Ministère de l'enseignement supérieur et de la recherche scientifique
Université Mohamed Chérif Messaâdia
- Souk Ahras -

Faculté des sciences et de technologie

Département: Mathématiques et Informatique

وزارة التعليم العالي والبحث العلمي
جامعة محمد الشريف مساعدية
– سوق أهراس –

كلية العلوم والتكنولوجيا
قسم الرياضيات والإعلام الآلي

Architecture des Ordinateurs

Notes de cours

2e Année Licence Informatique

CRED : 2
COEF : 5

Bekhouch Amara

2016-2017

Table des Matières

Chapitre 01

Introduction à l'architecture des ordinateurs.

Chapitre 02

Introduction au langage machine.

Chapitre 03

L'assembleur 80x86.

Chapitre 04

Interprétation et Compilation : Notions de base.

Chapitre 05

Interruptions et Appels systèmes.

Chapitre 06

Les architectures modernes.

Chapitre 01 : Introduction à l'architecture des ordinateurs

1.1. Introduction

Un ordinateur est une machine électronique programmable. Le principe de base de fonctionnement de cette machine est l'exécution des programmes organisés en mémoire sous forme d'un ensemble d'instructions stockées séquentiellement. L'ordinateur exécute ces instructions l'une après l'autre, dès sa mise sous tension. Donc, l'ordinateur est capable de lire de l'information, de la stocker, de la transformer en effectuant des traitements quelconques, puis de la restituer sous une autre forme.

Nous présentons dans ce chapitre, l'architecture simplifiée d'un ordinateur basée sur le modèle créé par John Von Neumann ainsi que les principes de son fonctionnement.

1.2. Architecture générale

Le modèle John Von Neumann créé en 1946 est considéré comme la première description d'un ordinateur dont le programme est stocké dans sa mémoire. L'architecture de la plupart des ordinateurs actuels se base sur ce modèle.

1.2.1. Principe de base

Les deux principaux constituants d'un ordinateur sont :

- *La mémoire principale :* (MP en abrégé) permet de **stocker** de l'information (programmes et données)
- *Le processeur :* **exécute** pas à pas les instructions composant les programmes.

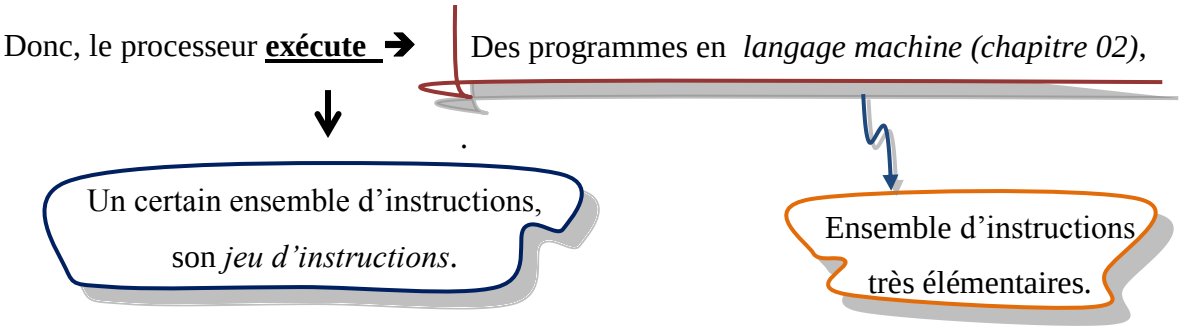
Notion de programme

Un programme peut être défini par un ensemble d'instructions élémentaires, exécutées dans l'ordre par le processeur.

Les instructions représentent des opérations élémentaires à savoir :

- Lecture du contenu d'un emplacement mémoire,
- La modification du contenu d'un emplacement mémoire,
- Une opération arithmétique, à titre d'exemple l'addition de deux nombres.
- … etc.

Chaque instruction est représentée en mémoire principale par un ou plusieurs octets qui désignent son code en langage machine! (nous apprenons la notion du langage machine dans le deuxième chapitre).



1.2.2. L'unité centrale

L'unité centrale dans cette architecture est interprétée par le processeur ; qu'est ce qu'un processeur ?

Le processeur peut être défini comme est un circuit électronique complexe chargé d'exécuter l'ensemble instructions et également de communiquer avec les unités d'échange.

Le cycle général d'exécution d'une instruction par le processeur est résumé par :

1. lecture en mémoire (MP) de l'instruction à exécuter ;
2. exécution ou réalisation du traitement correspondant (addition par exemple);
3. Passage à l'instruction suivante.

Dans l'architecture de Von Neumann sur laquelle se base la majorité des architectures actuelles, Le processeur est composé de deux parties (figure 1.1), l'unité de commande et l'unité de traitement :

– L'unité de commande : C'est l'unité chargée de la lecture en mémoire et également du décodage des instructions ;
– L'unité de traitement : l'exécution des différentes instructions est effectuée par cette unité appelée aussi *Unité Arithmétique et Logique* (U.A.L.),

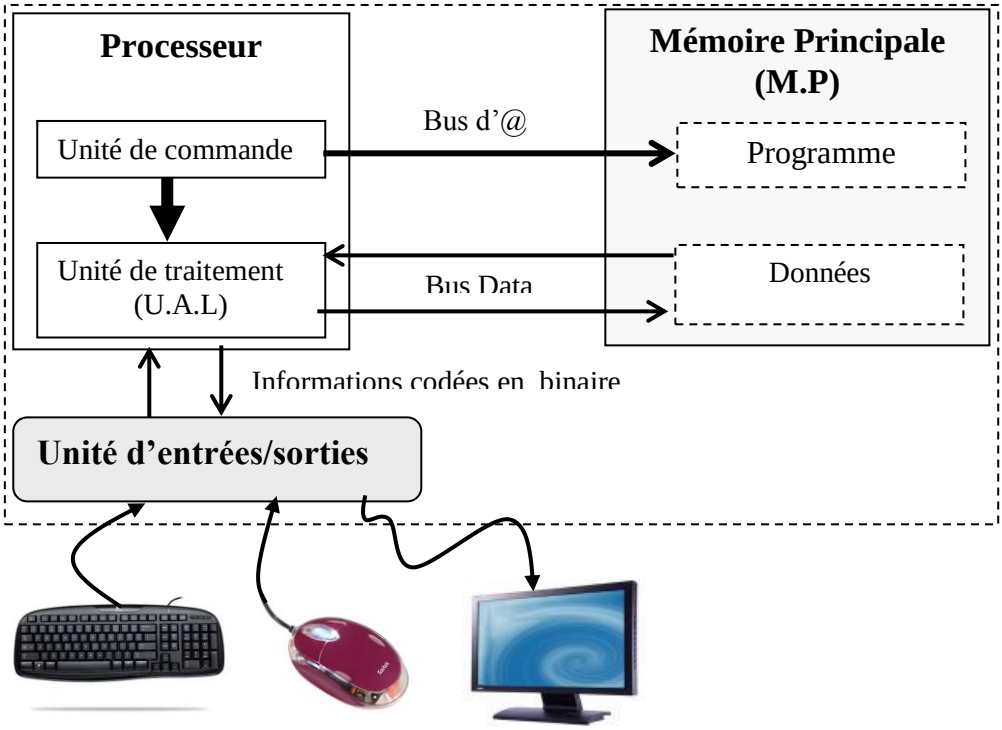


Figure 1.1 : *Architecture générale d'un ordinateur (à base du modèle de Von Neumann).*

1.3. La mémoire principale

1.3.1. Organisation

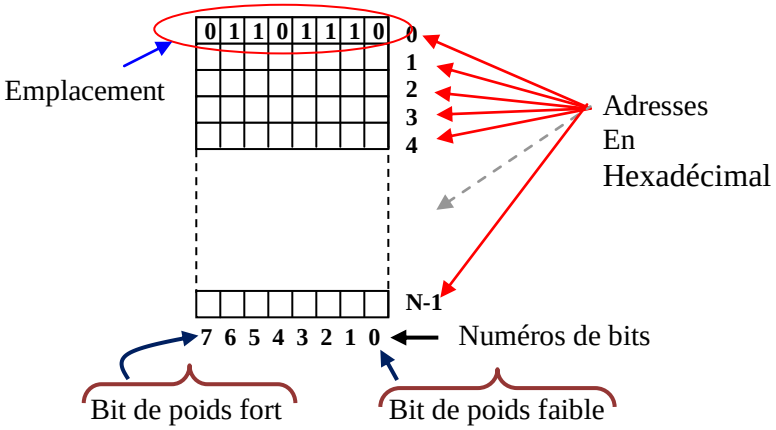


Figure 1.2 : *Structure générale de la mémoire principale.*

La mémoire est divisée en ***emplacements*** de taille fixe ; ces emplacements sont utilisés pour le stockage temporaire (lors de l'exécution) des instructions et des données des programmes.

Remarque : La plupart des ordinateurs d'aujourd'hui utilisent des emplacements mémoire d'un octet.

La mémoire principale est caractérisée par:

- Le nombre d'emplacements mémoires,
- L'adresse de l'emplacement : le plus souvent notée en hexadécimal (Chaque emplacement est repéré par une *adresse).*
- La taille de chaque emplacement.

Comment calculer sa *capacité (Taille mémoire)?*

La capacité est représentée par le nombre d'emplacements, exprimé en général en kilo-octets ou en méga-octets,

1.3.2. La lecture et l'écriture dans la mémoire

Les données et les instructions contenues dans les emplacements mémoire sont conservées jusqu'à coupure de l'alimentation électrique, où tout le contenu est perdu.

Pourquoi

Par ce qu'elle est volatile : Les mémoires externes (les disquettes et disques durs,…etc) ne sont pas volatiles c'est-à-dire que leur contenu ne s'efface pas après une coupure électrique.

Les opérations possibles sur la mémoire sont :

- ***Écriture d'un emplacement :*** La modification du contenu d'un emplacement mémoire dont l'adresse de cet emplacement est donnée par le processeur.
- ***Lecture d'un emplacement :*** La récupération par le processeur du contenu d'un emplacement mémoire sans modification. L'adresse de l'emplacement à lire est précisée par Le processeur.

Remarque :

- Le contenu des emplacements mémoire ne peut être modifié que par l'intervention du processeur.

- Les deux opérations de lecture et d'écriture au niveau de la mémoire principale sont réalisées sur un ou plusieurs octets en fonction de la notion du mot mémoire.

1.4. Le processeur central

Le processeur est parfois appelé CPU (*Central Processing Unit*) ou encore MPU (*Micro-Processing Unit*) pour les microprocesseurs.

Qu'est-ce que

Un ***microprocesseur***

Un microprocesseur est un processeur où tous les composants sont assemblés sur la même puce électronique (pastille de silicium), afin de :

- Réduire les coûts de fabrication.
- Augmenter la vitesse de traitement.

Remarque : Les microordinateurs sont tous équipés de microprocesseurs.

1.4.1. Les registres et l'accumulateur

Le processeur utilise toujours des ***registres :***

- Des petites mémoires internes très rapides d'accès,
- Utilisées pour stocker temporairement : Des données, des instructions ou des adresses mémoire.
- Le nombre exact de registres dépend du type de processeur et varie typiquement entre une dizaine et une centaine.
- Chaque registre stocke 8, 16, 32 bits ou 64 bits.

Les résultats des opérations arithmétiques et logiques effectuées par L'UAL sont stockés dans un registre appelé *accumulateur*. L'accumulateur est un registre qui est utilisé dans une proportion importante des instructions.

1.4.2. Architecture simplifiée

Un exemple de l'architecture interne d'un processeur central est présenté dans la figure 1.3. Comme le montre cette figure, ce processeur est doté d'une unité de commande, une unité arithmétique et logique (UAL), un décodeur et un ensemble de registres (IP, RI, accumulateur).

Mais qu'est - ce qu'un

***Décodeur* d'instructions**

Le décodeur d'instruction permet de lancer la partie de l'unité de commande nécessaire en fonction du code de l'instruction lu en mémoire.

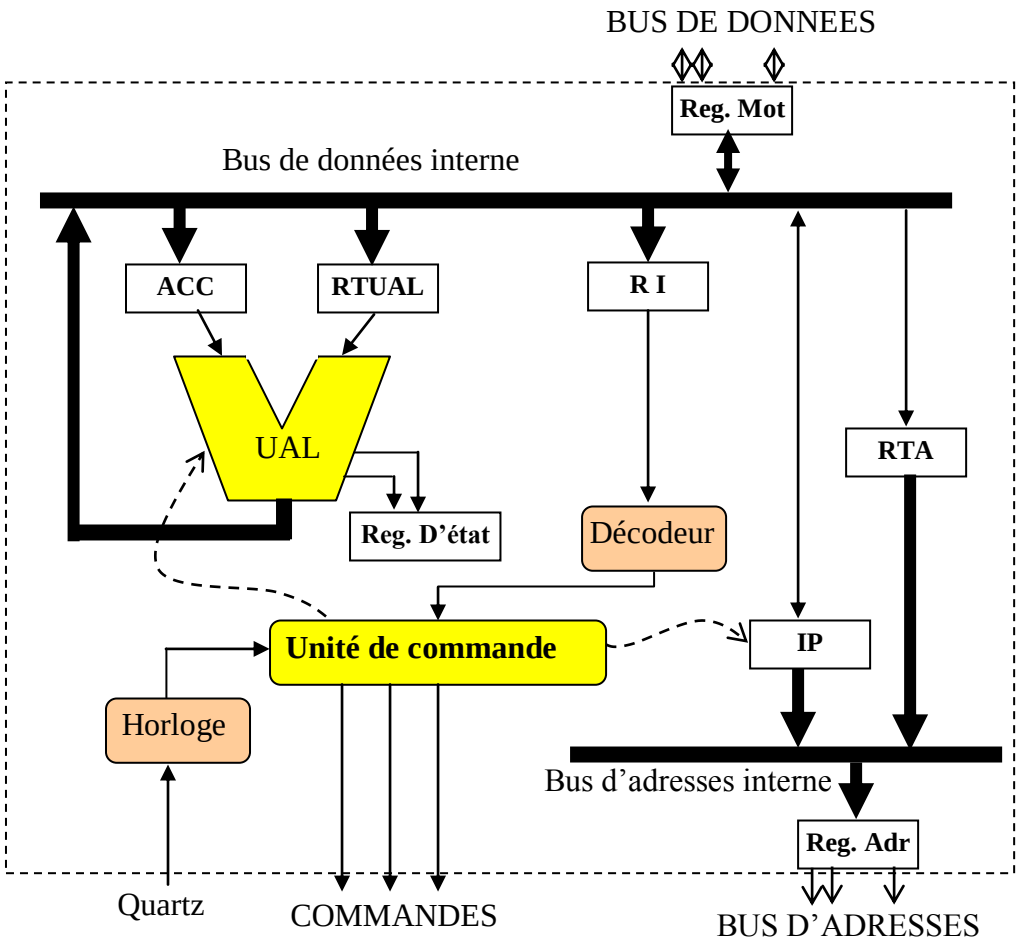


Figure 1.3 : *Schéma simplifié de l'architecture interne d'un processeur central.*

Quelques registres du MPU :

- ACC : Accumulateur ;
- IP : *Instruction Pointer* ou Compteur de Programme, contient toujours l'adresse de la prochaine instruction à exécuter en MP;
- RI : Registre Instruction, contient le code de l'instruction en cours d'exécution;
- RTUAL : Registre Tampon de l'UAL, stocke temporairement l'un des deux opérandes d'une instruction arithmétique.

- Reg. d'état : stocke les *indicateurs*, que nous étudierons plus tard ;
- RTA : Registre Tampon d'Adresse, utilisé pour accéder à une donnée en mémoire.

Le processeur communique avec les autres circuits de l'ordinateur à l'aide des signaux de commandes.

Par exemple, le type de l'opération à effectuer au niveau de la mémoire principale (lecture ou écriture) est réalisé à l'aide du signal R/W (Read/Write).

Exemple récapitulatif:

Les étapes d'exécution de l'instruction "*Ajouter 20 au contenu de l'emplacement mémoire d'adresse 100h*" :

1. Lecture et décodage de l'instruction (instruction d'addition) ;
2. la demande du contenu de l'emplacement 100h par processeur via le bus d'adresse.
3. La récupération du contenu de l'emplacement 100h par processeur et rangement de la valeur lue dans l'accumulateur ;
4. L'UAL ajoute 20 au contenu de l'accumulateur ;
5. Le contenu de l'accumulateur est écris en mémoire à l'adresse 100h.

Dans tout cela, quel est le rôle de *L'unité de commande et de L'UAL* (Figure 1.3).

- L'unité de commande déclenche chacune de ces actions dans l'ordre.
- L'UAL effectue l'opération d'addition.

1.5. Les bus

La liaison entre le processeur et la mémoire principale est assurée par deux *bus internes*, l'un pour les données, l'autre pour les instructions. C'est à l'aide de ces deux bus que les informations circulent entre le MPU et la MP.

Qu'est-ce qu'un ***bus***

Un *bus* est simplement un ensemble de *n* fils conducteurs, utilisés pour **transporter** *n* signaux binaires.

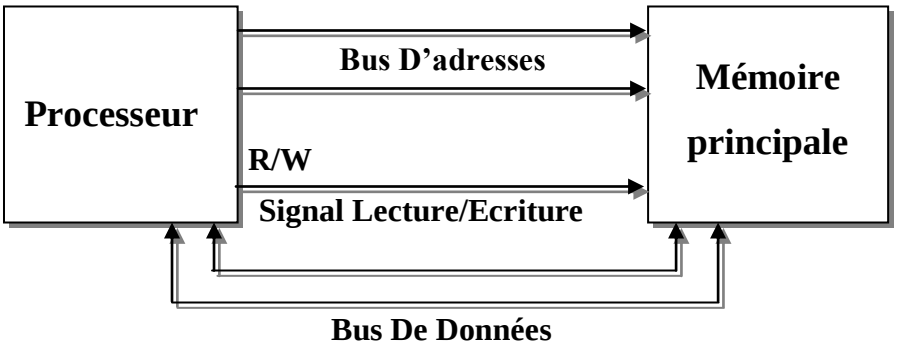


Figure 1.4 : *liaison entre le MPU et la MP : bus de données, bus d'adresse et signal lecture/écriture.*

Le bus d'adresse :

- Est un bus unidirectionnel : seul le processeur envoie des adresses.
- Il est composé de *a* fils ; on utilise donc des adresses de *a* bits.
- La mémoire peut posséder au maximum 2^a emplacements (adresses 0 à 2^a - 1).

Le bus de données : Est un bus bidirectionnel.

- Lors d'une lecture, c'est la mémoire qui envoie un mot sur le bus (le contenu de l'emplacement demandé) ;
- Lors d'une écriture, c'est le processeur qui envoie la donnée.

1.6. Exercices

Exercice 01.

Comparer entre l'architecture de Von Neumann et celle de HARVARD en présentant les avantages et les inconvénients de chaque modèle.

Exercice 02.

Dans une architecture Von Neumann, on considère les caractéristiques suivantes :

- 16777216 emplacements mémoire.
- 32 Mo comme taille mémoire.

