L'assembleur

Introduction

Ce cours vous semblera sans doute long et pénible car ce n'est pas forcément facile de comprendre tout ça, mais efforcez vous à bien vous concentrer dessus car c'est important si vous voulez devenir un bon cracker. Attention, il ne s'agit pas non plus d'apprendre tout cela par cœur. Je vous conseille de prendre des notes et de les garder à côté de vous quand vous crackez ou sinon, pour les fainéants, d'imprimer ce cours disponible au format imprimable sur le Guide PDF.

Dans ce cours vous allez apprendre les bases de l'assembleur. L'assembleur c'est le langage de programmation que vous voyez quand vous désassemblez un fichier dans WinDasm. C'est un langage assez proche du langage machine.

Sommaire :

1° Le processeur et les registres

appelle des registres. Il existe plusieurs types de registres et chacun a son utilité.

- **- registres généraux ou de travail**
  Ils servent à manipuler des données, à transférer des paramètres lors de l'appel de fonction DOS et à stocker des résultats intermédiaires.

- **- registres d'offset ou pointeur**
  Ils contiennent une valeur représentant un offset à combiner avec une adresse de segment

- **- registres de segment**
  Ils sont utilisés pour stocker l'adresse de début d'un segment. Il peut s'agir de l'adresse du début des instructions du programme, du début des données ou du début de la pile.

- **- registre de flag**
  Il contient des bits qui ont chacun un rôle indicateur.

A) Les registres généraux ou de travail

Les 4 registres généraux les plus utilisés sont : AX , BX , CX et DX.

- **AX** -- **accumulateur** -- sert à effectuer des calculs arithmétiques ou à envoyer un paramètre à une interruption
- **BX** -- **registre auxiliaire de base** -- sert à effectuer des calculs arithmétiques ou bien des calculs sur les adresses
- **CX** -- **registre auxiliaire (compteur)** -- sert généralement comme compteur dans des boucles
- **DX** -- **registre auxiliaire de données** -- sert à stocker des données destinées à des fonctions

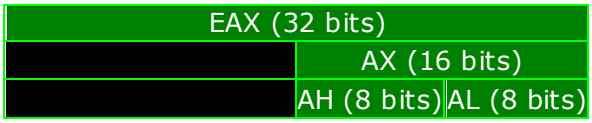
Ceci est leur utilisation théorique, mais dans la pratique ils peuvent être utilisés à d'autres usages.

Ces registres sont de 16 bits (petit rappel : 8 bits = 1 octet et 1 bit = 0 ou 1) et chacun de ses registres est divisé en 2 parties : L (comme Low pour désigner les bits de poids faible) et H (pour High afin de désigner les bits de poids forts). Par exemple : AX (AL et AH), BX (BL et BH), CX (CL et CH), DX (DL et DH).

Pour obtenir un registre de 32 bits il faut rajouter un "E" (pour Extended français "étendu") devant le registre. Par exemple EAX (32 bits) pour AX (16 bits), (EBX

pour BX, ECX pour CX, EDX pour DX). Il existe également des EAH ou EAL, mais la partie haute d'un registre 32 bits ne sera pas directement accessible.

Les bits correspondent en fait à leur capacité : 8 bits = nombre entre 0 et 255 (en décimal) au maximum, 16 bits = 65535 au maximum, 32 bits = 4 294 967 295 au maximum.



B) Les registres d'offset ou pointeur

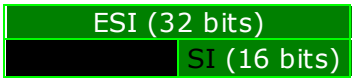
Voici les différents registres :

- **IP -- Pointeur d'instruction --** est associé au registre de segment CS (CS :IP) pour indiquer la prochaine instruction à exécuter. Ce registre ne pourra jamais être modifié directement ; il sera modifié indirectement par les instructions de saut, par les sous-programmes et par les interruptions
- **SI -- Index de source --** est principalement utilisé lors d'opérations sur des chaînes de caractères ; il est associé au registre de segment DS.
- **DI -- Index de destination --** est principalement utilisé lors d'opérations sur des chaînes de caractères ; il est normalement associé au registre de segment DS ; dans le cas de manipulation de chaînes de caractères, il sera associé à ES
- **SP -- Pointeur de pile --** est associé au registre de segment SS (SS :SP) pour indiquer le dernier élément de la pile.
- **BP -- Pointeur de base --** est associé au registre de segment SS (SS :BP) pour accéder aux données de la pile lors d'appels de sous-programmes (CALL).

