Langage Assembleur : les bases

.1. Introduction

Dire que « le langage assembleur est simple » a sa part de vérité. Son exécution requiert peu de mémoire et ses instructions sont, pour la plupart, de bas niveau. Alors pourquoi il a la réputation d'être difficile ?

Voici un programme simple en assembleur (masm) d'un programme qui ajoute deux nombres et affiche le résultat :

```
main PROC
mov eax,5          ; move 5 to the EAX register
add eax,6          ; add 6 to the EAX register
call WriteInt      ; display value in EAX
exit               ; quit
main ENDP
```

« WriteInt » simplifie le code considérablement, mais en générale le langage assembleur n'est pas difficile. Le programmeur doit donner plus d'importance aux petits détails ce qui rend le code beaucoup plus volumineux.

.2. Syntaxe du langage : Eléments de base

Dans ce qui suit, nous ferons le tour des éléments de base de la syntaxe du MASM.

.2.1. Constantes et expressions entières :

- La notation Microsoft est utilisée tout au long de ce chapitre. Les éléments entre crochets [..] sont facultatifs et les éléments entre accolades {..} nécessitent un choix de l'un des éléments inclus (séparés par le caractère |). Les éléments en italique désignent des éléments qui ont des définitions ou des descriptions connues.

Une constante entière est composée d'un signe, d'un ou plusieurs digits et d'une base :

[{+ | −}] digits [base]

Les bases possibles sont :

h Hexadecimal	r Encoded real
q/o Octal	t Decimal (alternate)
d Decimal	y Binary (alternate)
b Binary	

Exemples :

26 Decimal	42o Octal
26d Decimal	1Ah Hexadecimal
11010011b Binary	0A3h Hexadecimal
42q Octal	

- Une constante hexadécimale commençant par une lettre doit être précédée d'un zéro pour empêcher l'assembleur de l'interpréter comme un identifiant.

Une expression entière est une expression mathématique impliquant des valeurs entières et des opérateurs arithmétiques. La valeur de l'expression doit correspondre à un entier qui peut être stocké en 32 bits (0 à FFFFFFFFh).

Exemples :

4 + 5 * 2

12 - 1 MOD 5

-5 + 2

(4 + 2) * 6

.2.2. Constantes et expressions réelles :

Syntaxe :

[sign]integer.[integer][exponent]

Avec :

sign {+,-}

exponent E[{+,-}]integer

Exemples :

2.

+3.0

-44.2E+05

26.E5

.2.3. Constantes caractères et chaines de caractères

Un caractère ou une chaine de caractères est entourée de guillemets simples ou doubles.

Exemples :

'A'

"d"

'ABC'

'X'

.2.4. Mots réservés

Les mots réservés ont une signification connue par MASM. Il ne peuvent etre utilisé que dans des contextes bien définis.

Il y a plusieurs types de mots réservés :

- Mnémonique d'instruction
- Nom de registre
- Directive
- Opérateur
- Symboles prédéfinis

Le programmeur peut être amené à choisir un identifiant pour une variable, une procédure ou autre. Il y a un certain nombre de règles à respecter lors du choix d'un identifiant :

- Il peut contenir de 1 à 247 caractères
- Le premier caractère doit être une lettre, le caractère de soulignement _,@, ? ou &
- Il doit être diffèrent des mots réservés

.2.5. Directives

Une directive est une commande qui permet de définir des variables, des macros et des procédures. Elle peut attribuer un nom à un segment de mémoire …

L'exemple suivant explique la différence entre une directive et une instruction

```
myVar DWORD 26          ; DWORD directive
mov eax,myVar           ; MOV instruction
```

➢ Définir les segments :

Parmi les fonctions principales des directive est celle de définition des segments de code :

```
.data
; dans cette partie du code on définit les données
.code
; dans cette partie on écrit les instructions
```

.2.6. Instructions

Une instruction est une commande sui devient exécutable quand le programme est assemblé. Elle est traduite par l'Assembleur en langage machine, qui lui est exécuté par le CPU dans la phase de l'exécution.

➢ La syntaxe d'une instruction est la suivante :

```
[label:]  mnémonique  [opérandes] [;comment]
```

- Label

C’est un identifiant qui permet de référencer une instruction. Un label placé juste avant une instruction équivaut à son adresse (de la même manière, le label d’une variable équivaut à son adresse)

- Exemple :

```
compteur DWORD 100
```

Dans cet exemple, l’Assembleur affecte une valeur numérique au label compteur.

Il est possible de définir plusieurs données avec un seul label

- Exemple :

```
Tableau  DWORD 100,5467
         DWORD 200,400
```

Un « code label » est un label de code. Il est utilisé pour référencer l’instruction et servent dans des instructions telles que JMP et LOOP. Un label de code doit être suivi de « : »

- Exemple :

```
target:
mov ax,bx
...
jmp target
```

- Mnémonique d’instruction

C’est un mot court qui identifie une instruction.

- Exemple :

Mov Move (assign) one value to another

Add Add two values

Sub Subtract one value from another

mul Multiply two values

jmp Jump to a new location

call Call a procedure

- Opérandes

Une instruction peut avoir de zéro à trois opérandes.

Un opérande peut être un nom de registre, un opérande mémoire ou une constante.

Example	Operand Type
96	Constant (*immediate value*)
2 + 4	Constant expression
eax	Register
count	Memory

- Exemples :

```
stc                 ; set Carry flag
inc eax             ; add 1 to EAX
mov count,ebx       ; move EBX to count
imul eax,ebx,5      ; ebx is multiplied by 5, and the product is stored in the EAX register
```

- Commentaires

Les commentaires sur une seule ligne sont précédés par « ; »

Un ensemble de commentaires est précédé par COMMENT

- Exemples :

```
        ; set Carry flag
COMMENT &
```

Knqsdf

Sqdf

Sqdf

Sgert

&

- Structure d’un programme

TITLE titre du programme

INCLUDE ….

