SOMMAIRE

L'ASSEMBLEUR

---

1 - Les bases indispensables pour débuter

Pour cracker n'importe quel logiciel, il est indispensable de connaître le fonctionnement de l'assembleur et ses instructions.
Pour cela, je vous conseille vivement d'acheter les 2 livres suivants :
Assembleur " Une découverte pas à pas " de Philippe Mercier - Edition Marabout n°885 (environ 50 francs).
Assembleur " Théorie, pratique et exercices " de Bernard Fabrot - Edition Marabout n°1087 (environ 50 francs).

Comme vous le verrez, ce cours est surtout destiné à la programmation en asm.

Retour au sommaire

A - Définition de l'assembleur

L'assembleur est un language de programmation transformant un fichier texte contenant des instructions, en un programme que le processeur peut comprendre (programme en language machine).

Ce language machine a la particularité d'être difficile à programmer car il n'est composé que de nombres en hexadécimal (base 16). L'assembleur est une "surcouche" du language machine, il permet d'utiliser des instructions qui seront transformées en language machine donc il présente une facilité de programmation bien plus grande que le language machine. Le fichier texte qui contient ces instructions s'appelle le source.

Retour au sommaire

B - Le language hexadécimal

Nous allons aborder un aspect très important de la programmation en assembleur : le système de numérotation en hexadécimal.
Ce système est basé sur l'utilisation des chiffres et de certaines lettres de l'alphabet (de A à F). Vous connaissez bien entendu le système décimal (base 10).
En assembleur, les nombres décimals sont suivis d'un "d" (1000=1000d) mais en principe la majorité des assembleurs calculent en décimal par défaut.
La notation hexadécimale (base 16) implique qu'il faut disposer de 16 signes alignables dans une représentation et, comme les chiffres ne suffisent plus, on a décidé que les signes de 0 à 9 seraient représentés pas les chiffres 0..9 et les signes manquants pour obtenir 16 signes seraient les 6 premières lettres de l'alphabet soit A, B, C, D, E, F avec :

Hexadécimal	**Décimal**
A	10
B	11
C	12
D	13
E	14
F	15

Nous ne pouvons pas utiliser le G et les lettres qui suivent, donc nous augmenterons le premier chiffre ce qui donne 16d=10h. Continuez ainsi jusqu'à 255 qui fait FF. Et après. Et bien oncontinue. 256d=0100h (le h et le zéro qui précède indiquent que ce nombre est en hexadécimal). 257d=101h. 65535=FFFFh. Bon, je pense que vous avez compris.

Pour convertir des nombres du décimal en hexadécimal, vous pouvez utiliser la calculatrice de Windows en mode scientifique.

Exemples :
8 = **8***1
78 = **7***16 + **8***1
A78 = **10***256 + **7***16 + **8***1
EA78 = **14***4096 + **10***256 + **7***16 + **8***1

Retour au sommaire

C - Le calcul binaire

Je vais finir mon explication avec le calcul binaire. Vous savez probablement que les ordinateurs calculent en base 2 (0 ou 1). Le calcul binaire consiste à mettre un 1 ou un 0 selon la valeur désirée. Chaque 0 ou 1 est appelé un bit, 8 bits forment un octet.

Pour le nombre 1, nous avons 00000001b (le b signifie binaire). Un nombre binaire se décompose comme-ceci :
128|64|32|16|8|4|2|1
Pour le nombre 1, la première case est remplie (1=rempli). Pour avoir le nombre 2, nous aurons 00000010b et pour le nombre 3, nous aurons 00000011 (1 case=1 - 2ème case=1 => 1+2=3).
Le nombre 11111111b donnera 255 puisque en additionnant 1+2+4+...+128, on obtient 255. Plus il y a de bits, plus le nombre peut être grand, nous verrons plus tard qu'il existe différentes unités selon le nombre de bits.

Exemples :

Base 2	Base 10
00000001	1 = 1*1
00000011	3 = 1*2 + 1*1
00001011	11 = 1*8 + 0*4 * 1*2 + 1*1
00011011	27 = 1*16 + 1*8 + 0*4 + 1*2 + 1*1

Retour au sommaire

D - Conversion binaire ⇔ hexadécimal

Voici un exemple démontrant la simplicité de conversion d'un nombre binaire en hexadécimal :

Soit le nombre hexadécimal EA78 à convertir en binaire, 16 étant 2 à la puissance 4, chaque signe de la représentation hexadécimale sera converti en 4 signes binaires et le tour sera joué :

Hexadécimal	Décimal	Binaire
E	14	1110
A	10	1010
7	7	0111
8	8	1000

En alignant les représentations binaires obtenues, on trouve le nombre binaire suivant : 1110101001111000.
Inversement, pour convertir un nombre binaire en hexadécimal, c'est tout aussi simple, on regroupera les bits par groupe de 4 :
Soit 1110101001111000 à traduire en hexadécimal :
On le découpe d'abord en 4 : 1110 1010 0111 1000

Binaire	Décimal	Hexadécimal
1110	14	E
1010	10	10
0111	7	7
1000	8	8

Retour au sommaire

E - Le processeur et ses registres

Le processeur est composé de différentes parties. Les registres sont les éléments les plus importants du processeur pour celui qui programme en asm. Les registres sont souvent représentés comme des cases dans lesquelles sont stockées différentes valeurs de différentes tailles selon le type et le nom du registre. Il existe plusieurs types de registres :

- ***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.

Retour au sommaire

a) Les registres généraux.

On a plusieurs registres généraux (de travail) qui commencent par A,B,C et D. Ces quatre registres sont les plus utilisés.
Le 1er, AX (registre de 16 bits) qui se divise en deux petits registres de 8 bits, AL (l=low=bas) et AH (h=high=haut).
Il est utilisé lors d'opérations arithmétiques.
Nous avons ensuite BX (BL et BH), CX (CL et CH) et DX (DL et DH), ces registres sont divisés comme AX en 2 parties hautes et basses.
On peut rajouter un "E" devant les registres 16 bits afin d'obtenir un registre 32 bits.
Ce qui donne EAX,EBX,ECX et EDX. Notez que l'on ne peut avoir de EAH ou ECL. La partie haute d'un registre 32 bits, n'est pas directement accessible, on doit utiliser différentes instructions afin de la faire "descendre" dans un registre de 16 bits et pouvoir finalement l'utiliser.
Ces registres peuvent contenir une valeur correspondant à leur capacité.
AX=65535 au maximum (16 bits) et AL=255 au maximum (8 bits). Je répète que la partie haute du registre 32 bits ne peut pas être modifiée comme un registre. Elle peut être modifiée seulement si l'on modifie tout le registre 32 bits (y compris la partie basse), ou par le biais de quelques instructions qui permettent de copier la partie basse du registre dans la partie haute, mais cela ne nous concerne pas pour l'instant.
Les processeurs 286 et moins ne peuvent pas utiliser les registres 32 bits (EAX,EBX,ECX,EDX = impossible). Ce n'est qu'à partir du 386 que l'utilisation du 32 bits est possible.

Retour au sommaire

b) Les registres pointeurs ou d'offset

Pour terminer, je vais parler des registres pointeurs qui sont DI (destination index), SI (source index), BP (base pointer), IP (instruction pointer) et SP (stack pointer).

Ces registres peuvent eux aussi être étendus à 32 bits en rajoutant un "E" (EDI,ESI,EIP,ESP,EBP).

Ces registres n'ont pas de partie 8 bits, il n'y a pas de EDL ou de ESH.

