**République Algérienne Démocratique et Populaire**
**Ministère de L'Enseignement Supérieur et de la Recherche Scientifique**

**Université A.MIRA-BEJAIA**

جامعة بجاية
Tasdawit n Bgayet
Université de Béjaïa

**Faculté des sciences exactes**
**Département Informatique**

## Polycopié de cours

# ARCHITECTURE DES ORDINATEURS

### Cours avec séries TD et TP

**Dr. SAAD Narimane**

Le cours AO est destiné aux étudiants de 1ère année Ingénieur
Année universitaire
2024-2025 / 2025-2026

# Avant-propos

Le module « architectures des ordinateurs –AO» est un module destiné aux étudiants de première année Ingénieur Informatique. Il appartient à l'unité fondamentale UEF 1, il a le coefficient 4 et un crédit de 6.
Il est organisé sur 5 chapitres programmés sur 14 semaines. Le module nécessite une séance de cours + une séance de travaux dirigés + une séance de travaux pratiques par semaine.
Le module nécessite des prérequis (système d'entrés) qui sont des connaissances préliminaires et basiques des composants d'un ordinateur et du langage assembleur qui font sujet des modules système d'exploitation 1 et structures des machines 1 étudiés dans le premier semestre. Ces prérequis sont testés au départ afin de bien guider les étudiants et les orienter en cas de détection de lacunes de compréhension des notions de bases relatives à ce module.

Les chapitres sont expliqués en présentiel avec des résumés en ligne, les séances de cours sont accompagnées par des séances de travaux dirigés et de travaux pratiques, des tests, des devoirs afin d'assurer une bonne et équitable évaluation continue des étudiants, en plus d'une compréhension profonde par application des concepts.
Le module est clôturé par une évaluation sommative (examen final) en présentiel – système de sortie, contenant des exercices variés des 05 chapitres, afin d'assurer une évaluation globale correcte qui réfléchit le niveau de compréhension de tout le module.

**Matière :** Architecture des Ordinateurs
**Domain :** Informatique
**Filière :** 1ère année Ingénieur Informatique
**Volume Horaire du cours** : 1h30 *14 semaines
**Coefficient** : 4
**Crédits** : 6
**Evaluation** : Contrôle continu : 40%, Examen : 60%.

# Table des matières

# Liste des figures

# Abréviations

**ACC** Accumulateur
**AGP** Accelerated Graphics Port
**CAO** Conception Assistée par Ordinateurs
**CDROM** Compact Disc Read Only Memory
**CISC** Complex Instruction Set Computer
**CO** Compteur Ordinale
**CP** Compteur de Programme
**CPI** Cycles Par Instruction
**CPU** Central Processing Unit
**DDR SDRAM** Double Data Rate Synchrone Dynamic Random Access Memory
**DIMM** Dual In-Line Memory Module
**DRAM** Dynamic Random Access Memory
**DVDROM** Digital Versatile Disc Read Only Memory
**DVI** Digital Visual Interface
**EEPROM** Electically Erasable Programmable Read Only Memory
**EPROM** Erasable Programmable Read Only Memory
**E/S** Entrée / Sortie
**FIFO** First In First Out
**FPU** Floating Point Unit
**HDMI** High Definition Multimedia Interface
**IDE** Integrate Drive Electronique
**IEEE1394** Institute of Electrical and Electronics Engineers
**IR** Instruction Register
**ISA** Industry Standard Architecture
**I/O** Input / Output
**L1** Cache de niveau un
**L2** Cache de niveau deux
**L3** Cache de niveau trois
**LFU** Least Frequently Used
**LRU** Least Recently Used
**MAR** Memory Adress Register
**MBR** Memory Buffy Register
**MC** Mémoire Centrale
**MIPS** Millions d'Instructions Par Seconde
**MIPS** Microprocessor without Interlocked Pipeline Stages
**MP3** MPEG½ Audio Layer III
**PATA** Parallel Advanced Technology Attachement
**PC** Program Counter
**PCI** Peripheral Component Interconnect
**PROM** Programmable Read Only Memory
**PSW** Processor Status Word
**RAD** Registre d'Adresses
**RAM** Random Acces Memory
**RAM** Registre d'Adresse Mémoire
**RDO** Registre de Données
**RI** Registre d'Instruction
**RIM** Registre d'Information Mémoire

**RISC** Reduced Instruction Set Computer
**RJ45** Registered Jack 45
**ROM** Read Only Memory
**R/W** Read / Write
**SATA** Serial Advanced Technology Attachement
**SCSI** Small Computer System Interface
**SDRAM** Synchrone Dynamic Random Access Memory
**SIMD** Single Instruction Multiple Data
**SIMM** Single In-Line Memory Module
**SPDIF** Sony Philips Digital Interface
**SRAM** Static Random Access Memory
**UAL** Unité Arithmétique et Logique
**UC** Unité de Contrôle
**UCC** Unité de Contrôle et de Commande
**UE** Unité d'Exécution
**UIB** Unité d'Interfaçage avec le Bus
**USB** Universal Serial Bus
**UV-EPROM** Ultra Violet Erasable Programmable Read Only Memory
**VGA** Video Graphics Array
**VRAM** Video Random Acces Memory

# Introduction générale

L'architecture des ordinateurs combine la composition matérielle d'un ordinateur (hardware) et la couche logicielle (software). Les deux parties (software-hardware) sont entrelacées et affectent les uns des autres. L'évolution des ressources matérielles nécessite le développent d'une couche logicielle performante et efficace afin de bien exploiter les capacités des ordinateurs. Le but de ce module est d'étudier les architectures des ordinateurs logicielles et matérielles, de comprendre les composantes d'un ordinateur et leurs rôles ainsi que des manières de les commander et l'utiliser efficacement.
Le cours est scindé en un ensemble d'unités d'apprentissage qui vous permettent d'acquérir des compétences en matière de comprendre les composantes d'un ordinateur et leurs fonctionnements tel que les processeurs, les mémoires avec leurs différents types.
Ainsi que l'utilisation du langage de bas niveau qui est l'assembleur pour la réalisation des programmes rapides qui s'exécutent directement au niveau du processeur en utilisant directement ces registres. Aussi pour l'interaction directement avec les composantes matérielles des ordinateurs tel que : la mémoire, la carte graphique, la carte son, la carte réseau, les périphériques d'entrées/sorties.

Les 05 chapitres de ce module sont (système d'apprentissage) :

• Chapitre 01 : Organisation générale de l'unité centrale d'un ordinateur
• Chapitre 02 : Architecture interne des processeurs
• Chapitre 03 : Étude de cas : processeur Intel 8086
• Chapitre 04 : Étude de cas : Processeurs 32 et 64 bits
• Chapitre 05 : Architectures des processeurs récents

## ✓ Objectifs généraux du cours

Objectif du cours est de mettre en claire le principe de fonctionnement de l'ordinateur avec une présentation détaillée de son architecture.
Pour cela l'objectif du cours est :

- **Savoir,** Apprendre les notions de base sur les composants d'un ordinateur et leurs fonctionnements et l'interface des différents composants matériels d'un système informatique.

- **Savoir-faire,** Acquérir une connaissance de programmation en langage assembleur pour la conception des programmes qui manipule directement les composantes des ordinateurs tel que les registres, les mémoires et les périphériques d'entrées/ sorties.

- **Savoir-être,** Sensibiliser au rôle des concepteurs des processeurs et des architectures des ordinateurs.

# Chapitre 1 : Organisation générale de l'unité centrale d'un ordinateur

## 1. Généralités sur l'Ordinateur

***1.1. Le mot "information":*** Le mot information permet d'ajouter le sens aux données. L'information peut être : texte, chiffres, image, son, dessin, vidéo, etc…

***1.2. Définition "informatique":*** C'est en 1962 que **Philippe Dreyfus** employé le mot informatique pour définir le traitement automatique de l'information, il est composé des deux mots *information* et *automatique*.
Ce terme a été introduit en France, il est très répandu dans le monde à part dans les pays Anglo-saxons ou le terme dominant est computer. En général, l'informatique permet la manipulation, la gestion, l'organisation et le stockage de l'information. On peut citer parmi les avantages de l'informatique :

- ✓ Calculer avec une grande précision ;
- ✓ Gain en termes de temps ;
- ✓ Aider à prendre des décisions.

### *1.3. Historique des ordinateurs*

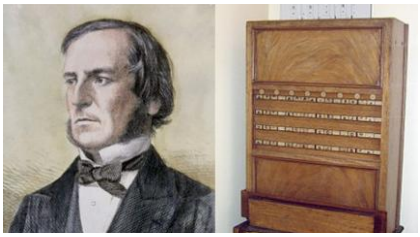
Un ordinateur (ou calculateur) est une machine "informatique" qui permet d'effectuer des calculs et des traitements ordonnés par un opérateur. L'ordinateur est né du travail de plusieurs ingénieurs et théoriciens pendant la Seconde Guerre mondiale. L'évolution de l'ordinateur actuel est passée par plusieurs inventions technologiques qui se classent en plusieurs générations.

- ➢ ***La génération zéro : les calculateurs mécaniques (Avant 1945)***

- ➢ ***Les abaques (avant 1600)***
  - Instruments mécaniques facilitant le calcul

- ➢ ***La Pascaline (1642) :***
  - Inventée par Blaise Pascal.
  - Première machine à calculer !


- ➢ ***Algèbre de Boole (1854)***
  - Algèbre binaire (booléenne), crée par George Boole,
  - Présente, avec les états électriques "inactif" et "actif", aura de nombreuses applications en informatique

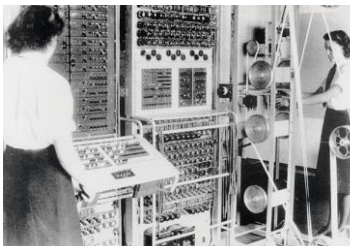


## ➢ *La première génération : le COLOSSUS, tubes à vide, ENIAC (1904 - 1945)*

### ➢ *Les tubes à vide (1904)*

- Également appelés tubes électroniques ou même lampes.
- C'est une source d'électrons.
- Remplacés plus tard par des semi-conducteurs

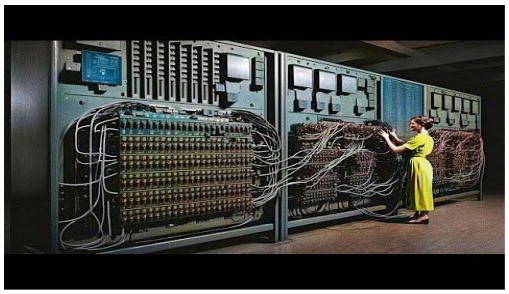
### ➢ *Le COLOSSUS (1941)*



- La Seconde Guerre Mondiale précipite l'avènement des ordinateurs.
- Les messages étaient chiffrés avec une machine ENIGMA,
- Besoin de trop de calculs, pour les décrypter → Création du 1er ordinateur électronique : le COLOSSUS.

.

### ➢ *L'ENIAC de John Mauchly (1945) :*

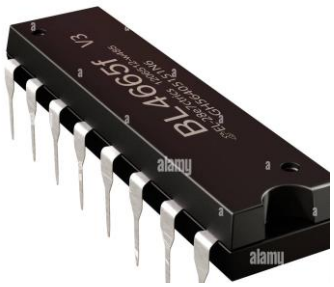


- **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer.
- 1er ordinateur électronique Turing-complet.
- Calculs en système décimal.
- Composé de 18 000 tubes à vide et 1500 relais.
- 6000 commutateurs et une forêt de câbles.
- Pèse 30 tonnes, et occupe 167 m2.
- Incapable d'enregistrer un programme.

## ➢ *La deuxième génération : les transistors (1955 - 1965)*

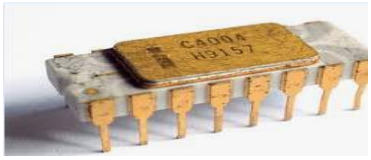
- Le transistor a été inventé en 1948
- En 1958: l'IBM 7000, 64 Koctets de mémoire, est le premier ordinateur intégrant des transistors
- Moins de consommation d'énergie électrique
- Moins d'espace occupé, la rapidité.
- 1960: Conception de ALGOL (ALGOrithmic Language), langage évolué de calcul scientifique

## ➢ *La troisième génération : les circuits intégrés (1965 - 1971)*



- Le circuit intégré a été inventé en 1958.
- Des dizaines de transistors sur une seule puce.
- L'intégration a permis des ordinateurs plus petits, plus rapides et moins chers.
- Apparition de la multiprogrammation (plusieurs programmes exécutés en même temps sur la même machine).
- Possibilité d'émulation d'anciens modèles.

## ➢ *La quatrième génération : les microprocesseurs (1971 - 1980)*



- Premier micro-processeur inventé par INTEL4004 en 1971.
- Intel 8008 (processeur 8 bits).
- Intel 8086 (16 bits).


• Apparition des ordinateurs personnels (PC) utilisés pour : Traitement de texte, Tableur
•En 1972 la conception du langage C, particulièrement adapté à la programmation et à l'utilisation de systèmes d'exploitation.
• Apparition d'Apple (1976)

➢ ***Cinquième génération: l'interface graphique et les réseaux***

• En 1984: Le McIntosh d'Apple introduit pour la première fois une interface graphique (menus, icônes...) et la souris, conception du langage C++, version orientée objet du langage C.

• En 1995: Windows 95 généralise l'interface graphique sur les PCs.

• Ordinateur de jeux : Jeux vidéo
• Ordinateur personnel (micro-ordinateur) : Ordinateurs portables ou de bureau
• Serveur : Serveurs de réseau
• Les japonais avaient annoncé pour les années 90 l'apparition d'un nouveau type d'ordinateurs «cinquième génération» dédiés à des applications d'Intelligence Artificielle.

- Jusqu'à nos jours……..

Donc, les premiers grands concepts datent depuis des siècles. Cependant cette science n'est développée que depuis un demi-siècle où l'architecture des ordinateurs a été le facteur clef de développement de l'informatique.

## 2. Concepts fondamentaux d'un ordinateur

### *2.1. Définitions*

**Un ordinateur (ou calculateur) est une machine capable de stocker et de traiter l'information d'une manière purement automatique. Il est capable de résoudre des problèmes en appliquant des instructions**

**L'architecture : On appelle architecture d'une machine informatique, l'apparence fonctionnelle présentée à l'utilisateur. Il s'agit d'une description de ce qui se passe du point de vue de l'utilisateur (le programmeur).**

**Instruction : Action à effectuer par l'ordinateur, correspondant à une étape dans un programme du système.**

**Programme : Ensemble d'instructions indiquant à un ordinateur un travail à effectuer.**

**Langage de programmation : Un programme écrit en langage évolué se trouve sous forme de code source. Pour pouvoir être exécuté par un ordinateur, les premiers programmes étaient écrits en langage machine (code binaire pur), puis en langage assembleur**

### *2.2. Les aspects de l'informatique*

L'informatique est représentée par deux aspects différents mais inséparables :

**2.2.1. Le HARDWARE :** Qui est composé de l'ensemble de matériels (l'ordinateur et ses périphériques).

**2.2.2. Le SOFTWARE :** Pour animer un ordinateur, il faut avoir des logiciels (programmes) qui trouvent place dans deux catégories :

- **Logiciel système** : qui est un ensemble de programmes chargés d'exploiter les ressources matérielles et de définir la façon dont ces derniers doit se comporter à l'égard des commandes introduites par un utilisateur. Exemples de logiciel système : le MS-DOS et Windows, UNIX etc...
- **Logiciel d'application :** qui est un ensemble de programmes destinés à réaliser des taches biens définies, tels que le traitement de texte Microsoft Word et le tableur Microsoft Excel etc…


**Figure 1.1. HARDWARE et SOFTWARE [20].**

- Un ordinateur (computer) est donc capable de :
  - ✓ Acquérir des informations ;
  - ✓ Conserver des informations ;
  - ✓ Effectuer des traitements sur des informations ;
  - ✓ Restituer des informations.

- Un ordinateur peut **recevoir** des **données** en **entrée** ⇨ « *fonction d'entrée* ».
  - ✓ **Stocker** ou **effectuer** sur ces données des **opérations** en fonction d'un **programme** «*fonction de traitement* ».
  - ✓ Et enfin **fournir** des **résultats** en **sortie** ⇨ « *fonction de sortie* ».


**Figure 1.2. Les entrées sorties d'un ordinateur**

### *2.3. L'***organisation** *de l'ordinateur :*

Concernant l'**organisation de base d'un ordinateur**, il doit posséder les unités fonctionnelles suivantes :

- **Unité de traitement** (*Processeur*) : cerveau de l'ordinateur, supervise les autres unités et effectue les traitements (exécution et calcule).
- **Unités de stockage** (*Mémoire*) : lieu de stockage des informations (programmes et données).
- **Unités d'entrées et de sorties** (*Périphériques*) : ce sont les unités qui sont destinées à recueillir les informations en entrée et à les restituer en sortie.
- **Bus de communication** assurent les connections entre les différentes unités.

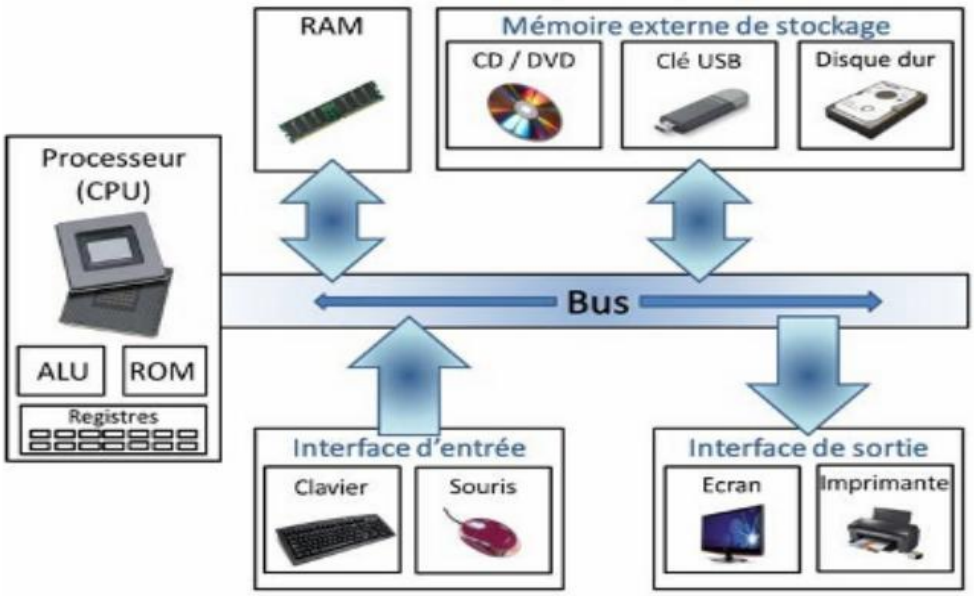



**Figure 1.3. Composants essentiels d'un ordinateur [13].**

### *2.4. Architecture de Base d'un ordinateur*

Pour comprendre le fonctionnement d'un ordinateur, il faut d'abord connaitre son architecture, les deux architectures informatiques les plus connues et utilisées dans l'ordinateur : **Von Neumann** et **Harvard.**

#### 2.4.1. Modèle de Von Neumann

Le modèle sur lequel sont basés les ordinateurs actuels est dû aux travaux de recherche de John Von Newman, publiés en 1946. Depuis ce temps, c'est l'architecture des ordinateurs modernes aussi largement utilisés, seules les technologies ont changé.
Dans l'architecture de **Von Neumann**, les programmes et les données sont stockés dans la même mémoire et gérés par le même sous-système de traitement de l'information.

Cette architecture est basée sur **4 parties** principales :

**1• Unité arithmétique et logique (ALU) :** Effectue les opérations de base.
**2• Unité de contrôle :** Chargée du séquençage des opérations. } **Processeur**

**3• Mémoire :** Contient les données et les programmes stockés
o *Mémoire vive*
*o Mémoire de masse*
**4• Entrées/Sorties :** permettent de communiquer avec le monde extérieur.

