Université Mohammed V
Faculté des Sciences, Rabat
Département d'Informatique

ARCHITECTURE DES ORDINATEURS PROGRAMMATION EN ASSEMBLEUR

Licence Fondamentale : Sciences Mathématiques et Informatiques

Préparer par :

Pr. Soumia ZITI

Année universitaire 2015 / 2016

Table des matières

I. Processeur

1. Définition

Un processeur constitue le cœur de tout ordinateur : il exécute les instructions qui composent les programmes que nous lui demandons d'exécuter. Les instructions sont stockées en mémoire (en dehors du processeur). L'ensemble de ces instructions compose le langage d'assemblage, ou assembleur

Pour exécuter un programme, le processeur lit les instructions en mémoire, une par une. Il connaît à tout instant l'adresse (l'endroit dans la mémoire) à laquelle se trouve la prochaine instruction à exécuter car il mémorise cette adresse dans son compteur ordinal.

Les instructions agissent sur des données qui sont situées soit en mémoire, soit dans des registres du processeur. Un registre est un élément de mémorisation interne au processeur et contenant une valeur. Le nombre des registres limité (14 pour le 8086). Pour accéder une donnée en mémoire, il faut spécifier son adresse. Pour accéder une donnée dans un registre, il faut spécifier son nom (chaque registre possède un nom qui est une chaîne de caractères).

Le 8086 est un processeur 16 bits, c'est-à-dire qu'il traite des données codées sur 16 bits.

2. Terminologie

- un bit est une valeur binaire qui, par convention, peut prendre la valeur 0 ou 1 ;
- un octet est une donnée codée sur 8 bits ;
- un mot est une donnée codée sur 16 bits ;
- un Koctet est un ensemble de 1024 octets

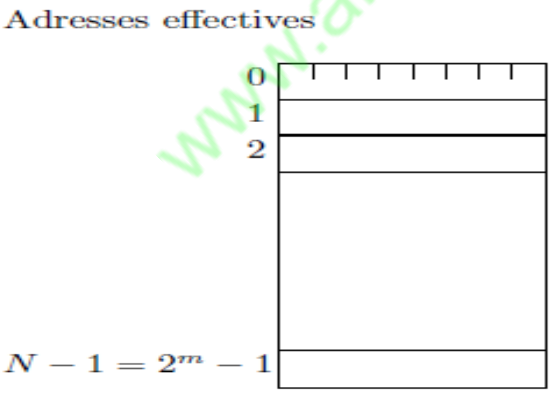
3. Mémoire

La mémoire est une séquence d'octets, tous numérotés de manière unique par un entier compris entre 0 et N −1. On dit alors que la capacité de la mémoire est de N octets. On a toujours N qui est une puissance de 2 ($N = 2^m$). On appelle adresse effective (AE) d'un octet en mémoire le numéro qui lui est associé

Pour le 8086 un octet est désigné par un couple (N° de segment, déplacement dans le segment) qui constituent une adresse segmentée (AS).

Un segment est un ensemble de 64 Koctets consécutifs. Le déplacement (ou offset) spécifie un octet particulier dans un segment. Segment et déplacement sont codés sur 16 bits et peuvent donc prendre une valeur comprise entre 0 et 65535. On a une relation entre adresses effectives et couple (segment, déplacement) sous la forme de l'équation :

$$\text{adresse effective} = 16 \times \text{segment} + \text{déplacement}$$



Une conséquence de la relation énoncée plus haut entre adresse effective et adresse segmentée est qu'un octet d'adresse effective donnée peut être accédé de plusieurs manières. Plus précisément, à chaque AE correspond $2^{12} = 4096$ AS différentes

4. Segmentation de la mémoire

Lors de son exécution, un programme utilise plusieurs segments mémoire :

- un segment contient les instructions du programme (le segment de code) ;
- un segment contient les données du programme (le segment de données) ;
- un segment contient la pile du programme (le segment de pile).

Ces trois segments peuvent se situer n'importe où en mémoire et même se recouvrir partiellement ou totalement. Leur attribution en mémoire est généralement réalisée automatiquement par le système d'exploitation sans que le programmeur ait à s'en soucier.

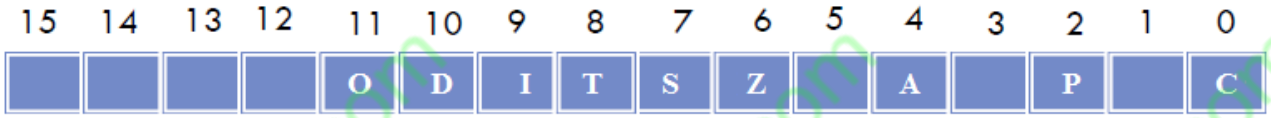
5. Registres du 8086

ax	ah	al
bx	bh	bl
cx	ch	cl
dx	dh	dl
si		
di		
sp	offset du sommet de pile	
bp		
ip	offset de la prochaine instruction à exécuter	
cs	numéro du segment d'instructions	
ds	numéro du segment de données	
es		
ss	numéro du segment de pile	
flags		

– *Les registres du 8086 et leur utilisation. Chaque registre contient 16 bits.*

Le 8086 possède 14 registres de 16 bits. Ce sont :

- **AX :** registre d'usage général contenant des données. Les 8 bits de poids faible se nomment al et les 8 bits de poids fort se nomment ah.
- **BX :** registre d'usage général contenant des données se décompose en BL et BH.
- **CX :** registre d'usage général contenant des données se décompose en CL et CH.
- **DX :** registre d'usage général contenant des données se décompose en DL et DH.
- si registre d'usage général contenant généralement le déplacement dans un segment d'une donnée.
- **DI :** registre d'usage général contenant généralement le déplacement dans un segment d'une donnée.
- **BP :** registre utilisé pour adresser des données dans la pile
- **SP :** registre pointeur de pile. Ce registre sera décrit plus loin
- **IP :** registre pointeur d'instruction (compteur ordinal). Ce registre indique la prochaine instruction à exécuter.
- **FLAGS :** registre d'indicateurs de l'état du processeur. Certains bits de ce registre portent des noms. Ce sont tous des indicateurs binaires :

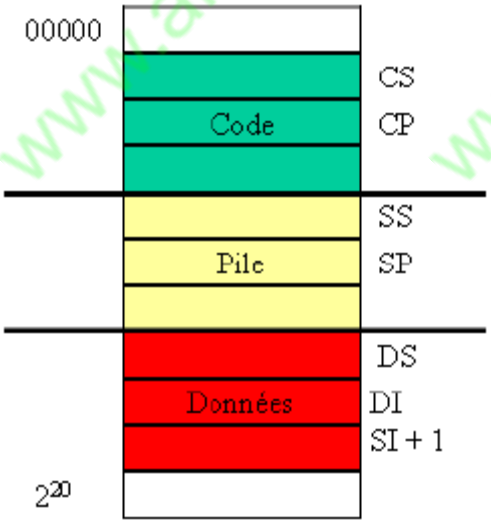


- **O le bit de débordement** (overflow) est positionné par la plupart des instructions arithmétiques pour indiquer s'il y a eut un débordement de capacité lors du calcul (un nombre trop grand ou trop petit)
- **D le bit de direction** est positionné à 1 si le transfert de données se fait en décrémentant les offsets, à 0 sinon (incrémentation des offsets).
- **S le bit de signe** est positionné par la plupart des instructions arithmétiques pour indiquer le signe du résultat (positif ou négatif).
- **Z le bit de zéro** est positionné par la plupart des instructions arithmétiques pour indiquer que le résultat du calcul est 0
- **C le bit de carry** (retenue) est positionné par la plupart des instructions arithmétiques pour indiquer si le calcul a engendré une retenue qui devra être reportée sur les calculs
- **A le bit dit retenue auxiliaire** (auxiliary carry) est positionné par la plupart des instructions arithmétiques pour indiquer une retenue entre bits de poids faible et bits de poids forts d'un octet, d'un mot ou d'un double mot.
- **P le bit de parité** est positionné par la plupart des instructions arithmétiques. Il indique si les 8 bits de poids faible du résultat comportent un nombre pair de 1.

