République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
ECOLE SUPÉRIEURE EN INFORMATIQUE
8 Mai 1945 - Sidi-Bel-Abbès

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي
المدرسة العليا للإعلام الآلي
8 ماي 1945 - سيدي بلعباس

Département : Classe préparatoire

Laboratoire : LABRI

Polycopié pédagogique

Auteur

BEKKOUCHE Mohammed

Titre

Architecture et programmation du CPU 8086

Cours destiné aux étudiants de

Premier Cycle : 1ière année classe préparatoire

Année : 2021-2022

Sommaire

Dans ce polycopié, nous allons apprendre **l'assembleur pour le microprocesseur 8086**. Pour cela, nous allons étudier les principales caractéristiques des microprocesseurs en particulier du microprocesseurs **8086** fabriqué par l'entreprise **Intel**, le premier processeur de la famille **x86**[1] qui est devenue l'architecture de processeur **la plus populaire au monde**. Nous allons aborder, comme il est le cas pour l'étude de tout microprocesseur, **l'architecture** et **la programmation** (le jeu d'instructions).

[1] La série x86 se compose de microprocesseurs compatibles avec le jeu d'instructions Intel 8086.

Introduction

Contrairement aux langages de **haut niveau**, **l'assembleur** ou « **langage d'assemblage** » est constitué d'instructions que le microprocesseur peut directement **comprendre** : c'est ce qu'on appelle **un langage de bas niveau**. Par conséquent, il est étroitement **lié au fonctionnement de la machine**. Cela explique aussi pourquoi il existe au moins autant de langages assembleurs que de **modèles de microprocesseurs**. Avant d'apprendre l'assembleur **Intel 8086**, il est donc primordial de s'intéresser à certains concepts de base, tels que **la mémoire** ou **le microprocesseur**.

Ce polycopié est rédigé à l'intention des étudiants de la première année classe préparatoire de l'école supérieure en informatique de Sidi Bel Abbès. Il constitue un manuel de cours et d'exercices pour le module Introduction aux Systèmes d'Exploitation 2 enseigné dans le deuxième semestre. Les lecteurs nécessitent d'avoir suivis les cours d'algorithmique (pour connaître par exemple les principes fondamentaux de l'algorithmique et avoir des notions de programmation, … etc.) et le cours d'architecture 1 (pour connaitre les systèmes de numération comme le binaire, l'octal et l'hexadécimal et avoir une idée sur l'architecture de base d'un ordinateur et savoir ce que c'est les circuits logiques, … etc.). Le cours Introduction au Système d'exploitation 1 n'est pas un prérequis. Le présent module est parallèle est non pas une suite du précédent.

L'objectif de ce cours :

- Présenter **les notions de base** nécessaires à la compréhension de l'architecture des systèmes utilisant des microprocesseurs 8086 :
  - Expliquer l'architecture d'un ordinateur dite architecture de von Neumann.
  - Fournir un aperçu des différents membres du microprocesseur 8086.
  - Décrire la fonction du microprocesseur et détailler son fonctionnement de base.
  - Définissez le contenu du système de mémoire dans l'ordinateur.
  - Décrire le mécanisme de la segmentation de la mémoire en 8086.
  - Décrire la fonction et le but de chaque registre dans le microprocesseur 8086.
  - Détaillez le registre de drapeau et le but de chaque bit de drapeau.
  - Expliquer le fonctionnement de la pile.
- Permettre aux étudiants de **programmer en langage machine** (**assembleur**) du 8086 :
  - Expliquer le fonctionnement de chaque mode d'adressage du 8086.
  - Sélectionner le mode d'adressage approprié pour accomplir une tâche donnée.
  - Expliquez le fonctionnement de chaque instruction de mouvement de données avec les modes d'adressage applicables.
  - Expliquer les objectifs des pseudo-instructions (directives) du langage assembleur.
  - Expliquer la syntaxe d'une instruction assembleur 8086.
  - Sélectionner l'instruction en langage assembleur appropriée pour accomplir un transfert ou des conversions de données.
  - Utiliser les instructions relatives aux drapeaux.
  - Utiliser les instructions pour modifier les bits du registre d'état (FLAGS).
  - Utiliser des instructions arithmétiques et logiques.
  - Utiliser AND, OR et OU Exclusive pour effectuer une manipulation de bits binaires.
  - Utiliser les instructions de décalage et de rotation.
  - Utiliser les instructions de manipulation des tableaux ou chaines de caractères.
  - Utiliser des instructions de saut conditionnelles et inconditionnelles pour contrôler le déroulement d'un programme.

- Utiliser les instructions d'appel (call) et de retour (ret) pour inclure des procédures dans la structure du programme.
- Expliquer le fonctionnement des interruptions logicielles et des instructions pour contrôler ces interruptions.
- Montrer comment mettre en place une procédure à l'aide de PROC et ENDP.
- Montrer comment mettre en place une macro à l'aide de MACRO et ENDM.
- Sélectionnez les instructions en langage assembleur appropriée pour accomplir une tâche spécifique.

Ce polycopié est structuré en cinq chapitres comme suit :

Dans le premier chapitre qui est introductif, nous donnerons des notions de base. Plus précisément, nous allons aussi expliquer les différents types de codage des nombres entiers nécessaires à la compréhension du reste du document.

Le deuxième chapitre décrit ce que s'est un système à base de microprocesseur en général (architecture d'un ordinateur, architecture d'une CPU, bus de données, bus d'adresses, …).

Le troisième chapitre est sur l'architecture du microprocesseur Intel 8086. Dans cette partie sont présentés, la description des signaux, l'organisation interne, les registres, la gestion de la mémoire et l'implémentation de la pile du 8086.

Le quatrième chapitre est consacré aux différents modes d'adressages que nous pouvons rencontrer en programmation assembleur 8086, ainsi que la structure d'une instruction et directive dans un code source 8086.

Enfin, dans le cinquième chapitre nous détaillerons les instructions supportées par le processeur 8086, comment accéder à la mémoire vidéo du i8086 et les Procédures et Macros en 8086.

Nous trouverons dans ce manuscrit plusieurs exemples, quelques exercices et leurs corrigés. Nous donnons également en fin de celui-ci une liste de références bibliographiques sur lesquelles s'est basée la rédaction de ce polycopié.
L'évaluation finale des étudiants se fait à travers :

1) Un examen intermédiaire et final sur table qui portent sur tout ce que les étudiants ont vu dans ce cours pendant le semestre. La note est de **60%** de la moyenne générale (**30%** pour chaque examen). Lors de ces examens, les étudiants auront :
   a) À résoudre des problèmes similaires ou proches aux problèmes traités lors des TPs et des interrogations.
   b) À répondre à des questions de synthèse (via des QCM) ¸
   c) À répondre des questions de réflexion. (les étudiants seront entraînés à répondre à ce type de questions par les questions posées lors des TPs et des cours)
2) Évaluation continue et régulières qui compte pour **40%** de la moyenne. Elle permet aux étudiants d'engranger des points tout au long du semestre, cette évaluation continue est réalisée par différentes formes, il s'agit :
   a) De la note des interrogations écrites,
   b) De la note de participation en classe,
   c) De la note de bonne conduite et assiduité.

Chapitre I : Notions de base

Introduction

Ce chapitre a pour but de présenter des concepts de base et définitions utiles. En effet, nous présentons les quatre **types de codage en binaire des nombres entiers**.

1. Quelques définitions

- **un bit** est une valeur binaire qui, par convention, peut prendre la valeur **0** ou **1**.
- **un octet** est une donnée codée sur **8 bits**. Les unités utilisées **pour quantifier une information** se basent sur l'octet, ce qui équivaut à **8 bits**.
- **un mot** est une donnée codée sur **16 bits** (**deux octets**).
- **un Ko (kilo-octet)** est un ensemble de **1024 octets (2^{10} octets)**.
- **un MO (mégaoctet)** est égal à **1024 Ko (2^{20} octets)**.

2. Le codage des nombres

Les instructions du 8086 manipulent **des valeurs numériques entières codées sur 1 ou 2 octets**. Nous décrivons ici le codage des nombres entiers. Nous distinguons quatre types de codage :

- **non signé** pour des nombres entiers forcément positifs ;
- **signé** pour des nombres entiers positifs ou négatifs.
- **décimal** compacté, ou décimal codé binaire
- **décimal non compacté**

C'est au programmeur de décider du codage qu'il utilise et à écrire son programme en conséquence. L'ordinateur ne sait pas quel est le codage utilisé. Il exécute simplement des instructions.

4.1 Représentation des entiers non signés en binaire

En utilisant **k bits**, **2^k** valeurs différentes peuvent être encodées. Si **k = 8**, on peut donc coder **256** valeurs différentes. On peut établir le codage suivant, sur 8 bits (cf. fig 1) :

Valeur	Codage
0	00000000
1	00000001
2	00000010
3	00000011
...	...
84	01010100
...	...
254	11111110
255	11111111

Figure 1: Représentation non signée des nombres entiers sur 8 bits

Ce code est utilisé pour représenter tout entier compris entre **0** et **255 (0 et $2^k - 1$)** avec **8 bits (k bits)**. Sur **un mot** (**16 bits**), on peut coder des valeurs allant de **0** à **65535** (**0** à **$2^{16} - 1$**). Parce qu'il ne représente que des nombres positifs, ce codage est appelé **non signé**.

4.2 Représentation des nombres entiers signés en binaire

Si l'on veut pouvoir représenter **des nombres positifs** et **négatifs**, l'idée naturelle consiste, plutôt qu'à coder des nombres compris entre **0** et **255**, à représenter des nombres entre **-128** et **+127**, ce qui représente toujours **256** valeurs différentes à coder avec **8 bits** (cf. fig 2).

Valeur	Codage
-128	10000000
-127	10000001
...	...
-45	11010011
...	...
-1	11111111
0	00000000
1	00000001
2	00000010
...	...
84	01010100
...	...
126	01111110
127	01111111

Figure 2: Représentation signée des nombres entiers sur 8 bits

L'intervalle des valeurs couvertes sur **k bits** est égal : [-2^{k-1}, $2^{k-1}-1$]. Ce codage se nomme codage par **complément à deux**. Dans **le codage signé**, on constate que le bit de poids fort vaut **1** pour tous les nombres négatifs, **0** pour tous les nombres positifs. Ce bit indique donc le signe du nombre codé. Aussi, ce bit se nomme **le bit de signe**. Il faut bien prendre garde que si **le bit de signe** vaut **0**, la valeur représentée est la même en codage signé ou non signé. Par contre, si **le bit de signe** vaut **1**, la valeur codée est supérieure à 127 en codage non signé, inférieure à 0 en codage signé.

4.2.1 Conversions avec le code complément à 2

Nous indiquons ici comment obtenir **le complément à deux** d'un nombre. Cette méthode donne la représentation binaire signée d'un nombre négatif. L'algorithme est simple. Soit à calculer la représentation du nombre **−n** (où **n** est un nombre positif). On effectue les opérations suivantes :

- écrire **n** sous forme binaire
- en complémenter tous les bits (**0 → 1**, **1 → 0**)
- ajouter la valeur **1** à ce nombre binaire

Nous présentons cet algorithme sur un exemple. Soit à calculer **le complément à deux** de **-92** :

- la représentation binaire sur **8 bits** de **92** est **01011100**
- par complément des bits, on obtient : **10100011**
- en ajoutant **1**, on obtient : **10100100**

Le complément à deux de **01011100** est donc **10100100**. **La représentation binaire signée** de **-92** est donc **10100100**

4.3 Codage décimal

Le codage décimal (ou BCD : Binary Coded Decimal qui signifie décimal codé binaire) permet de représenter un chiffre décimal de **0** à **9** par un ensemble de **4 bits**. **4 bits** codant en principe **16** valeurs différentes, seules **10** de ces **16** combinaisons sont effectivement utilisées en codage décimal, les autres n'ayant alors pas de sens (cf. fig 3) :

Valeur	Codage
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure 3: Codage décimal

On distingue deux types de représentation, **compactée ou non**. Un chiffre nécessitant **4 bits** pour son codage, on peut représenter deux chiffres par octet. Ce codage porte le nom de **codage décimal compacté**. On peut également ne mettre qu'un seul chiffre par octet, les **4 bits** de poids fort de l'octet étant inutilisés. On qualifie ce **codage de décimal non compacté**. Par exemple, 89 se code:

- **décimal compacté** : 10001001
- **décimal non compacté** : 00001000 00001001

Deux octets sont nécessaires dans la représentation non compactée.

4.4 Extension du signe

Le bit de signe, le bit le plus significatif, est **répété** dans **la nouvelle partie** pour convertir le nombre binaire en une plus grande largeur de bit. Par exemple, quand on transfère **l'octet 0**100 1010 dans **un mot** (**16 bits**), on répète le bit de signe dans la moitié supérieure : **0000 0000** 0100 1010. Également, l'octet **1**011 0101 deviendra le mot **1111 1111** 1011 0101.

Conclusion

Dans ce chapitre nous avons défini **des concepts de bases**. Nous avons vu la représentation non signée et signée des nombres entiers en binaire, le codage décimal (BCD) compacté et non compacté. Nous présentons dans le chapitre suivant **les systèmes à base de microprocesseur** en général.

Chapitre II : Microprocesseur et son architecture

Introduction

Les architectures d'ordinateurs représentent les moyens d'i**nterconnectivité** pour les composants matériels d'un ordinateur ainsi que **le mode de transfert** et de traitement des données exposé. Dans ce chapitre, nous décrivons l'architecture d'un ordinateur ou dit de **Von Neumann**. Nous décrivons ensuite les différents types de bus selon l'usage, la mémoire et les interfaces d'entrées sorties. Le reste du chapitre présente les éléments fonctionnels d'un CPU, le traitement des instructions et les catégories de processeurs généraux.

1. Architecture d'un ordinateur (Von Neumann)

L'**architecture de von Neumann** est un modèle structurel d'ordinateur dans lequel une unité de stockage (mémoire) unique sert à conserver à la fois les instructions et les données demandées ou produites par le calcul. Les ordinateurs actuels sont tous basés sur des versions améliorées de cette architecture.

Cette architecture est appelée ainsi en référence au mathématicien John von Neumann qui a élaboré la première description d'un ordinateur dont le programme est stocké dans sa mémoire.

L'**architecture de von Neumann** décompose l'ordinateur en 4 parties distinctes (cf. fig 4) :

- L'unité arithmétique et logique (UAL ou *ALU* en anglais) ou unité de traitement : son rôle est d'effectuer les opérations de base ;
- L'unité de contrôle, chargée du « séquençage » des opérations ;
- La mémoire qui contient à la fois les données et le programme qui indiquera à l'unité de contrôle quels sont les calculs à faire sur ces données. La mémoire se divise entre mémoire volatile (programmes et données en cours de fonctionnement) et mémoire permanente (programmes et données de base de la machine) ;
- Les dispositifs d'entrée-sortie, qui permettent de communiquer avec le monde extérieur.

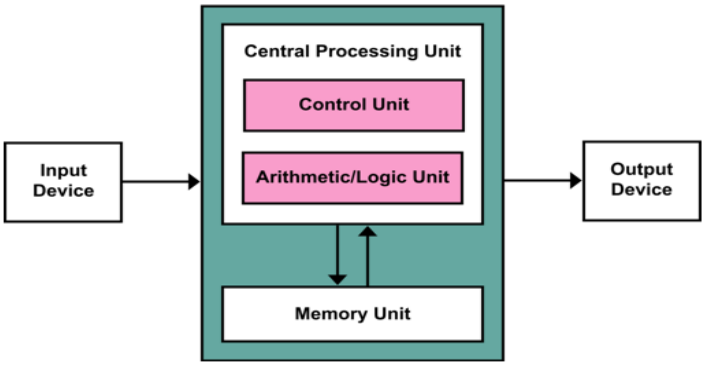


Figure 4: Architecture de von Neumann

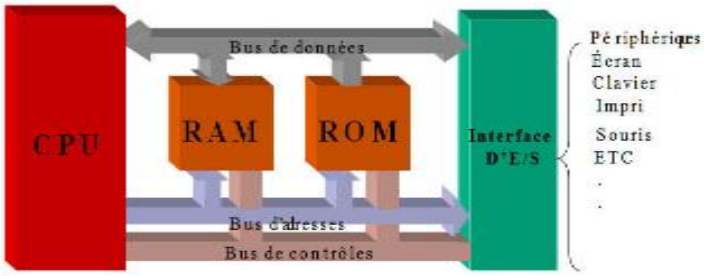


Figure 5: Architecture d'un système à base de microprocesseur

Un système à base de microprocesseur (cf. fig 5) est formé des trois éléments : Une unité CPU (central processing unit) ; une mémoire (ROM et RAM) ; des ports d'entrées/sorties. Les trois modules sont interconnectés comme le montre la figure 5 autour de trois bus : bus de données, bus d'adresses et bus de contrôles et commandes.

2. Bus

Selon **l'architecture de Von Neumann**, le microprocesseur échange des informations avec la mémoire et l'unité d'entrée-sortie, au moyen d'un ensemble de connexions appelé **bus**. On retrouve 2 principaux types de Bus : **Parallèle et Série** (cf. fig 6). On peut retrouver 3 types de Bus selon l'usage : le bus de données ; le bus d'adresses ; le bus de commandes et de contrôle.

Figure 6: Bus Parallèle et Série

2.1 Bus de données

C'est un ensemble de **fils bidirectionnels** qui va permettre le transfert de données entre les différents éléments du système. C'est par ce bus que sont transmises les données qui doivent être traitées par le microprocesseur. A l'inverse, c'est également par ce bus que transitent les résultats en sortie du microprocesseur. Autrement dit, toutes les données entrantes et sortantes du microprocesseur sont véhiculées par **le bus de données** qui fixe la longueur du mot échangé avec la mémoire.

2.2 Bus d'adresses

Le **bus d'adresses** (appelé parfois bus d'adressage ou bus mémoire) transporte les adresses mémoire auxquelles le processeur souhaite accéder pour lire ou écrire une donnée. Il est **unidirectionnel.** La largeur du bus d'adresses détermine la quantité de mémoire que le système peut adresser. Quand il y a **'n'** lignes d'adresse, il peut directement adresser $\mathbf{2^n}$ emplacements de mémoire. Par exemple, un microprocesseur **8085** a un bus d'adresse de **16 bits**. Par conséquent, il peut accéder à $\mathbf{2^{16} = 65536}$ emplacements de mémoire différents. Le bus d'adresse du **8086** est de **20 bits**, ce qui lui permet d'adresser **1 Mo** d'espace mémoire.

2.3 Bus de commandes et de contrôle

C'est un bus qui permet de véhiculer **les signaux de contrôles et de commandes** tels que **l'horloge**, **les signaux Rd/Wr** (les ordres de lecture et d'écriture de la mémoire) etc ... Ce bus sert à coordonner tous les échanges d'informations décrits précédemment. Il véhicule des données qui valident la mémoire et les ports d'entrées sorties. Il introduit **des délais d'attente** lorsque des informations sont envoyées à un périphérique qui présente une vitesse de traitement réduite. Le bus de commandes évite **les conflits de bus** lorsque deux éléments cherchent à communiquer en même temps.

2.4 Remarque

Dans certains cas, le **bus de données et le bus d'adresses** sont **multiplexés** sur un seul bus. Une logique externe doit alors effectuer le **démultiplexage**. Le terme "multiplexage" signifie d'une manière générale toute technique visant à véhiculer sur une même voie des signaux différents et indépendants.

3. Mémoire

La mémoire sert au rangement de deux types d'informations :

- **Des données :** les informations traitées par le microprocesseur.
- **Des instructions :** ensemble d'informations codées qui gère l'activité du microprocesseur.

Remarque :

- **La mémoire morte** (**ROM : Read Only Memory**) range en général le programme d'initialisation du système (exemple dans le PC elle range le BIOS : Basic Input Ouput systeme).
- **La mémoire vive** (**RAM : Random Access Memory**) sert au rangement des programmes utilisateurs c'est une mémoire volatile.

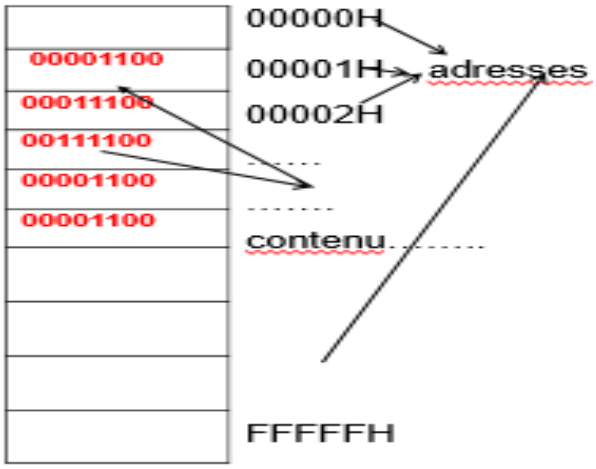


Figure 7: Organisation de la mémoire centrale

La mémoire centrale est faite à base de **semi-conducteur** (contrairement au CD-ROM, HDD, …). C'est **une mémoire vive** (accessible en lecture et écriture). Elle est **volatile** (s'efface en absence de courant). Elle est dite **aléatoire** RAM (toutes les cellules ont le même temps d'accès).
Une mémoire peut être représentée comme une armoire de rangement constituée de différents tiroirs (cf. fig 7). Chaque tiroir représente alors une case mémoire qui peut contenir un seul élément : des données. Le nombre de cases mémoires pouvant être très élevé, il est alors nécessaire de pouvoir les identifier par un numéro. Ce numéro est appelé adresse. Chaque donnée devient alors accessible grâce à son adresse. Avec une adresse de n bits il est possible de référencer au plus 2^n cases mémoire. Chaque case est remplie par un mot de données (sa longueur m est toujours une puissance de 2). Le nombre de fils d'adresses d'un boîtier mémoire définit donc le nombre de cases mémoire que comprend le boîtier (bus d'adresse). Le nombre de fils de données définit la taille des données que l'on peut sauvegarder dans chaque case mémoire (bus de données). En plus du bus d'adresses et du bus de données, un boîtier mémoire comprend une entrée de commande qui permet de définir le type d'action que l'on effectue avec la mémoire (lecture/écriture) et une entrée de sélection qui permet de mettre les entrées/sorties du boîtier en haute impédance.

4. Interfaces d'entrées sorties

Les interfaces d'entrées sorties vont permettre au microprocesseur de **communiquer avec le monde extérieur**. Nous trouvons **des ports**[2] utilisés exclusivement pour l'entrée, et d'autres **ports** exclusivement pour la sortie. Il existe aussi des ports bidirectionnels. Donc le microprocesseur peut lire des données à partir d'une interface d'entrée (exemple souris, clavier disque dur, Etc. ...) de même il peut restituer le résultat de son traitement au monde extérieur en adressent des interfaces de sortie (tel que les imprimantes l'écran etc. ...) donc les interfaces d'entrées / sorties vont **soulager le microprocesseur pour la communication avec le monde extérieur**.