.data

; (insert variables here)

.code

main PROC

; (insert executable instructions here)

exit

main ENDP

; (insert additional procedures here)

END main

- Premier programme

```
TITLE Add and Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
.386
.model flat,stdcall

include \masm32\include\masm32rt.inc

.code
main PROC

mov eax,10000h ; EAX = 10000h
add eax,40000h ; EAX = 50000h
sub eax,20000h ; EAX = 30000h

exit
main ENDP
END main
```

.3. Définition de données

.3.1. Syntaxe

La définition d'une donnée permet d'allouer de l'espace mémoire pour cette donnée. Le nom de la donnée est optionnel.

La syntaxe pour définir (déclarer) une donnée (variable) est :

[name] directive initializer [,initializer]...

La directive dans cette syntaxe peut être une parmi les types dans la table suivante :

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Ou dans la table suivante :

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

- Exemple :

```
compteur DWORD 12345
value1 BYTE 'A' ; character constant
value2 BYTE 0 ; smallest unsigned byte
value3 BYTE 255 ; largest unsigned byte
value4 SBYTE −128 ; smallest signed byte
value5 SBYTE +127 ; largest signed byte
val1 DB 255 ; unsigned byte
val2 DB -128 ; signed byte
```

```
list BYTE 10,20,30,40
list BYTE 10,20,30,40
 BYTE 50,60,70,80
 BYTE 81,82,83,84
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h ; list1 and list2 have the same contents
greeting1 BYTE "Good afternoon",0 ; greeting1 BYTE 'G','o','o','d'....etc.
greeting2 BYTE 'Good night',0
word1 WORD 65535 ; largest unsigned value
word2 SWORD -32768 ; smallest signed value
word3 WORD ? ; uninitialized, unsigned
quad1 QWORD 1234567812345678h
```

- L'opérateur DUP :

Il permet d'allouer de l'espace pour plusieurs données à la fois.

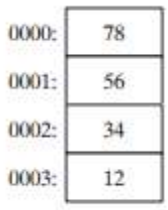
- Exemple :

```
BYTE 20 DUP(0) ; 20 bytes, all equal to zero
BYTE 20 DUP(?) ; 20 bytes, uninitialized
BYTE 4 DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
array WORD 5 DUP(?) ; 5 values, uninitialized
```

- Structure little-endian

Les CPU x86 enregistrent et lisent les données de la mémoire en utilisant la structure little-endian. (low to hight).

Par exemple la représentation de 12345678h en mémoire est la suivante :



- Si on modifie le programme vu dans la section précédante en ajoutant les variables :

```
TITLE Add and Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
.386
.model flat,stdcall

include \masm32\include\masm32rt.inc

.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
mov eax,val1 ; start with 10000h
add eax,val2 ; add 40000h
sub eax,val3 ; subtract 20000h
mov finalVal,eax ; store the result (30000h)

exit
main ENDP
END main
```

.3.2. Déclarer des données non initialisées

La directive .data ? est utilisée au lieu de la directive .data pour déclarer des données non initialisées.

- Exemple :

```
.data

smallArray DWORD 10 DUP(0) ; 40 bytes

.data?

bigArray DWORD 5000 DUP(?) ; 20,000 bytes, not initialized
```

.3.3. Les constantes symboliques

Une constante symbolique (ou définition de symbole) est créée en associant un identifiant (un symbole) à une expression entière ou du texte. Les symboles ne sont utilisés que par l'Assembleur lors de l'analyse d'un programme, et ils ne peuvent pas être changés à l'exécution.

- La directive « = »

Cette directive associe un nom à une expression entière. Sa syntaxe est la suivante :

Name = expression (x = 30 par exemple)

Quand le programme est assemblé toutes les occurrence de name sont remplacées par sa valeur.

- Exemple :

```
COUNT = 5
mov al,COUNT ; AL = 5
COUNT = 10
mov al,COUNT ; AL = 10
COUNT = 100
mov al,COUNT ; AL = 100
```

- Le compteur d'emplacement courant :

Comme son nom l'indique, il permet de représenter l'emplacement courant dans le segment courant (l'offset)

- Exemple :

```
selfPtr DWORD $          ; declares a variable named selfPtr and initializes it with its own location counter
```

.3.4. Calculer la taille des tableau et des chaines de caractère

Dans l'exemple qui suit, nous utilisons le compteur d'emplacement courant pour connaitre la taille d'une liste (tableau) d'éléments.

```
list BYTE 10,20,30,40

ListSize = ($ - list)
```

Les deux instructions doivent se suivre !! (la raison est évidente)

Cet exemple donne donne une taille erronée :

```
list BYTE 10,20,30,40

var2 BYTE 20 DUP(?)

ListSize = ($ - list)
```

Nous calculons la taille des chaines de caractère de la meme manière :

```
myString BYTE "This is a long string, containing"

 BYTE "any number of characters"

myString_len = ($ − myString)
```

Lorsqu'on calcule la taille d'un tableau d'un type autre que BYTE, il faut faire attention à diviser par la taille du dit type :

Exemple :

```
list WORD 1000h,2000h,3000h,4000h

ListSize = ($ − list) / 2

list DWORD 10000000h,20000000h,30000000h,40000000h

ListSize = ($ − list) / 4
```

.4. Affectation de données, adressage et arithmétique

.4.1. Instructions d'affectation

.4.1.1. Types d'opérandes

Pour donner plus de flexibilité au code, le langage assembleur utilise plusieurs types d'opérandes d'instructions. Les plus utilisés sont :

