

Année Universitaire 2011/2012

**Université de Tunis El-Manar
Faculté des Sciences de Tunis
Département des Sciences de l'Informatique**

**Sections : IF3
Module : Systèmes d'Exploitation**

Le langage Assembleur

1. Introduction

Le processeur d'un ordinateur ne peut exécuter que les instructions écrites dans un langage binaire appelé Langage machine. Ce langage est propre à chaque famille de processeurs. Etant donné sa forme (suite de bits), un programmeur ne peut pas développer ses programmes en utilisant un tel langage. Il a donc besoin de langages qui puissent exprimer des instructions en utilisant un code autre que le binaire.

Parmi ces langages, on trouve les langages Assembleur qui sont des langages très proches du processeur. Ainsi, ils restent très liés, comme les langages machines, à une famille de processeurs. Par exemple les processeurs Intel ont un langage assembleur qui est différent de celui des processeurs Motorola. Ces langages permettent de développer des programmes en utilisant ce que l'on appelle du code mnémotechnique (utilisation de symboles pour décrire les instructions d'un programme). Ce code mnémotechnique permet de développer des programmes qui sont à la fois performants (proches du processeur) et compréhensibles par le programmeur, alors que les programmes exécutables (binaires) sont totalement illisibles même pour un informaticien.

Contrairement aux langages évolués, le langage d'assemblage permet d'accéder à toutes les ressources (registres) et aux facilités de traitement d'un ordinateur. Il permet au programmeur de tirer profit de toutes les capacités d'un ordinateur, notamment en ce qui concerne le processeur. Ainsi, le langage assembleur est intimement lié au processeur et les programmes en assembleur dépendent donc directement de la machine qui les exécute.

Le langage assembleur est très proche du langage machine ; il dépend donc fortement du type de processeur. Ainsi il n'existe pas **un** langage assembleur, mais un langage assembleur par type de processeur.

2. Intérêts dans l'utilisation d'un langage assembleur

Etant donné qu'un langage assembleur est fortement lié au processeur, ceci permet de développer des programmes performants. Un programme développé en assembleur est exécuté bien plus rapidement que s'il avait été développé en langage évolué (en langages C ou JAVA par exemple). Cet aspect de performance est très important pour tous les programmes où le temps d'exécution est très important. En fait, étant donné que vous programmez directement le processeur, vous pouvez vous-même effectuer des optimisations sur votre code, suivant les cas. Cette optimisation est impossible à obtenir avec un compilateur.

De plus, un programme développé en assembleur occupe aussi beaucoup moins d'espace en mémoire que s'il avait été développé en langage évolué.

Certains programmes ou fonctions ne peuvent être programmés qu'en assembleur : c'est le cas des commandes fines de circuits, de périphériques, d'E/S, de drivers, de la programmation de cartes à puces, etc.

L'apprentissage de l'assembleur est une manière d'apprendre le fonctionnement de la machine au niveau physique.

Malgré tous ces avantages, le développement de programmes en assembleur pose quelques problèmes. L'assembleur décompose la logique d'un programme en actions élémentaires, c'est-à-dire en instructions de base du microprocesseur. Comme résultat, nous obtenons des programmes plus longs et plus fastidieux à écrire qu'en langage évolué, plus difficile aussi à modifier et à corriger, surtout s'ils ne sont pas écrits correctement et avec rigueur.

De plus, les programmes assembleurs dépendent de la machine pour laquelle ils ont été conçus. Ainsi, un programme écrit pour un microprocesseur donné ne fonctionnera pas sur un microprocesseur différent ou d'une famille différente. Par conséquent, les programmes en assembleur ne sont pas portables.

L'assembleur étant un langage de très bas niveau, il présente quelques difficultés mais qui sont facilement maîtrisables :

- L'écriture de code pourrait être longue (et ennuyeuse);
- Les erreurs ne sont pas faciles à repérer et supprimer;
- Il n'est pas toujours facile de comprendre et de modifier du code;
- Un programme assembleur n'est pas portable vers une autre architecture ;

Il existe plusieurs types d'assembleurs, parmi lesquels nous pouvons citer : **asm** (de Arrowsoft), **masm** (Microsoft Assembleur), **tasm** (Turbo Assembleur de Borland). La syntaxe est la même pour ces trois assembleurs. Il existe également **gas** (gnu c compiler), et **nasm** (Netwide Assembleur). Mais la syntaxe de ces deux derniers est assez différente.

3. Différences entre un compilateur et un assembleur

Tout microprocesseur contient en lui-même ce que l'on appelle un *jeu d'instructions*. Ces instructions, de très bas niveau, définissent des opérations très simples, comme par exemple, "*mettre telle valeur dans un emplacement précis de la mémoire*", ou "*additionner telle valeur avec telle autre valeur et mettre le résultat dans un emplacement précis de la mémoire*". Partant d'un programme source écrit en assembleur, l'assembleur va convertir un fichier source en un fichier exécutable contenant le code numérique binaire de chaque instruction, et donc compréhensible par le microprocesseur. L'assembleur ne fait que traduire le fichier source du langage humain vers le langage binaire. Par exemple, additionner 2 et 3 produit le code suivant en assembleur:

```
mov AX,2  
add AX,3
```

Une fois traduit en instructions machine 16 bits, ce code deviendra comme suit :

```
10111000 00000010  
00000101 00000011
```

(B8020503 en hexadécimal)

Par contre, un compilateur analyse un fichier source écrit en un langage dit évolué et transforme chaque instruction propre à ce langage en une suite d'instructions machines. On peut dire qu'il convertit le fichier source en programme assembleur, et ce n'est qu'ensuite qu'il génère le fichier exécutable contenant les codes binaires. En fait, le compilateur effectue une étape de plus que l'assembleur, c'est la transformation "fichier source écrit en langage évolué" vers "un fichier source écrit en assembleur". C'est justement l'inconvénient du compilateur : la transformation n'est pas toujours aussi bonne que celle que l'on aurait écrite directement en assembleur. Ceci fait que les programmes assembleurs sont toujours plus performants que les programmes écrits avec des langages évolués.

4. Assemblage et programme assembleur

L'assembleur permet de contrôler directement un processeur. Cela permet d'avoir une totale maîtrise du système et surtout permet de faire des programmes rapides par rapport aux langages de haut niveau (C++, Visual Basic, JAVA, etc.). En effet, bien que ces langages permettent de faire des programmes facilement et rapidement, ils n'optimisent pas le code d'exécution. Cela engendre donc des programmes (beaucoup) plus volumineux. Notons que l'on peut insérer de l'assembleur dans certains langages (Pascal et C par exemple) pour accélérer l'exécution du programme. L'exemple suivant illustre l'utilisation de l'assembleur dans un programme écrit en langage C :

```
void main(void) {  
int A = 20;  
asm{  
MOV AX, A  
SHL AX, 1  
}  
printf("AX = %d\n", _AX);  
}
```

En fait, la meilleure chose à faire pour pouvoir utiliser à la fois l'assembleur et le langage C (ou tout autre langage évolué), c'est de créer des fonctions en asm, de les assembler en tant que fichier objet, d'écrire des fichiers en-têtes (extension .h) pour déclarer vos fonctions au compilateur, puis de linker (édition de liens) vos fichiers objets à votre exécutable final.

