ASSEMBLEUR

Sommaire

Présentation	2
Compilation	3
Instructions	4
Adressage	6
Instructions arithmétiques et logiques	7
Affectation	10
Branchements	10
Pile	13
Procédures	14
Segmentation de la mémoire	16
Liste des principales instructions	17

Présentation :

Les prérequis nécessaires

Le langage assembleur est très proche du langage machine (c'est-à-dire le langage qu'utilise l'ordinateur: des informations binaires, soit des 0 et des 1). Il dépend donc d'un processeur. Il est donc nécessaire de connaître un minimum le fonctionnement d'un processeur pour pouvoir aborder cette partie. Un processeur réel a toutefois trop de registres et d'instructions pour pouvoir les étudier en détail. C'est pour cette raison que seuls les registres et les instructions d'un processeur simple seront étudiés

Le processeur en bref...

Un *processeur* est relié à la *mémoire* par l'intermédiaire d'une liaison appelée *bus*. Les données dont le processeur a besoin sont stockées dans ce que l'on appelle des registres (ils sont notés AX, BX, CX, DX, ...). Chacun a sa propre utilité:

Nom du registre	Taille
Accumulateur AX	16 bits
Registre auxiliaire BX	16 bits
Registre auxiliaire CX	16 bits
Pointeur d'instruction IP	16 bits
Registre segment CS	16 bits
Registre segment DS	16 bits
Registre segment SS	16 bits
Pointeur de pile SP	16 bits
Pointeur de pile BP	16 bits

Pourquoi utiliser l'assembleur?

Pour faire exécuter une suite d'instructions au processeur, il faut lui fournir des données binaires (souvent représentées en notation hexadécimale pour plus de lisibilité, mais cela revient au même...). Or, les fonctions en notation hexadécimale sont difficiles à retenir, c'est pourquoi le langage assembleur a été mis au point. Il permet de noter les instructions avec des noms explicites suivis de paramètres.
Voici par exemple à quoi peut ressembler un programme en langage machine:
A1 01 10 03 06 01 12 A3 01 14
Il s'agit de la représentation hexadécimale d'un programme permettant d'additionner les valeurs de deux cases mémoire et de stocker le résultat dans une troisième case. Il est évident que ce type d'écriture est difficilement lisible par nous, humains.
Ainsi, puisque toutes les instructions que le processeur peut effectuer sont associées à une valeur binaire chacune, on utilise une notation symbolique sous forme textuelle qui correspond à chaque fonction, c'est ce que l'on appelle le langage assembleur. Dans l'exemple précédent la séquence A1 01 10 signifie copier le contenu de la mémoire à l'adresse *0110h* dans le registre AX du processeur. Cela se note en langage assembleur:
MOV AX, [0110]
Toutes les instructions ont une notation symbolique associée (fournie par le fabricant du processeur). L'utilisation du langage assembleur consiste donc à écrire sous forme symbolique la succession d'instructions (précédées de leur adresse pour pouvoir repérer les instructions et

passer facilement de l'une à l'autre). Ces instructions sont stockées dans un fichier texte (le fichier *source*) qui, grâce à un programme spécifique (appelé "l'assembleur") sera traduit en langage machine.
Le programme précédent écrit en langage assembleur donnerait:

Adresse de l'instruction	Instruction en langage machine	Instruction en langage assembleur	Commentaires sur l'instruction
0100	A1 01 10	MOV AX, [0110]	Copier le contenu de 0110 dans le registre AX
0103	03 06 01 12	ADD AX, [0112]	Ajouter le contenu de 0112 à AX et mettre le résultat dans AX
0107	A3 01 14	MOV [0114], AX	Stocker AX à l'adresse mémoire 0114

L'écriture en langage assembleur, bien que restant rebutante, est beaucoup plus compréhensible pour un humain, car on a généralement moins de mal à retenir un nom qu'un numéro...

Source & Compilation :

La compilation du programme

Le programme doit être saisi dans un fichier texte non formatté (c'est-à-dire sans caractères en gras, souligné, avec des polices de caractères de différentes tailles, ...) appelé fichier source. En effet, l'assembleur (le programme permettant de faire la traduction du langage assembleur en langage machine) permet uniquement de créer un fichier assemblé à partir du fichier source (il devra comporter l'extension .ASM, en s'appelant par exemple *source.asm* ou n'importe quel autre nom suivi de l'extension .asm).
L'assembleur va fournir un *fichier objet* (dont l'extension est .obj) qui va contenir l'ensemble des instructions traduites en instructions machines. Ce fichier .OBJ ne pourra toutefois pas s'exécuter directement car il faut encore lier les différents fichiers.
Comment ça les différents fichiers?
En effet il est possible de construire un exécutable à partir de plusieurs fichiers sources (à partir d'un certain niveau de programmation il devient intéressant de créer des fichiers contenant des fonctions...). Ainsi, même si vous avez un seul fichier objet il vous faudra utiliser un programme (appelé *éditeur de liens*) qui va vous permettre de créer un fichier exécutable (dont l'extension sera .exe).

A quoi ressemble un fichier source en assembleur?