- Quelle est la taille de chaque emplacement ?
- Quelle est la plus grande adresse (en hexadécimal) mémoire peut être utilisée ?
- Quelle est la taille du bus de données et du bus d'adresses?

Chapitre 02 : Introduction au langage machine

2.1. Introduction

Nous allons étudier dans ce chapitre la programmation en langage machine et en assembleur d'un microprocesseur (seuls les registres et les instructions les plus simples seront étudiés). Le processeur à étudier est le 80x86 de la famille x86 de l'Intel. Nous dresserons également quelques exemples de manipulation à l'aide de *debug*.

2.2. Les processeur de la famille x86 de l'Intel

Les micro-ordinateurs de type PC et compatibles utilisent les processeurs de la famille x86 de l'Intel (ou un autre de la même architecture de base !).

Les premiers modèles de PC, commercialisés au début des années 1980, utilisaient le 8086 :

Le processeur 8086

Un microprocesseur 16 bits (Avec un bus d'adresses de 20 bits pour gérer jusqu'à 1 Mo de mémoire).

Les modèles suivants ont utilisé successivement le **80286, 80386, 80486 et Pentium (ou 80586)**.

L'évolution de ces processeurs est basée principalement sur:

- Augmentation de la fréquence d'horloge,
- Augmentation de la largeur de bus (32 bits d'adresse et de données),
- Introduction de nouvelles instructions (par exemple calcul sur les réels)
- Ajout de registres.

Remarque : Ces processeurs sont compatibles entre eux ; un programme écrit dans le langage machine du 80286 peut s'exécuter sans modification sur un 80486.

Les caractéristiques du processeur à étudier sont:

Registres:

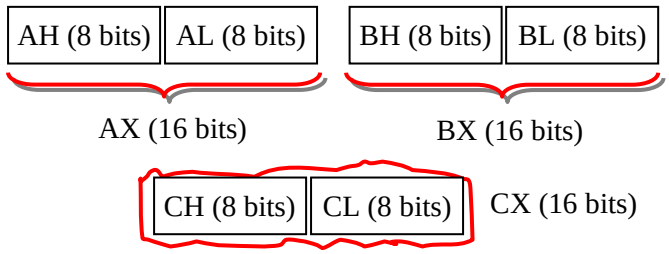
- Accumulateur AX (16 bits) ;
- Registres auxiliaires BX et CX (16 bits) ;
- Pointeur d'instruction IP (16 bits) ;
- Registres segments CS, DS, SS (16 bits) ;
- Pointeur de pile SP (16 bits), et pointeur BP (16 bits).

CPU 16 bits à accumulateur :

- Bus de données 16 bits ;
- Bus d'adresse 32 bits ;

Les registres de données de 16 bits peuvent être utilisés comme deux registres indépendants de 8 bits:

Par exemple AX devient la paire (AH, AL)



2.3. Architecture des instructions

Le microprocesseur fonctionne à l'aide d'un ensemble d'instructions appelé jeu des instructions (Instruction Set Architecture, ISA). Ces instructions sont codées en binaire.

2.3.1. Types d'instructions

Instructions arithmétiques et logiques: Les opérations effectuées généralement entre le contenu de l'accumulateur AX et une donnée. Cette donnée peut être le contenu d'un emplacement mémoire ou une constante.

- Addition : AX ← AX + donnée ;
- Soustraction : AX ← AX - donnée ;

Instructions d'affectation: Permettent le transfert de données entre le processeur (les registres) et la mémoire principale ou l'inverse.

MOV AX, donnée

Instructions de comparaison : Permettent la comparaison du contenu d'un registre et une donnée. Ces instructions positionnent également le contenu de quelques indicateurs.

CMP AX, donnée

Instructions de branchement : Utilisation : Par exemple, implémentation des boucles et tests. Réalisation : Par l'exécution de l'instruction autre que celle qui suit celle en cours en mémoire.

2.3.2. Le codage instructions

Une *instruction* est composée de deux champs :

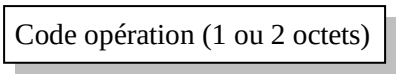
- Code opération : Code du type d'opération réalisée par l'instruction, addition, affectation,…etc.
- Code opérande : Indiquent les opérandes de l'instruction, une donnée, une adresse (la référence à une donnée en mémoire).

Donc, une instruction peut être codée par 1, 2, 3 ou 4 octets en fonction de son *mode d'adressage*. Alors, qu'est ce qu'un mode d'adressage ?

2.3.3. Les modes d'adressage

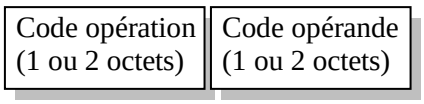
Les modes d'adressage peuvent être regroupés en cinq groupes: implicite, immédiat, direct, relatif et indirect (nous étudierons le mode indirect dans le chapitre 03).

Adressage implicite : Chaque instruction est codée sur 1 ou 2 octets où seulement le code opération est utilisé.



Dans ce mode, l'instruction ne peut porter que sur un registre particulier. Par exemple décrémenter le contenu du registre AX.

Adressage immédiat : Chaque instruction est composée de deux champs ; chacun codé sur 1 ou 2 octets.



Le champ opérande représente une donnée (Constante). Exemple : Stocker la valeur 20 à AX. L'opérande (20) est codé dans ce cas sur 2 octets. Par ce que le registre utilisé est de 2 octets.

Adressage direct : Le champ opérande contient l'*adresse* de la donnée en mémoire principale sur 2 octets.

Exemple: additionner le contenu du registre AX et le contenu de l'emplacement mémoire D'adresse 100 h

Adressage relatif : Ici, la taille du champ opérande est toujours égale 1 ! Par ce que le contenu est un *déplacement. Donc, ce mode* est utilisé pour les instructions de branchement.

Code opération (1 octet)	Déplacement (1 octet)

2.3.4. Langage machine vs langage symbolique

Voici un programme en langage machine 80x86, implanté à l'adresse 0200 H :

```
B8 10 00  03  06  20  02  89  C3
```

Cette représentation des instructions en Hexadécimal est compréhensible par le microprocesseur. Ainsi, le microprocesseur doit décoder cette écriture afin d'exécuter les opérations correspondantes.

Langage machine

C'est le langage *natif* d'un processeur, c'est donc un langage binaire composé de 1 et de 0, compréhensible par un ordinateur. On le manipule en utilisant l'assembleur.

Ce programme additionne le contenu du registre AX et de l'emplacement mémoire d'adresse 220h et range le résultat dans le registre BX.

L'obtention de la représentation Hexadécimal ci-dessus est effectuée à l'aide de la traduction des instructions en *langage assembleur (voir le chapitre suivant)* en *langage machine.*

Ainsi, la première instruction du programme ci-dessus (notée B8 10 00 en Héxadécimal) est notée en assembleur :

```
MOV  AX,  10
```

Permet de charger la valeur 10 (une donnée) dans le registre AX du processeur.

Langage symbolique du programme complet.

Adresse	Contenu MP	Langage	Symbolique	Observation
0200	B8 10 00	MOV AX,	10	Charger AX avec la valeur 10.
0203	03 06 20 02	ADD AX,	[0220]	Ajouter le contenu de 0220 a AX.
0207	89 C3	MOV BX,	AX	Ranger AX en BX.

Adresse de début de chaque instruction en hexadécimal.

Remarque : En langage symbolique, on indique toujours la destination, puis la source : MOV destination, source ; MOV est une instruction d'affectation.

MOV AX, [0220] → Le contenu de l'adresse mémoire 0220H vers AX

MOV [0220], BX → Le contenu du registre BX vers l'emplacement mémoire 0220H

La représentation en langage machine des instructions en langage symbolique peut être obtenue en utilisant l'utilitaire *Debug* (section 2.5). Quelques instructions en langage machine obtenues en utilisant *debug* sont données dans Annexe.

2.4. Exécution des instructions

Le CPU exécute les instructions en mémoire principale les unes après les autres ; autrement dit, les instructions sont exécutée séquentiellement à l'aide du registre IP qui contient l'adresse de la prochaine instruction à exécuter. Donc, lors de l'exécution « normale » d'un programme :

1. Lire et décoder l'instruction à l'adresse IP ;
2. IP ← IP + *taille de l'instruction (incrémentation)*;
3. Exécuter l'instruction.

2.4.1. Cas des instructions de branchement

Les instructions de branchement permettent de passer à une instruction autre que celle qui suit celle en cours en mémoire. Ce type de condition peut notamment se rencontrer dans les instructions de boucle (chapitre 03):

Pourquoi utiliser les Instructions de *branchement*

Permettent l'implémentation des boucles ainsi que le gain de l'espace mémoire (Pour ne pas stocker plusieurs fois une instruction).

Principe de l'exécution avec *branchement* : Modification du contenu du registre IP pour qu'il pointe à l'adresse d'une autre instruction que celle qui suit en cours d'exécution en MP.

Dans ce cas, la modification du contenu du registre IP s'effectue en se basant sur le mode d'adressage relatif:

IP = IP + *déplacement*

Déplacement (entier codé sur 8 bits*)* = adresse instruction visée - adresse instruction suivante

Exemple : Le programme suivant décrémente le contenu du registre AX jusqu'à la valeur 0. Le test est effectué par l'instruction de branchement JNE (*Jump if Not Equal*).

Adresse	**Contenu MP**	**Langage Symbolique**	**Observation**
0100	B8 05 00	MOV AX, 05	Charger AX avec la valeur
0103	48	DEC AX	Décrémenter AX (par 1)
0104	83 F8 00	CMP AX , 0	Comparer entre AX et 0
0107	75 FA	JNE 0103	Saut vs 0103 si AX<> 0
0109		Instruction ?	Exécution si condition non vérifiée

Remarque : le codage de l'instruction DEC AX en langage machine donne seulement 48 ; parce que le mode d'aressage de cette instruction est Implicite où seulement le code opération est utilisé (pas de code opérande). Cette instruction est codée sur 1 octet.

Le codage de l'instruction de branchement JNE 0103 est donné par 75FA ; 75 est le code opération et FA est le déplacement.

Déplacement = adresse instruction visée - adresse instruction suivante

= 0103 – 0109 = -6 (un nombre signé)

Donc, la représentation en Çà2 de -6 est : FA = déplacement

Dans le programme ci-dessus l'instruction CMP permet la comparaison de deux valeurs, à savoir, le contenu d'un registre et une autre valeur.

2.4.2. Exemples d'instructions de branchement

On distingue deux catégories de branchements, *inconditionnels* et *conditionnels*.

Branchement inconditionnel : Dans ce type d'instruction le saut est toujours effectué sans condition. Les principales instructions de branchement inconditionnel sont JMP et LOOP (LOOP est utilisée dans les procédures, voire le chapitre 03). Tout comme le cas des instructions conditionnels, l'opérande de JMP est un *déplacement* (adressage relatif).

Branchement conditionnels : s'exécutent en fonction d'une condition. Le saut vers l'instruction visée s'effectue seulement si la condition est vérifiée, sinon le registre IP doit

contenir l'adresse de l'instruction suivante. Voici quelques exemples d'instructions de branchement:

1) **JE** *Jump if Equal*
2) **JNE** *Jump if Not Equal*
3) **JG** *Jump if Greater*
4) **JLE** *Jump if Lower or Equal*
5) **JA** *Jump if Above*
6) **JBE** *Jump if Below or Equal*
7 **JB** *Jump if Below*

Généralement, les instructions de saut conditionnel figurent juste après une instruction de comparaison. Le principe de fonctionnement de l'instruction CMP par exemple, est exactement le même que l'opération de soustraction SUB, mais sans stockage du résultat. Son seul effet est donc de positionner **les indicateurs !?** Après la modification (positionnement) du contenu des indicateurs par une instruction de comparaison, les conditions des instructions de branchements conditionnels s'expriment en fonction des valeurs **des indicateurs !?** Autrement dit, le saut s'effectue ou non selon la valeur stock dans un **indicateur** spécifique modifié par une instruction CMP à titre d'exemple.

Indicateur !? Indicateur !? Indicateur !?, 

2.4.3. Les indicateurs

La taille de chaque indicateur est 1 bit, l'ensemble des indicateurs forme le *registre d'état (voir le chapitre 01)* du processeur. La modification du contenu des indicateurs est réalisée après certaines opérations.

La modification registre d'état n'est pas accessible globalement par des instructions;

Chaque indicateur est manipulé individuellement par des instructions spécifiques.

Exemple :

La valeur de l'indicateur CF (*l'indicateur de retenue*) est modifiée par les deux instructions STC et CLC.

STC	CF ← 1 *(SeT Carry)*
CLC	CF ← 0 *(CLear Carry)*

Quelques exemples d'indicateurs :

ZF *(Zero Flag ou indicateur de zéro)*

Positionné après une soustraction ou une comparaison, SI le résultat = 0 alors ZF=1.
SINON, ZF = 0.

Les 2 Opérandes sont égaux

CF *(Carry Flag ou l'indicateur de retenue) :* Comme son nom l'indique, l'indicateur contient la retenue du bit de poids fort obtenue après une addition ou une soustraction.

Exemples:

0 1 0 0	1 1 0 0
+ 0 1 1 0	+ 0 1 1 0
--------	--------
CF=0 1 0 1 0	**CF=1** 0 0 1 0

SF (*Sign Flag ou indicateur de signe) :* Représente le signe du résultat d'une opération d'addition ou de soustraction. Ainsi,

Si le bit de signe == 1
Alors SF est positionné à 1
Sinon
SF est positionné à 0.

Exemples:

0 1 0 0	1 1 0 0
+ 0 1 1 0	+ 0 1 1 0
--------	---------
SF=1 1 0 1 0	**SF=0** 0 0 1 0

OF (*Overflow Flag ou indicateur de débordement) :* OF=1 si il 'y a un débordement après une opération, par exemple d'addition ou de soustraction ; sinon OF = 0

Exemples :

0 1 0 0	1 1 0 0
+ 0 1 1 0	+ 0 1 1 0
---------	---------
OF=1 1 0 1 0	**OF=0** 0 0 1 0

2.5. Exemples de manipulation à l'aide de *debug*

Afin de maitriser les notions arborées ci-dessus, nous essayons dans cette section de présenter quelques exemples en termes de manipulation des adresses mémoire, contenu des emplacements mémoire, écriture d'un programme en langage symbolique et affichage du contenu de la mémoire après exécution,…etc. Cependant, ces manipulations nécessitent l'utilisation d'un utilitaire tel que *debug*.

Qu'est - ce que l'utilitaire *debug*

Le *debug* est un petit programme fourni par Microsoft avec tous les systèmes d'exploitations (depuis le DOS 2.0 jusqu'aux Windows XP). Le *debug* permet d'explorer la mémoire ou même pour écrire quelques lignes d'assembleur,…etc

Voici quelque commande du programme *debug* :

- D : la commande D est utilisée pour l'affichage du contenu d'une zone mémoire en hexadécimal ou en ASCII ;
- E : permet la modification du contenu d'un ou plusieurs emplacements mémoire;
- A : permet d'entrer un programme en assembleur; l'affichage après exécution est effectue en utilisant la commande D.
- U : à partir du contenu d'une zone mémoire (langage machine), cette commande offre la possibilité de l'affichage en langage symbolique de ce programme programme ;
- R : pour l'affichage de la valeur des registres du processeur ;
- T : pour l'affichage de la trace d'exécution d'un programme entrée en langage symbolique

On peut appeler l'utilitaire *debug* à partir de MS-DOS en utilisant la ligne de commande :

C:\>DEBUG ↵

-

Le tiret placé au début de la ligne indique que le *debug* est prêt à recevoir une commande. Le premier exemple à étudier est celui présenté dans la section 2.3.4.

Donc, pour entrer ce programme en langage symbolique à l'adresse 0200 h, en utilise la commande :

```
D:\DEBUG>debug
-A cs:0200
072A:0200 mov AX,10
072A:0203 ADD AX,[0220]
072A:0207 MOV BX,AX
072A:0209
-
```

Le CS est le registre du segment de code, l'adresse contenue dans ce registre est l'adresse segment est égal 072A dans l'exemple ci-dessus. L'affichage du contenu de la mémoire peut être effectué par la commande D:

```
-D cs:0200
072A:0200  B8 10 00 03 06 20 02 89-C3 00 00 00 00 00 00 00 ..... ........
072A:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-
```

L'adresse de début de ce programme est 0200 et le dernier octet de la dernière instruction se trouve à l'adresse 0208. La dernière instruction est présentée par 89 C3 comme le montre l'exemple précédent.

Le deuxième exemple est celui de la section 2.4.1. La saisie et l'affichage sont appliquées en utilisant successivement les deux commandes A et D comme suit :

```
-a cs:0100
072A:0100 mov AX,05
072A:0103 DEC AX
072A:0104 CMP AX,0
072A:0107 JNE 0103
072A:0109
-
```

L'affichage du contenu de la mémoire et également la correspondance de ce programme en langage machine sont données par :

Déplacement

```
-d CS:0100
072A:0100  B8 05 00 48 83 F8 00 75-FA 00 00 00 00 00 00 00 ...H...u........
072A:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
072A:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

Dans le programme précédent entré à l'adresse 0100h, la modification par exemple de l'adresse visée 0103h dans l'instruction de branchement JNE par l'adresse 0100h nécessite la modification du déplacement FA qui se trouve à l'adresse 0108h comme suit :

Déplacement = adresse visée – adresse suivante

Déplacement = 0100 – 0109 = -9

Déplacement = F7

La commande qui permet la modification d'un emplacement mémoire est « E ». la modification est donné par :

```
-E CS:0108 F7
-D CS:0100
072A:0100  B8 05 00 48 83 F8 00 75-F7 00 00 00 00 00 00 00  ...H...u........
072A:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
072A:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-
```

Pour confirmer si la modification fonctionne correctement ou non, nous pouvons utiliser la commande U qui permet d'afficher un programme en langage assembleur comme suit :

```
-U CS:0100
072A:0100 B80500          MOV     AX,0005
072A:0103 48              DEC     AX
072A:0104 83F800          CMP     AX,+00
072A:0107 75F7            JNZ     0100
```

75 : code opération
F7 nouveau déplacement

Nouvelle adresse visée

Remarque : les instructions JE et JNE sont parfois écrites JZ et JNZ (même code opération).

2.6. Exercice

Exercice 01.

-Compléter le programme ci-dessous :

Adresse	**Contenu MP**	**Assembleur**
0200	A10005	
		MOV [0006], AX
		MOV BX, 0025
0209	03060006	
	A30007	
		CMP AX, [0007]
	74F1	JE
	CD21	

- Donner les modes d'adressage pour chaque instruction.
- Quelle est la taille de ce programme ?
- Donner le contenu du segment de code CS :0200,

Exercice 02.

En exécutant la commande **d 100** sous Debug, on obtient le résultat suivant :

```
-d 100
1FC0:0100 10  FF  00  05  01 00  BB FF-FF 83 C3 01  29  C0 21 D8     .....).!.
1FC0:0110 31  D8 23  06  00  02  00  00-00 00  00  00  D8 00 9F 13    1.#.....
1FC0:0120 00  00  00  00  00  00  00  00-00 00  00  00  00  00 00 00  ........
1FC0:0130 00  00  00  00  00  00  00  00-00 00  00  00  00  00 00 00  ........
1FC0:0140 00  00  00  00  00  00  00  00-00 00  00  00  00  00 00 00  ........
1FC0:0150 00  00  00  00  00  00  00  00-00 00  00  00  00  00 00 00  ........
1FC0:0160 00  00  00  00  00  00  00  00-00 00  00  00  00  00 00 00  ........
1FC0:0170 00  00  00  00  00  00 00  00-00 00  00  00  00  00 00 00  ........
```

a. Quel est le rôle de la commande **d** ?

b. D'après le résultat affiché, quelle est la valeur binaire contenue dans l'emplacement mémoire d'adresse **011C** du segment **1FC0** (c'est-à-dire **1FC0:011C)** ?

c. replacer la valeur de l'emplacement d'adresse **1FC0:0134** par D7.

Exercice 03.

Comparer les deux processeurs, MIPS R3000 et Intel 8086 (modes d'adressage, jeu d'instructions, …).

Chapitre 03 : L’assembleur 80x86

3.1. Assembleur : Notions de base

3.1.1. Utilité de l'assembleur ?

Qu'est ce que *L'Assembleur*

C'est le langage le plus proche du matériel. Peut être considéré comme la version lisible du code machine. Chaque instruction assembleur est représentée par un code machine afin d'être compréhensible par le microprocesseur.

Pourquoi utiliser *L'assembleur*

Le langage assembleur à été mis au point principalement pour permettre une notation plus simple que la notation hexadécimale ou binaire du langage machine.