5. Processeur

Le processeur est **un circuit intégré** (une petite puce) à très haute intégration (jusqu'à **plusieurs milliards de transistors**) constitué de plusieurs circuits qui réalisent des fonctions de traitement (exécute des instructions comprenant un programme informatique). Un processeur effectue des opérations arithmétiques, logiques, de contrôle et d'entrée/sortie (E/S) et d'autres instructions de base spécifiées par les instructions du programme. Les termes processeur, unité centrale de traitement (CPU) et microprocesseur sont généralement liés en tant que synonymes. Il constitue **le**

[2] Les périphériques sont reliés au reste du système par des circuits appelés ports d'entrées et ports de sortie (certains ports peuvent combiner les deux fonctions).

cerveau de tout ordinateur. Les instructions sont stockées **en mémoire** (en dehors du processeur). Ces instructions (dont l'ensemble compose **le langage d'assemblage**, ou **assembleur**) sont très simples mais n'en permettent pas moins, en les combinant, de réaliser n'importe quelle opération programmable. Pour exécuter le programme, le processeur lit les instructions dans la mémoire **une par une**. Il connaît toujours l'adresse (emplacement en mémoire) de la prochaine instruction à exécuter car il stocke cette adresse dans **son compteur ordinal**.
Les instructions agissent sur des données qui sont situées soit **en mémoire**, soit dans **des registres** du processeur. Un registre est un élément de stockage à l'intérieur du processeur et contient une valeur. Le nombre de registres est très limité, dans le cas du 8086 vaut **14**. Pour accéder à une donnée en mémoire, son adresse doit être précisée. Pour accéder à une donnée d'un registre, vous devez spécifier son nom (chaque registre a un nom, qui est une chaîne de caractères). Parce que les registres sont à l'intérieur du processeur, ils peuvent être **lus très rapidement**. Le 8086 est **un processeur 16 bits**, c'est-à-dire qu'il traite des données codées sur **16 bits**.
Le processeur travail de pair avec une mémoire centrale et communique au travers de bus pour réaliser ces traitements. Il est théoriquement constitué d'une ou plusieurs **unités d'exécution**, d'**une Unité de Contrôle** et de **Commande** et d'un ensemble de **Registres.** Concrètement (actuellement) il peut aussi inclure de **la mémoire cache**, **un coprocesseur graphique**, **différents contrôleurs**, …
Il existe trois grands types de processeurs numériques :

- Central Processing Unit (**General Purpose Processor**, Micro Controler Unit, Digital Signal Processor)
- Circuits logiques Programmables (Field-Programmable Gate Array, Programmable Logic Device)
- GPU (Processeur graphique)

5.1 Architecture d'une CPU

Une CPU (cf. fig 8) est formée par les trois éléments fonctionnels interconnectés suivants :

- Registres.
- UAL : Unité arithmétique et logique.
- Circuit de contrôle.

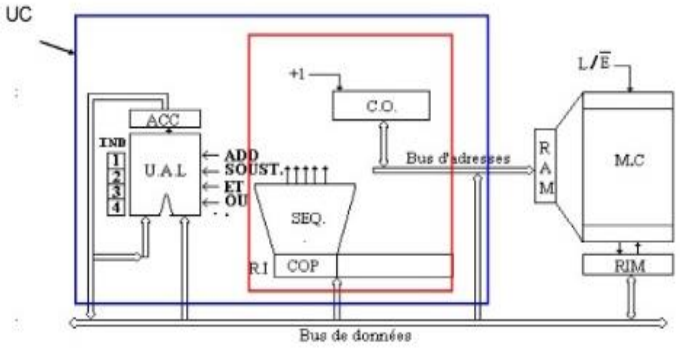


Figure 8: Architecture d'une CPU

5.1.1 Unité de contrôle

Le rôle de **l'unité de contrôle** (ou **unité de commande)** est de : **coordonner** et **synchroniser** le travail de toutes les autres unités (UAL, mémoire,…) ; **rechercher** (lecture de) l'instruction et des données à partir de la mémoire ; **décoder** l'instruction et **l'exécution** de l'instruction en cours ; **préparer** l'instruction suivante.
L'unité de contrôle comporte :

- Un registre qui s'appelle **compteur ordinal (CO)** ou **compteur de programme (CP)** ou **Instruction Pointer** (**IP** sur les processeurs Intel) : Il contient l'adresse de la prochaine instruction à exécuter (pointe vers la prochaine instruction à exécuter). Initialement il contient l'adresse de la première instruction du programme à exécuter.
- **Un registre instruction (RI)** : contient l'instruction en cours d'exécution. Chaque instruction est décodée selon son code opération grâce à un décodeur. C'est lui qui va "décoder" l'instruction contenue dans RI et générer les signaux logiques correspondant et les communiquer au séquenceur

- **Un séquenceur** : il organise (synchronise) l'exécution des instructions selon le rythme de l'horloge, il génère les signaux nécessaires pour exécuter une instruction.

5.1.2 Unité arithmétique et logique

Comme son nom l'indique, cette unité peut exécuter **deux types d'opérations**.

- **Opérations arithmétiques :** Elles incluent **l'addition** et **la soustraction** qui sont des opérations de base (une soustraction est une addition avec le complément à deux), **la multiplication** et **la division**. Les données traitées sont considérées dans des représentations entières.
- **Opérations logiques :** Ces opérations sont effectuées bit à bit sur les bits de même poids de deux mots, tel que **ET**, **OU**, **NOT**, **OU EXCLUSIF**, de même **les opérations de rotation** et **de décalage**. Elle reçoit ses opérandes (les octets qu'elle manipule) du bus de données. Celles-ci peuvent provenir de registres ou de la mémoire. A la fin d'une opération, **l'UAL** peut aller modifier certains bits du **registre d'état** (**FLAG**). Par exemple, dans le cas du débordement d'une addition (que le résultat de l'addition est trop grand pour entrer dans un registre), l'UAL va mettre le bit de débordement du **FLAG à 1**.

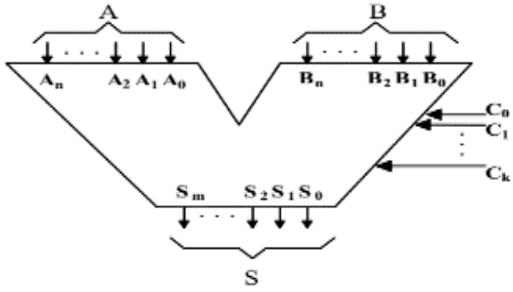


Figure 9: Unité arithmétique et logique

L'UAL est constituée d'un circuit logique combinatoire qui reçoit **deux opérandes A et B** et produit **le résultat S** selon l'indication appliquée sur **l'entrée C** (cf. fig 9). Les opérations réalisées peuvent être soit **arithmétiques** ou **logiques**.

L'UAL comporte :

- **Un registre d'état** : Il indique **l'état du déroulement de l'opération**. Il se compose d'un ensemble de bits appelés **indicateurs** (**drapeaux** ou **flags**). Ces indicateurs sont mis à jours (modifiés) après **la fin de l'exécution** d'une opération dans l'UAL. Par exemple, quand le résultat d'une opération est trop grand pour être contenu dans le registre cible (celui qui doit contenir le résultat de l'opération), un bit spécifique du registre d'état (**le bit OF**) est mis à **1** pour indiquer le débordement.
- **Un registre accumulateur** (**ACC**) : registre de travail qui sert à stocker un opérande (données) au début d'une opération et le résultat à la fin. **Il évite des appels fréquents à la mémoire**, réduisant ainsi les temps de calcul. Donc la plupart des opérations arithmétiques et logiques se font dans l'accumulateur.

Les principaux indicateurs sont :

- **Retenue** : mis à 1 si l'opération génère une retenue.
- **Signe** : mis à 1 si l'opération génère un résultat négatif.
- **Débordement** : mis à 1 s'il y a un débordement.
- **Zero** : mis à 1 si le résultat de l'opération est nul.

5.1.3 Registres

Le microprocesseur peut contenir d'autres registres, autre que **CO**, **RI** et **ACC**. Ces registres sont considérés comme une mémoire interne du microprocesseur. Ces registres sont **plus rapides que la mémoire centrale,** mais **leur nombre est limité.**

5.2 Traitement des instructions

Les étapes du traitement des instructions par la CPU (cf. fig 10) sont :

1. **FETCH :** récupération du code binaire d'une instruction en mémoire
2. **DECODE :** décodage du code opération de l'instruction récupérée
3. **EXECUTION :** Exécution de l'instruction décodée
4. **WRITEBACK :** Ecriture du résultat en mémoire ou dans les registres du CPU

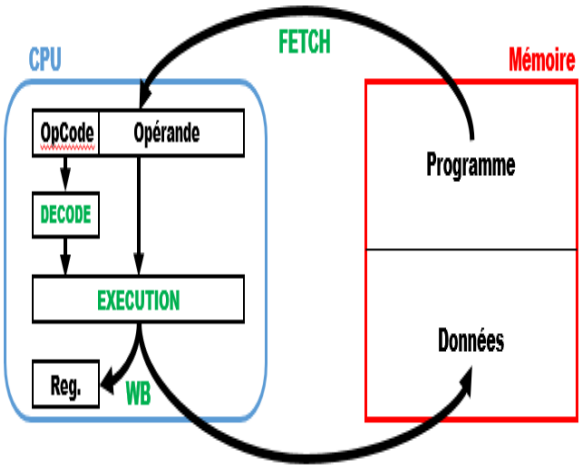


Figure 10: Etapes du traitement des instructions par le CPU

5.3 Jeu d'instructions

Le jeu d'instructions décrit l'ensemble des opérations élémentaires que le microprocesseur pourra exécuter. Il détermine en partie l'architecture du microprocesseur à réaliser et notamment celle du séquenceur. Les instructions que l'on retrouve dans chaque microprocesseur peuvent être classées en 4 groupes :

1. Transfert de données pour charger ou sauver en mémoire, effectuer des transferts de registre à registre, etc…

2. Opérations arithmétiques : addition, soustraction, division, multiplication

3. Opérations logiques : ET, OU, NON, NAND, comparaison, test, etc…

4. Contrôle de séquence : branchement, test, etc…

5.4 Architectures de processeur

Les processeurs généraux actuels se répartissent en deux grandes catégories appelées **CISC** pour ***Complex Instruction Set Computer*** et **RISC** pour ***Reduced Instruction Set Computer***. Les processeurs de ces deux catégories se distinguent par la conception de leurs jeux d'instructions. Les processeurs **CISC** possèdent un jeu étendu d'instructions complexes. Chacune de ces instructions peut effectuer plusieurs opérations élémentaires comme charger une valeur en mémoire, faire une opération arithmétique et ranger le résultat en mémoire. Au contraire, les processeurs **RISC** possèdent un jeu d'instructions réduit où chaque instruction effectue une seule opération élémentaire. Le jeu d'instructions d'un processeur **RISC** est plus uniforme. Toutes les instructions sont codées sur la même taille et toutes s'exécute dans le même temps (un cycle d'horloge en général).

5.5 Instruction Assembleur Intel

La structure la plus générale d'une instruction est la suivante :

Instruction Opérande1, Opérande2

L'opération est réalisée entre **les 2 opérandes** et le résultat est toujours récupéré dans **l'opérande de gauche**. Il y a aussi des instructions qui agissent sur **un seul opérande**. Les opérandes peuvent être **des registres**, **des constantes** ou **le contenu de cases mémoire**, on appelle ça **le mode d'adressage.**

Conclusion

Un ordinateur est composé de plusieurs parties, à la fois **matérielles** et logicielles. Au cœur de l'ordinateur se trouve le **processeur**, le matériel qui **exécute** les programmes informatiques. Il est constitué d'une ou plusieurs **unités d'exécution (unité arithmétique et logique)**, d'**une Unité de Contrôle** et de **Commande** et d'un ensemble de **Registres.** L'ordinateur possède également de la mémoire, souvent de plusieurs types différents dans un même système. La mémoire est utilisée pour **stocker les programmes** pendant que le processeur les exécute, ainsi que pour stocker **les données** que les programmes manipulent.

Chapitre III : Microprocesseur 8086

Introduction

Le processeur 8086 est le premier microprocesseur **16 bits** conçu par Intel en **1978**. Il constitue un bon exemple pour les débutant dans le domaine des microprocesseurs. Le 8086 doit sa renommée au fait qu'il a été utilisé par **IBM** pour construire **le premier** ordinateur personnel, c'est également le premier de la famille **x86** des processeurs qui ont équipé un très grand nombre d'ordinateurs personnels dans le monde. Il est l'un **des ancêtres** des processeurs actuels de Intel qui continue d'occuper une grande part du marché dans le domaine des ordinateurs personnels. Il est constitué de **29000 transistors** sur une puce de **32,7mm²** et peut fonctionner dans des systèmes mono ou multiprocesseurs.

Le processeur 8086 d'Intel est à la base des processeurs **Intel Core** actuels. Les processeurs successifs (de PC) se sont en effet construits petit à petit en ajoutant à chaque processeur des instructions et des fonctionnalités supplémentaires, mais en conservant à chaque fois les spécificités du processeur précédent. C'est cette façon d'adapter les processeurs à chaque étape qui permet qu'un ancien programme écrit pour **un 8086** fonctionne toujours sur un ordinateur équipé d'**un Intel Core**.

Avant d'écrire **le programme assembleur**, les concepteurs doivent avoir des connaissances suffisantes sur **le matériel** particulier du processeur, nous devons donc d'abord connaître **le matériel du processeur 8086**. Ce chapitre présent le brochage et l'architecture interne ainsi que les registres du 8086. Nous expliquons par la suite le principe de la segmentation de la mémoire et l'implémentation de la pile du 8086.

1. Brochage du 8086

Le 8086 se présente sous la forme d'un boîtier **DIP** (**Dual Inline Package**) à **40 broches** (cf. fig 11). Les fonctions de toutes ces broches seront décrites dans ce qui suit.

MN/$\overline{MX}$: Cette ligne c'est l'entrée du choix du mode de fonctionnement. Lorsqu'elle est à 1, le 8086 va fonctionner en mode minimal : microprocesseur à 8 bits et ne peut adresser que 64ko de mémoire donc il a 8 lignes de données et 16 lignes d'adresses. Lorsque MN/$\overline{MX}$ est à 0, c'est le cas du fonctionnement en mode maximal : microprocesseur 16 bits pouvant adresser 1MO de mémoire dont il doit avoir 16 lignes de données et 20 lignes d'adresses. Le mode maximal permet de réaliser également des systèmes multiprocesseurs.

AD15...AD0 : Ce sont 16 lignes multiplexées entre les lignes de A0 jusqu'à A15 du bus d'adresse et les lignes de données de D0 jusqu'à D15. Les concepteurs de microprocesseurs ont opté pour un bus multiplexé entre les adresses et les données afin de réduire le nombre de broches sur le microprocesseur, ce qui fait un démultiplexage est nécessaire pour pouvoir séparer les lignes d'adresses et de données.

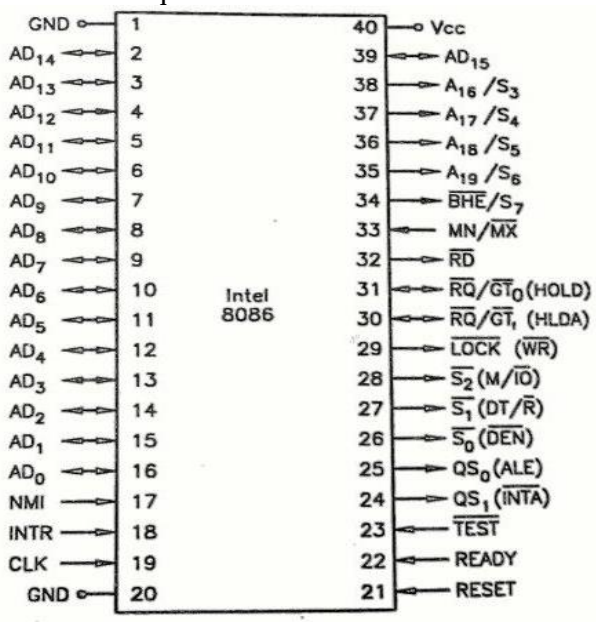


Figure 11: Brochage du 8086

S0 à S7 : Ce sont les signaux d'état, ils permettent d'indiquer le type d'opération en cours sur le bus.
A16/S3 à A19/S6 : Ce sont aussi des lignes multiplexées entre les lignes d'adresses de A16 jusqu'à A19 et les lignes d'état de S3 jusqu'à S6.
CLK : C'est un signal d'horloge qui cadence le fonctionnement du microprocesseur.
RESET : C'est le signal qui permet de réinitialiser le microprocesseur. Lorsque l'entrée RESET est mise à 1 pendant au moins quatre périodes d'horloge, le microprocesseur est réinitialisé.
READY : C'est l'entrée de synchronisation avec la mémoire ou bien avec un circuit d'entrées/sorties.
$\overline{\boldsymbol{TEST}}$ **:** Ce signal permet la mise en attente du microprocesseur.
INTR : C'est l'entrée de demande d'interruption masquable.
NMI : C'est l'entrée de demande d'interruption non masquable.
$\overline{\boldsymbol{INTA}}$ **:** C'est la sortie indiquant la réponse à une demande d'interruption.
HOLD : C'est une demande d'accès direct à la mémoire (DMA).
HLDA : C'est l'accusé de réception de l'accès DMA.
$\overline{\boldsymbol{RD}}$ **:** C'est le signal de lecture.
$\overline{\boldsymbol{WR}}$ **:** C'est le signal d'écriture.
DT/$\overline{\boldsymbol{R}}$ **:** C'est un signal qui permet d'indiquer le sens du transfert des données sur le bus de données. Lorsqu'elle est à 1, elle indique une émission donc un envoi d'une donnée à partir du microprocesseur vers l'extérieur. Lorsque cette ligne est à 0, elle indique une opération de réception.
M/$\overline{\boldsymbol{IO}}$ **:** Cette ligne permet de d'indiquer un accès à la mémoire ou bien à l'espace d'entrées sorties (M/$\overline{IO}$ = 1 : accès mémoire, M/$\overline{IO}$ = 0 : accès port d'E/S).
ALE : C'est la sortie indiquant que l'information qui circule dans bus AD est une adresse. Lorsqu'elle est 1, elle permet d'indiquer que le bus multiplexé entre adresses et données est entrain de véhiculer des adresses. Lorsque ALE = 1, ce sont des adresses qui sont présentes sur le bus multiplexé.
$\overline{\boldsymbol{DEN}}$ **:** C'est la sortie indiquant que l'information qui circule dans bus AD est une donnée.
$\overline{\boldsymbol{BHE}}$ **:** C'est la ligne de validation de la partie haute du bus de données (D8…D15).

2. Architecture interne du 8086

Le microprocesseur 8086 est divisé en deux unités fonctionnelles (cf. fig 12), à savoir l'**UE (Unité d'Exécution)** et l'**UIB (Unité d'Interfaçage avec le Bus)**. Le rôle de l'**UIB** est de gérer tous les transferts de données et d'adresses sur les bus pour l'UE. Cette unité envoie des adresses, récupère les instructions de la mémoire, lit les données des ports et de la mémoire et écrit des données dans les ports et la mémoire. L'**UE** exécute les instructions qui lui sont transmises par l'**UIB**.

Avec **le microprocesseur 8085**, le traitement des instructions se passait comme suit :

- Extraction des instructions par l'**UIB**.
- Exécution des instructions.
- Extraction des nouvelles instructions.

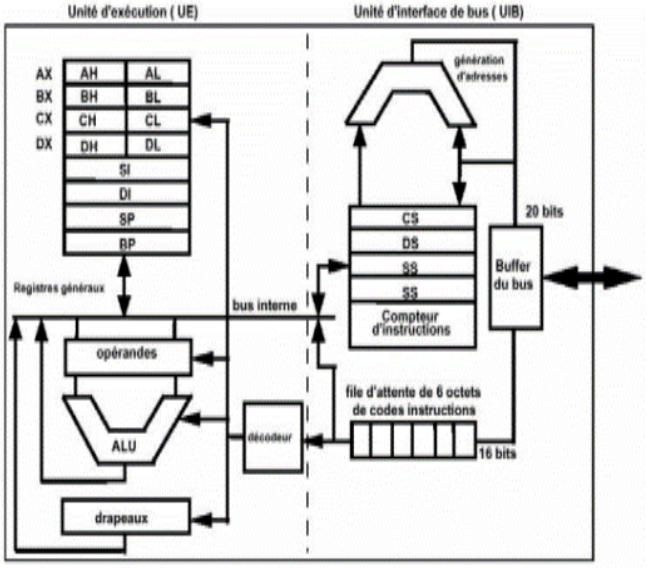


Figure 12: Architecture interne du 8086

Lorsque l'exécution d'une instruction est terminée, l'**UE** reste inactif un court instant, pendant que l'**UIB** extrait l'instruction suivante. Pour remédier à ce temps d'attente, le prétraitement ou traitement pipeline a été introduit dans le **8086**. Pendant que l'**UE** exécute les informations qui lui sont transmises, l'instruction suivante est chargée dans l'**UIB**. Les instructions qui suivront sont placées dans **une file d'attente**. Lorsque l'**UE** a fini de traiter une instruction l'**UIB** lui transmet instantanément l'instruction suivante, et charge la troisième instruction en vue de la transmettre à l'**UE**. De cette façon, l'**UE** est continuellement en activité.

On peut dire que le **8086** se compose essentiellement de deux unités : **la UIB** qui fournit l'interface physique entre le microprocesseur et le monde extérieur et l'**UE** qui comporte essentiellement l'**UAL de 16 bits** qui manipule les registres généraux de **16 bits**.

Remarque :

La file d'attente d'instructions contient des informations qui attendent d'être traitées par l'**UE**. La file d'attente est parfois appelée **capacité de traitement**. **Le microprocesseur 8086** est capable de mémoriser jusqu'à **six octets**. Les microprocesseurs actuels sont bien entendu équipés d'une file d'attente plus rapide et plus large, c'est à dire capable d'emmagasiner plus d'informations.