Comme dans tout programme le fichier source doit être saisi de manière rigoureuse. Chaque définition et chaque instruction doivent ainsi s'écrire sur une nouvelle ligne (pour que l'assembleur puisse les différencier) Le fichier source contient:

- Des définitions de données déclarées par des *directives* (mots spéciaux interprétés par l'assembleur, nous les étudierons plus tard, le but est ici de donner une idée de ce à quoi ressemble un fichier source) Celles-ci sont regroupées dans le segment de données délimité par les directives **SEGMENT** et **ENDS**
- Puis sont placées les ***instructions*** (qui sont en quelques sorte le cœur du programme), la première devant être précédée d'une étiquette, c'est-à-dire un nom qu'on lui donne. Celles-ci sont regroupées dans le *segment d'instructions* délimité par les directives **SEGMENT** et **ENDS**

- Enfin le fichier doit être terminé par la directive **END** suivi du nom de l'étiquette de la première instruction (pour permettre au compilateur de connaître la première instruction à exécuter
- (Les points-virgules marquent le début des commentaires, c'est-à-dire que tous les caractères situés à droite d'un point virgule seront ignorés)

Voici à quoi ressemble un fichier source (fichier .ASM):

```
donnees      SEGMENT; voici le segment de données dont
l'étiquette est donnees
      ;Placez ici les déclarations de données

donnees      ENDS; ici se termine le segment de donnees

ASSUME DS:data, CS: code

instr  SEGMENT; voici le segment d'instructions dont l'etiquette
est instr
      debut:       ;placez ici votre premiere instruction (son
etiquette est nommée debut)
                   ;Placez ici vos instructions

instr  ENDS; fin du segment d'instructions

END debut; fin du programme suivie de l'etiquette de la premiere
instruction
```

La déclaration d'un segment

Comme nous le verrons plus loin, les données sont regroupées dans une zone de la mémoire appelé *segment de données*, tandis que les instructions se situent dans un *segment d'instructions*. Le registre *DS* (Data Segment) contient le segment de données, tandis que le registre CS (Code Segment) contient le segment d'instructions. C'est la directive ***ASSUME*** qui permet d'indiquer à l'assembleur où se situe le segment de données et le segment de code.
Puis il s'agit d'initialiser le segment de données:

```
MOV AX, nom_du_segment_de_donnees
MOV DS, AX
```

Instructions :

A quoi ressemble une instruction?

L'instruction est l'élément clé de l'ordinateur car c'est elle qui permet de spécifier au processeur l'action à effectuer. Toutes les instructions sont stockées en mémoire et un compteur dans le processeur permet de passer de l'une à l'autre.
Une instruction est composée de deux éléments:

- le code opération: action à effectuer par le processeur
- le champ opérande: donnée ou bien adresse de la case mémoire contenant une donnée

La taille d'une instruction, c'est-à-dire le nombre de bits qu'elle occupe en mémoire dépend de l'instruction et de l'opérande, elle est généralement de quelques octets (1 à 4). L'octet est l'unité

qui est utilisée généralement car elle est pratique pour le stockage de certaines données (notamment les caractères).

Les types d'instructions

L'ensemble des instructions est appelé *jeu d'instruction*. On peut les répartir selon plusieurs catégories selon le type d'action que son exécution déclenche.

Instructions d'affectation

Les instructions d'affectations permettent de faire des transferts de données entre les registres et la mémoire, c'est-à-dire:

- soit une écriture (du registre vers la mémoire)
- soit une lecture (de la mémoire vers un registre)
-

Instructions arithmétiques et logiques

Ce type d'instructions porte sur le registre AX (l'accumulateur). Elle permettent d'effectuer une opération entre le registre AX et une donnée puis stocker le résultat dans AX. Ces instructions sont:

- l'addition: AX = AX + donnée
- la soustraction: AX = AX - donnée
- incrémentation: AX = AX + 1
- décrémentation: AX = AX - 1
- décalage à gauche
- décalage à droite

Les branchements conditionnels sont très utiles, notamment pour créer des boucles ou des structures conditionnelles.

Instructions de comparaison

Permet de comparer le registre AX à une donnée. Le résultat de la comparaison est indiquée grâce à ... des indicateurs...
Ces instructions (ainsi que les instructions de branchements) sont nécessaires pour permettre une "interactivité", c'est-à-dire qu'en fonction de certains paramètres le programme va pouvoir avoir divers comportement. Sans ces indicateurs les programmes auraient un compoirtement déterminé et le programme ferait constamment les mêmes actions (comme un film ...).

Instructions de branchement

Ce type d'instruction permet de sauter à une instruction non consécutive à l'instruction en cours. En l'absence de ce type d'instruction le processeur passe automatiquement à l'instruction suivante (c'est-à-dire l'instruction contiguë (en mémoire) à l'instruction en cours). C'est le registre IP qui repère l'instruction suivante à exécuter. Les instructions de branchement permettent donc de modifier la valeur de ce registre et ainsi de choisir la prochaine instruction à exécuter.
On distingue deux sortes de branchements:

- Les branchements conditionnels: en fonction d'une condition à satisfaire le processeur traitera le branchement indiqué ou l'instruction suivante.
    - condition satisfaite: exécution de l'instruction située à l'adresse indiquée
    - condition non satisfaite: exécution de l'instruction suivante
- Les branchements inconditionnels: on affecte une valeur définie au registre IP, c'est-à-dire que l'on ira exécuter l'instruction vers laquelle il pointe après celle en cours quoi qu'il arrive!