4.1. La phase d'assemblage

Le langage assembleur est un langage bas niveau utilisant des codes, mots clés ou symboles mnémoniques. Ces mots clés, ou mnémoniques, sont souvent la compression d'un mot ou d'une expression en anglais présentant l'action de l'instruction. Par exemple sur les processeurs 8086, l'instruction **MUL** réalise l'opération arithmétique de multiplication (**MUL**tiplier).

Demander à un processeur d'exécuter un programme assembleur n'est donc pas possible, car les circuits d'un processeur ne comprennent que le binaire. Pour cela, il faut traduire les instructions d'un programme assembleur en instructions machine du processeur associé. Cette traduction est assurée par un logiciel spécifique appelé l'**ASSEMBLEUR**. Cette opération de traduction est appelée **ASSEMBLAGE**.

Programme objet
Programme source

Assemblage

Programme exécutable

Edition de liens
(Link)

Figure 1. Etapes d'assemblage d'un programme assembleur.

Un programme écrit en langage d'assemblage doit subir un certain nombre de transformations avant de pouvoir être exécuté. Les figures 1 et 2 présentent les différentes étapes (ou phases) du traitement d'un programme en langage d'assemblage.

La première phase de traitement est l'assemblage. Le programme source est traduit en langage machine et le programme objet ainsi obtenu est stocké dans un fichier. La seconde phase de traitement est l'édition des liens. La troisième phase du traitement est l'exécution du programme proprement dite qui peut utiliser des données et qui produit des résultats.

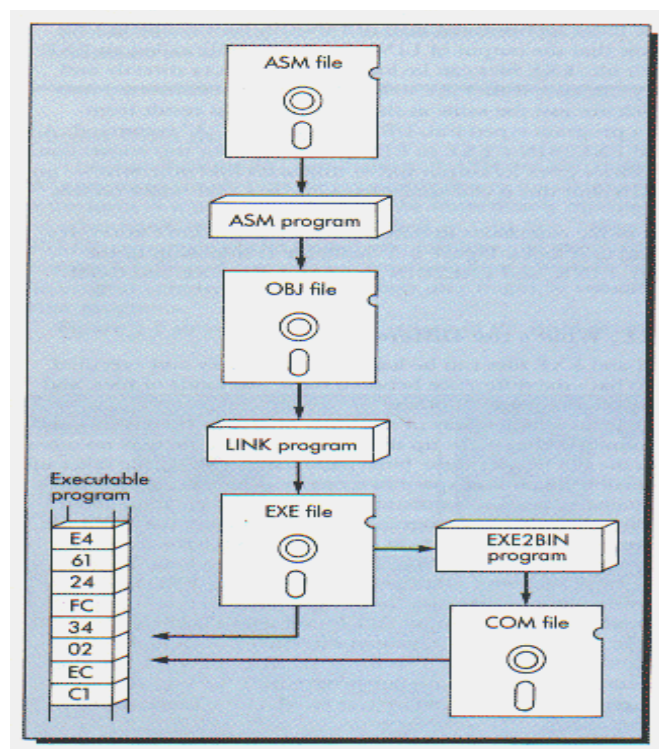


Figure 2. Etapes d'assemblage d'un programme assembleur.

4.2. La phase d'édition de liens

Le fichier .OBJ contient le produit binaire (résultat) de l'assemblage. Mais ce fichier n'est pas encore dans un format exécutable par le processeur. Ainsi, le système DOS ne peut pas le charger en mémoire centrale et encore moins l'exécuter. Pour ce faire, nous avons

encore deux étapes à réaliser. La première est celle de l'édition de liens (LINK, c'est-à-dire "lier" en anglais).

L'utilitaire LINK (ou TLINK selon les assembleurs) continue le travail commencé par l'assembleur en lisant le fichier .OBJ pour créer un fichier "exécutable" (.EXE). On peut "lier" plusieurs fichiers .OBJ en un seul et unique fichier .EXE, et c'est pourquoi on le nomme "éditeur" de liens.

Pour générer le fichier exécutable, à partir du fichier objet créé par le compilateur, il donc faut un linker. Il existe **link** (Microsoft), **tlink** (Borland), etc. Alors que l'assembleur se charge uniquement de transcrire en langage machine le code source du programme, l'éditeur de liens permet d'ajuster les adresses des différentes parties d'un programme assembleur.

4.3. La phase de chargement du programme à exécuter

On ne peut exécuter un programme que s'il a été préalablement chargé en mémoire centrale. L'assembleur range le programme objet en mémoire secondaire (disque) et l'éditeur de lien en fait un fichier exécutable. Mais avant de pouvoir exécuter ce programme, il faut le charger en mémoire centrale. Chaque instruction du programme possède une adresse relative qui lui a été attribuée par l'assembleur. Ainsi, il serait simple de placer le programme en mémoire de façon à ce que les adresses relatives correspondent aux adresses réelles des instructions (appelées adresses absolues). Ceci n'est pas possible car une partie de la mémoire centrale est occupée en permanence par le système d'exploitation. Pour cela, le programme ne pourra pas être chargé en mémoire à partir de l'adresse zéro ou de l'adresse 100h. Si la zone de mémoire disponible pour le programme commence, par exemple, à l'adresse physique 2C8F0 (2C8F:0000), la première instruction du programme (d'adresse relative zéro) sera placée à l'adresse 2C8F0 pour un programme .exe et l'adresse 2C9F0 pour un programme.com (d'adresse relative 100h) et les instructions suivantes aux adresses qui suivent: on dit alors que le chargeur a traduit le programme.

5. Les registres du microprocesseur 8080

Un registre est un élément de mémoire interne du microprocesseur. Sa capacité est très réduite, car son rôle n'est pas de stocker de données, mais plutôt des adresses ou les résultats intermédiaires de calculs. La plupart des commandes de l'assembleur utilisent les registres. Il est donc important de connaître leurs utilités respectives. Un registre n'est qu'une zone de stockage de chiffres binaires, à l'intérieur même du processeur. Le 8086 dispose d'un certain nombre de registres qui sont:

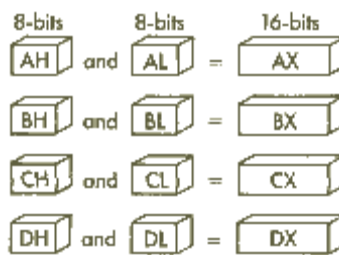
AX	accumulateur
BX, CX, DX	registres banalisés
DI, SI, BP	registres d'index
IP	pointeur d'instructions
SP	pointeur de pile
CS	registre de segment de code
DS	registre de segment de données
SS	registre de segment de pile
ES	extra segment
FLAGS	registre d'état.

