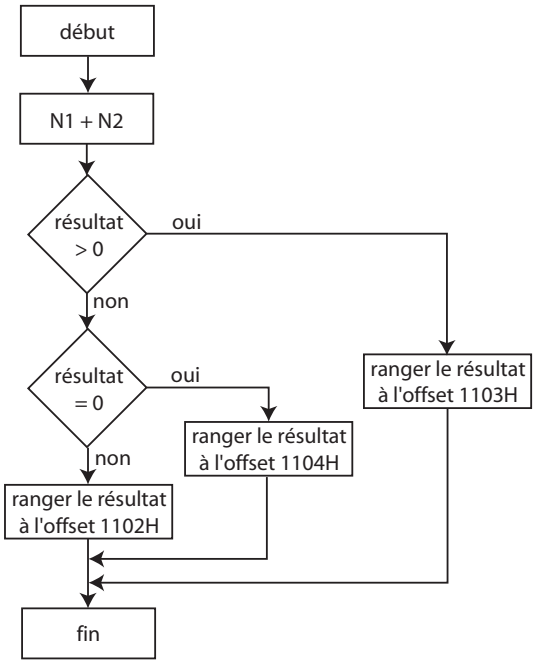
condition	nombres signés	nombres non signés
$=$	`JEQ label`	`JEQ label`
$>$	`JG label`	`JA label`
$<$	`JL label`	`JB label`
$\neq$	`JNE label`	`JNE label`

Exemple :

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

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 :



Programme :

```
                mov     al,[1100H]
                add     al,[1101H]
                js      negatif
                jz      nul
                mov     [1102H],al
                jmp     fin
negatif :       mov     [1103H],al
                jmp     fin
nul :           mov     [1104H],al
fin :           hlt
```

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 (au choix) et on l'appelle lorsqu'on en a besoin. Le programme ap-

pelant 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
```

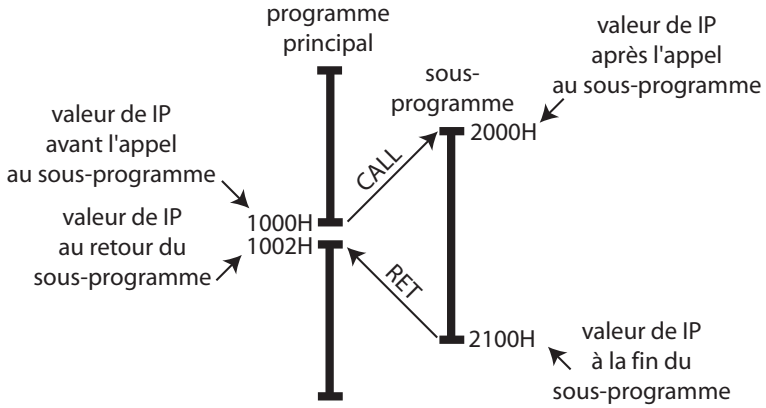
Remarque : une procédure peut être de type `NEAR` si elle se trouve dans le même segment ou de type `FAR` si elle se trouve dans un autre segment.
Exemple :