Adressage :

On appelle "mode d'adressage" la manière dont la donnée est spécifiée dans une instruction.
Selon le mode d'adressage la taille de l'instruction peut varier de 1 à 4 octets.
Il existe 5 modes d'adressage:

- le mode d'adressage implicite
- le mode d'adressage immédiat
- le mode d'adressage relatif
- le mode d'adressage direct
- le mode d'adressage indirect

Le mode d'adressage implicite

Le mode d'adressage implicite correspond à une instruction ne comportant pas d'opérande. L'instruction est composée du code opération uniquement et sa taille peut varier entre 1 octet et 2 octets selon l'opération.

code opération (1 ou 2 octets)

Ce type d'instruction porte généralement sur des registres. Les opérations d'incrémentation ou de décrémentation d'un registre on un mode d'adressage implicite.

Le mode d'adressage immédiat

On parle de mode d'adressage immédiat lorsque le code opérande contient une donnée. La taille de la donnée peut varier entre 1 et 2 octets.

code opération (1 ou 2 octets)	code opérande (1 ou 2 octets)

Ce type d'instruction met en jeu un registre et une valeur (qu'il s'agisse d'une affectation, une addition, une soustraction ou bien même une comparaison), la taille de l'opérande dépendra donc du type de registre mis en jeu (1 octet pour un registre 8 bits, 2 pour un registre de 16 bits).
Dans le cas de l'instruction *MOV BX, 8* l'opérande 8 sera codée sur 16 bits puisqu'il faut l'affecter à un registre 16 bits (BX).

Le mode d'adressage relatif

Ce type de mode d'adressage met en jeu un champ opérande contenant un entier relatif (sa taille est donc un octet).

code opération (1 octet)	code opérande (1 octet)

On l'utilise pour les opérations de saut, l'entier relatif est appelé *déplacement*, il correspond à la longueur du saut que le processeur doit effectuer dans les instructions.

Le mode d'adressage direct

Le code opérande d'une instruction en mode d'adressage direct, contrairement au mode d'adressage immédiat, contient l'adresse d'une donnée en mémoire (au lieu de contenir la donnée). Une adresse étant codée sur 16 bits, la taille du champ opérande est donc de 2 octets.

code opération (1 ou 2 octets)	code opérande (2 octets)

Il peut s'agir par exemple de l'affectation à un registre d'une donnée contenue dans une case mémoire. Ce mode d'adressage provoque un temps d'exécution de l'instruction plus long car l'accès à la mémoire principale est plus long que l'accès à un registre.

Le mode d'adressage indirect

Le mode d'adressage indirect permet d'accéder à une donnée par l'intermédiaire d'un registre (le registre BX) qui contient son adresse. Son utilité n'est pas apparente à ce stade, mais l'adressage direct est très utile lors de l'utilisation de tableaux (parcours des cases d'un tableau) car il suffit d'incrémenter BX de la taille d'une case pour passer d'une case à une autre...
En adressage directe, on affecte directement au registre accumulateur (AX) l'adresse d'une donnée
MOX AX, [110]
En adressage indirect, on affecte à AX l'adresse contenue dans le registre BX
MOV AX, [BX]
La notion d'adressage indirect sera mieux développée

Opérations arithmétiques & logiques :

Les instructions arithmétiques et logiques sont effectuées par l'unité arithmétique et logique. Il s'agit d'opération directement effectuées sur les bits de la donnée que l'on traite.
Sont comprises dans cette appellations:

- les instructions d'addition
- les instructions de soustraction
- les instructions de décalage
- les instructions de rotation
- les instructions logiques (ET, OU, ...)

Les opérations arithmétiques et logiques modifient l'état des indicateurs.

Instructions d'addition

Les opérations arithmétiques se font de la même façon en binaire qu'en base décimale. C'est-à-dire que lorsque l'on dépasse la valeur maxi au niveau du bit *n* (lorsque l'on dépasse la valeur 1) on a une retenue ou un report au bit *n+1*.
Voyons voir cela sur un exemple:

```
  0 0 0 1
+ 0 0 0 1
  - - - -
```

  0 0 1 0
Sur un exemple un peu plus compliqué:
  0 0 1 0
+ 0 0 1 1
  - - - -
  0 1 0 1

Décalage et rotation

Ces instructions permettent de décaler d'un côté ou de l'autre les bits des registres accumulateurs (AX et BX, et donc AH, AL, BH, BL). Cette opération qui semble inuile a en fait plusieurs applications très intéressantes, dont:

- permettre de lire un à un les bits du registre (car les bits sortant à gauche positionnent l'indicateur de retenue CR)
- permettre une multiplication par 2^n (en effet le fait de décaler un nombre binaire d'un chiffre à gauche le multiplie par 2, ainsi en effectuant cette opération *n* fois on obtient une multiplication par 2^n)
  exemple:
  00010 (2 en décimale)
  00100 (on décale à gauche on obtient 4)
  01000 (on décale à droite on obtient 8)
- permettre une division par 2^n (comme précédemment mais en effectuant une rotation sur la droite)

Une **opération de décalage** déplace chacun des bits d'un nombre binaire sur la gauche (ou la droite), mais ceux-ci sortent, c'est-à-dire qu'ils sont définitivement perdus, lorsqu'ils arrivent au bit de poids fort (ou de poids faible).
exemple:
0001110000
0011100000 (on décale d'un bit à gauche)
0111000000 (on décale d'un bit à gauche)
1110000000 (on décale d'un bit à gauche)
1100000000 (on décale d'un bit à gauche)
1000000000 (on décale d'un bit à gauche)
Une **opération de rotation** agit comme une opération de décalage à la différence près que les bits qui sortent d'un côté rentrent de l'autre...
exemple:
0001110000
0011100000 (on effectue une rotation d'un bit à gauche)
0111000000 (on effectue une rotation d'un bit à gauche)
1110000000 (on effectue une rotation d'un bit à gauche)
1100000001 (on effectue une rotation d'un bit à gauche)
1000000011 (on effectue une rotation d'un bit à gauche)
Les opérations courantes de rotation et de décalage sont les suivantes:

- **RCL *registre, 1*** (*Rotate Carry Left*):
  Effectue une rotation des bits sur la gauche en passant par l'indicateur de retenue CF. Le contenu de CF est introduit à droite, puis le bit de poids fort est copié dans CF.
- **RCR *registre, 1*** (*Rotate Carry Right*):
  Effectue une rotation des bits sur la droite en passant par l'indicateur de retenue CF. Le contenu de CF est introduit à gauche, puis le bit de poids faible est copié dans CF.

- **ROL *registre, 1*** (*Rotate Left*):
  Effectue une rotation des bits sur la gauche. Le bit de poids fort est copié dans CF et réintroduit à droite
- **ROR *registre, 1*** (*Rotate Right*):
  Effectue une rotation des bits sur la droite. Le bit de poids faible est copié dans CF et réintroduit à gauche
- **SHL *registre, 1*** (*Shift Left*):
  Décale les bits du registre indiqué de 1 bit vers la gauche. (Les bits sortants sont transférés dans l'indicateur de retenue CF mais ne sont pas réintroduits à droite)
- **SHR *registre, 1*** (*Shift Right*):
  Décale les bits du registre indiqué de 1 bit vers la droite. (Les bits sortants sont transférés dans l'indicateur de retenue CF mais ne sont pas réintroduits à gauche)

Les instructions **RCL** et **RCR** permettent de faire une lecture bit-à-bit du contenu du registre.
Les instructions **SHL** et **SHR** permettent de faire une multiplication par 2^n sur des entiers naturels (pas sur des entiers relatifs car le bit de poids fort disparait dès la première rotation).

Instructions logiques

Les instructions logiques sont au nombre de trois (ET, OU et OU Exclusif). Elles permettent de faire des opérations bit-à-bit sur des nombres binaires (c'est-à-dire en considérant chacun des bits indépendamment des autres, sans se soucier de la retenue).
Ce type d'instruction ce note de la manière suivante:
INSTRUCTION destination, source
destination désigne le registre ou l'adresse de la case mémoire qui contiendra le résultat de l'opération.
source peut ê aussi bien un registre, une constante ou une adresse.

- L'instruction **AND** (ET) multiplie les bits de même poids deux à deux et stocke le résultat dans le registre de destination. Cette instruction met donc le bit du résultat à 1 si les deux bits de même poids des de la source et de la destination sont tous deux à 1, sinon il le met à zéro.
  exemple:

```
    0 1 0 1
ET  0 1 1 0
    - - - -
    0 1 0 0
```

- L'instruction **OR** (OU) met donc le bit du résultat à 0 si les deux bits de même poids de la source et de la destination sont tous deux à 0, sinon il le met à un.
  exemple:

```
    0 1 0 1
OU  0 1 1 0
    - - - -
    0 1 1 1
```

- L'instruction **XOR** (OU Exclusif) met le bit du résultat à 1 si un des deux bits de même poids de la source et de la destination est égal à 1 (mais pas les deux), dans les autres cas il le met à zéro.
  exemple:

```
 0 1 0 1
 0 1 1 0
 - - - -
```

0 0 1 1

Chacune de ces instructions a de très nombreuses applications, il serait impossible de toutes les énumérer. En voici quelques unes:

- **Masquage**: Il est possible de masquer, c'est-à-dire mettre à zéro tous les bits qui ne nous intéressent pas dans un nombre. Pour cela il faut créer une valeur masque: un nombre dont les bits de poids qui nous intéressent sont à 1, les autres à 0. Il suffit alors de faire un **AND** entre le nombre à masquer et le masque pour ne conserver que les bits auxquels on s'intéresse.
  Par exemple, imaginons que l'on veuille masquer les 4 bits de poids faible (les 4 derniers bits) d'un nombre codé sur 8 bits (par exemple 10110101). Il suffit d'appliquer le masque 11110000, et l'on obtiendra 10110000. On ne conserve bien que les 4 premiers bits, les autres sont mis à zéro...
- **Inversion**: Il est possible d'inverser tous les bits d'un nombre en faisant un **XOR** avec un nombre contenant que des 1.