- Valeur immédiate
  Mov eax,32
- Nom d’un registre
  Mov eax,ebx
- Opérande mémoire :
  var1 BYTE 10h
  mov AL,var1

.4.1.2. L’instruction mov

MOV registre1, registre2 a pour effet de copier le contenu du registre2 dans le registre1, le contenu préalable du registre1 étant écrasé. Cette instruction vient de l'anglais « move » qui signifie « déplacer » mais attention, le sens de ce terme est modifié, car l'instruction MOV ne déplace pas mais place tout simplement. Cette instruction nécessite deux opérandes qui sont la destination et la source. Ceux-ci peuvent être des registres généraux ou des emplacements mémoire. Cependant, les deux opérandes ne peuvent pas être toutes les deux des emplacements mémoire. De même, la destination ne peut pas être ce qu'on appelle une valeur immédiate (les nombres sont des valeurs immédiates, des valeurs dont on connait immédiatement le résultat) donc pas de MOV 10, AX. Ceci n'a pas de sens, comment pouvezvous mettre dans le nombre 10, la valeur d’AX ? 10 n'est pas un registre.

MOV reg,reg

MOV mem,reg

MOV reg,mem

MOV mem,imm

MOV reg,imm

- ➢ Exemples :

Mov ax, bx ;

Mov ah, cl ;

Mov esi,edi ;

- Règles d'utilisation de mov :

- Il est interdit de transférer le contenu d'une case mémoire vers une autre case mémoire.
- Cs, EIP et IP ne sont jamais utilisés comme registre destination.
- On ne peut pas transférer un registre segment vers un autre registre segment.
- Les opérandes doivent avoir la même taille
- On ne peut pas utiliser une valeur immédiate avec un registre segment

- Exemple

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
.code
mov eax,0              ; EAX = 00000000h
mov al,oneByte         ; EAX = 00000078h
mov ax,oneWord         ; EAX = 00001234h
mov eax,oneDword       ; EAX = 12345678h
mov ax,0               ; EAX = 12340000h
```

- Copier le plus petit dans le plus grand

Quoique mov ne le permet pas, on peut contourner le problème.

Supposons que count(16 bits) doit être copiée dans ECX (32 bits), on peut faire ceci :

```
.data
count WORD 1
.code
```

```
mov ecx,0
mov cx,count
```

.4.1.3. L'instruction MOVX

Cette instruction copie le contenu de la source dans la destination tout en complétant avec des zero :

MOVZX reg32,reg/mem8

MOVZX reg32,reg/mem16

MOVZX reg16,reg/mem8

- exemple :

```
.data
byteVal BYTE 10001111b
.code
Movzx ax,byteVal      ; AX= 0000000010001111b
```

.4.1.4. L'instruction MOVSX

Cette instruction copie le contenu de la source dans la destination tout en complétant avec des 1

MOVSX reg32,reg/mem8

MOVSX reg32,reg/mem16

MOVSX reg16,reg/mem8

- exemple :

```
.data
byteVal BYTE 10001111b
.code
movsx ax,byteVal          ; AX = 1111111110001111b
```

.4.1.5. L'instruction XCHG

Permet d'échanger les valeurs de deux opérandes

XCHG reg,reg

XCHG reg,mem

XCHG mem,reg

- Exemple

```
xchg ax,bx          ; exchange 16-bit regs
xchg ah,al          ; exchange 8-bit regs
xchg var1,bx        ; exchange 16-bit mem op with BX
xchg eax,ebx        ; exchange 32-bit regs
;pour échanger deux opérandes mémoire :
mov ax,val1
xchg ax,val2
mov val1,ax
```

.4.1.6. Les opérandes offset

On peut ajouter un déplacement au nom d'une variable, ceci permet de créer un opérande d'offset direct. Cette opération permet d'accéder à des localisations mémoire qui n'ont pas forcement de labels.

Soit la définition de données suivante :

```
arrayB BYTE 10h,20h,30h,40h,50h
mov al,arrayB               ; AL = 10h
```

On peut accéder au deuxième octet en ajoutnat 2 :

```
mov al,[arrayB+2]           ; AL = 30h
```

Dans une expression telle que arrayB+2, nous ajoutons une constante à l'offset d'une variable. Le résultat est une adresse.

Cas d'un word :

```
.data
arrayW WORD 100h,200h,300h
.code
mov ax,arrayW ; AX = 100h
mov ax,[arrayW+2] ; AX = 200h
```

Cas d'un DWORD :

```
.data
arrayD DWORD 10000h,20000h
.code
mov eax,arrayD ; EAX = 10000h
mov eax,[arrayD+4] ; EAX = 20000h
```

Le programme suivant résume cette section :

```
TITLE Add and Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
.386
.model flat,stdcall
include \masm32\include\masm32rt.inc
.data
val1 WORD 1000h
val2 WORD 2000h
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h
```

```
arrayD DWORD 10000h,20000h
.code
main PROC
; Demonstrating MOVZX instruction:
mov bx,0A69Bh
movzx eax,bx ; EAX = 0000A69Bh
movzx edx,bl ; EDX = 0000009Bh
movzx cx,bl ; CX = 009Bh
; Demonstrating MOVSX instruction:
mov bx,0A69Bh
movsx eax,bx ; EAX = FFFFA69Bh
movsx edx,bl ; EDX = FFFFFF9Bh
mov bl,7Bh
movsx cx,bl ; CX = 007Bh
; Memory-to-memory exchange:
mov ax,val1 ; AX = 1000h
xchg ax,val2 ; AX=2000h, val2=1000h
mov val1,ax ; val1 = 2000h
; Direct-Offset Addressing (byte array):
mov al,arrayB ; AL = 10h
mov al,[arrayB+1] ; AL = 20h
mov al,[arrayB+2] ; AL = 30h
; Direct-Offset Addressing (word array):
mov ax,arrayW ; AX = 100h
```