Comme les registres généraux, ces registres peuvent être étendues à 32 bits en ajoutant le "E" : (EIP, ESI, EDI, ESP, EBP). Par contre ils ne possèdent pas de partie 8 bits (il n' y aura pas de EIH ou ESL par exemple).

Les registres SI et DI sont le plus souvent employés pour les instructions de chaîne et pour accéder à des blocs en mémoire.

Ces registres (sauf IP) peuvent apparaître comme des opérandes dans toutes les opérations arithmétiques et logiques sur 16 bits



C) Les registres de segment

Le processeur utilise des adresses pour stocker ou récupérer des données. L'adresse est composée de 2 parties de 32 bits (ou 16 bits selon le mode utilisé). La première est le segment et la deuxième est l'offset. L'adresse est présenté comme cela : segment:offset (par exemple : 0A000h:00000h). Voici les différents segments :

- **CS -- Registre segment de code --** indique l'adresse du début des instructions d'un programme ou d'une sous-routine
- **DS -- Registre segment de données --** contient l'adresse du début des données de vos programmes. Si votre programme utilise plusieurs segments de données, cette valeur devra être modifié durant son exécution
- **SS -- Registre segment de pile --** pointe sur une zone appelée la pile
- **ES -- Registre segment supplémentaire (Extra segment) --** est utilisé, par défaut, par certaines instructions de copie de bloc. En dehors de ces instructions, le programmeur est libre de l'utiliser comme il l'entend
- **FS -- Registre segment supplémentaire --** Rôle semblable à ES
- **GS -- Registre segment supplémentaire --** Rôle semblable à ES

D) Le registre Flag

Ce registre est un ensemble de 16 bits. Mais cet ensemble ne signifie rien, car les bits sont modifiés un par un. Et la modification d'un seul bit peut changer le comportement du programme. Les bits de cet ensemble sont appelés "indicateurs".

Bit 1 : CF
Bit 2 : 1
Bit 3 : PF
Bit 4 : 0
Bit 5 : AF
Bit 6 : 0
Bit 7 : ZF
Bit 8 : SF
Bit 9 : TF
Bit 10 : IF
Bit 11 : DF
Bit 12 : OF
Bit 13 : IOPL
Bit 14 : NT
Bit 15 : 0
Bit 16 : RF
Bit 17 : VM

Les bits 1, 5, 12, 13, 14, 15 de ce registre Flag de 16 bits ne sont pas utilisés.

Les instructions arithmétiques, logiques et de comparaison modifient la valeur des indicateurs.
Les instructions conditionnelles testent la valeur des indicateurs et agissent en fonction du résultat.

Voici à quoi les principaux indicateurs servent :

CF : *Carry Flag* - **Indicateur de retenue**

A la suite d'une opération, si le résultat est codé sur un nombre supérieur de bits, le bit de trop sera mis dans CF. Plusieurs instructions peuvent modifier son état : CLC pour le

mettre à 0, STC à 1 et CMC inverse sa valeur.

PF : *Parity Flag* - **Indicateur de parité**

Egal à 1 si le nombre de bits d'une opérande (paramètre d'une instruction) est pair.

AF : *Auxiliary carry Flag* - **Indicateur de retenue auxiliaire**

Cet indicateur ressemble à CF.

ZF : *Zero Flag* - **Indicateur zéro**

Si le résultat d'une opération est nul (égal à 0) ZF passera à 1. Cet indicateur est souvent utilisé pour savoir si deux valeurs sont égales (en les soustrayant).