- **SI** est appelé " Source Index ". Ce registre de 16 bits est principalement utilisé lors d'opérations sur des chaînes de caractères ; il est associé au registre de segment DS.
- **DI** est appelé " destination Index ". Ce registre de 16 bits 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.

- **BP** ou " Base Pointer ". Ce registre de 16 bits est associé au registre de segment SS (SS :BP) pour accéder aux données de la pile lors d’appels de sous-programmes (CALL).
- **SP** ou " Stack Pointer ". Ce registre de 16 bits est associé au registre de segment SS (SS :SP) pour indiquer le dernier élément de la pile.
- **IP** est appelé " Instruction Pointer ". Ce registre de 16 bits 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.

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

Le registre SP est utilisé pour accéder à la pile.

Nous ne l'employons presque pas, sauf pour du code très spécifique.

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

Retour au sommaire

c) Le processeur et ses registres de segment

Pour pouvoir chercher et placer des choses dans sa mémoire, un ordinateur a besoin de ce qu'on appelle une adresse. Celle-ci se compose de deux parties de 32 bits (ou 16 bits, cela dépend du mode employé). La première est le segment et la deuxième, l'offset.

Voici un exemple :

0A000h:00000h (adresse de la mémoire vidéo)

Dans cet exemple, il s'agit d'un adressage 16 bits. L'adressage 16 bits s'utilise en mode _réel_. Le mode réel est un état du processeur où il ne peut accéder à des blocs de taille supérieure à 16 bits (65536 bytes). Le 286 introduira un mode dit protégé qui permet de franchir cette barrière pour pouvoir accéder à quasiment tout la RAM en un bloc (ce mode protégé sera grandement amélioré à partir du 386).

Nous nous contenterons du mode réel au début, car il est plus simple mais il présente tout de même des limites. Sur 386, les segments ont été conservés pour garder une certaine comptabilité avec les vieilles générations de PC.

Les registres de segment ne sont que lus et écris sur 16 bits. Le 386 utilise un offset de 32 bits (sauf en mode réel où il travaille toujours en 16 bits). Voici la liste de ces segments, ils sont au nombre de 6 :

- **CS (code segment) = segment de code**

Ce registre indique l’adresse du début des instructions d’un programme ou d’une sous-routine

- **SS (stack segment) = segment de pile**

Il pointe sur une zone appelée la pile. Le fonctionnement de cette pile seront expliqués au chapitre 3.

- **DS (data segment) = Segment de données**

Ce registre 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.

- **ES (extra segment) = Extra segment**

Ce registre 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 (extra segment - seulement à partir du 386) = segment supplémentaire**
- **GS (extra segment - seulement à partir du 386) = segment supplémentaire**

Ces deux derniers registres ont un rôle fort similaire à celui du segment ES

Retour au sommaire

d) Le registre Flag

Le registre FLAG est en ensemble de 16 bits. La valeur représentée par ce nombre de 16 bits n'a aucune signification en tant qu'ensemble : ce registre est manipulé bit par bit, certains d'entre eux infuenceront le comportement du programme. Les bits de ce registres sont appelés " indicateurs ", on peut les regrouper en deux catégories :

- Les indicateurs d'état :

Bit	Signification	Abréviation
0	« carry » ou retenue	CF
2	parité	PF
4	retenue auxiliaire	AF
6	zéro	ZF
7	signe	SF
11	débordement (overflow)	OF

- Les indicateurs de contrôle :

Bit	Signification	Abréviation
8	trap	TF
9	interruption	IF
10	direction	DF

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. La description détaillée des flags est faite au chapitre 4.

Retour au sommaire

2 - Les premières instructions

A - La première instruction : MOV

Cette instruction vient de l'anglais "move" qui signifie DEPLACER 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 (deux variables) 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 connaît immédiatement le résultat) donc pas de MOV 10,AX. Ceci n'a pas de sens, comment pouvez-vous mettre dans le nombre 10, la valeur de AX ? 10 n'est pas un registre.

Une autre règle à respecter, les opérandes doivent être de la même taille donc pas de MOV AX,AL, cela me semble assez logique. On ne peut pas utiliser une valeur immédiate avec un registre de segment (MOV ES,5 = impossible mais MOV ES,AX = possible).

Le registre DS est le segment de données par défaut.

Ainsi au lieu de :

MOV AX, DS :[BX]

MOV AX, DS :[SI+2]

…

On aura :

MOV AX, [BX]

MOV AX, [SI+2]

…

Quelques exemples :

MOV AL,CL (place le contenu de CL dans AL) donc si AL=5 et CL=10, nous aurons AL=10 et CL=10.

MOV CX,ES:[DI] (place dans CX, le contenu 16 bits de l'emplacement ES:[DI]). Donc si le word (le word est l'unité correspondant à 16 bits) ES:[DI]=34000, nous aurons CX=34000.

MOV CL,DS:[SI] (place dans CL, le contenu 8 bits (byte) de l'emplacement DS:[SI]).

Donc si DS:[SI] (attention en 8 bits) = 12, CL vaudra 12.

Au sujet de ces emplacements mémoires, il faut que vous vous les représentiez comme des "cases" de 8 bits, comme le plus petit registre fait 8 bits, on ne peut pas diviser la mémoire en emplacements plus petits. On peut prendre directement 16 bits en mémoire (un word) ou carrément un dword (32 bits). Il est très important de saisir cette notion. Un exemple concret : vous avez des livres sur une étagère, le livre se trouvant toute à gauche est le 1er de la liste, nous nous déplaçons de gauche à droite.

Un livre est un byte (8 bits), 2 livres sont un word (16 bits), 4 livres font un dword (32 bits). Si nous faisons MOV AX,ES:[DI], nous allons prendre 2 livres, en commençant par le livre se trouvant à l'emplacement DI et en prenant ensuite le livre se trouvant à l'emplacement DI+1 (l'emplacement suivant dans votre rangée de livre, 1 byte plus loin en mémoire).

Voici un autre exemple :

MOV EAX,ES:[DI] (place un dword depuis ES:[DI] dans EAX). C'est comme si on copiait 4 livres dans EAX.

Et quelques exemples incorrects :

MOV 1,10 (impossible et pas logique)

MOV ES:[DI],DS:[SI] (incodable malheureusement)

MOV ES,10 (incodable, il faut passer par un registre général donc MOV ES,AX ou MOV

ES,CX mais pas de MOV ES,AL).

Ce qu'il faut retenir c'est que l'instruction MOV est comme le signe '=' MOV ES,CX c'est comme faire ES=CX.

Retour au sommaire

B - Une autre instruction : JMP

JMP est une simplification pour exprimer le mot JUMP qui signifie SAUTER en anglais.

JMP va servir dans le programme pour "passer" d'une opération à une autre, de sauter dans le programme à différents endroits pour effectuer d'autres tâches.

Je ne vais pas m'attarder dans des explications sur le pointeur d'instruction mais sachez que chaque instruction est désignée par un emplacement en mémoire (défini par CS:[IP]).

L'instruction JMP va donc modifier la valeur de IP et par conséquence changer d'instruction.

On peut utiliser JMP avec des valeurs immédiates.

Exemples :

JMP 10 (il aura pour effet de passer, non pas à l'instruction qui se trouve à la ligne 10 , mais à l'emplacement mémoire 10. Une instruction n'est pas forcément de taille 1, certaines peuvent s'étaler sur plusieurs bytes.)

Il est plus simple d'utiliser des "étiquettes qu'on peut écrire sous la forme "ETIQUETTES:".