```
mov ax,[arrayW+2] ; AX = 200h
; Direct-Offset Addressing (doubleword array):
mov eax,arrayD ; EAX = 10000h
mov eax,[arrayD+4] ; EAX = 20000h
mov eax,[arrayD+4] ; EAX = 20000h
exit
main ENDP
END main
```

4.2. Addition et soustraction

4.2.1. Incrémentation et décrémentation

INC reg/mem

DEC reg/mem

- Exemples :

```
.data
myWord WORD 1000h
.code
inc myWord ; myWord = 1001h
mov bx,myWord
dec bx ; BX = 1000h
```

4.2.2. Instruction ADD

```
data
var1 DWORD 10000h
var2 DWORD 20000h
.code
```

```
mov eax,var1 ; EAX = 10000h
add eax,var2 ; EAX = 30000h
```

4.2.3. Instruction SUB

```
.data
var1 DWORD 30000h
var2 DWORD 10000h
.code
mov eax,var1 ; EAX = 30000h
sub eax,var2 ; EAX = 20000h
```

4.2.4. Instruction NEG

Cette instruction inverse le signe de son opérande.

Application :

L'instruction en C : Rval = -Xval + (Yval - Zval);

Peut être traduite assembleur :

```
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 40
; first term: -Xval
mov eax,Xval
neg eax ; EAX = -26
; add the terms and store:
add eax,ebx
mov Rval,eax ; -36
```

4.2.5. Opérations arithmétiques

Les expressions arithmétiques sont évaluées moyennant les opérations ADD, SUB et NEG.

- Exemple : soit l'instruction en langage c suivante :

Rval = -Xval + (Yval - Zval);

En langage assembleur, on peut la traduire en le code suivant :

```
.data
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 40
.code
; first term: -Xval
mov eax,Xval
neg eax ; EAX = -26
; second term: (Yval - Zval)
mov ebx,Yval
sub ebx,Zval ; EBX = -10
; add the terms and store:
add eax,ebx
mov Rval,eax ; -36
```

4.2.6. Les flags affectés par l'addition et la soustraction

➢ Rappel sur le registre des flags :

Lorsqu'une instruction arithmétique est exécutée, le registre des flags est souvent affecté. Les valeurs des bits dans ce registre sont d'une grande utilité, c'est en effet grâce à ces valeurs qu'on peut vérifier l'état du programme après une instruction arithmétique.EFLAGS se compose de bits binaires qui contrôlent le fonctionnement du CPU ou reflètent le résultat de certaines opérations du processeur.

Le registre EFLAGS est composé de 32 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
0	NT	IOPF		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF	Appellation

Les 16 premiers bits (0 à 15) sont considérés comme les bots de drapeau.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
										ID	VIP	VIF	AC	VM	RF

Les case qui sont grisées correspondent aux bits réservés ou non définis par Intel.

- CF (Carry Flag), en français indicateur de retenue. Ce bit est positionné à l'état haut 1 s'il y a une retenue à l'issue de l'exécution d'une opération arithmétique.
  mov al,0FFh
  add al,1 ; AL = 00, CF = 1
  Exemple 2 :
  mov ax,00FFh
  add ax,1 ; AX = 0100h, CF = 0
  Exemple 3 :
  mov ax,0FFFFh
  add ax,1 ; AX = 0000, CF = 1
  Exemple 4 : soustraction et CF
  mov al,1
  sub al,2 ; AL = FFh, CF = 1
  Les instructions INC et DEC n'affectent pas le CF. L'instruction NEG appliquée à un opérande non nul met le CF à 1.
- PF (Party Flag), indicateur de parité. Ce bit est positionné à l'état haut 1 si le résultat d'une opération est un nombre pair de bit mis à 1.

```
mov al,10001100b
add al,00000010b ; AL = 10001110, PF = 1
sub al,10000000b ; AL = 00001110, PF = 0
```

- AC (Auxiliary carry), Ce bit est positionné à l'état haut 1 lorsqu'une opération arithmétique provoque un report du bit 3 au bit 4 dans un opérande de 8 bits

```
mov al,0Fh
add al,1 ; AC = 1
```

- ZF (Zero Flag), indicateur de zéro. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat est nul.

```
mov ecx,1
sub ecx,1 ; ECX = 0, ZF = 1
mov eax,0FFFFFFFFh
inc eax ; EAX = 0, ZF = 1
inc eax ; EAX = 1, ZF = 0
dec eax ; EAX = 0, ZF = 1
```

- SF (Sign Flag), indicateur de signe. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat est négatif.

```
mov eax,4
sub eax,5 ; EAX = -1, SF = 1
```

- OF (Overflow Flag), indicateur de débordement de capacité. Ce bit est positionné à l'état haut 1 si, après une opération, le résultat déborde la capacité du registre. Prenons, par exemple, l'addition de la valeur 0FFFF h maximale qui peut être stockés dans un mot de 16 bits avec une valeur 1 quelconque : 0FFFF h + 1h = 10000 h. Le résultat de cette addition est la valeur 10000h et ce nombre ne tient pas sur 16 bits ! Il y a donc un dépassement ! Cet indicateur sert également à signaler des débordements lors de l'utilisation d'arithmétique signé (-32767 à 32768 contre 0 à 65535).