```
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
```

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, c'est-à-dire que IP doit être rechargé avec l'adresse de cette instruction.

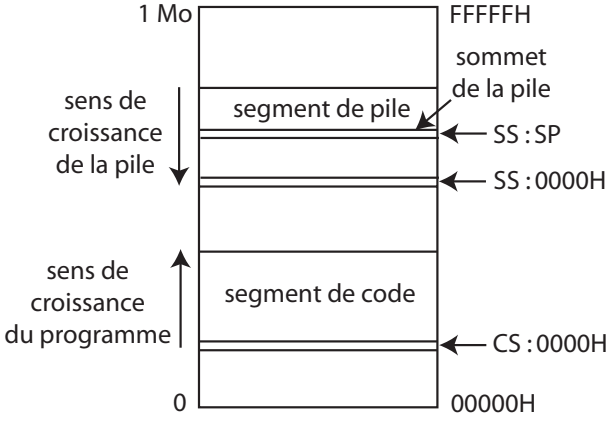


Avant de charger IP avec l'adresse du sous-programme, l'adresse de retour au programme principal, c'est-à-dire le contenu de IP, est sauvegardée dans une zone mémoire particulière appelée **pile**. Lors de l'exécution de l'instruction RET, cette adresse est récupérée à partir de la pile et rechargée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

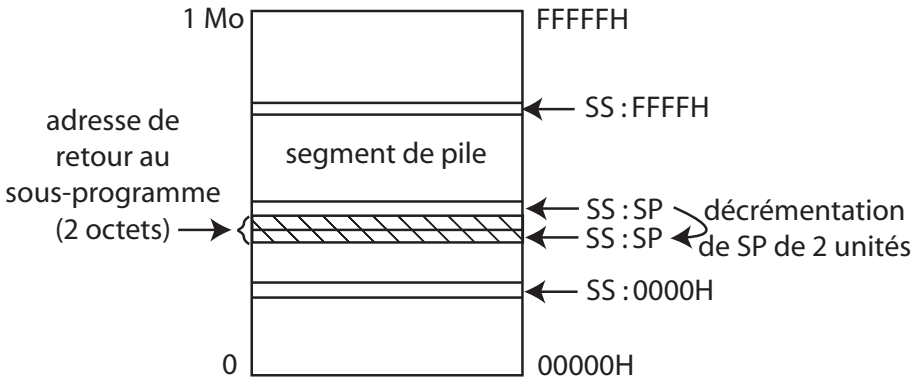
Le sommet de la pile est repéré par un registre appelé **pointeur de pile** (SP : Stack Pointer) qui contient l'adresse de la dernière donnée empilée.

La pile est définie dans le **segment de pile** dont l'adresse de départ est contenue dans le registre SS.



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**.



La pile peut également également servir à sauvegarder le contenu de registres qui ne sont pas automatiquement sauvegardés lors de l'appel à un sous programme :

- instruction d'empilage : `PUSH opérande`
- instruction de dépilage : `POP opérande`

où `opérande` est un registre ou une donnée sur 2 octets (on ne peut empiler que des mots de 16 bits).

Exemple :

```
push ax        ; empilage du registre AX ...
push bx        ; ... du registre BX ...
push [1100H]   ; ... et de la case mémoire 1100H-1101H
     ⋮
pop  [1100H]   ; dépilage dans l'ordre inverse de l'empilage
pop  bx
pop  ax
```

Remarque : la valeur de SP doit être initialisée par le programme principal avant de pouvoir utiliser la pile.

Utilisation de la pile pour le passage de paramètres : pour transmettre des paramètres à une procédure, on peut les placer sur la pile avant l'appel de la procédure, puis celle-ci les récupère en effectuant un adressage basé de la pile en utilisant le registre BP.

Exemple : soit une procédure effectuant la somme de deux nombres et retournant le résultat dans le registre AX :

- programme principal :

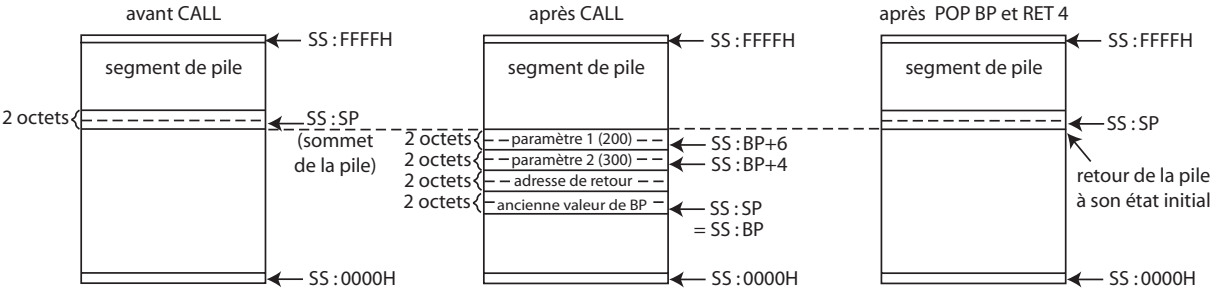
```
        mov  ax,200
        push ax        ; empilage du premier paramètre
        mov  ax,300
        push ax        ; empilage du deuxième paramètre
        call somme     ; appel de la procédure somme
```

- procédure `somme` :