```
MOV   AX,   10
ADD   AX,   [0220]
MOV   BX,  AX
```

Langage assembleur

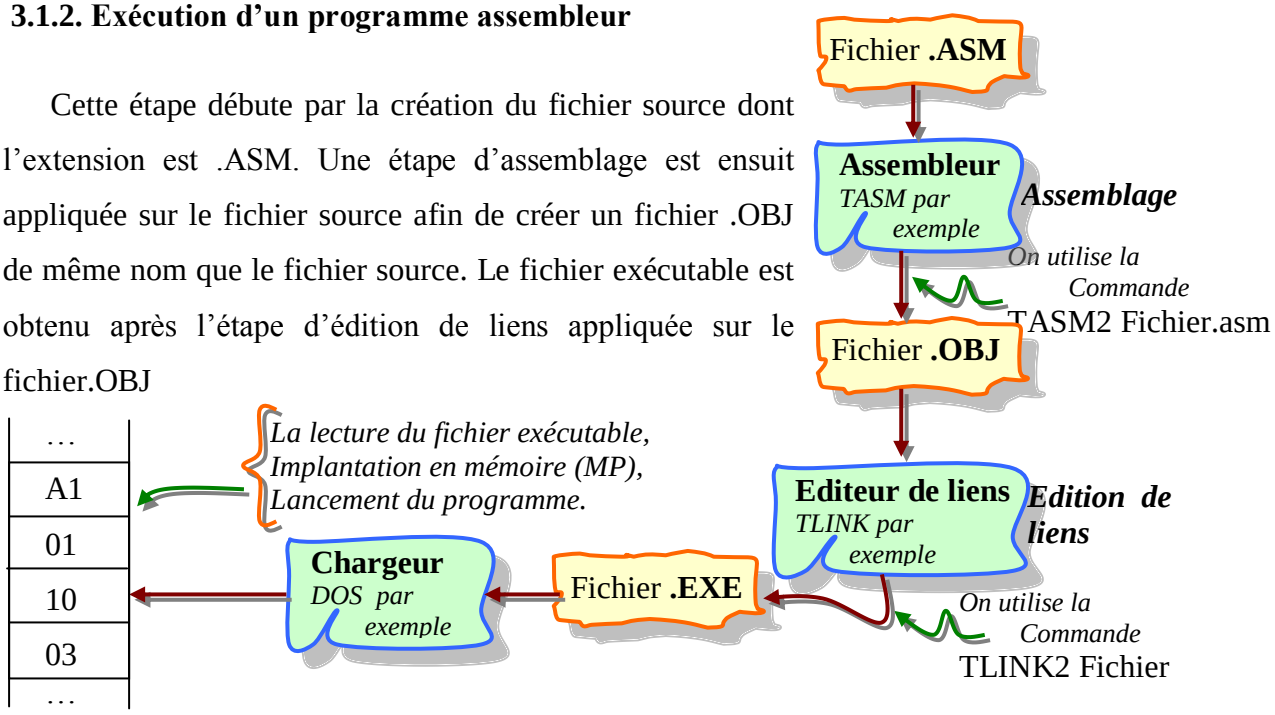
B8 10 00 03 06 20 02 89 C3

Exemple d'une notation hexadécimale

Remarque : Une documentation des différentes instructions en termes de langage machine du processeur est toujours fournie par les concepteurs de processeur, comme Intel.

3.1.2. Exécution d'un programme assembleur

Cette étape débute par la création du fichier source dont l'extension est .ASM. Une étape d'assemblage est ensuit appliquée sur le fichier source afin de créer un fichier .OBJ de même nom que le fichier source. Le fichier exécutable est obtenu après l'étape d'édition de liens appliquée sur le fichier.OBJ

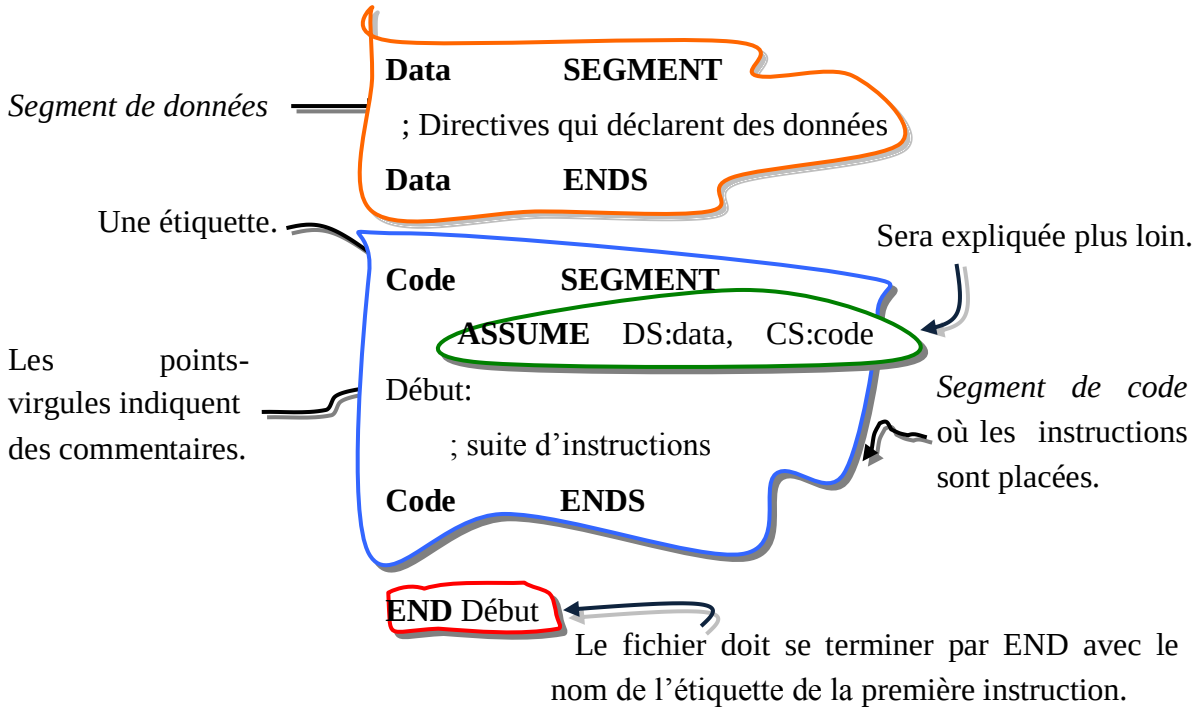


Un utilitaire spécial du système d'exploitation (DOS ici). Est utilisé comme *chargeur*.

Remarque : Il est bon de noter que dans le programme exécutable, les instructions (assembleur) ont été codées (langage machine).

3.1.3. Déclaration des segments

Un programme écrit en assembleur comprend des définitions de données et des instructions. Tout comme les autres langages de programmation, un ensemble de directives doit intervenir pour la déclaration des différents segments.



Remarque : directives, mots clef spéciaux que comprend l'assembleur.

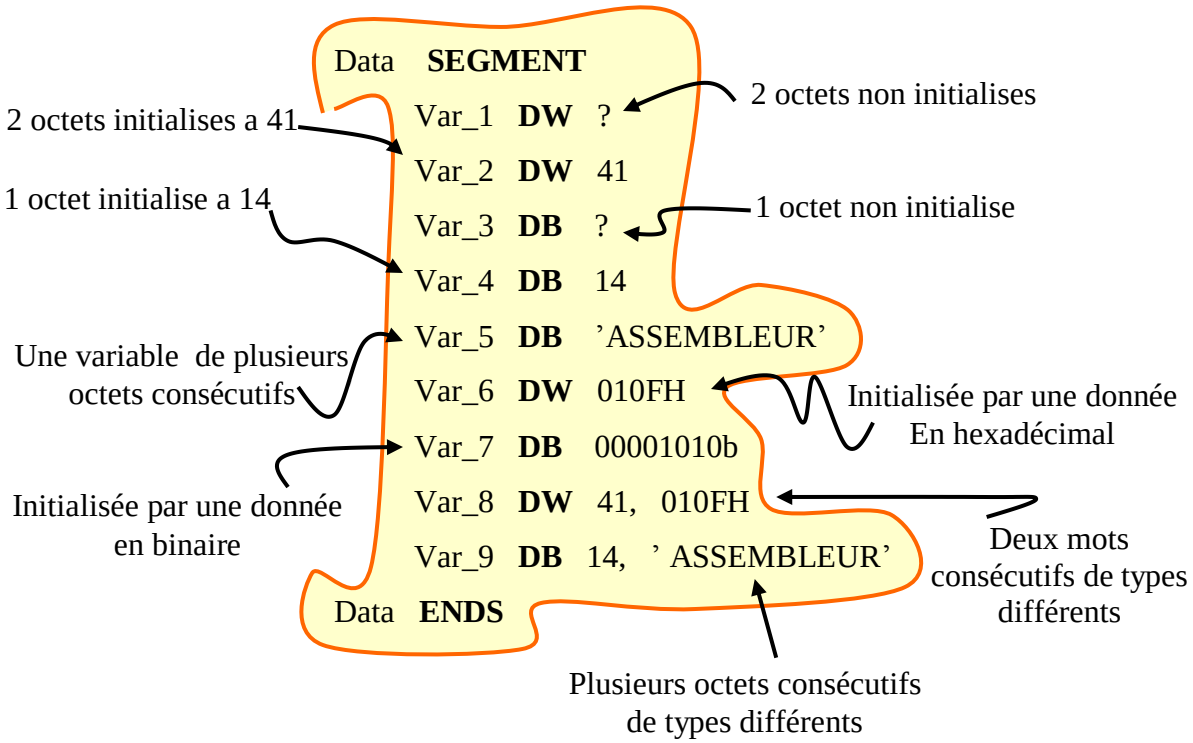
3.1.4. Les variables en assembleur

Les noms des variables en assembleur peuvent contenir au maximum 31 caractères. Ces caractères peuvent être des lettres, des chiffres ou l'un des caractère @, ? et _. Lors de la déclaration des variables l'assembleur attribue à chaque variable une adresse.

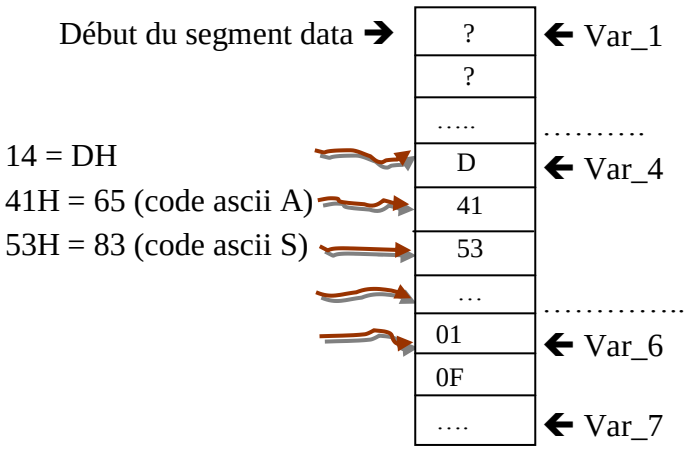
Taille des variables

La taille des variables est désignée par le nombre d'octets alloué en mémoire qui peut être 1, 2, 4 ,…etc. pour des raison de simplification, nous nous intéresserons dans ce support qu'à des variables de taille 1 ou 2 octets. Les différents types et tailles de variables sont détaillés

dans [01]. Les directives de déclaration des variables sont ; DB (*Define Byte*) pour la déclaration des variables de 1 octets et DW (*Define Word*) pour la déclaration des variables de 2 octets. L'exemple suivant montre quelques cas de déclaration pour les deux tailles



Donc, la mémoire (le segment de données) aura le contenu suivant (quelques exemples) :




```
MOV   AX,   Var_2
ADD   AL,   Var_4
MOV   Var_3,  55
```

Après la déclaration des variables dans le segment de données, ces variables peuvent être utilisées dans le programme (segment de code) en les désignant par leur nom.

Cas d'un tableau avec des éléments de même type

```
Tab_1  DB   10 dup (0)   ;10 octets initialisé à 0
Tab_2  DW   10  dup (?)  ;10 mots non initialises
```

Directive dup : est utilisée pour déclarer un tableau de n cases, toutes initialisées à la même valeur.

3.2. La notion de Segment de mémoire

Le mécanisme de gestion des adresses mémoire par les processeurs 80x86 d'Intel est basé sur la notion de segment mémoire. L'idée de base de cette notion est tout simplement le partage de la mémoire en plusieurs zones mémoire chacune chargée de contenir un certain type d'information. A titre d'exemple, les données doivent être stockées dans une zone mémoire appelée *segment de données* tandis que les instructions sont placées dans un *segment d'instructions*. La figure 3.1. Illustre un exemple d'organisation des différents segments mémoire ainsi que les registres utilisés pour chaque segment.

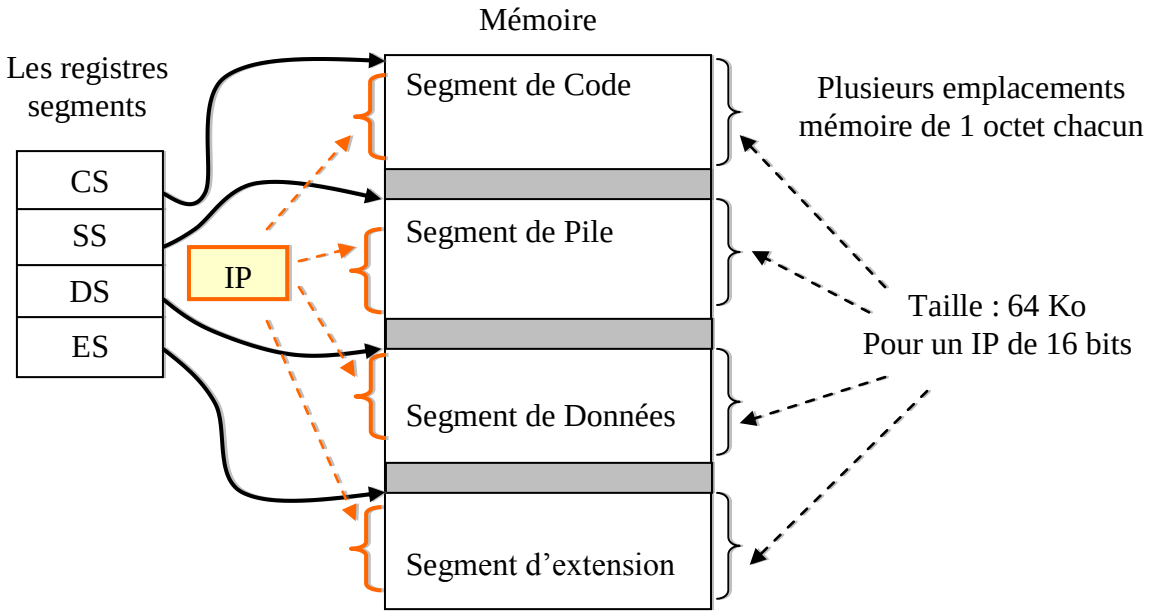


Figure 3.1 : *Illustration simple de l'organisation en plusieurs segments et les registres correspondants.*

Ainsi, les adresses de base des segments sont stockées dans des registres du CPU appelés *registres de segment* (CS, DS, SS, et ES). Les adresses mémoire sont codées en deux parties « segment » et « offset » comme le montre la figure 3.2. L'adresse segment est inscrite dans le registre qui correspond et l'offset dans le registre IP. Autrement dit, le registre IP contient les

adresses des instructions (instruction à exécuter), pour un IP par exemple de 16 bits, l'espace adressable dans chaque segment et de 2^{16} soit 64 Ko.

Le bus d'adresse du 80486 possède 32 bits (***registre segment*** (16 bits de poids fort) + ***déplacement*** (*offset*, 16 bits de poids faible).

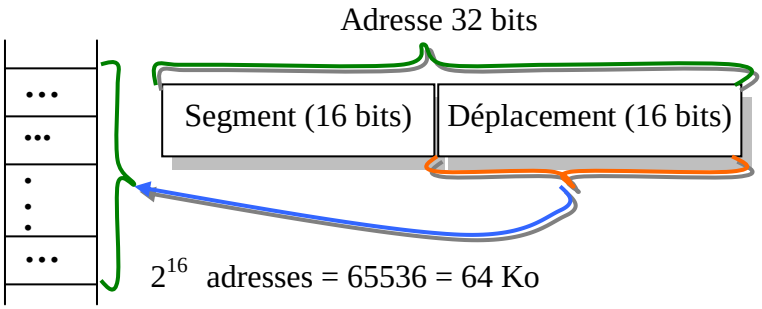


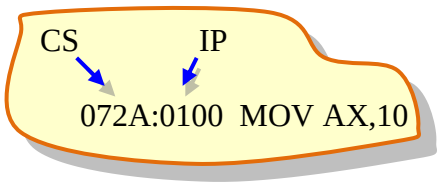
Figure 3.2. *Les deux parties d'une adresse mémoire (Adresses Logiques).*

Remarque:

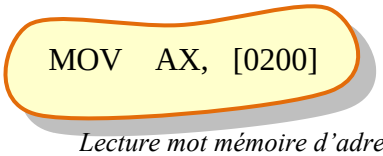
- *Les adresses physiques sont obtenues par : Adresse physique = Segment * 16 + Offset*
- *Les adresses que nous avons manipulées jusqu'ici sont des déplacements.*
- *Segment de mémoire = une zone mémoire adressable avec une valeur fixée du segment (les 16 bits de poids fort).*

Donc, deux registres spéciaux de 16 bits du CPU son utilisés pour pouvoir récupérer les données et également les instructions à partir de la mémoire.

Registre CS (*contient adresse segment d'instructions*) : dans ce cas l'adresse logique est composée par le CS et le IP (CS:IP). Exemple (voir la section 2.5, chapitre 02)



Registre DS (*adresse segment de données*) : Le registre DS est utilisé pour accéder aux données manipulées par le programme.
Ainsi, l'instruction



Lecture mot mémoire d'adresse DS:0200H.

Remarque : Afin que l'assembleur ne confonde pas les adresses de données et d'instructions, une directive « ASSUME » d'affectation soit utilisé pour indiquer à l'assembleur quel est le segment de données et celui de code. L'exemple de la section 3.1.3, montre la déclaration et l'utilisation de cette dérictive.

Pour une utilisation correcte des variables déclarées dans le segment de données, le registre DS doit être initialisé au début du programme assembleur de la façon suivante :

```
MOV   AX,   Nom_Segment_De_Donnees
MOV   DS,    AX
```

Exemple d'un programme en assembleur

L'exemple suivant réalise la permutation du contenu des deux registres AX et CX par l'utilisation d'une variable intermédiaire. Comme le montre cet exemple, le programme doit commencer par une étiquette (étiquette début) qui indique le début du segment de code.