```
mov al,+127
add al,1 ; OF = 1
Exemple 2 :
mov al,-128
sub al,1 ; OF = 1
Exemple 3 :
mov al,-128 ; AL = 10000000b
```

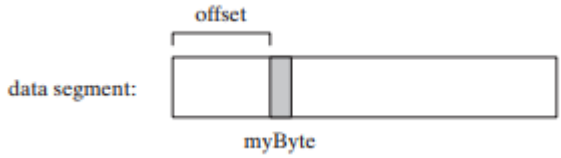
neg al ; AL = 10000000b, OF = 1
Exemple 4 :
mov al,+127 ; AL = 01111111b
neg al ; AL = 10000001b, OF = 0

4.3. Les opérateurs et directives des données

Les opérateurs et les directives ne sont pas des instructions exécutables. ils sont interprétés par l'assembleur. On peut utiliser un certain nombre de directives MASM pour avoir des informations sur les adresses et les tailles des données:

4.3.1. L'opérateur OFFSET

Cet opérateur retourne l'offset d'un label de donnée.



Exemple :

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?
; If bVal were located at offset 00404000
.code
mov esi,OFFSET bVal ; ESI = 00404000
mov esi,OFFSET wVal ; ESI = 00404001
mov esi,OFFSET dVal ; ESI = 00404003
mov esi,OFFSET dVal2 ; ESI = 00404007
```

Exemple 2 :

```
.data
myArray WORD 1,2,3,4,5
.code
mov esi,OFFSET myArray + 4
```

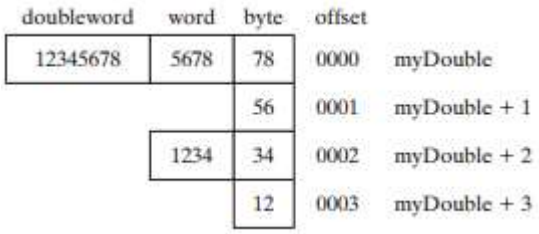
4.3.2. L'opérateur PTR

Soit le code suivant :

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble ; error
```

Mais si on utilise l'opérateur PTR

```
mov ax,WORD PTR myDouble ; correct ax=5678h
```




```
mov ax,WORD PTR [myDouble+2] ; 1234h
mov bl,BYTE PTR myDouble ; 78h
```

On peut aussi utiliser PTR pour affecter une petite source à une destination de taille plus grande :

```
.data
wordList WORD 5678h,1234h
.code
mov eax,DWORD PTR wordList ; EAX = 12345678h
```

4.3.3. L'opérateur TYPE

Soit le code suivant :

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?
```

Dans ce cas l'opérateur TYPRE retourne les valeurs suivantes :

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.4. L'opérateur LENGTHOF

Utilisé plus dans le cadre d'un tableau :

```
.data
byte1 BYTE 10,20,30
array1 WORD 30 DUP(?),0,0
array2 WORD 5 DUP(3 DUP(?))
array3 DWORD 1,2,3,4
digitStr BYTE "12345678",0
```

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

4.3.5. L'opérateur SIZEOF

```
.data
intArray WORD 32 DUP(0)
```

```
.code

mov eax,SIZEOF intArray ; EAX = 64
```

4.4. Adressage indirect

4.4.1. Opérandes indirects

En mode protégé, un opérande indirect peut être un registre général 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP) entouré de « [ ] ». Le registre est supposé contenir l'adresse d'une donnée.

Dans l'exemple suivant, ESI contient l'offset debyteVal. L'instruction MOV utilise l'opérande indirect comme source, le décalage dans ESI est déréférencé, et un octet est déplacé vers AL:

```
.data

byteVal BYTE 10h

.code

mov esi,OFFSET byteVal

mov al,[esi] ; AL = 10h
```

Si l'opérande de destination utilise un adressage indirect, une nouvelle valeur est placée en mémoire à l'emplacementpointé par le registre. Dans l'exemple suivant, le contenu du registre BL est copiéà l'adresse mémoire adressée par ESI.

```
mov [esi],bl
```

4.4.2. Tableaux

Les opérandes indirects sont des outils idéaux pour passer à travers les tableaux. Dans l'exemple suivant, arrayB contient 3 octets. Comme ESI est incrémenté, il pointe sur chaque octet, dans l'ordre:

```
.data

arrayB BYTE 10h,20h,30h

.code

mov esi,OFFSET arrayB
```

```
mov al,[esi] ; AL = 10h
inc esi
mov al,[esi] ; AL = 20h
inc esi
mov al,[esi] ; AL = 30h
```

4.4.3. Opérandes indexés

Un opérande indexé ajoute une constante à un registre pour générer une adresse effective. L'un des 32 des registres à usage général peuvent être utilisés comme registres d'index. Il existe différentes formes de notation autorisé par MASM (« [ ] » font partie de la notation):

Les deux notations suivantes sont équivalentes :

constant[reg]

[constant + reg]

Exemple :

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,OFFSET arrayW
mov ax,[esi] ; AX = 1000h
mov ax,[esi+2] ; AX = 2000h
mov ax,[esi+4] ; AX = 3000h
```

4.4.4. Pointeurs

Un pointeur est une variable qui contient l'adresse d'une deuxième variable. Ce sont des variables de 32 bits (DWORD). Dans l'exemple suivant, ptrB et ptrW contiennent les offsets de arrayB et arrayW

```
arrayB BYTE 10h,20h,30h,40h
```

```
arrayW WORD 1000h,2000h,3000h

ptrB DWORD arrayB ; ou ptrB DWORD OFFSET arrayB

ptrW DWORD arrayW ; ou ptrW DWORD OFFSET arrayW
```

On peut utiliser l'opérateur TYPEDEF pour créer un type qui sera reconnu par l'Assembleur. Dans l'exemple suivant, on utilise TYPEDEF avec les pointeurs :