3. Registres du 8086

Le jeu de registres contient l'ensemble des registres du microprocesseur (cf. fig 13). Un registre est une petite partie de mémoire intégrée au microprocesseur, dans le but de recevoir des informations spécifiques, notamment des adresses et des données stockées durant l'exécution d'un programme. Il existe plusieurs types de registres. Certains d'entre eux sont affectés à des opérations d'**ordre général** et sont

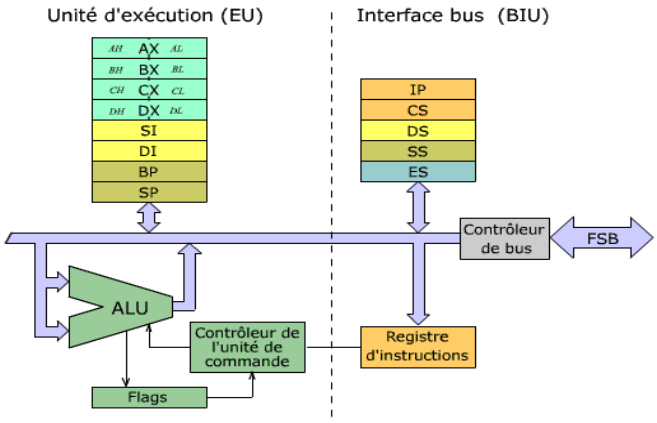


Figure 13: Registres du 8086

accessibles au programmeur à tout moment. Nous disons alors qu'il s'agit de **registres généraux**. D'autres registres ont des rôles bien plus spécifiques et ne peuvent pas servir à un usage non spécialisé.

3.1 Registres généraux

Les registres généraux (cf. fig 14) peuvent être utilisés dans toutes les opérations arithmétiques et logiques que le programmeur insère dans le code assembleur. Un registre complet présente une grandeur de **16 bits** comme le montre la figure, chaque registre est en réalité divisé en deux registres distincts de **8 bits**. De cette façon, nous pouvons utiliser une partie du registre si nous désirons y stocker une valeur n'excédant pas **8 bits**. Si, au contraire, la valeur que nous désirons y ranger excède **8 bits**, nous utiliserons le registre complet, c'est à dire **16 bits**.

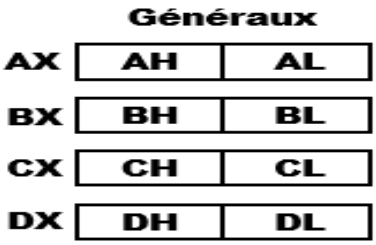


Figure 14: Registres généraux

Registre AX (Accumulateur) : Toutes les opérations de transferts de données avec les entrées-sorties ainsi que le traitement des chaînes de caractères se font dans ce registre, **de même les opérations arithmétiques et logiques. Il est obligatoire pour la multiplication et la division**. Les conversions en BCD du résultat d'une opération arithmétique (addition, soustraction, multiplication et la division) se font dans ce registre.

Registre BX (Registre de base) : Il est utilisé pour l'adressage de données dans une zone mémoire différente de la zone code : en général il contient une adresse de décalage par rapport à une adresse de référence) (Par exemple, l'adresse de début d'un tableau). Par défaut, **son offset est relatif au segment DS**. De plus il peut servir pour la conversion d'un code à un autre.

Registre CX (Le compteur) : Lors de l'exécution d'une boucle on a souvent recours à **un compteur de boucles** pour compter le nombre d'itérations, le registre CX a été fait pour servir comme compteur lors des instructions de boucle. **Le registre CL** sert en tant que compteur pour les opérations **de décalage et de rotation**, dans ce cas il va compter le nombre de décalages (rotation) de bits à droite ou à gauche.

Registre DX (Registre de donnée) : On utilise **le registre DX** pour les opérations de **multiplication et de division** (exemple : dans la multiplication et la division 16 bits, il sert comme extension au registre AX pour contenir un nombre 32 bits) mais surtout pour contenir le numéro d'un port d'entrée/sortie pour adresser les interfaces d'E/S.

3.2 Registres d'adressage ou pointeurs et d'index

Ces registres (cf. fig 15) sont plus spécialement adaptés au traitement des éléments dans la mémoire. Ils sont en général munis Zde propriétés **d'incrémentation** et **de décrémentation**. Un cas particulier de pointeur est le pointeur de pile (**Stack Pointer : SP**). Ce registre permet de pointer la pile pour stocker des données ou des adresses selon le principe du "**Dernier Entré Premier Sorti**" ou "**LIFO**" (**Last In First Out**).

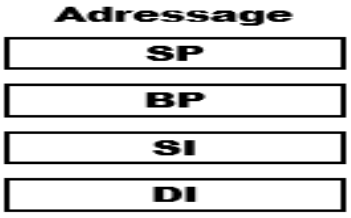


Figure 15: Registres d'adressage

L'index SI (Source Index) : Il permet de pointer la mémoire il forme en général un décalage (un offset) par rapport à une base fixe (**le registre DS**), il sert aussi pour les instructions de chaîne de caractères, en effet **il pointe sur le caractère source**.

L'index DI (Destination Index) : Il permet aussi de pointer la mémoire il présente un décalage par rapport à une base fixe (**DS** ou **ES**), il sert aussi pour les instructions de chaîne de caractères, **il pointe alors sur la destination**.

Les pointeurs SP et BP (Stack Pointer et Base Pointer) : Ils pointent sur la zone pile (une zone mémoire qui stocke l'information avec le principe **LIFO**), ils présentent un décalage par rapport à la base (**le registre SS**). **Le registre SP** pointe sur **la tête de la pile**. Pour **le registre BP** il a un rôle proche de celui de **BX**, mais il est généralement utilisé avec **le segment de pile**.

3.3 Registres segment

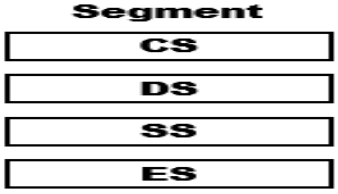


Figure 16: Registres de segment

Il y a quatre **registres de segment** dans le 8086 (cf. fig 16). Chaque registre de segment a une longueur de **16 bits** et chacun a un nom particulier : **CS (Code Segment)**, **DS (Data Segment)**, **ES (Extra Segment)** et **SS (Stack Segment)**. Les noms sont utiles pour comprendre comment les registres de segment sont utilisés pendant l'exécution du programme. Ces registres sont combinés avec d'autres registres internes lors de l'exécution du programme pour former une adresse système complète de 20 bits pour le bus d'adresse externe de la CPU. Chaque segment de mémoire ne peut excéder les **65535 octets**.

Le registre CS (code segment) : Il pointe sur le segment qui contient **les codes des instructions du programme en cours**. Il est un registre de **16 bits** contenant l'adresse d'un segment de **64 Ko** avec les instructions du processeur. Le processeur utilise le segment CS pour tous les accès aux instructions référencées par le registre de pointeur d'instruction (**IP**). Le registre CS ne peut pas être modifié directement. Si la taille du programme dépasse les **65535 octets** alors on peut diviser le code sur plusieurs segments (chacun ne dépasse pas les **65535 octets**) et pour basculer d'une partie à une autre du programme il suffit de changer la valeur du **registre CS** et de cette manière on résout le problème des programmes qui ont une taille supérieure à **65535 octets**.

Le registre DS (Data segment) : DS est un registre de **16 bits** contenant l'adresse d'un segment de 64 Ko avec les données du programme. Par défaut, le processeur suppose que toutes les données référencées par le registre général **BX** et le registre d'index (**SI**, **DI**) se trouvent dans le segment de données. Le registre segment de données pointe sur **le segment des variables globales du programme**, bien évidemment la taille ne peut excéder **65535 octets** (si on a des données qui dépassent cette limite, on utilise la même astuce citée dans la remarque précédente mais dans ce cas on change la valeur de **DS**).

Le registre ES (Extra segment) : Le registre de données supplémentaires **ES** est utilisé par le microprocesseur lorsque l'accès aux autres registres est devenu difficile ou impossible pour modifier des données, de même ce segment est utilisé pour **le stockage des chaînes de caractères**.

Le segment SS (Stack segment) : Le registre SS pointe sur **la pile**. Il est un registre de 16 bits contenant l'adresse d'un segment de **64 Ko** avec la pile de programme. Par défaut, le processeur suppose que toutes les données référencées par les registres du pointeur de pile (**SP**) et du pointeur de base (**BP**) se trouvent dans le segment de pile. Le registre **SS** peut être modifié directement à l'aide de l'instruction **POP**. **La pile** est une zone mémoire ou on peut sauvegarder les registres ou les adresses ou les données pour les récupérer après l'exécution d'un sous-programme ou l'exécution d'un programme d'interruption, en général il est conseillé de ne pas changer le contenu de ce registre car on risque de perdre des informations très importantes (exemple les passages d'arguments entre le programme principal et le sous-programme).

3.4 Registres de commande

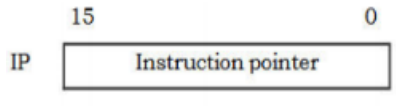


Figure 17: Registre IP

3.4.1 Le registre IP (Instruction Pointer) : Le **pointeur d'instruction** (IP) est un registre **16 bits** est analogue au **compteur de programme** (PC) dans les CPUs **8080/8085** (cf. fig 17).

Le pointeur d'instruction est mis à jour par l'UIB afin qu'il contienne l'offset de **la prochaine**

instruction à exécuter à partir du début du segment de code courant ; c'est-à-dire, IP pointe vers l'instruction suivante. **Le registre IP** est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante.

3.4.2 Le registre d'état (FLAGS) : Le registre d'état est utilisé pour contenir l'état de certaines opérations effectuées par le processeur. Par exemple, lorsque le résultat de l'opération est trop grand pour être inclus dans le registre cible (le registre qui doit contenir le résultat de l'opération), un bit spécifique (**bit OF**) du registre d'état est mis à 1 pour indiquer un débordement. **Les drapeaux (flags)** sont des indicateurs qui annoncent une condition particulière suite à une opération arithmétique ou logique. Le registre d'état du **8086** est formé par les bits suivants (cf. fig 18) : (**X** : bit non utilisé)

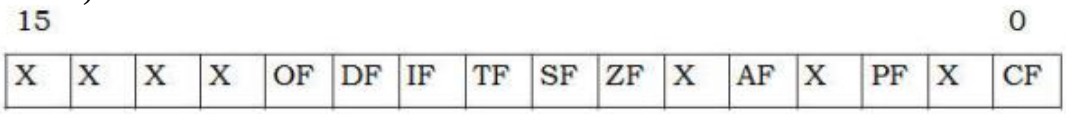


Figure 18: Registre d'état

CF (Carry Flag) : (**Retenue)** Ce drapeau est positionné à **1** lorsqu'il y a une retenue au bit de poids fort du résultat (**8** ou **16 bits**). Il intervient dans les opérations d'additions (**retenue**) et de soustractions (**borrow**) sur des entiers naturels. Il est positionné en particulier par les instructions ADD, SUB et CMP (comparaison entre deux valeurs). CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes.

Exemples (sur 8 bits pour simplifier) :

10010110	11011001
+ 01010100	+ 01010010
CF=0 11101010	CF=1 00101011

PF (Parity Flag) : (**Parité)** Si cet indicateur est positionné à **1**, le résultat de l'opération contient **un nombre pair de 1**. Il sera positionné à **0** si ce **nombre est impair**. L'indicateur de parité reflète uniquement la parité de la somme des bits de **l'octet de poids le plus faible** du résultat.

AF (Auxiliary Carry) : (**Demie retenue)** Ce bit est égal à **1** si on a **une retenue du quarter de poids faible dans le quarter de poids plus fort**.

ZF (Zero Flag) : (**Zéro)** Ce drapeau sera positionné à **1** si le résultat de l'opération sera **égal à zéro**. Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), **ZF=1** indique que les deux opérandes étaient égaux. Sinon, ZF est positionné à **0**.

SF (Sign Flag) : SF est positionné à **1** si le bit de poids fort du résultat d'une addition ou soustraction est **1** ; sinon **SF=0**. **SF** est utile lorsque l'on manipule des entiers signés, car **le bit de poids fort donne alors le signe du résultat**.

Exemples (sur **8 bits**) :

10010110	11011001
+ 01010100	+ 01010010
SF=1 11101010	SF=0 00101011

OF (Overflow Flag) : (Débordement) Ce bit indique un dépassement de capacité. Si on a un débordement arithmétique ce bit est mis à **1**, c-à-d le résultat d'une opération excède la capacité de l'opérande (registre ou case mémoire), sinon il est à **0**. Nous reconnaissons un dépassement de capacité lorsque nous faisons l'addition de deux nombres négatifs et le résultat est positif, ou bien lorsque nous faisons l'addition de deux nombres positifs et le résultat est négatif.

DF (Direction Flag) : (**Auto Incrémentation/Décrémentation)** utilisé pendant les instructions de chaîne de caractères pour auto incrémenter ou auto décrémenter le **SI** et le **DI.** Lorsqu'il est égal

à 1 ça veut dire une chaine de caractères est manipulée à partir de la plus haute adresse vers la plus faible adresse.

IF(Interrupt Flag) : (Masque d'interruption) pour masquer les interruptions venant de l'extérieur ce bit est mis à **0**, dans le cas contraire le microprocesseur reconnaît l'interruption de l'extérieur.

TF (Trap Flag) : Il met le CPU en **mode pas à pas (mode débogage)** pour faciliter la recherche des défauts d'exécution.

Remarque : Les instructions de branchements conditionnels utilisent les indicateurs (drapeaux), qui sont des bits spéciaux positionnés par l'UAL après certaines opérations. Chaque indicateur est manipulé individuellement par des instructions spécifiques.

4. Segmentation de la mémoire 8086

En respectant **l'architecture de Von Neumann**, le **8086** fonctionne avec une mémoire unique qui regroupe les instructions et les données. Certains des registres sont utilisés pour pointer le début de chaque partie en mémoire. On retrouvera : 1 segment de code (**CS**) ; 2 segments de données (**DS**, **ES**) ; 1 segment pour la pile (**SS**).

L'espace mémoire adressable par le 8086 est de 2^{20}= 1 048 576 octets = 1 Mo (20 bits du bus d'adresses). Il est divisé en quatre segments logiques allant jusqu'à **64 KOctets** chacun (cf. fig 19). L'accès à ces espaces est direct et simultané, or **le compteur de programme** est de 16 bits donc la possibilité d'adressage est de $\mathbf{2^{16}}$ = **64 Ko** (ce qui ne couvre pas la totalité de la mémoire), alors on utilise deux registres pour indiquer une adresse au processeur. Chaque **segment** débute à l'endroit spécifié par le registre segment. **Le déplacement (offset)** à l'intérieur de chaque segment se fait par **un registre de décalage** qui permet de trouver une information à l'intérieur du segment. **Exemple** la paire de registre **CS:IP** : pointe sur le code d'une instruction (**CS** registre segment et **IP** Déplacement).

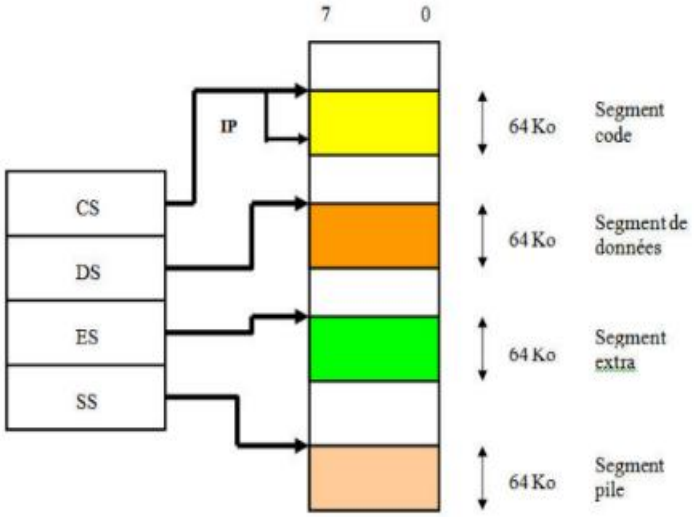


Figure 19: Segmentation de la mémoire 8086

Nous venons de voir qu'en assembleur, les données étaient normalement regroupées dans une zone mémoire nommée **segment de données**, tandis que les instructions étaient placées dans **un segment d'instructions** (de même pour **le segment pile** et **segment de données supplémentaires**). Ce partage se fonde sur la notion plus générale de segment de mémoire, qui est à la base du mécanisme de gestion des adresses par les processeurs 80x86. On a vu aussi que le registre **IP**, qui stocke l'adresse d'une instruction, fait lui aussi **16 bits**. Or, avec **16 bits** il n'est possible d'adresser que $\mathbf{2^{16}}$ = **64 Kilo octets**. Le bus d'adresses du 8086 possède **20 bits**. Cette adresse de **20 bits** est formée par la juxtaposition d'un registre segment (16 bits de poids fort) et d'un déplacement (offset, 16 bits de poids faible) :

Adresse physique = Segment * 16+ Offset

Le schéma de la figure suivante (cf. fig 20) illustre la formation d'une adresse **20 bits** à partir du segment et du déplacement sur **16 bits** :

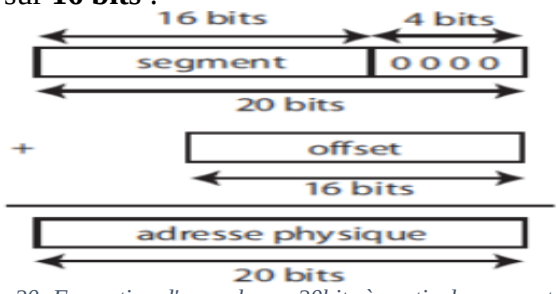


Figure 20: Formation d'une adresse 20bits à partie du segment et du déplacement sur 16 bits

Exemple : Pour CS = 1000h et IP = 2006h, Adresse physique = 1000**0**h + 2006h = 12006h

Les registres utilisables pour l'adressage sont associés, par défaut, à un des quatre registres de segment comme suit :

Registre Segment	Registres associés
CS	**IP**
SS	**BP,** SP
DS	**Valeur, DI, SI, BX**
ES	

Figure 21: Association entre registres d'adressage et registres de segment

Un programme assembleur peut au plus manipuler quatre segments de **64ko** donc au total une taille mémoire de **256ko**. Certains programmes ne nécessitent pas une telle taille, ils exigent **une taille plus petite**. Pour ce cas, il y a la possibilité de faire **chevaucher** les segments, c'est-à-dire que les segments vont avoir **des parties communes** de telle sorte à pouvoir réduire la taille de la mémoire utilisée (tous les segments). Nous pouvons faire recouvrir tous les segments et n'utiliser qu'un seul segment, c'est le cas des programmes dits **COM**[3].

Si nous divisons les **1MO** de la mémoire sur **64ko**, nous obtenons **16 segments** distincts ce qui fait que **4 bits** uniquement sont nécessaire pour définir les segments. Les concepteurs de Intel ont proposé des adresses segments non pas sur **4 bits** mais sur **16 bits**, ceci permet d'avoir $\mathbf{2^{16}}$ combinaisons différentes de segments. Cette technique va permettre de faire **chevaucher** les segments. La segmentation de la mémoire offre plusieurs avantages ; premièrement la séparation des données et des programmes ; deuxièmes l'application d'un même programme sur des données différentes ; troisièmement la réduction de la taille du code des instructions ; quatrièmement le fonctionnement en multitâches qui permet au processeur d'exécuter plusieurs programmes simultanément ; cinquièmement l'utilisation sans risque de l'adressage direct.

Fondamentalement, le processeur 8086 ne conserve pas les données en mémoire **dans l'ordre auquel nous nous attendons**. Au lieu de cela, les données sont stockées **à l'envers** octet par octet. En effet, un seul octet sera stocké en mémoire à **l'adresse prévue**. Mais un mot (deux octets) sera stocké en mémoire avec **l'octet de poids faible** à l'adresse du mot en mémoire, et avec **l'octet de poids fort** à l'adresse suivante. Par exemple, pour transférer le contenu d'un registre 16 bits vers la mémoire[**registres 16bit → mémoire**], deux octets successifs sont utilisés (**poids faible en premier**, **poids fort en suite**). Cette convention est dite : "**Little Indian**". L'inverse est appelé "**big-endian**" et est utilisé par d'autres processeurs. La taille peut être précisée dans l'instruction : «**BYTE PTR**» → 8 bits ou «**WORD PTR**» → 16 bits.

Exemple : INST **taille** [adr] , valeur

[3] Les programmes COM ont l'extension .com dans les ordinateurs.

5. Implémentation de la pile

La pile est une zone de mémoire qui fonctionne en mode **LIFO** (**Last In First Out** : **dernier entré, premier sorti**). Il y a deux opérations possibles sur la pile : **empiler** une donnée (mettre la donnée en haut de la pile) et **dépiler** une donnée (lire la donnée se trouvant en haut de la pile). Le sommet de la pile est repéré par un registre appelé pointeur de pile (**SP : Stack Pointer**). En combinaison avec **le segment de pile SS**, il pointe vers la dernière donnée empilée (**TOS : top of stack**) en mémoire. Le rôle du **pointeur de pile SP** (et de la pile vers laquelle il pointe) est le suivant. Quand un processeur exécute une instruction, il est possible qu'il soit interrompu par une "**Interruption**" (c'est-à-dire un appel au processeur qui est prioritaire aux instructions du programme qu'il traite). Il doit alors arrêter de s'occuper de l'instruction qu'il traite présentement pour s'occuper de l'interruption. Quand l'interruption sera traitée, il retournera à l'instruction qu'il traitait quand il a été interrompu. Mais pour cela, il doit se rappeler de cette instruction ainsi que de l'état de certains registres au moment où il traitait l'instruction. Donc pour ne pas les perdre, il les placera temporairement dans **une pile** (à l'intérieur de la mémoire RAM) et pourra les récupérer une fois l'interruption traitée. **Le pointeur de pile (SP)** donne donc l'adresse en mémoire de cette pile temporaire. **La pile** offre un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker **temporairement** des valeurs. Le schéma suivant montre comment une valeur est stocker dans la pile (**PUSH**) et comment elle est récupérée (**POP**) (cf. fig 22) :

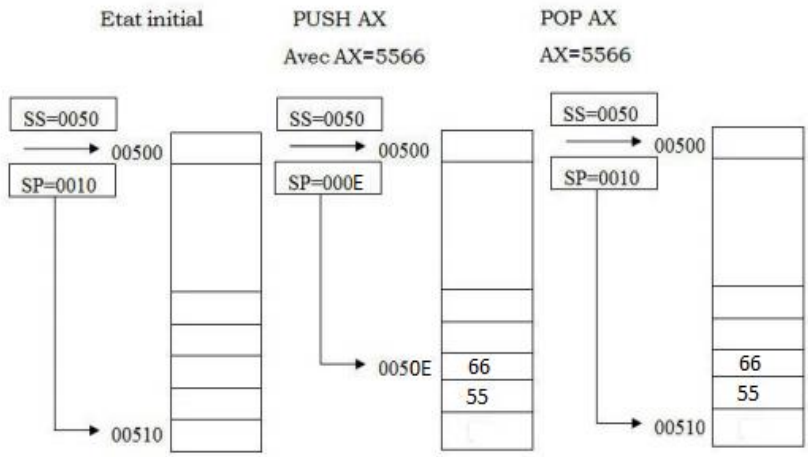


Figure 22: Fonctionnement des instructions PUSH et POP

6. Exercices

Exercice 1 :

Donnez **les adresses effectives** (**physiques**) à partir des adresses segmentées (tout en hexadécimal):

Adresse segmentée	Adresse effective
06D1:000D 059E:0000 0700:00FB FFFF:0005	

Où commence et où se termine chaque segment ?