Instructions d'affectation :

Qu'est-ce qu'une instruction d'affectation

Une instruction d'affectation est une instruction qui modifie le contenu d'un registre ou d'une case mémoire en y affectant une valeur, ou une adresse (d'où son nom...). C'est l'instruction MOV (provenant du mot anglais *move* qui signifie déplacer) qui permet non pas de "déplacer" une donnée mais plutôt copier une donnée d'un endroit à un autre, c'est-à-dire recréer une entité identique à un endroit différent.

Dans quel sens se font les échanges?

L'instruction MOV permet d'effectuer les transferts de données dans les deux sens (i.e. aussi bien du registre vers la mémoire que l'inverse). Ainsi, en notation symbolique on note la destination en premier puis la source, ce qui donne:
MOV destination, source
Voyons voir cela sur un exemple:
MOV AX, [0110] va copier le contenu de l'adresse 110H dans le registre AX
MOV [0110], AX va copier le contenu du registre AX à l'adresse 110H

Sauts branchements :

On appelle saut (ou branchement) en assembleur le fait de passer à une instruction autre que celle qui suit celle en cours en mémoire. En effet, en temps normal (c'est-à-dire sans instruction contraire) le processeur exécute les instructions séquentiellement, il exécute l'instructions située à l'emplacement mémoire suivant. C'est un registre spécial (le registre IP) qui indique l'adresse de l'instruction suivante à exécuter.
Dans certaines conditions il peut être intéressant de "choisir" la prochaine instruction à effectuer. Ce type de condition peut notamment se rencontrer dans les structures conditionnelles (saut si...) ou bien dans les structures de boucle (en effet dans le cas où on désire exécuter un grand nombre de fois une instruction il peut être intéressant d'utiliser une instruction de branchement, qui indique au processeur l'adresse de la prochaine instruction à exécuter au lieu de gaspiller la mémoire en stockant plusieurs fois la même instruction en mémoire).

Lors de l'exécution "normale" d'un programme, le processeur lit l'adresse contenue dans le registre IP, incrémente celui-ci pour qu'il pointe vers l'instruction suivante, puis exécute l'instruction contenue à l'adresse qu'il vient de lire. Lorsqu'il rencontre une instruction de saut (ou branchement), celle-ci va lui faire modifier le contenu du registre IP pour qu'il pointe à l'adresse d'une autre instruction.
On distingue ces instructions de saut en deux catégories suivant que:

- le saut est effectué quoi qu'il arrive (**saut inconditionnel**)
- le saut est effectué ou non selon l'état d'un registre (**saut conditionnel**)

Saut inconditionnel

L'instruction JMP permet d'effectuer un saut inconditionnel, c'est-à-dire que cette instruction va stocker dans le registre IP l'adresse de l'instruction que l'on veut exécuter. L'opérande de cette instruction (le paramètre) est donc l'adresse de l'instruction à laquelle on veut sauter. Une fois l'instruction de branchement exécutée le processeur lit le contenu du registre IP et saute donc directement à l'adresse de l'instruction que l'on vient de définir!
La taille de l'instruction JMP est de 1 bit.
On appelle *déplacement* (en anglais *offset*) le nombre d'octets (car il s'agit d'un nombre entier relatif codé sur 8 bits) qui séparent l'instruction suivante de l'instruction visée. Voyons voir cela sur le programme suivant:

Adresse	**Instruction en assembleur**	**Commentaire**
0100	MOV AX, [120]	copie le contenu de la case mémoire à l'adresse 0120H dans le registre AX
0103	JMP 0100	saute à l'adresse 0100
0104	MOV [120], BX	instruction non exécutée à cause du saut précédent...

La valeur du déplacement est ici de: 0100H - 0104H = -4

Saut conditionnel

Les instructions de saut conditionnel permettent d'effectuer un saut suivant une condition. Si celle-ci est réalisée le processeur saute à l'instruction demandée, dans le cas contraire il ignore cette instruction et passe automatiquement à l'instruction d'après, comme si cette instruction n'existait pas...
Les conditions pour chacune de ces instruction sont fonction de l'état de registres spécifiques appelés indicateurs (en anglais *flag*, ce qui signifie *drapeau*).