```
Data      SEGMENT
          Var_1    DW   ?
Data      ENDS
Code      SEGMENT
          ASSUME   DS:data,   CS:code
Début:
          MOV      AX,   data
          MOV      DS,    AX          ; Initialisation du DS
          MOV      Var_1,  AX
          MOV      AX,    CX    ; Copier contenu CX dans AX
          MOV      CX, Var_1    ; Copier contenu AX dans CX
          MOV      AH,   4CH    ; Retour au DOS:
          INT      21H
Code      ENDS
END       Début              ; étiquette de la 1ere instruction.
```

3.3. Les tableaux en assembleur

Comme nous l'avons vu dans la section 3.1.4, un tableau en assembleur peut être déclaré de plusieurs façons avec ou sans la directive DUP. De nombreuses facilités de déclaration et de traitement des tableaux sont données en assembleur, à savoir, la déclaration d'un tableau de plusieurs cases de types différents.

3.3.1. Le mode d'adressage indirect

A la différence du mode d'adressage direct, dans le mode indirect, l'adresse de la donnée à récupérer doit être contenue dans un registre intermédiaire (le registre BX). L'utilisation du mode d'adressage indirect est très utile, voire indispensable, pour plusieurs manipulations telle que la manipulation des tableaux. Par exemple, le parcours d'un tableau nécessite le stockage de l'adresse de début dans BX puis à chaque passage d'une case à l'autre il suffit d'incrémenter BX.

```
MOV   AX,   [100h]   ; Adressage direct (l'adresse de la donnée entre crochets).
MOV   AX,  [BX]      ; Adressage indirect (l'adressage de la donnée dans BX).
```

Exemple :

```
Data        SEGMENT
    Var_1   DB   10, 'ABC', 10H
    Var_2   DW   10
Data        ENDS
            ...
        MOV     BX,   offset Var_1
        MOV AL,   [BX]
        INC BX
```

La première étape est la récupération de *l'adresse* d'un tableau ou d'une donnée dans BX par l'intervention de la directive *offset*.

Chargement de la première valeur du tableau dans AL.

Passage vers la case suivante du tableau en incrémentant l'adresse dans BX.

Remarque :

- *L'instruction MOV BX, Var_2 permet de charger dans BX la valeur 10 et non son adresse.*
- *Dans le cas où le type des éléments du tableau est DW, le BX doit s'incrémenter par 2 (on considère que la taille des emplacements mémoire est de 1 octet).*

3.3.2. Exemple : parcours d'un tableau

Comme nous l'avons présenté dans l'élément précédent de cette section, le parcours d'un tableau utilise pleinement le mode d'adressage indirecte. Un exemple complet de parcours d'un tableau est présenté ci-dessous. Cet exemple permet de compter le nombre de cases

nulles dans un tableau d'octets de 10 éléments. On considère dans cet exemple que le tableau est déjà rempli.

```
Data        SEGMENT
    Tab_1   DB   10 DUP (0) ;10 octets initialisé à 0
    Compt   DB   0 ; variable compteur
Data        ENDS
Code        SEGMENT
ASSUME   DS:data, CS:code
Debut:      MOV   AX,    data
            MOV   DS,    AX
            MOV   BX,    offset Tab_1   ; adresse début tableau
            MOV CX, BX
            ADD CX, 9
ETQ1:       MOV   AL,    [BX]           ; lis 1 caractère
            CMP AL, 0      ; comparer AL et 0
            JNE ETQ2
            ADD Compt,1
ETQ2:       INC    BX          ; passe au suivant
            CMP    BX, CX      ; compare Adr courante et finale
            JNE    ETQ1        ; sinon recommencer
            MOV AH, 4CH
            INT 21H ;  Retour au DOS
Code        ENDS
            END Début
```

L'adressage indirect est parfois ambigu, pour stocker par exemple une valeur dans la mémoire à l'adresse qui se trouve dans BX on écrit :

Range 10 à l'adresse spécifiée par BX → `MOV [BX], 10`

L'ambiguïté vient du fait que la taille de la valeur à stocker en mémoire, ici 10 est inconnue 1, 2 ou 4 octets consécutifs.

Afin de remédier à ce problème, une directive spécifiant la taille de la donnée à transférer doit être utilisée :

```
MOV  byte ptr  [BX],   val
          ; La taille de la donnée, 1 octet
MOV  word ptr [BX],  val
 ;  La taille de la donnée,  2 octets
```

3.4. Le segment de la pile

Un autre moyen pour stocker et gérer temporairement des données en mémoire principale est la *pile*.

Mais, qu'est-ce que Une *Pile*

La pile est représentée par une partie de mémoire et un pointeur du *sommet* de la pile.

3.4.1. Registres Pile

Tout comme les autres segments de la mémoire (code, données,…), le segment de pile est gérée par un ensemble de registres processeur tels que le SS et SP.

- Le registre SS (*Stack Segment*) : c'est un registre segment qui contient l'adresse du segment de pile courant. Dans un programme assembleur, ce registre doit être initialisé au début du programme.
- Le registre SP (*Stack Pointer*) : Contient le déplacement du sommet de la pile. La valeur de ce registre est changée après chaque empilement ou dépilement. Autrement dit, le registre SP point toujours sur le dernier emplacement occupé de la zone mémoire réservée pour la pile. La figure 3.3 montre une représentation schématique de la pile.

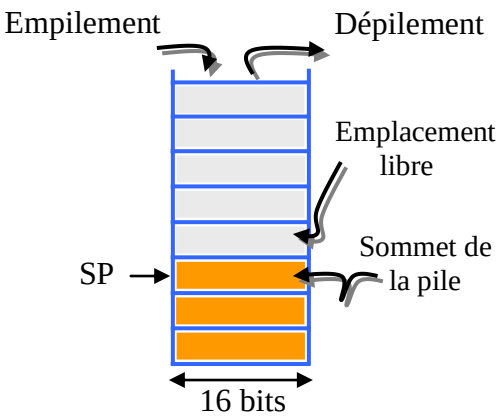


Figure 3.3 : *Une représentation schématique de la pile.*

L'opération d'empilement réalise le stockage d'un nouvel élément au sommet de la pile. Cette opérations nécessite la modification de l'adresse contenue dans le registre SP puis le transfère de la valeur du registre vers le sommet indiqué par cette adresse. La modification de l'adresse du sommet est donnée par :

– SP ← SP – 2 (où les éléments de la pile sont définis par *word*)

L'opération de dépilement consisté à transférer le sommet de la pile dans un registre tout en supprimant cet élément du sommet. Cette opération nécessite aussi la modification de l'adresse SP en utilisant :

– SP ← SP + 2

3.4.2. Déclaration d'une pile

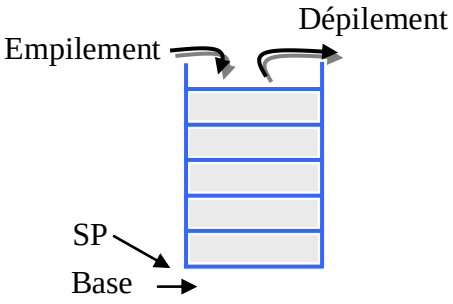
Le troisième segment de mémoire à étudier est le segment de pile. Ainsi, l'utilisation d'une pile requiert la déclaration préalable d'un segment de pile. Un exemple de déclaration d'une pile de 100 octets est illustré par :

Base, variable utilisée pour repérer l'adresse du bas de la pile

```
Pile    SEGMENT  stack
        DW  100  dup (?)
        Base  EQU  this word
Pile    ENDS
```

Mot clef stack, réservé pour la déclaration d'un segment de pile.

La base de la pile est repérée par l'étiquette *Base*. Dans le cas où la pile est vide, le registre SP doit contenir la même adresse que l'étiquette *Base*. Ensuite, une décrémentation de la valeur du SP est effectuée à chaque empilement.



Pour que la pile fonctionne correctement, une initialisation des registres SS et SP est nécessaire.

Initialisation à déclarer dans le segment de code :

```
ASSUME  SS:Pile
  MOV     AX,  Pile
  MOV     SS,  AX    ; initialisation segment pile
  MOV     SP,  Base  ; pile vide
```

3.4.3. Manipulation de la pile

La manipulation de la pile peut être établie principalement par les deux instructions PUSH et POP. Contrairement à la *FILE (*une structure de donnée où le principe de fonctionnement est FIFO ; First In First Out), le principe de fonctionnement de la pile est LIFO (*Last In First Out*, Premier Entré Dernier Sorti).

- **PUSH *registre*** empile le contenu du registre sur la pile. Par exemple PUSH AX
- **POP *registre*** déplace la valeur de sommet de la pile vers un registre avec suppression de la valeur du sommet. Par exemple POP AX.

Exemple : Permutation des valeurs de AX et BX en passant par la pile.

```
PUSH   AX    ; Pile <- AX
PUSH   BX    ; Pile <- BX
POP    AX    ; AX <- Pile
POP    BX    ; BX <- Pile
```

```
PUSH   AX          ; sauvegarde du AX
……….
MOV AX, 0
ADD AX, Var_2   ; on utilise AX pour
DEC AX             ; une autre opération
CMP AX, 0
……..
POP    AX    ; récupération de l'ancien AX
```

L'une des utilités de la pile, est la sauvegarde temporaire du contenu des registres.

Exemple : Pour sauvegarder la valeur du AX (valeur à utiliser ultérieurement) on doit stocker cette valeur dans une pile. Après l'utilisation du AX à d'autres traitements on peut récupérer l'ancienne valeur stockée dans la pile.

Remarque : L'instruction POP ne doit pas être utilisée si la pile est vide. Ceci peut permettre de stocker des valeurs imprévisibles.

3.5. Les Procédures en assembleurs

3.5.1. Définition

Dans le but de simplifier le code source et de minimiser la taille du programme, les procédures sont utilisées pour permettre l'exécution d'un ensemble d'instructions délimité par des mots clefs. L'exécution de cet ensemble d'instructions est achevée à l'aide d'un simple

appel à cette procédure. Ainsi, la notion de procédure est similaire à celle des sous programmes dans les autres langages de programmation.

En bref, la définition d'une *procédure*

Est un ensemble d'instructions pour l'exécution d'une action précise. Les procédures permettent une programmation plus souple et plus organisée en langage assembleur.

L'assembleur entreprend l'exécution des procédures à partir de l'adresse de leur première instruction.

3.5.2. Déclaration d'une procédure

La déclaration d'une procédure est implémentée dans le segment de code

```
Nom     PROC   near
        .......
        ......
        RET
Nom     ENDP
```

Le mot clef PROC : c'est une *Directive* pour la définition d'une procédure.

Un ensemble d'instructions pour la réalisation d'une action précise

Near : signifie que la procédure se trouve dans le même segment de code que le programme appelant.

3.5.3. Manipulation des procédures

Appel procédure

L'assembleur ne commence l'exécution d'une procédure qu'après l'appel à cette dernière par un programme *appelant*.

Directive pour appel procédure

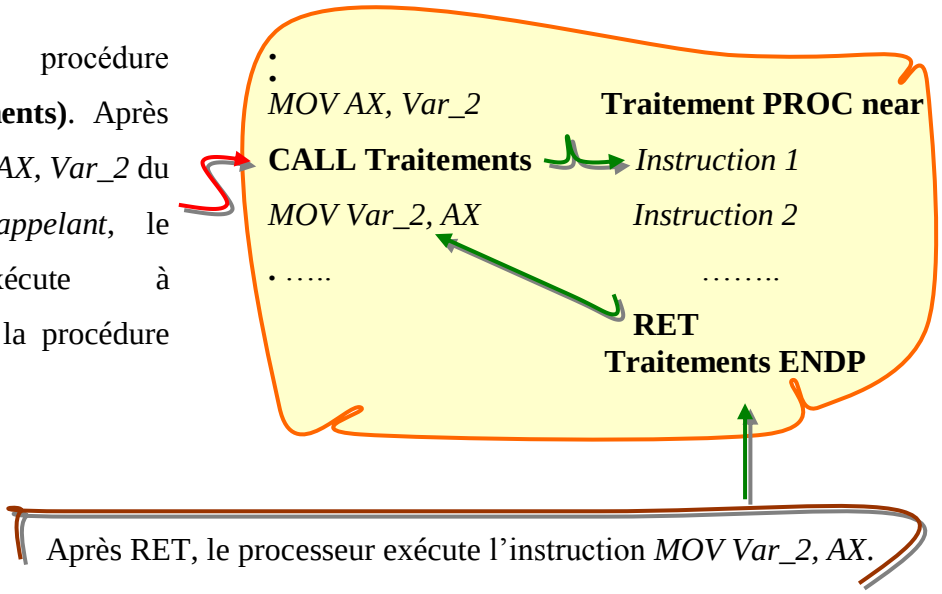
L'appel s'écrit simplement → **CALL** Nom de la procédure

Remarque :

- *CALL est une nouvelle instruction de branchement inconditionnel.*

- *En assembleur, Nom de la procédure désigne une adresse de début de la procédure sur 2 octets.*
- *Les procédures doivent être terminées (juste avant l'instruction ; Nom ENDP) par l'instruction RET.*

Appel d'une procédure (nommée **Traitements)**. Après l'instruction *MOV AX, Var_2* du du programme *appelant*, le processeur exécute à *l'instruction 1* de la procédure *Traitements*.

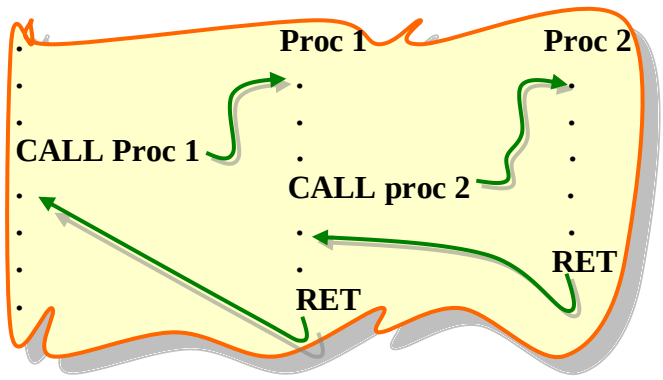


Comment le processeur retrouve-t-il les adresses des instructions (dans l'exemple ci-dessus) :

- *Instruction 1*
- *MOV Var_2, AX*

Dès l'instruction **CALL Traitements,** le processeur empile l'adresse de l'instruction MOV Var_2, AX (L'*adresse de retour*) par PUSH IP. Le retour à cette instruction est déclenché par l'instruction RET qui dépile l'adresse de retour dans le registre IP par POP IP.

Donc, l'instruction *RET* est considérée comme une instruction de branchement vers l'adresse de l'instruction empilée par *CALL nom procédure*.



Les appels imbriqués sont également possible en assembleur en utilisant le même principe d'appel et de retour.

Passage de paramètres

Représente la méthode utilisée pour l'envoi des *données* (nécessaires pour la réalisation des actions requises par la procédure) par le programme principale à la procédure. Après l'exécution des traitements requis, la procédure fournit les résultats obtenus au programme principal. Le passage de paramètres peut être effectué par plusieurs méthodes

1. *Passage de paramètres par registre* : Les paramètres à utiliser par la procédure sont envoyés via les registres processeur. Autrement dit, ces paramètres sont stockés dans les registre avant l'appel procédure.

Exemple : Une procédure "Permuter" pour la permutation du contenu de deux registres. Le contenu des registres est passé par les registres AX et BX.