Solution :

Adresse segmentée	**Adresse effective**	**Adresse du commencement du segment**		**Adresse de la fin du segment**	
		Adr seg	**Adr eff**	**Adr seg**	**Adr eff**
06D1:000D	06D1D	06D1:0000	06D10	06D1:FFFF	16D0F
059E:0000	059E0	059E:0000	059E0	059E:FFFF	159DF
0700:00FB	070FB	0700:0000	07000	0700:FFFF	16FFF
FFFF:0005	FFFF5	FFFF:0000	FFFF0	FFFF:FFFF ✗ FFFF:000F ✓	10FFEF ✗ FFFFF ✓

Exercice 2 :

Donnez **deux adresses segmentées** pour chaque adresse réelle (effective) (tout est en hexadécimal):

Adresse effective	Adresse segmentée 1	Adresse segmentée 2
FFFFF		
08A9F		

Solution :

Adresse effective	Adresse segmentée 1	Adresse segmentée 2
FFFFF	FFFF:000F	FFFE :001F
08A9F	0700:1A9F	0800 :0A9F

Conclusion

Le microprocesseur **8086** est une version améliorée du microprocesseur **8085** qui a été conçu par Intel en **1976**. Il s'agit d'un microprocesseur **16 bits** ayant **20 lignes d'adresse** et **16 lignes de données** qui fournit jusqu'à **1 Mo** de stockage. L'architecture du microprocesseur 8086 est composée de 2 unités principales, **l'UIB**, c'est-à-dire l'unité d'interface de bus et **l'UE**, c'est-à-dire l'unité d'exécution. Il contient un ensemble d'instructions puissant, qui fournit facilement des opérations telles que la multiplication et la division.

Chapitre IV : Programmation en langage d'assemblage du CPU 8086

Introduction

Le langage de programmation assembleur est **un langage de bas niveau** qui est développé en utilisant des mnémoniques. Le microprocesseur ne peut comprendre que **le langage binaire** comme les **0** ou les **1**, donc l'assembleur convertit le langage assembleur en langage binaire et le stocke dans la mémoire pour effectuer les tâches.
Dans le présent chapitre, nous utiliserons les informations vues dans les chapitres précédents pour **en savoir plus** sur le 8086. En effet, nous présentons les modes d'adressage, et la syntaxe d'une instruction et d'une directive dans **un code source** du 8086.

1. Modes d'adressage du 8086

Les instructions et leurs opérandes (paramètres) sont stockées en mémoire principale. La taille totale d'une instruction (**nombre de bits nécessaires pour la représenter en mémoire**) dépend du type d'instruction et aussi du type d'opérande. Chaque instruction est toujours codée sur **un nombre entier d'octets**, afin de faciliter son décodage par le processeur.
Une instruction est composée de deux champs :

- **Le code opération**, qui indique au processeur quelle instruction réaliser
- **Le champ opérande** qui contient la donnée, ou la référence à une donnée en mémoire (son adresse).

Les façons de désigner les opérandes constituent les "**modes d'adressage**". Selon la manière dont l'opérande (la donnée) est spécifié, c'est à dire selon le mode d'adressage de la donnée, une instruction sera codée par **1, 2, 3 ou 4 octets**.
Le microprocesseur 8086 possède **7 modes d'adressage** :

- **Adressage registre** → INST R , R
- **Adressage Immédiat** → INST R , im
- **Adressage direct** → INST R , [adr]
- **Adressage indirect** → INST R , Rseg : [Roff]
- **Adressage Basé** → INST R , Rseg : [Rb+dep]
- **Adressage Indexé** → INST R , Rseg : [Ri+dep]
- **Adressage Basé Indexé**→ INST R , Rseg : [Rb+Ri+dep]

1.1 Adressage registre

L'opération se fait sur **un ou 2 registres** : **INST R** ou **INST R , R**

Exemples :

INC AX ; incrémenter le registre AX (AX++)
MOV AX, BX ; copier le contenu de BX dans AX (AX<-BX)
MOV AX, BX ; cela signifie que l'opérande stocké dans le registre **BX** sera transféré vers le registre **AX**. Quand on utilise l'adressage registre, le microprocesseur effectue toutes les opérations d'une façon interne. Donc dans ce mode il n'y a pas d'échange avec la mémoire, ce qui augmente **la vitesse de traitement de l'opérande**.

1.2 Adressage immédiat

L'opérande est **une constante** (**valeur**) qui fait partie de l'instruction : **INST R, IM** ou **INST IM**

Exemples :

MOV AX, 243 ; charger le registre AX par le nombre décimal 243 (AX<- 243)
JMP 008 ; saut à l'instruction du numéro 008

MOV AL, 'A' ; charger le registre AL par le code ASCII du caractère 'A' (65)
MOV AX, 'A' ; charger le registre AH par 00 et le registre AL par le code ASCII du caractère 'A'
MOV AX, 'AB' ; charger AH par le code ASCII du caractère 'A' et AL par le code ASCII du caractère 'B' (66)

1.3 Adressage direct

Un des deux opérandes se trouve **en mémoire**. L'adresse de la case mémoire est précisé **directement** dans l'instruction : **INST R, [adr]** ou **INST [adr], R** ou **INST taille [adr], im**
L'adresse doit être placée entre crochets **[R_{seg}: adr]**. Si le segment (R_{seg}) n'est pas précisé, **DS** est pris par défaut.

Exemples :
MOV AX, [243] ; copier le contenu de la mémoire d'adresse DS:243 dans AX
MOV [123], AX ; copier le contenu de AX dans la mémoire d'adresse DS:123
MOV AX, [SS:243] ; copier le contenu de la mémoire SS:243 dans AX

1.4 Adressage Indirect

Un des deux opérandes se trouve **en mémoire**. L'adresse se trouve dans l'un de ces **4** registres : **BX**, **BP**, **SI** ou **DI :**

INST R, [R_{seg}: R_{off}] ou **INST [R_{seg}: R_{off}], R** ou **INST taille $\left[R_{seg}: R_{off}\right]$, im**

Si R_{seg} n'est pas spécifié le segment par défaut sera utilisé :

Registre	BX	BP	SI	DI
Segment par défaut	DS	SS	DS	DS

Exemples :
MOV AX, [BX] ; charger AX par le contenu de la mémoire d'adresse DS:BX
MOV AX, [BP] ; charger AX par le contenu de la mémoire d'adresse SS:BP
MOV AX, [SI] ; charger AX par le contenu de la mémoire d'adresse DS:SI
MOV AX, [ES:BP] ; charger AX par le contenu de la mémoire d'adresse ES:BP
L'adressage indirect est divisé en **3** catégories selon le registre d'offset utilisé: **l'adressage Basé**, **l'adressage indexé** et **l'adressage basé indexé** :

Mode d'adressage	Basé	Indexé	Basé Indexé
Registres utilisés	BX BP	DI SI	BX, DI BX, SI BP, DI BP, SI

1.5 Adressage Basé

L'offset se trouve dans l'un des registres de base **BX** ou **BP**. On peut préciser **un déplacement** qui sera ajouté au contenu de R_b (registre de base) pour déterminer l'offset :

INST R, [R_{seg}: R_b+dep] ou **INST [R_{seg}: R_b+dep], R** ou **INST taille [R_{seg}: R_b+dep], im**

Exemples :
MOV AX, [BX] ; charger le contenu de la mémoire d'adresse DS:BX
MOV AX, [BX+5] ; charger AX par le contenu de la mémoire d'adresse DS:BX+5
MOV AX, [BP-200] ; charger AX par le contenu de la mémoire d'adresse SS:BP-200
MOV AX, [ES:BP] ; charger AX par le contenu de la mémoire d'adresse ES:BP

Remarque : Les syntaxes suivantes sont identiques : **MOV AX, [BX+2] MOV AX, [BX]+2 MOV AX, 2[BX]**

1.6 Adressage Indexé

L'offset se trouve dans l'un des deux registres d'index **SI** ou **DI**. On peut préciser **un déplacement** qui sera ajouté au contenu de $\boldsymbol{R_i}$ (registre d'index) pour déterminer l'offset :

INST R, [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_i}$+dep] ou **INST [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_i}$+dep], R** ou **INST taille [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_i}$+dep], im**

Exemples :

MOV AX, [SI] ; charger AX par le contenu de la mémoire d'adresse DS:SI
MOV AX, [SI+500] ; charger AX par le contenu de la mémoire d'adresse DS:SI+500
MOV AX, [DI-8] ; charger AX par le contenu de la mémoire d'adresse DS:DI-8
MOV AX, [ES:SI+4] ; charger AX par le contenu de la mémoire d'adresse ES:SI+4

1.7 Adressage Basé Indexé

L'offset de l'adresse de l'opérande est **la somme d'un registre de base**, **d'un registre d'index** et **d'un déplacement** optionnel. Si $\boldsymbol{R_{seg}}$ n'est pas spécifié, le segment par défaut **du registre de base** est utilisé : **INST R, [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_b + R_i}$+dep]** ou **INST [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_b + R_i}$+dep], R** ou **INST taille [$\boldsymbol{R_{seg}}$: $\boldsymbol{R_b}$+$\boldsymbol{R_i}$+dep], im**

Exemples :

MOV AX, [BX+SI] ; AX est chargé par la mémoire d'adresse DS:BX+SI
MOV AX, [BX+DI+5] ; AX est chargé par la mémoire d'adresse DS:BX+DI+5
MOV AX, [BP+SI-8] ; AX est chargé par la mémoire d'adresse SS:BP+SI-8
MOV AX, [BP+DI] ; AX est chargé par la mémoire d'adresse SS:BP+DI

2. Code source

Comme tout programme, un programme écrit en assembleur (programme source) comprend **des définitions**, **des données** et **des instructions**, qui s'écrivent chacune sur une ligne de texte. Les données sont déclarées par **des directives**, mots clefs spéciaux que comprend l'assembleur (donc ils sont destinés à l'assembleur). **Les instructions** sont destinées au microprocesseur.

2.1 Instructions

La syntaxe des instructions est comme suivie :

{Label :} mnémonique {opérande} { ; commentaire}

Le champ Label (étiquette)

Il est optionnel et représente l'adresse de l'instruction. Il est destiné pour marquer une ligne qui sera la cible d'**une instruction de saut ou de branchement**. Un label peut être formée par 31 caractère alphanumérique ({A…Z} {a… z} {0...9}) au maximum. Les noms des registres ainsi que la représentation mnémonique des instructions et les directives (voir plus loin) ne peuvent être utilisées comme Label. **Le champ Label doit se terminer par ' : '** et ne peut commencer par un chiffre. De même il n'y a pas de distinction entre minuscules et majuscules.

Le champ mnémonique

Un mnémonique est, généralement, composé uniquement de lettres. Souvent c'est la compressions d'un mot ou d'une expression en anglais présentant l'action de l'instruction. Quelques mnémoniques que nous retrouverons souvent : **MOV**, **CMP**, **LOOP**, ... Par exemple MUL permet la multiplication. (MULtiply). On pourra indifféremment écrire les mnémoniques avec des lettres minuscules ou majuscules.

Le champ opérande

Les opérandes d'une instruction indiquent les données à traiter sur lesquelles nous voulons agir :

- Soit sous la forme de **valeurs constantes** (valeurs immédiates),
- Soit en spécifiant **l'adresse mémoire** (l'emplacement en mémoire) où se trouve la donnée,
- Soit **le nom d'un registre** contenant la donnée ou contenant **l'adresse mémoire de la donnée**.

En général, dans une instruction : **Mnémonique op1, op2**

op1 : représente la destination. C'est l'endroit où le résultat sera stocké.

op2 : représente la source. C'est la donnée qui sera éventuellement combiné avec op1 pour calculer le résultat.

Dans la plupart des instructions, la source et la destination doivent être de même taille.
Nous pouvons trouver de instructions à opérande implicite, exemple : **MUL BH**. Aussi, nous ne pouvons pas utiliser tous les types d'opérandes n'importe où. Les types autorisés selon l'instruction et nombres d'opérandes sont donnés dans la partie « **Jeu d'instructions du 8086** ».

Le champs commentaire

Un commentaire commence par un caractère **;** et se termine en fin de ligne. Il est **optionnel** mais très intéressant lorsque on programme en assembleur en raison **du nombre d'instructions qui est assez élevé**, ce qui va rendre **la compréhension des programmes** assez délicate et difficile. Pour cette raison, lorsque on programme en assembleur il vaut mieux mettre des commentaires pour que le programme soit **lisible pour les utilisateurs**.

Exemple d'une instruction

ET1 : MOV AX , 200h ; mettre la valeur 200h dans le registre AX

2.2 Directives

Pour programmer en assembleur, on doit utiliser, en plus des instructions assembleur, **des directives** ou **pseudo-instructions**. **Une directive** est une information que le programmeur fournit au compilateur. Elle n'est pas transformée en une instruction en langage machine. Elle n'ajoute donc **aucun octet au programme compilé**. Donc **les directives** sont des déclarations qui vont guider l'assembleur.
Une directive est utilisée par exemple pour **créer de l'espace mémoire** pour des variables, pour **définir des constantes**, etc... Pour déclarer une directive il faut utiliser la syntaxe suivante :

{Nom} Directive {opérande} { ; commentaire}

- **Le champ opérande** dépend de la directive.
- **Le champ commentaire** a les mêmes propriétés vues dans le paragraphe précèdent.
- **Le champ Nom** indique le nom des variables : c'est un champ optionnel (selon la directive).

Les constantes numériques

- **100110b** est une constante binaire dont la valeur décimale est **38**.
- **37o** est une constante octale dont la valeur décimale est **31**.
- **4ch** est une constante hexadécimale dont la valeur décimale est **76**.

Puisqu'une constante numérique doit toujours commencer par un chiffre, une constante hexadécimale dont le premier caractère est une lettre sera précédée d'un **0**. Ainsi, la valeur **c3** sera notée **0c3h** dans un source assembleur.

Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entre **'**. Si on veut mettre un caractère ' dans une constante, il suffit de la doubler.

Exemples :

- 'hello world'
- 'l''arbre'

Déclaration d'une constant

La directive EQU associe une valeur à un symbole qui pourra être ensuite utilisé dans le programme à la place de la constante qu'elle définit.

Syntaxe : nom EQU constante

Par exemple, la commande :

TROIS EQU 3

définit une constante qui s'appelle **TROIS** dont la valeur est **3**. Une fois cette déclaration effectuée, toute occurrence de l'identificateur **TROIS** sera remplacée par la valeur indiquée.

Déclarations de variables

L'espace mémoire est utilisé pour **stocker des constantes** (chaînes de caractères, valeurs numériques, ...) ou **des variables**. Avant d'employer une variable, il faut préalablement la **déclarer** (comme dans tout langage). Déclarer une variable (et ceci est vrai aussi dans les langages de haut niveau comme Pascal, mais c'est un peu caché) revient toujours à **réserver un emplacement en mémoire** pour y stocker la valeur de cette variable.

Les directives DB/DW/DD

Ces directives sont utilisées pour **déclarer les variables** : L'assembleur attribue à chaque variable **une adresse**. Dans le programme, on repère les variables grâce à leurs noms. Les noms des variables sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre. Le nom peut comporter **des majuscules**, **des minuscules**, **des chiffres**. Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

DB (Define byte):

La directive DB définit une variable de **8 bits** : c-à-d elle réserve **un espace mémoire d'un octet**, donc les valeurs qu'on peut stocker pour cette directive sont :

- comprises entre **0** et **255** (pour les nombres non signés),
- et de **-128** jusqu'à **127** pour les nombres signés.

Syntaxe : [nom] DB constante [, constante]

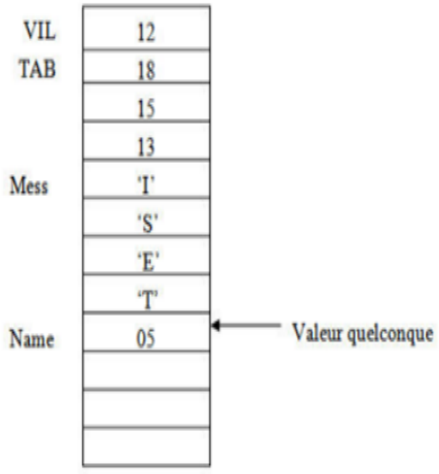


Figure 23: Utilisation de la directive DB

Exemples : (voir fig 23)

Vil DB 12H ; Définit une variable (un octet) de valeur Initiale 12

Tab DB 18H, 15H, 13H ; définit un tableau de 3 cases ;(3 octet) qui démarre à partir de l'adresse ;TAB

Mess DB 'ISET' ; définit aussi un tableau mais les valeurs de ; chaque case n'est autre que le code ascii de chaque lettre

Name DB ? ; définit une variable 8 bits de valeur initiale quelconque

DW (define word):

La directive DW définit une variable de **16 bits** : c-à-d elle réserve un espace mémoire d'un mot, donc les valeurs qu'on peut stocker pour cette directive sont comprises :

- entre **0** et **65535** (pour les nombres non signés),
- et de **-32768** jusqu'à **32767** pour les nombres signés.

Syntaxe : [nom] DW constante [, constante]

Exemple : (voir fig 24)

TT1 DW 500H ; réserve deux cases mémoire (un mot) a partir de l'adresse TT1

TAB1 DW 10H,11H,14H ; réserve un tableau de 6 cases chaque valeur sera mise sur deux cases

YY DW ? ; réserve un mot dans la mémoire de valeur initial quelconque.

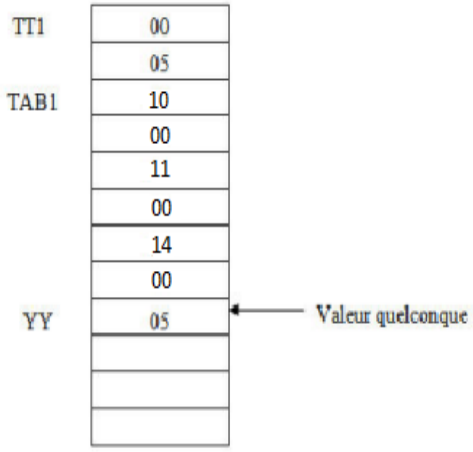


Figure 24: Utilisation de la directive DW

DD (Define Double) :

La directive DD réserve un espace mémoire de **32 bits** (**4 cases mémoire** ou **4 octets**)

Syntaxe : [nom] DD constante [, constante]

Exemple : ff DD 15500000H

Directive dup :

Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive **dup**.

Exemple :

tab DB 100 dup (15) ; 100 octets valant 15

y DW 10 dup (?) ; 10 mots de 16 bits non initialisés

Directives Word PTR et Byte PTR

Dans certains cas, **l'adressage mémoire est ambigu**. Par exemple, si l'on écrit :

MOV [BX], 0 ; range 0 a l'adresse spécifiée par BX

L'assembleur ne sait pas si l'instruction concerne **1**, **2** ou **4 octets** consécutifs. Afin de lever l'ambiguïté, nous devons utiliser une directive spécifiant **la taille de la donnée** à transférer :

MOV byte ptr [BX], 0 ; concerne 1 octet

MOV word ptr [BX], 0 ; concerne 1 mot de 2 octets

ORG :

(Ex.: **ORG 100h**): indique au compilateur que le programme doit être chargé à l'adresse **CS:100h** (par défaut pour les programmes .COM du i8086).

ASSUME :

Elle permet de définir le nom logique d'un segment, par exemple, elle permet d'indiquer à l'assembleur quel est le segment de données et celui de codes.

Exemple : ASSUME CS : CODE

Cela indique à l'assembleur que le segment logique nommé **CODE** contient les instructions du programme et devrait être traité comme **un segment de code**.

x SEGMENT / ENDS :

Les directives SEGMENT et ENDS permettent de définir les segments de codes et de données (ou **.data** / **.code**).

Conclusion

Dans ce chapitre, nous avons abordé quelques **sujets importants** relatifs à la programmation du microprocesseur 8086. Nous avons discuté des informations sur le mode d'adressage du microprocesseur 8086 et chaque mode d'adressage a été expliqué avec **des exemples**.
Une instruction est directement traduite en quelque chose que le CPU peut **exécuter**. **Une directive** est quelque chose que l'assembleur peut **interpréter** et dit quelque chose sur la façon dont les instructions doivent être assemblées. Nous avons aussi présenté dans ce chapitre la syntaxe des instructions and directives pour la programmation en langage assembleur du processeur 8086.

Chapitre V : Jeu d'instructions du 8086

Introduction

Le jeu d'instructions est l'ensemble des **instructions-machine** qu'un processeur d'ordinateur peut **exécuter**. Ces instructions-machines permettent d'effectuer des opérations élémentaires ou plus complexes. Le jeu d'instructions définit quelles sont les instructions **supportées** par le processeur. On peut diviser les instructions du 8086 en **8 groupes** comme suit :

- Instructions de transfert et conversions de données (**MOV**, **XCHG**, **CBW**, …).
- Instructions relatifs aux drapeaux (Registre d'Etat) (**CLC**, **STC**, **CMC**, …).
- Instructions arithmétiques (**ADD**, **SUB**, **MUL**, **DIV**, ...).
- Instructions logiques (**NOT**, **AND**, **OR**, **XOR**, ...).
- Instructions de décalage (**SHR**, **SHL**, **ROL**, **ROR**, ...).
- Instructions de branchement (**JMP**, **Jxx**, **LOOP**....).
- Instructions de manipulation des tableaux ou chaines de caractères (**LODSB**, **LODSW**, **STOSB**, **STOSW**, …).
- Instructions d'interruption.

Dans ce chapitre, nous allons détailler la plupart des instructions de chacun de ces groupes. Nous expliquons aussi la méthode comment accéder à **la mémoire vidéo** et **les procédures et macros** en 8086.

Notation

Types d'opérandes :
REG : AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
SREG : DS, ES, SS, et seulement comme deuxième opérande : CS.
memory : [BX], [BX+SI+7], variable, etc...
immediate : 5, -24, 3Fh, 10001101b, etc...