```
PBYTE TYPEDEF PTR BYTE

.data

arrayB BYTE 10h,20h,30h,40h

ptr1 PBYTE ? ; uninitialized

ptr2 PBYTE arrayB ; points to an array
```

4.5. Les instructions JMP et LOOP

Par défaut, l'Assembleur exécute les instructions séquentiellement. Mais une instruction peur être conditionnée.

Pour tester si une condition est vraie avant d'exécuter une action, on dispose des instructions permettent d'activer des branchements "saut", avec ou sans fonction de test et de choix. Le déroulement du programme est alors dévié de son chemin courant vers un autre, spécifique du choix. On distingue ces instructions de saut en deux catégories suivant que :

- Le saut est effectué quoi qu'il arrive « saut inconditionnel ».
- Le saut est effectué ou non selon l'état d'un registre « saut conditionnel ».

Saut inconditionnel

Ce sont des branchements obligatoires, le processeur saute directement à l'adresse destination. Il procède aux sauts sans aucun test préliminaire. Cela comprend également l'utilisation des interruptions. JMP, CALL, RET, INT, IRET

Saut conditionnel

Ces branchements sont effectifs après validation d'une condition. Il saute à l'adresse seulement si la condition est vraie. Dans le cas contraire, l'exécution se

poursuit avec l'instruction suivante. Ces instructions sont liées aux instructions de test.

LOOP, JCC, REP.

4.5.1. Instruction JMP

L'instruction JMP provoque un transfert inconditionnel vers une destination, identifiée par une étiquette de code. La syntaxe est : JMP destination

Lorsque le microprocesseur exécute un transfert inconditionnel, l'offset de destination est déplacé dans le pointeur d'instruction.

L'instruction JMP permet de créer une boucle en sautant à l'étiquette en haut de la boucle:

```
top:
.
.
jmp top ; repeat the endless loop (JMP is unconditional)
```

4.5.2. Instruction LOOP

Le registre ECX est utilisé comme compteur dans les boucles. Par exemple, pour répéter 10 fois une instruction en assembleur, on peut mettre la valeur 10 dans ECX, écrire l'instruction précédée d'une étiquette qui représente son adresse en mémoire, puis faire un LOOP à cette adresse. Lorsqu'il reconnaît l'instruction LOOP, le processeur "sait" que le nombre d'itérations à exécuter se trouve dans ECX. Il se contente alors de décrémenter ECX, de vérifier que ECX est différent de 0 puis de faire un saut "jump" à l'étiquette mentionnée. Si ECX vaut 0, le processeur ne fait pas de saut et passe à l'instruction suivante.

Exemple :

```
mov ax,0
 mov ecx,5
L1:
 inc ax
```

loop L1

À la fin de la boucle : AX= 5 et ECX = 0

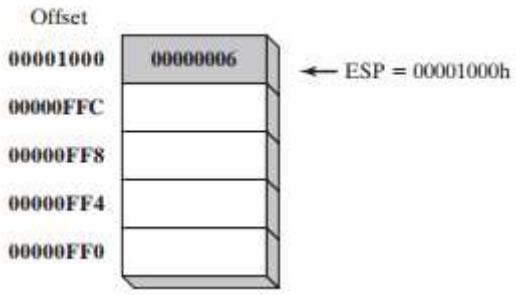
5. Procédures

5.1. Opérations sur la pile

Une pile est une zone de mémoire dans laquelle on peut stocker temporairement des registres. Il s'agit d'un moyen d'accéder à des données en les empilant, telle une pile de livres, puis en les dépilant pour les utiliser. Ainsi il est nécessaire de dépiler les valeurs stockées au sommet (les dernières à avoir été stockées) pour pouvoir accéder aux valeurs situées à la base de la pile.

L'utilisation de la pile est très fréquente pour la sauvegarde temporaire du contenu des registres et pour le passage de paramètres lorsqu'un langage de haut niveau fait appel à une routine en assembleur. La taille de la pile varie en fonction de la quantité d'informations qu'on dépose ou qu'on en retire.

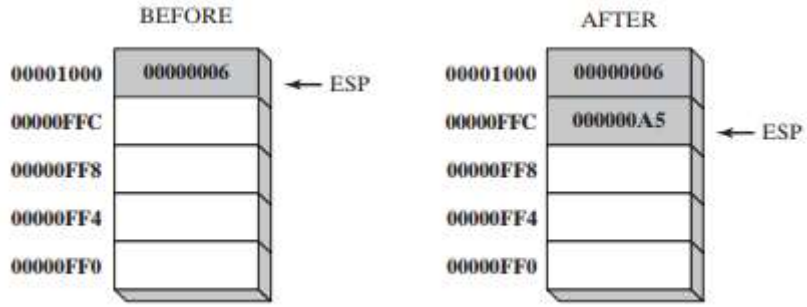
Dans l'architecture x86 32bits, le registre ESP sert à indiquer l'adresse du sommet d'une pile dans la RAM. Les instructions "PUSH" et "POP" permettent respectivement d'empiler et de dépiler des données.



Chaque élément de la pile contient 32 bits (mode protégé 32 bits).

❖ L'opération Push

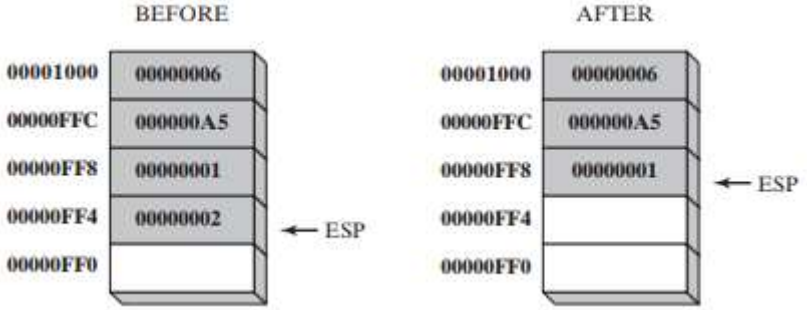
Une opération push décrémente le pointeur de 4 et copie la valeur dans la nouvelle localisation pointée. Le schéma suivant montre l'effet d'une instruction push de la valeur 000000A5



Noter que la pile d'exécution se développe vers le bas dans la mémoire, des adresses plus élevées aux adresses inférieures.

- **L'opération POP**

L'opération POP supprime une valeur de la pile puis incrémente le pointeur pour pointer sur le prochain dernier élément de la pile. Le schéma suivant montre les effets d'un pop de la valeur 00000002



- **<u>Exemples d'utilisation de la pile</u> :**

La pile d'exécution peut servir dans plusieurs cas :

- Enregistrer les valeurs des registres avant qu'ils ne soient modifiés par le programme
- Les instructions "CALL" et "RET" utilisent la pile pour appeler une fonction et la quitter par la suite en retournant à l'instruction suivant immédiatement l'appel.
- En cas d'interruption, les registres EFLAGS, CS et EIP sont automatiquement empilés. Dans le cas d'un changement de niveau de priorité lors de l'interruption, les registres SS et ESP le sont aussi.
  - **<u>Exemple : inverser une chaine de caractère</u>**

Le programme ci-dessous parcourt une chaine de caractère et place les caractères dans la pile. Il récupère ensuite les caractères de la pile et les

insère de nouveau dans la chaine. La logique LIFO de la pile permet donc d'inverser les caractères.