SF : *Sign Flag* - **Indicateur de signe**

SF passe à 1 quand le résultat est signé (négatif ou positif).

IF : *Interruption Flag* - **Indicateur d'interruption**

Il permet d'enlever la possibilité au processeur de contrôler les interruptions. Si IF = 1 les interruptions sont prises en compte et si IF = 0 elles sont ignorées. L'instruction STI met IF à 1 et CLI le met à 0.

DF : *Direction Flag* - **Indicateur de direction**

C'est ce Flag qui donne l'indication sur la manière de déplacer les pointeurs (références) lors des instructions de chaînes (soit positivement, soit négativement). STD le met à 1 et CLD à 0.

OF : *Overflow Flag* - **Indicateur de dépassement**

Il permet de trouver et de corriger certaines erreurs produites par des instructions mathématiques. Très utile pour éviter les plantages. Si OF=1 alors nous avons affaire à un Overflow.

Les autres indicateurs ne sont pas importants nous n'allons donc pas nous attarder dessus.

Petit exemple sur ces indicateurs avec une opération en binaire :

0 1 1 0 + 0 1 0 1 -------- 1 0 1 1	• ZF=0 (le résultat 1011 n'est pas nul) • SF=1 (le signe du résultat est négatif) • CF=0 (il n'y a pas de retenue)

2° La Pile et ses instructions

A) Fonctionnement de la pile

La pile ("stack" en anglais) est un emplacement qui sert à stocker de petites données. Cette mémoire temporaire fonctionne comme une pile d'assiette : c'est-à-dire que la dernière assiette que vous empilez sera la première à être prise et la première assiette empilée (celle qui est tout en dessous) sera la dernière prise. J'espère avoir été clair dans cette métaphore. Pour prendre un vrai exemple, si je mets la valeur "A" dans la pile puis je mets "B" et après "C". Pour récupérer ces valeurs le processeur va d'abord prendre "C", puis "B" et ensuite "A". Là si vous ne comprenez toujours pas je ne peux plus rien pour vous !

Pour l'API GetDlgItemText, Windows a besoin de ces informations :

HWND hDlg, // (1) handle de la boîte de dialogue
int nIDDlgItem, // (2) identifiant de la boîte de dialogue
LPTSTR lpString, // (3) pointe vers le buffer pour le texte
int nMaxCount // (4) taille maximum de la chaîne

On peut donc mettre avant d'appeler cette fonction :

```
MOV EDI, [ESP+00000220]     place l'handle de la boite de dialogue dans EDI
PUSH 00000100               push (4) taille maxi de la chaîne
PUSH 00406130               push (3) adresse du buffer pour le texte
PUSH 00000405               push (2) identificateur du contrôle
PUSH EDI                    push (1) hanlde de la boite de dialogue
CALL GetDlgItemText         appelle la fonction
```

B) L'instruction PUSH

Vous avez pu voir cette instruction au dessus. Push signifie "pousser" en français. C'est avec cette instruction qu'on place une valeur sur la pile. La valeur "poussé" doit être de 16 ou 32 bits (souvent un registre) ou une valeur immédiate.

Par exemple :

```
PUSH AX
PUSH BX
PUSH 560
```

AX est placé en haut de la pile en premier mais ce sera 560 qui en ressortira en premier.

C) L'instruction POP

Mettre des données sur la pile c'est bien, mais le but c'est de les récupérer par la suite. C'est donc à ça que sert l'instruction POP : récupérer les données placées sur la pile.

Par exemple (suite à l'exemple au dessus) :

```
POP CX
POP BX
POP AX
```

Ces instructions mettront la première instruction du haut de la pile dans CX, c'est à dire 560. Et ensuite BX reviendra dans BX et AX dans AX.

La pile est très utilisée pour retrouver les valeurs intactes des registres. Car le nombre de registres est limité, donc on utilise la pile lorsque l'on n'a plus de place. Mais la pile est plus lente que les registres.