Lorsque deux opérandes sont requis pour une instruction, ils sont séparés par une virgule. Par example:
REG, memory

Lorsqu'il y a deux opérandes, les deux opérandes doivent avoir la même taille (à l'exception des instructions de décalage et de rotation). Par exemple :

```
AL, DL
DX, AX
m1 DB ?
AL, m1
m2 DW ?
AX, m2
```

Certaines instructions autorisent plusieurs combinaisons d'opérandes. Par exemple :

```
memory,immediate
REG,immediate
memory,REG
REG, SREG
```

La plupart des instructions modifient les indicateurs du registre FLAGS. Un tableau indique l'effet de l'instruction couramment décrite sur ces indicateurs. Il a la forme suivante :

O	D	I	T	S	Z	A	P	C

La première ligne indique le nom des bits intéressants de FLAGS. La deuxième ligne (ici c'est vide) indique l'effet de l'instruction sur un bit en particulier.
On note :
* le bit est modifié en fonction du résultat de l'exécution de l'instruction.
? le bit a une valeur indéfinie après l'exécution de l'instruction.
1 le bit est mis à 1.
0 le bit est mis à 0.
Une case vide indique que l'indicateur n'est pas modifié par l'instruction.

1. Instructions de transfert et conversions de données

MOV

Syntaxe : MOV destination, source
Cette instruction copie l'opérande Source dans l'opérande Destination. Ceci correspond donc à l'affectation des langages de haut niveau A : = B.

REG, memory
memory, REG
REG, REG
Memory, immediate
REG, immediate
SREG, memory
memory, SREG
REG, SREG
SREG, REG

O	D	I	T	S	Z	A	P	C

L'instruction MOV ne peut pas :

- Modifier la valeur des registres **CS** et **IP**.
- Copier la valeur d'un registre de segment dans un autre registre de segment (doit d'abord copier dans un registre général).
- Copier une valeur immédiate dans un registre de segment (doit d'abord copier dans un registre général).

Algorithme :
operande1=operande2
Exemples :
MOV AX, BX ; Transfert d'un registre de 16 bits vers un registre de 16 Bits
MOV AH, CL ; Transfert d'un registre de 8 bits vers un registre de 8 bits
MOV AX, [Val1] ; Transfert du contenu d'une case mémoire d'adresse
; Val1 de 16 bits vers AX
MOV [Val2], AL ; Transfert du contenu du AL vers une case mémoire
; d'adresse Val2

Remarques :

Il est strictement interdit de transférer **le contenu d'une case mémoire vers une autre case mémoire** comme suit :

MOV [Val1], [Val2]

Pour remédier à ce problème on va effectuer cette opération sur **deux étapes** :

MOV AL, [Val2]
MOV [Val1], AL

On n'a pas le droit aussi de **transférer un registre segment vers un autre registre segment** sans passer par un autre registre :

MOV DS, ES

On va passer comme la première instruction :

MOV AX, ES
MOV DS, AX

Le CS n'est jamais utilisé comme registre destination.

LEA (Load Effective Address)

Elle transfert l'adresse **offset** (**décalage**) d'un opérande mémoire dans **un registre de 16 bits**. Cette commande a le même rôle que l'instruction **MOV** avec **offset : LEA BX, TAB_VAL** (c'est équivalent à **MOV BX, offset TAB_VAL**).

REG, memory

O	D	I	T	S	Z	A	P	C

Exemple 1 :

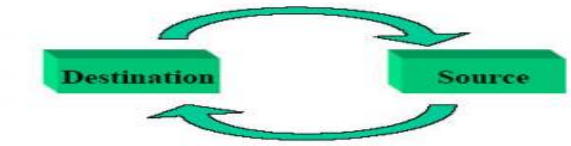
```
MOV BX, 35h
MOV DI, 12h
LEA SI, [BX+DI]    ; SI = 35h + 12h = 47h
```

Exemple 2 :

```
org 100h
LEA AX, m       ; AX = offset de m
RET
m  dw  1234h
END
```

XCHG

Elle permet de **commuter la source avec la destination** comme suit :




```
XCHG  AX,BX     ; elle permute entre AX et BX
Exemple AX=0123   et BX = 5678
Après l'exécution de l'instruction on aura :
AX=5678    et BX = 0123
```

REG, memory memory, REG REG, REG

O	D	I	T	S	Z	A	P	C

Algorithme :
operande1 <-> operande2
Exemple :
```
MOV AL, 5
MOV AH, 2
XCHG AL, AH   ; AL = 2, AH = 5
XCHG AL, AH   ; AL = 5, AH = 2
RET
```

PUSH

Syntaxe : PUSH SOURCE
Elle permet d'**empiler** une valeur de **16 bits** dans le haut de la pile.

REG
SREG
memory
immediate

O	D	I	T	S	Z	A	P	C

Algorithme :
SP = SP - 2
SS:[SP] (le haut de la pile) = operande
Exemple :
```
MOV AX, 1234h
PUSH AX
POP DX ; DX = 1234h
RET
```

POP

Syntaxe : POP destination
Elle permet de **dépiler** une valeur de **16 bits** sur le haut de la pile.

REG
SREG
memory

O	D	I	T	S	Z	A	P	C

Algorithme :
operande = SS:[SP] (le haut de la pile)
SP = SP + 2
Exemple :
```
MOV AX, 1234h
PUSH AX
POP  DX     ; DX = 1234h
RET
```

PUSHA

Empile tous les registres à usage général **AX, CX, DX, BX, SP, BP, SI, DI** dans la pile.

O	D	I	T	S	Z	A	P	C

Algorithme :
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH SP
PUSH BP
PUSH SI
PUSH DI

POPA

Dépile tous les registres à usage général **DI, SI, BP, SP, BX, DX, CX, AX** de la pile. La valeur **SP est ignorée**, elle est dépilée mais ne modifie pas le registre SP.

O	D	I	T	S	Z	A	P	C

Algorithme :
POP DI
POP SI
POP BP
POP xx (la valeur SP est ignorée)
POP BX
POP DX
POP CX
POP AX

CBW (convert byte to word)

Étend AL dans **AH** en respectant le signe.

O	D	I	T	S	Z	A	P	C

Algorithme :
Si le bit du poids fort de AL = 1 alors :

- AH = 255 (0FFh)

sinon

- AH = 0

Exemple :

```
MOV AX, 0   ; AH = 0, AL = 0
MOV AL, -5  ; AX = 000FBh (251)
CBW         ; AX = 0FFFBh (-5)
RET
```

CWD (convert Word to Double Word)

Étend AX dans **DX** en respectant le signe

O	D	I	T	S	Z	A	P	C

Algorithme :
Si le bit du poids fort de AX = 1 alors :

- DX = 65535 (0FFFFh)

sinon

- DX = 0

Exemple :
```
MOV DX, 0   ; DX = 0
MOV AX, 0   ; AX = 0
MOV AX, -5  ; DX AX = 00000h:0FFFBh
CWD         ; DX AX = 0FFFFh:0FFFBh
RET
```

2. Instructions relatifs aux drapeaux

***CLC* : (CLear Carry)** positionne **le drapeau C** (l'indicateur de retenue) à 0.
***STC* : (Set Carry)** positionne **le drapeau C** (l'indicateur de retenue) à 1.
***CMC* :** Complémente **le drapeau C** (l'indicateur de retenue).
***CLD* :** Positionne **le drapeau D** (l'indicateur de direction) à 0.
***STD* :** Positionne **le drapeau D** (l'indicateur de direction) à 1.
***CLI* :** Positionne **le drapeau I** (l'indicateur d'interruption) à 0.
***STI* :** Positionne **le drapeau I** (l'indicateur d'interruption) à 1.
***LAHF (Load AH from Flags)* :** Copie **l'octet bas** du registre d'état dans AH.

				O	D	I	T	S	Z		A		P		C
15				11	10	9	8	7	6		4		2		0

	7	6	5	4	3	2	1	0
(AH) ←	SF	ZF	X	AF	X	PF	X	CF

O	D	I	T	S	Z	A	P	C

***SAHF (Store AH into Flags)* : Opération inverse** de LAHF : Transfert AH dans **l'octet bas** du registre d'état.

O	D	I	T	S	Z	A	P	C
				*	*	*	*	*

***PUSHF* : empile** le registre d'indicateurs FLAGS. La valeur du pointeur de sommet de pile **SP** est ensuite diminuée de 2.

O	D	I	T	S	Z	A	P	C

Algorithme :
```
SP = SP - 2
SS:[SP] (le haut de la pile) = flags
```
***POPF* : dépile** la valeur du registre d'indicateurs FLAGS. La valeur du pointeur de sommet de pile **SP** est augmentée de 2

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

Algorithme :
```
flags = SS:[SP] (le haut de la pile)
SP = SP + 2
```

3. Instructions arithmétiques

***ADD* : addition** sans retenue [ADD AX , 5 signifie: AX ← AX + 5]
Syntaxe : ADD Destination, Source

Elle permet d'**additionner** le contenu de la source (octet ou un mot) avec celui de la destination, le résultat est mis dans la destination.

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
operande1 = operande1 + operande2
Exemple :
La retenue est positionnée en fonction du résultat de l'opération.
Par exemple, si les instructions suivantes sont exécutées :
MOV AX, 0A9h
ADD AX, 72h
alors le registre **AX** contient la valeur **1Bh**, **le bit C** est positionné à la valeur **1**, la retenue auxiliaire **A** est mise à **0**.
Si nous considérons les deux instructions suivantes :
MOV AX, 09h
ADD AX, 3Ah
alors le registre **AX** contient la valeur **43h**, **le bit C** est mis à **0**, la retenue auxiliaire **A** est mise à **1**.

***ADC* : addition** avec retenue [ADC AX , 5 signifie: AX ← AX + 5 + C]
Syntaxe : ADC Destination, source
Elle permet d'**additionner** le contenu de la source (octet ou un mot) avec celui de la destination et la retenue (**CF**) le résultat est mis dans la destination.

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
operande1 = operande1 + operande2 + CF
Exemple d'addition sur 32 bits :
Nous voulons additionner une donnée se trouvant dans les registres **AX** et **DX** à une donnée se trouvant dans les registres **BX** et **CX** et ranger le résultat dans les registres **AX** et **DX** (**AX** et **BX** contiennent les poids faibles, **DX** et **CX** les poids forts).
Le code est alors le suivant :
ADD AX, BX
ADC DX, CX
Exemples :
ADC AX, BX ; AX = AX + BX + **CF**(addition sur **16 bits**)
ADC AL, BH ; AL = AL + BH + **CF**(addition sur **8 bits**)

ADC AL, [SI] ; AL = AL + le contenu de la case mémoire pointé par SI + **CF**
ADC [DI], AL ; le contenu de la case mémoire pointé par DI
; est additionné avec AL + **CF**, le résultat est
; mis dans la case mémoire pointé par DI

***SUB* : soustraction** sans retenue [SUB AX , 3 signifie: AX ← AX – 3]
Syntaxe : SUB Destination, source
Elle permet de **soustraire** la destination de la source (octet ou un mot) le résultat est mis dans la destination.

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
operande1 = operande1 - operande2
Exemple :
La retenue est positionnée en fonction du résultat de l'opération
Par exemple, si les instructions suivantes sont exécutées ;
MOV AX, 39h
SUB AX, 18h
le registre **AX** contient ensuite la valeur **21h**. Les bits **Z**, **S** et **C** du registre **FLAGS** sont mis à **0** car le résultat **n'est pas nul**, **son signe est positif** et **aucune retenue n'est générée**.
Si nous considérons les deux instructions suivantes :
MOV AX, 26h
SUB AX, 59h
le registre **AX** contient ensuite la valeur **CDh**. Le bit **Z** est mis à **zéro**. Les bits **C**, **A** et **S** sont mis à **1**.
Exemples :
SUB AX, BX ; AX = AX - BX (Soustraction sur **16 bits**)
SUB AL, BH ; AL = AL - BH (Soustraction sur **8 bits**)
SUB AL, [SI] ; AL = AL - le contenu de la case mémoire pointé par SI
SUB [DI], AL ; le contenu de la case mémoire pointé par DI
; est soustraite de AL , le résultat est mis
; dans la case mémoire pointé par DI

***SBB* : soustraction** avec retenue [SBB AX , 3 signifie: AX ← AX - 3 - C]
Syntaxe : SBB Destination, source
Elle permet de **soustraire** la destination de la source et **la retenue** (octet ou un mot) le résultat est mis dans la destination.

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
operande1 = operande1 - operande2 – CF
Exemple :
STC
MOV AL, 5
SBB AL, 3; AL = 5 - 3 - 1 = 1
RET
Exemple de soustraction sur 32 bits :
Nous voulons soustraire une donnée se trouvant dans les registres **BX** et **CX** d'une donnée se trouvant dans les registres **AX** et **DX** et ranger le résultat dans les registres **AX** et **DX** (**AX** et **BX** contiennent les poids faibles, **DX** et **CX** les poids forts).
Le code est alors le suivant :
SUB AX, BX
SBB DX, CX
***INC* : Incrémentation** [INC AX signifie: AX ← AX + 1], ne touche pas le C
Syntaxe : INC Destination
Elle **ajoute 1** au contenu de l'opérande, sans modifier l'indicateur de retenue.

REG
memory

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

Algorithme :
operande = operande + 1
Exemple :
Soient les instructions assembleur suivantes :

mov al, 3fh
inc al
AL contient ensuite la valeur **40h**. **Le bit Z** est mis à **0** (le résultat de l'incrémentation n'est pas nul), l'indicateur de retenue auxiliaire **bit A** est mis à **1** (passage de retenue entre les bits **3** et **4** lors de l'incrémentation), les bits **bit O** et **bit S** mis à **0** (pas de débordement de capacité, le signe du résultat est positif).
***DEC* : Décrémentation** [DEC AX signifie: AX ← AX - 1], ne touche pas le C
Syntaxe : DEC Destination
Elle **soustrait 1** au contenu de l'opérande, sans modifier l'indicateur de retenue.

REG
memory

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

Algorithme :
operande = operande - 1

Exemple 1 :
Soient les instructions assembleur suivantes :
MOV AL, 01h
DEC AL
AL contient ensuite la valeur **00**. Les bits **Z et P** sont mis à **1**, les bits **O**, **A** et **S** sont mis à **0**.
Exemple 2 :
DEC AX ; AX = AX - 1 (décrémentation sur **16 bits**).
DEC AL ; AL = AL -1 (décrémentation sur **8 bits**).
DEC [SI] ; [SI] = [SI] - 1 le contenu de la case mémoire
; pointé par SI sera décrémenter
***MUL / IMUL* :** multiplication (non signée / signée)
Syntaxe : MUL/IMUL Source
Les deux instructions **IMUL** et **MUL** effectuent des multiplications. L'instruction **IMUL** effectue la multiplication d'opérandes signés. L'instruction **MUL** effectue la multiplication d'opérandes non signés. Les indicateurs de retenue (**C**) et de débordement (**O**) sont mis à un si le résultat ne peut pas être stockée dans l'opérande destination.

REG memory

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

Algorithme :
Lorsque l'opérande est un octet :
- AX = AL * operande

Lorsque l'opérande est un mot :
- (DX AX) = AX * operande

Exemple 1 :
```
MOV AL, 200   ; AL = 0C8h
MOV BL, 4
MUL BL        ; AX = 0320h (800)
RET
```
Exemple 2 :
```
MOV AL, -2
MOV BL, -4
IMUL BL      ; AX = 8
RET
```
Exemple 3 :
Pour bien comprendre la différence entre les instructions imul et mul, regardons les exemples suivants :
MOV BX, 435
MOV AX, 2372
IMUL BX
A l'issue de l'exécution de ces 3 instructions, **AX** contient la valeur **BE8C** et **DX** la valeur **F**, soit la valeur hexadécimale **FBE8C**, c'est-à-dire **1031820**, le produit de **435** par **2372**. Les deux données étant positives, le résultat est le même que l'on utilise l'instruction **IMUL** ou l'instruction **MUL.**
Considérons maintenant la séquence d'instructions :

MOV BX, -435
MOV AX, 2372
IMUL BX
A l'issue de leur exécution, **AX** contient la valeur **4174** et **DX** contient la valeur **FFF0**, soit la valeur -**1031820**. Si l'on remplace **IMUL** par **MUL**, le résultat n'a pas de sens.
***DIV / IDIV* :** division (non signée / signée)
Syntaxe : DIV/IDIV Source
Les deux instructions **DIV** et **IDIV** réalisent les opérations de division et de calcul de reste. **DIV** l'effectue sur des données non signées, **IDIV** sur des données signées.

REG memory

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Algorithme :
Lorsque l'opérande est un octet :
- AL = AX / operand
- AH = reste (modulo)

Lorsque l'opérande est un mot :
- AX = (DX AX) / operand
- DX = reste (modulo)

Exemple 1 :
```
MOV AX, 203   ; AX = 00CBh
MOV BL, 4
DIV BL        ; AL = 50 (32h), AH = 3
RET
```
Exemple 2 :
```
MOV AX, -203 ; AX = 0FF35h
MOV BL, 4
IDIV BL      ; AL = -50 (0CEh), AH = -3 (0FDh)
RET
```
Exemple 3 :
Pour bien comprendre le fonctionnement des instructions DIV et IDIV, prenons l'exemple suivant:
MOV BX, 435
MOV AX, 2372
DIV BX
A l'issue de l'exécution de cette séquence d'instructions, le registre **AX** contiendra la valeur **5** qui est le quotient de **2372** par **435**, et le registre **DX** vaudra **197** qui est le reste de la division. Si on remplace **DIV** par **IDIV**, le résultat est inchangé puisque le diviseur et le dividende sont tous deux positifs.
Si l'on considère maintenant la séquence :
MOV BX, -435
MOV AX, 2372
IDIV BX
Nous aurons ensuite **AX** qui contient la valeur **-5** (soit **FFFB** en hexadécimal) et **DX** la valeur **197**.
Si l'on remplace **IDIV** par **DIV**, le résultat n'a pas de sens.

***NEG* :** Négation par **complément à 2** [NEG AX signifie: AX = 5 → AX = -5 en complément à 2]
Syntaxe : NEG Destination
Elle soustrait l'opérande destination (octet ou mot) de **0** le résultat est stocker dans la destination, donc avec cette opération on réalise **le complément à deux d'un nombre.**

REG
memory

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :

- Inverser tous les bits de l'opérande.
- Ajouter 1 à l'opérande inversé.

Exemple :

```
MOV AL, 5  ; AL = 05h
NEG AL     ; AL = 0FBh (-5)
NEG AL     ; AL = 05h (5)
RET
```

***CMP* :** compare les opérandes et positionne les drapeaux en fonction du résultat. L'opérande n'est pas modifié.
Syntaxe : CMP Destination, Source
C'est l'instruction la plus utilisée pour positionner les indicateurs avant d'effectuer **une instruction de saut conditionnel**. Elle soustrait la source de la destination, qui peut être un octet ou un mot, le résultat n'est pas mis dans la destination, en effet cette instruction touche uniquement les indicateurs pour être testé avec une autre instruction ultérieure de saut conditionnel. Les indicateurs susceptibles d'être touché sont : **OF, SF, ZF, AF, PF, CF**.

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :

- operande1 - operande2
- Le résultat n'est stocké nulle part, les drapeaux sont modifiés (**OF, SF, ZF, AF, PF, CF**) en fonction du résultat.

Exemple :
A l'issue de l'exécution des deux instructions suivantes :
MOV AL, 23
CMP AL, 34
Le registre **AL** n'est pas modifié par l'exécution de l'instruction **CMP** et contient toujours la valeur affectée auparavant **23.** L'indicateur de retenue **C** est positionné à **1**, ce qui indique que le deuxième opérande de l'instruction **CMP** est supérieur à la valeur du premier opérande. L'indicateur de zéro **Z** valant **0**, cela indique que les deux données sont différentes (sinon, la soustraction d'un nombre à lui-même donne **0**, donc le bit **Z** est positionné à **1**). Le bit de signe **S** est également mis à **1** car **23 − 34** est un nombre négatif.

4. Instructions logiques

***AND* : et logique** [AND AL, 01011100b sachant AL = 00110110 donne AL ← 00010100]
Syntaxe : AND Destination, Source
Elle réalise un **et-logique bit à bit** entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

REG, memory memory, REG REG, REG memory, immediate REG, immediate

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Ces règles s'appliquent :
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
Exemple :
Considérons les deux lignes suivantes :
MOV AL, 36h
AND AL, 5Ch
Le registre **AL** contient ensuite la valeur **14h** obtenue de la manière suivante :

36h ∧ 5ch	0011 0110 0101 1100
14h	0001 0100

Les bits **S** et **Z** sont mis à zéro et le bit **P** à **1**
***OR* : ou logique** [OR AL, 01011100b sachant AL = 00110110 donne AL ← 0111 1110]
Syntaxe : OR Destination, source
Elle réalise un **ou-logique bit à bit** entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

REG, memory memory, REG REG, REG memory, immediate REG, immediate

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Ces règles s'appliquent :
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0

Exemple :
Considérons les deux lignes suivantes :
MOV AL, 36h
OR AL, 5Ch
Le registre **AL** contient ensuite la valeur **7eh** obtenue de la manière suivante :

36h V 5ch	0011 0110 0101 1100
7eh	0111 1110

Les bits **S** et **Z** sont mis à zéro et le bit **P** à **1**
***XOR* : ou exclusif** [XOR AL, 01011100b sachant AL = 00110110 donne AL ← 01101010]
Syntaxe : XOR Destination, source
Elle réalise un **ou-exclusif bit à bit** entre l'opérande source et l'opérande destination. Le résultat est rangé dans l'opérande destination.

REG, memory memory, REG REG, REG memory, immediate REG, immediate

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Ces règles s'appliquent :
1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
0 XOR 0 = 0
Exemple :
Considérons les deux lignes suivantes :
MOV AL, 36h
XOR AL, 5Ch
Le registre **AL** contient ensuite la valeur **6Ah** obtenue de la manière suivante :

36h ⊕ 5ch	0011 0110 0101 1100
6ah	0110 1010

Les bits **S** et **Z** sont mis à zéro et le bit **P** à **1**
***NOT* : négation booléenne** [NOT AL, sachant AL = 00110110 donne AL ← 11001001]
Syntaxe : NOT Destination
Elle réalise **la complémentation à 1** d'un nombre.

REG memory

O	D	I	T	S	Z	A	P	C

Algorithme :

- Si le bit est 1, le mettre à 0
- Si le bit est 0, le mettre à 1

Exemple :

Considérons les deux lignes suivantes :

MOV AL, 36h
NOT AL

Le registre **AL** contient ensuite la valeur **c9h** obtenue de la manière suivante :

¬ 36h	0011 0110
c9h	1100 1001

Les indicateurs du registre **FLAGS** ne sont pas modifiés par l'instruction **NOT**.