Les indicateurs

Les indicateurs sont des registres dont l'état est fixé par l'UAL après certaines opérations. Les indicateurs font partie de ce que l'on appelle le *registre d'état* qui n'est pas directement accessible par les autres instructions, seules des instructions spécifiques permettent de les manipuler.
Voyons voir certains de ces indicateurs:

- **CF (Carry Flag)**: c'est l'indicateur de *retenue*
  Il intervient lorsqu'il y a une retenue après une addition ou une soustraction entre des entiers naturels. Lorsqu'il y a une retenue il est positionné à 1, dans le cas contraire à 0
- **OF (Overflow Flag)**: cet indicateur (indicateur de débordement: *overflow* = *débordement*) intervient lorsqu'il y a un débordement, c'est-à-dire lorsque le nombre de bits sur lesquels lesnombres sont codé n'est pas suffisant et que le résultat d'une opération

n'est pas codable avec le nombre de bit spécifié (il peut par exemple arriver dans ces conditions que la somme de deux nombres positifs donnent un nombre négatif). Dans ce cas l'indicateur OF est positionné à 1

- **SF (Sign Flag)**: c'est l'indicateur de signe. SF donne tout simplement le signe du bit de poids fort. Or, le bit de poids fort donne le signe du nombre (1 si le signe est négatif, 0 s'il est positif). Il simplifie le test du signe d'un entier relatif.
- **ZF (Zero Flag)**: L'indicateur de zéro permet de savoir si le résultat de la dernière opération était nul. En effet, dans ce cas, l'indicateur ZF est positionné à 1 (0 dans le cas contraire). Il permet notamment de déterminer si deux valeurs sont ègales, en effectuant leur soustraction, puis en observant l'état de l'indicateur de zéro.

Voyons voir les états de ces indicateurs sur un exemple:

```
  0 1 1 0
+ 0 1 0 1
  - - - -
  1 0 1 1
```

- OF=1 (la somme de deux nombres positifs est négative
- 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

Forcer l'état de certains indicateurs

Les instructions STC et CLC permettent de positionner "manuellement" l'indicateur de retenue (CF).
L'instruction STC (**S**e**T** **C**arry) positionne l'indicateur CF à 1.
L'instruction CLC (**CL**ear **C**arry) positionne CF à 0.

Instruction de comparaison

L'instruction CMP permet de tester la valeur d'un registre (AX) avec une autre valeur. Sa seule action est de positionner l'indicateur ZF à 1 en cas d'égalité, ou plus exactement lorsque la soustraction des deux valeurs donne un résultat nul. En ce sens il effectue la même chose que SUB à la seule différence près qu'il ne modifie pas les opérandes.
Par exemple, l'instruction:

```
CMP AX, 2
```

positionne à 1 l'indicateur ZF si la valeur contenue dans le registre AX vaut 2, dans le cas contraire il le met à zéro...

Les sauts conditionnels

Les branchements conditionnels (ou sauts conditionnels) permettent au processeur de traiter l'instruction située à un emplacement mémoire indiqué si une certaine condition est vérifiée. Dans le cas contraire (condition non réalisée), le processeur ignorera cette instruction, il traitera donc l'instruction suivante.
La (ou les) condition(s) à satisfaire dépend(ent) de l'état d'indicateurs. Ainsi les branchements conditionnels doivent généralement être placée après une opération qui va modifier l'état d'un ou plusieurs indicateurs (une instruction CMP ou autre).
Selon l'intitulé de l'instruction, les conditions à satisfaire sont différentes:

- **JA** (*Jump if above*, ce qui signifie *saute si au-delà)*
  effectue un saut si **ZF=0 et CF=0**
- **JB** (*Jump if Below*, ce qui signifie *saute si en-deça*)
  effectue un saut si **CF=1**
- **JBE** (*Jump if Below or Equal*, ce qui signifie *saute si en-deça ou égal)*
  effectue un saut si **ZF=1 ou CF=1**
- **JE** (*Jump if Equal*, ce qui signifie *saute si égalité)*
  effectue un saut si **ZF=1**
- **JG** (*Jump if Greater*, ce qui signifie *saute si supérieur)*
  effectue un saut si **ZF=0 et SF=OF**
- **JLE** (*Jump if Lower or Equal*, ce qui signifie *saute si inférieur ou égal)*
  effectue un saut si **ZF=1 ou SF différent de OF**
- **JNE** (*Jump if Not Equal*, ce qui signifie *saute si non-égalité)*
  effectue un saut si **ZF=0**

Notions de pile :

Utilité d'une pile

Une pile est une zone de mémoire dans laquelle on peut stocker temporairement des registres. Il s'agit d'un moyen d'accéder à des données en les empilant, telle une pile de livre, puis en les dépilant pour les utiliser. Ainsi il est nécessaire de dépiler les valeurs stocker au sommet (les dernières à avoir été stockées) pour pouvoir accéder aux valeurs situées à la base de la pile.
En réalité il s'agit d'une zone de mémoire et d'un pointeur qui permet de repérer le sommet de la pile.
La pile est de type LIFO (Last In First Out), c'est-à-dire que la première valeur empilée sera la dernière sortie (Si vous empilez des livres, il vous faudra les dépiler en commençant par enlever les livres du dessus. Le premier livre empilé sera donc le dernier sorti!).