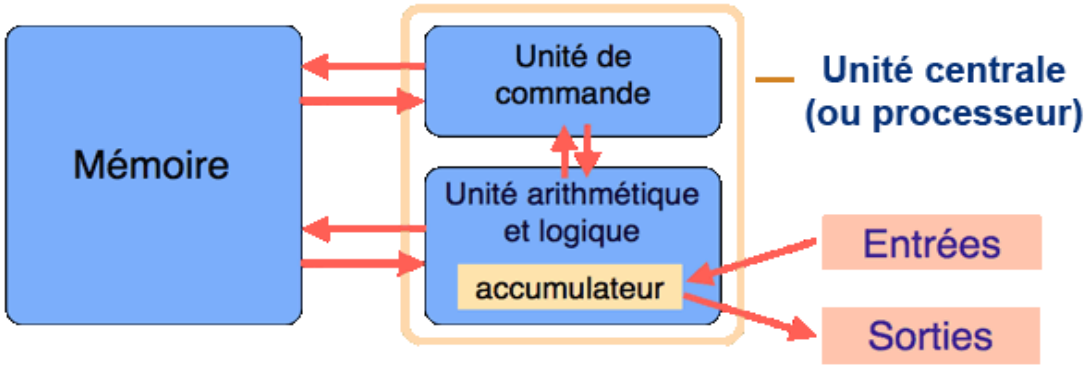



**Figure 1.4. L'architecture de Von Neumann [6].**

### 2.4.1. Modèle de Harvard

Le nom de cette architecture vient du nom de l'université Harvard où une telle architecture a été mise en pratique pour la première fois avec le Mark I en 1944.
Dans l'architecture de **Harvard** on sépare systématiquement la mémoire de programme de la mémoire des données, les échanges s'effectuent de manière double entre l'unité centrale et les deux mémoires, ce qui permet une grande souplesse pour l'enregistrement et l'utilisation des données. Les deux mémoires peuvent avoir des caractéristiques différentes : Taille du mot, Timing, Technologie, Structure de l'adresse, Accès simultanés aux données et aux instructions.

Architecture **Harvard** est utilisé dans le traitement du signal numérique spécialisé typiquement pour le traitement des données vidéo et audio.

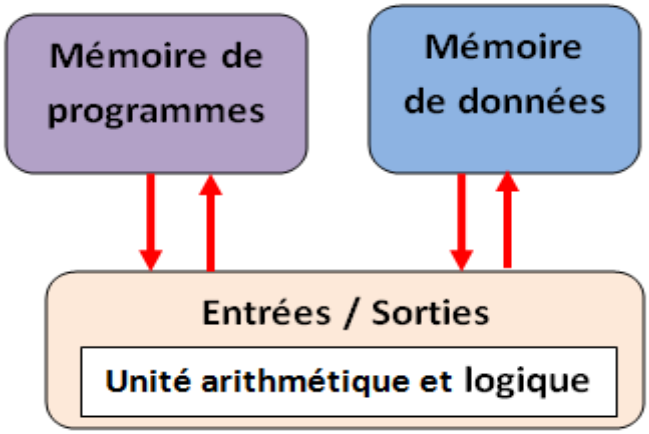



**Figure 1.5. L'architecture de Harvard [6].**

### *2.5. Les principaux composants d'un ordinateur*

Les composants matériels de l'ordinateur sont architecturés autour d'une carte principale comportant quelques circuits intégrés et beaucoup de composants électroniques tels que résistances, etc... Tous ces composants sont reliés par un grand nombre de connecteurs : cette carte est appelée « carte mère ».

#### 2.5.1. La carte mère « Mainboard » ou « Motherboard »

Elle permet à tous ses composants de fonctionner ensemble efficacement car elle assure la connexion physique des différents composants (processeur, mémoire, carte d'entrées/sorties, ...) par l'intermédiaire de différents bus (adresses, données et commande). La qualité de la carte mère est vitale puisque la performance de l'ordinateur dépend énormément d'elle. C'est sur cette carte que sont connectés les autres éléments :

- ✓ Le microprocesseur (cerveau de l'ordinateur);
- ✓ La mémoire (RAM : Random Access Memory, la mémoire cache);
- ✓ Le disque dur, le lecteur de CD-ROM, le lecteur de disquettes;
- ✓ Les périphériques internes : carte de son, carte vidéo.

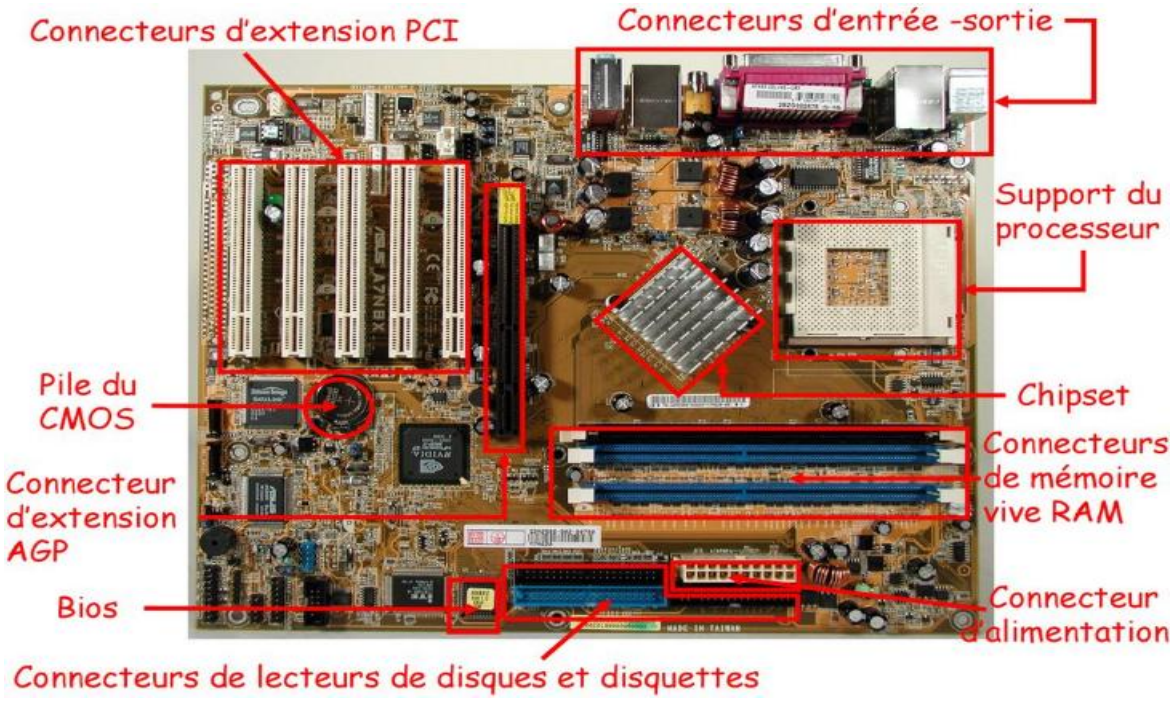



**Figure 1.6 : Le modèle de carte mère [18].**

❖ **Caractéristiques d'une carte mère**

Il existe plusieurs façons de caractériser une carte mère, notamment selon les caractéristiques suivantes :

**a. Le facteur d'encombrement *(ou facteur de forme, en anglais form factor)* :** on désigne les dimensions, l'agencement et les caractéristiques électriques de la carte mère. Il existe différents formats de cartes mères, comme par exemple : en 1995 ***ATX (****Advanced Technology eXtended)*, en 2005 ***BTX (****Balanced Technology eXtended,)*, *... etc.*

**b. Le chipset : *(traduisez jeu de composants ou jeu de circuits)* :** C'est une interface d'entrée/sortie. Elle est chargée de gérer la communication entre le microprocesseur et les périphériques. C'est le lien entre les différents bus de la carte mère.

**c. Le BIOS *(Basic Input Output System)* :** C'est un programme responsable de la gestion du matériel (clavier, écran, disque dur, liaisons séries et parallèles, etc..). Il est sauvegardé dans une mémoire morte (ROM) et agit comme une interface entre le système d'exploitation et le matériel.

**d. Le type de support :** On distingue deux catégories de supports :

- ✓ ***Sockets :*** un **socket** (en anglais) est le nom du connecteur destiné au processeur.
- ✓ ***Slots :*** un **slot** (en anglais) est une fente rectangulaire dans laquelle on insère un composant : une barrette de ***mémoire vive,*** pour enficher un ***processeur***,…

**e. Les ports de connexion :** Ils permettent de connecter des périphériques sur les différents bus de la carte mère. Il existe deux sortes de connecteurs (ou ports) :

- ✓ ***Les connecteurs internes :*** Il existe des connecteurs internes pour connecter des cartes d'extension (***PCI*** 'Peripheral Component Interconnect', ***SATA*** 'Serial ATA'…..etc).
- ✓ ***Les connecteurs externes*** (aussi appelé I/O Panel (Input/Output Panel) en anglais) : Il existe des connecteurs externes pour connecter d'autres périphériques externes à l'ordinateur : ***USB*** 'Universal Serial Bus', ***RJ45*** **'**Registered Jack**'**, ***VGA*** **'**Video Graphics Array**'**, ***DVI*** **'**Digital Visual Interface**'**, ***HDMI*** **'**High Definition Multimedia Interface**'**, …etc

### 2.5.2. Le microprocesseur

❖ **Définition et rôle d'un microprocesseur**

Un microprocesseur est un processeur (CPU « *Central Processing Unit* ») dont tous les constituants sont réunis sur la même puce (chip ou circuit intégré), afin de réduire les coûts de fabrication et d'augmenter la vitesse de traitement.

Le processeur (CPU) est le cerveau de l'ordinateur, Il permet les échanges de données entre les différents composants (disque dur, mémoire RAM, Carte graphique, …) et de manipuler des informations numériques, c'est-à-dire des informations codées sous forme binaire et d'exécuter les instructions stockées en mémoire. Les microprocesseurs utilisent des petits transistors pour faire des opérations de base, il y en a plusieurs millions sur un seul processeur.


**Figure 1.7 : Un processeur [4].**

❖ **Les principaux éléments d'un microprocesseur :** Sont

- ✓ **Une horloge** qui rythme le processeur. À chaque TOP d'horloge, le processeur effectue une instruction. Ainsi plus l'horloge a une fréquence élevée, plus le processeur effectue d'instructions par seconde

- ✓ **Une unité de gestion** des bus qui gère les flux d'informations entrant et sortant;
- ✓ **Une unité d'instruction** qui lit les données, les décode puis les envoie à l'unité d'exécution;
- ✓ **Une unité d'exécution** accomplit les tâches données par l'unité d'instruction

Le processeur travaille, en fait, grâce à un nombre très limité des fonctions comme des **expressions logiques (ET, OU, NON, etc.)**, des **expressions mathématiques (addition, soustraction, multiplication, etc.)**. Celles-ci sont directement câblées sur les circuits électroniques. Le processeur traite donc les informations compliquées à l'aide d'instructions simples.

❖ **Architecture interne**

Un microprocesseur est construit autour de deux éléments principaux :

- ✓ **Une unité de commande.**
- ✓ **Une unité de traitement**.

Associés à des registres chargées de stocker les différentes informations à traiter. Ces trois éléments sont reliés entre eux par des bus interne permettant les échanges d'informations

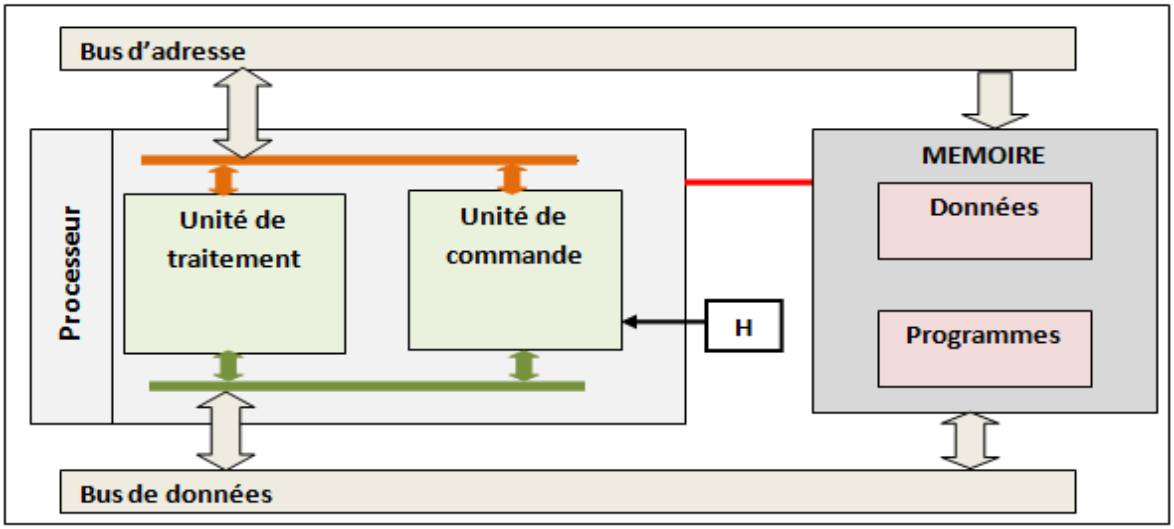



**Figure 1.8: Architecture interne d'un processeur [3].**

✓ ***L'unité de commande***

Elle permet de séquencer le déroulement des instructions. Elle effectue la recherche en mémoire de l'instruction. Comme chaque instruction est codée sous forme binaire, elle en assure le décodage pour enfin réaliser son exécution puis effectue la préparation de l'instruction suivante. Pour cela, elle est composée de :

- ➢ **Séquenceur** (ou ***bloc logique de commande***) chargé de synchroniser l'exécution des instructions au rythme d'une horloge. Il est ainsi chargé de l'envoi des signaux de commande.
- ➢ **Compteur ordinal :** Constitué par un registre contenant l'adresse de l'instruction en cours.
- ➢ **Registre d'instruction :** Constitué par un registre contenant l'instruction à exécuter.
- ➢ **Décodeur d'instruction** identifie l'instruction à exécuter qui se trouve dans le registre RI, puis d'indiquer au séquenceur la nature de cette instruction afin que ce dernier puisse déterminer la séquence des actions à réaliser.

✓ ***L'unité de traitement***

C'est le cœur du microprocesseur. Elle regroupe les circuits qui assurent les traitements nécessaires à l'exécution des instructions :

- **L'Unité Arithmétique et Logique (UAL)** est un circuit complexe constituée par un certain nombre de circuits tels que : complémenteur, additionneur, décaleur, portes logiques, et qui assure les fonctions :
  - Arithmétique (Addition, soustraction, multiplication et division « +, -, *, / »).
  - Logiques (ET, OU, Comparaison, Décalage, etc…)
  - Comparaison, décalage à droite ou à gauche, incrémentation, décrémentation, mise à 1 ou à 0 d'un bit, test de bit.
- **Le registre d'état** est généralement composé de 8 bits à considérer individuellement. Chacun de ces bits est un indicateur dont l'état dépend du résultat de la dernière opération effectuée par l'UAL. On les appelle indicateur d'état ou flag. On peut citer par exemple les indicateurs de : retenue (carry : C), signe (Sign : S)
- **Les accumulateurs** sont des registres de travail qui servent à stocker un opérande au début d'une opération arithmétique et le résultat à la fin de l'opération.
- **Les registres auxiliaires** permettent de stocker le résultat des instructions exécutées par l'ALU.

Les composants internes du microprocesseur sont relient par des ***bus de communication*** aux périphériques (mémoires et interfaces E/S)

### 2.5.3. La mémoire

❖ **Définition de mémoire**

On appelle « mémoire » tout composant électronique capable d'enregistrer, de conserver et de restituer des informations (stocker des données). On distingue ainsi deux grandes catégories de mémoires :

- ✓ **La mémoire centrale (Interne)**
- ✓ **La mémoire de masse (ou auxiliaire, externe) :**

❖ **Les principales caractéristiques d'une mémoire**

Les principales caractéristiques d'une mémoire sont les suivantes :

- **La capacité :** représentant le volume global d'informations que la mémoire peut stocker. Elle s'exprime en octet (8 bits).
- **Le format des données :** correspondant au nombre de bits que l'on peut mémoriser par case mémoire.
- **Le temps d'accès :** correspondant à l'intervalle de temps entre la demande de lecture/écriture et la disponibilité de la donnée.
- **Le temps de cycle :** représentant l'intervalle de temps minimum entre deux accès successifs.
- **Le débit :** définissant le volume d'information échangé par unité de temps, exprimé en bits par seconde.
- **La non volatilité :** caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement.

❖ **Les types de mémoire**

✓ **La mémoire centrale (vive)**

La mémoire vive est appelée RAM (Random Access Memory). C'est la mémoire principale du système. Elle permet de stocker de manière temporaire des données lors de l'exécution d'un programme. La mémoire vive est volatile : elle permet uniquement de stocker des données tant qu'elle est alimentée électriquement. Chaque fois que l'ordinateur est éteint, toutes les données présentes en mémoire sont immédiatement effacées.


**Figure 1.9: La mémoire vive (RAM) [4].**

✓ **La mémoire de masse**

C'est une mémoire périphérique de grande capacité utilisée pour le stockage permanent ou la sauvegarde des informations. Elle utilise pour cela des supports magnétiques (disque dur, ZIP) ou optiques (CDROM, DVD) et la ROM (Read Only Memory ou mémoire en lecture seule) est appelée **mémoire morte**, ou mémoire non volatile car elle ne s'efface pas lors de la mise hors tension du système.


**Figure 1.10: La mémoire morte (ROM) [4].**

On a aussi :

✓ **Les registres**

Ce sont les éléments de mémoire les plus rapides. Ils sont situés au niveau du processeur et servent au stockage des opérandes et des résultats intermédiaires.

✓ **La mémoire cache**

C'est une mémoire rapide de faible capacité destinée à accélérer l'accès à la mémoire centrale en stockant les données les plus utilisées.

### 2.5.4. Les Bus

❖ **Définition de Bus**

Un bus est un ensemble de fils (conducteurs électriques) qui permet l'échange et le transfert des informations binaires entre les éléments internes de l'ordinateur. Il y a plusieurs bus spécialisés en fonction des types de périphériques concernés et de la nature des informations transportées : ***adresses, commandes ou données***.

### ❖ Caractéristiques d'un bus

Un bus est caractérisé par :

- ✓ **Sa largeur :** un bus est caractérisé par le volume d'informations qui peuvent être envoyées en parallèle (exprimé en bits). Ainsi la **largeur** désigne le nombre de bits qu'un bus peut transmettre simultanément.
- ✓ **Sa vitesse :** est le nombre de paquets de données envoyés ou reçus par seconde. Elle est également définie par sa **fréquence** (exprimée en Hertz).
- ✓ **Son débit :** Le **débit** maximal du bus (ou le **taux de transfert** maximal) est la quantité de donnée qu'il peut transfert par unité de temps, en multipliant sa largeur par sa fréquence.

### ❖ Différents types de bus

Selon la nature de l'information à transporter, on retrouve trois types de bus d'information (figure 10) en parallèle dans un système de traitement programmé de l'information :

- ✓ **Bus de données** : c'est un bidirectionnel, il assure le transfert des informations (opérations et données) entre le microprocesseur et son environnement. Son nombre de lignes est égal à la capacité de traitement du microprocesseur.
- ✓ **Bus d'adresses (bus d'adressage ou mémoire) :** c'est un bus unidirectionnel, il permet la sélection des informations à traiter dans un espace mémoire (ou espace adressable). Il peut avoir $2^n$ emplacements, avec **n** = nombre de conducteurs du bus d'adresses.
- ✓ **Bus de commande (bus de contrôle)** : c'est un bidirectionnel, constitué par quelques conducteurs qui assurent la synchronisation des flux d'informations sur les bus donnés ou adresses.

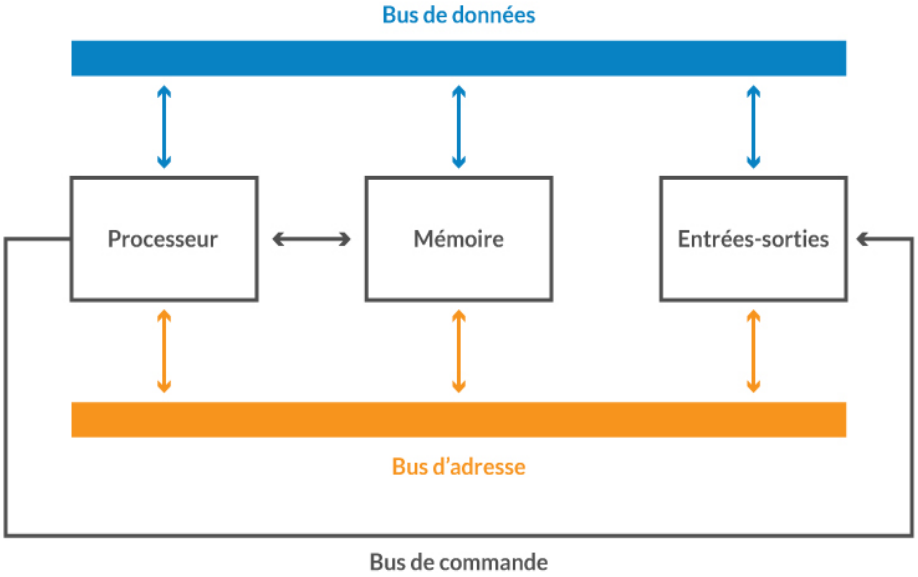



**Figure 1.11: Les différents types de bus [21].**

# Chapitre 2 : Architecture interne des processeurs

## Introduction

Le **processeur** est le véritable **cerveau** de l'ordinateur, est un composant électronique minuscule, en relation étroite avec la mémoire centrale. Le microprocesseur exécute les fonctions d'unité centrale « principale » d'ordinateur (CU), est considéré comme l'unité de traitement de l'ordinateur qui exécute les instructions du programme et les données à traiter. Dans ce chapitre, nous expliquerons son **rôle** ainsi que sa **puissance**, caractérisée par le nombre d'instructions qu'il est capable de traiter par seconde. Nous allons étudier la programmation des différentes conventions relatives à l'écriture des programmes en langage d'assemblage.