Avec les étiquettes, il suffit d'écrire JMP ETIQUETTES pour arriver directement à l'instruction qui suit le étiquette. Un petit exemple de code:

```
DEBUT :
MOV AX,14
JMP DEBUT
```

Ce programme ne fait rien sinon placer continuellement 14 dans AX, on appelle cela une boucle.

Nous verrons plus tard qu'il existe différents types de sauts.

Retour au sommaire

C - Quelques instructions arithmétiques : ADD et SUB

ADD sert à additionner, et nécessite deux opérandes : une source et une destination.

La destination prendra la valeur de source + destination. Les règles de combinaison sont les mêmes que pour MOV mais avec SUB et ADD, l'utilisation des registres de segment est impossible.

Exemples :

```
ADD AX,CX (si AX=10 et CX=20 alors nous aurons AX=30 et CX=20)
ADD EAX,EBX
ADD AX,-123 (l'assembleur autorise les nombres négatifs)
SUB DI,12
```

Quelques exemples impossibles :

```
ADD AX,CL (comme pour MOV, un 16 bits avec un 8 bits sont incompatibles)
SUB ES:[DI],DS:[SI]
ADD ES,13 (impossible car c'est un registre de segment)
```

Retour au sommaire

3 - Pile - Interruptions

A - La pile (Stack) et ses instructions

La pile (stack en anglais) est un emplacement où des données de petites tailles peuvent être placées. Cette mémoire temporaire (elle se comporte comme la mémoire vive mais en plus petit) est utilisée aussi dans les calculatrices et dans tous les ordinateurs. Le système qu'elle emploie pour stocker les données est du principe du "dernier stocké, premier à sortir" (LIFO -last in, first out). Cela veut dire que lorsqu'on place une valeur (un registre, un nombre) dans la pile, elle est placée au premier rang (placée en priorité par rapport aux autres valeurs dans la pile). Lors de la lecture de la pile pour récupérer la valeur, ce sera la première qui sera prise. La pile se comporte comme une pile d'assiettes à laver.

Au fur et à mesure que les assiettes sont sales, vous les empilez. L'assiette qui se trouve tout en bas sera la dernière à être lavée, tandis que la première assiette empilée sera lavée au tout début. Même chose en asm, sauf les assiettes vont finir dans des registres.

Avant que n'importe quel appel de fonction (API Windows par exemple), le programme doit pousser tous les paramêtres auxquels la fonction s'attend sur la pile. On utilise donc cette pile comme une mémoire d'échange des paramêtres entre fonction/programme, ou encore comme mémoire des variables locales des programmes/fonctions.

Prenons l'exemple suivant :

La fonction GetDlgItemText de l'API Windows exige les paramêtres suivants :

1 - handle of dialog box : l'handle de la boîte de dialogue.
2 - Identifier of control : identificateur du contrôle
3 - Address of buffer for text : adresse du tampon pour le texte
4 - Maximum size of string : format maximum pour la chaîne

Par conséquent ceux-ci ont pu être passés comme ainsi :

```
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 ----------------; call la fonction
```

Facile hein ?? Ceci peut être une des manière les plus simples pour cracker un numéro de série, si vous savez l'adresse du tampon pour le numéro de série, dans ce cas-ci 00406130, poser juste un breakpoint (permettra d'arrêter le programme à cette adresse précise), et vous finirez habituellement vers le haut ou autour du procédé qui produit la vraie publication du serial !

Retour au sommaire

Les instructions de la pile

a) PUSH.

Nous avons tout d'abord l'instruction PUSH qui signifie POUSSER. Cette instruction permet de placer une valeur au sommet de la pile. PUSH doit être accompagné d'une valeur de 16 ou 32 bits

(souvent un registre) ou d'une valeur immédiate.

Exemple 1:

```
PUSH AX
PUSH BX
```

En premier lieu, AX est placé en haut de la pile et ensuite BX est placé au sommet, AX est repoussé au deuxième rang.

Exemple 2:

```
PUSH 1230
PUSH AX
```

Nous plaçons dans la pile, la valeur 1230 (qui aurait pu être aussi un 32 bits) et ensuite AX, ce qui place 1230 au deuxième rang.

Retour au sommaire

b) POP.

Passons maintenant à l'instruction de "récupération" qui se nomme POP (sortir-tirer).Cette instruction demande comme PUSH, une valeur de 16 ou 32 bits (seulement un registre).Elle prend la première valeur de la pile et la place dans le registre qui suit l'instruction.

Exemple 1:

Nous avions PUSH AX et PUSH BX et maintenant nous allons les sortir de la pile en utilisant

```
POP BX
POP AX
```

Nous retrouvons les valeurs initiales de AX et BX mais nous aurions eu aussi la possibilité de faire d'abord POP AX et POP BX, ce qui aurait placé dans AX, la valeur BX (1er dans la pile) et dans BX, la valeur AX (2ème).

Exemple 2:

La première partie était PUSH 1230 et PUSH AX. Nous allons placer dans CX, la valeur de AX (1er de pile) et dans DX, le nombre 1230 (2ème puisque "pilé" après)

```
POP CX
POP DX
```

Dans CX, nous avons la valeur de AX et dans DX, nous avons 1230.

La pile est très utile pour garder la valeur d'un registre que l'on va utiliser afin de le retrouver intact un peu plus loin. Souvent dans des routines, le programmeur doit utiliser tout les registres mais leur nombre est limité (utilisation des registres au maximum = gain de vitesse). Le programmeur va donc utiliser les fonctions de la pile pour pouvoir utiliser les registres et garder leur valeur actuelle pour une utilisation ultérieure. Malheureusement les instructions de la pile ralentissent le programme, la pile étant de toute façon plus lente que les registres qui sont au coeur du CPU.

Retour au sommaire

B - Les interruptions - Instructions

Dans un ordinateur, il y a plusieurs périphériques (imprimante, souris, modem,...). Vous connaissez certainement le BIOS qui se charge chaque fois que vous allumez votre ordinateur.

Il y a différents BIOS qui dépendent des constructeurs mais ils sont généralement tous compatibles aux niveaux des interruptions. Le BIOS possède différentes fonctions, ce sont les interruptions. Ces

interruptions sont divisées en 7 parties principales (il y en a d'autres):

- Fonctions vidéos (int 10h)
- Fonctions disques (int 13h)
- Fonctions de communication sur port série (int 14h)
- Fonctions système - Quasiment inutiles (int 15h)
- Fonctions clavier (int 16h)
- Fonctions imprimante (int 17h)
- Fonctions de gestion de l'horloge (int 1Ah)

Il n'y en a qu'une qui est importante, il s'agit de INT. Elle permet d'appeler une des 7 catégories d'interruptions. Pour pouvoir sélectionner une fonction, il faut configurer les registres avec certaines valeurs que vous trouverez dans la liste d'interruptions (une dernière fois, procurez-vous en une !!!). Souvent, il faut juste changer AX mais parfois, tout les registres sont modifiés. Ensuite on appelle INT accompagné du numéro de catégorie de la fonction désirée. Par exemple, pour rentrer en mode texte 80x25 caractères, il faut écrire ceci:

```
MOV AX,03h
INT 10h
```

Le nombre 03 qui se trouve dans AX, signifie que nous voulons le mode texte 80x25 (13h est le mode 320x200x256 couleurs). L'instruction INT 10h appelle la catégorie 10h (fonctions vidéos).

Et c'est tout. Les interruptions permettent de faire plus facilement certaines actions qui demanderaient une programmation plus compliquée.