```
Permuter    PROC   near
            MOV  Var_1,   AX    ;Var_1<- AX
            MOV AX,   BX
            MOV BX, Var_1
            RET
Permuter    ENDP
            MOV   AX,     10
            MOV   BX,     20
            CALL  Permute
            Instruction
```

Avantage : Sa mise en œuvre est simple,

Inconvénient : ne convient que si le nombre de paramètres est petit car le nombre de registres est limité.

2. *Passage de paramètres par pile* : Les paramètres doivent être empilés avant *appel procédure*. Ensuite, la procédure dépile ces paramètres pour traitement. Le dépilement sans sauvegarde de l'adresse IP empilé par l'appel procédure CALL provoque la perte de l'adresse de retour. Pour pallier ce problème, la récupération des paramètres empilés doit être effectuée sans dépilement. Cette technique exige donc l'intervention du registre BP (*Base Pointer*), qui permet d'accéder et de récupérer le contenu d'une case pile sans dépilement.

Ainsi, on lira le sommet de la pile avec :

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

Mode d'adressage indirect

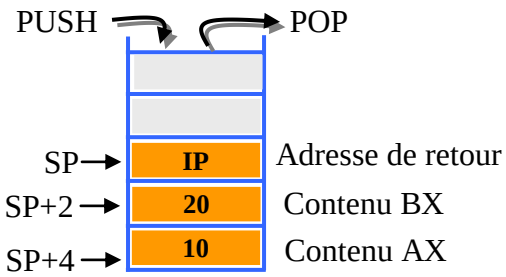
```
MOV  AX,  [BP+2]   ; 2 car 2 octets par mot de pile.
```

Lecture du mot suivant avec

Appel de la procédure "Permuter" avec passage par la pile :

```
PUSH   AX
PUSH   BX
CALL   Permuter
```

Le contenu de la pile après le CALL est :



Le sommet de la pile contient l'adresse de retour empilée par CALL après l'empilement de AX et BX.

L'exemple de la procédure Permuter peut être modifié comme suit :

```
Permuter    PROC   near          ; AX <- arg1 + arg2
            MOV    BP,  SP       ; adresse sommet pile
            MOV    AX,  [BP+2]   ;
            MOV    BX,  [BP+4]   ;
            RET
Permuter    ENDP
```

Le passage de paramètres par pile offre la possibilité d'utiliser un nombre plus grand de paramètres. Cette méthode permet également de sauvegarder les résultats de la procédure dans la pile ou également dans les registres processeur.

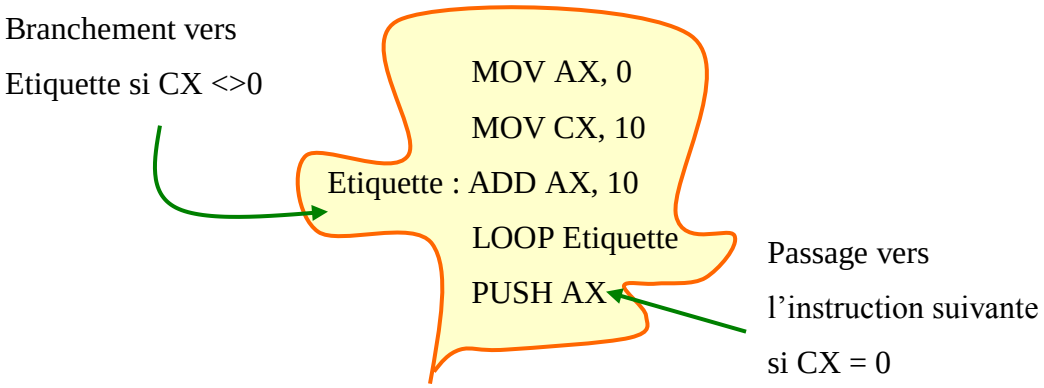
3.6. Les instructions de boucle

Dans le langage assembleur, les instructions de boucle sont nombreuses, mais la plus importante est l'instruction LOOP. Cette instruction contrôle à chaque itération le contenu du registre CX, c'est-à-dire, elle décrémente à chaque itération le contenu de CX de 1. Voir [01] pour les autres instructions de boucle.

Si CX est différente de zéro alors

IP = IP + *déplacement*

La condition d'arrêt de la boucle est CX = 0. C'est-à-dire, le passage vers l'instruction suivante est effectué lorsque CX = 0.



L'empilement d'AX est effectué après l'exécution de l'instruction ADD AX, 10 dix fois.

3.7. Exercices

Exercice 01. Ecrire un programme en assembleur 8086 qui permet de trouver le maximum dans un tableau d'octets de taille 10 emplacements. L'adresse mémoire de début du tableau est 200h. Le maximum doit être ajouté à l'emplacement mémoire d'adresse 250h.

Exercice 02.

Ecrire un programme, en langage assembleur 8086, qui permet d'additionner les nombres inférieurs ou égaux à 10 dans un tableau d'octets mémoire de longueur 20 et débutant à l'adresse 100h, le résultat doit être placé à l'adresse 200h s'il est supérieur à 100, sinon à 300h.

Exercice 03.

1. Ecrire un programme, en langage assembleur 8086, qui permet de trier par ordre croissant un tableau d'octets mémoire de longueur N = 10.

2. Donner le contenu des registres utilisés après exécution.

Exercice 04.

Ecrire un programme en assembleur 80x86 qui permet de compter le nombre de cases dont le contenu est égal à 0 en utilisant les procédures.

Exercice 05.

1. Ecrire un programme en assembleur 80x86 qui permet de concaténer deux chaines de caractères déclarées dans le segment des données et d'afficher le résultat.

2. Ecrire un programme en assembleur 80x86 qui permet la concaténation et l'affichage par l'utilisation de deux procédures « *Concaténer* » et « *Afficher* ».

Chapitre 04 : Interprétation et Compilation : Notions de base

4.1 Notions de base

Aujourd'hui, la majorité des programmeurs utilisent les langages de programmations de haut niveau, à savoir, le C ou le C++, java sans la moindre connaissance du mécanisme de fonction et d'exécution du langage assembleur. Ceci est dû d'une part à la complexité et ambigüité des syntaxes de ce langage. D'autre part, les syntaxes et les structures des langages de haut niveau sont plus simples en termes de compréhensions et implémentation. Par exemple, un code de quelques lignes en langage de haut niveau, prendrait en assembleur quelques dizaines de lignes (par fois quelques centaines de lignes). Néanmoins, les langages d'assemblages peuvent être considérés comme la clef pour la compréhension de l'architecture et de fonctionnement des processeurs.

Afin d'approfondir les connaissances sur le fonctionnement et l'exécution des programmes en assembleur, nous examinons dans ce chapitre la manière dont le compilateur traduit le C en langage assembleur (et ainsi en langage machine). Nous allons commencer par étudier la notion d'interprétation et de compilation d'un programme tout en présentant quelques langages de chaque type. Nous verrons ensuite les étapes intervenant dans le processus de génération des fichiers binaires des programmes. Le chapitre se termine par quelques exemples de traduction des programmes C en langage assembleur.

4.1.1 Interprétation d'un programme

L'exécution d'un programme à partir de son code source, ne peut être réalisée qu'à travers deux méthodes, l'interprétation ou la compilation (traduction).

L'interprétation ? On parle donc *D'interpréteur* de programmes

Un interpréteur : C'est le programme chargé du décodage du programme instruction par instruction tout en exécutant les actions correspondantes.

La figure 4.1. Montre un schéma simplifié pour l'interprétation d'un programme source.

4.1.2. Compilation et exécution d'un programme

La deuxième méthode pour l'exécution d'un programme est la compilation, parfois appelée traduction d'un programme. Le processus de compilation passe par plusieurs étapes comme le montre la figure 4.2.

Qu'est ce qu'un

Compilateur

Le compilateur : Cette méthode nécessite la traduction de chaque instruction du fichier source en une suite d'instructions en langage machine. L'exécution est ensuite achevée à partir du fichier généré (fichier binaire).

Remarque : l'exécution des programmes compilés est plus rapide que les programmes interprétés car la traduction est déjà faite.

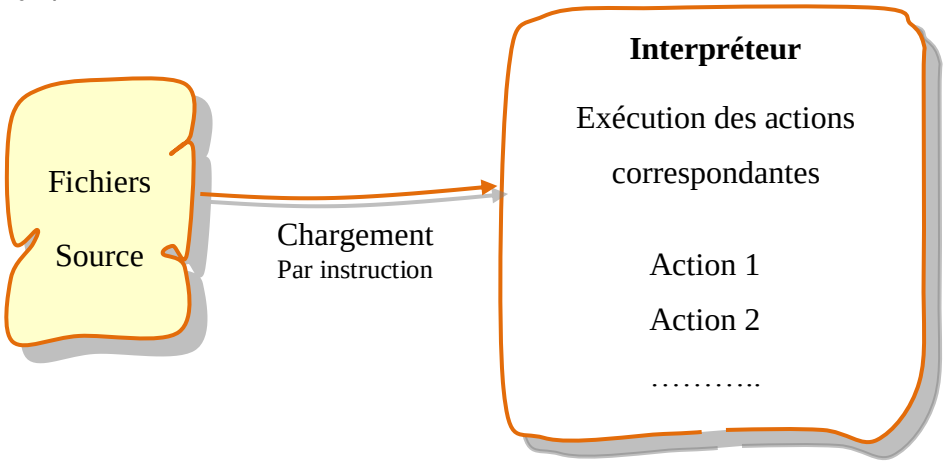


Figure 4.1 : *Un schéma général (simplifié) de l'interprétation d'un programme.*

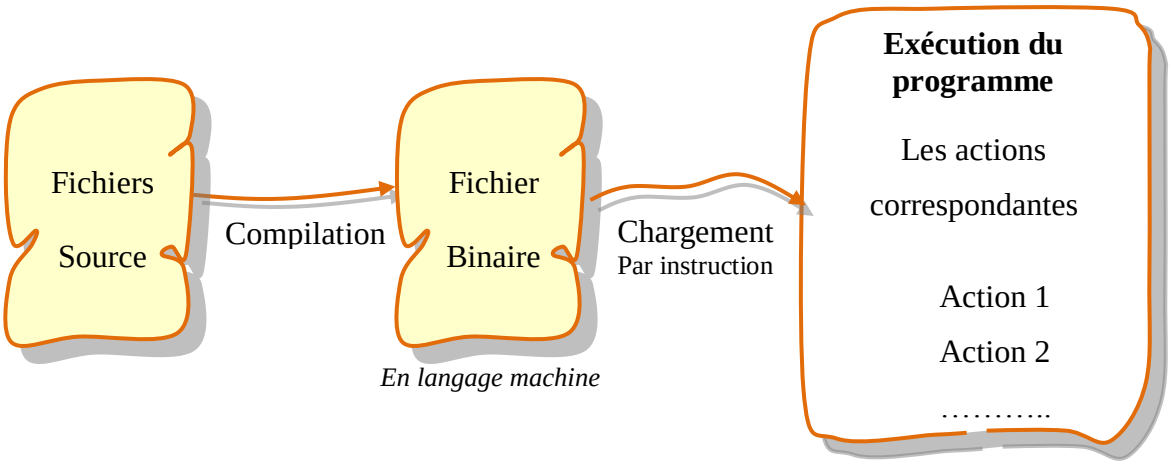
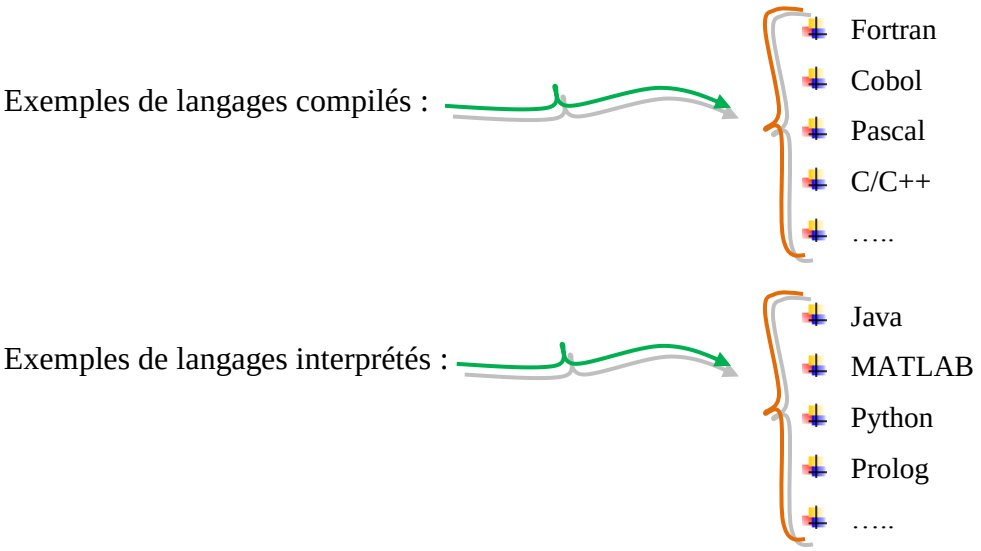


Figure 4.2 : *Un schéma général (simplifié) d'exécution d'un programme par compilation.*

4.1.3 Exemples d'interpréteurs et de compilateurs

Le classement des langages de programmations de haut niveau peut être élaboré en fonction de plusieurs critères. Nous distinguerons dans cette section les langages seulement en fonction de la méthode d'exécution.



Remarque : la notion de compilation et d'interprétation semble parfois ambigüe. A titre d'exemple, l'exécution d'un programme en langage Java passe premièrement par une phase de compilation vers un langage intermédiaire (bytecode), par la suite ce bytecode sera lui même interprété.

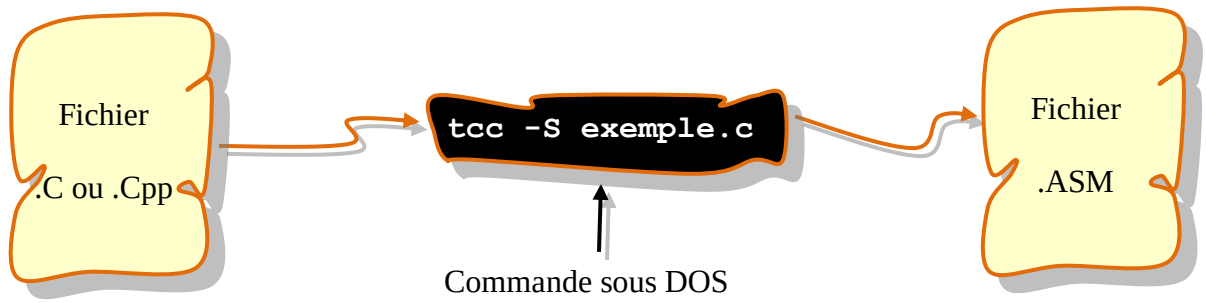
4.2 Du langage C vers l'assembleur

4.2.1. Création d'un fichier .ASM à partir d'un fichier .C

Afin de bien comprendre le fonctionnement de la pile en assembleur nous essayerons dans cette section d'étudier la traduction en assembleur des programmes en langage de haut niveau. Nous nous intéressons seulement aux programmes en langage C. pour ce faire nous utilisons :

- Un compilateur Turbo C++ version 3 sous DOS,
- Des entiers de 16 bits;
- Un modèle de mémoire "small".

La méthode à utiliser pour générer un fichier .ASM à partir d'un fichier .c ou .cpp est :



4.2.2 Exemple 1 : cas d'un programme simple

Soit le programme .C suivant [02] :

```
void main(void) {
   char X = 11;
   char C = 'A';
   int Res;
   if (X < 0)
      Res = -1;
   else
      Res = 1;
}
```

Un exemple de programme en langage C

Dans le programme ci-dessus, trois variables sont définies : deux variables, *X*, *C* initialisées successive à *11 et 'A'* et *Res* de type *Int* non initialisée. Ce programme permet de ranger 1 ou -1 dans la variable *Res* en fonction du signe de la variable X.

Le fichier .ASM obtenu après la traduction par Turbo C est:

Pas de segment de données : les variables sont allouées sur la pile par **`SUB sp, 4.`**

Allocation de 4 octets.
1 octet pour X
1 octet pour C
2 octet pour Res

`char C = 'A';`

`Res = -1;`

Étiquette

`}//` fin du programme

```
_TEXT SEGMENT byte public 'CODE'
 ASSUME cs:_TEXT
_main PROC near

PUSH bp
MOV bp,sp
SUB sp, 4
MOV byte ptr [bp-1], 11
MOV byte ptr [bp-2], 65
CMP byte ptr [bp-1], 0
JGE @1@86
MOV word ptr [bp-4], 65535
JMP @1@114
@1@86:
MOV word ptr [bp-4], 1
@1@114:
MOV sp,bp
POP bp
RET

_main ENDP
_TEXT ENDS
END
```

```
Void main(void) {
```
Ici c'est une procédure
```
    PROC near


Char X = 11;

Int Res;
If (X < 0)
```

Saut conditionnel vers étiquette

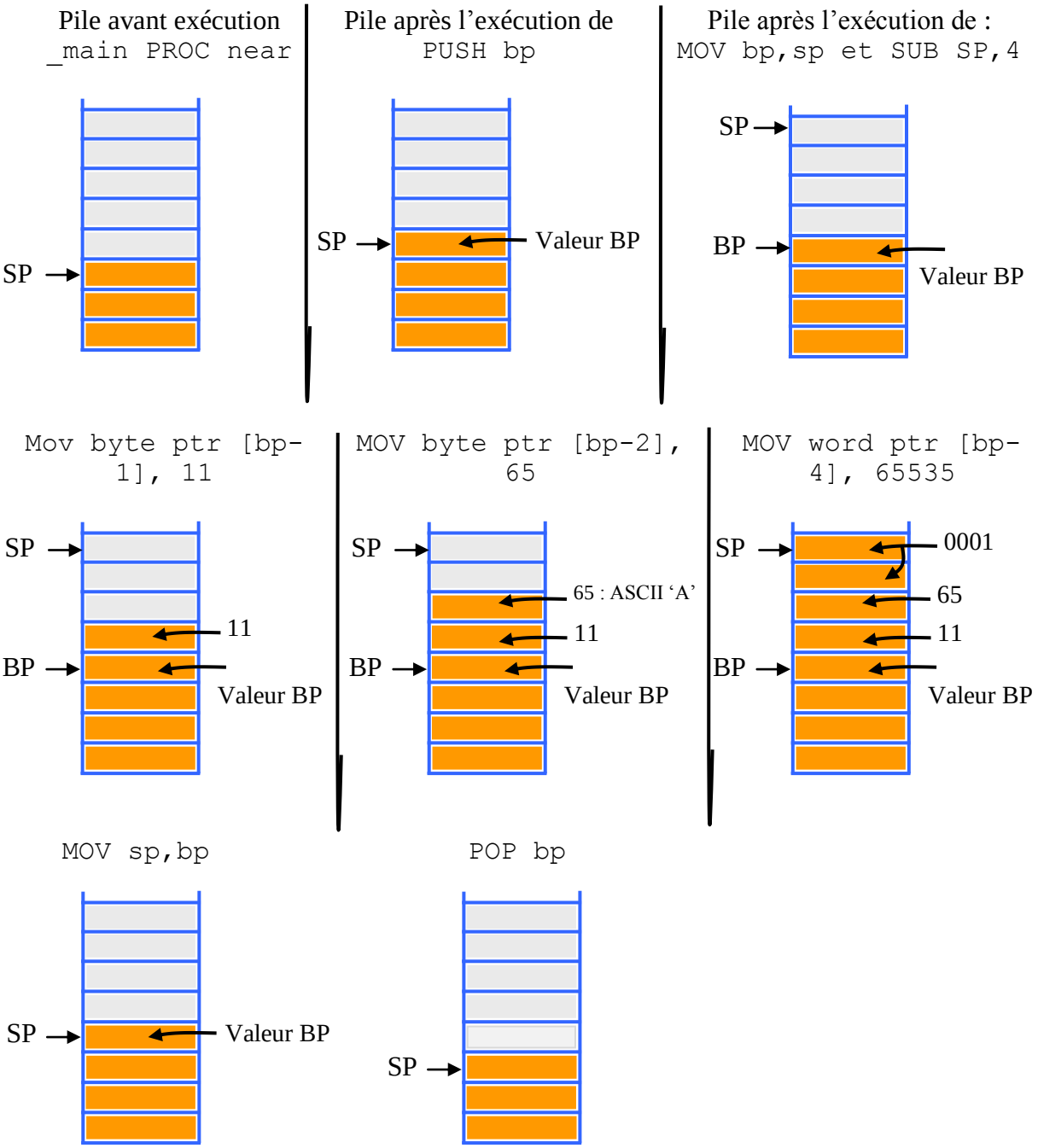
```
Else
Res = 1;
```

Étiquette placée par le compilateur

Remarque : Comme nous l'avons vu dans le chapitre précédent, `MOV byte ptr [bp-1], 11` *indique que la valeur 11 doit être stockée en mémoire su un octet.*

Dépilement

La pile lors de l'exécution



4.2.3. Exemple 2 : Cas d'un sous programme C

L'exemple de cette section [02] présente un cas d'une procédure en assembleur avec passage de paramètre par pile. Après l'appel procédure le résultat est stocké dans le registre AX. Après le CALL, le résultat est empilé.