```
TITLE Reversing a String
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) – 1
.code
Main PROC
; Push the name on the stack.
mov ecx,nameSize
mov esi,0
L1: movzx eax,aName[esi]                ; get character
push eax                                ; push on stack
inc esi
loop L1
; Pop the name from the stack, in reverse,
; and store in the aName array.
mov ecx,nameSize
mov esi,0
L2: pop eax                             ; get character
mov aName[esi],al                       ; store in string
inc esi
loop L2
mov edx,OFFSET aName
exit
```

main ENDP

END main

5.2. Définir et utiliser une procédure

Une procédure est un sous-programme qui effectue une tache bien définie. Dans les langages de haut niveau les appellations fonction ou méthode sont parfois employées.

❖ La directive PROC

D'une manière informelle, on peut dire qu'une procédure est un bloc d'instructions qui a un nom et qui se termine par une instruction return.

En langage assembleur, une procédure est déclarée en utilisant les directives PROC et ENDP. Elle doit avoir un nom. (Dans les exemples vus dans ce cours, il y avait toujours une procédure appelée main)

Qu'on on veut créer une procédure autre que le main il faut qu'elle finisse par l'instruction RET (main est un cas particulier, elle de termine par exit)

```
sample PROC
.
.
 ret
sample ENDP
```

❖ Les labels des variables dans les procédures

Par défaut, une variable est uniquement visible dans la procédure ou elle a été déclarée. Cette règle n'affecte en générale que les instructions loop et jump.

jmp Destination (dans ce cas, le label destination doit être définit dans la même procédure que cette instruction)

Exemple de procédure :

```
;-----------------------------------------------------
Sumof PROC
```

```
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
; signed or unsigned.
; Returns: EAX = sum
;---------------------------------------------------
 add eax,ebx
 add eax,ecx
 ret
SumOf ENDP
```

❖ **Les instructions CALL et RET**

L'instruction CALL appelle une procédure en ordonnant au processeur de poursuivre l'exécution à une nouvelle adresse mémoire. La procédure utilise une instruction RET (return from procedure) pour amener le processeur à retourner au point dans le programme où la procédure a été appelée.

- ❖ L'instruction CALL place son adresse de retour dans la pile (push) et copie l'adresse de la procédure appelée dans le pointeur d'instruction. Lorsque la procédure est exécutée, son instruction RET récupère l'adresse de retour de la pile (pop) et la place dans le pointeur d'instruction. En mode 32 bits, la CPU exécute l'instruction en mémoire pointé par EIP (registre du pointeur d'instruction).
- ➢ Exemple

Supposons que dans main, une instruction CALL est localisée à l'offset 00000020. Typiquement, cette instruction nécessite 5 octets de mémoire, de sorte que la déclaration suivante (un MOV dans ce cas) est située à l'offset 00000025:

```
            main PROC
00000020    call MySub
00000025    mov eax,ebx
```

Supposons ensuite que la première instruction exécutable dans MySub se trouve à l'offset 00000040

MySub PROC

00000040 mov eax,edx

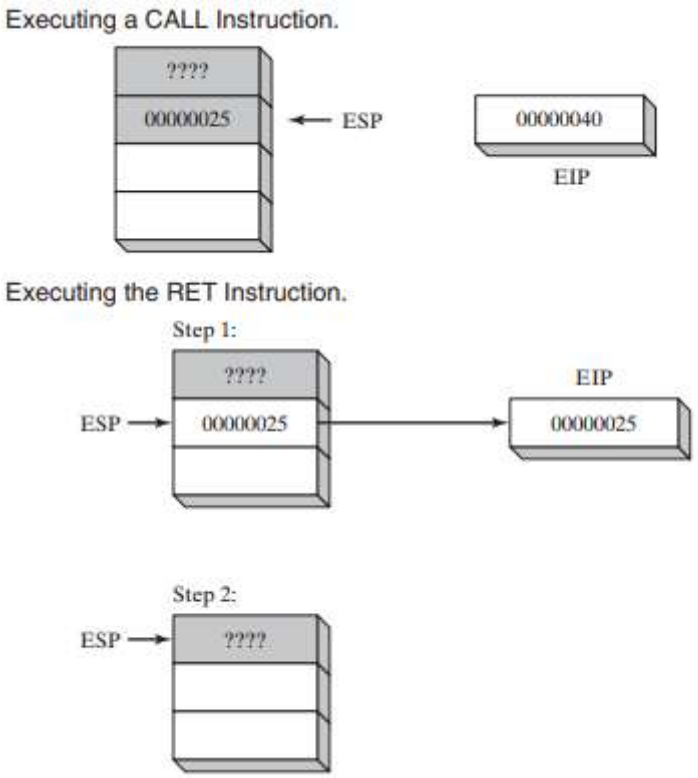
.

.

ret

MySub ENDP

Quand l'instruction CALL est exécutée, l'adresse suivante est stockée dans la pile et l'adresse de MySub est placée dans EIP. L'exécution de RET dans MySub récupère l'adresse pointée par ESP et la place dans EIP.



- Passage d'arguments registres à une procédure

En langage assembleur, il est préférable de passer les arguments des procédures dans les registres généraux

Exemple :

Soit une fonction ArraySum qui reçoit deux paramètres (un tableau et le nombre de ses éléments). Elle calcule et retourne la somme des éléments du tableau dans EAX.

```
ArraySum PROC
push esi ; save ESI, ECX
push ecx
mov eax,0 ; set the sum to zero
L1: add eax,[esi] ; add each integer to sum
add esi,TYPE DWORD ; point to next integer
loop L1 ; repeat for array size
pop ecx ; restore ECX, ESI
pop esi
ret ; sum is in EAX
ArraySum ENDP
.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?
.code
main PROC
mov esi,OFFSET array ; ESI points to array
mov ecx,LENGTHOF array ; ECX = array count
call ArraySum ; calculate the sum
mov theSum,eax ; returned in EAX
```

- L'opérateur USES

Cet opérateur permet de lister les noms des registres modifiés dans une procédure. Suite à l'utilisation de cet opérateur, l'Assembleur génère des

instructions PUSH pour sauvegarder la valeur des registres dans la pile, et des instructions POP pour récupérer ces valeurs après la fin de la procédure.

Exemple :

Suite à ce code :

```
ArraySum PROC USES esi ecx
mov eax,0 ; set the sum to zero
L1:
add eax,[esi] ; add each integer to sum
add esi,TYPE DWORD ; point to next integer
loop L1 ; repeat for array size
ret ; sum is in EAX
```

Le code suivant est généré par l'assembleur :

```
ArraySum PROC
push esi
push ecx
mov eax,0 ; set the sum to zero
L1:
add eax,[esi] ; add each integer to sum
add esi,TYPE DWORD ; point to next integer
loop L1 ; repeat for array size
pop ecx
pop esi
ret
ArraySum ENDP
```

- Exemple de programme utilisant des procédures :

```
TITLE Integer Summation Program
.code
main PROC
; Main program control procedure.
; Calls: PromptForIntegers,
; ArraySum, DisplaySum
exit
main ENDP
PromptForIntegers PROC
.
.
ret
PromptForIntegers ENDP
ArraySum PROC
ret
ArraySum ENDP
DisplaySum PROC
ret
DisplaySum ENDP
END main
```

6. Les structures conditionnelles en assembleur

Une structure conditionnelle est formée par un bloc d'instructions dont l'exécution est conditionnée par une expression logique.

6.1. La directive IF

Sa syntaxe est la suivante :

```
.if( boolean-expression )
 statement-list-1
.elseIf; optional
 statement-list-2
.else ; optional
statement-list-3
```

« boolean-expression » n'est évaluée que pendant l'exécution, voici quelques exemples de conditions valables :

```
eax > 10000h
val1 <= 100
val2 == 100
val3 != eax
(val2 == 100) && (eax > 10000h)
(val2 != ebx) ||  (val2 == 100)
```

6.2. Les boucles

Appelées également les structures répétitives ou itératives. Les directives .REPEAT et .WHILE offrent une solution alternative pour mettre en place des boucles avec CMP et des instructions de saut conditionnel

La directive .REPEAT exécute les instructions du corps de la boucle avant de tester la condition de boucle qui est placée derrière la directive .UNTIL

```
.Repeat
;instructions
.Until condition
```

Le test de la condition est réalisé à la fin, les instructions à l'intérieur sont alors exécutées au moins une fois.

La directive .WHILE est un peu l'inverse de la précédente : elle teste la condition avant d'exécuter le premier tour de boucle :

```
.While condition
;instructions
.endW
```

- Exemple 1: somme des 100 premiers nombres à l'aide de .while

```
Mov eax, 0
Mov ebx, 0
.while eax <100
        Inc eax
        Add ebx, eax
.endW
```

- Exemple 2 : somme des 100 premiers nombres à l'aide de .repeat

```
Mov eax, 0
Mov ebx, 0
.repeat
        Inc eax
        Add ebx, eax
.until eax ==100
```

- Exemple 3 : somme des 100 premiers nombres en sauvegardant la somme des 50 premiers dans une variable res.

```
Mov eax, 0
Mov ebx, 0
.while eax < 100
        .if eax ==50
```

```
            Mov res, ebx ;
            .endif
            Inc eax
            Add ebx, eax
.endW
```

- Remarque :

Il peut être préférable, dans certains cas, de sortir de la boucle avant que la condition d'arrêt n'ait eu lieu. La directive utilisée est .break

.Break [.if condition]