Retour au sommaire

4 - Les flags - Les sauts conditionnels - CMP

A - Les flags - Les indicateurs

Les flags, "indicateurs", sont des bits qui donnent certaines informations. Ils sont regroupés dans le registre de flag. Ces flags sont modifiés en fonction de l'exécution des différentes instructions. Celles-ci changent la valeur des flags selon le résultat obtenu. Voici une liste des différents flags et leur utilité. Les bits marqués du 1 ou du 0 ne sont pas utilisés.

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

Nous n'étudierons que les 12 premiers, ce sont les plus importants.

---

a) CF

Retenue

Nous avons tout d'abord CF (Carry Flag). C'est le flag dit de retenue.

Dans les opérations mathématiques, il arrive que le résultat de l'opération soit codé sur un nombre supérieur de bits. Le bit en trop est placé dans CF. De nombreuses instructions modifient aussi CF. Par exemple, les instructions CLC et CMC, la première mettant à zéro CF et la deuxième qui inverse la valeur de CF. STC (set carry) met le flag à 1.

---

b) PF

Parité

Il s'agit du Parity Flag. La valeur de ce flag est 1 si le nombre de bits d'une opérande (paramètre d'une instruction) est pair.

---

c) AF

Retenue auxiliaire

AF est l'auxiliary carry qui ressemble à CF.

---

d) ZF

Zéro

Il s'agit du Zero Flag qui est misà un lorsque un résultat est égal à 0. Souvent utilisé pour les diverses opérations, il est utile pour éviter des problèmes de divisions (je vous rappelle que diviser par zéro est impossible).

---

e) SF

Signe

SF signifie Signe Flag. Simplement, sa valeur passe à 1 si nous avons un résultat signé (négatif ou positif).

---

f) IF

Interruption

IF pour Interrupt Flag, enlève la possibilité au processeur de contrôler les interruptions.

Si IF=0, le processeur ne commande pas et si IF=1 alors c'est le contraire.

L'instruction STI provoque IF=1 et CLI met IF=0.

---

g) DF

Direction

Le flag DF est le Direction Flag. 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).

Deux instructions lui sont associées, il s'agit de CLD et STD.

---

h) OF

Débordement

OF est l'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. Il existe une instruction qui s'occupe de ce Flag, c'est INTO qui déclenche l'exécution du code qui se trouve à l'adresse 0000:0010 (interruption 4).

C'en est tout pour les flags, vous comprendrez leur utilité au fur et à mesure de votre progression mais passons aux instructions conditionnelles qui leur sont directement associées.

Retour au sommaire

B - Les instructions conditionnelles

Nous abordons une partie qui est nécessaire lors de la création d'un programme. Souvent, le programme doit faire une action selon la valeur d'un résultat. Les instructions conditionnelles comme leur nom l'indique, sont des instructions qui font une action selon un résultat. Elles se basent sur les flags pour faire leur choix.

Vous vous souvenez de l'instruction JMP, il s'agissait d'un simple saut vers une autre partie du programme. D'autres instructions comme JMP font des sauts mais selon certains critères, on les appelle des sauts conditionnels. Voici la liste des ces instructions avec la valeur de l'indicateur nécessaire à l'exécution.

JB - JNAE - JC
Below - Not Above or Equal - Carry
CF = 1

JAE - JNB - JNC
Above or Equal - Not Below - Not Carry
CF=0

JE - JZ
Equal - Zero
ZF=1

JNE - JNZ
Not Equal - Not Zero
ZF=0

JO - JNO
Overflow - Not Overflow

OF=1 - OF=0

JP - JPE
Parity - Parity Even
PF=1

JNP - JPO
No Parity - Parity Odd
PF=0

JS - JNS
Signed - Not Signed
SF=1 - SF=0

JA - JNBE
Above - Not Below or Equal
CF=0 et ZF=0

JBE - JNA
Below or Equal - Not Above
CF=1 ou ZF=1

JG - JNLE
Greater - Not Less or Equal
ZF=0 et SF=OF

JGE - JNL
Greater or Equal - Not Less
SF=OF

JL - JNGE
Less - Not Greater or Equal
SF (signé)=OF

JLE - JNG
Less or Equal - Not Greater
ZF=1 ou SF (signé)=OF

Pour tester tout cela, nous avons besoin d'une instruction, c'est CMP qui nous aidera à le faire.

Retour au sommaire

C - L'instruction CMP

Cette instruction va servir à tester différentes valeurs et modifier les flags en fonction du résultat. CMP est un SUB qui ne change ni la source ni la destination, seulement les flags. Un CMP BX,CX

sera comme un SUB BX,CX à l'exception près que BX ne sera pas modifié.

Si BX=CX alors BX-CX=0 donc le flag ZF sera égal à 1. Si nous voulons faire un saut avec "égal à" (JNE ou JZ qui demande ZF=1), nous avons ZF=1 et comme JZ demande que ZF=1 pour faire le saut, nous avons donc le saut. Souvenez-vous simplement que la plupart du temps, il s'agit de comparaisons du genre :

effectue le saut :

plus grand que (nombres non-signés) ---> JA

plus petit que (nombres non-signés) ---> JB

plus grand que (nombres signés) -------> JG

plus petit que (nombres signés) -------> JL

égal à (signé et non-signé) ----------> JE ou parfois JZ

il suffit de rajouter un 'n' après le 'j' pour avoir la même instruction mais exprimée de facon négative

ne saute pas si :

plus grand que (nombres non-signés) ---> JNA (jump if _not above_)

...et ainsi de suite.

Retour au sommaire

5 - Instructions mathématiques

A - Les instructions mathématiques

a) MULTIPLICATION : MUL / IMUL

Je vais continuer avec d'autres instructions pour compléter et vous permettre d'effectuer d'autres opérations. Pour l'instant, le cours est bien théorique mais attendez encore un peu...

Bon, commençons par la multiplication, très utile. L'assembleur utilise l'instruction MUL pour les opérations non signées et IMUL pour les opérations signées. MUL nécessite une opérande (un paramètre). Cette opérande s'appelle la source. Le nombre qui doit être multiplié se trouve OBLIGATOIREMENT dans AX. La source est un registre (BX,CX). Si vous avez AX=2 et BX=5 et que vous faîtes MUL BX, vous obtiendrez AX=10 et BX=5.

Pour IMUL, c'est pareil, les assembleurs acceptent de mettre des nombres négatifs dans un registre (MOV AX,-10). Pour ce qui est des nombres négatifs, sur 16 bits par exemple, -10 sera comme 65525 (65535-10). Le processeur soustrait à la valeur limite du registre (ici 16 bits= 65535), la valeur absolue du nombre négatif.

Donc pour 16 bits, les nombres signés iront de -32768 à 32767. IMUL s'utilise comme MUL sauf qu'il faut indiquer que l'on travaille avec des nombres signés ou non-signés. C'est pourquoi, il faut toujours effacer DX (mettre DX à 0) lors d'une multiplication non-signée.

Dans le cas d'un IMUL, il faut utiliser l'instruction CWD (convert word to doubleword), qui 'étend' le signe de AX dans DX. Omettre ces instructions peut conduire à des résultats erronés. De même, multiplier par DX donne de drôles de résultats.

Retour au sommaire

b) DIVISION : DIV / IDIV

Comme pour MUL et IMUL, nous avons DIV (nombres non signés) et IDIV (valeurs signées). DIV divise la valeur de AX par le source. Comme pour IMUL et MUL, il faut que DX soit correctement paramétré.