Tous ces registres sont des registres 16 bits. Néanmoins les registres **AX**, **BX**, **CX**, **DX** peuvent être subdivisés en deux registres de huit bits supérieurs et inférieurs, et être référencés par les notations **xH** et **xL**: par exemple AH et AL pour AX ou BH et BL pour le registre BX et ainsi de suite. En d'autres termes $AX = 256 * AH + AL$.

5.1. Registres de données

Il existe quatre registres de 16 bits : AX, BX, CX, DX. Ils servent pour le calcul et le stockage. Cependant, chacun d'eux peut être scindé en deux registres de 8 bits. L'octet de gauche d'un registre 16 bits est dit de poids fort, et il est le plus significatif tandis que celui de droite (le moins significatif) est dit de poids faible. Ainsi, AX peut être décomposé en deux registres: AH et AL. Il en va de même des registres BX, CX et DX. D'une manière générale, les programmes en assembleur placent dans ces registres des nombres issus de la mémoire, effectuent des opérations sur ces registres et replacent ensuite les résultats en mémoire.

Peu de programmes s'écartent de ce schéma de base. Quant aux registres autres que AX, BX, CX et DX, ils sont souvent utilisés pour les déplacements de nombres entre la mémoire et ces 4 registres. Ces registres sur 16 bits peuvent être utilisés sous la forme de 2 registres 8 bits chacun. Les registres 8 bits correspondant à AX, BX, CX et DX se nomment AL, AH, BL, BH, CL, CH, DL, DH.



Par conséquent, on peut inscrire une valeur quelconque sur 8 bits dans le registre BL, sans que cela n'affecte le contenu du registre BH. Les registres de données sont en général interchangeables : malgré leurs fonctions, ils sont en effet d'un usage général. Cependant, on utilise souvent AX pour des opérations simples avec des instructions qui adressent implicitement ce registre sans qu'il soit nécessaire de le spécifier.

Les différents registres sont utilisés pour assurer les fonctions suivantes :

- **AX** est l'accumulateur;
- **BX** est le registre de " Base " ; il peut parfois être utilisé comme base d'indexage.
- **CX** est le registre " Compteur ";
- **DX** est le registre propre aux manipulations des " Données ". Dans les opérations sur 32 bits, avant l'apparition du processeur INTEL 386, les 16 bits de poids fort sont placés dans CX et ceux de poids faible, dans DX.

Ce sont donc les tâches spécifiques des registres CX, DX, AX et BX. Mais rappelons-le : ils sont interchangeables !

5.2. Registres d'index

Les registres d'index DI et SI et les registres "pointeurs de pile" (BP et SP) sont des registres spécifiques au langage assembleur. Ils sont définis comme suit :

- **SI** est le registre Source (Source Index) et DI le registre de destination (Destination Index) dans les opérations d'indexage.
- **BP** est le registre de base de la pile. C'est lui qui pointe sur l'adresse de base de la pile.
- **SP** (Stack pointer), est le pointeur de pile qui pointe sur le sommet de la pile. On dit de ces registres d'indexage qu'ils "pointent" sur un emplacement mémoire. Ainsi, si SI contient F000h, on dira qu'il pointe sur l'offset F000h.

5.3. Registres de segment

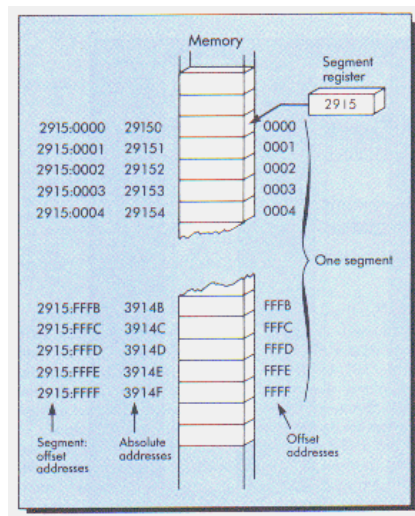
Dans l'environnement du système d'exploitation DOS, nous devons donc être capables de représenter des nombres allant de 0 à 1 048 575. Pour générer les adresses nécessaires à une mémoire de 1 méga-octet il faut donc avoir 20 bits. Le processeur x86 utilise des mots de 20 bits pour adresser la mémoire réelle. Le jeu complet des 1 048 576 adresses différentes est appelé "espace adressable de 1 méga-octet". Cependant l'unité centrale du x86 ne comporte aucun registre de 20 bits puisque les registres sont de 16 bits. Un registre unique ne peut donc référencer que 64K de mémoire ($2^{16} = 65\,536$). Il est donc nécessaire de combiner deux registres pour accéder à la mémoire. Le x86 en mode réel combine un des neuf registres généraux avec l'un des quatre registres de segments pour former une adresse de 20 bits. Les registres de segment sont alors appelés :

- segment code (CS) ;
- segment données (DS) ;
- segment extra (ES) ;
- segment pile (SS).

À chaque type d'accès en mémoire correspond par défaut un registre de segment et parfois un registre général. Par exemple, CS et IP sont combinés pour accéder à la prochaine instruction à exécuter sous la forme CS:IP. Les registres SS et SP sont combinés en SS:SP pour toutes les opérations concernant la pile. Il est parfois possible d'utiliser un registre de segment autre que celui utilisé par défaut en le spécifiant explicitement.

Si nous mettons bout à bout deux registres de 16 bits, nous obtenons une adresse sur 32 bits. Nous pourrions ainsi adresser 4 gigaoctets ($2^{32} = 4\,294\,967\,296$). Pour adresser 1 méga-octet, nous n'avons besoin que de 4 bits en plus des 16 bits d'un registre général. Cette combinaison est obtenue en décalant le registre de segment de 4 bits et en ajoutant le résultat au contenu du registre général pour obtenir l'adresse sur 20 bits. Le résultat est appelé adresse effective (EA). L'adresse effective est placée sur le bus d'adresse afin de pouvoir accéder à l'emplacement mémoire correspondant. On peut donc considérer l'adresse effective comme constituée de deux parties. Le registre de segment définit une zone mémoire de 64K, et le registre général spécifie un déplacement à partir de l'origine de ce segment de 64K (c'est-à-dire une adresse sur 16 bits à l'intérieur de ce segment). L'adresse effective est exprimée en donnant le registre segment et le registre général séparés par deux points. Une adresse mémoire peut donc être symboliquement représentée par DS:SI ou explicitement par : 2915:0004. Le segment de 64K est défini par le registre DS et le déplacement dans ce segment est contenu dans le registre SI. L'adresse du segment est 2915 et le déplacement dans le segment est 0004 en hexadécimal. L'adresse effective est donc 29154 hexadécimal. Notez

que le décalage de l'adresse du segment est obtenu en ajoutant un zéro à droite. Ainsi, 2915:0004 devient $2915 + 0004$, soit 29154, qui est l'adresse effective. Le même calcul se fait pour le cas de 2915:FFFB; $2915 + \text{FFFB} = 3914\text{B}$.