3° Les sauts conditionnels et le CMP

A) Les sauts conditionnels

Dans les différents sauts il y a les inconditionnels (qui sautent sans conditions) comme le JMP (Jump) et ceux qui sont conditionnels (qui sautent donc si la condition est vérifiée). Dans cette dernière "famille" il y a du monde. ;-) Les valeurs des indicateurs sont nécessaires pour la vérification de la condition.

Voici les principaux sauts, sachez que J = Jump if (Saut si) :

Indicateur	Valeur	Saut	Signification
CF	1	JB	*Below* - Inférieur
		JBE	*Below or Equal* - Inférieur ou égal
		JC	*Carry* - Retenue
		JNAE	*Not Above or Equal* - Non supérieur ou égale
	0	JA	*Above* - Supérieur
		JAE	*Above or Equal* - Supérieur ou égal
		JNB	*Not Below* - Non inférieur
		JNC	*Not Carry* - Pas de retenue
ZF	1	JE	*Equal* - Egal
		JNA	*Not Above* - Non supérieur
		JZ	*Zero* - Zero (Saut en cas d'égalité)
	0	JNBE	*Not Below or Equal* - Non supérieur ou égal
		JNE	*Not Equal* - Non égal
		JNZ	*Not Zero* - Non zéro (saut si différent)
PF	1	JP	*Parity* - Pair
		JPE	*Parity Even* - Pair
	0	JNP	*Not Parity* - Non pair

		JPO	*Parity Odd* - Non pair
OF	1	JO	*Overflow* - Dépassement
	0	JNO	*Not Overflow* - Pas de dépassement
SF	1	JS	*Signed* - Signé
	0	JNS	*Not Signed* - Non signé

Et quelques autres :

JG - JNLE
Greater (arithmétiquement supérieur) - Not Less or Equal (arithmétiquement non inférieur ou égale)
ZF=0 et SF=OF

JGE - JNL
Greater or Equal (arithmétiquement supérieur ou égal) - Not Less (non inférieur arithmétiquement)
SF=OF

JL - JNGE
Less (arithmétiquement inférieur) - Not Greater or Equal (arithmétiquement non supérieur ou égal)
SF (signé)=OF

JLE - JNG
Less or Equal (arithmétiquement inférieur ou égal) - Not Greater (non supérieur arithmétiquement)
ZF=1 ou SF (signé)=OF

B) L'instruction CMP

Cette instruction permet d'effectuer des tests entre des valeurs et de modifier les flags suivant le résultat. Le fonctionnement du CMP est semblable au SUB (que l'on verra plus loin et qui sert pour la soustraction). Il s'utilise de cette façon : CMP AX, BX. Cela permet de vérifier si ces 2 valeurs sont égales car il fait AX-BX, si c'est égal à 0 alors ZF=1 sinon ZF=0.

Le plus souvent il est suivi de ces sauts et leur inverse (en rajoutant un "N" après le "J", par exemple JNE pour JE...) :

JA : plus grand que (nombres non-signés)
JB : plus petit que (nombres non-signés)
JG : plus grand que (nombres signés)
JL : plus petit que (nombres signés)
JE ou JZ : égal à (signé et non-signé)

4° Les opérations mathématiques

A) Addition et Soustraction : ADD et SUB

Ces 2 instructions nécessitent 2 opérandes : la source et la destination. Ce qui donnera : Destination = Source + Destination. Et en assembleur : ADD Destination, Source.
Exemple avec AX = 4 et BX = 6 :

```
ADD AX, BX
```

Après cette opération : AX = 4+6 = 10 et BX = 6

Le fonctionnement de SUB est identique.

Il est impossible d'ajouter ou soustraire un 16 bits avec un 8 bits par exemple. Donc ADD BX, CL sera impossible !

B) Multiplication : MUL / IMUL

L'instruction MUL est utilisé pour les nombres non signés et IMUL pour les signés (petit rappel : les nombres signés peuvent être négatifs contrairement aux non signés, par exemple en 16 bits en non signé les nombres vont de 0 à 65535 et en signé ils vont de -32768 à 32767). Contrairement aux opérations précédentes, la multiplication n'a besoin que d'une seule opérande : la source. La destination et donc le nombre qui devra être multiplié se trouvera toujours et obligatoirement dans AX. La source est un registre.