Le processeur 8086 comporte également des registres 16 bits pour contenir des numéros de segment :

- **CS (code segment)** : segment contenant le programme en cours d'exécution
- **DS (data segment)** : segment contenant les données
- **ES (registre segment)** : auxiliaire pour adresser des données
- **SS (stack segment)** : segment contenant la pile

Les valeurs des registres CS, DS et SS sont automatiquement initialisées par le système d'exploitation au lancement du programme. Dès lors, ces segments sont implicites, c'est-à-dire que si l'on désire accéder à une donnée en mémoire, il suffit de spécifier son offset sans avoir à se soucier du segment.



II. Assembleur

1. Définition

Un programme en assembleur a une forme bien particulière. Chaque ligne d'un code source assembleur comporte une instruction. Chaque ligne est composée de champs. De gauche à droite, on a :

- **le champ étiquette**, qui peut être vide est un identificateur composé de lettres, chiffres et de caractères $, %, _ et ? (exemples boucle, fin_de_tant_que, ...)
- **le champ mnémonique** (le nom de l'instruction) comme **MOV, CMP, LOOP**, ... Elles peuvent indifféremment être écrites avec des lettres minuscules ou majuscules.
- **le champ opérande** (les arguments de l'instruction), qui peut être vide pour indiquer les données à traiter, soit sous la forme de valeurs constantes, soit en spécifiant l'adresse mémoire (l'emplacement en mémoire) où se trouve la donnée, soit le nom d'un registre contenant la donnée ou contenant l'adresse mémoire de la donnée.
- **le champ commentaire**, qui peut être vide qui doit commencer par un caractère ; et se termine en fin de ligne.

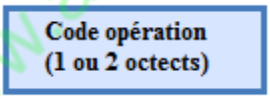
La plupart des instructions ont leur premier opérande qui indique une donnée et l'endroit où il faut ranger le résultat (donnée et résultat sont stockés au même endroit), le deuxième qui indique une donnée. Par exemple, dans l'instruction **mov ah, 4ch**, ah est la destination de l'instruction, 4ch la donnée source.

2. Mode d'adressage

C'est la manière d'interpréter les bits d'un champ d'adresse en vue de la localisation de l'opérande
Ils existent cinq types d'adressage : par registre (ou implicite), immédiat, direct, indirect, basé et immédiat.

a. L'adressage par registre ou implicite

Dans ce mode, il n'existe aucun accès mémoire pour les opérandes, l'instruction contient seulement le code opération, sur 1 ou 2 octets, et l'instruction porte sur des registres ou spécifie une opération sans opérande



➢ ***Exemple :***

```
INC AX
```

b. L'adressage immédiat

C'est l'adressage dans lequel la valeur de l'opérande figure directement dans l'instruction sans avoir besoin de faire un nouvel accès mémoire.

➢ ***Exemple :***

```
Add AX,valeur   ou MOV AX,10
```

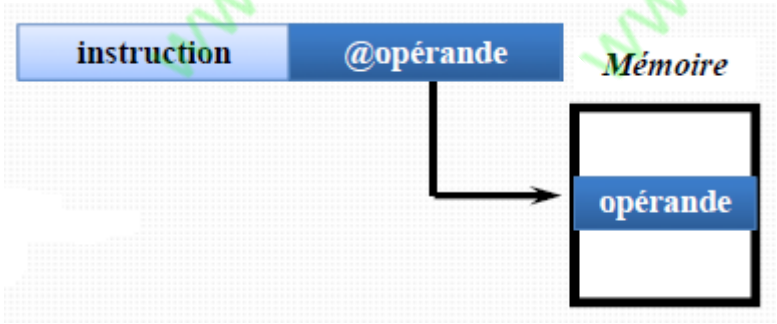
c. L'adressage direct

Dans cet adressage un des opérandes est l'emplacement mémoire dont l'adresse figure dans l'instruction (une fois l'instruction lue, il faut aller faire un accès mémoire pour obtenir l'opérande désiré)

➢ ***Exemple :***

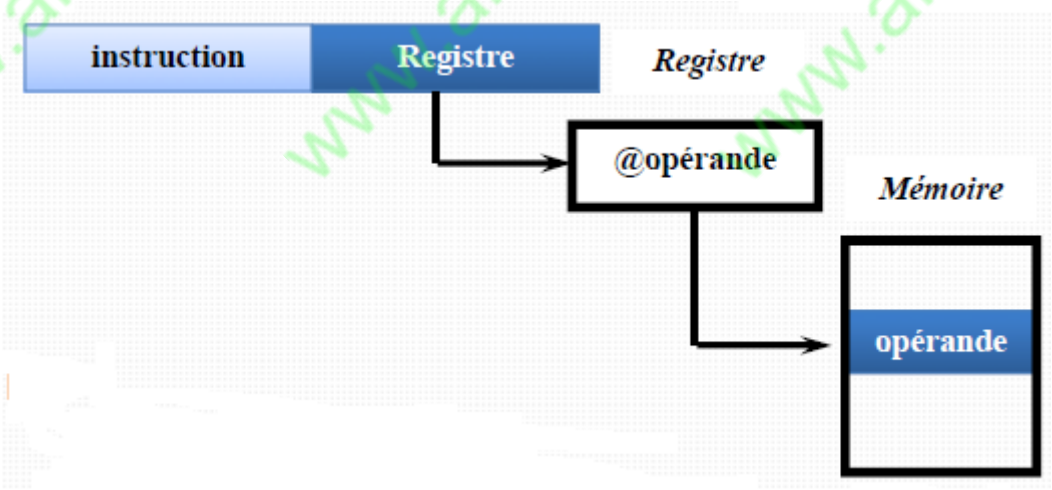
```
Add AX,[adresse]
```

l'instruction `MOV AX,[130h];` Met la valeur contenue dans l'emplacement d'adresse 130h dans AX



d. L'adressage indirect

L'adresse de la variable est contenue dans un registre de base ou d'index : 2 accès mémoires à condition que le type de la donnée soit accordé avec le registre utilisé

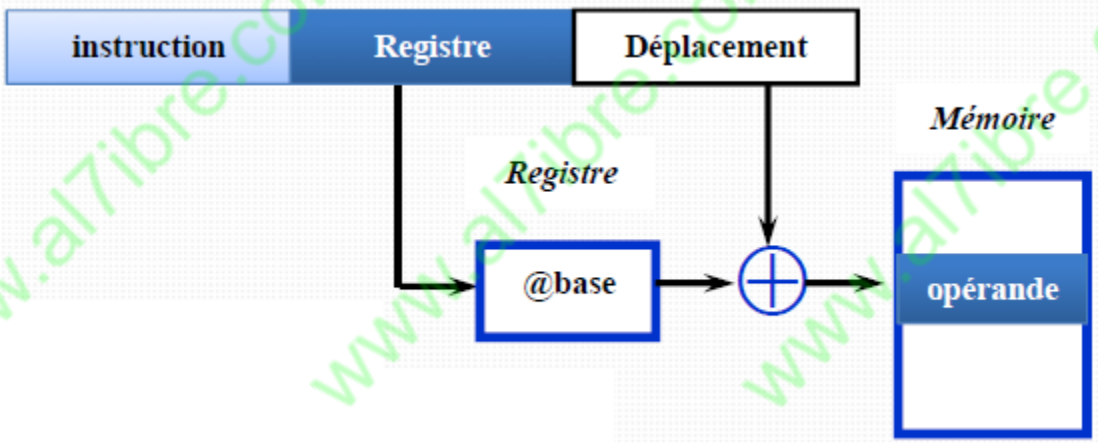


➢ ***Exemple :***

```
Mov BX,adresse   puis   Add AX,[BX]
```

e. L'adressage basé

Similaire à l'adressage indirect par registre sauf qu'un déplacement est ajouté à la base (adresse effective = adresse base + déplacement)