***TEST* : similaire à AND**, seuls les indicateurs sont positionnés

Syntaxe : TEST Destination, source

Cette instruction effectue **un et-logique** entre ses deux opérandes. Cependant, contrairement à l'instruction AND, elle **ne modifie pas l'opérande destination**. Elle se contente de positionner les indicateurs du registre **FLAGS**.

REG, memory memory, REG REG, REG memory, immediate REG, immediate

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Ces règles s'appliquent :

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

Exemple :

```
MOV AL, 00000101b
TEST AL, 1        ; ZF = 0, PF = 0, SF = 0
TEST AL, 10b      ; ZF = 1, PF = 1, SF = 0
RET
```

5. Instructions de décalage et rotation

Nous décrivons ici des opérations classiques des langages d'assemblage qui se nomment **décalages et rotations**. On les rencontre couramment car leur utilisation simplifie grandement certains traitements. **Les rotations** considèrent un opérande (octet ou mot) comme **un tore** dont **elles décalent les bits**. Lors du décalage, **un bit déborde** d'un côté, à gauche ou à droite, selon le sens de la rotation. **Une opération de décalage** consiste simplement **à décaler tous les bits** d'une donnée. Contrairement aux rotations qui considèrent une donnée (un octet ou un mot) comme **un tore**, un décalage considère la donnée comme **une file** ; ainsi, le bit qui « **déborde** » de la retenue **est perdu**.

Syntaxe des instructions de rotation et de décalage :

INST Destination, Compteur ; compteur est le nombre de décalages

memory, immediate
REG, immediate
memory, CL
REG, CL

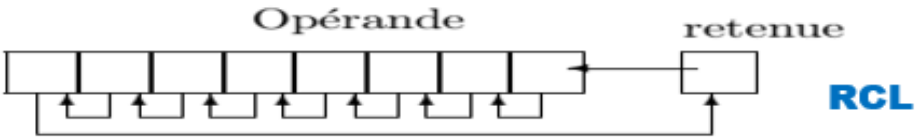
O	D	I	T	S	Z	A	P	C
*								*

OF = 0 si le premier opérande garde le signe d'origine, OF = 1 sinon.

5.1 Instructions de rotation

***RCL* :** Rotation à gauche avec retenue

Le bit de poids fort est mis dans l'indicateur de retenue **CF**, la valeur de cet indicateur étant préalablement mise dans **le bit de poids faible**.

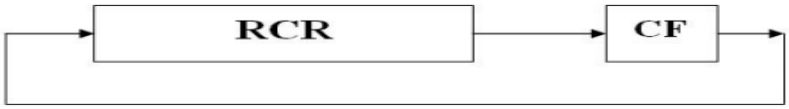


Exemple :

```
STC          ; set carry (CF=1).
MOV AL, 1Ch  ; AL = 00011100b
RCL AL, 1    ; AL = 00111001b,  CF=0.
RET
```

***RCR* :** Rotation à droite avec retenue

Le bit de poids faible de l'opérande destination est mis dans l'indicateur de retenue **CF**, la valeur de cet indicateur étant préalablement mise dans **le bit du poids fort** de l'opérande destination.

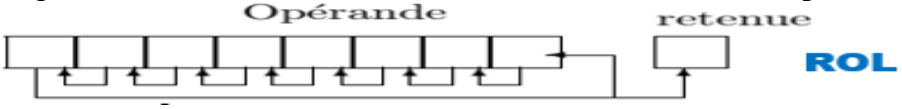


Exemple :

```
STC          ; set carry (CF=1).
MOV AL, 1Ch  ; AL = 00011100b
RCR AL, 1    ; AL = 10001110b,  CF=0.
RET
```

***ROL* :** Rotation à gauche avec écrasement de la retenue

Le bit de poids fort est mis dans l'indicateur de retenue **CF** et dans **le bit de poids faible** de l'opérande. L'ancienne valeur de l'indicateur de retenue **n'est pas utilisée**.

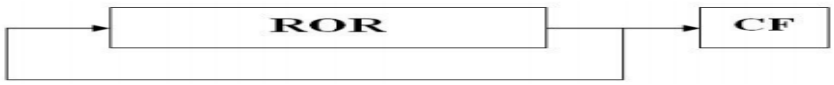


Exemple :

```
MOV AL, 1Ch  ; AL = 00011100b
ROL AL, 1    ; AL = 00111000b,  CF=0.
RET
```

***ROR* :** Rotation à droite avec écrasement de la retenue

Le bit de poids faible est mis dans l'indicateur de retenue **CF** lors de la rotation ainsi que dans **le bit de poids fort** de l'opérande. L'ancienne valeur de l'indicateur de retenue **n'est pas utilisée**.



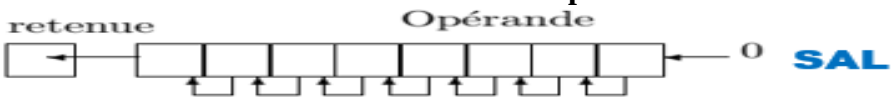
Exemple :

```
MOV AL, 1Ch      ; AL = 00011100b
ROR AL, 1        ; AL = 00001110b,  CF=0.
RET
```

5.2 Instructions de décalage

***SAL / SHL* :** Décalage à gauche avec **0** en entrée et **CF** en sortie

SAL et **SHL** sont **des synonymes** et peuvent être utilisées l'une pour l'autre indifféremment. **SAL/SHL** effectue **un décalage vers la gauche**, sauvegardant **le bit de poids fort** dans l'indicateur de retenue **CF** et mettant un **0** dans **le bit de poids faible**.

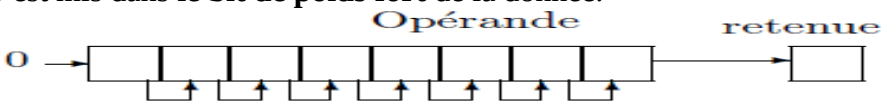


Exemple :

```
MOV AL, 0E0h     ; AL = 11100000b
SAL AL, 1        ; AL = 11000000b,  CF=1.
RET
```

***SHR* :** Décalage à droite avec **0** en entrée et **CF** en sortie

SHR effectue **un décalage vers la droite**. **Le bit de poids faible** est mis dans la retenue **CF**. Un **0** est mis dans **le bit de poids fort** de la donnée.

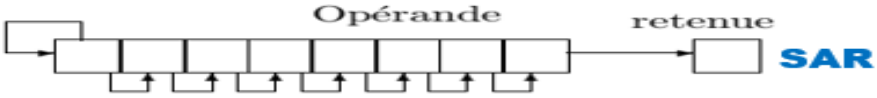


Exemple :

```
MOV AL, 00000111b
SHR AL, 1        ; AL = 00000011b,  CF=1.
RET
```

***SAR* :** Décalage à droite avec **CF** en sortie et **duplication** du bit de poids fort

SAR effectue **un décalage vers la droite**, sauvegardant **le bit de poids faible** dans l'indicateur de retenue **CF**. Par ailleurs (et c'est là toute la différence avec l'instruction précédente), **le bit de poids fort** est conservé. **Le bit de poids fort** de la donnée initiale est donc **dupliqué**.



Exemple :

```
MOV AL, 0E0h     ; AL = 11100000b
SAR AL, 1        ; AL = 11110000b,  CF=0.
MOV BL, 4Ch      ; BL = 01001100b
SAR BL, 1        ; BL = 00100110b,  CF=0.
RET
```

Il faut noter que pour un nombre, **un décalage d'une position vers la gauche correspond à une multiplication de ce nombre par 2** et **qu'un décalage d'une position vers la droite** correspond à **une division entière par 2**. En généralisant, un décalage de **l** positions vers la gauche correspond à une multiplication par $\mathbf{2}^{\boldsymbol{l}}$ et un décalage de **l** positions vers la droite correspond à une division par $\mathbf{2}^{\boldsymbol{l}}$. Il faut bien noter, et c'est ce qui justifie l'existence des deux instructions **SAR** et **SHR** et

leur subtile différence, que **SAR** effectue une division sur un nombre en **représentation signée** tandis que **SHR** effectue une division sur un nombre en **représentation non signée**.

6. Instructions de branchement

Ce sont les instructions de **sauts de programme**. Elles permettent de faire **des sauts** dans l'exécution d'un programme (**rupture de séquence**). Ces instructions n'affectent pas les Flags :

O	D	I	T	S	Z	A	P	C

Dans cette catégorie on trouve toutes **les instructions de branchement (inconditionnel et conditionnel)**, de **boucle** et d'**interruption**.

6.1 Instructions de branchement inconditionnel

***JMP* :** Saut inconditionnel

Syntaxe : JMP etiquette

L'instruction **JMP** effectue **un saut inconditionnel** à **l'étiquette** spécifiée. Contrairement à **un saut conditionnel**, le saut est toujours effectué, le registre **FLAGS** n'intervenant en rien dans cette opération.

Exemple :

```
MOV AX, 00    ; Initialisation d'AX à 0
MOV BX, 00    ; Initialisation de BX à 0
MOV CX, 01    ; Initialisation de CX à 1
L20:
ADD AX, 01    ; Incrémenter AX
ADD BX, AX    ; Ajouter AX à BX
SHL CX, 1     ; décalage gauche de CX, cela permet de doubler la valeur de CX
JMP L20       ; répéter les instructions
```

***CALL* :** appel de sous-programme

Syntaxe : CALL etiquelle

La notion de **procédure** en assembleur correspond à celle de sous-programme dans d'autres langages. Dans la figure 25, nous faisons l'appel d'une procédure nommée **calcul**. Après **l'instruction B**, le processeur passe à **l'instruction C** de la procédure, puis continue jusqu'à rencontrer **RET** et revient à **l'instruction D**.

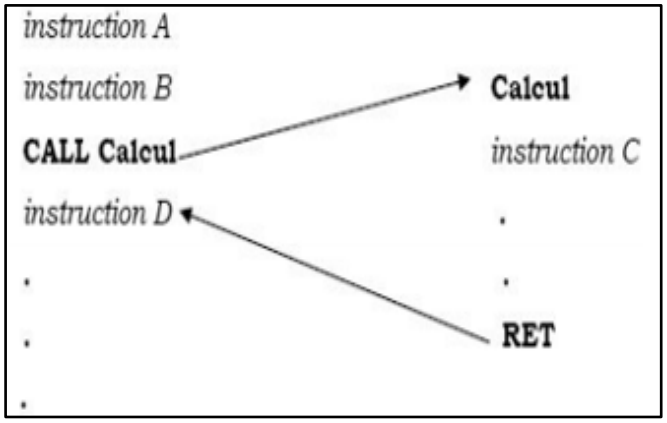


Figure 25: Appel d'une procédure

Une procédure est une suite d'instructions effectuant **une action précise**, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme. Les procédures sont repérées par **l'adresse de leur première instruction**, à laquelle on associe **une étiquette** en assembleur. L'exécution d'une procédure est déclenchée par **un programme appelant**. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite. L'appel d'une procédure est effectué par **l'instruction CALL**. La fin d'une procédure est marquée par **l'instruction RET**.

Exemple :

```
ORG 100h
CALL p1
ADD AX, 1
RET       ; return to OS.
p1 PROC   ; procedure declaration.
MOV AX, 1234h
RET       ; return to caller.
p1 ENDP
```

***RET* :** retour de sous-programme

RET ne prend pas d'argument et permet rendre **le contrôle** au sous-programme appelant : le processeur passe à l'instruction placée immédiatement après le **CALL**. Mais comment le processeur retrouve-t-il la valeur de l'adresse de cette instruction ? Le problème est compliqué par le fait que l'on peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure ci-contre (cf. fig 26).

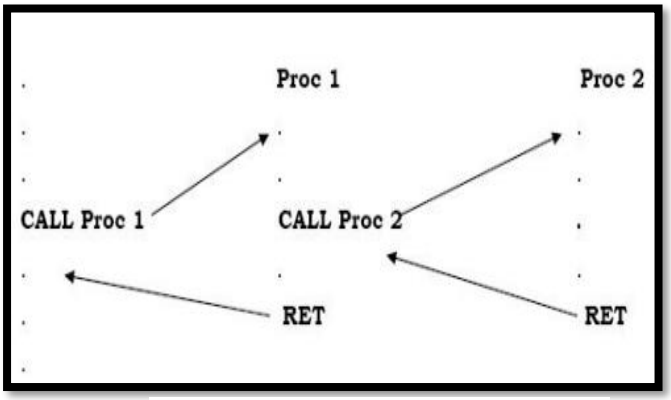


Figure 26: Retour de sous-programme

L'adresse de retour, utilisée par **RET**, est en fait sauvegardée sur la pile par l'instruction **CALL**. Lorsque le processeur exécute l'instruction **RET**, il dépile l'adresse sur la pile (comme POP), et la range dans **IP**.

L'instruction **CALL** effectue donc les opérations suivantes :

- Empiler la valeur de **IP**. A ce moment, **IP** pointe sur l'instruction qui suit le **CALL**.
- Placer dans **IP** l'adresse de la première instruction de la procédure (donnée en argument).

Et l'instruction **RET** :

- Dépiler une valeur et la ranger dans **IP**.

Remarque concernant les procédures :

Une procédure peut être de type **NEAR** si elle se trouve dans **le même segment** que le programme principal ou de type **FAR** si elle se trouve dans un autre segment. La différence entre eux c'est que dans le premier cas le processeur doit empiler une seule valeur dans la pile c'est le registre **IP** mais dans le deuxième cas il faut empiler le registre **IP** ainsi que le registre segment **CS** et bien sûr il faut les dépiler pendant le retour de la procédure.

Passage de paramètres pour les procédures :

En général, une procédure effectue un traitement sur des données (**paramètres**) qui sont fournies par le programme appelant, et produit un **résultat** qui est transmis à ce programme. Plusieurs stratégies peuvent être employées :

1. **Passage par registre :** les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).
2. **Passage par la pile :** les valeurs des paramètres sont empilées. La procédure lit la pile

Exemple avec passage par registre :

On va écrire une procédure (**SOMME**) qui calcule la somme de **2** nombres naturels de **16 bits**. Convenons que les entiers sont passés par les registres **AX** et **BX**, et que le résultat sera placé dans le registre **AX** :

```
ORG 100h
MOV AX, 6
MOV BX, Truc
CALL SOMME
MOV Truc, AX
RET
Truc dw 10
SOMME PROC
ADD AX, BX ; AX <- AX + BX
RET
SOMME ENDP
```

Exemple avec passage par la pile :

Cette technique met en œuvre un nouveau registre, **BP** (**Base Pointer**), qui permet de lire des valeurs sur la pile sans les ne dépiler ni modifier **SP**. Le registre **BP** permet un mode d'adressage indirect spécial, de la forme : **MOV AX, [BP+6]**

Cette instruction charge le contenu du mot mémoire d'adresse **BP+6** dans **AX**. Ainsi, nous lirons le sommet de la pile avec :

MOV BP, SP ; BP pointe sur le sommet
MOV AX, [BP] ; lit sans dépiler

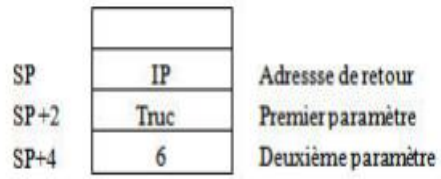
Et le mot suivant avec :

MOV AX, [BP+2] ; 2 car 2 octets par mot de pile

L'appel de la procédure (**SOMME2**) avec passage par la pile est :

```
PUSH 6
PUSH Truc
CALL SOMME2
```

La procédure **SOMME2** va lire la pile pour obtenir la valeur des paramètres. Pour cela, il faut bien comprendre quel est le contenu de la pile après le **CALL** :



Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP empilée par **CALL**). Chaque élément de la pile occupe deux octets.

La procédure SOMME2 s'écrit donc :

```
SOMME2 PROC ; AX <- arg1 + arg2
MOV BP, SP ; adresse sommet pile
MOV AX, [BP+2] ; charge argument 1
ADD AX, [BP+4] ; ajoute argument 2
RET
SOMME2 ENDP
```

6.2 Instructions de branchement conditionnel

***Jxx* : Sauts conditionnels** en fonction des drapeaux (cf. fig 27).

Toutes les instructions de saut conditionnel prennent le même type d'opérande.

Toutes ces instructions fonctionnent selon le schéma suivant. Quand **la condition est vraie**, **un saut** est effectué à l'instruction située à **l'étiquette** spécifiée en opérande. Notons que les tests d'ordre (inférieur, supérieur, ...) se comprennent de la manière suivante. Supposons que nous comparions deux données par une instruction **CMP** et que nous exécutions ensuite une instruction **JG**, c'est-à-dire « **saut si plus grand** ». Il y aura alors saut si la valeur du premier opérande de l'instruction **CMP** est supérieur à la valeur du deuxième opérande.

JA	étiquette	saut si supérieur (C =0 et Z =0)
JAE	étiquette	saut si supérieur ou égal (C =0)
JB	étiquette	saut si inférieur (C =1)
JBE	étiquette	saut si inférieur ou égal (C =1 ou Z =1)
JC	étiquette	saut si retenue (C =1)
JCXZ	étiquette	saut si CX vaut 0
JE	étiquette	saut si égal (Z =1)
JG	étiquette	saut si supérieur (Z =0 et S =0)
JGE	étiquette	saut si supérieur ou égal (S =0)
JL	étiquette	saut si inférieur (S ≠0)
JLE	étiquette	saut si inférieur ou égal (Z =1 ou S ≠0)
JNC	étiquette	saut si pas de retenue (C =0)
JNE	étiquette	saut si non égal (Z =0)
JNO	étiquette	saut si pas de débordement (O =0)
JNP	étiquette	saut si pas de parité (P =0)
JNS	étiquette	saut si pas de signe (S =0)
JO	étiquette	saut si débordement (O =1)
JP	étiquette	saut si parité (P =1)
JS	étiquette	saut si signe (S =1)

Figure 27: Instructions de saut conditionnel

Exemple :

```
include 'emu8086.inc'
ORG 100h
MOV AL, 5
CMP AL, -5 ; ZF = 0 et SF = 0
JG  label1
PRINT 'AL is not greater -5.'
JMP exit
label1:
PRINT 'AL is greater -5.'
exit:
RET
```

6.2.1 Ecriture des tests

Les tests (**if**) peuvent s'écrire en assembleur de différentes manières en fonction de la condition et de l'instruction de saut utilisée. Une instruction **if (op1 [operateur] op2) then Action** peut se traduire comme suit :

```
CMP op1, op2
Jxx et1              ; tel que Jxx ⇔ NOT [opérateur]
Action
et1:
```

Une instruction **if (op1 [operateur] op2) then Action1 else Action2** peut se traduire comme suit:

```
CMP op1, op2
Jxx et1          ; tel que Jxx ⇔ [opérateur]
Action2
jmp et2
```

et1:
Action1
et2:

Exemple :

Nous donnons un exemple de codage d'un **test**. Nous voulons effectuer **la division** d'un nombre par un autre. La division n'est possible que si **le diviseur est non nul**. Nous devons tester si l'opérande diviseur est nul avant d'exécuter la division. Si le diviseur est nul, nous désirons que le résultat du traitement soit **-1**. Sinon, le résultat est le quotient des deux nombres. Voici le code à implémenter :

if (val=0) then
    al←-1
else
    al← $\frac{ax}{val}$

Ci-dessous le code 8086 équivalent :

```
org 100h
mov ax, …
cmp val, 0
je et1
mov bl, val
div bl
jmp et2
et1:
mov al, -1
et2:
ret
val db …
```

6.3 Instructions de boucle

***LOOPx* :** contrôle de **boucle** en fonction d'un compteur (cf. le tableau suivant)

Syntaxe : LOOPx etiquette

Instruction	Mise à jour	Branchement
LOOP	**CX ← CX-1**	**CX ≠ 0**
LOOPZ, LOOPE	**CX ← CX-1**	**(CX ≠ 0) et (Z=1)**
LOOPNZ, LOOPNE	**CX ← CX-1**	**(CX ≠ 0) et (Z=0)**

***LOOP* :** boucle

LOOP fonctionne avec le registre **CX** qui joue le rôle de **compteur de boucles**. **LOOP** décrémente le compteur sans modifier aucun des indicateurs. Si le compteur est différent de **0**, un saut à l'étiquette opérande de l'instruction **LOOP** est réalisé.

Algorithme :

CX = CX – 1
Si CX <> 0 alors
- Saut

Sinon
- pas de saut, continue

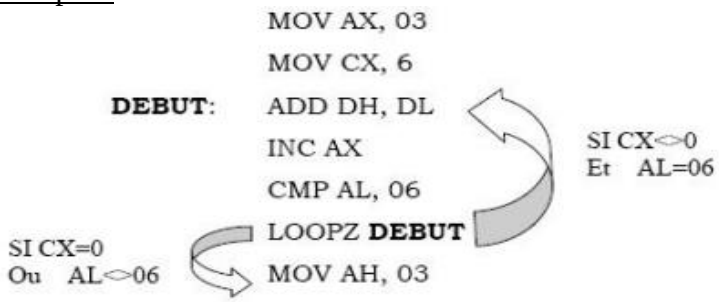
***LOOPE / LOOPZ* :** boucle si égale ou si égale à zéro
Il y a saut à l'étiquette spécifiée en opérande tant que le compteur est **non nul** et que l'indicateur **Z** vaut **1**.
Algorithme :
CX = CX – 1
Si CX <> 0 et ZF=1 alors

- Saut

Sinon

- pas de saut, continue

Exemple **:**



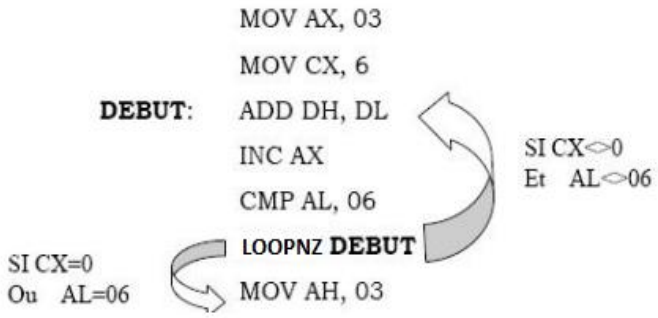
***LOOPNE / LOOPNZ* :** boucle si non égale ou si non égale à zéro
Il y a saut à l'étiquette spécifiée en opérande tant que le compteur est **non nul** et que l'indicateur **Z** vaut **0**
Algorithme :
CX = CX – 1
Si CX <> 0 et ZF=0 alors

- Saut

Sinon

- pas de saut, continue

Exemple :



6.3.1 Ecriture des boucles

Les instruction **LOOP** facilitent l'écriture des boucles de type « **for** » grâce à l'auto-décrémentation du registre **CX** et à la condition de branchement basée sur celui-ci.
Une boucle **for i := 10 down to 1 do Action** peut se traduire comme suit :

```
MOV CX, 10
et1:
Action
LOOP et1
```

Les autres types de boucles peuvent être traduites à l'aide des instructions de branchement en fonction de la condition.