5.4. Pointeur d'instructions

Le registre IP (Instruction pointer) est le pointeur d'instructions. Il indique la prochaine instruction à exécuter par le processeur. Dans certains autres processeurs, on l'appelle aussi le "Program Counter" - le compteur d'instruction de programme ou le compteur ordinal). Il a la responsabilité de guider le processeur lors de ses déplacements dans le programme en langage machine, instruction par instruction. Le registre IP est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante. Le x86 dépend entièrement du registre IP pour connaître l'instruction suivante.

5.5. Registre Drapeau

Le registre Drapeau est un ensemble de 16 bits. La valeur représentée par ce nombre de 16 bits n'a aucune signification en tant qu'ensemble: ce registre est manipulé bit par bit, certains d'entre eux auront une influence sur le comportement du programme. Les bits de ce registre sont appelés « *Indicateurs* » ou « *Flags* », et nous pouvons les regrouper en deux catégories:

5.5.1. Indicateurs d'état

Bit	signification	abréviation	TDEBUG
0	"Carry" ou Retenue	CF	c
2	Parité	PF	p
9	Retenue auxiliaire	AF	a
6	Zéro	ZF	z
7	Signe	SF	s

11	débordement (overflow)	OF	o
----	---------------------------	----	---

5.5.2. Indicateurs de contrôle

Bit	signification	abréviation	TDEBUG
8	trap	TF	
9	interruption	IF	i
10	direction	DF	d

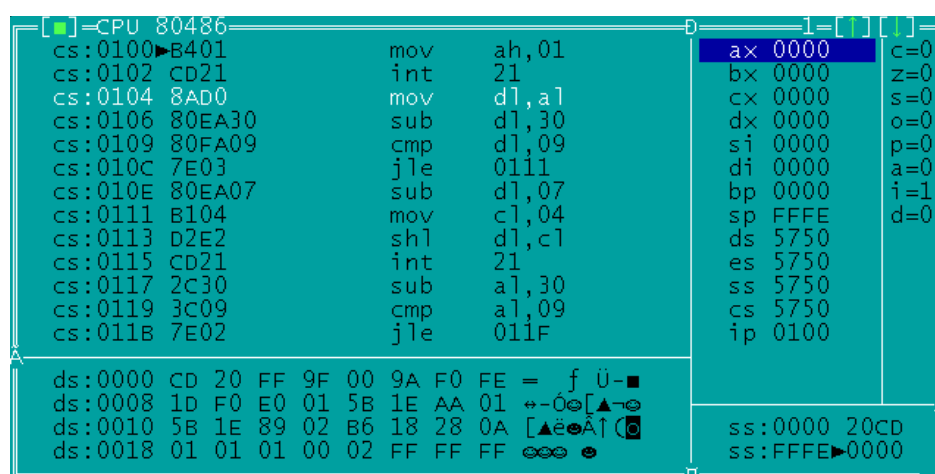
Les bits 1, 5, 12, 13, 14, 15 de ce registre FLAG de 16 bits ne sont pas utilisés. Les indicateurs sont en relation directe avec certaines instructions: Les instructions arithmétiques, logiques et de comparaison modifient la valeur des indicateurs (voir manuel de l'assembleur).

DEC AX ; affecte les indicateurs CF, ZF, SF, OF, AF
 CMP AX,1 ; affecte les indicateurs CF, ZF, SF, OF, AF

Les instructions conditionnelles testent la valeur des indicateurs et agissent en fonction du résultat.

JA - teste les indicateurs CF et ZF
 JNE - teste l'indicateur ZF
 JS - teste l'indicateur SF

Lors d'une instruction non arithmétique ou logique, les indicateurs CF et OF sont remis à 0. Nous verrons que dans certains cas les indicateurs ne sont pas modifiés, dans certains cas ils le sont et, parfois, ils sont indéfinis...



Le registre Drapeau dans TDEBUG se situe en haut à droite des autres registres

5.5.3. Signification des principaux indicateurs

L'indicateur CF (CARRY ou retenue): Il sera mis à 1 s'il y a eu retenue lors de la dernière instruction arithmétique.

L'indicateur OF (OVERFLOW ou débordement) : Il sera mis à 1 si le résultat d'une addition de 2 nombres positifs donne un nombre négatif et inversement. En règle générale, cet indicateur est activé quand le signe change alors qu'il ne devrait pas. En arithmétique non signée, la valeur de cet indicateur n'aura pas de signification.

L'indicateur ZF (ZERO): Il sera mis à 1 (activé, set) si le résultat d'une instruction arithmétique a donné zéro et il le restera jusqu'à la prochaine instruction arithmétique.

L'indicateur SF (SIGN ou signe) Il sera mis à 1 si le résultat d'une instruction a donné un nombre négatif (bit de poids fort =1), il sera mis à 0 dans le cas contraire.

L'indicateur DF (DIRECTION) Il est modifié par une instruction CLD ou STD et pas par le résultat d'une instruction (voir manipulation de chaînes de caractères).

L'indicateur PF (PARITÉ) Il est mis à 1 si le résultat d'une opération contient un nombre pair de bits 1.

5.5.4. Tests et branchements (sauts)

Pour effectuer un branchement (un saut dans un programme assembleur), il faut dans beaucoup de cas faire au préalable une comparaison, c'est-à-dire une opération de test. Il existe une seule instruction de cette catégorie permettant de comparer deux registres ou emplacements de mémoire: c'est l'instruction **CMP**. Les opérations de comparaison sont toujours suivies d'une instruction de branchement conditionnel car elles affectent les indicateurs du registre *Drapeau*. Le résultat de la comparaison est indiqué par les indicateurs. Les cinq formes disponibles de cette instruction sont:

CMP reg, imm	CMP AX, 0Ah
CMP reg, mem	CMP AX, [BX]
CMP mem, imm	CMP [BX], 0Ah
CMP mem, reg	CMP [BX], AX
CMP reg, reg	CMP AX, BX

"reg" étant un registre de 8 ou 16 bits (sauf un registre de segment);

"mem" est une adresse (ou un identificateur);

"imm" représente une valeur immédiate (constante).

Note: une opération de comparaison est en fait une soustraction qui n'affecte aucune opérande (voir instructions arithmétiques):

$$\text{CMP A,B} \iff (\text{A}-\text{B})$$

Dans le cas de comparaison de caractères, le microprocesseur 8086 effectue la soustraction des codes ASCII des caractères.

5.5.4.1. Conditions de branchement

Il existe deux catégories d'instructions pour tester les conditions de branchements. La première peut s'exercer sans le recours de comparaisons préalables. On peut, par exemple, utiliser l'instruction de branchement JNZ après une comparaison mathématique du type DEC ou INC.

Le deuxième groupe comprend des tests constitués de combinaisons d'indicateurs que l'on utilise généralement en relation avec CMP (comparer) pour construire des boucles. Cette instruction de comparaison est capable de calculer des relations plus complexes entre deux valeurs : comme par exemple, savoir si un nombre est inférieur ou égal à un autre.