Donc si nous avons AX=12 et BX=5, ensuite IDIV BX, nous aurons AX=2 et DX=2 (DX est le reste).

N'oublier pas qu'on ne peut pas diviser par 0 !!! Si vous passez outre, le programme retourne au DOS et affiche 'Div by 0'.

Retour au sommaire

c) SHR et SHL

SHR permet de diviser plus rapidement les registres. SHR demande deux opérandes: le nombre à diviser et le diviseur. SHR divise par une puissance de deux.

Pour diviser par deux AX,BX,CX ou DX (et les autres registres aussi), on met SHR AX,1 (1 parce que 2^1=2), si on veut diviser BX par 4, on fait SHR BX,2 (2 car 2^2=4), SHR CX,3 (2^3=8) divise CX par 8. Compris ? C'est très rapide, pour diviser AX par 256, on aura SHR AX,8. SHL, c'est pareil sauf qu'on multiplie : SHL AX,4 (2^4=16) multiplie AX par 16. On peut combiner aussi des SHL pour multiplier par une valeur qui n'est pas une puissance de 2.

Si nous voulons multiplier AX par 320, on remarque que X*320 = X*256+X*64, or 64 et 256 sont des puissances de 2, donc nous allons multiplier par 256 et multiplier par 64, et additionner les 2 résultats :

```
MOV AX,12 ;on veut multiplier 12 (=X)
MOV BX,AX ;on copie dans BX (AX=X ; BX=X)
SHL AX,8 ;on multiplie par 256 (AX=X*256)
SHL BX,6 ;on multiplie par 64 (BX=X*64)
ADD AX,BX ;on additionne dans AX (AX=X*256 + BX) -> (AX=X*256+X*64=X*320)
```

et voila. cependant dans certains cas, il est préférable d'utiliser MUL car une décomposition en puissance de 2 prendrait plus de temps...

Retour au sommaire

d) NEG

NEG demande une opérande : la destination. Il change le signe de la destination. (en inversant tous les bits et ensuite en ajoutant 1)

Par exemple si AX=12, alors NEG AX donnera AX=-12. Si CL=-3 alors NEG CL changera la valeur de CL en 3. Simple non ?

Retour au sommaire

B - Les nombres à virgules

Le problème en assembleur, est que nous ne pouvons utiliser les nombres à virgules directement dans les registres (sauf avec un coprocesseur mais il demande une programmation différente basée sur une pile). Nous utiliserons les nombres à virgule fixe. Il s'agit simplement d'effectuer des calculs avec des valeurs assez grandes qui seront ensuite, redivisée pour retrouver un résultat dans un

certain intervalle.

Nous voudrions avoir 0.5 dans AX, mais AX n'accepte pas de nombre à virgule, nous allons simplement multiplier AX par 256, or 256*0.5=128, parfait, c'est un nombre entier :

```
MOV AX,128
```

maintenant, nous allons multiplier 300 par 0.5 (donc par 128)

```
MOV CX,300
XOR DX,DX ;comme MOV DX,0
MUL CX
```

dans AX, nous avons (300*128), mais ce nombre est 256x trop grand ! il suffit de le rediviser (qui est en fait un simple décalage peu coûteux en temps de calcul) :

```
SHR AX,8
```

Nous avons dans AX = (300*128)/256 = 150. Et voilà ! nous avons effectué une MUL avec un nombre à virgule fixe mais seulement en utilisant des entiers. Cependant, il convient d'être prudent avec les nombres signés et les grands nombres, on arrive aisément à des dépassements. Sachez aussi que plus vous utilisez un SHL ou SHR avec une opérande grande, plus le résultat sera précis (plus de décimales disponibles).

On aurait pu fort bien faire :

```
MOV AX,64 ;(0.5*128)
MOV CX,3200
XOR DX,DX
MUL CX
SHR AX,7 ;AX/128
```

Nous avons simplement une virgule fixe '128x plus grande'.

Retour au sommaire

C - Les nombres négatifs

Dans notre système numérique nous utilisons le signe "-" pour préciser qu'un nombre est négatif. Mais lorsqu'on veut représenter un nombre négatif en base binaire, on ne dispose pas du signe "-". On ne peut utiliser, en effet, que les deux états possibles du bit : 0 ou 1.

La convention utilisée sur PC est celle du complément à 2 d'un nombre. Ainsi, si on veut représenter -3 sur octet, il faudra :

- calculer la représentation binaire de 3
- inverser les 8 bits de l'octet
- ajouter 1 au résultat obtenu

La représentation binaire de 3 sur un octet est la suivante :

00000011

On inverse à présent tous les bits de l'octet :

11111100

Et on ajoute à présent 1 à ce résultat :

11111101

Cet octet vaut donc :

binaire	1111	1101
décimal	15	13
hexadécimal	F	D

De cette manière, dès que le bit le plus significatif (bit de poids le plus fort; c'est le bit 7 sur un octet) vaut 1, c'est que le nombre est négatif.

Ci-joints deux exemples pour mieux comprendre:

- Représentation, sous forme hexadécimale, un **octet** contenant le nombre décimal -19.

représentation binaire de 19	0	0	0	1	0	0	1	1
inversion de tous les bits	1	1	1	0	1	1	0	0
addition d'une unité	1	1	1	0	1	1	0	1

Il faut décomposer le résultat obtenu en demi-octet :

binaire	1110	1101
décimal	14	13
hexadécimal	E	D

Le nombre -19, codé sur un octet, vaut EDh.

- Représentation, sous forme hexadécimale, un **mot** contenant le nombre décimal -19.

Lorsqu'il faut convertir un mot en double mot, ou un octet en mot, 2 cas peuvent se reproduire : soit le nombre est négatif (bit de poids fort mis à 1), soit il est positif (bit de poids fort mis à 0). Si le nombre est positif les nouveaux bits seront mis à 0; s'il est négatif ils seront mis à 1. Dès lors, il suffit de tester l'état du bit de poids fort.

La valeur hexadécimale, codée sur un octet, de -19 est EDh; le bit de poids fort est allumé et, donc, tous les nouveaux bits seront mis à 1.

binaire	1111	1111	1110	1101
décimal	15	15	14	13
hexadécimal	F	F	E	D

Le nombre décimal -19, codé sur un mot, vaut FFEDh.

Retour au sommaire

D - Les instructions logiques

Les instructions logiques manipulent les bits individuellement. Elles font intervenir des opérandes et des emplacements mémoire de 8 ou 16 bits mais les opérations sur ces opérandes se font **bit par bit**.

a) AND

L'instruction NAD effectue un ET LOGIQUE sur les 2 opérandes spécifiées, le résultat est placé dans la première opérande. Rappelons qu'un ET LOGIQUE donne le résultat 1 si les 2 opérandes sont 1 et donne 0 dans les autres cas.

Les combinaisons possibles sont donc :

0 AND 0 = 0

0 AND 1 = 0
0 AND 0 = 0
1 AND 1 = 1

Exemple :

```
MOV AX, 29
MOV BX, 7
AND AX, BX
```

```
  0001 1101 (AX=29)
AND 0000 0111 (BX=7)
    ------------
    0000 0101 (AX=5)
```

b) OR

Cette instruction effectue un OU LOGIQUE INCLUSIF sur les opérandes spécifiées, le résultat est placé dans la première opérande. Un OU LOGIQUE INCLUSIF donne un résultat 0 si les deux opérandes sont 0 et donne 1 dans les autres cas.