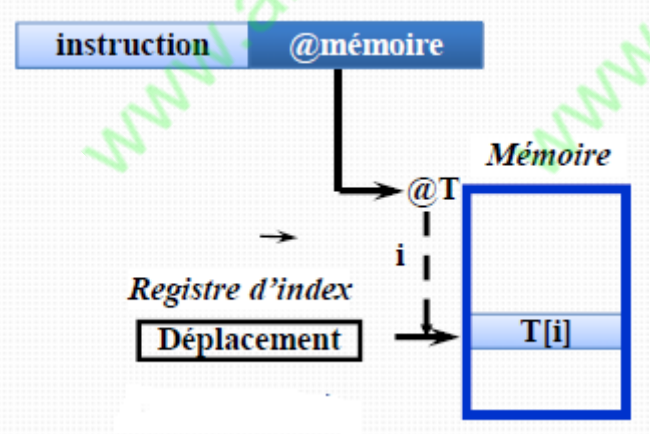


➢ ***Exemple :***

```
Mov BX,adresse   puis   Add AX,[BX+4]
```

f. L'adressage indexé (Add,[adresse+index])

On utilise un registre d'index SI ou DI plus un déplacement (L'adressage est noté par des crochets). Cet adressage est utile pour le parcours de tableaux



➢ ***Exemple :***

```
MOV SI,2        puis        MOV AX, T[SI]
```

3. Emplacement de données en mémoire

Il y a 16 manières de spécifier le déplacement dans un segment mémoire : étiquette, **[bx], [si], [di], BX +étiquette, BX +si+étiquette, BX +di+étiquette, [bx+si], [bx+di], [bp+si], [bp+di], bp+étiq, bp+si+étiq, bp+di+étiq, si+étiq, di+étiq.**
Avec **étiq** indique l'étiquette d'une donnée définie dans la section de données du programme où déplacement est une constante codée sur 16 bits.
Les adresses de début de segment de données et de code en utilisant les symboles @data et @code respectivement.

III. Langage assembleur

1. Les constantes

g. Les constantes numériques

On peut spécifier les constantes numériques dans les bases 2, 8, 10 et 16. Une constante numérique commence toujours par un chiffre. A priori, une constante est notée en base 10.
Pour l'exprimer **en base 2**, une constante ne doit comporter que des chiffres 0 et 1 et être terminée par la lettre b.
Pour l'exprimer **en base 8**, une constante ne doit comporter que des chiffres 0, 1, 2, 3, 4, 5, 6 ou 7 et se terminer par la lettre o.
Pour l'exprimer **en base 16**, une constante ne doit comporter que des chiffres de 0 à 9 ou des lettres de A à F pour les valeur hexadécimales correspondant aux nombres décimaux 10, 11, 12, 13, 14 et 15 et se terminer par la lettre h.

➢ ***Exemples :***

– 100111b est une constante binaire dont la valeur décimale est 39
– 36o est une constante octale dont la valeur décimale est 30
– 4Dh est une constante hexadécimale dont la valeur décimale est 77
Puisqu'une constante numérique doit toujours commencer par un chiffre, une constante hexadécimale dont le premier caractère est une lettre sera précédée d'un 0. Ainsi, la valeur c3 sera notée 0c3h dans un code source assembleur.

h. Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entre deux apostrophes ''. Si on veut mettre un caractère ' dans une constante, il suffit de la doubler. Exemples :
– 'hello world'
– 'l''arc'

i. Déclaration d'une constante

La directive EQU associe une valeur à un symbole qui pourra être ensuite utilisé dans le programme à la place de la constante qu'elle définit.

- ***Syntaxe : nom EQU constante***
- ***Exemple :***

```
ZERO EQU 0
```

Définit une constante qui s'appelle `ZERO`, dont la valeur est 0. Une fois cette déclaration effectuée, toute occurrence de l'identificateur `ZERO` sera remplacée par la valeur indiquée.

2. Les déclarations de variables

L'espace mémoire est utilisé pour stocker des constantes (chaînes de caractères, valeurs numériques, …) ou des variables. Avant d'employer une variable, il faut préalablement la déclarer. Déclarer une variable revient toujours à réserver un emplacement en mémoire pour y stocker la valeur de cette variable.

a. La directive DB

[nomVar] DB constante1 [, constante2] […]

Réserve et initialise un octet, `nomVar` est un symbole permettant d'accéder à cet octet

- ***Exemple :***

Les 4 lignes suivantes :

```
OCTET1 DB 33
DEUX_OCTETS DB 34, 62
LETTRE DB 'a'
CHAINE DB 'hello world !', 13, 10, '$'
TABLEAU1 DB 10 dup (5)
TABLEAU2 DB 20 dup ( ?)
```

Définissent 6 symboles comme suit :
– **`OCTET1`** référence une donnée codée sur un octet dont la valeur sera initialisée à 33
– **`DEUX_OCTETS`** référence une donnée codée sur un octet dont la valeur sera initialisée à 34, l'octet suivant étant initialisé avec 62. On peut considérer que `DEUX_OCTETS` référence un tableau de 2 octets ;
– **`LETTRE`** référence une donnée dont la valeur sera initialisée avec le code ASCII du caractère `'a'` ;
– **`CHAINE`** référence une chaîne de caractères composée des caractères `'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', ' ', '!'`, des caractères de code 13 (retour en début de ligne) et 10 (aller à la ligne suivante) et enfin du caractère `'$'`.
– **`TABLEAU1`** référence un tableau de 10 octets dont la valeur initiale des éléments est 5

– **TABLEAU2** référence un tableau de 20 octets non initialisés

La valeur initiale d'un symbole peut être exprimée de différentes manières : une valeur, plusieurs valeurs séparées par des virgules, le code d'un caractère, une chaîne de caractères, chacun des caractères initialisant un octet...

b. La directive DW

[nomVar] DW constante1 [, constante2] [...]

Réserve et initialise un mot (16 bits), `nomVar` est un symbole permettant d'accéder à ce mot.

➢ *Exemples*

```
MOT1 DW 1546
MOT2 DW ((45 * 23) / 9 + 24)
TMOT DW 20 DUP (0)
```

– le symbole **MOT1** référence une donnée codée sur un mot et qui sera initialisée avec la valeur 1546 ;

– le symbole **MOT2** référence une donnée codée sur un mot et qui est initialisée avec la valeur 139 ;

– le symbole **TMOT** référence un tableau de 20 mots initialisés avec la valeur 0

Dans cet exemple une valeur constante peut être spécifiée avec une expression numérique (à condition qu'elle soit calculable lors de l'assemblage). Les octets composant un mot sont stockés de la manière suivante : le poids faible est mis dans le premier octet, le poids fort dans le deuxième octet. Ainsi, si on déclare :

`M DW 1234h` ; la valeur 34h est mise à l'adresse M et la valeur 12h à l'adresse M+1.

3. Les instructions arithmétiques et logiques

a. Les instructions arithmétiques

Comme dans de nombreux processeurs, le processeur 8086 possède des instructions +, −, × et ÷ qui traitent des données entières codées sur un octet ou un mot. Pour faire des opérations sur des données plus complexes (des nombres flottants par exemple), on devra les programmer.