Les instructions PUSH et POP

Les instructions PUSH et POP sont les instructions qui servent à empiler et dépiler les données.
PUSH *registre* met le contenu du registre dans la pile (empilement)
POP *registre* récupère le contenu de la pile et le stocke dans le registre (dépilage)
Ainsi, l'instruction *PUSH BX* empile le contenu du registre BX, et l'instruction *POP AX* récupère le contenu du sommet de la pile et le transfère dans AX.

Utilisation de la pile sur un exemple

Dans l'exemple suivant, que l'on imaginera au milieu d'un programme, on stocke les valeurs contenues dans AX et BX pour pouvoir utiliser ces deux registres, puis une fois l'opération accomplie on remet les valeurs qu'ils contenaient précédemment...

```
PUSH AX
PUSH BX

MOV AX, [0140]
ADD BX, AX
MOV [0140], BX
```

```
POP BX
POP AX
```

Les registres SS et SP

Les registres SS et SP sont deux registres servant à gérer la pile:

- SS (*Stack Segment*, dont la traduction est *segment de pile*) est un registre 16 bits contenant l'adresse du segment de pile courant. Il doit être initialisé au début du programme
- SP (*Stack Pointer*, littéralement *pointeur de pile*) est le déplacement pour atteindre le sommet de la pile (16 bits de poids faible)

SP pointe vers le sommet, c'est-à-dire sur le dernier bloc occupé de la pile. Lorsque l'on ajoute un élément à la pile, l'adresse contenue dans SP est décrémentée de 2 octets (car un emplacement de la pile fait 16 bits de longueur). En effet, lorsque l'on parcourt la pile de la base vers le sommet, les adresse décroissent.
Par contre l'instruction POP inccrémente de 2 octets (16 bits) la valeur de SP.

- **PUSH**: SP <- SP - 2
- **POP**: SP <- SP + 2

Ainsi, lorsque la pile est vide SP pointe sous la pile (la case mémoire en-dessous de la base de la pile) car il n'y a pas de case occupée. Un POP provoquera alors une erreur...

Déclarer une pile

Pour pouvoir utiliser une pile, il faut la déclarer, c'est-à-dire réserver un espace mémoire pour son utilisation, puis initialiser les registres avec les valeurs correspondant à la base de la pile, ainsi que son sommet (rappel: situé sous la pile lorsque celle-ci est vide).
Ainsi pour définir une pile il s'agit tout d'abord de la déclarer grâce à la directive SEGMENT stack.
Suite aux déclarations, il faut écrire une séquence d'initialisation :

```
ASSUME SS:segment_pile

MOV  AX, segment_pile
MOV  SS, AX        ; initialise le segment de pile

MOV  SP, base_pile ; copier l'adresse de la base de la pile dans
SP
```

Il n'est pas possible de faire directement *MOV SS, segment_pile* car cette instruction n'existe pas!

Les procédures :

La notion de procédure

En langage assembleur, on appelle *procédure* un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel de la procédure. Cette notion de sous-programme est généralement appelée fonction dans d'autres langages. Les fonctions et les procédure permettent d'exécuter dans plusieurs parties du programme une série d'instruction, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une procédure peut

faire appel à elle-même, on parle alors de procédure récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).

La déclaration d'une procédure

Etant donnée qu'une procédure est une suite d'instruction, il s'agit de regrouper les instructions composant la procédure entre des mots clés. L'ensemble de cette manipulation est appelée *déclaration de procédure*.
Ces mots clés permettant la déclaration de la procédure sont le *une étiquette* (qui représente le nom de la fonction) précédant le mot clef *PROC* marquant le début de la procédure, suivi de *near* (qui signale que la procédure est situeacute;e dans le même segment que le programme appelant) et *RET* désignant la dernière instruction, et enfin le mot-clé *ENDP* qui annonce la fin de la procédure. Ainsi une déclaration de procédure ressemble à ceci:

```
étiquette       PROC            near
                instruction1
                instruction2
                ...
                RET
étiquette       ENDP
```

Appel d'une procédure

C'est l'instruction *CALL* qui permet l'appel d'une procédure. Elle est suivie soit d'une adresse 16 bits, désignant la position du début de la procédure, ou bien du nom de la procédure (celui de l'étiquette qui précède le mot clé *PROC*).

Comment l'appel et la fin de la procédure fonctionnent?

Lorsque l'on appelle une procédure, la première adresse de la procédure est stocké dans le registre IP (pointeur d'instruction), le processeur traite ensuite toutes les lignes d'instructions jusqu'à tomber sur le mot clé *RET*, qui va remettre dans le registre IP l'adresse qui y était stocké avant l'appel par *PROC*.
Cela paraît simple mais le problème provient du fait que les procédures peuvent être imbriqués, c'est-à-dire que de saut en saut, le processeur doit être capable de revenir successivement aux adresses de retour. En fait, à chaque appel de fonction l'instruction *CALL*, le processeur empile l'adresse contenue dans le registre IP (il pointe alors sur l'instruction suivant l'instruction CALL) avant de la modifier, à l'appel de l'instruction *RET* (qui ne prend pas d'arguments) le contenu de la pile est dépilé puis stocké dans le registre IP.