### 1. Processeur

Aussi appelé CPU (*Central Processing Unit*), c'est un circuit électronique cadencé au rythme d'une horloge interne, envoie des impulsions, appelées « **top** ». Toutes les avancées technologiques se concentrent sur ce composant, qui travaille toujours plus vite et effectue des opérations de plus en plus compliquées.

- Les principales caractéristiques d'un microprocesseur :
  - ✓ **Le format des mots de données** : 8 bits,16 bits, etc.
  - ✓ **La taille de l'espace adressable** : dépend du nombre de bits d'adresses (ex: 65536 emplacements pour 16 bits).
  - ✓ **La puissance de traitement** : s'exprime en CPI (Cycle Per Instruction) ou MIPS (Millions d'Instructions Par Seconde)
  - ✓ **Le jeu d'instructions** :
    Etendu (CISC)
    Réduit (RISC)

- Le microprocesseur exécute le programme, qui est une suite d'instructions. Une instruction est une opération SIMPLE sur un (ou plusieurs) mot(s) de données :
  - ✓ Lecture (LOAD) ou Ecriture (STORE) en mémoire
  - ✓ Opération logique (ET, OU, etc.)
  - ✓ Opération arithmétique (addition, soustraction, etc.)

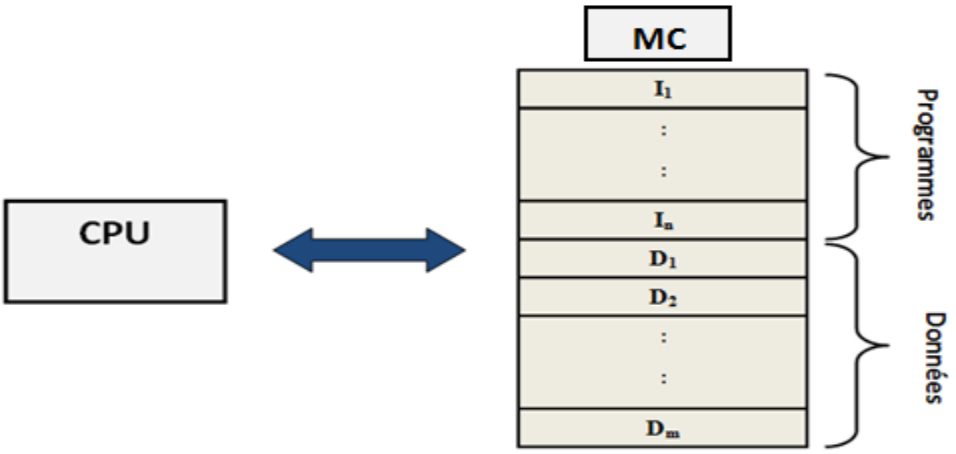



**Figure 2.1. Relation CPU et Mémoire Centrale [3].**

### *1.1. Rôle du processeur*

Le rôle du processeur est d'exécuter les instructions composant le programme. Il se charge de tous les **calculs mathématiques** et des **transferts de données internes et externes**. Il décide (en fonction bien sûr des instructions du programme en cours d'exécution) se passe à l'intérieur de l'ordinateur, car il permet de manipuler des informations numériques, c'est-à-dire des informations **codées sous forme binaire** et d'exécuter les instructions stockées en mémoire.

### *1.2. Les composants internes d'un processeur et leur fonction*

L'unité de traitement (CPU) est composée de 02 unités fonctionnelles, qui sont l'unité arithmétique et logique (UAL) et l'unité de contrôle (UCC). L'UAL s'occupe de toutes les **opérations arithmétiques et logiques** qu'effectue l'unité de traitement (CPU), alors que l'UCC **commande l'exécution** de toutes les opérations à tous les niveaux (L'UAL, mémoire et l'unité d'entrée /sortie), ainsi que **le contrôle** de leurs déroulement.

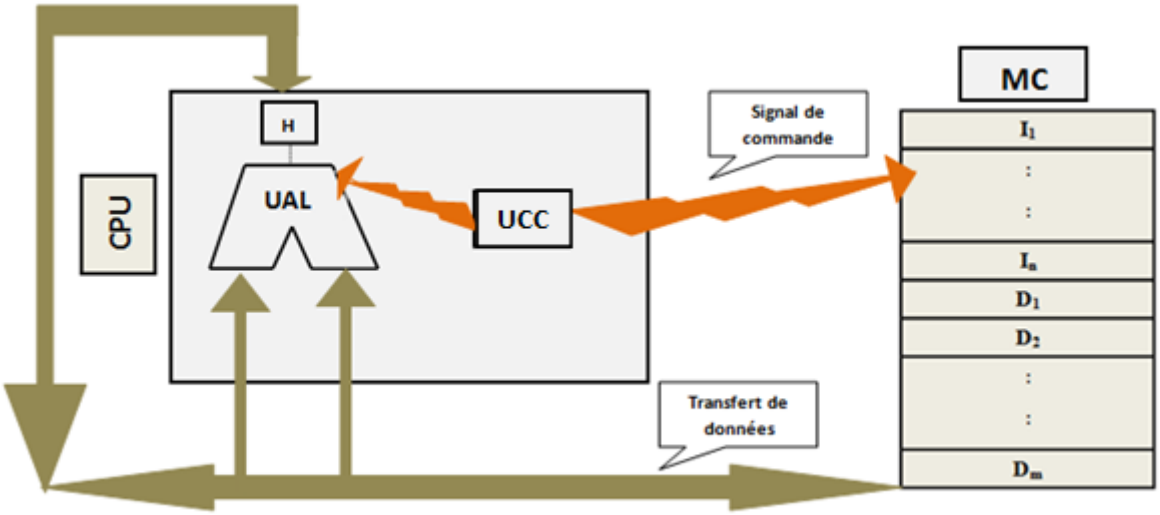



**Figure 2.2. Relation UAL et UCC [3].**

### *1.3.Fonctionnement de l'ensemble UAL et UCC*

On suppose qu'un programme quelconque est en cours d'exécution donc l'ensemble de ses instructions ainsi que les données sont en mémoire à des adresses bien définies.

➢ Le déroulement d'une instruction sera comme suit :

- ✓ **L'UCC cherche en MC une instruction en envoyant « l'adresse+la commande » au MC,**
- ✓ **L'instruction sera transférée vers l'UCC pour la décodée et déterminer l'opération,**
- ✓ **Des signaux générés et envoyés vers l'UAL pour déclencher l'exécution de l'opération,**
- ✓ **Les données nécessaires sont transférées de la MC vers l'UAL**
- ✓ **Après l'exécution de l'instruction le résultat est envoyé vers la MC.**

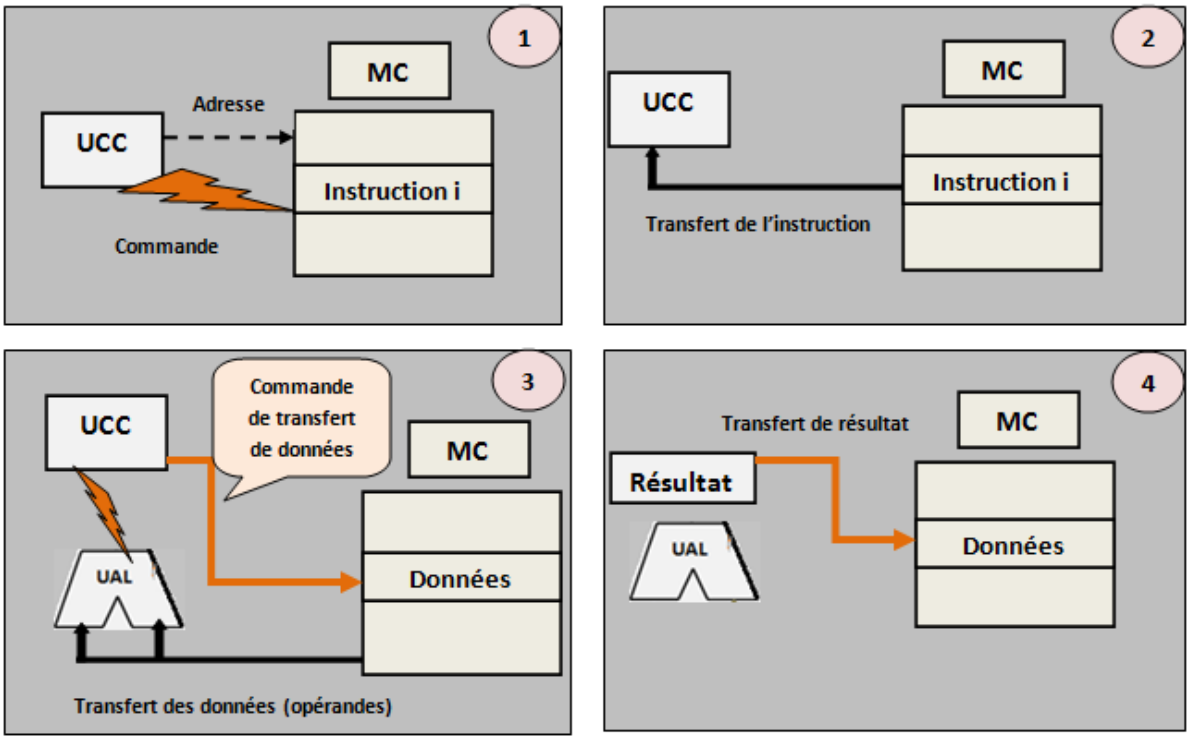



**Figure 2.3. Etapes d'exécution d'une instruction [3].**

## 2. Les Registres

Un registre est une cellule mémoire la plus petite ayant une fonction particulière, elle peut mémoriser des instructions, des adresses, des résultats intermédiaires, …etc. Un registre permet de mémoriser une information sur n bits.

Le nombre de registres diffère d'un ordinateur à un autre, cela dépend de son architecture, mais certains registres principaux existent dans la plupart des machines.

### *2.1. Caractéristiques des registres*

Un registre est une mémoire locale et privée du processeur, elle est caractérisée par :

- ✓ Mémorise une Information temporaire
- ✓ Capacité limitée (1 à 128bits)
- ✓ Les informations mémorisées peuvent être Visibles ou invisibles (accessibles par l'utilisateur ou uniquement par le processeur).

### *2.2. Les types de registres*

En général il existe trouve deux types de registres :

- ✓ **Le registre d'adresse :** qui contient l'adresse d'un mot-mémoire (pointeurs) connectés sur le bus adresses
- ✓ **Le registre mot :** qui contient le contenu d'un mot-mémoire, permettent de manipuler des données à vitesse élevée. Ils sont connectés au bus des données interne au microprocesseur.
- ✓ **Un registre de contrôle :** il modifie ou contrôle le comportement général d'un processeur ou d'un autre périphérique numérique.

> **Remarque :** *Un **registre mot** a la même **taille** qu'un **mot-mémoire**, alors qu'un **registre d'adresse** doit permettre **d'adresser tous les mots de la mémoire**.*

A partir de ces types de registres on va citer les registres principaux :

- **1. Compteur ordinal (CO):** Comme son nom l'indique, ce registre sert à compter les instructions contient l'adresse mémoire de la prochaine instruction à exécutée. Le CO est incrémenté automatiquement après chaque utilisation.
- **2. Le registre d'instruction (RI):** Ce registre contient l'instruction en cours d'exécution, la taille du RI est égale à la taille du mot mémoire.
- **3. L'accumulateur (ACC) :** L'accumulateur est un registre de l'UAL qui set essentiellement à contenir le résultat de l'opération effectuée comme il peut contenir aussi l'un des deux opérandes avant l'exécution de l'opération et recevoir le résultat après. La taille de l'ACC égale à la taille du mot mémoire. L'ACC peut être accédé par le programmeur.

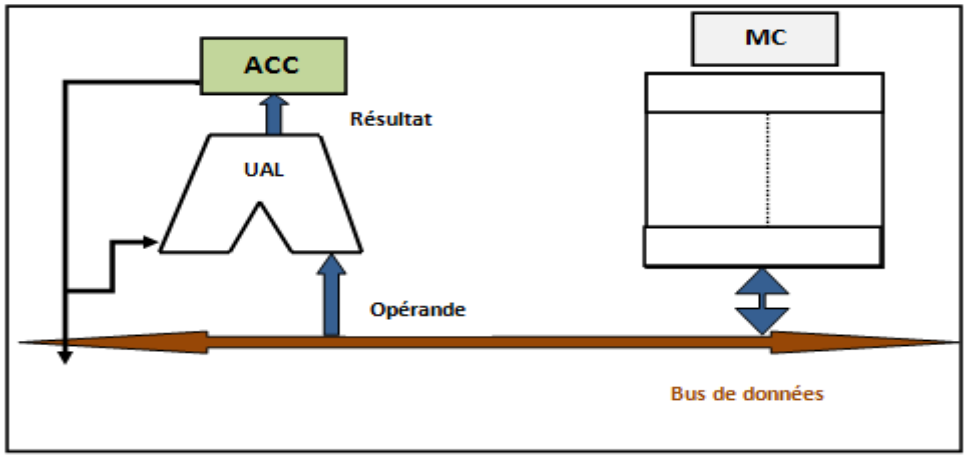



**Figure 2.4. Rôle de l'accumulateur [3].**

- **4. Les registres généraux (R0, …, Rn):** Registres de **stockage intermédiaire** pour **les calculs**, permettent de limiter l'accès à la mémoire en conservant **les informations les plus utilisées** par le processeur ; Exemple : MOV C,B: transfert contenu du B dans C.
- **5. Le registre d'index (ou d'indice):** Ces registres sont généralement destinés à la manipulation d'indices de tableaux, c'est-à-dire pour l'**adressage indexé**. L'incrémentation et la décrémentation de ces registres se fait automatiquement après chaque opération.
- **6. Registre de base :** Ce type de registre sert **à calculer l'adresse effective**, des instructions ou des données d'un programme dont les instructions ne sont pas placées en mémoire d'une façon contigüe. Le registre de base contient **une adresse de référence** laquelle est ajoutée à l'adresse contenue dans le champ adresse de l'instruction.
- **7. Le registre d'état : (PSW : Program Status Word) :** Il est appelé aussi **registre de condition** (pointer sur le début des zones mémoires), ce registre contient des bits indicateurs appelés drapeaux (**ou flags**) qui indique des états particuliers tels que : C
  - ***Signe de résultat (****N pour* ***Négatif, P*** *pour* ***Positif)***
  - ***Résultat nul (****symbolisé par* ***Z pour Zéro)***
  - ***Eventuelle retenue (****symbolisé par* ***C pour Carry)***
  - ***Dépassement de capacité (****symbolisé par* ***O pour Overflow).***

- **8. Le registre pointeur de pile (SP : Stack pointer) :** Contient **l'adresse du sommet de la pile** en mémoire centrale ce qui permet de gérer une **zone de mémoire** sous forme

**d'une pile. La lecture/ écriture se fait** selon l'adresse inscrite dans le registre pointeur d'un mot chargé dans la pile.

**Pour un traitement rapide, l'exécution d'une instruction par le processeur passe par ces étapes :**

1) ***Charger*** la prochaine ***instruction*** à exécuter dans le registre instruction (***RI***).
2) ***Modifier*** le compteur ordinal (**CO**) pour qu'il pointe sur ***l'instruction suivante***.
3) ***Décoder*** (analyser) ***l'instruction chargée***.
4) ***Localiser*** en mémoire d'éventuelles **données nécessaires** à l'instruction.
5) **Charger**, si nécessaire, les **données dans les registres généraux**.
6) **Exécuter l'instruction**.
7) **Stockage des résultats** à leurs destinations respectives.
8) **Recommencer** à l'étape 1.

**Remarque :** Les registres **Compteur ordinal (CO)**, Le **registre d'index** (ou d'indice), **registre de base** et Le **registre pointeur de pile** sont des **registres d'adresse**, alors que Le **registre d'instruction (RI)**, **L'accumulateur (ACC)**, les **registres généraux** sont des **registres de données** et le **registre d'état** est **un registre de contrôle.**

## 3. Unité Arithmétique et Logique (UAL)

### *3.1. Rôle et composants de l'UAL*

L'UAL réalise les **opérations arithmétiques** et **logiques,** doté de circuits logiques capables de réaliser ces opérations, nécessaires au fonctionnement de l'ordinateur. Elle possède les registres suivants :

- ✓ **L'accumulateur (ACC) :** réservé pour l'UAL stocke le résultat de l'opération
- ✓ **Les registres arithmétiques** : destinés pour les opérations arithmétiques (+, -, *, /) ou logiques (NOT, AND, OR, XOR),
- ✓ **Les registres de base et d'index** (calcul d'adresse par rapport à une base ou index).
- ✓ **Les registres généraux** (ex : stockage de résultats intermédiaires).
- ✓ **Le registre d'état PSW :** qui indique l'état du système.