➢ *Addition sans retenue*

```
ADD registre/variable, registre
ADD registre, registre/variable
ADD registre/variable, constante
```

L'instruction ADD effectue l'addition du contenu du registre source au registre destination, sans report de retenue : destination ← source + destination. La retenue est positionnée en fonction du résultat de l'opération. Par exemple si nous considérons les deux instructions suivantes :

```
mov AX , a9h
add AX , 72h
```

Alors le registre AX contient la valeur 1bh, le bit (C) est positionné à la valeur 1, la retenue auxiliaire (A) est mise à 0.

Par contre si nous considérons les deux instructions suivantes ;

```
mov AX , 09h
add AX , 3ah
```

Alors le registre AX contient la valeur 43h, le bit (C) est mis à 0, la retenue auxiliaire (A) est mise à 1.

➢ *Soustraction sans retenue*

SUB registre/variable, registre

SUB registre, registre/variable

SUB registre/variable, constante

L'instruction SUB effectue la soustraction du contenu du registre source au registre destination, sans report de retenue : destination ← destination - source
La retenue est positionnée en fonction du résultat de l'opération.
Par exemple, si les instructions suivantes sont exécutées ;

```
mov AX , 39h
sub AX , 18h
```

Le registre AX contient ensuite la valeur 21h. Les bits (Z), (S) et (C) du registre FLAGS sont mis à 0 car le résultat n'est pas nul, son signe est positif et aucune retenue n'est générée.
Si nous considérons les deux instructions suivantes ;

```
mov AX , 26h
sub AX , 59h
```

Le registre AX contient ensuite la valeur cdh. Le bit (Z) est mis à zéro. Les bits (C), (A) et (S) sont mis à 1.

➢ *Les multiplications en assembleur*

MUL registre/variable ou IMUL registre/variable

L'instruction IMUL effectue la multiplication d'opérandes signés. L'instruction MUL effectue la multiplication d'opérandes non signés.
Les indicateurs de retenue (C) et de débordement (O) sont mis à un si le résultat ne peut pas être stockée dans l'opérande destination.

❖ **Remarque** :

Une addition ou une soustraction de données codées sur n bits donne un résultat sur au plus n + 1 bits. Le bit supplémentaire est la retenue serais e stocké dans le bit (C) de flags. Par contre, une multiplication de deux données de n bits donne un résultat sur 2n bits.
Dans leur première forme qui prend une donnée 8 bits en opérande (donc le résultat est sur 16 bits), les instructions MUL et IMUL effectuent le produit de la valeur contenue dans le registre AL avec la valeur de l'opérande fourni. Le résultat est placé dans le registre AX.
Dans leur deuxième forme qui prend une donnée 16 bits en opérande (donc le résultat est sur 32 bits), les instructions MUL et IMUL effectuent le produit de la valeur contenue dans le registre AX avec la valeur de l'opérande fourni. Le résultat est placé dans la paire de registres DX qui contient le poids fort et AX qui contient le poids faible du

❖ **Exemple de multiplication sur 8 bits :**

Soient les instructions suivantes :

```
mov al, 4
mov ah, 25
imul ah
```

Après l'exécution de ces 3 instructions, le registre AH contient la valeur 100, produit de 4 par 25.
Pour bien comprendre la différence entre les instructions MUL et IMUL regardons les exemples suivants :

```
mov BX , 435
mov AX , 2372
```

```
imul BX
```

Après l'exécution de ces 3 instructions, AX contient la valeur be8c et DX la valeur f, soit la valeur hexadécimale fbe8c, c'est-à-dire 1031820, le produit de 435 par 2372. Les deux données étant positives, le résultat est le même que l'on utilise l'instruction IMUL ou l'instruction MUL.
Considérons maintenant la séquence d'instructions :

```
mov BX , -435
mov AX , 2372
imul BX
```

À l'issue de leur exécution, AX contient la valeur 4174 et DX contient la valeur fff0, soit la valeur négative -1031820. Si l'on remplace IMUL par MUL, le résultat n'a pas de sens.

➢ *Les divisions en assembleur*

DIV registre/variable ou IDIV registre/variable

Les deux instructions DIV et IDIV réalisent les opérations de division et de calcul de reste. DIV l'effectue sur des données non signées, IDIV sur des données signées.
Dans tous les cas, le dividende est implicite. Le diviseur est fourni en opérande. Le résultat se compose du quotient et du reste de la division. Le reste est toujours inférieur au diviseur. On a le choix entre :
– la division d'une donnée 16 bits stockée dans AX par une donnée 8 bits qui fournit un quotient dans AL et un reste dans AH sur 8 bits.
– la division d'une donnée 32 bits stockée dans la paire de registres DX (poids fort) et AX (poids faible) par une donnée 16 bits qui fournit un quotient dans AX et un reste dans DX sur 16 bits.
Soient les quelques lignes d'assembleur suivantes :

```
mov AX , 37
mov dx, 5
div dl
```

Après leur exécution, le registre AX contient la valeur 7, quotient de 37 par 5 et le registre DX contient le reste de la division, 2.
Pour les deux instructions, si le diviseur de la division est nul, un message *Division par zero* sera affiché automatiquement.
Dans le cas d'une division signée IDIV, le reste a le même signe que le dividende et sa valeur absolue est toujours inférieure au diviseur.
Pour bien comprendre le fonctionnement de ces deux instructions, prenons l'exemple suivant :

```
mov BX , 435
mov AX , 2372
div BX
```

Après l'exécution de cette séquence d'instructions, le registre AX contiendra la valeur 5 qui est le quotient de 2372 par 435, et le registre DX vaudra la valeur 197 qui est le reste de la division. Si on remplace DIV par IDIV, le résultat est inchangé puisque le diviseur et le dividende sont tous deux positifs.
Si l'on considère maintenant la séquence :

```
mov BX , -435
mov AX , 2372
idiv BX
```

On aura ensuite AX qui contient la valeur -5 (soir FFFb en hexadécimal) et DX la valeur 197. Si l'on remplace IDIV par DIV, le résultat n'a pas de sens.

➢ *Décrémentation*

DEC registre/variable

DEC soustrait 1 au contenu de l'opérande, sans modifier l'indicateur de retenue.
Soient les instructions assembleur suivantes ;

```
mov al, 01h
dec al
```

AL contient ensuite la valeur 00. Le bit (Z) est mis à 1, les bits (O), (P), (A) et (S) mis à 0.

➢ *Incrémentation*

INC registre/variable

INC ajoute 1 au contenu de l'opérande, sans modifier l'indicateur de retenue.
Soient les instructions assembleur suivantes ;

```
mov al, 3fh
inc al
```

AL contient ensuite la valeur 40h. Le bit (Z) est mis à 0 (le résultat de l'incrémentation n'est pas nul), l'indicateur de retenue auxiliaire, le bit (A) est mis à 1 (passage de retenue entre les bits 3 et 4 lors de l'incrémentation), les bits (O) et bit (S) sont mis à 0 (pas de débordement de capacité, le signe du résultat est positif).

➢ *Opposé d'un nombre (Négation par complément à 2)*

NEG registre/variable

NEG transforme la valeur d'un registre ou d'un opérande mémoire en son complément à deux. Ainsi, après exécution des instructions ;

```
mov AX , 35
neg AX
```

Le registre AX contient la valeur -35.

b. Les instructions booléennes et logiques

Ces instructions sont disponibles sur tous les processeurs et agissent sur les bits des données et non leurs valeurs

➢ *Le ET logique*

AND registre/variable, registre

AND regisre, registre/variable

AND registre/variable, constante

AND réalise un ET logique bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.
Considérons les deux lignes suivantes :

```
mov al, 36h
and al, 5Ch
```

Le registre AL contient ensuite la valeur 14h obtenue de la manière suivante ;

```
  36h    0011 0110
^ 5Ch    0101 1100
  14h    0001 0100
```

Les bits (S) et (Z) sont mis à zéro et le bit (P) à 1.