5.6. Le compteur d'instructions (IP)

Puisque le pointeur d'instruction est le registre servant à exécuter les instructions, il est nécessairement modifié par un saut conditionnel. Si la condition du test est remplie, on place une nouvelle valeur dans le compteur du programme. Si, par contre, les conditions du test ne sont pas remplies (comme dans le cas d'une instruction JNZ (Jump if Not Zero) avec l'indicateur Z activé), alors le pointeur d'instruction suit son cours normalement, et le programme se déroule comme si aucun saut conditionnel n'avait été rencontré (exécution séquentielle sans saut). Si les conditions du test sont remplies, l'octet de données (c'est-à-dire l'octet qui suit l'instruction de test) est additionné à l'IP (Instruction Pointer). Par exemple, si l'IP contient 102h, et si le processeur se rend compte que l'indicateur Z est désactivé (en lisant l'octet complémentaire, disons 10h, de JNZ), la nouvelle valeur de l'IP, après une instruction JNZ 10, sera le résultat de l'addition : 10h plus 102h. L'instruction qui sera exécutée immédiatement après le JNZ est celle qui se trouve à l'adresse 122h (102h+10h).

Les tableaux suivants résument l'ensemble des registres qui peuvent être utilisés en programmation assembleur.

Registres de Segment

Nom	Taille	Fonction
Cs (Code Segment)	16 bits	Mémoire le segment où se trouve le code en cours d'exécution (ne peut pas être modifié par le programme).
Ds (Data Segment)	16 bits	Mémoire le segment où se trouve les données du programme.
Ss (Stack Segment)	16 bits	Mémoire le segment où se trouve la pile de données du programme
Es (Extra Segment)	16 bits	Ce segment peut être utilisé pour faire ce que l'on veut. Par exemple, il peut pointer sur la mémoire écran.

Registres de Travail

Nom	Taille	Fonction
Ax (Accumulateur)	16 bits	On l'utilise généralement pour des opérations arithmétiques, telles que MUL (multiplication) ou DIV (division). Ax peut se diviser en deux sous-registres de 8 bits. Ah représente les 8 premiers bits, et Al les 8 derniers.
Bx (Base)	16 bits	Bx est utilisé lors de l'accès à une zone mémoire sous forme de tableau ; il représente l'indice de ce tableau. Par exemple, on écrira <code>MOV Dx, Es:[Bx]</code> . Bx peut se diviser en deux sous-registres de 8 bits. Bh représente les 8 premiers bits, et Bl les 8 derniers.
Cx (Compteur)	16 bits	Lors de l'appel d'instructions comme REP (répéter) ou LOOP (boucle), c'est le registre Cx qui est lu. Cx peut se diviser en deux sous-registres de 8 bits. Ch représente les 8 premiers bits, et Cl les 8 derniers.
Dx (Données)	16 bits	Ce registre est généralement utilisé pour stocker des données provisoires. Dx peut se diviser en deux sous-registres de 8 bits. Dh représente les 8 premiers bits, et Dl les 8 derniers.

Registres d'Offset

(à combiner avec une adresse de segment)

Nom	Taille	Fonction
Si (Source Index)	16 bits	Lors de l'utilisation d'opérations sur les chaînes de caractères, comme MOVSB ou SCASB, Ds:[Si] désigne la variable 'source'.
Di (Destination Index)	16 bits	Lors de l'utilisation d'opérations sur les chaînes de caractères, comme MOVSB ou SCASB, Es:[Di] désigne la variable 'destination'.
Bp (Base Pointeur)	16 bits	Bp a un rôle proche de celui de Bx, mais il est généralement utilisé avec le segment de pile (Ss:[Bp]).
Ip (Instruction Pointeur)	16 bits	Cs:[Ip] indique la prochaine instruction à exécuter. Tout comme Cs, Ip ne peut être manipulé par le programme exécuté.

Sp (Stack Pointeur)	16 bits	Ss:[Sp] indique le dernier élément de la pile. Chaque opération PUSH (empiler) ou POP (dépiler) modifie le registre Sp.
----------------------------	---------	---

6. Structure d'un programme assembleur

Comme pour tout langage de programmation, le fichier contenant le programme source doit être saisi de manière rigoureuse, en respectant un certain nombre de règles. Chaque définition et chaque instruction doivent ainsi s'écrire sur une nouvelle ligne (pour que l'assembleur puisse différencier les différentes instructions) Le fichier source contient:

1. Un nom du programme sous **TITLE** suivi d'un nom. Cette partie est facultative.
2. Une partie pour déclarer une pile qui est définie dans le segment de pile délimité par les directives **SEGMENT STACK** et **ENDS**
3. Des définitions de données déclarées par des directives. Celles-ci sont regroupées dans le segment de données délimité par les directives **SEGMENT** et **ENDS**
4. Puis sont placées les instructions (qui sont en quelque sorte le cœur du programme), la première devant être précédée d'une étiquette, c'est-à-dire par un nom qu'on lui donne. Celles-ci sont regroupées dans le segment d'instructions délimité par les directives **SEGMENT** et **ENDS**
5. Enfin, le fichier doit être terminé par la directive **END** suivi du nom de l'étiquette de la première instruction (pour permettre au compilateur de connaître la première instruction à exécuter (Les **points-virgules** marquent le début des commentaires, c'est-à-dire que tous les caractères situés à droite d'un point virgule seront ignorés) Voici à quoi ressemble un fichier source (fichier .ASM):

TITLE nomprogramme ; cette directive permet de nommer votre programme

Pile **SEGMENT STACK**; segment de pile dont le nom est Pile (ou tout autre nom)

; déclarer la pile et sa taille

Pile **ENDS**

Donnees **SEGMENT** ; voici le segment de données dont l'étiquette est Donnees
; (ou tout autre nom)

; Placez ici les déclarations de données

Donnees **ENDS**; ici se termine le segment de donnees

Lecode **SEGMENT** ; voici le segment d'instructions dont l'étiquette est Lecode
; (ou tout autre nom)

ASSUME DS:donnee, CS: Lecode

debut: ; placez ici votre première instruction (son étiquette est nommée debut)

; placez ici le reste de vos instructions

Lecode ENDS ; fin du segment d'instructions

END debut ; fin du programme suivie de l'étiquette de la première instruction

Voici un exemple de petit programme que vous pouvez tester sur machine :

```
Title exemple ; exemple de programme
; Le segment de données
Data Segment
    Message DB "Bonjour, le monde !$"
Data EndS

; Le segment de code
Code Segment
    Assume Cs : Code, Ds : Data

Main Proc
; Affichage du message
    Mov Ah, 09h
    Mov Dx, Offset Message
    Int 21h

; Indicateur de fin du programme,
; et appel à MS-DOS
    Mov Ax, 0C00h
    Int 21h
Main EndP
Code EndS
End Main
```

6.1. Déclaration d'un segment

Pour l'instant, oublions le segment de pile. Les données sont regroupées dans une zone de la mémoire appelée segment de données, tandis que les instructions se situent dans un autre segment qui est le segment d'instructions ou segment code. Le registre DS (Data Segment) contient l'adresse du segment de données, tandis que le registre CS (Code Segment) contient celle du segment d'instructions. C'est la directive ASSUME qui permet d'indiquer à l'assembleur où se situent le segment de données et le segment de code. Puis il s'agit d'initialiser le segment de données:

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

La même procédure est faite quand viendra le temps d'utiliser le segment de pile.