Le passage de paramètres

Une procédure effectue généralement des actions sur des données qu'on lui fournit, toutefois dans la déclaration de la procédure il n'y a pas de paramètres (dans des langages évolués on place généralement les noms des variables paramètres entre des parenthèses, séparés par des virgules).
Il existe toutefois deux façons de passer des paramètres à une procédure:

- **Le passage des paramètres par registre** : on stocke les valeurs dans les registres utilisés dans la procédure
- **Le passage des paramètres par pile**: on stocke les valeurs dans la pile avant d'appeler la procédure, puis on lit le contenu de la pile dans la procédure

Le passage de paramètres par registres

C'est une méthode simple pour passer des paramètres:
Elle consiste à écrire sa procédure en faisant référence à des registres dans les instructions, et de mettre les valeurs que l'on désire dans les registres juste avant de faire l'appel de la fonction...
Le passage des paramètres par registre est très simple à mettre en oeuvre mais elle est très limité, car on ne peut pas passer autant de paramètres que l'on désire, à cause du nombre limité de registres. On lui préfèrera le passage des paramètres par pile.

Le passage de paramètres par pile

Cette méthode de passage de paramètres consiste à stocker les valeurs des paramètres dans la pile avant l'appel de procédure (grâce à l'instruction *PUSH*), puis de lire le contenu de la pile grâce à un registre spécial (*BP*: Base pointer) qui permet de lire des valeurs dans la pile sans les dépiler, ni modifier le pointeur de sommet de pile (SP).
L'appel de la procédure se fera comme suit:

```
PUSH parametre1 ;où parametre1 correspond à une valeur ou une adresse
PUSH parametre2 ;où parametre1 correspond à une valeur ou une adresse
CALL procédure
```

La procédure commencera par l'instruction suivante:

```
MOV BP, SP    ;Permet de faire pointer BP sur le sommet de la pile
```

Puis pourra contenir des instructions du type:

```
MOV AX, [BP]     ;Stocke la valeur contenue dans le sommet de la pile dans AX, sans dépiler
MOV BX, [BP+2]   ;Stocke la valeur contenue dans le mot suivant de la pile dans BX (un mot fait 2 octets), sans dépiler
```

Segmentation de la mémoire :

Qu'est-ce que la segmentation de la mémoire?

En assembleur, on appelle segment de mémoire le mécanisme de base de la gestion des adresses pour des processeurs de type 80x86. Les instructions sont stockées dans le registre IP. Il s'agit d'adresses dont la taille est de 16 bits. Or, avec 16 bits il n'est possible d'adresser que 2^{16}= 64 Ko. Les bus d'adresses récents possèdent toutefois 32 bits. Les adresses qu'il véhicule sont donc constituées de deux composantes de 16 bits:

- un segment de 16 bits
- un déplacement (ou *offset* en anglais) de 16 bits

Les segments CS et DS

Les segments sont stockés dans des registres de 16 bits, dont les deux principaux sont:

- Le segment de données: **DS** (Data Segment)
- Le segment de code: **CS** (Code segment)

Le segment CS sert au processeur à lire le code d'une instruction. Lors de la lecture de celui-ci il crée une adresse de 32 bits formeacute;e de la paire constituée par le registre segment CS et le registre de déplacement IP, on note cette paire CS:IP

Le registre segment CS est initialisé automatiquement au chargement du programme sur le segment contenant la première instruction à exécuter.
Le segment DS sert au processeur pour aller chercher les données stockées à une adresse. Il crée alors la paire constitué du registre segment DS et de l'adresse 16 bits indiquée dans l'instruction. C'est au programmeur de définir dans le code source l'adresse du segment de donnée à utiliser, en initialisant le registre DS.

Liste des principales instructions :

Tableau des principales instructions en assembleur

Voici une liste (non exhaustive) des principales instructions en assembleur des processeurs 80x86, ainsi que du code machine qui leur est associé et de leur taille en mémoire. Les valeurs *val* et les adresses *adr* sont données sur 16 bits.

Instruction en assembleur	Instruction en code machine	Taille de l'instruction (en octets)	Descriptif de l'instruction
ADD AX, *Val*	05	3	Ajoute à AX la valeur indiquée et stocke le résultat dans AX
ADD AX, *Adr*	03 06	4	Ajoute à AX la valeur stockée à l'adresse indiquée et stocke le résultat dans AX
CMP AX, *Val*	3D	3	Compare AX et la valeur indiquée
CMP AX, *Adr*	3B 06	4	Compare AX et la valeur stockée à l'adresse indiquée
DEC AX	48	1	Décrémente AX (soustrait 1)
INC AX	40	1	Incute;crémente AX (ajoute 1)
JA *adr*			Saut à l'adresse indiquée si CF=0
JB *adr*			Saut à l'adresse indiquée si CF=1
JE *adr*	74	2	Saut à l'adresse indiquée si égalité
JG *adr*	7F	2	Saut à l'adresse indiquée si supérieur
JLE *adr*	7^E	2	Saut à l'adresse indiquée si inférieur
JNE *adr*	75	2	Saut à l'adresse indiquée si non égalité