➢ *Le OU logique*

```
OR registre/variable, registre
OR regisre, registre/variable
OR registre/variable, constante
```

OR réalise un OU logique bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.
Considérons les deux lignes suivantes :

```
mov al, 36h
or al, 5ch
```

Le registre AL contient ensuite la valeur 7Eh obtenue de la manière suivante ;

```
  36h    0011 0110
∨ 5ch    0101 1100
  7eh    0111 1110
```

Les bits (S) et (Z) sont mis à zéro et le bit (P) à 1.

➢ *Le OU exclusif*

```
XOR registre/variable, registre
XOR registre, registre/variable
XOR registre/variable, constante
```

XOR réalise un ou-exclusif bit à bit entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.
Considérons les deux lignes suivantes :

```
mov al, 36h
and al, 5ch
```

Le registre AL contient ensuite la valeur 6Ah obtenue de la manière suivante ;

```
     36h    0011 0110
XOR  5Ch    0101 1100
     6Ah    0110 1010
```

Les bits (S) et (Z) sont mis à zéro et le bit (P) à 1.

➢ *La Négation logique*

```
NOT registre/variable
```

NOT transforme la valeur d'un registre ou d'un opérande mémoire en son complément logique bit à bit.
Considérons les deux lignes suivantes :

```
mov al, 36h
not al
```

Le registre AL contient ensuite la valeur C9h obtenue de la manière suivante ;

```
¬ 36h    0011 0110
C9h      1100 1001
```

Les bits (S) et (P) sont mis à 1, le bit (Z) à 0.

c. Les instructions de décalage et rotation

Nous décrivons ici des opérations classiques des langages d'assemblage qui se nomment décalages et rotations. On les rencontre couramment car leur utilisation simplifie grandement certains traitements : **RCL, ROL, RCR et ROR**

Les rotations sont des opérations logiques binaires fréquemment utilisées. Elles considèrent un opérande (octet ou mot) comme un tore dont elles décalent les bits. Lors du décalage, un bit déborde d'un coté, à gauche ou à droite, selon le sens de la rotation. Selon le cas, quelques détails diffèrent ;

RCL : le bit de poids fort est mis dans l'indicateur de retenue C, la valeur de cet indicateur étant préalablement mise dans le bit de poids faible

ROL : le bit de poids fort est mis dans l'indicateur de retenue C et dans le bit de poids faible de l'opérande. L'ancienne valeur de l'indicateur de retenue n'est pas utilisée

➢ *L'instruction RCL.*

```
RCL registre/variable, 1
RCL registre/variable, CL
```

RCL effectue une rotation à gauche de l'opérande destination indiqué, 1 ou CL fois, en prenant en compte le contenu de l'indicateur de retenue. Le bit de poids fort de l'opérande destination est mis dans la retenue. Le contenu de la retenue est mis dans le bit de poids faible de l'opérande destination.

➢ *L'instruction RCR.*

```
RCR registre/variable, 1
RCR registre/variable, CL
```

RCR effectue une rotation à droite de l'opérande destination indiqué, 1 ou CL fois, en prenant en compte le contenu de l'indicateur de retenue. Le bit de poids faible de l'opérande destination est mis dans la retenue. Le contenu de la retenue est mis dans le bit de poids fort de l'opérande destination.

➢ *L'instruction ROL :*

```
ROL registre/variable, 1
ROL registre/variable, CL
```

ROL effectue une rotation à gauche de l'opérande destination indiqué, 1 ou CL fois, sans prendre en compte le contenu de l'indicateur de retenue. Le bit de poids fort est mis dans l'indicateur de retenue lors de la rotation ainsi que dans le bit de poids faible de l'opérande.

➢ *L'instruction ROR :*

```
ROR registre/variable, 1
ROR registre/variable, CL
```

ROR effectue une rotation à droite de l'opérande destination indiqué, 1 ou CL fois, sans prendre en compte le contenu de l'indicateur de retenue. Le bit de poids faible est mis dans l'indicateur de retenue lors de la rotation ainsi que dans le bit de poids fort de l'opérande.

d. Les décalages en assembleur

Une opération de décalage consiste simplement à décaler tous les bits d'une donnée. Contrairement aux rotations qui considèrent une donnée (un octet ou un mot) comme un tore, un décalage considère la donnée comme une file ; ainsi, le bit qui 'déborde' de la retenue est perdu.

➢ *Les instructions SAL et SHL :*

```
SAL registre/variable, 1    ou    SHL registre/variable, 1
SAL registre/variable, CL   ou    SHL registre/variable, CL
```

SAL et SHL sont des synonymes et peuvent être utilisées l'une pour l'autre indifféremment.
Elles effectuent un décalage 1 ou CL fois vers la gauche, sauvegardant le bit de poids fort dans l'indicateur de retenue et mettant un 0 dans le bit de poids faible

➢ *L'instruction SHR :*

```
SHR registre/variable, 1
SHR registre/variable, CL
```

SHR effectue un décalage 1 ou CL fois vers la droite. Le bit de poids faible est mis dans la retenue. Un 0 est mis dans le bit de poids fort de la donnée

➢ *L'instruction SAR :*

```
SAR registre/variable, 1
SAR registre/variable, CL
```

SAR effectue un décalage 1 ou CL fois vers la droite, sauvegardant le bit de poids faible dans l'indicateur de retenue. Cependant le bit de poids fort est conservé et serais donc dupliqué.
Considérons les deux lignes suivantes :

```
mov al, 16h
mov cl, 3
sal al, cl
```

Le registre AL contient ensuite la valeur B0h. En effet, 16h s'écrit 00010110 en binaire. Si on décale cette valeur de trois positions vers la gauche, on obtient 10110000, soit B0h Le bit (C) est mis à 0.
Considérons les deux lignes suivantes :

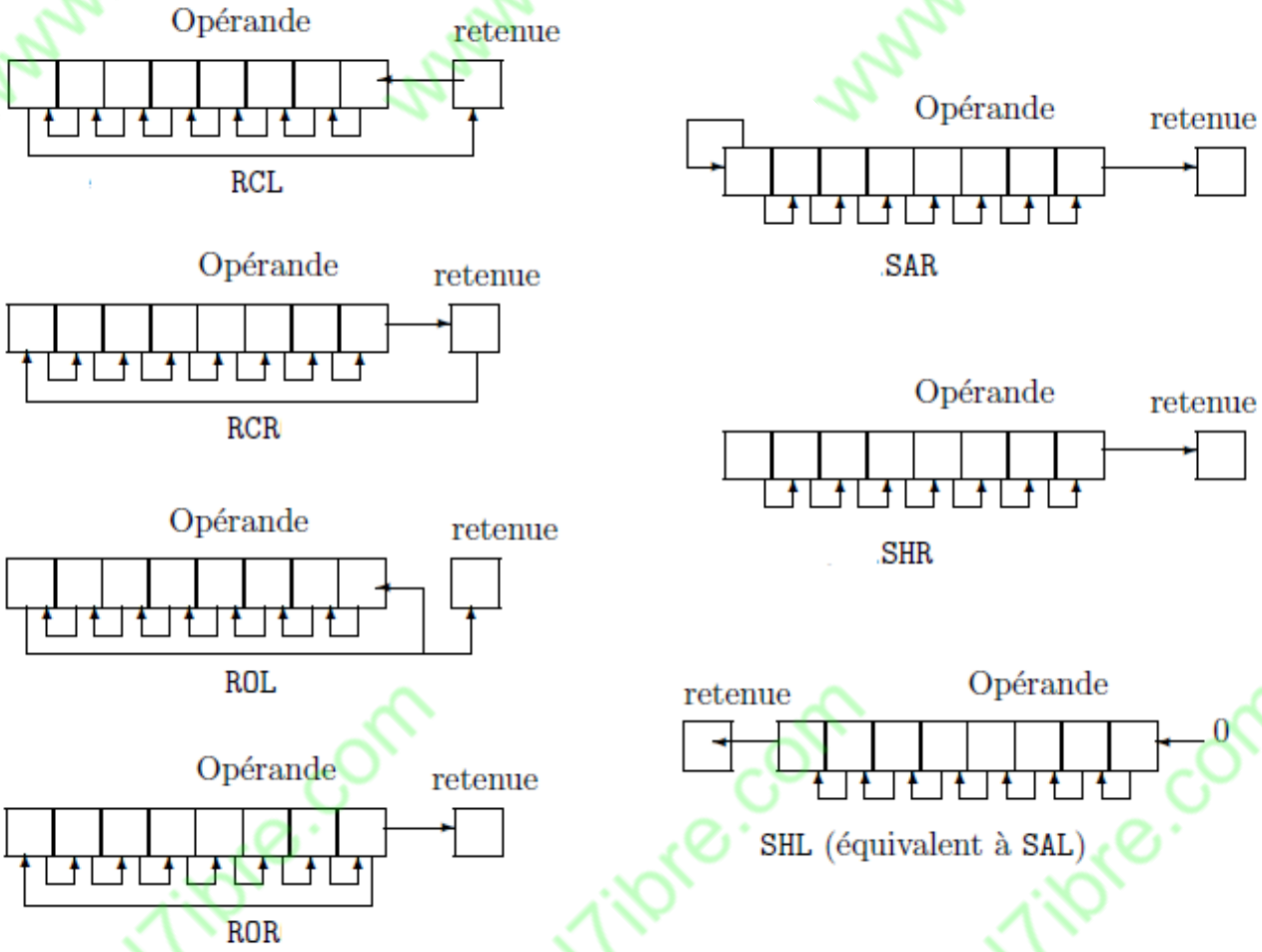
```
mov al, 36h
mov cl, 3
sar al, cl
```