Exemple avec AX = 4 et BX = 6

```
MUL BX
```

Résultats : AX = 4*6 = 24 et BX = 6

IMUL fonctionne de la même façon, sauf qu'il faut utiliser l'instruction CWD (convert word to doubleword), qui va permettre "d'étendre" le signe de AX dans DX. Si cela n'est pas fait les résultats pourront être erronés.

C) Division : DIV / IDIV

Pareil que MUL et IMUL : DIV sert pour les nombres non signés et IDIV pour les non signés. Cette instruction divise la valeur de AX par la source.

Exemple avec AX = 18 et BX = 5 :

```
IDIV BX
```

Résultats : AX = 3 et DX = 3 (DX étant le reste)

D) Autre division et multiplication : SHR et SHL

Ces instructions permettent de diviser des registres avec SHR et les multiplier avec SHL. Ils demandent 2 opérandes : le nombre à diviser et le diviseur pour SHR et le nombre à multiplier et le multiplicateur pour SHL. Ils fonctionnent en puissance de 2. Donc si on veut diviser AX par 4 on mettra : SHR AX, 2 (car 2^2 = 4). Pour diviser BX par 16 on mettra : SHR BX,4 (car 2^4 = 16). Pour SHL c'est pareil : pour multiplier CX par 256 on mettra donc : SHL CX,8 (2^8 = 256). Pour multiplier ou diviser des nombres qui ne sont pas des puissances de 2 il faudra combiner des SHR ou SHL.
Petit rappel de mathématiques : x^0 donnera toujours 1 quelque soit x !

Exemple : on veut multiplier 5 par 384. On voit que 384 = 256 + 128, ce qui nous arrange car 256 et 128 sont des puissances de 2.

- MOV AX,5 >> On met 5 dans AX >> AX = 5
- MOV BX,AX >> On met AX dans BX >> BX = AX
- SHL AX,8 >> On multiplie AX par 256 (2^8) >> AX = AX*256 = 5*256 = 1280
- SHL BX,7 >> On multiplie BX par 128 (2^7) >> BX = BX*128 = 5*128 = 640
- ADD AX,BX >> On ajoute BX à AX >> AX = AX + BX = 1280 + 640 = 1920

Et vous pouvez vérifier : 5*384 = 1920 ! ;-)

Mais vous me direz : "pourquoi ne pas faire simplement un MUL ?". Et bien car SHL est beaucoup plus rapide à exécuter. Mais pour des nombres qui ont une décomposition trop longue, mieux vaut bien sur utiliser un MUL.

E) L'opposé d'un nombre : NEG

L'instruction la plus simple à mon avis : NEG sert à rendre un nombre positif en négatif et inversement. Il ne demande qu'une seule opérande : la destination. Si AX = 5, alors un NEG AX mettra AX = -5. Ou si BX = -14 alors après le NEG BX, BX = 14.

5° Les nombres à virgule et les nombres négatifs

A) Les nombres à virgule

Le problème en assembleur c'est qu'on ne peut pas utiliser directement les nombres à virgule. Nous utiliserons les nombres à virgule fixe. Il s'agit simplement d'effectuer des

calculs avec des valeurs assez grandes qui seront ensuite, redivisées pour retrouver un résultat dans un certain intervalle.

Par exemple nous voulons multiplier 20 par 0,25. Comme on ne peut pas mettre 0,25 dans un registre on va le mettre en nombre entier en le multipliant par 256 par exemple. 0.25*256 = 64. On le multipliera à 20 et ensuite ce résultat étant 256 fois trop grand on le redivisera par 256 :

- MOV AX,64 >> On place 0,25*256 dans AX >> AX = 64
- MOV BX,20 >> On place 20 dans BX >> BX = 20
- MUL BX >> On multiplie AX par BX >> AX = AX*BX = 64*20 = 1280
- SHR AX,8 >> On divise AX par 256 (2^8) >> AX = AX/256 = 1280/256 = 5

Le résultat de 20*0,25 est donc bien égal à 5.