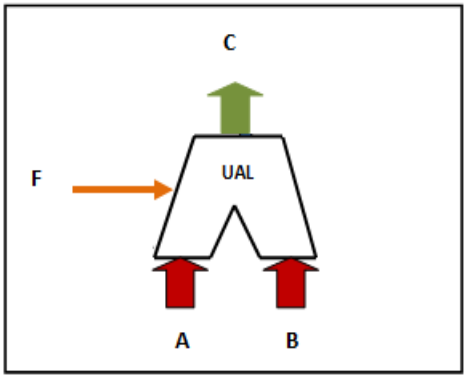
- **A et B sont les opérandes (les données en entrée) et C est le résultat de l'opération, F est la fonction à exécuter.**
- **Pour réaliser une UAL permet de faire les opérations logiques de base (ET, OU, NON) et la somme de deux nombres binaires ⇨ quelle est la composition de l'UAL ?**

⇩

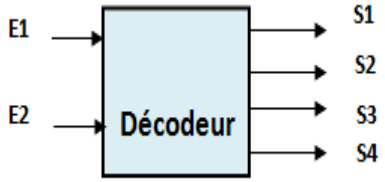
- **Cette UAL traite 04 opérations : A ou B, A et B, non B, A+B**
- **Ce circuit sélectionne l'opération à exécuter,**
- **Ce circuit doit avoir 2 entrées E1, E2**
- **Pour effectuer 4 choix : 00, 01, 10, 11**

⇩

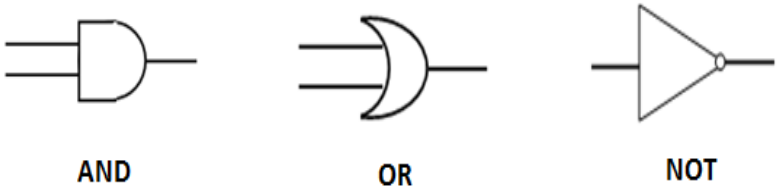
**Ce circuit (UAL) : Est un décodeur à 2bits**



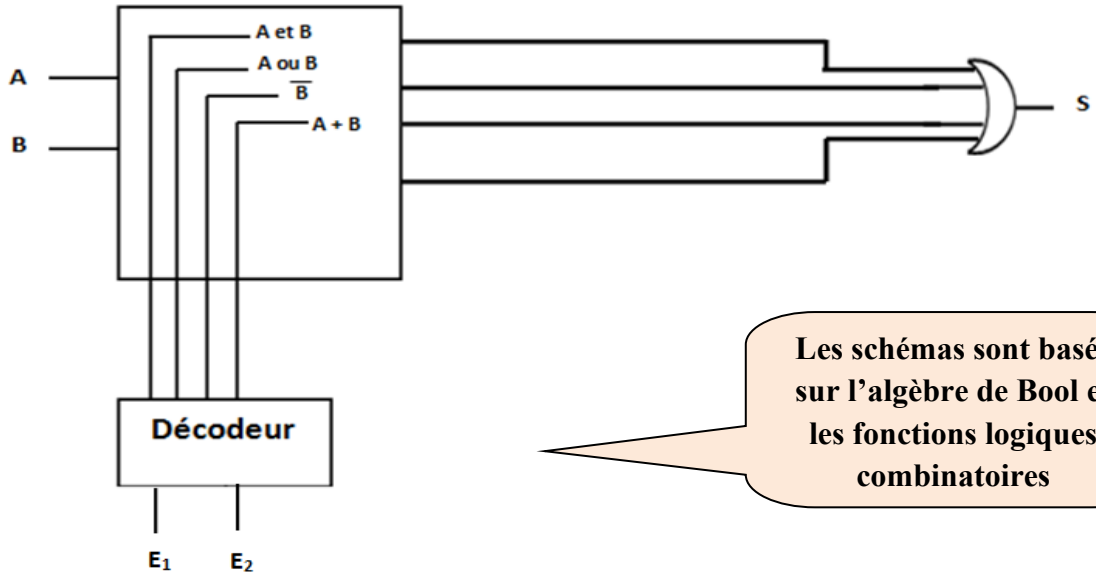
### *3.2. Présentation des composants de l'UAL (additionneur, comparateur, décodeur)*




Pour réaliser les 03 opérations logiques A et B, A ou B et Non B ⇨ on a besoin de 3 portes logiques

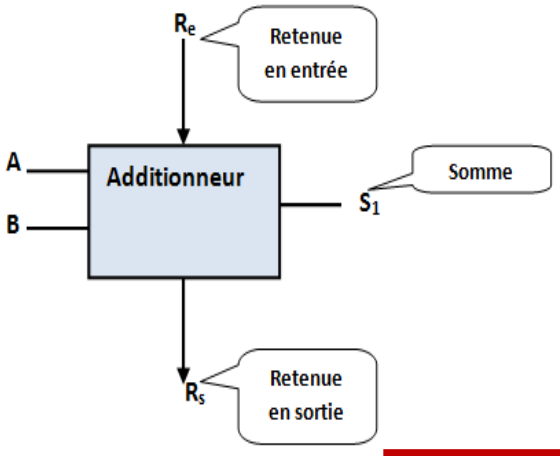


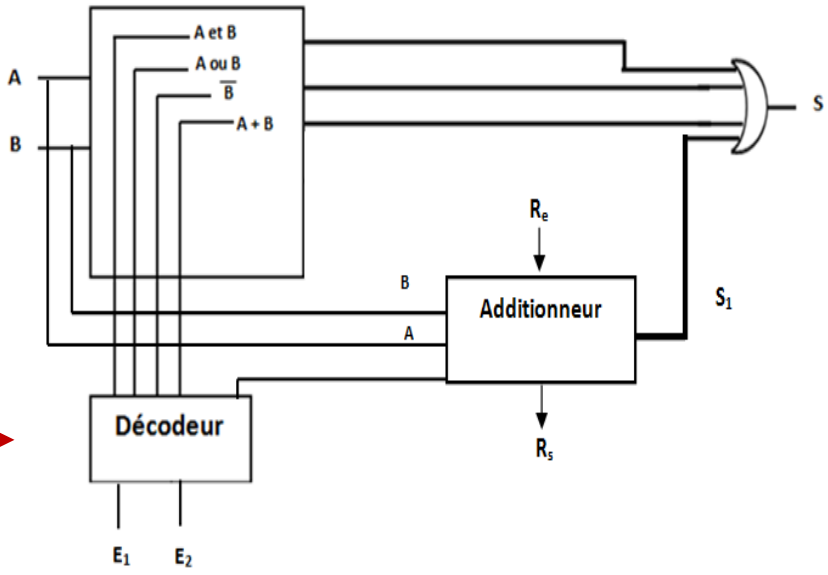

Le résultat d'une seule opération de l'UAL est transmis à un instant donné vers la sortie S.




**Les schémas sont basés sur l'algèbre de Bool et les fonctions logiques combinatoires**

**Le schéma de cette UAL sera comme suit:**



- ✓ Pour construire une UAL à **n bits**, il faut relier **n circuits** d'UAL à **1bit**
- ✓ Les **circuits qui réalisent les opérations arithmétiques et logiques** constitue **l'unité de calcul** de l'UAL, cette unité comporte **des registres** pour **mémoriser** les **données et le résultats** et **des bus** qui **transmettent** les données entre registres et l'unité de calcul

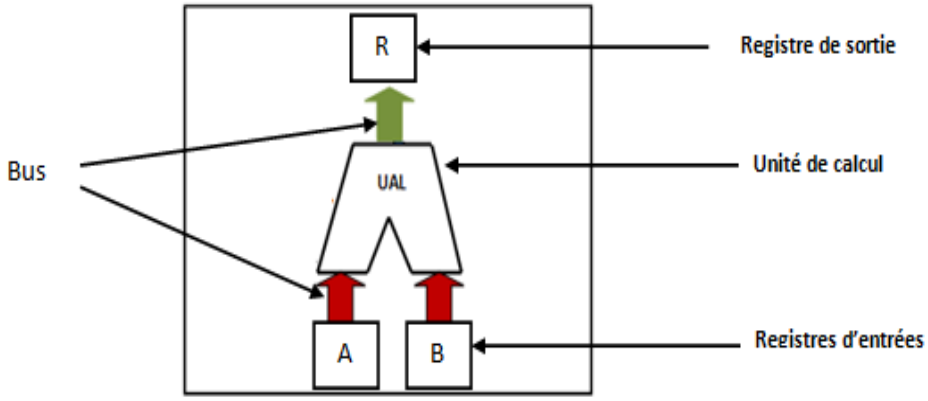



**Figure 2.5. Registres et Bus de l'UAL [3].**

## 4. Unité de commande et de contrôle (UCC)

### *4.1. Rôle de l'UCC*

Unité de commande accomplit deux fonctions distinctes ; commander et contrôler, elle dirige le fonctionnement de toutes les autres unités de l'ordinateur (UAL, Mémoire, Entrée/Sortie), pour le faire elle utilise des circuits pour décoder les instructions du programme et les transformer en signaux de commandes vers toutes ces unités.

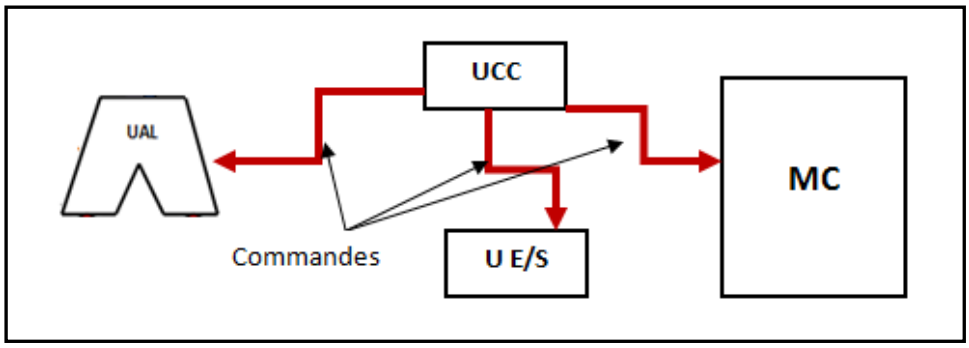



**Figure 2.6. Rôle de l'UCC [3].**

### *4.2. Les composants de l'UCC*

Elle comprend une mémoire très rapide qui lui permet de stocker des résultats temporaires ou des informations de commande. Cette mémoire est formée de quelques registres, chaque registre ayant une fonction particulière.

- **Le compteur ordinal (CO) :** Le registre le plus important est qui pointe sur la prochaine instruction à exécuter.
- **Le registre instruction (RI) :** qui contient l'instruction en cours d'exécution. La taille du **RI** est égale à la taille du mot mémoire.

- **Le décodeur :** Un circuit combinatoire qui détermine quelle opération doit être effectuée.
- **Le séquenceur :** Le séquenceur est un automate générant les signaux de commande nécessaires pour actionner et contrôler les unités participant à l'exécution d'une instruction donnée.

✓ Cet automate peut être réalisé de deux façons :

- Câblé : Circuit séquentiel
- Microprogramme : Suite de micro-instructions stockées dans une mémoire rapide (ROM ou EEPROM).

Dans un séquenceur câblé, à chaque instruction exécutable correspond un sous circuit capable de commander son déroulement.

- **L'horloge :** Une horloge est un système logique qui émet régulièrement une suite d'impulsions calibrées. L'intervalle de temps entre deux impulsions est appelé période ou temps de cycle. Le rôle de l'horloge dans l'UCC est de générer des signaux périodiques qui définissent le cycle de base (cycle machine) qui correspond à la durée élémentaire régissant le fonctionnement de la machine. De ce fait l'horloge synchronise toutes les actions du processeur.

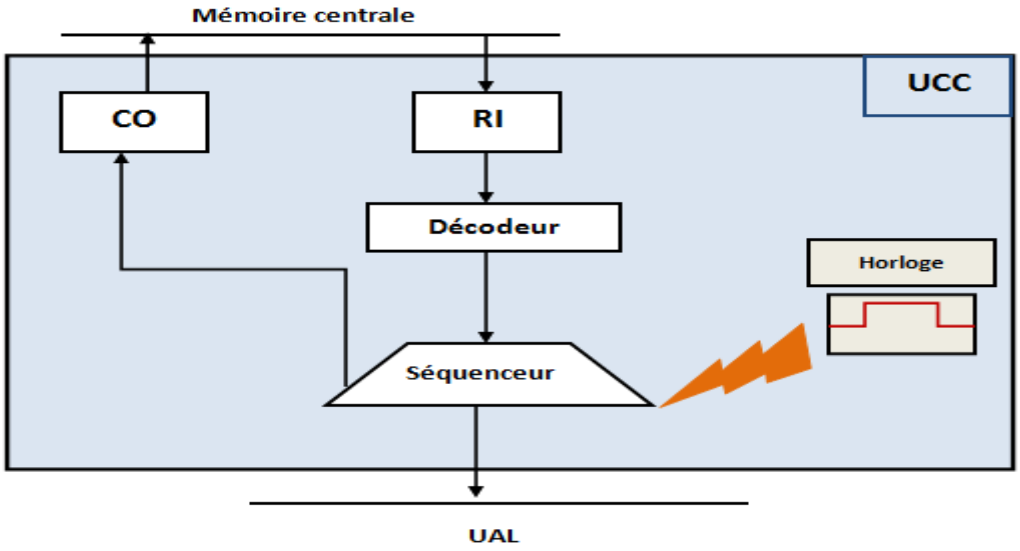



**Figure 2.7. Structure de l'UCC [3].**

### *4.3. Le fonctionnement de l'UCC*

L'instruction à exécuter comprend deux parties :

✓ La partie contenant le code opération, qui est essentielle pour spécifier le type de l'action demandé

✓ La partie contenant l'adresse de l'opération (ou plusieurs champs d'adresses)
**Instruction= code opération+ adresse de l'opérande**

✓ Les instructions sont placées en mémoire à des adresses séquentielles, sauf autre cas (branchement, saut,……), l'UCC cherchera ces instructions dans la mémoire.

L'exécution d'une instruction par l'UCC passe par les étapes suivantes :

1. Le compteur ordinale **(CO)** donne l'adresse de la nouvelle instruction à exécuter et l'envoie vers la mémoire centrale.

2. Une commande de lecture générée par l'UCC permet de lire l'instruction dans la mémoire MC.
3. Chargement de l'instruction à exécuter depuis la mémoire dans le registre instruction **(RI).**
4. Modification du compteur ordinal pour qu'il pointe sur l'instruction suivante.
5. Décodage de l'instruction que l'on vient de charger.
6. Localisation dans la mémoire des éventuelles données utilisées par l'instruction.
7. Chargement des données, si nécessaire, dans les registres internes.
8. Le séquenceur envoie un signal de commande vers l'UAL pour l'exécution de l'instruction.
9. Pour le stockage des résultats à leurs destinations respectives, le séquenceur envoie un signal de commande d'écriture en mémoire, le résultat est transféré de l'UAL vers la mémoire.
10. Retour à l'étape 1 pour exécuter l'instruction suivante.

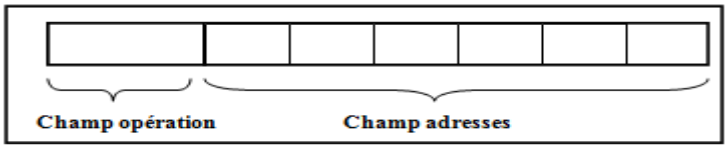
## 5. Jeu d'instruction

### *5.1. Structure d'une instruction machine*

Un programme est une suite d'instructions écrites dans un langage de programmation, ces instructions sont ensuite introduites en langage machine pour qu'elles soient traitées par la machine avec le microprocesseur, pour cela on l'appelle **instruction machine.**

Une instruction machine comporte un ou plusieurs champs d'informations qu'elle doit fournir au processeur, ces informations concernent les éléments suivants :

- ✓ Un champ **code opération** (champ obligatoire),
- ✓ Un ou plusieurs champs **d'adresses (adresse de l'opérande, adresse du résultat, adresse de l'instruction suivante)**




Instruction machine

***Remarque : Le format d'une instruction diffère d'une machine à une autre et a évolué depuis la première génération jusqu'à aujourd'hui***

### *5.2. Les différents formats d'instructions machine :*

- ➢ Dans la plupart des ordinateurs actuels, on utilise des instructions à **une seule adresse ;** en effet on a plus besoin d'un **champ adresse** de **l'instruction suivante** car le **compteur ordinale (CO)** joue ce rôle, quant au **champ adresse du résultat**, il est remplacé par un registre spécial de l'**UAL** appelé **accumulateur**

- ➢ On a aussi la machine dont l'instruction machine ne comporte pas d'adresse (**machine à zéro adresse**), cette machine utilise une mémoire **LIFO (Last In First Out)** appelée aussi **Pile.**

- Les formats d'instructions possibles liées à l'évolution des générations d'ordinateurs, cité en dessus : *(n =0, ……4) adresses pour indiquer le format de l'instruction machine*
  - ✓ ***Machine à 4 adresses : Utilisé dans les premiers ordinateurs***
    Elle comporte : **Un champ pour code opération + quatre (04) champs d'adresses :** « Adresse du 1er opérande, Adresse du 2eme opérande, Adresse du résultat, Adresse de l'instruction suivante »
    Exemple : 30+15

- L'adresse de l'instruction addition est 185,
- L'adresse première opérande (30) est 174, L'adresse deuxième opérande (15) est 177
- L'adresse de l'instruction suivante est 230

**Le contenu du registre instruction RI :**

| ADD | 174 | 177 | 185 | 230 |
|---|---|---|---|---|

Le contenue de la Mémoire centrale sera :

| | |
|---|---|
| | ... |
| 165 | ADD 174 177 185 230 |
| | ... |
| 174 | 30 |
| | .... |
| 177 | 15 |
| | ... |
| 185 | 45 |
| | ... |
| 230 | L'instruction suivante |

- ✓ ***Machine à 3 adresses :***
  Le **champ de l'instruction suivante est supprimé** ; car le **compteur ordinal (CO)** qui est chargé de calculer cette adresse

**Le contenu du registre instruction RI :**

| ADD | 174 | 177 | 185 |
|---|---|---|---|

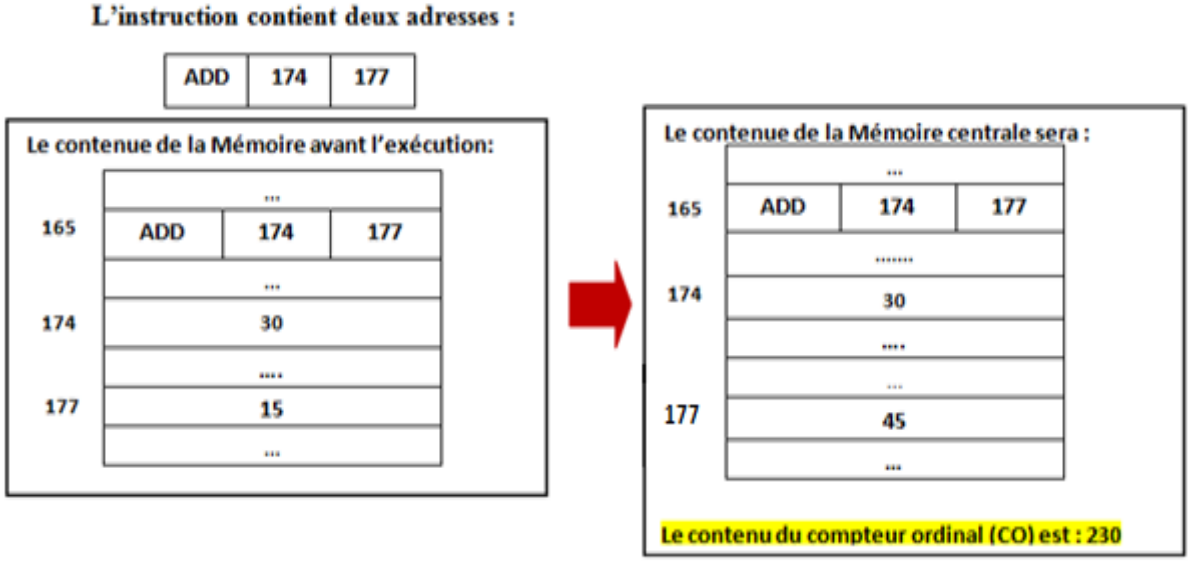
Le contenue de la Mémoire centrale sera :

| | |
|---|---|
| | ... |
| 165 | ADD 174 177 185 |
| | ..... |
| 174 | 30 |
| | .... |
| 177 | 15 |
| | ... |
| 185 | 45 |
| | ... |

Le contenu du compteur ordinal (CO) est : 230

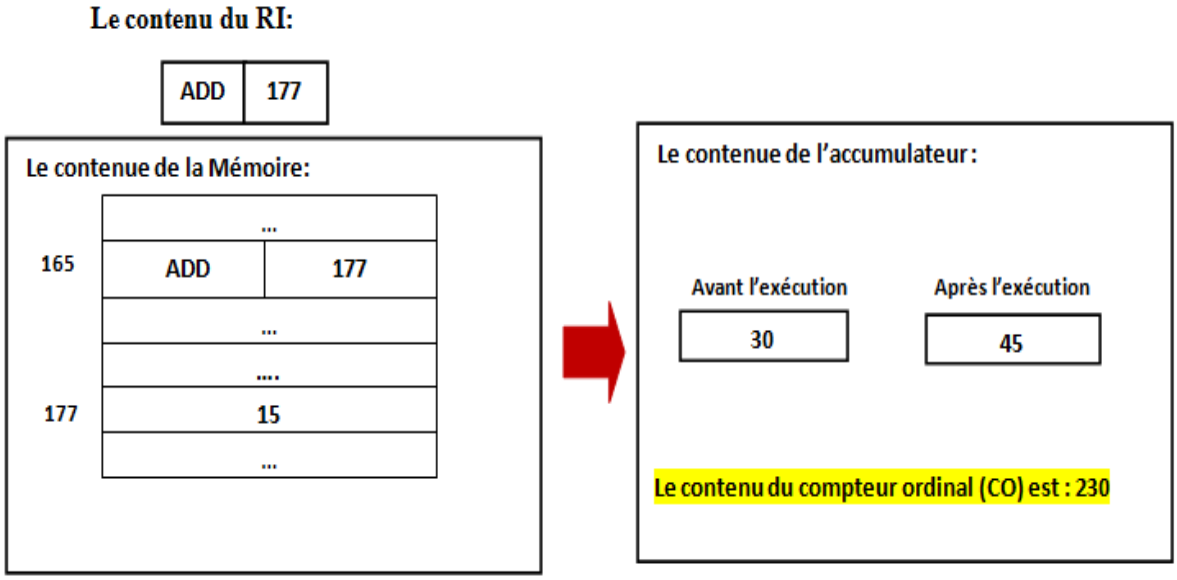
✓ ***Machine à 2 adresses :***

Le champ, **adresse du résultat** a été supprimée, l'adresse du résultat sera rangée **à la place du deuxième opérande**. (L'adresse ne contient que deux adresses)

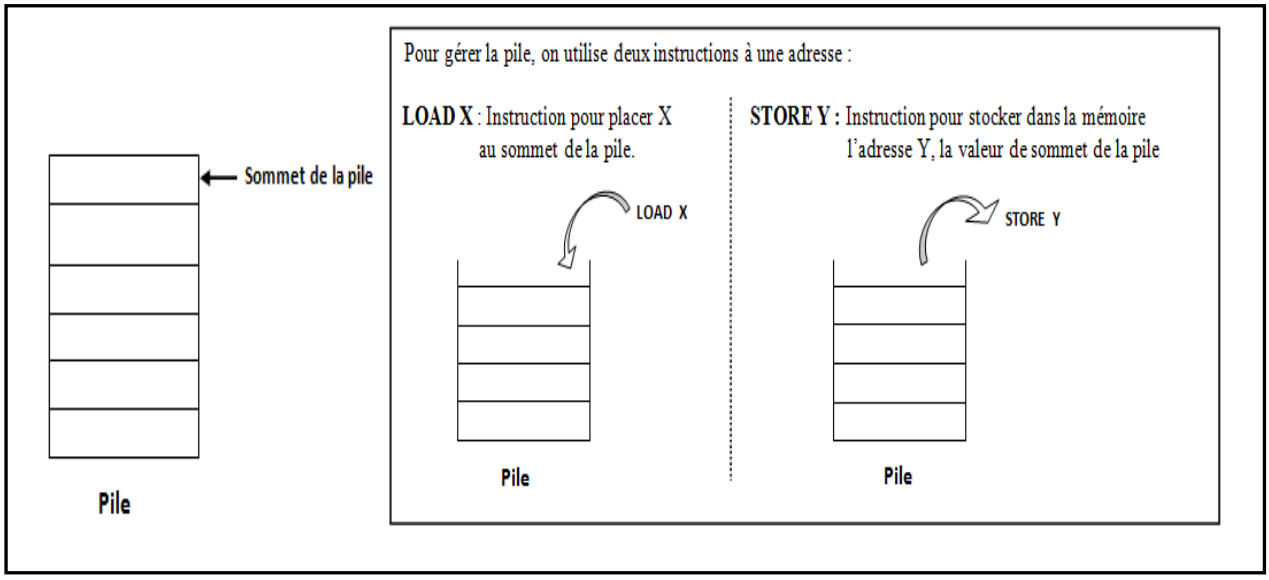



✓ ***Machine à 1 adresse***

Dans ce cas le format de l'instruction ne contient qu'une seule adresse, c'est l'adresse de premier opérande, le **deuxième opérande** se trouve **dans l'accumulateur**, donc pas besoin de son adresse.