Le registre AL contient ensuite la valeur 05h. Le bit (C) est mis à 1.
La différence entre les instructions SAR et SAL n'apparaît que si le bit de poids fort de la donnée vaut 1. Le tableau suivant donne, pour deux valeurs de la donnée (mise dans le registre AL), l'effet des différents décalages.

```
Si al contient 70 alors          Si al contient f0 alors
sar AL, 1 →  AL contient 38      sar AL, 1 →  AL contient f8
shr AL, 1 →  AL contient 38      shr AL, 1 →  AL contient 78
```

Il faut noter que pour un nombre, un décalage d'une position vers la gauche correspond à une multiplication de ce nombre par 2 et qu'un décalage d'une position vers la droite correspond à une division entière par 2. En généralisant, un décalage de *k* positions vers la gauche correspond à une

multiplication par 2^k et un décalage de k positions vers la droite correspond à une division par 2^k .



Rotation et décalage en assembleur

e. Les instructions de transfert

➢ *Transfert d'une valeur*

Ces instructions réalisent des transferts de données entre 2 adresses mémoire, 2 registres, ou entre un registre et la mémoire. Ceci correspond donc à l'affectation des langages de haut niveau A:= B.

```
MOV registre/variable, registre
MOV registre, registre/variable
MOV registre, registre de segment
MOV registre de segment, registre
MOV registre/variable, constante
```

Avec registre de segment indique l'un des registres CS, DS, ES ou SS.
L'instruction MOV effectue le transfert d'une donnée vers un registre ou une adresse mémoire.

Aucun mode d'adressage de MOV ne permet de transférer une valeur d'un registre de segment vers un autre registre de segment

Par exemple, supposons que nous voulions transférer le contenu du registre DS dans le registre ES. Nous pouvons l'écrire sous la forme suivante :

```
mov AX , ds
mov es, AX
```

La valeur du registre AX a été perdue lors de l'exécution de la première instruction

➢ *Chargement de l'offset d'une donnée*

LEA registre/variable, étiquette

Cette instruction charge l'offset de la donnée référencée dans le deuxième opérande qui est, en général, une donnée déjà déclarée dans le segment de données.

➢ *Échange de valeurs*

XCHG registre/variable, registre

XCHG registre, registre/variable

L'instruction XCHG échange le contenu de deux emplacements mémoire ou registres. Supposons que le registre AX contienne la valeur 20 et le registre DX la valeur 10, l'exécution de l'instruction : `xchg AX, DX` entraînera : AX contient la valeur 10 et DX la valeur 20.

4. Les tests

En assembleur, il n'existe pas de tests comme dans les langages C. Cependant, il est bien entendu possible de réaliser des tests. On utilise pour cela les bits du registre FLAGS comme condition de test et une instruction de branchement conditionnelle (saut si certains bits du registre FLAGS valent 0 ou 1) pour déclencher la partie alors ou la partie sinon du test.

Les bits du registre FLAGS sont positionnés par les instructions que nous avons déjà vues (instructions arithmétiques, logiques, ...). Ils sont également positionnés par des instructions spécialement conçues pour réaliser des tests et qui n'ont d'autres effets que de positionner les bits de FLAGS en fonction de certaines conditions sur leurs opérandes. Ces instructions sont CMP et TEST.
Une fois les indicateurs positionnés, une instruction dite de saut conditionnel teste un bit ou une combinaison de bits de FLAGS et, en fonction du résultat :
– effectue une rupture de séquence (un saut) vers un endroit précis dans le code où l'exécution se poursuit normalement.
– continue en séquence si le test ne donne pas un résultat positif.

f. Les instructions de comparaison

➢ *Comparaison CMP*

C'est l'instruction la plus utilisée pour positionner les indicateurs avant d'effectuer une instruction de saut conditionnel.

```
CMP registre/variable, registre
CMP registre, registre/variable
CMP registre/variable, constante
```

Elle permet de comparer deux valeurs et soustrait le second opérande du premier, sans modifier l'opérande destination, mais en positionnant les indicateurs en fonction du résultat.
Ainsi, si le résultat de la soustraction est nul, l'indicateur (Z) est positionné à 1, ce qui signifie que les deux valeurs comparées sont égales. En suivant les mêmes raisonnements, on peut savoir si les deux valeurs sont différentes, ordonnées strictement ou non.
Après l'exécution des deux instructions suivantes :

```
mov al, 23
cmp al, 34
```

Le registre AL n'est pas modifié par l'exécution de l'instruction CMP et contient toujours la valeur affectée auparavant 23. L'indicateur de retenue (C) est positionné à 1, ce qui indique que le deuxième opérande de l'instruction CMP est supérieur à la valeur du premier opérande. L'indicateur de zéro (Z) valant 0, ce qui signifie que les deux données sont différentes (sinon, la soustraction d'un nombre à lui-même donne 0, donc le bit (Z) est positionné à 1). Le bit de signe (S) est également mis à 1 car 29 − 34 est un nombre négatif.

g. Les instructions de saut conditionnel

Toutes les instructions de saut conditionnel prennent le même type d'opérande

```
JA    étiquette     saut si supérieur (C =0 et Z =0)
JAE   étiquette     saut si supérieur ou égal (C =0)
JB    étiquette     saut si inférieur (C =1)
JBE   étiquette     saut si inférieur ou égal (C =1 ou Z =1)
JC    étiquette     saut si retenue (C =1)
JCXZ étiquette      saut si CX vaut 0
JE    étiquette     saut si égal (Z =1)
JG    étiquette     saut si supérieur (Z =0 ou S =O)
JGE   étiquette     saut si supérieur ou égal (S =O)
JL    étiquette     saut si inférieur (S 6=O)
JLE   étiquette     saut si inférieur ou égal (Z =1 ou S 6=O)
JNC   étiquette     saut si pas de retenue (C =0)
JNE   étiquette     saut si non égal (Z =0)
JNO   étiquette     saut si pas de débordement (O =0)
JNP   étiquette     saut si pas de parité (P =0)
JNS   étiquette     saut si pas de signe (S =0)
JO    étiquette     saut si débordement (O =1)
JP    étiquette     saut si parité (P =1)
JS    étiquette     saut si signe (S =1)
```