Une boucle **while (op1 [operateur] op2) do Action** peut se traduire comme suit :

```
et1:
CMP op1, op2
Jxx et2          ; tel que Jxx ⇔ NOT [opérateur]
Action
jmp et1
et2:
```

Une instruction **repeat Action until (op1 [operateur] op2)** peut se traduire comme suit :

```
et1:
Action
CMP op1, op2
Jxx et1          ; tel que Jxx ⇔ NOT [opérateur]
```

Exemple 1 :

Programme pascal	Programme 8086 equivalent
AX := 1; BX := 3; CX := 0; **Repeat** AX := AX × BX; CX := CX + 1; **until** (CX = BX);	**ORG 100h** **MOV AX, 1** **MOV BX, 3** **MOV CX, 0** **Etq:** **MUL BX** **INC CX** **CMP CX,BX** **JNE Etq** **ret**

Ce programme calcule A := A * 3 trois fois : (1 * 3) =3 ; (3 * 3)=9 ; (9 * 3)=27

Exemple 2 :

Programme pascal	Programme 8086 equivalent

AX := 1;
BX := 3;
CX := 0;
While (CX <> BX) **do**
Begin
AX := AX × BX;
CX := CX + 1;
End;
	ORG 100h
	MOV AX, 1
	MOV BX, 3
	MOV CX, 0
	Etq:
	CMP CX,BX
	JE Fin
	MUL BX
	INC CX
	JMP Etq
	Fin:
	ret

6.4 Instructions d'interruption

Les interruptions peuvent être considérées comme un certain nombre de **fonctions**. **Ces fonctions** rendent la programmation beaucoup plus **facile**, au lieu d'écrire un code pour **imprimer un caractère**, vous pouvez simplement appeler **l'interruption** et elle fera tout pour vous. Nous appelons ces fonctions **des interruptions logicielles**. Les interruptions sont également déclenchées

par différents **matériels**, on les appelle **des interruptions matérielles**. Actuellement, nous nous intéressons uniquement aux **interruptions logicielles**.
Pour faire **une interruption du logiciel**, il y a une instruction **INT**, elle a une syntaxe très simple:

INT valeur

Où la valeur peut être un nombre entre **0** et **255** (ou **0** à **0FFh**), généralement nous utiliserons des nombres **hexadécimaux**. Chaque interruption peut avoir des **sous-fonctions**. Pour spécifier une sous-fonction, le registre **AH** doit être défini avant d'appeler l'interruption. Chaque interruption peut avoir jusqu'à **256 sous-fonctions** (nous obtenons donc **256 * 256** = **65536 fonctions**). En général, le registre **AH** est utilisé, mais parfois d'autres registres peuvent être utilisés. En général, d'autres registres sont utilisés pour transmettre **des paramètres et des données** à une sous-fonction.

Exemple :

Pour appeler la sous-fonction «**00h**» de l'interruption «**21h**» on exécutera la partie de code suivante:

```
MOV AH, 00h     ; sélection de la sous-fonction
INT 21h         ; appel à l'interruption
```

L'une des principales interruptions utilisée est la « **21h** », et ses fonctions sont nombreuses.

6.4.1 Principales sous-fonctions de l'interruption 21h

La fonction **00h** : met fin au programme :

```
MOV AH, 0
INT 21h
```

La fonction **01h** : attend la saisie d'un caractère au clavier et renvoi son code ASCII dans le registre **AL** :

```
MOV AH, 1
INT 21h
```

La fonction **02h** : affiche sur l'écran le caractère correspondant au code ASCII de la valeur contenue dans le registre **DL** (après exécution **AL = DL**) :

```
MOV AH, 2
MOV DL, 'a'
INT 21h
```

La fonction **09h** : permet d'afficher une suite de caractères depuis l'adresse contenue dans **DX**. La chaîne doit être terminée par le caractère **$** :

```
ORG 100h
MOV DX, OFFSET msg
MOV AH, 9
INT 21h
RET
msg db 'hello world $'
```

La fonction **0Ah** : permet de saisir une chaîne de caractère dans **DS: DX**, le premier octet est la taille du tampon, le deuxième octet est le nombre de caractères réellement lus. Cette fonction n'ajoute pas **'$'** à la fin de la chaîne. Pour imprimer avec **INT 21h** / **AH = 9**, vous devez insérer le caractère dollar à la fin de celle-ci et démarrer l'impression à partir de l'adresse **DS:DX+2** :

```
ORG 100h
MOV DX, OFFSET buffer
MOV AH, 0ah
INT 21h
```

```
JMP print
Buffer db 10, ?, 10 dup(' ')
print:
XOR BX, BX
MOV BL, buffer[1]
MOV buffer[BX+2], '$'
MOV DX, OFFSET buffer+2
MOV AH, 9
INT 21h
RET
```

7. Instructions de manipulation des tableaux ou chaines de caractères

Notation

La directive **OFFSET** est utile pour récupérer l'adresse d'une variable (similaire à **LEA**)

Exemple : **mov bx, offset t1** ; récupère l'adresse de t1
mov bx, offset tab1 ; récupère l'adresse du 1er élément de tab1

Les registres **SI** te **DI** sont utilisées pour manipuler les tableaux :

- La chaîne Source se trouve à l'adresse **DS:SI**
- La chaîne Destination se trouve à l'adresse **ES:DI**
- **SI** et **DI s'incrémentent** ou **se décrémentent** pour parcourir les tableaux
- Le drapeau **D** définit le sens de parcours du tableau (**D=1 → sens inverse**)

***LODSB / LODSW* :** chargement d'un élément de chaîne depuis la mémoire ([**DS:SI**]) dans le registre accumulateur.

LODSB charge le registre **AL** (respectivement, **AX** pour l'instruction **LODSW**) avec **l'octet** (respectivement **le mot**) pointé par le registre **SI**. Une fois le chargement effectué, il est automatiquement ajouté **1** (respectivement **2**) à **SI** si l'indicateur **D** vaut **0**, ou retiré **1** (respectivement **2**) si l'indicateur **D** vaut **1**.

O	D	I	T	S	Z	A	P	C

Algorithme :

AL = DS:[SI] pour LODSB (AX = DS:[SI] pour LODSW)
Si DF = 0 alors

- SI = SI + 1 pour LODSB (SI = SI + 2 pour LODSW)

sinon

- SI = SI - 1 pour LODSB (SI = SI - 2 pour LODSW)

***STOSB / STOSW* :** Écriture en mémoire (dans [ES:DI]) depuis l'accumulateur.

STOSB écrit en mémoire **l'octet** (respectivement **le mot** pour l'instruction **STOSW**) se trouvant dans le registre **AL** (respectivement dans le registre **AX**) en mémoire à l'adresse pointée par le registre **DI**. Le segment destination est forcément **ES**. Une fois la copie réalisée, **DI** est automatiquement modifié. Si l'indicateur de direction vaut **0**, **1** (respectivement **2**) est ajouté à **DI**. Si l'indicateur de direction vaut **1**, **1** (respectivement **2**) est retiré à **DI**.

O	D	I	T	S	Z	A	P	C

Algorithme :
ES:[DI] = AL pour STOSB (ES:[DI] = AX pour pour STOSW)
Si DF = 0 then

- DI = DI + 1 pour STOSB (DI = DI + 2 pour pour STOSW)

sinon

- DI = DI – 1 pour STOSB (DI = DI - 2 pour pour STOSW)

Exemple :
Cet exemple consiste à **multiplier par 2** tous les éléments d'un tableau d'entiers :

```
org 100h
.data
TABLEAU DW 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
RES DW 10 DUP (0)
.code
mov ax, @data
mov ds, ax
mov es, ax
lea si, TABLEAU
lea di, RES
mov cx, 10
BOUCLE1: lodsw
mov bl, 2
mul bl
stosw
loop BOUCLE1
ret
```

***MOVSB / MOVSW* :** Transfert de [DS:SI] → [ES:DI]
MOVSB copie **l'octet** (respectivement **le mot** pour l'instruction **MOVSW**) pointé par **SI** vers **l'octet** (respectivement **le mot**) pointé par **ES:DI**. Le segment destination est forcément **ES**. Par contre, le segment de la source qui est **DS** par défaut peut être redéfini. Une fois la copie réalisée, **SI** et **DI** sont automatiquement modifiés. Si l'indicateur de direction vaut **0**, **1** (respectivement **2**) est ajouté à **SI** et **DI**. Si l'indicateur de direction vaut **1**, **1** (respectivement **2**) est retiré à **SI** et **DI**.

O	D	I	T	S	Z	A	P	C

Algorithme :
ES:[DI] = DS:[SI]
if DF = 0 then

- SI = SI + 1 pour MOVSB (SI = SI + 2 pour MOVSW)
- DI = DI + 1 pour MOVSB (DI = DI + 2 pour MOVSW)

else

- SI = SI – 1 pour MOVSB (SI = SI - 2 pour MOVSW)
- DI = DI – 1 pour MOVSB (DI = DI - 2 pour MOVSW)

***SCASB / SCASW* :** Comparaison entre accumulateur et [ES:DI]
SCASB compare la valeur de **l'octet** (respectivement le mot pour l'instruction **SCASW**) contenu dans le registre **AL** (respectivement **AX**) à **l'octet** (respectivement **le mot**) pointé par **ES:DI**. La donnée en mémoire est toujours référencée via le registre de segment **ES**. Une fois la comparaison

réalisée, **DI** est automatiquement modifié. Si l'indicateur de direction vaut **0**, **1** (respectivement **2**) est ajouté à **DI**. Si l'indicateur de direction vaut **1**, **1** (respectivement **2**) est retiré à **DI**.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
AL - ES:[DI] pour SCASB (AX - ES:[DI] pour SCASW)
Modifier les drapeaux en fonction du résultat:
OF, SF, ZF, AF, PF, CF
Si DF = 0 alors

- DI = DI + 1 pour SCASB (DI = DI + 2 pour SCASW)

sinon

- DI = DI – 1 pour SCASB (DI = DI - 2 pour SCASW)

***CMPSB / CMPSW* :** Comparaison entre [DS:SI] et [ES:DI].
CMPSB compare **l'octet** (respectivement **le mot** pour l'instruction **CMPSW**) pointé par **SI** à **l'octet** (respectivement **le mot**) pointé par **ES:DI**. Le segment destination est forcément **ES**. Par contre, le segment de la source qui est **DS** par défaut peut être redéfini. Une fois la comparaison réalisée, **SI** et **DI** sont automatiquement modifiés. Si l'indicateur de direction vaut **0**, **1** (respectivement **2**) est ajouté à **SI** et **DI**. Si l'indicateur de direction vaut **1**, **1** (respectivement **2**) est retiré à **SI** et **DI**.

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Algorithme :
DS:[SI] - ES:[DI]
Modifier les drapeaux en fonction du résultat:
OF, SF, ZF, AF, PF, CF
SI DF = 0 alors

- SI = SI + 1 pour CMPSB (SI = SI + 2 pour CMPSW)
- DI = DI + 1 pour CMPSB (DI = DI + 2 pour CMPSW)

sinon

- SI = SI – 1 pour CMPSB (SI = SI - 2 pour CMPSW)
- DI = DI – 1 pour CMPSB (DI = DI - 2 pour CMPSW)

***REP / REPE / REPZ / REPNE / REPNZ* :** Répéter une instruction de traitement de chaînes de données.
Ces instructions permettent d'**itérer** sur toute une chaîne les opérations élémentaires vues précédemment. Cette instruction **préfixe** en fait l'opération à itérer. Elle donne tout son intérêt aux instructions **LODS**, **STOS**, **MOVS**, **CMPS**, **SCAS**. **REP** ne peut préfixer que les instructions **LODS**, **STOS**, **MOVS**. Un compteur (registre **CX**) indique le nombre d'itérations à effectuer. Les préfixes **REPE** et **REPNE** ne peuvent préfixer que les instructions **CMPS** et **SCAS**. Pour les préfixes **REPE** et **REPNE**, un compteur (registre **CX**) indique le nombre maximal d'itérations à effectuer. Par ailleurs, l'instruction élémentaire est répétée tant qu'une condition (égalité ou non-égalité entre les opérandes de **CMPS** ou **SCAS**) n'est pas remplie. Notons que **REPZ** est un synonyme de **REPE** ; **REPNZ** est un synonyme de **REPNE**.

O	D	I	T	S	Z	A	P	C
					*			

***REP* :** Répèter LODS / STOS / MOVS tant que CX≠0
Répéter les instructions suivantes **LODSB**, **LODSW**, **STOSB**, **STOSW**, **MOVSB**, **MOVSW CX** fois.
Algorithme :
check_cx:
Si CX <> 0 alors

- Executer l'instruction de chaine spécifiée
- CX = CX - 1
- revenir à check_cx

Sinon

- sortir du cycle REP

Explications :

- **REP LODS** : charge **CX** éléments de la chaîne pointée par **SI** dans **AL** (ou **AX**)
- **REP STOS** : écrit **CX** éléments de la chaîne pointée par **ES:DI** avec la valeur contenue dans **AL** (ou **AX**).
- **REP MOVS** : copie **CX** éléments de la chaîne pointée par **SI** vers la chaîne pointée par **ES:DI**.

Exemple :
Cet exemple permet de **copier** un tableau de caractères dans un autre tableau :

```
org 100h
.data
CHAINE db 'hello world' ; chaine source
RES db 11 dup (?) ; chaine cible
.code
mov ax, @data
mov ds, ax
mov es, ax
lea di, RES ; offset chaine cible
lea si, CHAINE ; offset chaine source
mov cx, 11 ; longueur de la chaine
rep movsb ; copie
ret
```

***REPE / REPZ* :** répète SCAS / CMPS tant que CX≠0 et [ES:DI]= AL / AX / [SI]
Répétez les instructions suivantes **SCASB**, **SCASW**, **CMPSB**, **CMPSW** tant que **ZF = 1** (le résultat est égal), **CX** fois au maximum.
Algorithme :
check_cx:
Si CX <> 0 alors

- Executer l'instruction de chaine spécifiée
- CX = CX - 1
- Si ZF = 1 alors:
  - revenir à check_cx

  sinon
  - sortir du cycle REPE

sinon

- sortir du cycle REPE

Explications :

- **REPE SCAS :** compare au plus **CX** éléments de la chaîne pointée par **ES:DI** avec la valeur du registre **AL**, ou **AX** selon le cas. Les itérations sont poursuivies tant que les éléments de la chaîne sont égaux à la valeur du registre et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction **REPE SCAS** est terminée.
- **REPE CMPS :** compare au plus **CX** éléments de la chaîne pointée par **ES:DI** avec ceux de la chaîne pointée par **SI**. Les itérations sont poursuivies tant que les éléments des deux chaînes sont égaux et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction **REPE CMPS** est terminée.

Exemple : **Recherche d'une chaîne de caractères dans une autre**.
A l'issue de cette recherche, le registre **AX** indiquera le résultat et vaudra **1** si la chaîne est trouvée, **0** sinon :

```
org 100h
.data
CH1 db 'hello world'
CH2 db 'rl'
.code
mov ax, @data
mov ds, ax
mov es, ax
lea si, CH1
mov ax, 1
mov bx, 10
TQ1: lea di, CH2
mov cx, 2
repe cmpsb
je FTQ1 ; sous-chaîne trouvée
dec bx
jne TQ1
mov ax, 0
FTQ1:
ret
```

***REPNE / REPNZ* :** répète SCAS / CMPS tant que CX≠0 et [ES:DI]≠ AL / AX / [SI]
Répétez les instructions suivantes **SCASB**, **SCASW**, **CMPSB**, **CMPSW** tant que **ZF = 0** (le résultat n'est pas égal), **CX** fois au maximum.

Algorithme :

check_cx:
Si CX <> 0 alors

- Executer l'instruction de chaine spécifiée
- CX = CX - 1
- Si ZF = 0 alors:
  - revenir à check_cx

  sinon
  - sortir du cycle REPE

sinon

- sortir du cycle REPE

Explications :

- **REPNE SCAS :** compare au plus **CX** éléments de la chaîne pointée par **ES:DI** avec la valeur du registre **AL**, ou **AX**, selon le cas. Les itérations sont poursuivies tant que les éléments de la chaîne sont différents de la valeur du registre et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction **REPNE SCAS** est terminée.

- **REPNE CMPS :** compare au plus **CX** éléments de la chaîne pointée par **ES:DI** avec ceux de la chaîne pointée par **SI**. Les itérations sont poursuivies tant que les éléments des deux chaînes sont différents et tant que le compteur n'est pas nul. Dès que l'une de ces conditions n'est plus vérifiée, l'instruction **REPNE CMPS** est terminée.

Exemple :

Recherche d'un caractère dans une chaîne de caractères en utilisant l'instruction **SCAS**. Le registre **BX** indiquera le résultat : **BX=1** si le caractère est trouvé, **0** sinon :

```
org 100h
.data
CHAINE db 'hello
world'
.code
mov ax, @data
mov es, ax
mov bx, 0
mov al, 'w'
lea di, CHAINE
mov cx, 11
repne scasb
jne ETQ
mov bx, 1
ETQ:
ret
```

Tableaux récapitulatifs

		D=0	*D=1*
REP	*LODSB*	*MOV AL, DS:[SI]* *INC SI*	*MOV AL, DS:[SI]* *DEC SI*
	LODSW	*MOV AX, DS:[SI]* *ADD SI, 2*	*MOV AX, DS:[SI]* *SUB SI, 2*
	STOSB	*MOV ES:[DI], AL* *INC DI*	*MOV ES:[DI], AL* *DEC DI*
	STOSW	*MOV ES:[DI], AX* *ADD DI, 2*	*MOV ES:[DI], AX* *SUB DI, 2*
	MOVSB	*MOV R, DS:[SI]* *MOV ES:[DI], R* *INC SI* *INC DI*	*MOV R, DS:[SI]* *MOV ES:[DI], R* *DEC SI* *DEC DI*
	MOVSW	*MOV R, DS:[SI]* *MOV ES:[DI], R* *ADD SI, 2* *ADD DI, 2*	*MOV R, DS:[SI]* *MOV ES:[DI], R* *SUB SI, 2* *SUB DI, 2*

		D=0	D=1
REPE / REPNE	*SCASB*	*CMP AL, ES: [DI] INC DI*	*CMP AL, ES: [DI] DEC DI*
	SCASW	*CMP AX, ES: [DI] ADD DI, 2*	*CMP AX, ES: [DI] SUB DI, 2*
	CMPSB	*MOV R, ES:[DI] CMP DS:[SI], R INC SI INC DI*	*MOV R, ES:[DI] CMP DS:[SI], R DEC SI DEC DI*

	CMPSW	*MOV R, ES:[DI]
		CMP DS:[SI], R
		ADD SI, 2
		ADD DI, 2*
			*MOV R, ES:[DI]
			CMP DS:[SI], R
			SUB SI, 2
			SUB DI, 2*

8. Accès à la mémoire vidéo du i8086

La mémoire vidéo est une zone en mémoire ou sont stockées les informations affichées sur l'écran. Une écriture sur cette zone mémoire affichera quelque chose sur l'écran (cf. fig 28). Cette zone débute à l'adresse **B800h:0000h** (B8000h). Chaque position (caractère sur l'écran) occupe **2 octets**. Le premier contiendra **le code ASCII** du caractère. Le second contiendra **le code couleur** de la position (voir Tableau 1) :

- Les 4 bits de poids **faible** donneront la couleur du caractère.
- Les 4 bits de poids **fort** donneront la couleur du fond.

Tableau 1: Code des couleurs

Hex	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Light Gray
8	Dark Gray
9	Light Blue
A	Light Green
B	Light Cyan
C	Light Red
D	Light Magenta
E	Yellow
F	White

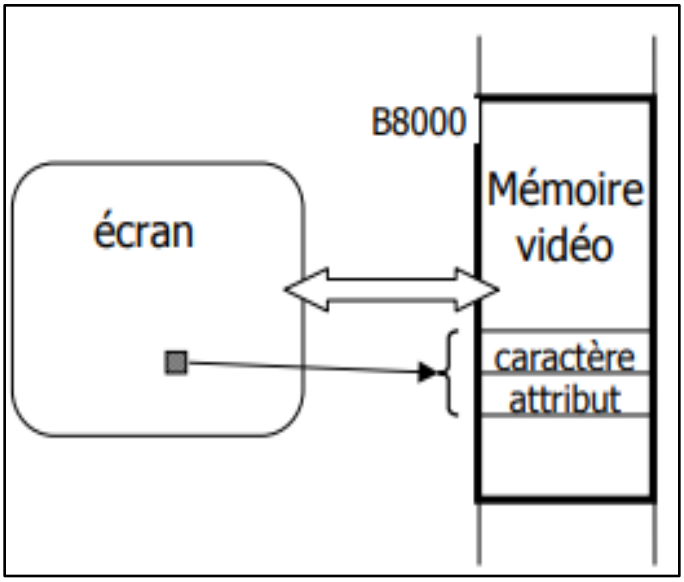


Figure 28: Mémoire vidéo du 8086

Exemple :

```
mov ax, 0B800h
mov ds, ax
mov [0] , 0E261h
```

affichera : a

On évite d'utiliser le registre **DS** pour les écritures en mémoire vidéo (**ES** est préférable). Si l'écran est configuré en mode **80 caractères par ligne**. Chaque ligne correspond à **160 octets** dans la mémoire vidéo. Pour écrire un "**A**" en **rouge sur noir** à la colonne **20** de la ligne **10**, il faudra écrire **'A'=65=41h** (code ascii de **A**) à la position **10*160 + 20*2 = 1640** et **04** dans la position suivante. La ligne **0** débute à la position mémoire **0**, la ligne **1** à la position **160**..., la ligne **10** à la position **1600** …

Pour initialiser **un mode graphique**, nous utilisons l'interruption 10h (10h : Interruptions graphiques) et la fonctions 00h dont voici les deux paramètres :

AH=00h :Numéro de la fonction initialiser un mode graphique

AL=N° du mode vidéo :Le numéro du mode vidéo (voir Tableau 2)

Et enfin : **INT 10h**

Tableau 2: Modes graphiques en utilisant la fonction 00h de l'interruption INT 10h

AL=	mode	Résolution texte	dimensions caractère	Résolution graphique	Couleurs	pages	Segment
00h	T	40x25	9x16	360x400	16	8	B800
01h	T	40x25	9x16	360x400	16	8	B800
02h	T	80x25	9x16	720x400	16	8	B800
03h	T	80x25	9x16	720x400	16	8	B800
04h	G	40x25	8x8	320x200	4	.	B800
05h	G	40x25	8x8	320x200	4	.	B800
06h	G	80x25	8x8	640x200	2	.	B800
07h	T	80x25	9x16	720x400	mono	.	B800
0Dh	G	40x25	8x8	320x200	16	8	A000
0Eh	G	80x25	8x8	640x200	16	4	A000
0Fh	G	80x25	8x14	640x350	mono	2	A000
10h	G	80x25	8x14	640x350	16	.	A000
11h	G	80x30	8x16	640x480	mono	.	A000
12h	G	80x30	8x16	640x480	16	.	A000
13h	G	40x25	8x8	320x200	256	.	A000

Exemple : Hello world

```
org 100h
; set video mode
mov ax, 3     ; text mode 80x25, 16 colors, 8 pages (ah=0, al=3)
int 10h       ; do it!
; cancel blinking and enable all 16 colors:
mov ax, 1003h
mov bx, 0
int 10h
; set segment register:
mov     ax, 0b800h
mov     ds, ax
; print "hello world"
; first byte is ascii code, second byte is color code.
mov [02h], 'H'
mov [04h], 'e'
mov [06h], 'l'
mov [08h], 'l'
mov [0ah], 'o'
mov [0ch], ','
mov [0eh], 'W'
mov [10h], 'o'
mov [12h], 'r'
mov [14h], 'l'
mov [16h], 'd'
mov [18h], '!'
; color all characters:
mov cx, 12  ; number of characters.
mov di, 03h ; start from byte after 'h'
c:  mov [di], 11101100b   ; light red(1100) on yellow(1110)
    add di, 2 ; skip over next ascii code in vga memory.
    loop c
; wait for any key press:
mov ah, 0
int 16h
ret
```

9. Procédures et Macros en 8086

9.1 Procédures

La procédure est une partie du code qui peut être **appelée** à partir de votre programme afin d'effectuer une tâche spécifique. **Les procédures** rendent le programme plus **structurel** et plus **facile** à comprendre. Généralement, **la procédure** revient **au même point** d'où elle a été appelée.