Faîtes bien attention à ne pas effectuer de dépassement, surtout avec les nombres signés ou les grands nombres. En utilisant SHR ou SHL, plus l'opérande est grande et plus le résultat sera précis, car il y aura plus de décimales disponibles.

B) Les nombres négatifs

Un autre problème, c'est qu'en décimal pour un nombre négatif on place un "-" devant le nombre, mais en binaire on ne peut mettre que des 0 ou des 1. Pour pouvoir mettre un nombre en négatif on utilise la méthode du complément à 2. Voici en quoi cela consiste :

- Convertir le nombre en binaire s'il ne l'est pas déjà
- Inverser les bits du nombre (inverser les 0 en 1 et 1 en 0)
- Ajouter 1 au nombre obtenu

Voici un exemple avec le nombre décimal 5 au format 8 bits (octet) :

- On le convertit en binaire : 5(d) = 00000101(b)
- On inverse : 00000101 --> 11111010 (ici le format, 8 bits, est important car en 4 bits par exemple cela donnera : 1010)
- On ajoute 1 : 11111010 + 00000001 = 11111011

-5 est donc égal à 11111011 en binaire.

Pour que le nombre soit négatif il faut qu'il soit signé. En 8 bits en non signé les nombres vont de 0 à 255 et en signé ils vont de -128 à 127.

Quand le nombre est signé et en 8 bits, on peut savoir s'il est négatif en regardant le 8ème bit (en partant bien sur de la droite). S'il est égal à 1 alors le nombre est négatif. Pour les nombres à 4 bits c'est le 4ème qui compte.

Pour transformer -5 en hexadécimal, on décompose le nombre binaire en demi-octet (11111011 en 1111 et 1011). 1111b = 15d = Fh et 1011b = 11d = Ah. Donc -5d = FAh. (Petit rappel : d = décimal, b = binaire, h = hexadécimal, pour ceux qui n'auraient pas encore compris).

6° Les instructions logiques

Les opérations de ces instructions se font bit à bit. Et les opérandes et emplacements mémoires qu'elles utilisent sont en 8 ou 16 bits.

Petit rappel : pour convertir de tête un nombre binaire en décimal : chaque rang d'un nombre binaire est un multiple de 2.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Ainsi par exemple 10110110 donnera 128+32+16+4+2 = 182 :

1	0	1	1	0	1	1	0
128	64	32	16	8	4	2	1

A) Et logique : AND

AND (ET) effectue un ET LOGIQUE sur 2 opérandes et le résultat se retrouvera dans la première opérande. Le ET LOGIQUE donne le résultat 1 uniquement quand les 2 opérandes sont 1, sinon il met 0.

opérande 1	opérande 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Exemple avec AX = 15 et BX = 26 :

```
  AND AX,BX
```

Résultat :

```
          0000 1111 (AX=15)
AND       0001 1010 (BX=26)
         -------------
          0000 1010 (AX = 10)
```

B) Ou logique inclusif : OR

OR effectue un OU LOGIQUE INCLUSIF sur 2 opérandes et le résultat se retrouvera dans la première opérande. Le OU LOGIQUE INCLUSIF donne le résultat 0 uniquement quand les 2 opérandes sont 0, sinon il met 1.

opérande 1	opérande 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Exemple avec AX = 15 et BX = 26 :

```
OR AX,BX
```

Résultat :

```
          0000 1111 (AX=15)
OR        0001 1010 (BX=26)
          -------------
          0001 1111 (AX = 31)
```

C) Ou logique exclusif : XOR

XOR effectue un OU LOGIQUE EXCLUSIF sur 2 opérandes et le résultat se retrouvera dans la première opérande. Le OU LOGIQUE EXCLUSIF donne le résultat 1 quand les 2 opérandes sont différentes, sinon il met 0.