Toutes ces instructions fonctionnent selon le schéma suivant. Quand la condition est vraie, un saut est effectué à l'instruction située à l'étiquette spécifiée en opérande.
Notons que les tests d'ordre (inférieur, supérieur, ...) se comprennent de la manière suivante : Supposons que nous comparions deux données par une instruction CMP et que nous exécutions ensuite une instruction JG, c'est-à-dire 'saut si plus grand', il y aura alors saut si la valeur du deuxième opérande de l'instruction CMP est supérieur à la valeur du premier opérande.
L'étiquette référencée dans l'instruction de saut conditionnel ne doit pas se trouver trop loin de l'instruction de saut. Sinon, une erreur d'assemblage est déclenchée et le message :
Relative jump out of range by xxx bytes est affiché

h. Les instructions de saut inconditionnel

➢ *L'instruction JMP*

JMP étiquette

L'instruction JMP effectue un saut inconditionnel à l'étiquette spécifiée. Contrairement à un saut conditionnel, le saut est toujours effectué, le registre FLAGS n'intervenant en rien dans cette opération.

Généralement parlant, un test dans un langage évolué a la forme suivante :

```
SI (condition vraie) ALORS
    action-alors
SINON
     action-sinon
FIN_SI
```

Si la valeur de l'expression condition est vraie, les instructions composant la partie action-alors sont exécutées. Sinon, les instructions de la partie action-sinon sont exécutées.
En assembleur, ce type de construction est réalisé à l'aide du squelette de programme suivant :

```
calcul de la condition
Jcc SINON1
action-alors
...
JMP FSI1
SINON1:
action-sinon
...
FSI1:
...
```

Où **Jcc** dénote l'une des instructions de saut conditionnel.
Remarquons que :
– le calcul de la condition positionne les indicateurs du registre FLAGS
– en fonction de ce positionnement, un saut conditionnel est réalisé par l'instruction Jcc (où cc représente la condition à tester) vers la partie sinon du test. Si la condition est vérifiée, aucun saut n'est réalisé et le programme poursuit son exécution en séquence avec les instructions de la partie action-alors
– une fois les instructions de la partie alors exécutées, un saut doit être réalisé (instruction JMP FSI1) pour atteindre la fin de la structure de test afin de ne pas également exécuter la partie sinon
– une étiquette (SINON1) est déclarée indiquant le début de la partie action-sinon où le branchement conditionnel doit être exécuté pour déclencher l'exécution des instructions de la partie sinon du test
– une fois la partie sinon exécutée, on continue en séquence avec les instructions situées après le test (c'est-à-dire, après l'étiquette FSI1)

5. Les boucles en assembleur

Comme pour les tests, il n'existe pas de structure générale en assembleur pour coder une boucle. Cependant, à l'aide des instructions vues précédemment pour réaliser des tests, on peut coder n'importe quel type de boucle. On verra également que des instructions simplifient grandement le codage des boucles pour.

a. Principe général

Le principe de la réalisation des boucles tant-que ou jusqu'à est assez particulier. Une boucle est essentiellement composée de deux éléments :
– une condition de sortie qui indique quand la boucle doit être interrompue, qu'il faut sortir de la boucle et continuer l'exécution des instructions en séquence
– un corps de boucle qui spécifie l'action à réaliser pendant que la condition de sortie n'est pas vérifiée, à chaque itération de la boucle
La condition de sortie va donc être codée d'une manière ressemblant aux tests. Un saut conditionnel testera cette condition et entraînera, quand la condition est vérifiée, la sortie de la boucle.
Le corps de la boucle devra pour sa part 'boucler', c'est-à-dire, une fois exécuté, entraîner le recalcul de la condition de sortie et la prise de décision quant à la poursuite ou non des itérations.

b. Types de boucles

➢ *Boucles tant-que*

Le squelette d'une boucle tant-que, dans un langage de haut niveau, est le suivant :

```
TANT-QUE (condition) FAIRE
     action
FIN_TQ
```

Cela va se traduire en assembleur sous la forme suivante :

```
TQ1: calcul de la condition
     Jcc FTQ1
     action
     ...
     JMP TQ1
FTQ1:
```

...
Où Jcc dénote l'une des instructions de saut conditionnel vues plus haut.
Nous notons les points suivants :
– une étiquette TQ indique le début de la boucle
– la boucle commence par une évaluation de la condition de sortie qui positionne les indicateurs du registre FLAGS
– en fonction de la valeur des indicateurs (donc du résultat du test de sortie), un saut conditionnel est effectué en fin de boucle (Jcc FTQ1), pour quitter la boucle le moment venu
– on trouve ensuite le corps de la boucle, terminé par une instruction de saut inconditionnel vers le début de la boucle (JMP TQ1) qui permettra de ré-évaluer la condition d'arrêt après chaque itération
– une étiquette FTQ indiquant la fin de la boucle
Pour des raisons de clarté, nous utiliserons un format spécial pour les étiquettes codant une boucle TANT-QUE. Une étiquette TQ indique le début de la structure de boucle et une étiquette FTQ indique la fin de la structure de boucle. Ces étiquettes seront numérotées, les boucles étant numérotées dans le programme de 1 à n, par convention.

➢ *Boucles répéter*

Le squelette d'une boucle REPETER dans un langage de haut niveau est le suivant :

```
REPETER
     action
JUSQUA (condition vraie)
```

Cela se traduit de la manière suivante en assembleur :

```
REPETER1:
     action
     ...
     calcul de la condition
     Jcc REPETER1
```

Remarquons que :
– une étiquette REPETER repère le début de la boucle
– on trouve ensuite le corps de la boucle
– à l'issue de l'exécution du corps, on trouve l'évaluation du test d'arrêt qui positionne les indicateurs du registre FLAGS
– une instruction de saut conditionnel effectue un branchement pour continuer les itérations si la condition d'arrêt n'est pas vérifiée

➢ *Les boucles pour*

Une boucle *pour* est généralement codée à l'aide d'une instruction de la famille LOOP.
LOOP boucles contrôlées par un compteur

LOOP étiquette : saut court si le compteur est différent de 0

LOOPE étiquette : saut court si compteur différent de 0 et Z =1

LOOPNE étiquette : saut court si compteur différent de 0 et Z =0

LOOP fonctionne avec le registre CX qui joue le rôle de compteur de boucles, elle décrémente le compteur sans modifier aucun des indicateurs. Si le compteur est différent de 0, un saut à l'étiquette opérande de l'instruction LOOP est réalisé. La syntaxe d'une boucle pour est :

```
POUR indice := 1 A n FAIRE
     action
FPOUR
```

Cependant, en assembleur, seules existent des boucles ayant un indice variant d'une certaine valeur n à 1, en décrémentant sa valeur à chaque itération. Anssi, la boucle précédente doit être transformée comme suit :

```
POUR indice := n A 1, pas := -1 FAIRE
     action
FPOUR
```

Qui se traduit en assembleur de la manière suivante :

```
mov cx, n
POUR1: ...
actio1
l    oop POUR1
```

Exemple : Ecrire un programme qui affiche dix fois la chaîne de caractères 'hello world'.

```
Data Segment
msg: db 'hello world', 13, 10, '$'
     End Segment
Code Segment
mov AX , @data     ; initialisation de la valeur
```

```
mov ds, AX   ; de ds
mov cx, 10    ; initialisation du compteur de
boucles
;
; corps de boucle
;
boucle: mov ah, 9    ; affichage du message
lea dx, msg
int  21h
loop boucle     ; contrôle de la boucle
...
```

Le début initialise le registre DS pour qu'il contienne le numéro du segment de données et le compteur de boucles. Le corps de la boucle contient alors un appel à la routine affichant un message à l'écran.

