Bible du cracker

Cours d'assembleur par Falcon

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.