Les combinaisons possibles sont donc :

0 OR 0 = 0
0 OR 1 = 1
0 OR 0 = 1
1 OR 1 = 1

Exemple :

```
MOV AX, 0104h
```

On souhaite mettre à 1 les bits 2 et 4 sans modifier les autres bits :

```
OR AX, 0014h
```

```
    00000001 00000100 (AX)
OR  00000000 00010100 (=0014h)
    ----------------------
    00000001 00010100
```

c) XOR

Cette instruction effectue un OU LOGIQUE EXCLUSIF sur les opérandes spécifiées, le résultat est placé dans la première opérande. Un OU LOGIQUE EXCLUSIF donne un résultat 1 si les deux opérandes sont différentes et donne 0 dans les autres cas.

Les combinaisons possibles sont donc :

0 XOR 0 = 0
0 XOR 1 = 1
0 XOR 0 = 1
1 XOR 1 = 0

Exemple :

```
XOR AX, AX  (remet à 0 le registre AX)
XOR BH, BH  (remet à 0 le registre BH)
```

XOR AH, 6 (crée un masque 00000110 (=6 en décimal) qui sert à complémenter les bits 1 et 2 de AH.)

d) NOT

Cette instruction effectue un NON LOGIQUE sur l'opérande spécifiée ; le résultat est placé dans l'opérande. Un NON LOGIQUE inverse (complémente) la valeur d'un bit, les 2 combinaisons possibles sont donc :

NOT 0 = 1
NOT 1 = 0

Exemple :

NOT AX (inverse tous les bits de AX)
NOT BH (inverse tous les bits de BH)

e) TEST

Cette instruction teste la valeur d'un ou plusieurs bits en effectuant un ET LOGIQUE sur les opérandes. Les opérandes ne sont pas modifiées par l'opération : ce sont les indicateurs qui donnent le résultat du TEST.(cf les instructions conditionnelles)

Exemple :

TEST AX,1 (effectue un ET LOGIQUE avec AX : si le bit 1 de AX est égale à 1, l'indicateur ZF sera activé.)

Retour au sommaire

6 - La mémoire et ses instructions

Nous allons aborder sa mise en place et voir d'une façon plus précise, la manière de l'utiliser.

Il existe pour cela plusieurs instructions. Nous avons vu que "MOV SI, offset MESSAGE" plaçait dans SI, la valeur de l'offset de MESSAGE. Pour une donnée dans un autre segment, il faut aussi utiliser le "MOV AX,seg MESSAGE" et ensuite "MOV DS,AX". En effet, les registres ES, DS, FS et GS n'acceptent pas de valeur numérique directe mais seulement des registres.

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.

Privilégiez l'emploi de LEA plutôt que du MOV. Cela rend le code plus clair et plus compact.

Pour pouvoir déplacer des blocs entiers de mémoire, nous n'allons pas lire chaque octet et le placer dans l'autre bloc mais nous emploierons les instructions :

MOVSB
MOVSW
MOVSD

La première pour déplacer une (B)ytes, la deuxième un (W)ord et la dernière pour déplacer un (D)words qui font 32 bits.

Ces instructions demandent que la source (DS:SI) et l'arrivée (ES:DI) soient configurées.

Il ne faut pas diriger les données vers n'importe quel emplacement de la mémoire. Vous planteriez à coup sur le programme. Nous verrons plus loin, comment allouer des blocs de mémoire.

Par exemple dans DS:SI, nous pouvons avoir les données d'une image et dans ES:DI, l'adresse de la mémoire vidéo (A000h:0000). Si nous voulons copier 10000 bytes, nous emploierons directement le MOVSD (si nous travaillons avec un 386 ou plus) ou avec MOVSW.

Pour plusieurs MOVSx, il est inutile de faire une boucle. L'instruction REP est là pour cela.

Il faut utiliser conjointement avec REP, le registre CX qui contient le nombre d'itérations (le nombre de fois que l'on répete le MOVSx).

Pour les 10000 bytes, nous avons :

```
MOV CX,10000
REP MOVSB
```

Si nous utilisons les Words (plus rapide), c'est :

```
MOV CX,5000 (un word=2 bytes)
REP MOVSW
```

Finalement, avec le 32 bits, c'est :

```
MOV ECX,2500 (un dword=2 words=4 bytes) - utilisation d'un registre étendu
REP MOVSD
```

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

Maintenant, si nous voulons garder des données ou les placer dans un emplacement de la mémoire ?

Nous utiliserons les instructions :

```
STOSB
STOSW
STOSD
```

Comme pour les MOVSx, cette instruction stocke un byte, un word et un dword.

A la différence des déplacements, les STOSx 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. Ces instructions sont utiles pour effacer l'écran par exemple ou remplir un bloc mémoire avec la même valeur.

```
MOV CX,1000
MOV AX,0
REP STOSB
```

Dans cette exemple, CX équivaut à 1000, pour 1000 répétitions. Ensuite, le MOV AX,0 qui est la valeur qui sera placée dans le bloc mémoire. Pour finir, le REP STOSB qui dit que nous plaçons 10000 bytes de valeur 0 à l'emplacement de ES:DI. En fait, ici nous placons AL, on ne tient pas compte de AH dans un STOSB.

Pour réserver de la mémoire, il faut employer les interruptions, ici nous restons dans la mémoire conventionnelle, c'est à dire que l'on ne dépasse pas les 640ko attribués par le DOS. Si l'on veut plus de mémoire, il faut utiliser les modes dits protégés ou flat. La programmation de ces modes est différentes et il faut avoir quelques notions sur le DOS et la gestion de la mémoire. Si vous voulez dès maintenant en savoir plus, lisez des documentations comme celles d'Intel.

Pour la mémoire conventionnelle, on doit d'abord libérer l'emplacement occupé par notre programme :

```
MOV BX,taille ; la taille=(taille du prog en octet / 16)+1
```

```
MOV AH,04ah
INT 21h
```

Ici, taille=4096 correspond 64ko. Ajustez donc la taille en fonction de votre programme.

J'ai ajouté 1 à la taille pour éviter des erreurs (normalement pas nécessaire, on ne sait jamais ;).

Ensuite, il faut réserver le bloc de mémoire:

```
MOV AH,48h
MOV BX,mem_voulue ; (mem_voulue=(mémoire_désirée/16)+1)
INT 21h
JC NOT_ENOUGH_MEM
```

La mémoire voulue se calcule comme pour la libération de la mémoire. Le JC NOT_ENOUGH_MEMORY est utile pour détecter les erreurs. Si la mémoire n'est pas disponible, votre programme ira au label NOT_.... et fera ce que vous voulez pour indiquer l'erreur (affichage d'un texte, retour au DOS). IMPORTANT, il faut conserver la valeur de AX après avoir exécuté le INT 21h. AX contient le segment du bloc alloué et il faut garder cette valeur pour pouvoir la réutiliser lors du désallouage.

Finalement, il faut désallouer les blocs de mémoire APRES les avoir utilisé (sinon quel intérêt ?).

```
MOV BX,4096 ; (taille du bloc)
MOV AX,segment_bloc ; (le segment du bloc, nous l'avons précieusement gardé)
MOV ES,AX
MOV AH,49h
INT 21h
```

Le fait de ne pas désallouer de la mémoire va vous en enlever pour des programmes qui seraient exécutés par la suite.

Retour au sommaire

7 - LES INSTRUCTIONS ASSEMBLEUR

Vous pourrez constater qu'à la section J (consacrée aux sauts : jump), j'y ai introduit des codes hexadécimaux. Ceux-ci sont la représentation des commandes assembleur en hexadécimal, ils nous permettrons de patcher nos programmes.