```
int ma_fonction( int x, int y ) {
   return x + y;
}
void main(void) {
   int X = 11;
   int Y = 22;
   int Res;
   Res = ma_fonction(X, Y);
}
```

Un exemple de programme en langage C en utilisant une fonction

Le code assembleur correspondant est le suivant :

```
_TEXT SEGMENT byte public 'CODE'
; int ma_fonction( int x, int y ) {
  ASSUME cs:_TEXT
  _ma_fonction PROC near
  PUSH bp
  MOV bp,sp
; return x + y;
  MOV ax, [bp+4]
  ADD ax, [bp+6]
; }
  POP bp
  RET
  _ma_fonction ENDP
; void main(void) {
  ASSUME cs:_TEXT
  _main PROC near
  PUSH bp
  MOV bp,sp
  SUB sp,6
; int X = 11;
  MOV [bp-2], 11
; int Y = 22;
  MOV [bp-4], 22
; int Res;
; Res = ma_fonction(X, Y);
  PUSH word ptr [bp-4]
  PUSH word ptr [bp-2]
  CALL _ma_fonction
  ADD sp, 4
  MOV [bp-6],ax
; }
```

```
    MOV sp,bp
    POP bp
    RET
    _main ENDP
    _TEXT ENDS
;
```

Pour obtenir l'état de la pile lors de l'exécution vous pouvez appliquer la même méthode du premier exemple.

4.3. Exercices

Exercice 01.

Donner le contenu de la pile après l'exécution de chaque instruction pour l'exemple de la section 4.2.3.

Exercice 02.

1. Donner la traduction en assembleur de la fonction récursive suivante :

```
int produit(int x, int y)
{
  if (x > 0)
    return produit(x - 1, y) + y;
  else
    return 0;
}

int main(void)
{
  int a = 4, b = 3;