7. Les instructions

Une instruction, en langage assembleur, est divisée en champs. La structure typique d'une instruction assembleur est définie schématiquement comme suit :

Étiquette :	Code opération (mnémonique)	Opérande(s)
--------------------	-----------------------------	-------------

Les différents champs d'une instruction sont généralement séparés par un ou plusieurs espaces. Le nombre d'opérandes du 3ème champ varie d'une machine à l'autre et va de 0 à 3. Après ces champs, il est souhaitable d'ajouter des commentaires, pour comprendre ce que fait un programme.

Exemple d'une partie d'un programme assembleur :

```
Donnees1    DW      1

Adresse1 :  MOV      AX, Donnees1
            ADD      AX, BX
            JMP      Adresse2
```

- La première instruction, appelée directive en assembleur, permet de définir la variable Donnees1 et de lui réserver un espace de la taille d'un mot mémoire (Define Word).
- La deuxième instruction, portant l'étiquette Adresse1, est une instruction assembleur qui réalise un transfert de données (MOV). En particulier, elle transfère le contenu de la variable Donnees1 dans le registre AX.
- La troisième instruction effectue une addition (ADD) entre 2 registres (AX et BX) et stocke le résultat dans le premier registre.
- La dernière instruction effectue un saut inconditionnel (Jump) à l'adresse Adresse2 qui doit se trouver quelque part dans le programme.

1. Opérande et étiquette

Contrairement au langage machine, le langage d'assemblage permet de donner des noms alphanumériques aux variables et aux étiquettes (adresses des instructions), ce qui facilite la lecture d'un programme assembleur par les programmeurs.

Par exemple, supposons que l'on désire effectuer un branchement (un saut). En langage machine, on doit donner, sous forme binaire, la position mémoire exacte où se trouve l'instruction vers laquelle on veut se brancher. Dans le langage d'assemblage, il suffit de faire précéder l'instruction (où l'on veut se brancher) d'une étiquette symbolique et de donner cette étiquette comme opérande de l'opération de branchement.

De la même façon, pour les opérandes, on n'est plus obligé de donner leurs adresses binaires en mémoire centrale. Il suffit de donner à chaque opérande un nom (suite de caractères), et ce nom sera utilisé à chaque fois que l'on voudra référencer cet opérande. Il en est de même pour les registres, puisque chacun d'eux a un nom qui lui est propre et qui lui permet d'être référencé dans un programme assembleur.

2. Principales instructions du microprocesseur 8086

Instruction en assembleur	Descriptif de l'instruction
MOV d,s	copie de la source dans la destination
ADD AX, adr	Ajoute à AX la valeur stockée à l'adresse indiquée et stocke le résultat dans AX
SUB AX, adr	Soustraire de AX la valeur stockée à l'adresse indiquée et stocke le résultat dans AX
MUL s	multiplie AL par s et place le résultat (16bits) dans AX
DIV s	divise AX par s et place le résultat dans AL et le reste dans AH

CMP AX, Val	Compare AX et la valeur indiquée
CMP AX, adr	Compare AX et la valeur stockée à l'adresse indiquée
DEC AX	Décrémente AX (soustrait 1)
INC AX	Incrémente AX (ajoute 1)
JMP adr	Saut incondtionnel
JA adr	Saut à l'adresse indiquée si CF=0
JB adr	Saut à l'adresse indiquée si CF=1
JE adr	Saut à l'adresse indiquée si égalité
JG adr	Saut à l'adresse indiquée si supérieur
JLE adr	Saut à l'adresse indiquée si inférieur
JNE adr	Saut à l'adresse indiquée si non égalité
PUSH s	Empiler s
POP d	Dépiler dans s la tête de la pile
REP	Une instruction précédée par REP est répétée tant que CX n'est pas nul. CX est automatiquement décrémenté.
REPZ/REPE	répète tant qu'il y a égalité (Z=1) et CX non nul.
REPZ/REPNE	répète tant qu'il y a différence et CX non nul.
LOOP adresse	décrémentation automatique de CX, saut à adresse si CX non nul, ligne suivante si CX=0.
LOOPZ/LOOPE adresse	itérer tant que CX<>0 et égalité (Z=1).
LOOPNZ/LOOPNE adresse	bouclage tant que CX<>0 et Z=0.
CALL adresse	appel de sous-programme.

3. Exemple d'opérandes et d'étiquettes

Tab	DW	1	définition d'une variable Tab de 1 mot (DW = Define Word).
Dix	DC	10	définition d'une constante Dix qui a comme valeur 10 (DC = Define Constant)
Boucle :	MOV	AX, Dix	Transfert de la valeur 10, de la constante Dix, dans le registre AX.
	MOV	Tab, BX	Transfert de la valeur du registre BX dans la variable Tab.
	JMP	Boucle	Saut incondtionnel à l'adresse définie par l'étiquette boucle.

4. Littéraux

En langage machine, toute constante doit être codée en binaire. Le langage d'assemblage permet de définir des valeurs entières ou réelles dans différentes bases (2, 8, 10 ou 16) ainsi que des chaînes de caractères. C'est toujours l'assembleur qui s'occupe de leur conversion. L'indication de la base s'effectue en plaçant un caractère particulier au début de chaque donnée. Ainsi, une donnée binaire est suivie par un **b**, une donnée hexadécimale par un **h**. S'il n'y a pas de caractère particulier, elle est considérée comme étant une donnée décimale. Les chaînes de caractères sont entourées du signe '.

'A' 65 41h 01000001b '01'

12337 3031h 0011000000110001b

5. Directives

Les directives ou pseudo-instructions, sont des instructions non exécutables qui n'ont pas de code machine équivalent. Elles représentent des directives (d'où leur nom) données à l'assembleur, qui lui fournissent des indications pour traduire le programme. Comme pour les instructions exécutables, on utilise les directives en les référençant par leur code mnémonique. Il existe différentes sortes de directives (voir manuel assembleur). Nous allons, dans ce qui suit, étudier quelques unes (les éléments qui sont entre croches [] sont facultatifs) :

- **La directive DB (Define Byte)**

[nom] DB constante [, constante]

Réserve et initialise un octet. **nom** est un symbole permettant de référencer cet octet. Par exemple, les lignes suivantes :

- NB_OCTET DB 36
- LETTRE DB 'e'
- CHAINE DB 'hello world !'
- TABLEAU DB 10 dup (0)

définissent 4 variables comme suit :

- NB_OCTET référence une donnée codée sur un octet dont la valeur initiale est 36.
- LETTRE référence une donnée dont la valeur sera initialisée avec le code ASCII du caractère 'e' ;
- CHAINE référence une chaîne de caractères composée des caractères 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', ' ', '!',
- TABLEAU référence un tableau de 10 octets (DB) dont la valeur initiale des éléments est 0

- **La directive DW (Define Word)**

[nom] DW constante [, constante]