✓ ***Machine à zéro adresse (ou à pile)***
Avant d'expliquer le mécanisme de cette machine, on va présenter le fonctionnement du pile.




- Les codes d'instructions : « **ADD, Load, Store** » appartiennent au **langage Assembleur**.
- Le tableau suivant présente quelques instructions du **machine à zéro et à une adresse** :

| Instruction | Description |
|---|---|
| ADD | Addition |
| MULL | Multiplication |
| DIV | Division |
| SUB | Soustraction |
| STA | Rangement |
| LOAD | Chargement |
| STORE | Stockage |

**Exemple :** Soit à exécuter l'expression suivante: **A=X+Y-Z** **ou : X=6, Y=5, Z=1**
La suite d'instruction correspondante à l'évaluation de cette expression est la suivante :
**LOAD X , LOAD Y, ADD, LOAD Z, SUB, STORE A**

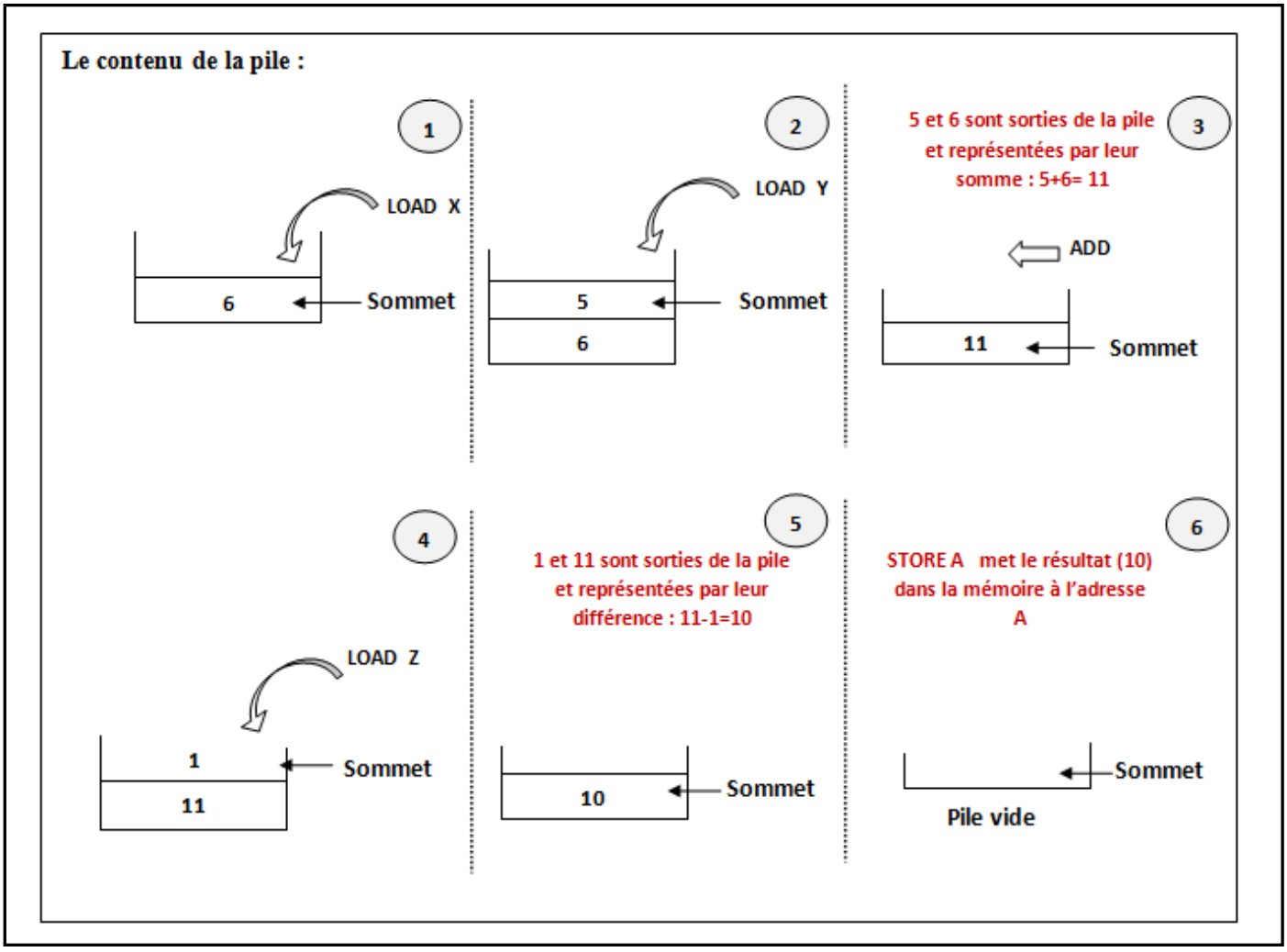



**Figure 2.8. Suite d'instructions m*achine* à zéro adresse [3].**

### *5.3. Architecture du jeu d'instructions :*

On appelle jeu d'instructions, un ensemble d'instructions de base qu'un processeur utilise pour exécuter ses traitements, on déduit que chaque machine possède son propre jeu d'instruction. Il existe deux architectures du jeu d'instructions des machines :

- ✓ **Architecture RISC (Reduced Instruction Set Computer) :** qui signifie ***processeur à nombre d'instructions réduit.***
- ✓ **Architecture CISC (Complexe Instruction Set Computer) :** qui signifie ***processeur à jeu d'instructions complexes.***

| Architecture CISC | Architecture RISC |
|---|---|
| - Jeu d'instructions large<br>- Instructions complexes<br>- Instructions de tailles différentes<br>- Instructions de durées différentes<br>- Séquenceur micro-programmé<br>- **Richesse du langage machine** | - Jeu d'instructions réduit<br>- Instructions simples<br>- Instructions de mêmes tailles<br>- Instructions de mêmes durées<br>- Séquenceur câblé<br>- **Efficacité du langage machine** |

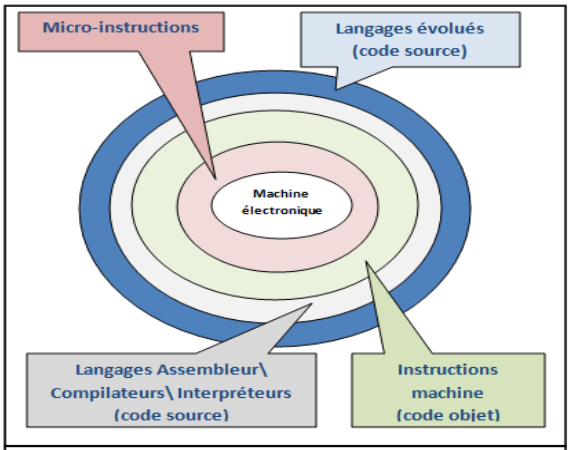
Avec l'évolution de la technologie, les jeux d'instructions des différents processeurs ont commencé à se ressembler, à présent presque tous les processeurs proposent des jeux d'instructions très similaires. Les instructions qu'on retrouve dans toute machine peuvent être classées comme suit :

- **Les instructions de transfert des données :**
  - **Entre la mémoire et les registres (ex : Load, Store)**
  - **Entre registre et registre (ex : Move)**
- **Les instructions de traitement des données :**
  - **Opérations arithmétiques (ex: ADD, SUB, MUL, DIV,….)**
  - **Opérations logiques (ex : AND, OR, NOT, XOR, ……)**
- **Les instructions de contrôle de l'exécution :**
  - **Branchements conditionnels**
  - **Branchements inconditionnels ou sauts**
  - **Appel de procédure**
- **Les instructions d'entrées / sorties.**

### *5.4. Architecture du jeu d'instructions :*

Le modèle de programmation d'un processeur définit la manière dont les instructions accèdent à leurs opérandes et leur façon d'être décrites dans le langage assembleur du processeur. Sur la plupart des machines, on retrouve deux modèles de programmation utilisées sur deux types de processeurs :

- ✓ **Les architectures à registres généraux**
- ✓ **Les architectures à Pile**



Les niveaux de programmation

#### 5.4.1. Architecture à registres généraux

Dans ce type d'architecture, les instructions lisent leurs opérandes dans des registres et y écrivent leurs résultats. Cet ensemble de registres est appelé fichier de registres généraux.

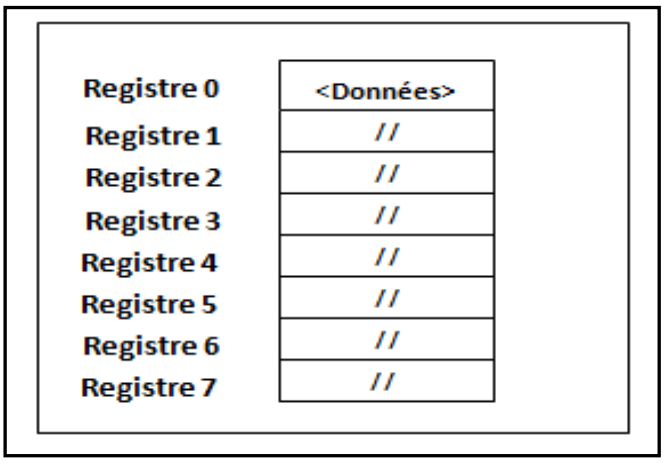
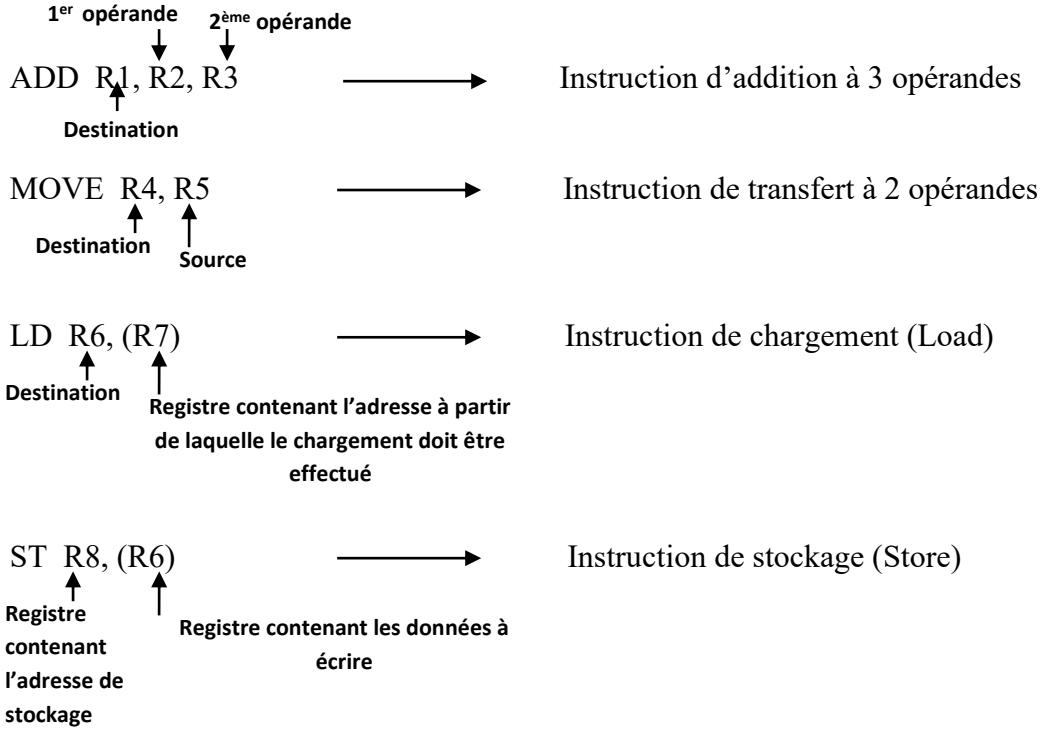


**Figure 2.9. Fichier de registres généraux [3].**

L'accès au registre est aléatoire, chaque registre possède un identificateur. Les instructions doivent spécifier les registres qui contiennent leurs opérandes d'entrée et celui dans lequel leur résultat doit être écrit. On utilise en général le format d'instruction à 3entrées.
Exemples d'instructions :




**L'entrée la plus à gauche est réservé au registre de destination pour mettre le résultat**

**Exemple:** On veut écrire un programme pour un processeur à registres généraux qui réalise le calcul suivant : **(a+b) x c** avec **a, b et c des constantes**. Le programme sera comme suit :

```
MOVE   R1, # a
MOVE   R2, # b        # X: signifie que X est une constante
ADD    R3, R1, R2
MOVE   R4, #c
MUL    R5, R3, R4
```

- Les deux premières instructions **MOVE** placent les constantes **a** et **b** dans les registres **R1 et R2** respectivement, l'instruction **ADD** additionne le contenu des registres **R1** et **R2**et met **le résultat** dans **R3.** La quatrième instruction met la constante **c** dans le registre **R4**, l'instruction **MUL** multiple le contenu de **R3** et **R4** et met le **résultat** dans **R5.**

***- Le programme précédent peut s'écrire comme suit :***

```
MOVE   R1, # a
MOVE   R2, # b        On a utilisé 3 registres au lieu 5 registres.
ADD    R2, R1, R2
MOVE   R1, #c
MUL    R3, R2, R1
```

### 5.4.2. Architecture à Pile

Dans ce type d'architecture, les instructions lisent leurs opérandes et écrivent leurs résultats dans une pile (**LIFO** : **Last In First Out**). Comme on l'a déjà vu avec « instruction à zéro adresse » ; une pile est une structure de données qui obéit au principe « le dernier qui entre est le premier qui sort (PEPS) »

- ✓ **Avec le jeu d'instructions *à zéro adresse* on a cité deux instructions *(LOAD, STORE)* qui permettent de *gérer la pile et d'accéder à la mémoire*.**
- ✓ **L'architecture à registres généraux : L'instruction de chargement (LOAD) et de stockage (STORE) sont *les seules qui peuvent accéder à la mémoire*.**

L'architecture Pile utilise deux instructions de base, **PUSH** et **POP**

- ✓ **L'opération PUSH :** permet de placer une donnée au sommet de la pile en poussant les données précédentes vers le bas.
- ✓ **L'opération POP:** permet de retirer la valeur du haut de la pile pour être utilisée comme entrée d'une instruction.

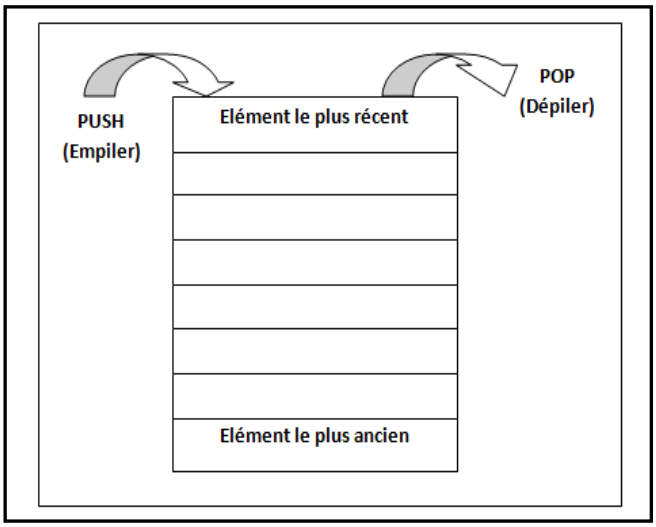
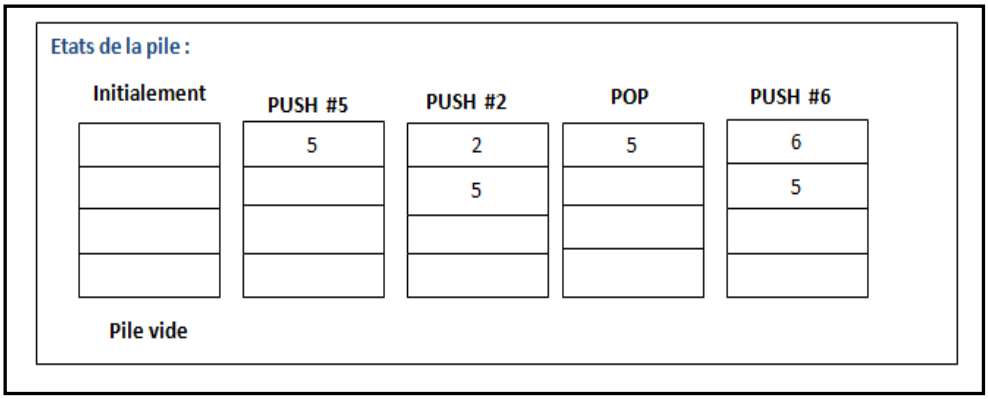



**Figure 2.10. Schéma d'une pile [3].**

**Exemple1 :** On considère le jeu d'instruction suivant :

```
PUSH  #5
PUSH  #2
POP
PUSH #6
```



## 6. Mode d'adressage

Le mode d'adressage nous renseigne sur la manière dont l'instruction, adresse ses opérandes. Un champ d'adresse d'une instruction peut référencer un mot mémoire ou un registre. On présentera par la suite quelques modes d'adressage (M.A) et on expliquera la manière dont ils opèrent.

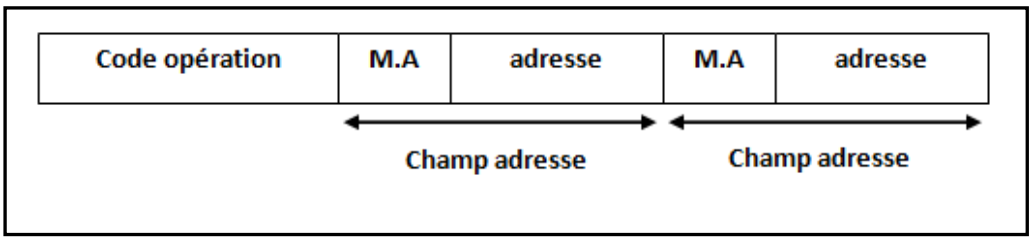



**Figure 2.11. Instruction à deux champs d'adresse (Le mode d'adressage fait partie du champ d'adresse) [3].**

Le mode d'adressage peut être partie intégrante du champ d'adresse (Figure 11) ou défini par le code opération lorsque celui-ci impose un type déterminé (Figure 12).

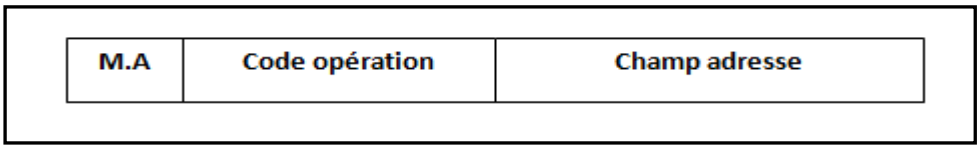



**Figure 2.12. Mode d'adressage défini par le code opération [3].**

### *6.1. Adressage immédiat*

Dans ce cas, l'opérande est contenu dans le champ adresse de l'instruction.