```
somme   proc
        push bp        ; sauvegarde de BP
        mov  bp,sp     ; faire pointer BP sur le sommet de la pile
        mov  ax,[bp+4] ; récupération du deuxième paramètre
        add  ax,[bp+6] ; addition au premier paramètre
        pop  bp        ; restauration de l'ancienne valeur de BP
        ret  4         ; retour et dépilage des paramètres
somme   endp
```

L'instruction `ret 4` permet de retourner au programme principal et d'incrémenter le pointeur de pile de 4 unités pour dépiler les paramètres afin de remettre la pile dans son état initial.

Etat de la pile :



5.6 Méthodes de programmation

Etapes de la réalisation d'un programme :

- Définir le problème à résoudre : que faut-il faire exactement ?
- Déterminer des algorithmes, des organigrammes : comment faire ? Par quoi commencer, puis poursuivre ?
- Rédiger le programme (**code source**) :
  - utilisation du jeu d'instructions (mnémoniques) ;
  - création de documents explicatifs (documentation).
- Tester le programme en réel ;
- Corriger les erreurs (**bugs**) éventuelles : **déboguer** le programme puis refaire des tests jusqu'à obtention d'un programme fonctionnant de manière satisfaisante.

Langage machine et assembleur :

- Langage machine : codes binaires correspondant aux instructions ;
- Assembleur : logiciel de traduction du code source écrit en langage assembleur (mnémoniques).

Réalisation pratique d'un programme :

- Rédaction du code source en assembleur à l'aide d'un éditeur (logiciel de traitement de texte ASCII) :
  - `edit` sous MS-DOS,
  - `notepad` (bloc-note) sous Windows,
- Assemblage du code source (traduction des instructions en codes binaires) avec un assembleur :
  - `MASM` de Microsoft,
  - `TASM` de Borland,
  - `A86` disponible en shareware sur Internet, ...

  pour obtenir le **code objet** : code machine exécutable par le microprocesseur ;
- Chargement en mémoire centrale et exécution : rôle du système d'exploitation (ordinateur) ou d'un moniteur (carte de développement à base de microprocesseur).

Pour la mise au point (débogage) du programme, on peut utiliser un programme d'aide à la mise au point (comme `DEBUG` sous MS-DOS) permettant :

- l'exécution pas à pas ;
- la visualisation du contenu des registres et de la mémoire ;
- la pose de points d'arrêt ...

Structure d'un fichier source en assembleur : pour faciliter la lisibilité du code source en assembleur, on le rédige sous la forme suivante :

```
labels          instructions          commentaires
label1 :        mov ax,60H            ; ceci est un commentaire ...
    ⋮               ⋮                     ⋮
sous_prog1      proc near             ; sous-programme
    ⋮               ⋮                     ⋮
sous_prog1      endp
    ⋮               ⋮                     ⋮
```

Directives pour l'assembleur :

- Origine du programme en mémoire : `ORG offset`
  Exemple : `org 1000H`
- Définitions de constantes : `nom_constante EQU valeur`
  Exemple : `escape equ 1BH`
- Réservation de cases mémoires :

  ```
  nom_variable DB valeur_initiale
  nom_variable DW valeur_initiale
  ```

  DB : **Define Byte**, réservation d'un octet ;
  DW : **Define Word**, réservation d'un mot (2 octets).
  Exemples :

  ```
  nombre1  db  25
  nombre2  dw  ?              ; pas de valeur initiale
  buffer   db  100 dup (?)    ; réservation d'une zone mémoire
                              ; de 100 octets
  ```