Réserve et initialise un mot (16 bits). Voici quelques exemples :

- MOT DW 1559
- MOT2 DW ((45 * 23) / 9 + 23)
- TMOT DW 20 DUP (0)
- le symbole MOT référence une donnée codée sur un mot et aura pour valeur initiale la valeur 1559 ;

- le symbole MOT2 référence une donnée codée sur un mot et qui est initialisée avec la valeur 138, qui est le résultat de l'expression arithmétique donnée en opérande ;
 - le symbole TMOT définit un tableau de 20 mots initialisés avec la valeur 0

Exemples de directives :

TITLE 'Titre du programme'			
Vecteur	DS	50	Définition de la variable vecteur et réservation de 50 mots.
Zero	DC	0	Définition de la constante zéro qui a la valeur 0.
	PLEN	50	50 lignes par page (PLEN = Page LENgth)
	END		Directive indiquant la fin du programme

8. Expressions arithmétiques

Contrairement à un langage évolué, les expressions arithmétiques utilisées pour calculer la valeur d'une variable, comme par exemple dans l'expression arithmétique suivante $A = B + C / D$, ne sont pas admises dans les langages assembleur. Elles doivent être programmées en utilisant plusieurs instructions.

9. Macro et sous-programme

Certains assembleurs permettent de structurer des programmes. Pour cela, ils offrent généralement la possibilité de grouper une séquence d'instructions sous la forme d'un sous-programme ou d'une macro-instruction. Ces deux structures ont pour but de modulariser le programme et d'éviter l'écriture répétée d'un groupe d'instructions qui est fréquemment utilisé.

1. Macro-instruction

L'idée d'une macro-instruction, ou plus simplement d'une macro, consiste à isoler la séquence d'instructions que l'on veut éviter de répéter et à lui attribuer un nom symbolique, grâce auquel on pourra lui faire référence. Chaque fois que, dans le programme, on fait référence à ce nom, l'assembleur le remplace par la séquence d'instructions correspondantes.

Un assembleur qui autorise l'utilisation de macro est appelé un macro-assembleur. Presque tous les assembleurs actuels offrent cette possibilité.

L'utilisation de macros présente plusieurs avantages. Elle permet d'abord d'étendre le jeu d'instructions de la machine, car chaque macro peut être utilisée comme toute autre instruction. Ensuite, les programmes sources sont plus courts, plus structurés et ainsi plus faciles à comprendre et à modifier. Leur qualité s'en trouve augmentée, car lorsqu'une macro fonctionne correctement, on est assuré que ce sera toujours le cas. Leur utilisation permet

aussi d'épargner du temps de programmation. Les instructions qui servent à définir et à délimiter une macro (par exemple, MACRO et ENDM) sont des cas typiques de directives. Lors de la phase d'assemblage, chaque appel d'une macro est remplacé par le corps de la macro correspondante.

Exemple d'une macro pour calculer le cube d'un nombre

```
MACRO CUBE (valeur, valeur-cube)
    MOVE    AX, valeur
    MOVE    BX, valeur,
    MUL     AX, BX           ; (AX := AX * BX)
    MUL     AX, BX
    MOVE    valeur-cube, AX
ENDM
```

2. Sous-programme

Les sous-programmes sont définis comme les macros : ils ont aussi pour but d'éviter d'avoir à répéter l'écriture de séquences d'instructions que l'on veut utiliser plusieurs fois. Une différence essentielle avec les macros réside dans le fait que les instructions qui composent un sous-programme constituent une entité bien séparée du programme principal. Cette séparation existe toujours après la traduction. Ainsi, un sous-programme n'existe qu'en un seul exemplaire en mémoire centrale, et c'est seulement à l'exécution que toute référence à ce sous-programme provoque un branchement à ce sous-programme.

Cette manière de procéder offre les mêmes avantages que les macros mais, en plus, elle permet de minimiser la taille du code exécutable, ce qui n'est pas le cas des macros. Par contre, elle pose de nouveaux problèmes : il faut connaître les adresses des sous-programmes et il faut aussi penser à sauvegarder l'adresse de retour lors de l'exécution d'un sous-programme. L'adresse de retour est l'adresse de l'instruction qui suit l'instruction d'appel du sous-programme. Il faut souligner aussi que, dans certains langages comme Fortran, les sous-programmes peuvent être compilés et traduits séparément pour être ensuite être utilisés par différents programmes.

Pour résumer la principale différence entre macros et sous-programme, on peut dire que les appels à une macro sont remplacés par le corps de la macro pendant la traduction alors que les appels à un sous-programme sont traités lors de l'exécution.

```
MACRO
ENDM
Programme
Fin_programme
```

```
Sous-programme
Sous-programme
```

Sauvegarder état de la machine

Fin-sous-programme

Restaurer état de la machine

Retour au programme principal

Programme

Programme

Fin_programme

Fin_programme

a. (b)

Figure 2 : Différences entre Macro et sous-programme

(a) Macro, (b) Sous-programme

3. Passage de paramètres

Une autre différence majeure entre macros et sous-programmes réside dans le passage des paramètres. Un programme peut échanger des données avec des macros ou avec des sous-programmes à l'aide de paramètres. Un paramètre est une variable dont le nom est connu, mais dont le contenu n'est précisé qu'au moment de l'exécution. Une macro ou un sous-programme, s'écrit en utilisant des paramètres dits **formels** (paramètres virtuels sans valeur précise). Ces paramètres formels sont remplacés par des paramètres appelés **effectifs** (ou réels), lors de l'appel. Cette substitution, entre paramètres formels et paramètres réels, s'effectue lors de la traduction pour les macros et lors de l'exécution pour un sous-programme. Les paramètres peuvent être passés des différentes manières.

Dans le cadre des macros, le passage des paramètres est relativement simple, car l'expansion de la macro (remplacement de son appel par les instructions qui la composent) est effectuée pendant la traduction et l'on connaît forcément l'adresse des paramètres effectifs. La définition de la macro contient des paramètres formels (par exemple, **valeurs** et **valeurcube** dans la macro cube définie plus haut) et tout appel à la macro contient des paramètres effectifs. C'est l'assembleur qui se charge de la substitution pendant la traduction.

Pour les sous-programmes, il existe plusieurs techniques de passage des paramètres. Les deux principales sont :

- Passage de paramètres par valeurs
- Passage de paramètres par adresse.
- Le passage par valeurs consiste à recopier la valeur à transmettre dans une zone connue d'un sous-programme, qui peut être une zone en mémoire ou un registre. Avec ce type de passage, les sous-programmes ne travaillent que sur une copie des paramètres et ainsi, toute modification d'un paramètre n'est possible qu'à l'intérieur d'un sous-programme (elle est invisible à l'extérieur d'un sous-programme). Dès que l'on retournera au programme appelant, le paramètre retrouvera sa valeur initiale, puisqu'elle n'a pas été modifiée. Ceci permet une certaine protection des paramètres car le sous-programme ne travaille que sur une copie de ceux-ci ;
- Le passage par référence consiste à transmettre au sous-programme les adresses des paramètres. Les sous-programmes travaillent donc effectivement sur les données du

programme appelant. Dans ce cas, toute modification de la valeur des paramètres à l'intérieur d'un sous-programme sera visible (perceptible) de l'extérieur.