  printf("%d * %d = %d\n", a, b, produit(a, b));
  return 0;
}
```

2. Etudier l'état de la pile lors de l'exécution.

Chapitre 05 : Interruptions et Appels systèmes

5.1. Définition d'une interruption

Une interruption peut être vue comme un mécanisme qui permet au matériel externe (périphériques) de communiquer avec le processeur. Les interruptions permettent ainsi d'interrompre par exemple l'exécution d'un processus suite à un événement extérieur déclenché par un signal d'interruption. Ces interruptions sont appelées interruptions matérielles (voir section 5.2):

Les interruptions sont déclenchées par un *signal d'interruption*

C'est un signal électrique envoyé au processeur sur *une borne spéciale*.

Section 5.3.1

L'interruption d'un processus est contrôlée par une routine dite « *routine d'interruption* » ou « *traitement d'interruption* ». Ce traitement consiste:

- Soit, à masquer cette interruption jusqu'à la fin de l'exécution du traitement en cours, on parle ainsi d'une *Interruption masquable*. Dans ce cas, le processeur *démasque* ces interruptions après la fin du traitement.
- Soit, à exécuter immédiatement cette interruption à l'arrivée, on parle ainsi, *d'une interruption non masquable*.

Le traitement d'une interruption est effectué par l'exécution d'un *traitant d'interruption ([03] pour plus de détails)*

Qu'est ce qu'un *Traitant d'interruption*

- Sont appelés automatiquement lorsqu'une interruption survient,
- Sont similaires à des procédures ordinaires,
- Ils doivent se terminer par l'instruction IRET.
- IRET: Reprendre l'exécution à l'endroit où elle avait été interrompue.
- Sont appelés à travers la table des *vecteurs d'interruptions* par l'instruction INT n

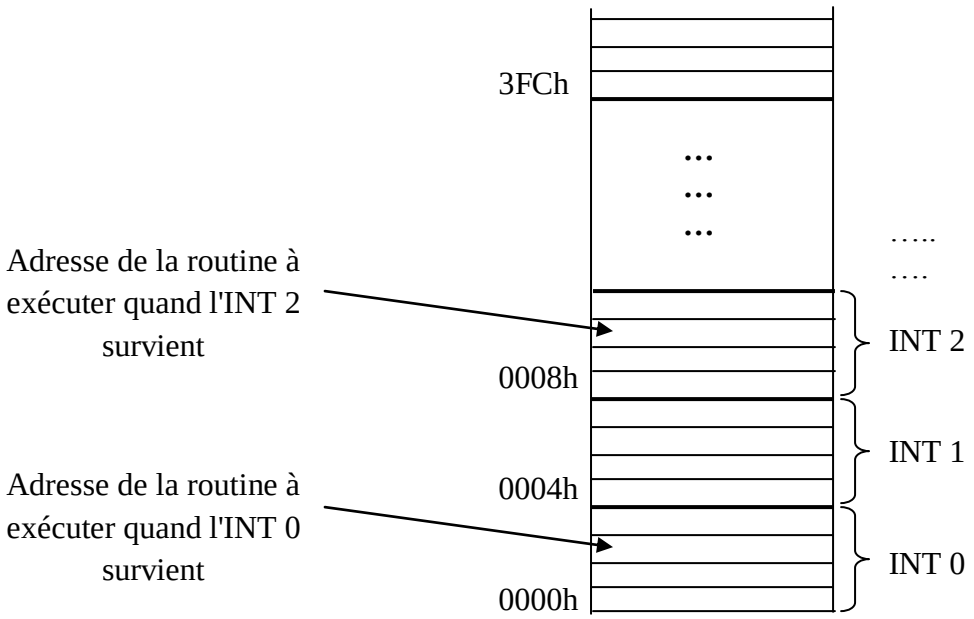
Les vecteurs d'interruptions

Un vecteur d'interruption est l'un des éléments d'une zone mémoire appelée table des vecteurs d'interruptions. Cette table contient les adresses des traitants d'interruptions où

chaque élément de cette table correspond à *un vecteur d'interruption*. La table des vecteurs est implémentée à partir du premier emplacement de la RAM ; adresse 00000000h. La taille de cette table est 256*4 octet soit, 256 vecteurs de 4 octets chacun. L'appel de chaque vecteur est effectué par l'instruction *INT n* où le microprocesseur se branche à l'adresse 4*n (chaque vecteur occupe 4 octets).

Exemple :

L'adresse du traitant d'interruption appelé par le vecteur d'interruption 0 (INT 0) est stockée dans les 4 premiers emplacements mémoire de 0000h à 0003h. Le MPU se branche vers cette interruption lorsqu'une erreur de division par 0 est survient.



5.2. Types d'interruptions

Les interruptions sont classées généralement en trois grands types :

Interruptions matérielles: C'est une interruption signalée au processeur via un événement externe (touche clavier, Signale Horloge, … etc). Ces interruptions peuvent être *Masquable* ou *Non Masquable*

Interruptions logicielles: permettent d'appeler des fonctions du système d'exploitation et du BIOS à l'aide des instructions spécifiques. A titre d'exemple, l'instruction INT n où n est le numéro de vecteur d'interruption dans la *table des vecteurs d'interruptions*.

Exceptions (ou Déroutement): déclenchées par une erreur interne du processeur telles que par exemple division par zéro, débordement,… , etc).

5.3. Interruption matérielle

Nous étudions dans cette section le fonctionnement des interruptions matérielles. Nous nous focalisons sur l'exemple du PC. Elles sont appelées IRQ (IRQ: Interrupt ReQuest), la figure 5.1 montre un exemple de 9 interruptions (IBM PC/XT) numérotées de 0 à 8.

N° d'interruption	Adresse de la routine d'interruption
IRQ0	Horloge Interne
IRQ1	Clavier
IRQ2	
IRQ3	Interface série 2
IRQ4	Interface série 1
IRQ5	Disque dure
IRQ6	Lecteur de disquette
IRQ7	L'interface parallèle
IRQ8	Horloge temps réel

Figure 5.1. *Un exemple de quelques interruptions matérielles.*

5.3.1. Bornes pour les interruptions

Les processeurs de la famille 80x86 possèdent trois bornes pour gérer les interruptions : NMI (borne pour les interruptions non masquables), INTR, et INTA comme il est présenté dans la figure 5.2.

Remarque :

Les interruptions non masquable sont le plus souvent utilisées pour la détection des erreurs matérielles (par exemple panne d'alimentation, erreur mémoire, …,etc).

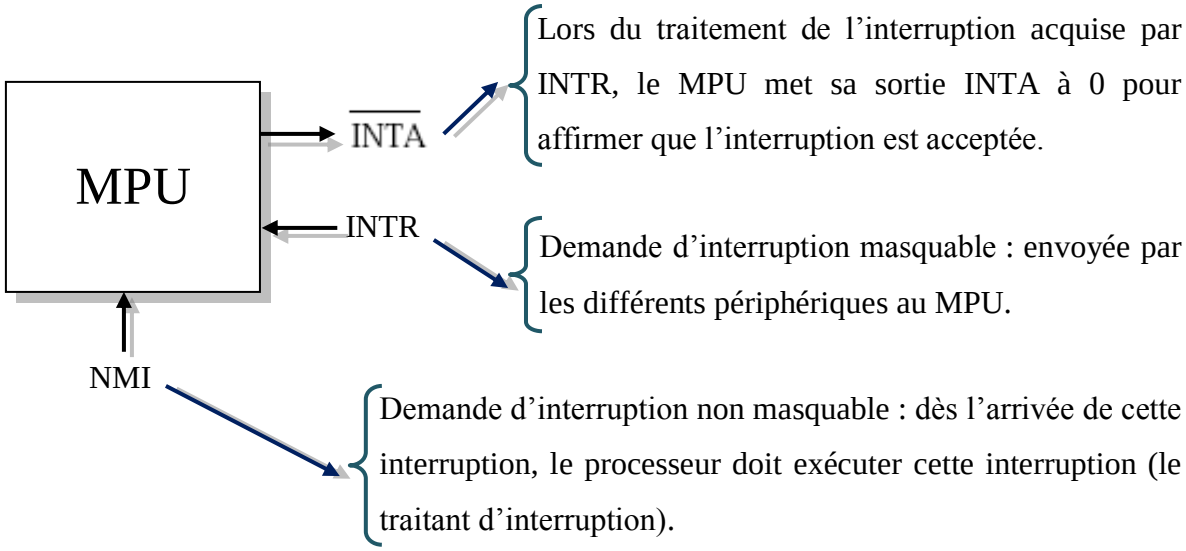


Figure 5.2 : *Bornes d'interruptions.*

5.3.2. Contrôleur d'interruptions dans un PC

Les demandes d'interruptions provenant de l'ensemble des périphériques sont gérées par un circuit intermédiaire (extérieur au processeur) appelé *le contrôleur d'interruptions*. Ce circuit reçoit les demandes d'interruptions par ses bornes IRQ comme le montre la figure 5.3. Ensuite, les demandes recueillies sont envoyées une par une au processeur à travers la borne INTR. Les demandes sont envoyées une à une car le processeur ne dispose que d'une seule borne pour les demandes d'interruptions.

La figure 5.3 montre un schéma simplifié de la liaison entre le contrôleur d'interruption (*PIC, Programmable Interruption Controler) et le MPU.*

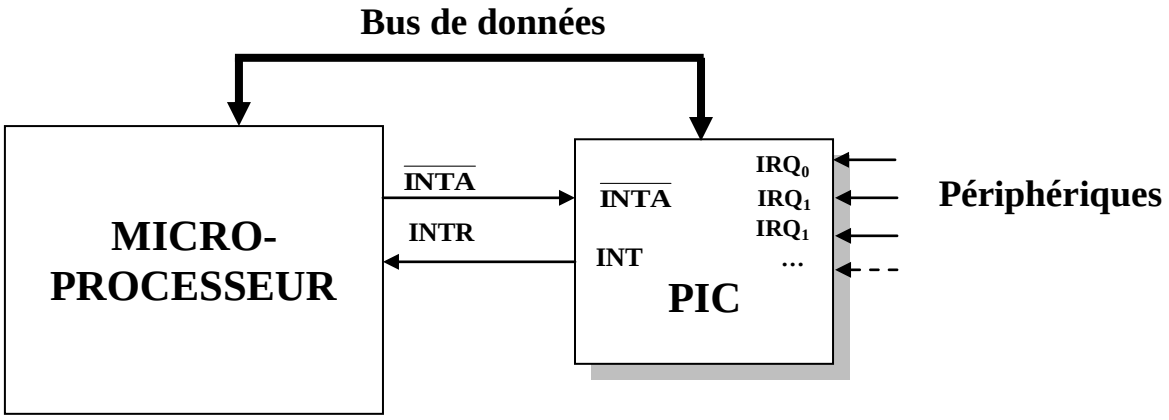


Figure 5.3 : *Schéma simplifié du contrôleur d'interruptions.*

Le contrôleur ne peut envoyer une nouvelle demande d'interruption au processeur qu'après l'acquisition de l'autorisation sur INTA par le processeur. Ce signal d'autorisation est envoyé par le processeur après l'achèvement du traitement de l'interruption en cours. La section suivante donne les différentes étapes d'exécution d'une demande d'interruption.

5.3.3. Déroulement d'une interruption masquable

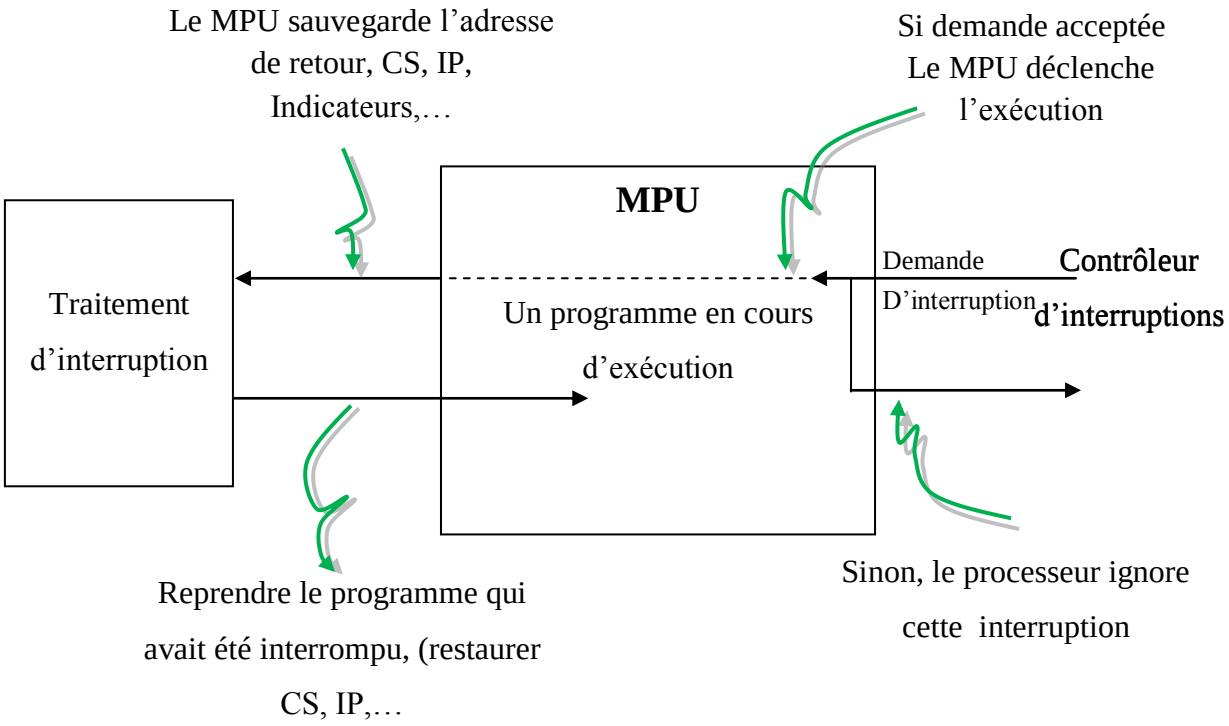


Figure 5.4 : *Déroulement d'une interruption masquable.*

Les grandes lignes de déroulement d'une interruption masquable sont données par le schéma explicatif de la figure 5.4. Ainsi, les étapes d'exécutions peuvent être résumées comme suit:

Etape 01 : Après avoir reçu la demande d'interruption envoyée par le contrôleur sur sa borne INTR, le microprocesseur accepte ou ignore cette demande en fonction du contenu de l'indicateur IF,

Si Indicateur IF == 1, la demande est acceptée
Sinon la demande est ignorée

Indicateur IF
(*Interrupt Flag)*

C'est l'un des bits *du registre d'état* (discuté dans les chapitres 01 et 02) du MPU. La modification de IF est réalisée à l'aide de :

- Instructions CLI (***CL**ear **I**F*, IF ==0),
- Instruction STI (***S**e**T** **I**F*, IF == 1).

Remarque : Après l'acquisition du signal d'interruption, le microprocesseur ne peut entamer le traitement de cette interruption qu'après l'exécution de l'instruction en cours.

Etape 02 : Dans cette étape, le contrôleur d'interruption est informé si sa demande est acceptée ou non via la borne INTA. En effet, la valeur 0 sur cette sortie signifie que la demande est acceptée par le microprocesseur.

Etape 03 : Si la réponse du MPU est positive, c'est-à-dire que la demande est acceptée, le contrôleur d'interruption met le numéro de cette demande d'interruption sur le bus de données comme le montre la figure 5.3. Le numéro de la demande d'interruption à envoyer au MPU est associé à l'une des bornes IRQ_i

Etape 04 : A l'aide du numéro d'interruption récupéré par le MPU via le bus de données, Le processeur retrouve le vecteur d'interruption qui corresponde. L'exécution du traitant d'interruption se déroule ensuite normalement.

Remarque :

1. Avant l'exécution du traitant d'interruption le processeur :

- *Positionne l'indicateur IF à 0 afin de masquer les interruptions suivantes;*
- *Empile le contenu du registre d'état afin de sauvegarder les différents indicateurs;*
- *Empile le contenu des registres CS et IP pour ne pas perdre l'adresse de retour comme le montre la figure 5.4 ;*

2. L'appel du traitant d'interruption correspond à un appel système ou ce qu'on appel une interruption logicielle (voir la section 5.4).

Etape 05: à la fin du traitant d'interruption, l'instruction IRET dépile le contenu des registres CS, IP et registre d'état. Ceci, permet de reprendre l'exécution du programme qui avait été interrompu.

5.4 Appel système

5.4.1. Modes d'exécution : *superviseur* et *utilisateur*

Les modes d'exécution implémentés par les processeurs sont nombreux, nous nous focaliserons dans ce cours seulement sur les deux modes, superviseur et utilisateur, pour plus de détails, veuillez vous référer à [04]:

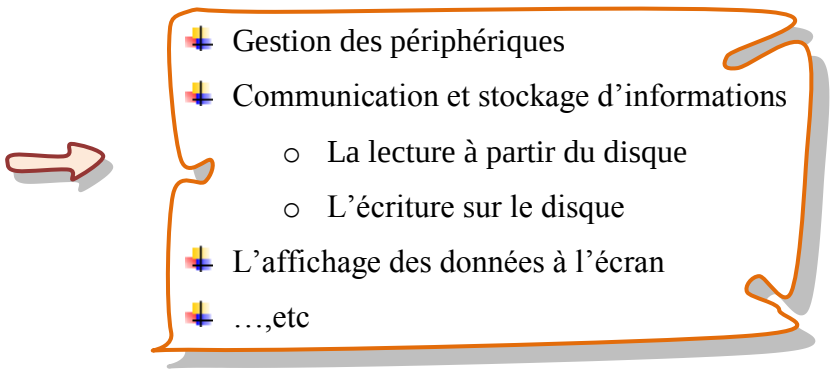
Le mode superviseur : Appelé aussi mode noyau car seul le *système d'exploitation* doit y avoir accès (mode d'exécution du système). Dans ce mode toutes les instructions sont autorisées, autrement dit, c'est le mode d'exécution privilégié dont l'accès est pleinement autorisé aux instructions du processeur, aux mémoires, …etc.

Le mode utilisateur: Les applications *utilisateurs* doivent toujours s'exécuter dans ce mode. A la différence du mode superviseur, l'accès n'est pas totalement autorisé, à titre d'exemple, l'usage d'une instruction interdite provoque une *erreur*. Toutefois, le passage du mode utilisateur vers le mode superviseur peut être provoqué par, à titre d'exemple, un appel système ou par l'arrivée d'une interruption.

5.4.2. Appel système

Les appels système sont des fonctions fournies par le mode superviseur du système d'exploitation. Ces fonctions offrent la possibilité au mode utilisateur de demander au SE une action par exemple sur le matériel (interdit par le mode utilisateur). Les appels système permettent d'appeler les routines (traitants d'interruptions) de service du BIOS et du système d'exploitation, on parle alors d'interruptions logicielles.

Parmi les fonctions de ces *interruptions*



Les appels système sont implémentés par des instructions machine permettant de basculer le processeur facilement en mode superviseur. Ces instruction varient d'un système à l'autre, nous avons vu un exemple dans la première section ; *l'instruction INT*.

5.4.3. Exemples

Fonctions BIOS

Le BIOS (*Basic Input Output System*) !c'est quoi ?

- Un petit programme se trouve en ROM (EEPROM ou FLASH)
- Forme la couche basse de tous les systèmes d'exploitation sur PC.
- Responsable de la gestion du matériel : clavier, écran, disques durs, …etc.

INT	**Fonction**	
0	Division par 0	appelé automatiquement lors de div. par 0
5	Copie d'écran	
10H	Ecran	gestion des modes vidéo
12H	Taille mémoire	
13H	Gestion disque dur	(initialiser, lire/écrire secteurs)
14H	Interface série	
16H	Clavier	(lire caractère, état du clavier)

Fonctions MS-DOS

Contrairement au BIOS, les fonctions de l'MS-DOS s'appellent toutes par un seul vecteur d'interruption 21H (33*4 dans la table des vecteurs d'interruptions). En effet, avant d'utiliser INT 21h, la fonction à appeler doit être précisée dans le registre AH comme suit :

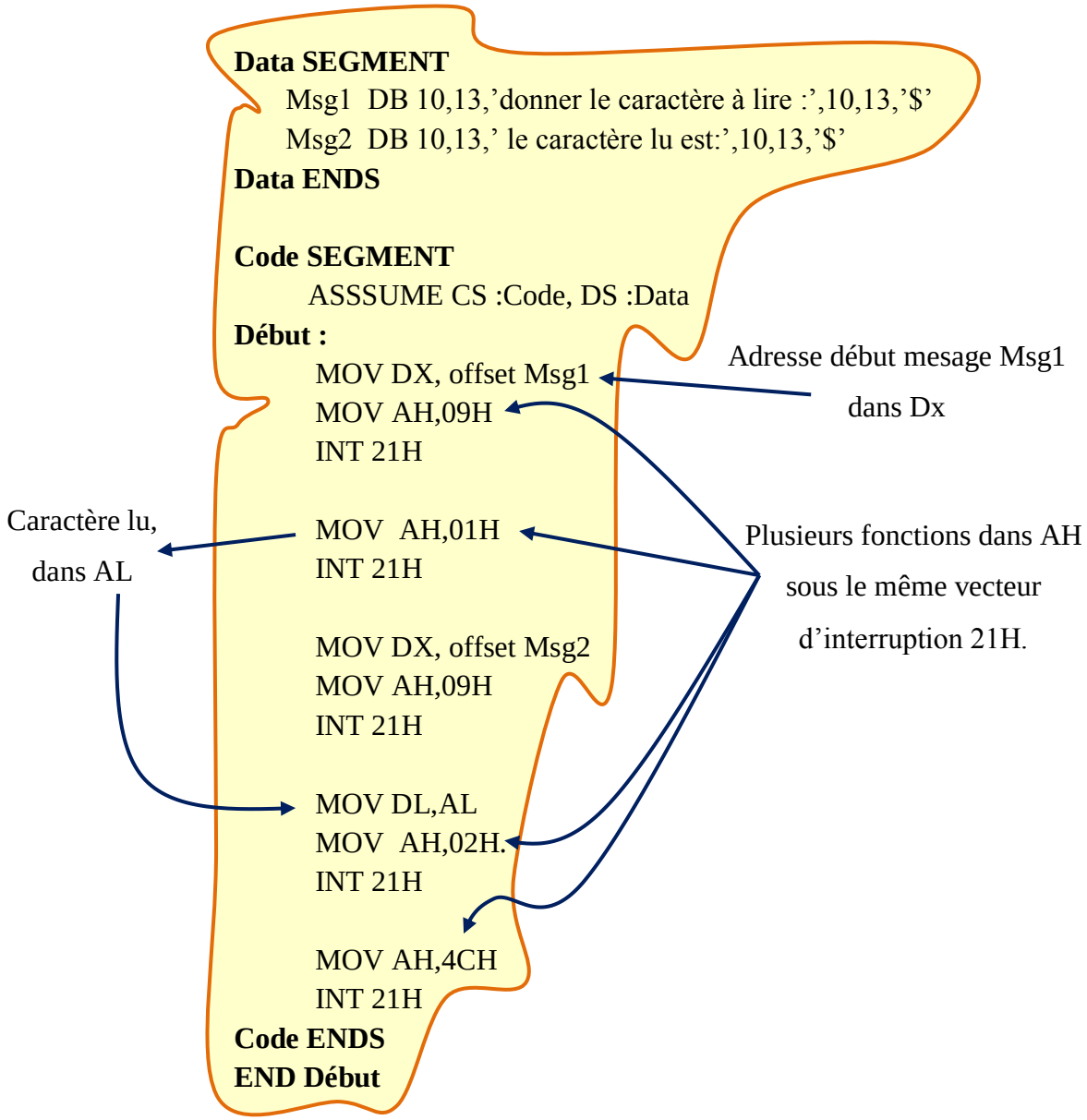
```
MOV AH, numero_fonction
INT 21H
```

Exemple : Quelques fonctions du DOS (Consulter [05] pour la liste entière):

Numéro	**Fonction**
01H	Lecture caractère met le code ascii lu dans AL
02H	Affiche caractère code ascii dans registre DL
06H	lecture directe d'un caractère via le registre DL
09H	Affiche une chaîne de caractères où l'adresse début dans DX
0AH	Entrée d'une chaîne de caractères
0BH	Lit état clavier met AL=1 si caractère, sinon 0.

2CH	Demande l'heure courante.
2DH	Fixer l'heure courante.
4CH	Permet de terminer un programme et de revenir au DOS

Exemple de programme: Affichage d'une chaine de caractère et lecture et affichage d'un caractère sur l'écran :



5.5. Exercices

Exercice 01 : Après un appel système, expliquez comment le processeur se branche à un traitant d'interruption?

Exercice 02 : Dans la table des vecteurs d'interruptions, les adresses des routines d'interruptions restent les même après le redémarrage du PC ou non, pour quoi ?

Exercice 03 : la même question que l'exercice 02, pour les adresses de la table des vecteurs elle-même.

Chapitre 06 : Les architectures modernes

6.1. Introduction

Un microprocesseur est le cerveau de l'ordinateur. Cet élément peut être considéré comme un processeur dont tous les composants ont été convenablement miniaturisés pour être regroupés dans quelques millimètres carrés. Le microprocesseur est un circuit complexe constitué de plusieurs millions de composants, il est chargé d'effectuer les calculs nécessaires au fonctionnement de l'ordinateur et au traitement des données.

La puissance des microprocesseurs continue de s'accroître et leur encombrement diminue régulièrement. Ceci a permis un développement incroyable des systèmes informatique ainsi qu'une expansion appréciable de la téléphonie et de l'Internet.

6.2. Architectures CPU

La classification des architectures CPU est élaborée en fonction de la nature des instructions exécutées par les microprocesseurs.

6.2.1. Architecture CISC & RISC

Plusieurs architectures sont envisageables dont les plus importantes sont CISC (Complex Instruction Set Computer) et RISC (Reduced Instruction Set Computer).

Architecture CISC

Cette architecture a été créée principalement pour compenser le temps d'exécution entre la mémoire et le processeur. Ceci est dû au fait que la mémoire fonctionnait très lentement par rapport au processeur. Donc, l'idée est de soumettre au microprocesseur des instructions complexes qui demanderaient autant d'accès mémoire que plusieurs petites instructions.

Caractéristiques :

- Un grand nombre d'instructions spécialisées,
- Instructions complexes prenant plusieurs cycles d'horloge,
- Peu de registres CPU,
- Instructions de tailles différentes,
- Beaucoup de modes d'adressage,
- Compilateur simple.