Pour accéder aux instructions plus rapidement, cliquez sur la lettre désirée :

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A :	
AAA	ajustement ascii de <AL> après une addition. Format BCD
AAD	ajustement ascii de <AX> avant une division BCD
AAM	ajustement ascii de <AL> après une multiplication. Format BCD
AAS	ajustement ascii de <AL> après une soustraction.
ADC	addition avec retenue <CF>
ADD	addition sans retenue
AND	opération logique ET
ARPL	ajuste le niveau du privilège (mode protégée)
Retour	
B :	

BOUND	teste un nombre par rapport à des limites
BSF	cherche 1 bit de droite à gauche
BSR	cherche 1 bit de gauche à droite
BSWAP	conversion d'un registre en un format INTEL
BT	charge 1 bit en CF
BTC	charge 1 bit en CF et complément
BTR	charge 1 bit en CF puis le met à 0 dans la source
BTS	charge 1 bit en CF puis le met à 1 dans la source
Retour	
C :	
CALL	appel de sous programme
CBW	convertit l'octet signé de <AL> en un mot dans <AX>
CDQ	convertit le mot signé de EAX en un quadruple mot dans <EDX:EAX>
CLC	mets l'indicateur à 0 dans CF
CLD	mets l'indicateur à 0 dans DF
CLI	mets l'indicateur à 0 dans IF
CLTS	mets à 0 l'indicateur de changement de contexte Task-Switch-Flag mode protégée.
CMC	complémente l'indicateur CF
CMP	comparaison logique
CMPS[b][w]	compare des chaînes de caractères
CMPSD	compare des chaînes de caractère 80386()
CMPXCHG	compare l'accumulateur avec AL, AX, ou EAX
CWD	convertit le contenu signe de AX en un double mots DX:AX
CWDE	" " " " " "" " "" " " " " " " " "" " " " " " " " dans EAX
Retour	
D :	
DAA	ajustement décimal de AL après une addition BCD
DAS	ajustement décimal de AL après une soustraction BCD
DEC	décrementation
DIV	division non signée
Retour	
E :	
ENTER	construit un cadre de pile pour une procédure de haut niveau.
ESC	accès pour le coprocesseur
Retour	
H :	
HLT	arrêt du processeur en attente d'un événement externe
Retour	
I :	
IBTS	insère une chaîne de bits (mode protégée)
IDIV	division signée

IMUL	multiplication signée
IN	lit un octet ou mot sur un port périphérique
INC	incrémentation
INS[b][w]	lit une chaîne sur un port
INT	interruption logiciel
INTO	active l'interruption 4 si l'indicateur OF est armé
INBD	efface le contenu de la mem cache du 486
INVLPG	exploitation du 486 en mode virtuel. multiprocesseur 8088
IRET	retour d'interruption
IRETD	retour d'interruption depuis un segment de 32bits

Retour

J :

JA	0F87	branchement si supérieur
JAE	0F83	branchement si supérieur ou égal
JB	0F82	branchement si inférieur
JBE	0F86	branchement si inférieur ou égal
JC	0F82	branchement si CF est à 1
JNC	0F83	branchement si CF est à 0
JCXZ	E3	branchement si CX ECX est à 0
JECXZ	E3	branchement si le registre ECX est à 0
JE	74	branchement en cas d'égalité (=JZ)
JG	0F8F	branchement si arithmétiquement supérieur
JGE	0F8D	branchement si arithmétiquement supérieur ou égal
JL	0F8C	branchement si arithmétiquement inférieur
JLE	0F8D	branchement si arithmétiquement inférieur ou égal
JMP	EB	branchement à l'adresse indiquée
JNA	0F86	branchement si non supérieur
JNAE	0F82	branchement si non supérieur ou égal
JNB	0F83	branchement si non inférieur
JNBE	0F87	branchement si non inférieur ou égal
JNE	75	branchement en cas de non égalité (=JNZ)
JNG	0F8E	branchement si arithmétiquement non supérieur
JNGE	0F8C	branchement si arithmétiquement non supérieur ou égal
JNL	0F8D	branchement si non inférieur arithmétiquement
JNLE	0F8E	branchement si arithmétiquement non inférieur ou égale
JNO	0F81	branchement si l'indicateur OF est à 0
JNP	0F8B	branchement si parité impaire (indicateur PF à 0)
JNS	0F89	branchement si positif
JNZ	0F85	branchement si diffèrent (=JNE)
JO	0F80	branchement si OF est à 1

JP	0F8A	branchement si parité pair indicateur PF est à 1
JPE	0F8A	" "
JPO	0F8B	branchement si parité impair indicateur PF est à 0
JS	0F88	branchement si négatif
JZ	0F84	branchement en cas égalité (=JE)
Retour		

L :	
LAHF	charge en AH la partie basse du registre des indicateurs
LAR	charge le droit accès (mode protégée)
LDS	charge une adresse physique en DS: registre
LEA	charge une adresse effective
LEAVE	libère le cadre de pile, installer par entrée
LES	charge une adresse physique en ES: registre
LFS	charge une adresse physique en FS: registre
LGDT	charge le registre de la table des descripteurs globaux (mode protégée)
LGS	charge une adresse physique en GS: registre
LIDT	charge le registre de la table des descripteurs d'interruption (MODE P)
LLDT	charge le registre de la table des descripteurs locaux
LMSW	charge le mot d'état de la machine (mode protégée)
LOCK	verrouille le bus
LODS[b][w]	charge AL/AX avec le contenu de DS:SI
LODSD	charge EAX avec le contenu de DS:SI
LOOP	branchement tant que CX #0
LOOPE	branchement tant que CX #0et ZF =1
LOOPNZ	branchement tant que CX =0
LOOPZ	branchement tant que CX =1
LSL	charge une limite de segment (mode protégée)
LSS	charge une adresse physique en SS: registre
LTR	charge le registre de tache (mode protégée)
Retour	

M :	
MOV	transfère une donnée
MOVS	transfère une chaîne de caractère octet par octet ou mot pour mot de DS:SI en ES:DI
MOVSD	transfère une chaîne de caractères double mot par double mot
MOVSX	transfert avec extension de signe
MOVZX	transfert avec une extension de 0
MUL	multiplication non signée
Retour	

N :	
NEG	négation par complément de 2
NOP	pas d'opération

NOT	opération logique NON complément à 1
Retour	
O :	
OR	opération logique OU inclusif
OUT	transmets un octet ou mot à un periph
OUTS[b][w]	transmets une chaîne à un port
OUTSD	transmets un double mot à un port
Retour	
P :	
POP	dépile un mot
POPA	dépile les registres
POPAD	dépile tous les registres 32 bits
POPF	dépile un mot et le transfère vers le registre des indicateurs
POPFD	dépile un DOUBLE MOT et le transfère vers le registre des indicateurs sur 32bits
PUSH	empile une valeur
PUSHA	empile tous les registres
PUSHAD	empile tous les registres 32 bits
PUSHF	empile le registre des indicateurs
PUSHFD	empile le registre des indicateurs à 32bits
Retour	
R :	
RCL	rotation à gauche à travers CF
RCR	rotation à droite à travers CF
REP[z][nez]	préfixes de répétition
REP[e][ne]	pour traiter les chaînes de caractère en association avec CX et les indicateurs
RET[n][f]	retour de sous programme
ROL	rotation à gauche
ROR	rotation à droite
Retour	
S :	
SAHF	copie AH dans la partie basse du registre des indicateurs.
SAL	décalage à gauche avec introduction de 0
SAR	décalage à droite avec signe
SBB	soustraction non signée avec prise en compte de CF
SCAS[b][w]	compare une chaîne octet par octet ou mot par mot avec le contenu de AL/AX
SCASD	compare une chaîne double mot par double mot avec le contenu EAX
SETA	initialisation à 1 si CF et ZF sont à 0, sinon initialisation à 0
SETAE	initialisation à 1 si CF est à 0, sinon init. à 0
SETB	initialisation à 1 si CF est à 1, sinon init. à 0
SETBE	initialisation à 1 si CF ou ZF est à 1, sinon initialisation à 0
SETE	initialisation à 1 si ZF est à 1, sinon init. à 0