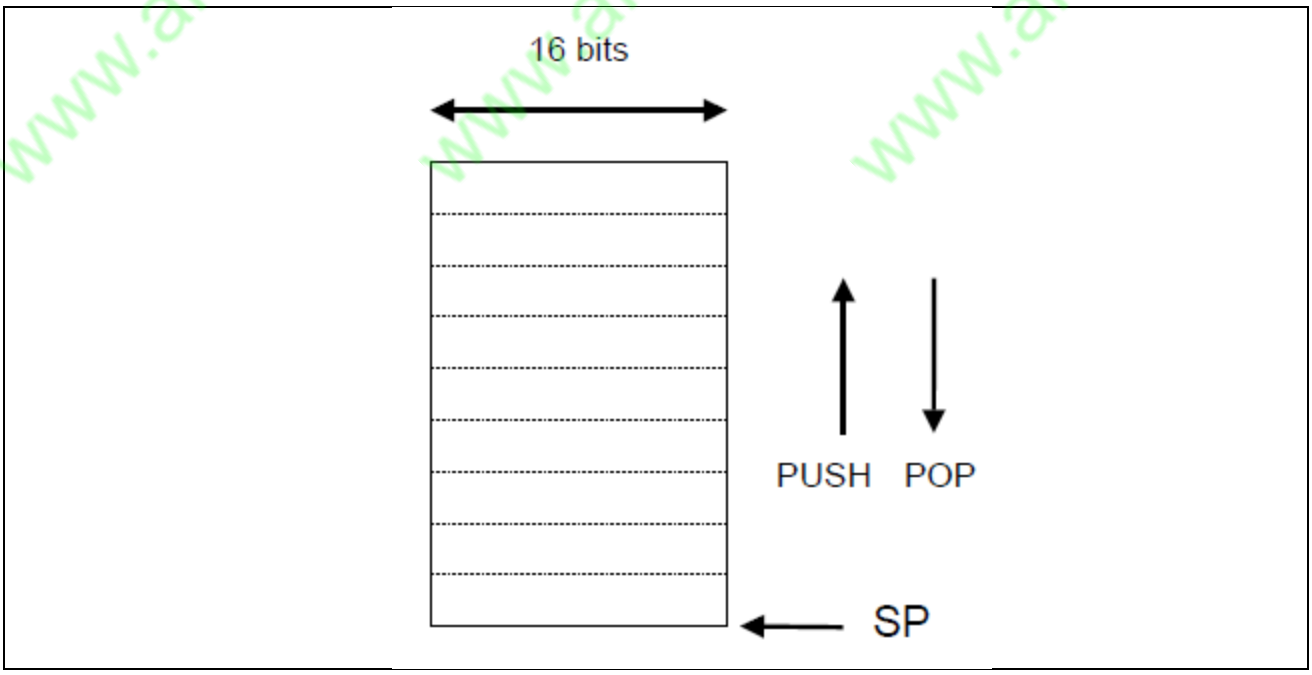
Il est très courant de parcourir un tableau dans une boucle tout en voulant sortir de la boucle lorsqu'une condition est vérifiée (parce que l'on cherche une valeur dans le tableau possédant une propriété particulière par exemple). Des instructions sont disponibles en assembleur pour réaliser ce genre de boucle, les instructions LOOPE et LOOPNE (les instructions LOOPZ et LOOPNZ étant des synonymes). Comme pour les boucles LOOP, le registre CX joue le rôle de compteur de boucle. Par ailleurs, à chaque itération, une condition (la valeur du bit Z du registre FLAGS) est testée. Pour l'instruction LOOPE, il y a saut à l'étiquette spécifiée en opérande tant que le compteur est non nul et que l'indicateur Z vaut 1. Pour l'instruction LOOPNE, il y a saut à l'étiquette spécifiée en opérande tant que le compteur est non nul et que l'indicateur Z vaut 0.

IV. PILE

Structure de « rangement » de données définie dans un segment de mémoire particulier (.exe) ou dans le même que le code et les données (.com) fonctionnement *LIFO* (Last In, First Out). L'accés à cette zone mémoire n'est jamais absolue mais toujours relativement à un registre. Elle est gérée au moyen de deux registres :

- **Un registre de base (« *Base Pointer* », ou BP)** : qui pointe sur le début de la zone mémoire allouée pour les variables locales du programme courant
- **Un registre de sommet de pile (« *Sttack Pointer* », ou SP) :** il pointe sur le dernier mot mémoire alloué et il contient le déplacement du sommet de la pile (16 bits de poids faible de son adresse) par rapport au registre SS (*Stack Segment*) qui le registre segment contenant l'adresse du segment de pile courant (16 bits de poids fort de l'adresse). Ce dernier est normalement initialisé au début du programme et reste fixé par la suite.

La pile peut conserver plusieurs valeurs, la valeur dépilée par POP est la *dernière* valeur empilée; c'est pourquoi on parle ici de pile LIFO (*Last In First Out*,Premier Entré Dernier Sorti).



1. Accès à la pile

L'adresse du dernier élément posé dans l'emplacement SS:SP. Ils existent deux opérations essentielles pour manipuler une pile :

- **PUSH** ***registre :*** qui empile le contenu d'un registre de 16 bits sur la pile.

L'instruction PUSH effectue les opérations suivantes :

SP ← SP - 2
[SP] ← valeur du registre 16 bits.

Notons qu'au début (pile vide), SP pointe ``sous'' la pile

- **POP** ***registre :*** qui dépile la pile en retirant la valeur en haut de la pile et en la plaçant dans le registre de 16 bits spécifié.

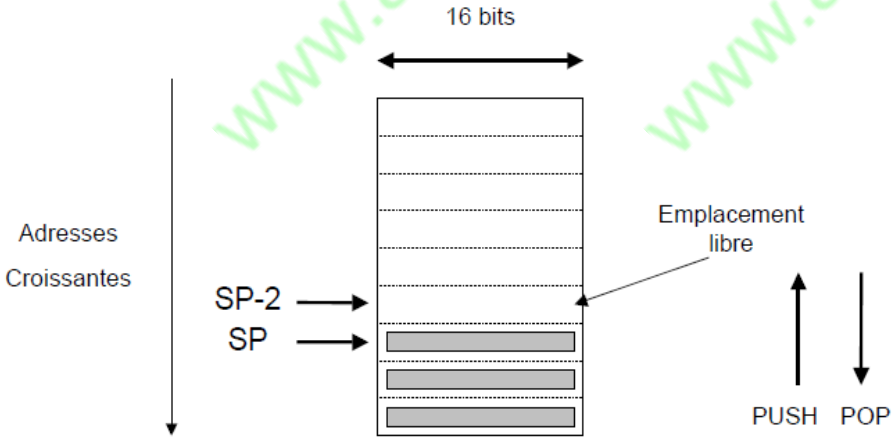
L'instruction POP effectue le travail inverse :

registre destination ← [SP]
SP ← SP + 2

Si la pile est vide, POP va lire une valeur en dehors de l'espace pile, donc imprévisible.

– **Exemple** : transfert d'AX vers BX en passant par la pile.

```
PUSH AX  ; Pile ← AX
POP BX   ; BX  ← Pile
```



La pile est souvent utilisée pour sauvegarder temporairement le contenu des registres :

```
; AX et BX  contiennent des données à conserver
PUSH AX
PUSH BX
MOV BX , truc ; on utilise AX
ADD AX , BX  ; et BX
POP BX       ; récupère l'ancien BX
POP AX       ; et l'ancien AX
```