2. Les interruptions

Les interruptions sont des signaux technologiques (physiques) qui sont utilisés pour faire fonctionner un ordinateur. Elles constituent un élément fondamental pour l'exécution de programmes par un processeur.

Chaque interruption porte un numéro de référence (numéro en hexadécimal, c'est à dire suivi de la lettre h), comme par exemple 20h, 21h. Ces interruptions peuvent être utilisées dans un programme assembleur grâce à l'instruction INT (INTerrupt). Ainsi, lorsque le microprocesseur rencontre un code d'interruption, noté INT avec son numéro, il accède au début de la mémoire centrale dans une partie appelée « Table des vecteurs d'interruptions » : il trouve dans cette table l'adresse du programme qu'il doit exécuter en réponse à cette interruption. Il exécute ce programme puis retourne au programme interrompu (celui qui a exécuté une instruction INT).

En ce qui concerne les opérations d'E/S (tel que l'affichage sur écran ou la saisie au clavier), on va faire appel au système d'exploitation MS/DOS pour réaliser ces opérations. Pour cela il faut déclencher une interruption du programme en cours pour donner la main au système DOS. Celui-ci va exécuter la fonction DOS demandée (selon le code de l'interruption), puis il y aura un retour vers le programme en cours d'exécution.

Exemples :

Affichage d'un caractère

```
Mov DL," A"           ; caractère à afficher
Mov AH, 2              ; fonction n° 2 qui spécifie la fonction à
                        ; exécuter (dans notre cas l'affichage)
Int 21h                ; appel système (fonction DOS)
```

Saisie d'un caractère (avec écho, i.e avec affichage sur écran du caractère saisi)

```
Mov AH, 1              ; fonction n° 1
Int 21h                ; (résultat dans AL)
```

Saisie d'un caractère (sans écho)

```
Mov AH, 7              ; fonction n° 7
Int 21h                ; appel de l'interruption 21H
                        ; résultat dans le registre AL
```

Arrêt d'un programme assembleur (à mettre en fin de programme)

```
Mov AH, 4Ch            ; code de terminaison
Int 21h
```

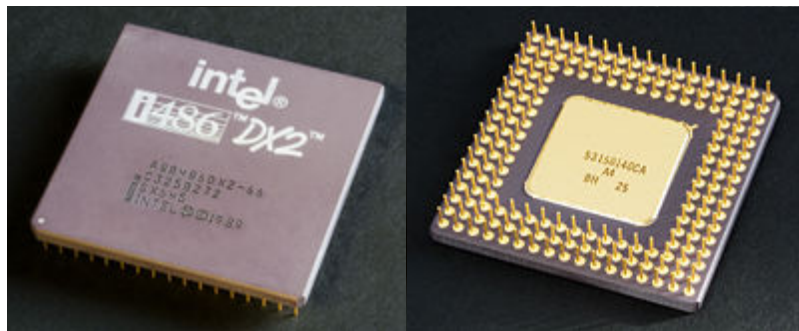
Affichage d'une chaîne de caractères

```
MOV AH, 9
LEA DX, chaine          ; équivalent à Mov DX, offset chaine
Int 21h
```

Saisie d'une chaîne de caractères

```
MOV AH, 10          ; fonction n° 10
Int 21h             ; appel de l'interruption 21h
                    ; l'adresse de la chaîne saisie sera mise dans
                    ; le registre dans DX
```

3. Quelques informations utiles sur les processeurs x86



Boîtier d'un microprocesseur Intel i486DX2-66 MHz

Les processeurs de la famille x86 comptent approximativement deux décennies de domination sur l'univers de l'ordinateur personnel. Le Pentium 4™, par exemple, est un x86. En réalité, un x86 est un processeur compatible avec son plus lointain prédécesseur : le 8086 créé par Intel™ en 1978. Il va de soi que la compatibilité n'est assurée que dans un sens. Autant un programme écrit en assembleur pour fonctionner sur un 8086 a toutes les chances de fonctionner sur un de ses descendants du 21ème siècle, autant l'inverse est très peu probable, à moins que son programmeur soit un grand nostalgique et ait par hasard retrouvé sous un meuble une disquette portant un assembleur de l'époque comme MASM.

La famille des x86 est divisée en plusieurs générations. Le 8086 est un processeur 16-bits muni d'un bus d'adressage 20-bits. Le 8088, construit à peu près en même temps que le 8086, est un processeur moins cher, qui n'a qu'un bus de données 8-bits, bien que son fonctionnement interne soit totalement identique à celui du 8086.

Le 8086, bien que très compétitif à sa création, commence à montrer ses lacunes vers le milieu des années 80. Ses deux principales lacunes sont : sa limitation à 1 Mo de mémoire système (les 20 bits d'adressage), et l'absence totale des mécanismes de protection de la mémoire, ce qui limite le développement de systèmes d'exploitation multi-tâches — ceux-ci se sont en réalité développés bien plus tard en ce qui concerne les PC : Linux apparaît en

1991.

Le 80286 a donc remplacé le 8086, et son apparition marque le début d'une ère plus complexe, puisqu'à partir d'ici, les x86 ont plusieurs modes d'exécution. Tous les successeurs du 80286 comportent un *mode réel* ou *mode d'adressage réel*. Lorsqu'un x86 fonctionne en mode réel, il se comporte comme un 8086, c'est-à-dire que l'accès à la mémoire système se fait de manière très rudimentaire : un registre de segment contient l'adresse physique du segment mémoire qui nous intéresse, et un registre pointeur contient une adresse à l'intérieur du segment. De plus, son jeu d'instructions est celui du 8086. Le 80286 introduit le *mode protégé*. Un registre de segment ne contient alors plus une adresse physique marquant le début du segment, mais un sélecteur de segment. Un sélecteur de segment est constitué d'un numéro de descripteur et d'un certain nombre d'indicateurs. Le processeur va alors rechercher le descripteur de segment correspondant dans une table, et celui-ci indique, entre autres, l'adresse physique (24 bits cette fois) de début du segment. Le 80286 contient aussi les rudiments du système d'adressage actuel par plages de mémoire virtuelle.

Le 80386 est le premier processeur x86 32-bits. Il offre de plus un système de mémoire virtuelle complet.

Le 80486 introduit de nouvelles instructions de calcul sur virgules flottantes.

L'époque qui suit est celle des Pentium™, des Celeron™, des Athlon™, etc. qui se différencient de leurs prédécesseurs par différentes optimisations, en particulier, le traitement en simultané de deux instructions consécutives, ainsi que différentes nouvelles instructions dont les extensions MMX, SSE et 3Dnow.

Finalement l'Opteron et l'Athlon 64 d'AMD, ainsi que le Xeon 64 bits d'Intel utilisent un jeu d'instructions de 64 bits.