SETG	initialisation à 1 si ZF est à 0 et SF=OF, sinon init. à 0
SETGE	initialisation à 1 si SF=OF, sinon init. à 0
SETL	initialisation à 1 si SF#OF, sinon init. à 0
SETLE	initialisation à 1 si ZF est à 1 et SF#OF, sinon init à 0
SETNA	initialisation à 1 si CF ou ZF est à 1, init à 0
SETNAE	initialisation à 1 si CF est à 1, sinon init. à 0
SETNB	initialisation à 1 si CF est à 0, sinon init à 0
SETNBE	initialisation à 1 si CF et ZF sont à 0, sinon, init. à 0
SETNE	initialisation à 1 si ZF est à 0, sinon init à 0
SETNG	initialisation à 1 si ZF est à 1 ou SF#OF, sinon init. à 0
SETNGE	initialisation à 1 si SF#OF, sinon init. à 0
SETNL	initialisation à 1 si SF et OF sont égaux, sinon init. à 0
SETNLE	initialisation à 1 si ZF est à 0 et SF=OF, sinon init à 0
SETNO	initialisation à 1 si OF est à 0, sinon init à 0
SETNP	initialisation à 1 si PF est à 0, sinon init à 0
SETNS	initialisation à 1 si SF est à 0, sinon init à 0
SETNZ	initialisation à 1 si ZF est à 0, sinon init à 0
SETO	initialisation à 1 si OF est à 1, sinon init à 0
SETP	initialisation à 1 si PF est à 1, sinon init à 0
SETPE	" " " " "" " " " " " " " " " " " " "" "" " "" "
SETPO	initialisation à 1 si PF est à 0, sinon init à 0
SETS	initialisation à 1 si SF est à 1, sinon init à 0
SETZ	initialisation à 1 si ZF est à 1, sinon init à 0
SGDT	sauvegarde le registre de la table des descripteurs globaux(mode prot.)
SHL	voir SAL
SHLD	décalage double à gauche
SHR	décalage à droite avec introduction de 0
SHRD	décalage double à droite
SIDT	sauvegarde le registre de la table des interruptions. (mode protégée)
SLDT	sauvegarde le registre de la table des descripteurs locaux (mode P)
SMSW	sauvegarde le mot d'état de la machine (mode P)
STC	mets à 1 l'indicateur CF
STD	mets à 1 l'indicateur DF
STI	mets à 1 l'indicateur IF
STIOS[b][w]	transfert octet par octet, mot par mot le contenu de AL en ES:DI
STOSD	transfert double mot par double mot le contenu de EAX en ES:DI
STR	sauvegarde le registre de tache (mode Protégée)
SUB	soustraction non signée
Retour	
T :	

TEST	test si un bit est à 1
Retour	
V :	
VERR	test l'autorisation de lecture d'un segment (mode protégée)
VERW	test l'autorisation d'écriture dans un segment (mode protégée)
Retour	
W :	
WAIT	attends que la ligne BUSY ne soit plus actif
Retour	
X :	
XADD	addition signée
XBINVD	efface le contenu de la mémoire cache du 486
XBTS	prends une chaîne de bits (mode protégée)
XCHG	échange les contenus de 2 registres
XLAT	charge en AL l'octet de la table DS:BX+AL
XOR	opération logique ou exclusive
Retour	

Retour au sommaire

8 - Table ASCII

Comme pour les commandes en assembleur, les caractères imprimables ou non imprimables (retour chariot par exemple) ont une correspondance en hexadécimal.

Caractères non imprimables					**Caractères imprimables**								
Nom	**Ctrlcaract**	**Dec**	**Hex**	**Caract**	**Dec**	**Hex**	**Caract**	**Dec**	**Hex**	**Caract**	**Dec**	**Hex**	**Caract**
null	Ctrl-@	0	00	NUL	32	20	Space	64	40	@	96	60	`
Début d'entête	Ctrl-A	1	01	SOH	33	21	!	65	41	A	97	61	a
Début de texte	Ctrl-B	2	02	STX	34	22	"	66	42	B	98	62	b
Fin de texte	Ctrl-C	3	03	ETX	35	23	#	67	43	C	99	63	c
end of xmit	Ctrl-D	4	04	EOT	36	24	$	68	44	D	100	64	d
enquiry	Ctrl-E	5	05	ENQ	37	25	%	69	45	E	101	65	e
acknowledge	Ctrl-F	6	06	ACK	38	26	&	70	46	F	102	66	f
Cloche	Ctrl-G	7	07	BEL	39	27	'	71	47	G	103	67	g
backspace	Ctrl-H	8	08	BS	40	28	(	72	48	H	104	68	h
horizontal tab	Ctrl-I	9	09	HT	41	29	)	73	49	I	105	69	i
line feed	Ctrl-J	10	0A	LF	42	2A	*	74	4A	J	106	6A	j
vertical tab	Ctrl-K	11	0B	VT	43	2B	+	75	4B	K	107	6B	k
form feed	Ctrl-L	12	0C	FF	44	2C	,	76	4C	L	108	6C	l

carriage feed	Ctrl-M	13	0D	CR	45	2D	-	77	4D	M	109	6D	m
shift out	Ctrl-N	14	0E	SO	46	2E	.	78	4E	N	110	6E	n
shift in	Ctrl-O	15	0F	SI	47	2F	/	79	4F	O	111	6F	o
data line escape	Ctrl-P	16	10	DLE	48	30	0	80	50	P	112	70	p
device control 1	Ctrl-Q	17	11	DC1	49	31	1	81	51	Q	113	71	q
device control 2	Ctrl-R	18	12	DC2	50	32	2	82	52	R	114	72	r
device control 3	Ctrl-S	19	13	DC3	51	33	3	83	53	S	115	73	s
device control 4	Ctrl-T	20	14	DC4	52	34	4	84	54	T	116	74	t
neg acknowledge	Ctrl-U	21	15	NAK	53	35	5	85	55	U	117	75	u
synchronous idel	Ctrl-V	22	16	SYN	54	36	6	86	56	V	118	76	v
end of xmit block	Ctrl-W	23	17	ETB	55	37	7	87	57	W	119	77	w
cancel	Ctrl-X	24	18	CAN	56	38	8	88	58	X	120	78	x
end of medium	Ctrl-Y	25	19	EM	57	39	9	89	59	Y	121	79	y
substitute	Ctrl-Z	26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
escape	Ctrl-[	27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
file separator	Ctrl-\	28	1C	FS	60	3C	<	92	5C	\	124	7C	|
group separator	Ctrl-]	29	1D	GS	61	3D	=	93	5D	]	125	7D	}
record separator	Ctrl-^	30	1E	RS	62	3E	>	94	5E	^	126	7E	