opérande 1	opérande 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Exemple avec AX = 15 et BX = 26 :

```
XOR AX,BX
```

Résultat :

```
          0000 1111 (AX=15)
XOR       0001 1010 (BX=26)
```

```
------------
 0001 0101 (AX = 21)
```

D) Non logique : NOT

NOT effectue un NON LOGIQUE sur une opérande et le résultat se retrouvera dans l'opérande. Le NON LOGIQUE inverse la valeur de chaque bit.

opérande	NOT
0	1
1	0

Exemple avec AX = 15 :

```
  NOT AX
```

Résultat :

```
NOT       0000 1111 (AX=15)
        ------------
          1111 0000 (AX = -16 en signé et 240 en non signé)
```

E) TEST

Cette instruction teste la valeur d'un ou plusieurs bits en effectuant un ET LOGIQUE sur les opérandes. Le résultat ne modifie pas les opérandes mais les indicateurs.

Exemple :

```
  TEST AX,1    >> Effectue un ET LOGIQUE sur  le premier bit de AX avec 1, si il
```
est égal à 1, ZF passera à 1 et inversement (si 0 alors ZF=0).

Un TEST AX,AX (souvent utilisé) permet de voir si AX est égal à 0 ou non.

7° La mémoire et ses instructions

A) LEA

Pour placer par exemple la valeur de l'offset de MESSAGE dans SI il suffit de mettre : MOV SI,Offset-Message. Mais on ne peut pas procéder de cette façon pour les registres DS, ES, FS, GS car ceux ci n'acceptent pas les valeurs numériques, ils n'acceptent que les registres. Il faudra donc passer par : MOV AX,Seg-Message puis MOV DS,AX.

Il existe une autre instruction, LEA, qui permet de placer l'offset dans un registre mais en plus court. Par exemple : LEA SI,MESSAGE placera dans SI l'offset de Message. L'utilisation de LEA à la place de MOV rend le code plus clair et plus compact.

B) MOVSx

Pour déplacer des blocs entiers de mémoires il faut utiliser les instructions : MOVSB, MOVSW ou MOVSD selon le nombre de bits à déplacer.

- MOVSB : (B) = Byte (Octet : 8 bits) (Attention !! En anglais "byte"= un octet et "bit"= un bit)
- MOVSW : (W) = Word (Mot : 16 bits)
- MOVSD : (D) = DWord (Double Mot : 32 bits)

Si on veut déplacer 1000 octets (bytes) en utilisant la commande MOVSB il faudra qu'elle se répète 1000 fois. Donc l'instruction REP est utilisée comme une boucle. Il faut placer auparavant le nombre de boucles à effectuer dans le registre de compteur (CX) :

```
MOV CX,1000     >> Nombre de boucles à effectuer dans le compteur CX
REP MOVSB       >> Déplace un octet
```

Pour aller plus vite on peut les déplacer par Mot (Word) :

```
MOV CX,500      >> 1000 bytes = 500 words
REP MOVSW       >> Déplace un mot
```

Ou alors en Double Mot (DWord) :

```
MOV CX,250      >> 1000 bytes = 500 words = 250 dwords
REP MOVSD       >> Déplace un Double Mot
```

A chaque MOVSx, DI augmente de 1,2 ou 4 bytes

C) STOSx

Pour garder des données ou les placer dans un emplacement de la mémoire on utilise les instructions STOSB, STOSW ou STOSD.

Pareil que les MOVSx, ces instructions servent respectivement à stocker un byte, un word et un dword. Par contre ils utilisent AL, AX (EAX 32bits) comme valeur à stocker. La valeur de AL,AX,EAX sera stockée dans ES:DI. A chaque STOSx, DI est augmenté de 1,2 ou 4 bytes.

Exemple :

```
MOV CX,10     >> nombre de répétitions placées dans CX, ici 10
MOV AX,0      >> valeur à déplacer, ici 0
REP STOSB     >> stocker AX dans ES:DI
```

Nous aurons donc 10 bytes de 0 dans ES:DI.