La syntaxe de la déclaration de procédure :

nom PROC
    ; code de la procédure
RET
nom ENDP

- **nom** : est le nom de la procédure, le même nom doit être en haut et en bas
- Les procédures sont appelées grâce à la directive **CALL**.
  Exemple : call print_string
- Les procédures doivent être déclarées à la fin du code avant le **END**.

Exemple :

Cet exemple appelle la procédure **m1**, fait **MOV BX, 5** et retourne à l'instruction suivante après **CALL**: **MOV AX, 2**

```
ORG   100h
CALL  m1
MOV   AX, 2
RET             ; retour au système d'exploitation
m1    PROC
MOV   BX, 5
RET             ; revenir à l'appelant
m1    ENDP
END
```

Un autre exemple d'une procédure qui reçoit deux paramètres dans les registres **AL** et **BL**, multiplie ces paramètres et retourne le résultat dans le registre **AX**.

Dans cet exemple, la valeur du registre **AL** est mise à jour chaque fois que la procédure est appelée, le registre **BL** reste inchangé, donc cet algorithme calcule **2** à la puissance de **4**, donc le résultat final dans le registre **AX** est **16** (ou **10h**).

```
ORG 100h
MOV AL, 1
MOV BL, 2
CALL m2
CALL m2
CALL m2
CALL m2
RET ; retour au système d'exploitation.
m2 PROC
MUL BL ; AX = AL * BL.
RET ; revenir à l'appelant.
m2 ENDP
END
```

9.2 Macros

Les macros sont comme des procédures, mais pas vraiment. **Les macros ressemblent** à des procédures, mais elles **n'existent** que jusqu'à ce que votre code soit compilé, après compilation toutes **les macros** sont **remplacées** par de vraies instructions. Si vous avez déclaré **une macro** et ne l'avez jamais utilisée dans votre code, le compilateur **l'ignorera** simplement.

Définition de macro :

nom MACRO [parametres,...]
<instructions>
ENDM

Contrairement aux procédures, **les macros** doivent être définies **au-dessus** du code qui l'utilise.

Exemple :

```
MyMacro MACRO p1, p2, p3
MOV AX, p1
MOV BX, p2
MOV CX, p3
ENDM
ORG 100h
MyMacro 1, 2, 3
MyMacro 4, 5, DX
RET
```

Le code dans l'exemple se **traduit** comme suit :

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Les macros sont développées directement dans le code, par conséquent, s'il y a **des étiquettes** à l'intérieur de la définition de macro, vous pouvez obtenir l'erreur «**Déclaration en double**» lorsque la macro est utilisée **deux fois ou plus**. Pour éviter ce problème, utilisez la directive **LOCAL** suivie

de noms **de variables**, **d'étiquettes** ou de **noms de procédure**, comme illustré dans l'exemple ci-dessous :

```
MyMacro2 MACRO
        LOCAL label1, label2

        CMP AX, 2
        JE label1
        CMP AX, 3
        JE label2
        Label1:
                INC AX
        Label2:
                ADD AX, 2
ENDM
ORG 100h
MyMacro2
MyMacro2
RET
```

9.3 Comparaison

- Lorsque vous souhaitez utiliser une procédure, vous devez utiliser l'instruction **CALL**, par exemple : **CALL MyProc**
- Lorsque vous souhaitez utiliser une macro, vous pouvez simplement taper **son nom**. Par exemple : **MyMacro**
- La procédure est située à une adresse spécifique en mémoire, et si vous utilisez la même procédure **100 fois**, le CPU **transférera le contrôle** à cette partie de la mémoire. **Le contrôle sera retourné** au programme par l'instruction **RET**. **La pile** est utilisée pour conserver **l'adresse de retour**.
- La macro est **développée** directement dans le code du programme. Donc, si vous utilisez la même macro **100 fois**, le compilateur **étend** la macro **100 fois**.
- Vous devez utiliser **la pile** ou tout autre **registre à usage général** pour passer des paramètres à la procédure
- Pour passer des paramètres à la macro, vous pouvez simplement les taper **après** le nom de la macro. Par exemple : **MyMacro 1, 2, 3**
- Pour marquer la fin de la macro directive **ENDM** suffit
- Pour marquer la fin de la procédure, vous devez taper **le nom de la procédure** avant la directive **ENDP**

9.4 Libraires et macros

Une librairie est un fichier contenant un ensemble de programmes (ou sous-programmes) facilitant la programmation. Les librairies sont appelées à l'aide de la directive **INCLUDE** suivie du nom du fichier. **Exemple : include emu8086.inc**

La librairie '**emu8086.inc**' fournie les macros suivantes :

- **PUTC caractere** : affiche le caractère donné en ASCII ou entre ''.
- **GOTOXY colonne, ligne** : positionne le curseur.
- **PRINT chaine** : affiche une chaine de caractère.
- **PRINTN string** : affiche une chaine de caractère + nouvelle ligne et retour chariot.

Exemple :

Pour utiliser l'une **des macros** expliquées dans le slide précédent, tapez simplement **son nom** quelque part dans votre code, et si nécessaire, **les paramètres**, comme montré dans l'exemple :

```
include emu8086.inc
ORG  100h
PRINT 'Hello World!'
GOTOXY 10, 5  ; Déplacer le curseur à la 5ème ligne, 10ème colonne
PUTC 65       ; 65 est le code ASCII pour 'A'
PUTC 'B'
RET           ; retour au système d'exploitation.
END           ; directive pour arrêter le compilateur.
```

9.5 Procédures emu8086.inc

La librairie '**emu8086.inc**' fournie également un ensemble de procédures qui doivent être déclarées à la fin du code avant le **END** avec **DEFINE_nom_procedure**

- **PRINT_STRING** : affiche une chaine de caractère se terminant par **zéro**, à partir de l'adresse donnée dans **DS:SI**, cette procédure doit être déclarée avec : **DEFINE_PRINT_STRING**
- **PTHIS** : affiche une chaine de caractère se terminant par zéro, la chaine est définit (**DB**) juste après l'appel à la procédure, cette procédure doit être déclarée avec : **DEFINE_PTHIS**

  Exemple:
  ```
  CALL PTHIS
  db 'bonjour', 0
  ```
- **CLEAR_SCREEN** : efface l'écran, cette procédure doit être déclarée avec : **DEFINE_CLEAR_SCREEN**
- **GET_STRING** : lit une chaine de caractère se terminant par **zéro**, la chaine est écrite à partir de l'adresse **DS:DI**, la taille du buffer est donnée dans **DX**, cette procédure doit être déclarée avec : **DEFINE_GET_STRING**
- **SCAN_NUM** : récupère un nombre saisit dans **CX**, cette procédure doit être déclarée avec : **DEFINE_SCAN_NUM**
- **PRINT_NUM_UNS** : affiche un nombre non-signé se trouvant dans **AX**, cette procédure doit être déclarée avec : **DEFINE_PRINT_NUM_UNS**
- **PRINT_NUM** : affiche un nombre signé se trouvant dans **AX**, cette procédure doit être déclarée avec : **DEFINE_PRINT_NUM** et **DEFINE_PRINT_NUM_UNS**

Exemple :

```
include 'emu8086.inc'
ORG 100h
LEA SI, msg1 ; ask for the number
CALL print_string
CALL scan_num ; get number in CX.
MOV AX, CX ; copy the number to AX.
; print the following string:
CALL pthis
DB 13, 10, 'You have entered: ', 0
CALL   print_num ;print number in AX.
RET      ; return to operating system.
msg1   DB  'Enter the number: ', 0
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
```

```
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
DEFINE_PTHIS
END        ; directive to stop the compiler.
```

10. Exercices

Exercice 1 :

Ecrivez un programme assembleur 8086 qui fait la soustraction de deux nombres de 32 bits chacun en utilisant des déclarations de variables.

Solution :

```
ORG 100h
.DATA
X dd 9ABCDEF0h
Y dd 12345678h
Z dd 0h
.CODE
MOV AX, X
MOV CX, Y
LEA BX, X
ADD BX, 2
MOV DX, [BX]
LEA BX, Y
ADD BX, 2
SUB AX, CX
SBB DX, [BX]
LEA BX, Z
MOV [BX], AX
MOV [BX+2], DX
RET
```

Exercice 2 :

Ecrivez une suite d'instructions en assembleur 8086 qui permet de remplir un tableau TAB3 qui sera le produit des éléments des deux tableaux de départ TAB1 et TAB2. Les éléments de TAB1 sont des mots (sur 16bits) signés et ceux de TAB2 sont des octets signés, Exemple :

TAB1 dw 1234H, 5678h, 9ABCh

TAB2 db -10h, 0FFh, 1Ah

Solution :

```
ORG 100h
LEA BX, TAB1
LEA SI, TAB2
LEA DI, TAB3
MOV CX, 3
ETQ :
MOV AL, [SI]
CBW
MOV DX, [BX]
IMUL DX
MOV [DI], AX
MOV [DI+2], DX
ADD BX, 2
INC SI
ADD DI, 4
LOOP ETQ
RET
TAB1 dw 1234H, 5678h, 9ABCh
TAB2 db -10h, 0FFh, 1Ah
TAB3 dd 3 dup(?)
```

Exercice 3 :

Ecrivez le programme assembleur 8086 qui calcule l'expression suivante : $X=(-A+2+B*C)/(D-3)$, tel que X,A,B,C,D sont des variables signées quelconque sur 16 bits.

Solution :

ORG 100h **MOV AX, A** **NEG AX** **ADD AX, 2** **CWD** **MOV BX, AX** **MOV CX, DX** **MOV AX, B** **IMUL C** **ADD AX, BX** **ADC DX, CX**	**MOV BX, D** **SUB BX, 3** **IDIV BX** **MOV X, AX** **RET** **A dw 1** **B dw 2** **C dw 3** **D dw 4** **X dw ?**

Exercice 4 :

Donnez la sortie du programme assembleur 8086 suivant :

ORG 100h
MOV AX, 1101110011110000b
MOV BX, 0DCh
MOV CX, 16
Etiq:
SHL AX, 2
RCR BX, 2
LOOP etiq
MOV AX, BX
RET

Solution :

AX = 1101110011110000 BX = 0000000011011100 1ière itération : AX = 0111001111000**00** **CF=1** BX = **01**00000000110111 CF=0 2ième itération : AX = 1100111100000**00** **CF=1** BX = **11**01000000001101 CF=1	3ième itération : AX = 0011110000000**00** **CF=1** BX = **11**11010000000011 CF=0 4ième itération : AX = 1111000000000**00** **CF=0** BX = **10**11110100000000 CF=1 5ième itération : AX = 1100000000000**00** **CF=1** BX = **01**10111101000000 CF=0
6ième itération : AX = 0000000000000**00** **CF=1** BX = **01**01101111010000 CF=0 7ième itération : AX = 0000000000000**00** **CF=0** BX = **00**01011011110100 CF=0 8ième itération : AX = 0000000000000**00** **CF=0** BX = **00**00010110111101 CF=0	9ième itération : AX = 0000000000000**00** **CF=0** BX = **10**00000101101111 CF=0 10ième itération : AX = 0000000000000**00** **CF=0** BX = **10**10000001011011 CF=1 11ième itération : AX = 0000000000000**00** **CF=0** BX = **10**10100000010110 CF=1
12ième itération : AX = 0000000000000**00** **CF=0** BX = **00**10101000000101 CF=0	15ième itération : AX = 0000000000000**00** **CF=0** BX = **00**10100010101000 CF=0

13ième itération : AX = 0000000000000**00** **CF=0** BX = **10**0010101000000 1 CF=0 14ième itération : AX = 0000000000000**00** **CF=0** BX = **10**1000101010000 0 CF=0	16ième itération : AX = 0000000000000**00** **CF=0** BX = **00**0010101000101010 CF=0 **Résultat :** AX = 0A2Ah

Exercice 5 :

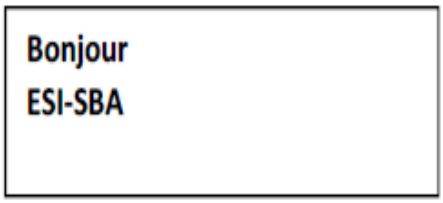
Donnez l'équivalent en assembleur 8086 de la portion de code suivante :
WHILE (B > 0) { R = A % B ; A = B ; B = R ; }
Tel que A, B, R sont des nombres non signés sur 16 bits et A >= B initialement, le résultat est dans A à la fin de la boucle.

Solution :

```
ORG 100h
MOV AX, A
MOV BX, B
ETQ :
CMP BX, 0
JZ FIN
MOV DX, 0
DIV BX
MOV AX, BX
MOV BX, DX
JMP ETQ
FIN :
RET
A DW 98
B DW 56
```

Exercice 6 :

Ecrire le programme assembleur qui affiche le message ci-contre par accès direct à la mémoire vidéo, sachant que le mode vidéo par défaut est **un mode texte**, affichant **80** caractères par ligne. La première lettre du message est le premier caractère de la ligne.



Solution :

```
org 100h
mov ax,0b800
mov es,ax
mov si,offset m1
mov di,0
mov cx,7
rep movsw
mov si,offset m2
mov di,160
mov cx,7
rep movsw
ret
m1 db 'B',0fh,'o',0fh,'n',0fh,'j',0fh,'o',0fh,'u',0fh,'r',0fh
m2 db 'E',0fh,'S',0fh,'I',0fh,'-',0fh,'S',0fh,'B',0fh,'A',0fh
end
```

Exercice 7 :

Ecrire **une procédure** (non récursive) en assembleur qui calcule **le produit** des nombres de **N jusqu'à M**, tel que :

- **N** est donné dans **AX**
- **M** dans **BX**
- le résultat dans **CX**.

Solution :

```
org 100h
mov ax,n
mov bx,m
call nm
ret
nm proc
mov cx,ax
push ax
et:
inc cx
cmp cx,bx
jg fin
mul cx
jmp et
fin:
mov cx,ax
pop ax
ret
nm endp
n dw 5
m dw 7
End
```

Exercice 8 :

Ecrire une macro qui lit une chaine de caractère saisie au clavier et l'enregistre dans une variable donnée **en paramètre** à la macro (le buffer et de **20 caractères**).
Demander à l'utilisateur **son nom** et l'enregistre dans une variable donnée à la macro (20 caractères maximum). Faire la même chose pour **le prénom**.

Solution :

```
include emu8086.inc
org 100h
lecture macro chaine
mov dx,20
mov di, offset chaine
call get_string
endm
printn "entrez votre nom :"
lecture nom
printn
printn "entrez votre prenom :"
lecture prenom
printn
ret
nom db 20 dup(?)
prenom db 20 dup(?)
define_get_string
```

Exercice 9 :

Demander à l'utilisateur **son âge** et l'enregistre **en mémoire** à l'aide d'une procédure de la librairie **emu8086**. **Convertir** l'âge obtenu en chaine de caractère, à l'aide d'une procédure.

Solution :

```
include emu8086.inc
org 100h
printn "entrez votre age :"
call scan_num
printn
mov age_num,cl
mov al,age_num
mov bx,offset age_string
call num_to_string
ret
age_num db ?
age_string db 3 dup(?)
define_scan_num
```

```
; l'âge est donné dans AL
; la chaine est écrite à l'adresse BX
num_to_string proc
mov cl,10
div cl
add ah,48
mov [bx+1],ah
add al,48
mov [bx],al
mov [bx+2],0
ret
num_to_string endp
```

Conclusion

Chaque microprocesseur reconnaît un ensemble d'instructions appelé **jeu d'instructions** (**Instruction Set**) défini par le fabricant. Nous avons appris **le jeu d'instructions** du microprocesseur 8086 d'Intel comme les instructions de transfert de données, les instructions logiques, les instructions arithmétiques et les instructions de branchement, etc.

Au fil du temps, le jeu d'instructions **x86** a subi de nombreux changements. La plupart d'entre eux ne sont que **des suppléments** au jeu d'instructions d'origine pour fournir de nouvelles fonctions.

Conclusion générale

Le langage d'assemblage donne **un contrôle direct** sur le processeur. Cela donne **un contrôle total** sur le système et permet d'écrire des programmes **plus rapides**, notamment par rapport aux langages de **haut niveau** (C++, Basic Pascal, etc.). En effet, ces langages rendent la programmation plus facile et plus rapide, mais **ils n'optimisent pas l'exécution** du code. Notons que nous pouvons **inclure** des programmes en langage assembleur dans un langage particulier (tel que Pascal ou C) pour **accélérer** l'exécution du programme. Dans ce polycopié, nous avons couvert les aspects importants de **la programmation assembleur du 8086**. De l'architecture interne du microprocesseur aux modes d'adressage, nous avons découvert comment tout cela fonctionne. Chaque instruction du jeu d'instructions du 8086 a été décrite et plusieurs exemples de programmes ont été donnés.

Pour aller plus loin, certains points restent à explorer :

1. **Interruptions matérielles** : Un des points qui restent à détailler c'est le cas lorsque les interruptions sont déclenchées par différents **matériels** (**interruptions matérielles**). Pour avoir du détail sur ce type d'interruptions, le lecteur peut se référer au cours suivant (Bejaoui, 2017).
2. **Code machine des instructions en mémoire** : Les règles de détermination des codes machines des instructions nécessitent d'être approfondies dans ce polycopié. L'ouvrage (Oumnad, 2007) donne des explications sur ces règles.

Références bibliographiques

Badsi, H. (2018). *Cours Introduction aux Systèmes d'Exploitation 2.* ESI-SBA.

Bejaoui, F. (2017, 11 11). *MICROPROCESSEUR 8086 ARCHITECTURE ET PROGRAMMATION* . Récupéré sur https://www.technologuepro.com/cours-genie-electrique/cours-2-microprocesseur-8086-architecture-programmation/

Benoît, M. (s.d.). *Apprendre l'assembleur*. Récupéré sur https://benoit-m.developpez.com/assembleur/tutoriel/

Brey, B. B. (1999). *The Intel microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium II processors: architecture, programming, and interfacing.* Prentice-Hall, Inc.

Catsoulis, J. (2005). *Designing Embedded Hardware.* O'Reilly Media, Inc.

CAUCHOIS, C. (1999/2000). *Support de cours : Assembleur d'INTEL.* Institut Universitaire de Technologie d'Amiens.

Coffron, J. (1983). *Programming the 8086/8088.* Sybex. Récupéré sur https://books.google.dz/books?id=6fcnAQAAIAAJ

Comparaison CISC/RISC. (s.d.). Récupéré sur https://www.irif.fr/~carton/Enseignement/Architecture/Cours/Architectures/risc.html

Computer Systems Performance Evaluation and Prediction. (Paul J. Fortier and Howard E. Miche). Computer Systems Performance Evaluation and Prediction.

Dandamudi, S. P. (2013). *Introduction to assembly language programming: from 8086 to Pentium processors.* Springer Science and Business Media.

Emu8086 documentation. (s.d.). Récupéré sur https://yassinebridi.github.io/asm-docs/

General purpose registers in 8086 microprocessor. (s.d.). Récupéré sur https://www.geeksforgeeks.org/general-purpose-registers-8086-microprocessor/

H. Megherbi Lessons. (s. d.). *Vidéos* [Chaîne YouTube]. YouTube. https://www.youtube.com/channel/UCF6pHtktlxEbDmfV1p2nqdQ/videos

Kant, K. (2007). *Microprocessors and Microcontrollers: Architecture, Programming and System Design 8085, 8086, 8051, 8096.* PHI Learning Pvt. Ltd.

Mazor, S. (2010). Intel's 8086. *IEEE Annals of the History of Computing*, 75--79.

MCIL BBA. (2020, avr 28). *Le microprocesseur 8086* [Vidéo]. YouTube. https://www.youtube.com/watch?v=iEb72cXKUn4&t=38s

Microprocessor - 8086 Overview. (s.d.). Récupéré sur https://www.tutorialspoint.com/microprocessor/microprocessor_8086_overview.htm

Modèle d'architecture de Von Neumann. (s.d.). Récupéré sur https://info.blaisepascal.fr/nsi-modele-darchitecture-de-von-neumann

Oumnad, A. (2007). *Microprocesseurs de la famille 8086.* Ecole des Ingénieurs Mohamadia.

Paul J. Fortier, H. E. (2002). *Computer Systems Performance Evaluation and Prediction.* Digital Press.

Preux, P. (1995). *Assembleur i8086.* IUT informatique du Littoral.

Rahmoun, A. (2017). *Cours système d'exploitation 2.* ESI-SBA.

Simple 8086 Assembly Language Programs with Explanation. (s.d.). Récupéré sur https://www.elprocus.com/8086-assembly-language-programs-explanation/

Triebel, W. A. (1991). *The 8088 and 8086 microprocessors: programming, interfacing, software, hardware, and applications.* prentice hall.

Table des matières