Exemple : MOV R2, #100 Le contenu de R2 : **100**

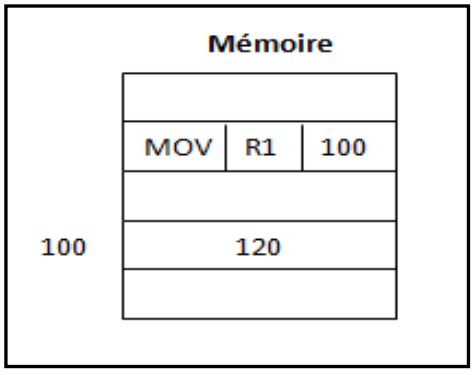
### *6.2. Adressage direct*

Dans ce cas, le champ adresse de l'instruction contient l'adresse effective de l'opérande en mémoire centrale.

Exemple : MOV R1, 100

Cette instruction met dans le registre R1, l'opérande qui se situe à l'adresse 100 en mémoire.

Le contenu de R1 : **120**



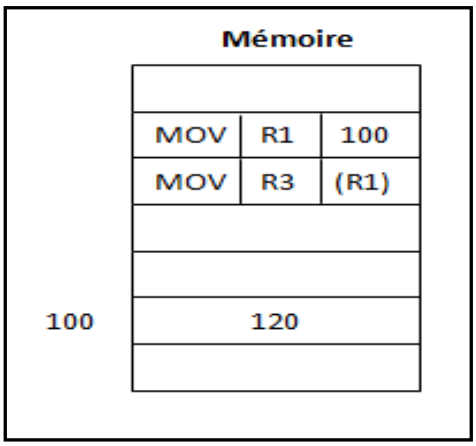
### *6.3. Adressage indirect*

Le champ adresse de l'instruction contient l'adresse d'un pointeur, c'est-à-dire l'adresse effective de l'opérande

Exemple : MOV R3, (R1)

- ✓ Si l'on suppose que R1 contient la valeur 100
- ✓ Alors, après l'instruction MOV R3, (R1)
- ✓ **R3 contiendra la valeur 120**

| | |
|---|---|
| R1 | 100 |
| R3 | 120 |



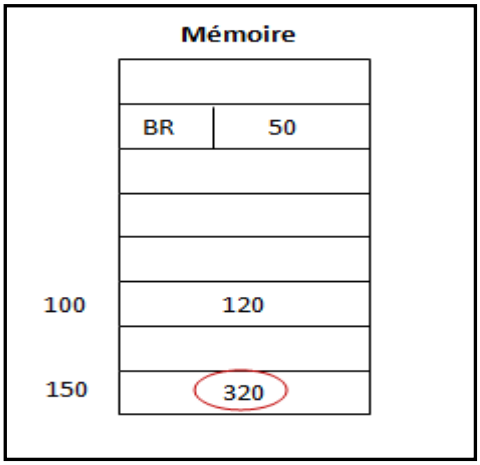

### *6.4. Adressage relatif*

L'adresse de l'opérande est obtenue en additionnant le contenu du compteur ordinal (CO) au contenu du champ adresse de l'instruction. Ce type d'adressage est utilisé, en général dans les **instructions de branchement**.

Exemple : BR, 50 / On suppose que le contenu du compteur ordinal est 100.

| | |
|---|---|
| CO | 100 |

- ✓ Adresse de l'opérande est 100+50,
- ✓ Alors l'opérande est 320



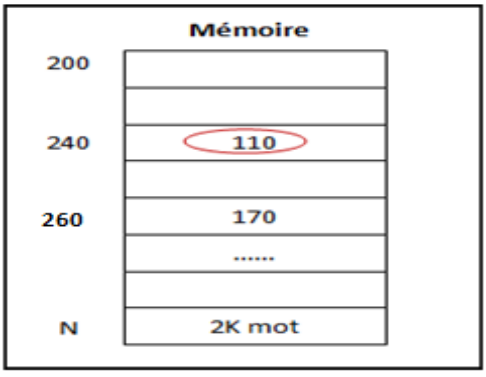

### *6.5. Adressage basé*

L'adresse de l'opérande est obtenue en additionnant le contenu du registre de base au contenu du champ adresse de l'instruction. Le registre de base contient l'adresse du début d'un bloc de 2K mots. Le champ adresse de l'instruction contient le déplacement dans le bloc.

Exemple : R1, 40

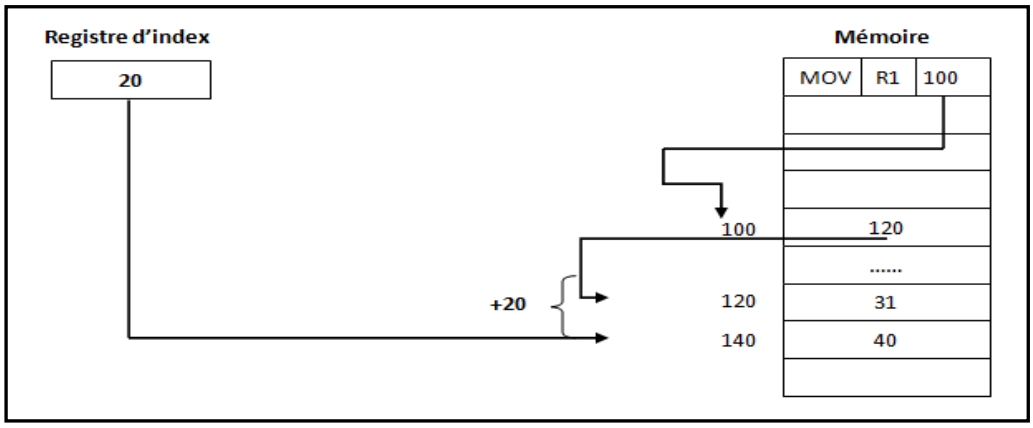
| | |
|---|---|
| RB | 200 |

- ✓ Adresse de l'opérande est 200+40



### *6.6. Adressage indexé*

Dans ce type d'adressage, l'adresse de l'opérande est calculée en additionnant le contenu du champ adresse de l'instruction au contenu du registre d'index. Ce type d'adressage est utilisé pour les tableaux, l'index permet de référencer un élément dans un tableau.




- ✓ **Les modes d'adressage (relatif, basé, indexé) travaillent sur le principe suivant : l'adresse effective est calculée sur la base du champ d'adresse de l'instruction qu'on additionne à un déplacement.**
  **Adresse effective= base+ déplacement**
- ✓ **Possible de combiner les modes d'adressage pour obtenir d'autres**
  **Exemple : Adressage indirect indexé**

## 7. Étapes d'exécution d'une instruction

La figure suivante présente le cycle d'instruction avec ses déférentes étapes : phases de recherche, décodage, exécution, écriture et stockage des résultats en MC.
Remarque : RA, RM des registres auxiliaires pour l'exécution.

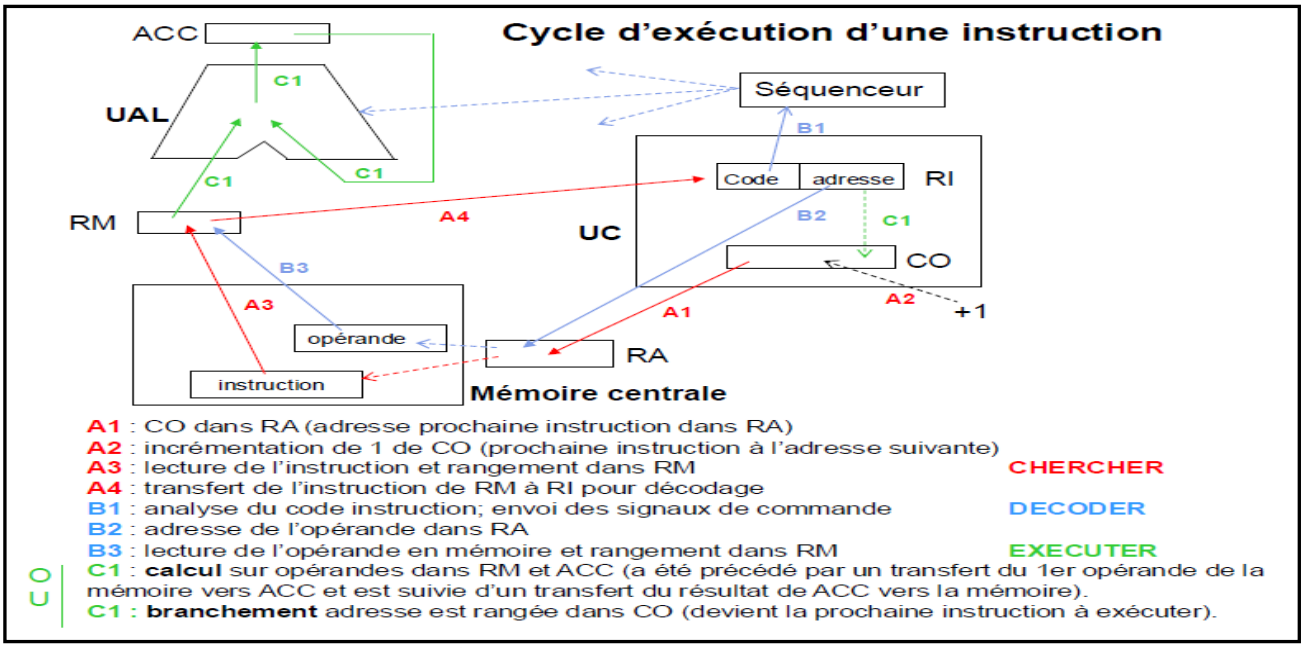



**Figure 2.13. Cycle d'exécution d'une instruction [23].**

# Chapitre 3 : Étude de cas : processeur Intel 8086

## Introduction

Le 8086 (développé en 1978) est le premier MP de type x86. Il est équipé d'un bus de données de 16 bits et un bus d'adresses de 20 bits et fonctionne à des fréquences diverses selon plusieurs variantes : 5, 8 ou 10 MHz. Le processeur 8086 d'Intel est à la base des processeurs Pentium. En effet, les processeurs successifs (de PC) se sont construits petit à petit en ajoutant à chaque processeur des instructions et des fonctionnalités supplémentaires, mais en conservant les spécificités du processeur précédent. Un registre est un élément de mémorisation interne au processeur et contenant une valeur. Les registres sont en nombre très limité (14 pour le 8086). Pour accéder à une donnée en mémoire, il faut spécifier son adresse. Pour accéder à une donnée dans un registre, il faut spécifier son nom (chaque registre possède un nom qui est une chaine de caractères). Le 8086 est un processeur 16 bits, c'est-à-dire qu' il traite des données codées sur 16 bits.

***Remarque :***

**– Un bit est une valeur binaire qui, par convention, peut prendre la valeur 0 ou 1 ;**
**– Un octet est une donnée codée sur 8 bits,**
**– Un mot est une donnée codée sur 16 bits,**
**– Un Koctet est un ensemble de 1024 octets,**
**– Un Moctet est un ensemble de 1024 Koctets.**

### 1. La gestion de la mémoire « Adressage segmenté »

La mémoire est une séquence d'octets, tous numérotés de manière unique par un entier compris entre 0 et N −1. On dit alors que la capacité de la mémoire est de N octets. On a toujours N qui est une puissance de 2 ($N=2^m$).

On appelle adresse effective (AE) d'un octet en mémoire le numéro qui lui est associé (voir Figure 3.1).

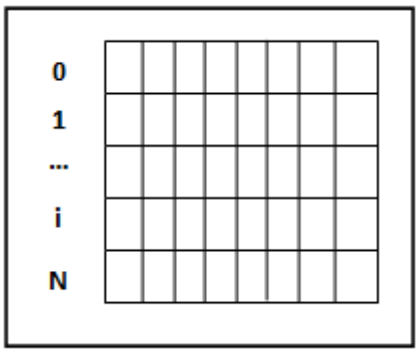



**Figure 3.1. Adresses effectives d'une mémoire de N octets [7].**

Pour le 8086, un octet est désigné par un couple (***numéro de segment, déplacement dans le segment***) avec :

- ✓ **Un segment** est un ensemble de 64 Koctets consécutifs.
- ✓ **Un déplacement (ou offset)** spécifie un octet particulier dans un segment.

Segment et déplacement sont codés sur **16 bits** et peuvent donc prendre une valeur comprise **entre 0 et 65535**, ou :

Adresse effective = 16 × segment + déplacement

Un octet d'adresse effective donnée peut être accédée de plusieurs manières. Plus précisément, à chaque AE correspond $2^{12}$ = 4096 différentes.

Lors de son exécution, un programme utilise plusieurs segments mémoire :

- ✓ Un segment contient les instructions du programme (**le segment de code**) ;
- ✓ Un segment contient les données du programme (**le segment de données**) ;
- ✓ Un segment contient la pile du programme (**le segment de pile**). Ce segment très important.

Ces trois segments peuvent se situer n'importe où en mémoire et même se recouvrir partiellement ou totalement. Leur attribution en mémoire est généralement réalisée automatiquement par le système d'exploitation sans que le programmeur ait à s'en soucier.

## 2. L'architecture du 8086

### *2.1. L'architecture externe du 8086*

Le microprocesseur 8086 est un circuit intégré se présente sous forme d'un boîtier de 40 broches alimenté par une alimentation unique de 5V. (Figure 3.2)

- ➢ Il possède un bus multiplexé adresse/donnée de 20 bits.
- ➢ Le bus de donnée occupe 16 bits ce qui permet d'échanger des mots de 2 octets
- ➢ Le bus d'adresse occupe 20 bits ce qui permet d'adresser 1 Mo

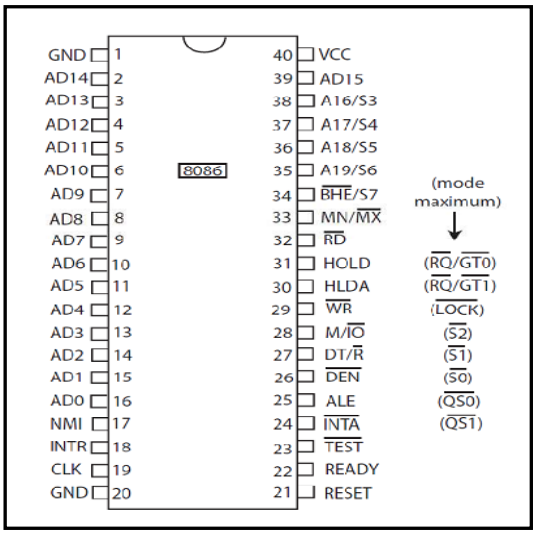



**Figure 3.2. L'architecture externe du 8086[10].**

### *2.2. L'architecture interne du 8086*

Il existe deux unités internes distinctes : l'UE (Unité d'Exécution) et l'UIB (Unité d'Interfaçage avec le Bus).

- Le rôle de l'UIB est de récupérer et stocker les informations à traiter, et d'établir les transmissions avec les bus du système. Il fournit l'interface physique entre le microprocesseur et le monde extérieur.
- L'UE exécute les instructions qui lui sont transmises par l'UIB. Il comporte essentiellement l'UAL de 16 bits qui manipule les registres généraux de 16 bits aussi. Pendant que l'UE exécute les informations qui lui sont transmises, l'instruction suivante est chargée dans l'UIB.

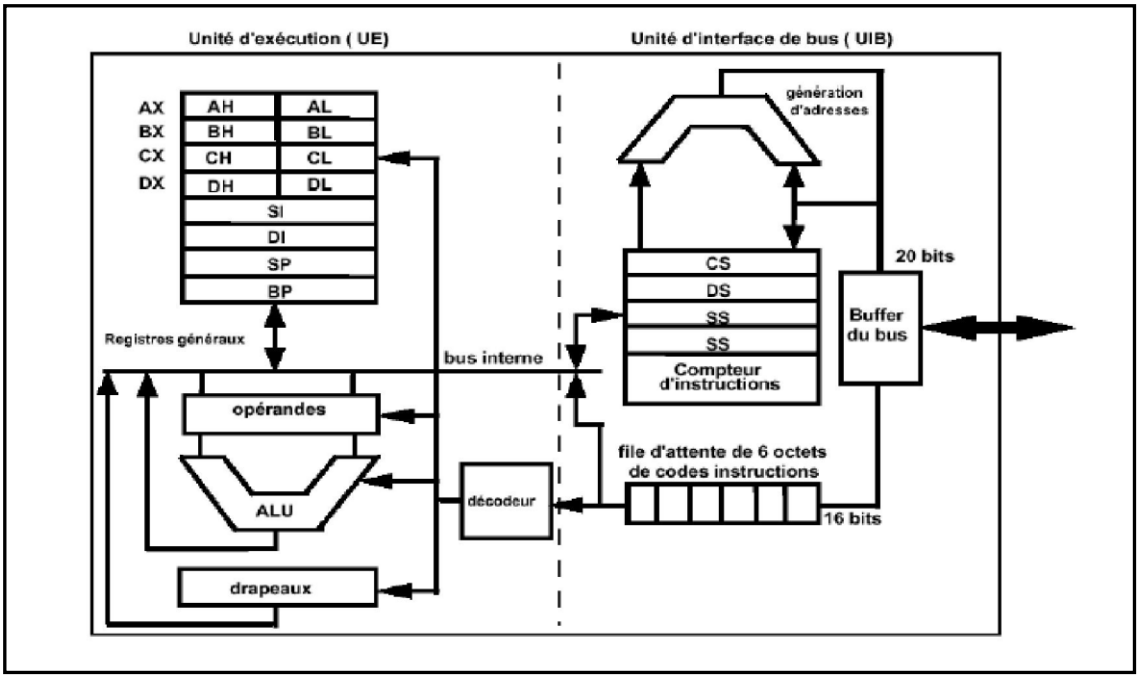



**Figure 3.3. L'architecture interne du 8086 [10].**

### *2.3. Les registres du 8086*

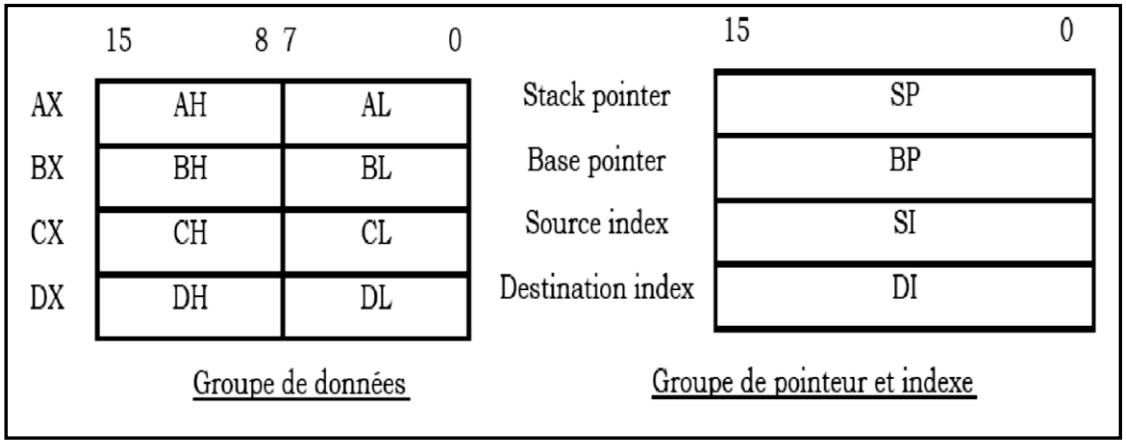
Dans le but de recevoir des informations spécifiques, notamment des adresses et des données stockées durant l'exécution d'un programme. Il existe plusieurs types de registres du microprocesseur 8086 sont tous des registres 16 bits. Ils sont répartis en 5 catégories: ***registres de travail, registres de segment, registres pointeurs, registres index et registres spéciaux***.

#### ✓ Les registres généraux de travail

En effet, les registres généraux peuvent être utilisés dans toutes les **opérations arithmétiques et logiques** que le programmeur insère dans le code assembleur.

Chaque registre est en réalité divisé en deux registres distincts de 8 bits. Le programmeur dispose de 8 registres internes de 16 bits qu'on peut diviser en deux groupes :

- **Groupe de données:** formé par 4 registres de 16 bits (AX, BX, CX et DX) chaque registre peut être divisé en deux registres de 8 bits (AH, AL, BH, BL, CH, CL, DH et DL)
- **Un groupe de pointeur et indexe:** formé de 4 registres de 16 bits (SI, DI, SP et BP) et font généralement référence à un emplacement en mémoire (segment).