Inconvénients :
- Fréquences d'accès à la mémoire élevée dû au nombre limité de registres,
- Les instructions complexes ne sont que rarement utilisées,
- Décodeur complexe (microcode) : Parce que le code machine des instructions varie d'une instruction à l'autre.

Exemples :
- La plupart des vieilles architectures,
- Famille Intel,
- Famille AMD,
- …,etc.

Architecture RISC

L'idée de base de cette architecture repose sur le fait que :

80% des traitements des langages de haut niveau faisaient appel a seulement 20% des instructions du microprocesseur.

Caractéristiques :
- Un jeu d'instructions relativement réduit,
- Chaque instruction est censée être exécutée en un seul cycle d'horloge,
- Un nombre important de registres CPU,
- Instruction au format fixe,
- Peu de modes d'adressage,
- Décodeur simple.

Remarque : le nombre élevé de registres permet de diminuer la fréquence d'accès à la mémoire. Cette caractéristique favorise également une utilisation optimale de la technique pipeline (section 6.4).

Inconvénients : Compilateur complexe et programmation assembleur difficiles

Exemples :
- Famille MIPS,
- Famille Motorola,
- Power PC,
- Sun Sparc,
- …etc

6.2.2. Autres architectures

Toujours dans le but d'augmenter la vitesse d'exécution des microprocesseurs plusieurs architectures ont été étudiées et implémentées. L'amélioration de ces architectures vise constamment les instructions à exécuter (nombre, format,…etc). Dans cette section nous exposerons brièvement les deux architectures *VLIW* (Very Long Instruction Word) et *EPIC* (Explicitely Parallel Instruction Computer). Pour plus de détails veillez-vous référer à [06].

Architecture VLIW

Caractéristiques :
- Chaque instruction peut faire 128, 256 bits de long, voire plus,
- Chaque instruction est codée en + opérations codée chacune sur 32 bits,
- Ces opérations peuvent être exécutées simultanément,
- 64 registres,

Exemple : *CPU Transmeta Crusoe*

Architecture EPIC

Caractéristiques :
- instruction de longueur 128 bits,
- Chaque instruction comprenant 3 instructions de 41 bits + 5 bits Pour l'indentification du type d'instruction,
- Les mêmes instructions de la famille x86 sont gardées dans cette architecture,

Exemple : *CPU Intel ITANIUM*.

6.3. Cycle d'exécution des instructions

Dés le démarrage de l'ordinateur le CPU commence sans arrêt l'exécution des programmes chargés en mémoire. Le cycle d'exécution d'une instruction dépend de l'instruction à exécuter, à titre d'exemple, le cycle d'exécution d'une instruction sans opérande (seulement code opération) est différent de celui d'une instruction avec opérande. La figure 6.1 [07] illustre un diagramme d'état du cycle d'exécution d'une instruction.

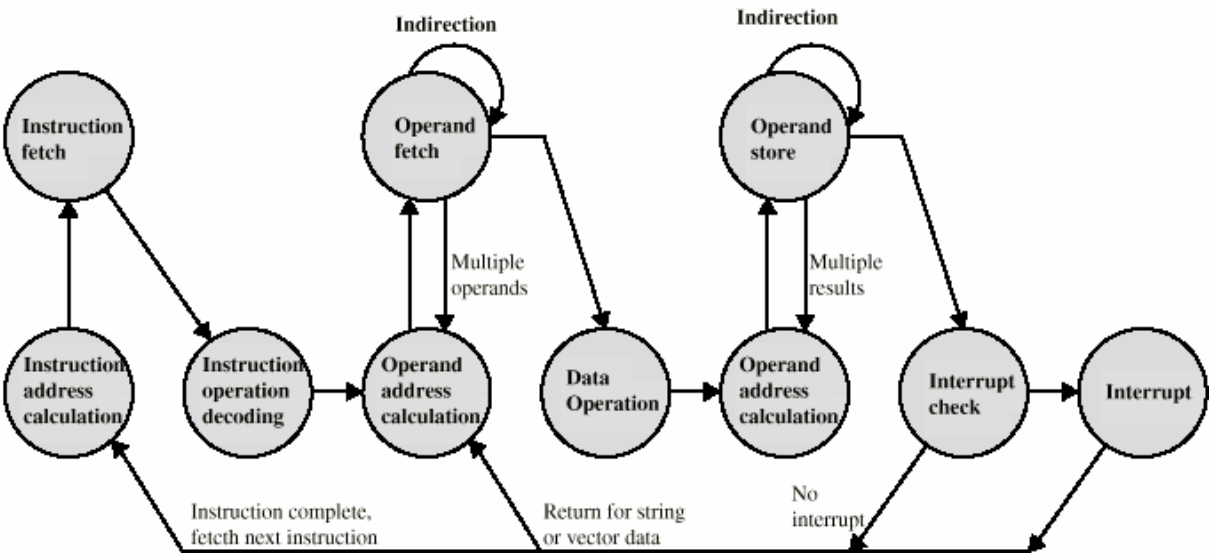


Figure 6.1 : *Diagramme d'état du cycle d'exécution d'une instruction*.

Les phases d'exécution sont résumées comme suit:

- **Phase 1:** *Fetch* **(***Recherche de l'instruction)*
  1. L'unité de commande met le contenu du registre IP (l'adresse de l'instruction suivante) sur le bus d'adresse.
  2. Le contenu de l'emplacement mémoire de cette adresse est ensuite envoyé via le bus de données.
  3. Stockage de l'instruction dans le registre RI (registre instruction).
- **Phase 2 :** Décodage de l'instruction

  Cette phase est composée de deux étapes :

  Décodage de l'instruction :
  1. Lecture éventuelle des autres mots d'instruction (selon le format),
  2. Transformation de l'instruction par l'unité de commande en une suite de commandes élémentaires nécessaires au traitement de l'instruction.

  Recherche éventuelle de l'opérande :
  3. Si l'instruction contient un code opérande, une donnée doit donc être récupérée à partir de la mémoire par l'unité de commande via le bus de données.
  4. Ensuite, l'opérande est stocké dans un registre du CPU.
- **Phase 3 :** *Exécution de l'instruction*
  1. Le microprogramme exécute l'instruction.

2. Positionnement des indicateurs par la modification du *registre d'état*.
3. Sauvegarde de l'adresse de l'instruction suivante dans le registre IP.

Le microprocesseur est caractérisé par sa puissance de traitement d'instructions par seconde. Cette caractéristique est estimée en utilisant :

$$MIPS = \frac{F_H}{CPI}$$

$$CPI = \frac{NombreCyclesPourUnProgramme}{NbrIns}$$

Où :

MIPS (Millions d'Instructions Par Seconde) désigne la puissance de traitement.

CPI (Cycle Par Instruction) est le nombre moyen de cycles d'horloge par instruction.

F_H représente la fréquence d'horloge en MHz (dépend de la technologie et de l'architecture matérielle).

NbrIns (nombre d'instructions) dépend de l'efficacité des compilateurs.

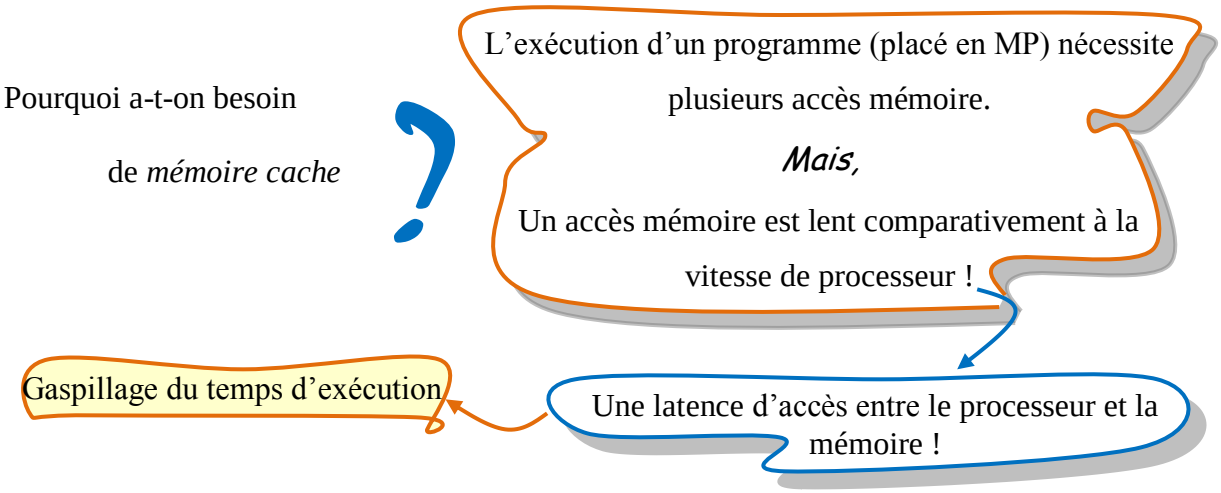
6.4. Amélioration des performances CPU

En effet, l'augmentation des performances d'un microprocesseur peut être réalisée par l'interférence de plusieurs techniques, à savoir :

- L'augmentation de la fréquence d'horloge : Cette technique présente l'inconvénient d'élévation de température provoquée par le surcroît de consommation, un système de refroidissement adéquat est donc nécessaire.
- La diminution du CPI à travers l'amélioration du jeu d'instruction en termes de format, complexité, …etc. Le CPI peut également être amélioré par l'optimisation du compilateur.
- L'optimisation du temps des transferts entre la mémoire et le processeur.
- …etc

En conséquence, de nombreuses améliorations ont été inventées et adoptées dans la constitution des CPU en tenant compte de la corrélation entre les points cités ci-dessus.

6.4.1 La mémoire cache



Principe de la mémoire cache

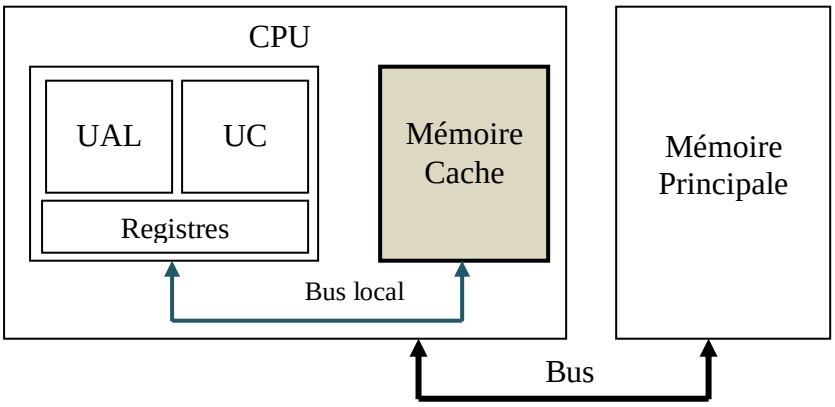


Figure 6.2 : *Principe de la mémoire cache.*

C'est une mémoire volatile très rapide située entre le processeur est la MP. Sa taille est beaucoup plus petite que celle de la MP. Cette mémoire est utilisée notamment pour diminuer les accès à la mémoire principale. En d'autres termes, son rôle primordial est le stockage temporaire des instructions et données les plus susceptibles d'être réutilisées par le CPU.

Le principe dans ce cas est simple, si le processeur a besoin par exemple d'une instruction, il va d'abord la chercher dans la mémoire cache :

- Si cette instruction est présente dans le cache, Il la récupère, on parle donc de succès.
- Si cette instruction n'est pas disponible, il va la chercher dans la MP, dans ce cas c'est un échec. Après la récupération de l'instruction à partir de la MP, le processeur doit la sauvegarder aussitôt dans la mémoire cache.

6.4.2. Technique du pipeline

Architecture simple :

Dans une architecture microprocesseur simple les instructions sont exécutées les unes après les autres, c'est-à-dire, séquentiellement comme le montre la figure 6.3.

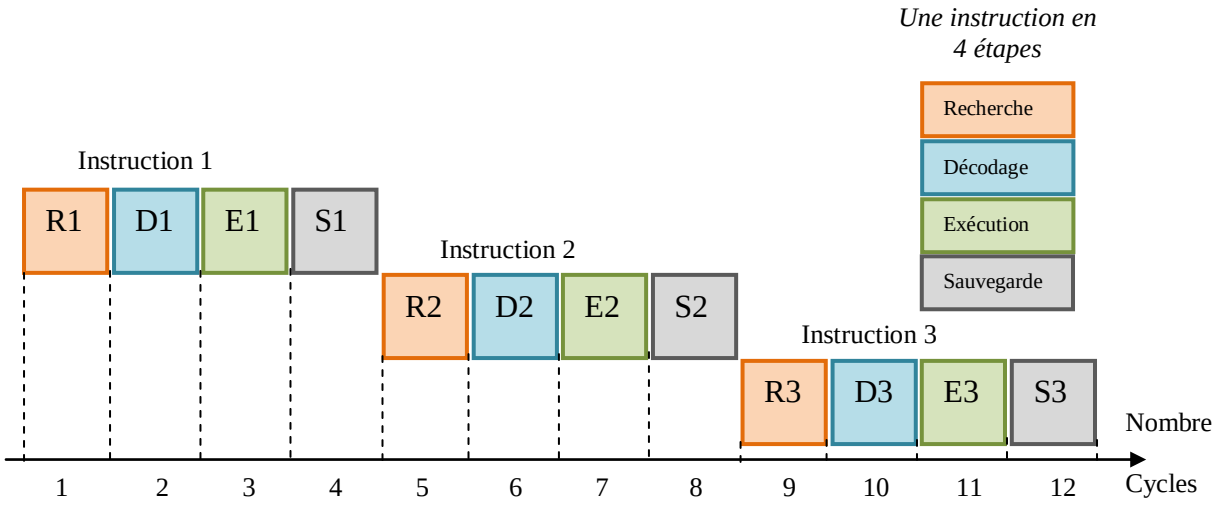


Figure 6.3 : *Exemple d'exécution dans une architecture microprocesseur simple.*

L'exécution d'une étape d'une instruction nécessite l'usage des ressources CPU. Lorsque le CPU entame une étape d'une instruction, les ressources attribuées aux autres étapes restent inutilisées ce qui montre clairement l'inefficacité de cette architecture.

Architecture pipeline :

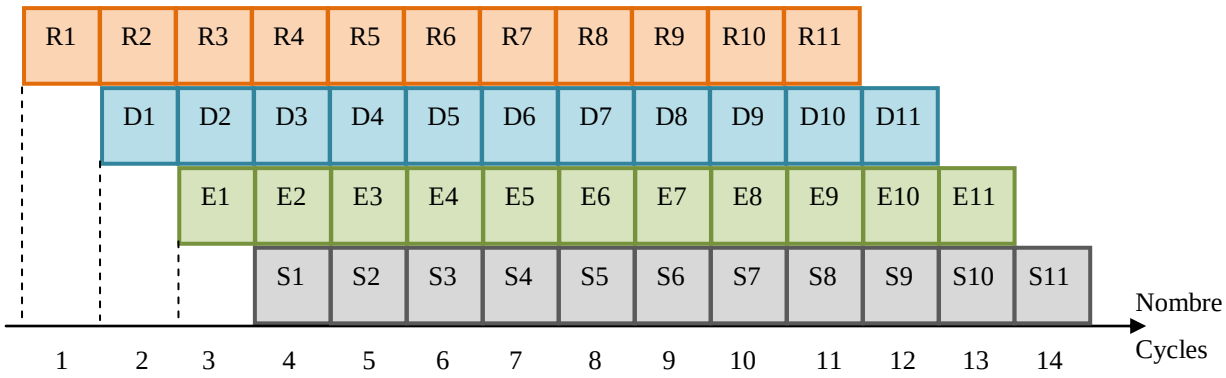


Figure 6.4 : *Exemple d'exécution dans une architecture pipeline.*

Comme le montre la figure 6.4, cette architecture exploite pleinement les ressources CPU à chaque étape des diverses instructions. En effet, lorsque l'exécution de la première étape

d'une instruction est achevée, le CPU commence l'exécution de la première étape de l'instruction suivante en même temps que la seconde étape de la première instruction et ainsi de suite jusqu'à la fin du programme. Une architecture pipeline est caractérisée par son nombre d'étages représentant le nombre d'étapes d'une instruction. Vis-à vis l'architecture simple (figure 6.3), cette technique améliore considérablement l'efficacité du microprocesseur. Le gain assuré par cette technique peut être estimé par:

.

$$Gain = \frac{Nbr * Cyl}{Nbr + Cyl - 1}$$

Où :

Nbr : Le nombre d'instructions exécuté.

Cyl : nombre de cycles d'horloge par instruction.

Exemples :

- Pentium 2 ➔ pipeline de 12 étages.
- Pentium 3 ➔ pipeline de 10 étages.
- Pentium 4 ➔ pipeline de 20 étages.
- L'Athlon d'AMD ➔ pipeline de 11 étages.

Problèmes liés à cette architecture

Les problèmes liés à cette architecture sont appelés aussi ***aléas***. En fait, ces problèmes gênent l'exécution normale des instructions du pipeline ce qui entrave l'obtention de la performance maximale. Les *aléas* les plus fréquents dans ce cas sont :

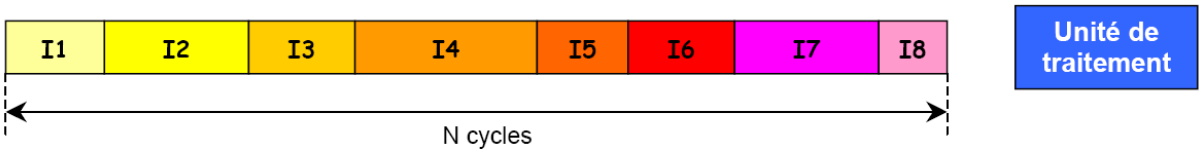
- **Aléa structurel :** Intervient dans le cas où deux instructions réclament la même ressource du processeur.
- **Aléa de données :** Intervient lorsqu'une instruction accède au résultat pas encore produit d'une instruction précédente
- **Aléa de contrôle :** Se produit dans le cas des instructions de branchement, l'exécution d'un saut est réalisée avec un certain retard.

6.4.3. Architecture super-scalaire

Cette technique consiste à exécuter plusieurs instructions en même temps (plusieurs instructions dans un cycle d'horloge). Ceci, est réalisé par l'implémentation de plusieurs

unités de traitement qui fonctionnent en parallèle. La figure 6.5 illustre la différence entre une architecture simple et super-scalaire [02].

Architecture scalaire :



Architecture super-scalaire :

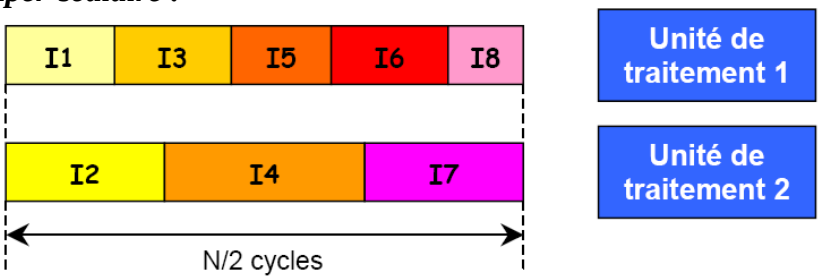


Figure 6.5 : *Utilisation de 2 unités de traitement pour une architecture super-scalaire.*

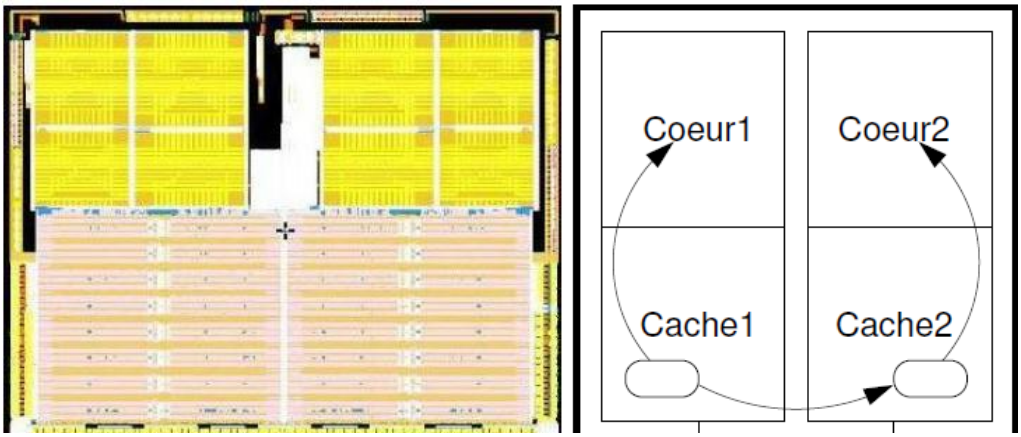
Exemple : c'est l'architecture utilisée dans le CPU Pentium d'Intel.

Remarque : Afin de maximiser les performances CPU, les deux techniques, pipeline et super-scalaire, sont combinées. Le principe dans ce cas est simple, il s'agit d'utiliser le pipeline dans toutes les unités de traitement implémentées.

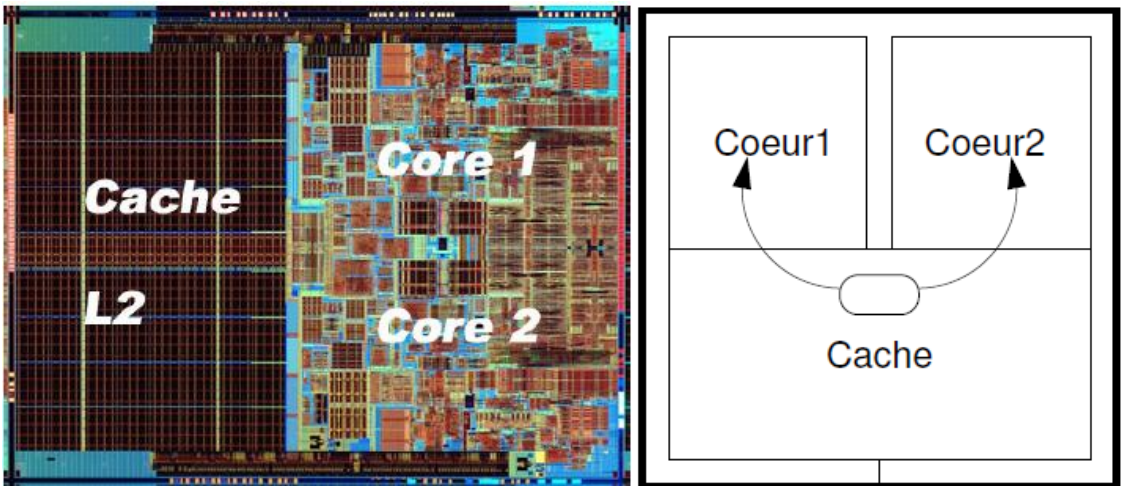
6.4.4 Les processeurs Multi-Cœurs

Un microprocesseur multi-cœur est un processeur avec plusieurs unités de calcul (cœurs physiques) fonctionnant en parallèle. Chaque cœur physique peut exécuter les programmes indépendamment des autres. En d'autres termes, ces cœurs physiques disposent de toutes les unités nécessaires à l'exécution d'un programme, à savoir, registres, UAL,…etc. Chaque cœur utilise soit sa propre mémoire cache soit une mémoire partagée entre les divers cœurs (suivant l'architecture adopté). La figure 6.6 présente quelques exemples des CPU multi-cœurs.

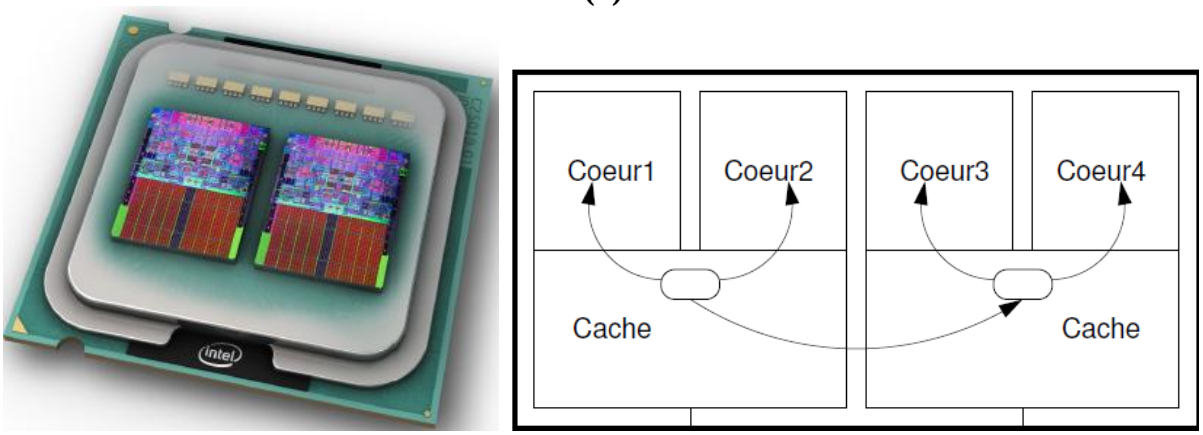
Cette nouvelle technologie a permis d'améliorer énormément les performances des CPU. L'un des avantages important de ces architectures modernes est la multiplication de la puissance de calcul sans l'augmentation de la fréquence d'horloge. Ceci permet donc d'éviter le problème d'élévation de température (chaleur dissipée).



(a)



(b)



(c)

Figure 6.6. *Exemples d'architecture multi-cœurs, (a) chaque cœur dispose d'une cache, (c) deux cœurs avec une seule cache partagée, (c) CPU à quatre cœurs d'Intel.*

Exemples :

- Power4 (IBM, 2001),

- Opterons (AMD, 2005)
- Ultra SPARC IV(Sun, 2003)
- PA-8800(HP, 2004)
- Montecito (Intel Itanium 2, 2006).
- Core i7 (Intel, depuis 2008).
- …,etc

Bibliographie

[01] Paul A. Carter, « Langage Assembleur PC », mars 2005

[02] Emmanuel Viennet, "Architecture des ordinateurs", GTR 1999-2000, IUT de Villetaneuse, Département GTR.

[03] Laurent Poinsot, 'architecture et système', UMR 7030 - Université Paris 13 - Institut Galilée.

[04] Alain Bron, 'théorie des systèmes des exploitation', version 2.8,

[05] http://www.gladir.com/LEXIQUE/INTR/INT21.HTM

[06] Max Mignotte, 'Introduction aux systèmes informatiques, le CPU et la mémoire II', DIRO IFT 1215, Département d'informatique et de recherche opérationelle.

[07] W. Stallings « Computer Organization and Architecture, designing for performance », Eighth Edition, Prentice Hall Upper Saddle River, NJ 07458, 2010.

Annexe

Le tableau suivant pressente une liste (non exhaustive) de quelques instructions en assembleur des processeurs 80x86, le code machine associé à chaque instruction assembleur est également donnée. Dans le tableau suivant la taille de chaque instruction est aussi présentée. Les valeurs *val* et les adresses *adr* sont données sur 16 bits. Code op signifie le code opération de l'instruction

Symbole	Code op	Octets	Symbole
MOV AX, *valeur*	B8	3	AX ← *valeur.*
MOV BX, *valeur*	BB	3	BX ← *valeur.*
MOV CX, *valeur*	B9	3	CX ← *valeur.*
MOV DX, *valeur*	BA	3	DX ← *valeur.*
MOV AX, *BX*	89D8	2	AX ← *BX.*
MOV AH, *val*	B4	2	AL ← *valeur.*
MOV DL, AL	88C2	2	DL ← *AL.*
MOV AX, [*adr*]	A1	3	AX ← Contenu de l'adresse *adr.*
MOV [*adr*], AX	A3	3	Range AX à l'adresse *adr.*
MOV [*adr*], BX	891E	4	Range BX à l'adresse *adr.*
ADD AX, *valeur*	05	3	AX ← AX + *valeur*
ADD AX, [*adr*]	03 06	4	AX ← AX + contenu de *adr*
ADD AX, CX	01C8	2	AX ← AX + CX
SUB AX, *valeur*	2D	3	AX ← AX - *valeur*
SUB AX, [*adr*]	2B 06	4	AX ← AX - contenu de *adr*
SUB BX, CX	29 CA	2	BX ← BX - CX
DIV BX	F7F3	2	AX ← AX/BX
SHR AX, 1	D1E8	2	Décale AX à droite
SHL AX, 1	D1E0	2	Décale AX à gauche
INC AX	40	1	AX ← AX + 1
INC BX	43	1	BX ← BX + 1
DEC AX	48	1	AX ← AX - 1
DEC BX	4B	1	BX ← BX - 1
CMP AX, *valeur*	3D	3	Compare AX et valeur
CMP AX, [*adr*]	3B06	4	Compare AX et contenu de *adr*
CMP BX, *valeur*	83	3	Compare BX et valeur
CMP BX, [*adr*]	3B1E	4	Compare BX et contenu de *adr*
JMP *adr*	EB	2	Saut inconditionnel (adr relatif).
JE *adr*	74	2	Saut si =
JNE *adr*	75	2	Saut si ≠

JG *adr*	7F	2	Saut si >
JLE *adr*	7E	2	Saut si ≤
JA *adr*			Saut si CF = 0
JB *adr*			Saut si CF = 1
MOV AH, 4C	B4 4C	2	