➢ **Les registres du groupe de données :**

- ✓ ***Registre AX (Accumulateur):*** toutes les opérations de transferts de données avec les E/S, il est utilisé implicitement comme accumulateur (dans les instructions de multiplication et la division). Ne peut pas servir pour l'adressage.
- ✓ ***Registre BX (registre de base):*** il est utilisé pour l'adressage de données dans une zone mémoire différente de la zone code.
- ✓ **Registre CX (Le compteur):** il a été fait pour servir comme compteur lors des instructions de boucle. Ne peut pas servir pour l'adressage.
- ✓ ***Registre DX* (Data):** il est utilisé pour les opérations de division et de multiplication, il sert comme extension au registre AX pour contenir un nombre 32 bits pour la multiplication et 16 bits pour la division mais surtout pour contenir le numéro d'un port d'E/S. Ne peut pas servir pour l'adressage.

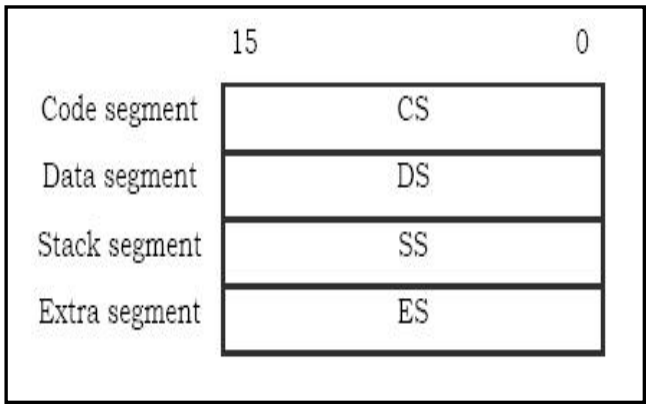
➢ **Les registres du groupe de pointeur et indexe :**

- ✓ ***L'indexe SI (Source Indexe) :*** Il permet de pointer la mémoire (Adressage). Il forme, en général, un décalage (**un offset**) par rapport à une base fixe (le registre DS), Certaines instructions de déplacement de données l'utilisent comme index de l'opérande source. Il sert pour les instructions de chaîne de caractères.
- ✓ ***L'indexe DI (Destination indexe):*** Il permet aussi de pointer la mémoire (Adressage). Il sert, aussi, pour les instructions de chaîne de caractères, il pointe alors sur l'opérande destination.
- ✓ ***Les pointeurs SP (Stack pointer):*** Pointe sur la tête de la pile (une zone mémoire qui stocke l'information avec le principe LIFO). Utilisé pour l'accès à la pile, ils présentent un décalage par rapport à la base (le registre SS)
- ✓ ***Les pointeurs BP (Base pointer) :*** Adressage comme registre de base, utilisé pour l'accès à la pile, ils présentent un décalage par rapport à la base (le registre SS)

✓ **Les registres segments**

Le 8086 a quatre registres segments de 16 bits chacun :

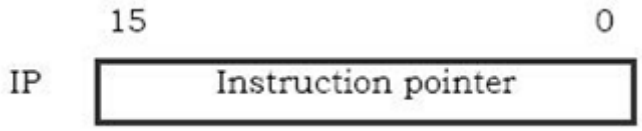
- ✓ **CS (Code Segment) :** Définit le début de la mémoire programme, les adresses des différentes instructions du programme
- ✓ **DS (Data Segment) :** Début de la mémoire de données dans laquelle sont stockées toutes les données traitées par le programme
- ✓ **ES (Extra Segment) :** Début d'un segment auxiliaire pour données.
- ✓ **SS (Stack Segment) :** Début de la pile LIFO (Last IN, First Out).




Ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le début de chacun d'entre eux. Chaque segment de mémoire ne peut excéder les 65535 octets (64 ko), ensuite on fait varier le registre ces registrent sont combiné avec les registres d'offset pour former les adresses. Une case mémoire est repérée par une adresse de la forme **RS: RO**. Ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le d'une zone mémoire de 64Ko, ensuite on fait varier le registre d'offset qui précise l'adresse relative par rapport à cette position.

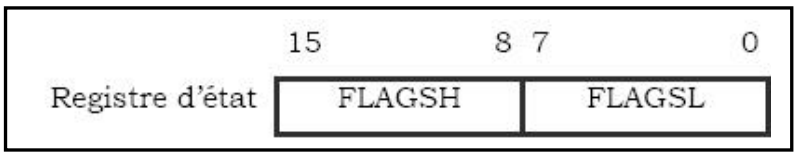
### ✓ Les registres IP ou compteur de programme

Instruction Pointer ou Compteur de Programme, contient l'adresse de l'emplacement mémoire où se situe la **prochaine instruction à exécuter**. Autrement dit, il doit indiquer au processeur la prochaine instruction à exécuter. Le registre IP est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante.

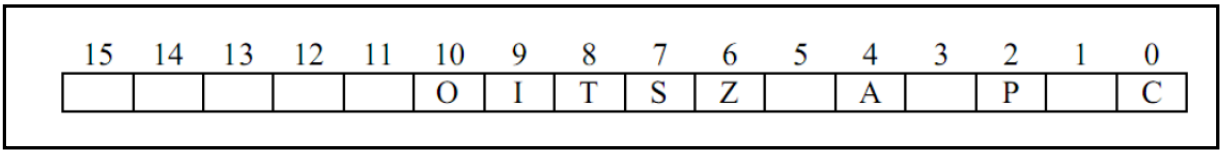



### ✓ Le registre d'état ou Flag

Le registre d'état FLAG sert à contenir l'état de certaines opérations effectuées par le processeur. Six bits reflètent les résultats d'une opération arithmétique ou logique et 3 participent au control du processeur.




Le registre d'état du 8086 est formé par les bits suivants :




P : (Parité) indique que le nombre de 1 est un nombre pair.
Z : (Zéro) Indique que le résultat d'une opération arithmétique ou logique est nul.

S : (Signe) Reproduit le bit de poids fort d'une quantité signée sur 8 bits ou sur 16 bits ; S=0 : positif, S=1 : négatif
I : (Interruption) autorise ou non la reconnaissance des interruptions : I = 0 alors Interruptions autorisées. I = 1 alors Interruptions non autorisées.
**Par exemple :** Quand le résultat d'une opération est trop grand pour être contenu dans le registre cible un bit spécifique du registre d'état (le bit OF) est mis à 1 pour indiquer le débordement.

## 3. Les modes d'adressage propres au 8086.

### *3.1 Mode d'adressage immédiat*

Dans le mode d'adressage immédiat, l'opérande source est une donnée sur 8 ou 16 bits. L'exécution d'une telle instruction est très rapide.

**Exemples:** MOV CX, 500H ; dans CX=0000 0101 0000 0000 =0500H

MOV CX, -40H ; dans CX=1111 1111 1100 0000 =FFC0H

### *3.2 Mode d'adressage registre*

Dans le mode d'adressage registre, l'opérande à utiliser est contenu dans un des registres généraux du CPU ( 8 bits ou 16 bits ). La longueur de l'opérande source et l'opérande destination doivent être identiques.

**Exemples:** MOV AX, BX et MOV AL, BL

### *3.3 Mode d'adressage direct*

Le mode d'adressage direct spécifie complètement dans l'instruction l'emplacement mémoire qui contient l'opérande. L'adresse **Registre_seg: Offset** doit être placée entre **[ ]**, si le segment n'est pas précisé, DS est pris par défaut.

**Exemples :** MOV AX, COMPTE ; COMPTE est une variable déjà déclarée Dans cette instruction, le contenu des cases mémoire dont l'adresse est pointée par DS: COMPTE et DS: COMPTE+1 sera transféré dans AX.

MOV AX, [243] : Copier le contenu de la mémoire d'adresse DS:243 dans AX
MOV [123], AX : Copier le contenu de AX dan la mémoire d'adresse DS:123

### *3.4 Mode d'adressage indirect*

Dans le mode d 'adressage registre indirect, l'adresse n'est pas donnée ou plus précisément l'offset n'est pas précisé directement dans l'instruction mais il se trouve dans l'un des 4 registres d'offset BX, BP, SI ou DI et c'est le registre qui sera précisé dans l'instruction : **[Registre segment : Registre offset]** et qu'il faudrait évidemment charger au préalable par la bonne adresse.

Exemples : MOV BX, offset COMPTE

MOV AX, [BX]

La première instruction signifie qu'on a mis dans le registre BX, l'offset de la variable COMPTE. La deuxième instruction signifie qu'on à charger le registre AX par le contenu de la case mémoire dont l'adresse se trouve pointée par le registre BX, (qui est l'offset de COMPTE).

D'autres exemples :

MOV AX, [BX] ; Charger AX par le contenu de la mémoire d'adresse DS:BX

- ✓ **Mode d'adressage relatif à une base (ou adressage basé)**

L'offset se trouve dans l'un des deux registres de base BX ou BP. On peut préciser un déplacement qui sera ajouté au contenu de Roff pour déterminer l'offset, Avec le registre BX on peut accéder à divers structure de données qui se trouvent en différents endroits de la mémoire. Pour ce la il suffit de mettre l'adresse de base dans le registre de base (BX ou BP) et on pointe les éléments de la structure par leurs déplacement par rapport à la base. Pour se déplacer vers une autre structure, il suffit de changer le registre de base.

**Exemples:** MOV AX, [BX] : Charger AX par le contenu de la mémoire d'adresse DS:BX

MOV AX, [BX+5] : Charger AX par le contenu de la mémoire d'adresse DS:BX+5

- ✓ **Mode d'adressage direct indexé**

Dans le mode d'adressage indexé, l'adresse effective est la somme d'un label ou d'un déplacement et d'un registre indexe SI ou DI.

**Exemples :** MOV AX, [SI] ; Charger AX par le contenu de la mémoire d'adresse DS:SI

MOV AX, [SI+500] ; Charger AX par la mémoire d'adresse DS:SI+500
MOV AX, [DI-8] ; Charger AX par la mémoire d'adresse DS:DI-8

### *3.5 Mode d'adressage basé indexé*

L'offset de l'adresse de l'opérande est la somme d'un registre de base, d'un registre d'index et d'un déplacement optionnel. Si Rseg n'est pas spécifié, le segment par défaut du registre de base est utilisé : Ce modes d'adressage est intéressant dans le cas où on veut accéder à une structure bidimensionnelle comme les matrices. Dans ce cas le registre de base contient l'adresse de départ du tableau, le registre index contient le numéro de la ligne et le déplacement le numéro de la colonne.

**Exemples :** MOV AX, [BX+SI] ; AX est chargé par la mémoire d'adresse DS:BX+SI
MOV AX, [BX+DI+5] ; AX est chargé par la mémoire d'adresse DS:BX+DI+5

## 4. Étapes de fonctionnement d'une instruction dans le 8086

Le langage assembleur est très proche du langage machine (des informations en binaire). Il dépend donc du type de processeur. L'utilisation du langage assembleur consiste à écrire sous forme symbolique la succession d'instructions. Ces instructions sont stockées dans un fichier texte (le fichier source) qui, grâce à un programme spécifique (assembleur) sera traduit en langage machine. Donc l'avantage de l'assembleur est de générer des programmes efficaces et rapides à l'exécution.

**Exemple:** Pour copier le contenu de la mémoire à l'adresse 0110h dans le registre AX du processeur 8086, on écrit : MOV AX, [0110]

Pour traduire un programme en langage machine, on doit passer par cycle d'exécution suivant :

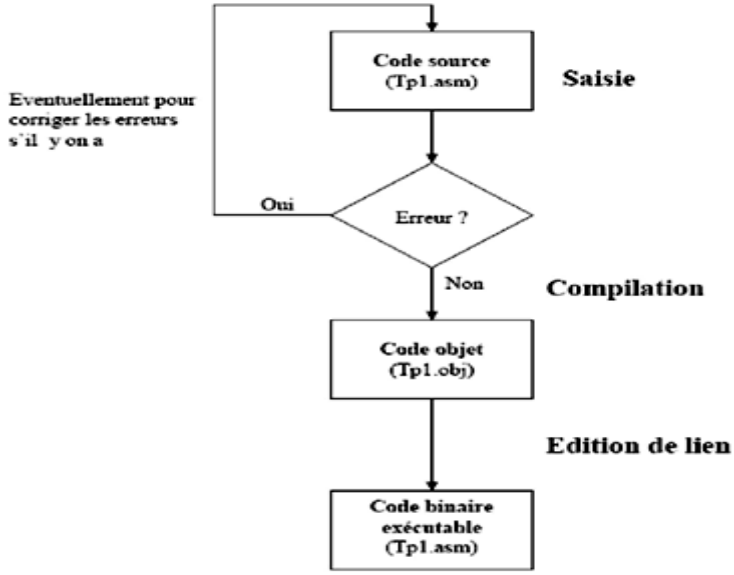



**Figure 3.4. Cycle d'exécution d'un programme en langage assembleur [10].**

### *4.1. Constitution d'un programme*

Un programme assembleur est un ensemble de déclarations constitué de directives et d'instructions

✓ **Les directives**

➢ **Les directives de données : (EQU; DB; DW ; DD)**

Déclaration d'une constante

- ***EQU :*** **Nom EQU expression** ; assigne le nom de l'expression au nom

Exemples : Pi EQU 3.14 ; valeur constante

➢ **Déclaration des variables**

- ***DB (define byte) :*** [nom] DB expression

Exemple : *Nb_max DB 123 ;* déclaration d'une variable initialisée à 123.

- ***DW (define word) :*** [nom] DW expression DW réserve des mots à deux octets.
- ***DD (define double) :*** [nom] DD expression DD réserve des mots à quatre octets.
- ***Dup :*** Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive dup:

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

➢ **Les directives Word PTR et Byte PTR**

On doit utiliser une directive spécifiant la taille de la donnée à transférer :

Exemples : MOV byte ptr [BX], val ; concerne 1 octet

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

✓ **Les directives de segment et de procédure (SEGMENT ; ASSUME ; PROG/END PROG)**

➢ ***SEGMENT*** : cette directive délimite un segment, elle permet de contrôler la relation entre la génération du code objet et la gestion des segments logiques ainsi générés.

Syntax de la directive: **nom_seg SEGMENT [align][combine][‘class’]**

- ***ASSUME :*** La directive ASSUME permet d'indiquer à l'assembleur quel est le segment de données et celui de codes (etc...), afin qu'il génère des adresses correctes. Puis il s'agit d'initialiser le segment de données DS (même chose pour : ES et SS)
  MOV AX, nom_du_segment_de_données;
  MOV DS, AX ;
- ***PROG / END PROG :*** Les directives permettent d'indiquer le début et la fin du programme.

> **Rappel :**
>
> Un processeur est relié à la mémoire par l'intermédiaire d'une liaison appelée bus. **Les données** dont le processeur 8086 a besoin sont stockées dans des registres **(AX, BX, CX, DX, IP, SI, CS, DS, SS, ...)**. Chacun a sa propre utilité. Les registres **AX, BX, CX et DX** sont les registres les plus utilisés pour **les calculs**.

```
PROGRAM  Exemple1
Pile SEGMENT STACK
        ; On met les directives pour réserver de l'espace mémoire.
PILE ENDS
Data SEGMENT
        ; On met les directives de données pour réserver de la mémoire
        ; Pour les variables qui seront utilisées dans le programme.
Data ENDS
Extra SEGMENT
        ; On met les directives pour déclarer les variables (les chaînes de
        ; Caractères).
Extra ENDS
Code SEGMENT
    ASSUME  cs : code, ds, data, es : pile :ss :pile
PROG
    Mov AX,Data
    Mov DS,AX
    Mov AX,Extra
    Mov ES,AX
    Mov AX,pile
    Mov SS,AX
        ; mettre les instructions du programme
Code ENDS
  END PROG
```

**Figure 3.5. Exemple d'un cycle d'exécution complet [10].**

### *4.2. Les instructions*

Chaque microprocesseur reconnait un ensemble d'instructions appelé jeu d'instructions (Instruction Set) fixé par le constructeur. Pour les microprocesseurs classiques, le nombre d'instructions reconnues varie entre 75 et 150 (microprocesseurs CISC : Complex Instruction Set Computer). Il existe aussi des microprocesseurs dont le nombre d'instructions est trés réduit (microprocesseurs RISC : Reduced Instruction Set Computer) : entre 10 et 30 instructions, permettant d'améliorer le temps d'exécution des programmes. Le 8086 possède 92 types d'instructions de base qu'on peut diviser les instructions du 8086/88 en groupes comme suit :

- ✓ **Instructions de transfert de données.**
- ✓ **Instructions arithmétiques**
- ✓ **Instructions logiques.**
- ✓ **Instructions de sauts de programme.**
- ✓ **Instructions de boucles de répétition**.

#### ✓ Instructions de transfert de données :

Elles permettent de déplacer des données d'une source vers une destination :

- ***Registre vers mémoire ;***
- ***Registre vers registre ;***
- ***Mémoire vers registre***.

*Remarque :* le microprocesseur 8086 n'autorise pas les transferts de mémoire vers mémoire (pour ce faire, il faut passer par un registre intermédiaire)

| Usage | Nom | Fonction |
|---|---|---|
| Général | MOV | Transfert d'octets ou de mots |
| | PUSH | Chargement de la pile |
| | POP | Déchargement de la pile |
| | PUSHA | Chargement de tous les registres dans la pile |
| | POPA | Déchargement de tous les registres dans la pile |
| | XCHG | Echange d'octet ou de mot |
| | XLAT | Translation d'octet |
| Entrées-sorties | IN | Entrée de mot ou d'octet |
| | OUT | Sortie de mot ou d'octet |
| Adresses | LEA | Chargement de l'adresse effective |
| | LDS | Chargement du pointeur avec DS |
| | LES | Chargement du pointeur avec ES |
| Indicateurs | LAHF | Transfert des indicateurs dans AH |
| | SAHF | Rangement de AH dans les indicateurs |
| | PUSHP | Chargement des indicateurs dans la pile |
| | POPF | Déchargement des indicateurs de la pile. |

- **MOV :** Elle permet de transférer les données (un octet ou un mot) d'un registre à un autre registre ou d'un registre à une case mémoire, sa syntaxe est comme suit :

**MOV destination, source**

Exemples :

MOV AX, BX : Transfert d'un registre de 16 bits vers un registre de 16 bits.

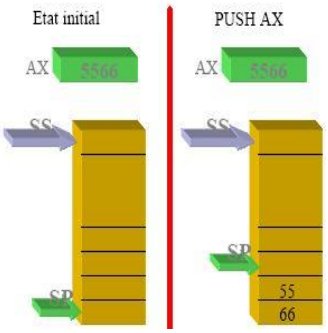
MOV AH, CL : Transfert d'un registre de 8 bits vers un registre de 8 bits.

MOV AX, [Val1] ; Transfert du contenu d'une case mémoire 16 bits vers AX.

MOV [Val2], AL ; Transfert du contenu du AL vers une case mémoire d'adresse Val2.

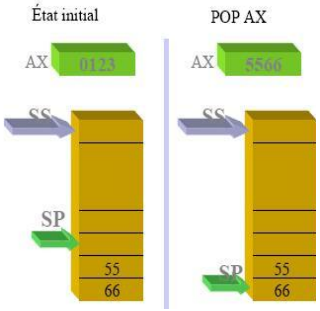
- **PUSH :** Elle permet d'empiler les registres du CPU sur le haut de la pile

Syntaxe : **PUSH source**

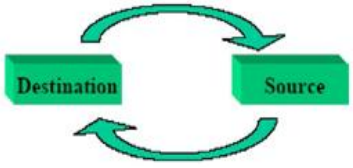



- **POP :** Elle permet de dépiler les registres du CPU sur le haut de la pile.

Syntaxe : **POP destination**




- **XCHG :** Elle permet de permuter la source avec la destination




Exemple :

**XCHG** AX, BX : elle permute entre AX et BX.

Si AX=8976 et BX=5678, alors après l'exécution de l'instruction on aura AX=5678 et BX=8976

- **IN / OUT :** Elle permet de récupérer des données d'un port (donc de la périphérie) ou restituer des données à un port, dans les deux cas s'il s'agit **d'envoyer ou de recevoir un octet on utilise l'accumulateur AL**, ou s'il s'agit **d'envoyer ou de recevoir un mot on utilise l'accumulateur AX**.

Syntaxe : **IN** ACCUMULATEUR, DX

**OUT** DX, ACCUMULATEUR

### ✓ Instructions arithmétiques

| Usage | Nom | Fonction |
|---|---|---|
| Addition | ADD | Addition sur un octet ou un mot |
| | ADC | Addition sur un octet ou un mot avec retenue |
| | INC | Incrémentation de 1 |
| | AAA | Ajustement ASCII |
| | DAA | Ajustement décimal |
| Soustraction | SUB | Soustraction sur un octet ou un mot |
| | SBB | Soustraction sur un octet ( mot ) avec retenue |
| | DEC | Décrémentation de 1 |
| | NEG | Mètre un octet ou un mot en négatif |
| | CMP | Comparaison d'octet ou mot |
| | AAS | Ajustement ASCII |
| | DAS | Ajustement décimal |

| Usage | Nom | Fonction |
|---|---|---|
| Multiplication | MUL | Multiplication d'octet ou de mot **non signée** |
| | IMUL | Multiplication d'octet ou de mot **signée** |
| | AAM | Ajustement ASCII |
| Division | DIV | Division d'octet ou de mot **non signée** |
| | IDIV | Division d'octet ou de mot **signée** |
| | AAD | Ajustement ASCII |
| | CBW | Conversion d'un octet en un mot |
| | CWD | Conversion d'un mot en double mots |

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

**Destination ⟵ Destination + source**

Syntaxe : **ADD** Destination, source

Exemples :

ADD AX, BX ; AX = AX + BX (addition sur 16 bits).

ADD AL, [SI] ;AL = AL + le contenu de la case mémoire pointé par SI.

ADD [DI], AL ; le contenu de la case mémoire pointé par DI est additionnée avec AL, le résultat est mis dans la case mémoire pointé par DI.

- **ADC (Addition avec retenue) :** Elle permet d'additionner le contenu de la source (octet ou un mot) avec celui de la destination et la retenue (CF); le résultat est mis dans la destination.

Syntaxe : **ADC** Destination, source

Exemples :

ADC AX, BX ; AX = AX + BX + CF (addition sur 16 bits )

ADC AL, BH ; AL = AL + BH + CF (addition sur 8 bits )

ADC AL, [SI] ; AL = AL + le contenu de la case mémoire pointé par SI + CF

- **SUB (Soustraction) :** Elle permet de soustraire la destination de la source (octet ou un mot) le résultat est mis dans la destination.

Syntaxe : **SUB** Destination, source

Exemples :

SUB AX, BX ; AX = AX - BX (Soustraction sur 16 bits ). SUB AL, BH : AL = AL - BH (Soustraction sur 8 bits ).

SUB AL, [SI] ; AL = AL - le contenu de la case mémoire pointé par SI.

SUB [DI], AL ; le contenu de la case mémoire pointé par DI= le contenu de la case mémoire pointé par DI - AL .

- **SBB (Soustraction avec retenue) :** Elle permet de soustraire la destination de la source et la retenue (octet ou un mot) le résultat est mis dans la destination.

Syntaxe : **SBB** Destination, source

Exemples :

SBB AX, BX ; AX = AX – BX - CF (Soustraction sur 16 bits ).

SBB AL, [SI] ; AL = AL - le contenu de la case mémoire pointé par SI – CF.

- **INC (Incrémentation):** Elle permet d'incrémenter le contenu de la destination.
  Syntaxe : **INC** Destination

Exemples :

INC AX : AX = AX + 1 (incrémentation sur 16 bits).

INC AL : AL = AL +1 (incrémentation sur 8 bits).

INC [SI] : [SI] = [SI] + 1 Le contenu de la case mémoire pointé par SI sera incrémenter.

- **DEC (Décrémentation) :** elle permet de décrémenter le contenu de la destination.
  Syntaxe : **DEC** Destination

Exemples :

DEC AX ; AX = AX - 1 (décrémentation sur 16 bits).

DEC [SI] ; [SI] = [SI] – 1 ; le contenu de la case mémoire pointé par SI sera décrémenté.

- **NEG (Négatif) :** elle soustrait l'opérande destination (octet ou mot) de 0 le résultat est stocker dans la destination. Donc cette opération réalise le complément à deux d'un nombre.

  Syntaxe : **NEG** Destination

Exemples :

NEG AX ; AX = 0 – AX.

NEG AL ; AL = 0 – AL.

NEG [SI] ; [SI] = 0 - [SI].

- **CMP (Comparaison):** Elle soustrait la source de la destination, qui peut être un octet ou un mot, le résultat n'est pas mis dans la destination, en effet ***cette instruction touche uniquement les indicateurs*** pour être tester ***avec un autre instruction ultérieure de saut conditionnel***. Les indicateurs susceptibles d'être touché sont : AF, CF, OF, PF, SF, ZF. Cette instruction va nous **permettre de comparer deux nombres**.

  Syntaxe : **CMP** Destination , Source

- **MUL :** Elle effectue une multiplication non signée de l'opérande source avec l'accumulateur.
  Syntaxe : **MUL** Source

Si la source est **un octet** alors elle sera **multipliée par l'accumulateur AL** le **résultat sur 16 bits** sera **stocké dans le registre AX**. Si la source est **un mot** alors elle **sera multipliée avec l'accumulateur AX** le **résultat de 32 bits** sera **stocké dans la paire des registres AX et DX**. Cette multiplication **traite les données en tant que nombres non signés**.

- **DIV :** Elle effectue **une division non signée** de **l'accumulateur** par l'opérande source.

Syntaxe : **DIV** Source

Si **l'opérande est un octet**, alors **on récupère le quotient** dans le registre **AL** et **le reste dans le registre AH**.

Si **l'opérande est un mot**, alors **on récupère le quotient** dans le registre **AX** et **le reste dans le registre DX**.

### ✓ Instructions logiques

| Usage | Nom | Fonction |
|---|---|---|
| Logique | NOT | Inversion logique sur un octet ou un mot |
| | AND | Et logique |
| | OR | Ou logique |
| | XOR | Ou exclusif |
| | TEST | Et logique sans résultat, affecte uniquement les indicateurs du registre des flags. |
| Décalages | SHL | Décalage logique à gauche |
| | SAL | Décalage arithmétique à gauche |
| | SHR | Décalage logique à droite |
| | SAR | Décalage arithmétique à droite |
| Rotation | ROL | Rotation à gauche |
| | ROR | Rotation à droite |
| | RCL | Rotation à gauche à travers le bit de retenue |
| | RCR | Rotation à droite à travers le bit de retenue |

- **NOT (Négation) :** Elle réalise la **complémentation à 1** d'un nombre.

Syntaxe : **NOT** Destination

Exemple :

```
MOV AX, 500H    ; AX = 0000 0101 0000 0000
NOT AX          ; AX = 1111 1010 1111 1111
```

- **AND :** Elle permet de faire un ET logique entre la destination et la source (octet ou mot) le résultat est mis dans la destination.
  Syntaxe : **AND** Destination, source

Exemples :

MOV AX , 503H ; AX = 0000 0101 0000 0011

AND AX , 0201H ; 0000010100000011 x 0000001000000001 = 000000000000 0001

AND AX, BX ; AX = AX . BX (Et logique entre AX et BX)

AND AL, BH ; AL = AL . BH (ET logique sur 8 bits)

AND AL, [SI] ; AL = AL AND le contenu de la case mémoire pointé par SI.

- **OR :** Elle permet de faire un **OU logique** entre la destination et la source (octet ou mot) le résultat est mis dans la destination.

Syntaxe : **OR** Destination, source

Exemples :

MOV AX , 503H ; AX = 0000 0101 0000 0011

OR AX , 0201H ; 0000010100000011 OR 0000001000000001 = 0000011100000011

- **SHL :** Elle permet de faire **le décalage logique à gauche** de la destination.

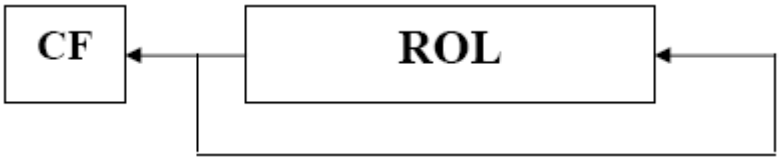

Syntaxe : **SHL** destination, compteur

Exemples :

SHL AL,1 ; Pour faire un seul décalage.

MOV CL, 3
SHL AL, CL } Pour faire trois décalages de suite

- **ROL :** Elle permet de faire la rotation à gauche de la destination.



Syntaxe : **ROL** destination, compteur

Exemples :

ROL AL,1 ; Pour faire une seule rotation

MOV CL,4
ROL AL,CL } Pour faire quatre rotations de suite

✓ **Instructions de sauts et de branchement**

| Type | Nom | Fonction |
|---|---|---|
| Branchements inconditionnels | CALL | Appel à un sous programme |
| | RET | Retour d'un sous programme |
| | JMP | Saut |
| Branchements conditionnels (arithmétique non signée) | JA/JNBE | Si supérieur / Si non inférieur ou non égal |
| | JAE/JNB | Si supérieur ou égal/ Si non inférieur |
| | JB/JNAE | Si inférieur/si non supérieur ni égal |
| | JBE/JNA | Si inférieur ou égal/si non supérieur. |
| Branchements conditionnels (arithmétique signée) | JG/JNLE | Si plus grand/si pas inférieur ni égal |
| | JGE/JNL | Si plus grand ou égal/Si pas inférieur |
| | JL/JNGE | Si moins que/Si pas plus grand ni égal |
| | JLE/JNG | Si moins que ou égal/Si pas plus grand |

| Type | Nom | Fonction |
|---|---|---|
| Branchement conditionnels ( flags) | JC | Si retenue |
| | JE/JZ | Si égal/Si zéro |
| | JNC | Si pas de retenue |
| | JNE/JNZ | Si non égal / Non zéro |
| | JNO | Si pas de débordement |
| | JNP/JPO | Si pas de parité/ Si parité impaire |
| | JNS | Si pas de signe |
| | JO | Si débordement |
| | JP/JPE | Si parité / Si parité paire |
| | JS | Si signe (négatif) |
| Boucles | LOOP | Boucle |
| | LOOPE/LOOPZ | Boucle si égal/Si zéro |
| | LOOPNE/LOOPNZ | Boucle si différent/si diff 0 |
| | JCXZ | Branchement si CX=0 |
| Interruptions | INT | Interruption |
| | INTO | Interruption si débordement |
| | IRET | Retour d'interruption. |

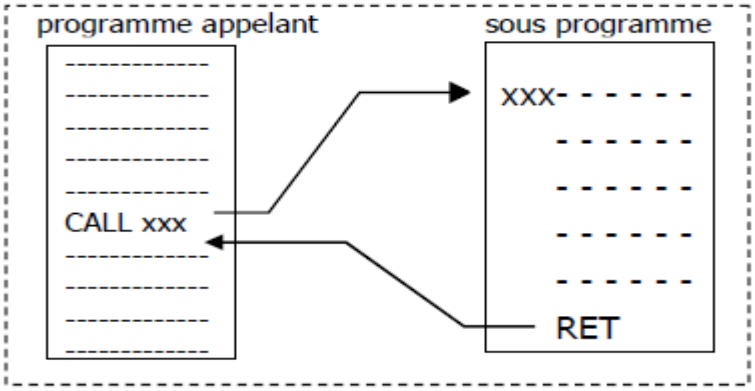
- **Branchements inconditionnels**

➢ **CALL label :** Appel d'une procédure (sous-programme) qui commence à la ligne « label ». La position de l'instruction suivant le CALL est empilée pour assurer une poursuite correcte après l'exécution du sous-programme.
Lors de l'exécution de l'instruction CALL, le pointeur d'instruction IP est chargé avec l'adresse de la première instruction du sous-programme. Lors du retour au programme appelant, l'instruction suivant le CALL doit être exécutée, c'est-à-dire que IP doit être rechargé avec l'adresse de cette instruction.

➢ **JMP (Saut inconditionnel) :** Elle **transfert**, **sans condition**, la commande à l'emplacement de destination. L'opérande Cible peut être obtenu à partir de l'instruction elle-même (JMP direct) ou à partir de la mémoire ou à partir d'un registre indiqué par l'instruction.

Syntaxe : **JMP** cible

➢ **RET Retour de sous-programme**. L'exécution du programme continue à la position récupérée dans la pile.
➢ **INT n :** appel à **l'interruption** logicielle n° : n




- **Branchements conditionnels**

Les branchements conditionnels sont conditionnés par l'état des indicateurs (drapeaux) qui sont eux même positionnés par les instructions précédentes.
Dans la suite nous allons utiliser la terminologie :

- **supérieur** ou **inférieur** pour les nombres non signés
- **plus petit** ou **plus grand** pour les nombres signés
- + pour l'opérateur logique OU

➢ **JE/JZ label (Jump if Equal or Zero )** Aller à la ligne label si résultat nul ou si égalité. C'est-à-dire si Z=1
➢ **JNE/JNZ :( Jump Si non égal / Non zéro)** Elle: **transfert, avec condition**, la commande à l'emplacement de destination. Si ZF=0 alors IP = IP + déplacement.

Syntaxe : **JNZ** cible

✓ **Instructions boucles de répétition**

| Type | Nom | Fonction |
|---|---|---|
| Boucles | LOOP | Boucle |
| | LOOPE/LOOPZ | Boucle si égal/Si zéro |
| | LOOPNE/LOOPNZ | Boucle si différent/si diff 0 |
| | JCXZ | Branchement si CX=0 |

- **LOOP :** L'instruction loop fonctionne automatiquement avec le registre CX (compteur). Quant le processeur rencontre une instruction loop, il décrémente le registre CX. Si le résultat n'est pas encore nul, il reboucle à la ligne portant l'étiquette **label**, sinon il continue le programme à la ligne suivante

  Syntaxe: **LOOP** label

- **LOOPE** (whileEqual)**, LOOPZ** (Loop While Zero) Décrémente le registre CX (aucun flag n'est positionné) on reboucle tant que CX est différent de zéro et le flag Z est égal à 1. La condition supplémentaire sur Z, donne la possibilité de quitter une boucle avant que CX ne soit égal à zéro.
- **LOOPNE, LOOPNZ** (Loop While Not Equal)**:** Décrémente le registre CX et reboucle tant que CX est différent de zéro et le flag Z est égal à 0. Fonctionne de la même façon que loopz.
- **JCXZ label :** branchement à la ligne **label** si CX = 0 indépendamment de l'indicateur Z.

## 5. Génération du code machine

### *5.1. Etapes de la réalisation d'un programme*

1) Définir le problème à résoudre : que faut-il faire exactement ?
2) Déterminer des algorithmes, des organigrammes : comment faire ? Par quoi commencer, puis poursuivre ?
3) Rédiger le programme (**code source**) :

- ✓ utilisation du jeu d'instructions (mnémoniques) ;
- ✓ création de documents explicatifs (documentation).

4) Tester le programme en réel ;
5) Corriger les erreurs (**bugs**) éventuelles : **déboguer** le programme puis refaire des tests jusqu'à obtention d'un programme fonctionnant de manière satisfaisante.

- **Langage machine et assembleur :**
  - ✓ Langage machine : codes binaires correspondant aux instructions ;
  - ✓ Assembleur : logiciel de traduction du code source écrit en langage assembleur (mnémoniques).

### *5.2. Réalisation pratique d'un programme*

1) Rédaction du code source en assembleur à l'aide d'un éditeur (logiciel de traitement de texte ASCII) :

- ✓ Edit sous MS-DOS,
- ✓ Notepad (bloc-note) sous Windows,

2) Assemblage du code source (traduction des instructions en codes binaires) avec un assembleur :

- ✓ MASM de Microsoft,
- ✓ **Emu 8086**, ...

3) Pour obtenir le **code objet** : code machine exécutable par le microprocesseur ;

- ✓ Chargement en mémoire centrale et exécution : rôle du système d'exploitation (ordinateur) ou d'un moniteur (carte de développement à base de microprocesseur).

4) Pour la mise au point (débogage) du programme, on peut utiliser un programme d'aide à la mise au point (comme DEBUG sous MS-DOS) permettant :

- ✓ l'exécution pas à pas;
- ✓ la visualisation du contenu des registres et de la mémoire ;
- ✓ la pose de points d'arrêt ...

### *5.3. Structure d'un fichier source en assembleur*

Pour faciliter la lisibilité du code source en assembleur, on le rédige sous la forme suivante :

**Labels instructions commentaires**

```
label1 : mov ax, 60H   ;ceci est un commentaire ...
...
...
sous prog1 proc near ; sous-programme
...
...
sous prog1 endp
...
...
```

## 6. Gestion des entrées/sorties par le 8086

Une interface d'entrées/sorties est un circuit intégré permettant au microprocesseur de communiquer avec l'environnement extérieur (périphériques) : clavier, écran, imprimante, modem, disques, processus industriel, ...Les interfaces d'E/S sont connectées au microprocesseur à travers les bus d'adresses, de données et de commandes. Les jonctions entre les interfaces et les périphériques sont appelés **port**. Le 8086 dispose d'un espace mémoire de 1 Mo (adresse d'une case mémoire sur 20 bits) et d'un espace d'E/S de 64 Ko (adresse d'un port d'E/S sur 16 bits).

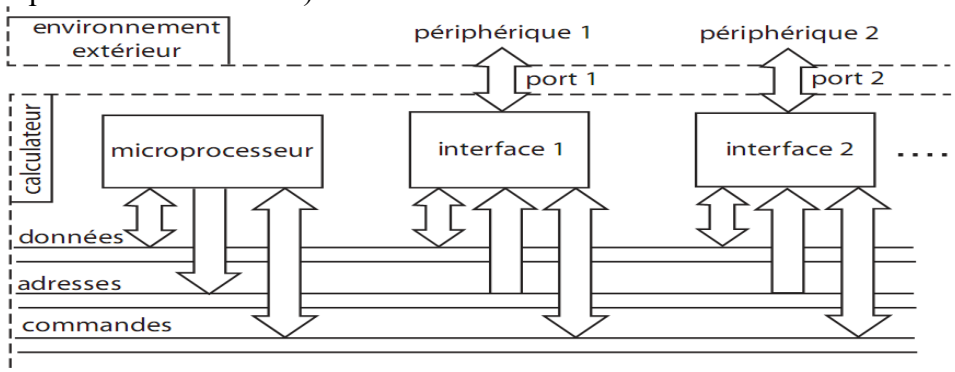



**Figure 3.6. Les interfaces d'entrées/sorties et la relation avec le microprocesseur [10].**

Exemple :

| interface | port | exemple de périphérique |
|---|---|---|
| interface parallèle<br>interface série | port parallèle<br>port série | imprimante<br>modem |

### *6.1. Instructions d'accès aux ports d'E/S*

Les instructions de lecture et d'écriture d'un port d'E/S sont respectivement les instructions **IN** et **OUT**.

✓ **Lecture d'un port d'E/S :**

- Si l'adresse du port d'E/S est sur un octet :
  - ✓ IN al, adresse : lecture d'un port sur 8 bits
  - ✓ IN ax, adresse : lecture d'un port sur 16 bits

- Si l'adresse du port d'E/S est sur deux octets :
  - ✓ IN al, dx: lecture d'un port sur 8 bits
  - ✓ IN ax, dx: lecture d'un port sur 16 bits
  - ✓ Où le registre DX contient du port d'E/S à lire.

✓ **Ecriture d'un port d'E/S :**

- Si l'adresse du port d'E/S est sur un octet :
  - ✓ OUT adresse, al : lecture d'un port sur 8 bits
  - ✓ OUT adresse, ax : lecture d'un port sur 16 bits
- Si l'adresse du port d'E/S est sur deux octets :
  - ✓ OUT dx, al: lecture d'un port sur 8 bits
  - ✓ OUT dx, ax : lecture d'un port sur 16 bits
  - ✓ Où le registre DX contient du port d'E/S à lire.

Exemple :

- Lecture d'un port d'E/S sur 8 bits à l'adresse 4221h :

```
mov dx, 4221h
in al, dx
```

- Ecriture de la valeur AF37 H dans le port d'E/S sur 16 bits à l'adresse 52 h :

```
Mov ax, AF37H
OUT 52 h, ax
```

## 7. Gestion de la pile

Une pile est une zone mémoire servant à stocker temporairement des valeurs.
On ne peut stocker qu'une information à la fois et l'élément dépilé à un moment donné est celui qui a été empilé en dernier: c'est la structure LIFO (Last In, First Out).

- ✓ Les opérations ne se font que sur 16bits.
- ✓ La pile commence au segment SS et elle finit dans le même segment mais à l'offset SP.
- ✓ A noter que cette pile est remplie à l'envers: le premier élément est situé à une adresse plus haute que le dernier élément
- ✓ Les données ne sont pas effacées après un POP

- ✓ Un PUSH écrase les données précédemment dans la pile

### *7.1. L'instruction d'empilement (PUSH)*

L'empilement d'une donnée est réalisé par l'instruction: **PUSH donnée**

Où donnée est un registre 16bits ou un emplacement mémoire désigné par adressage direct ou indirect.

L'empilement s'effectue en deux étapes:

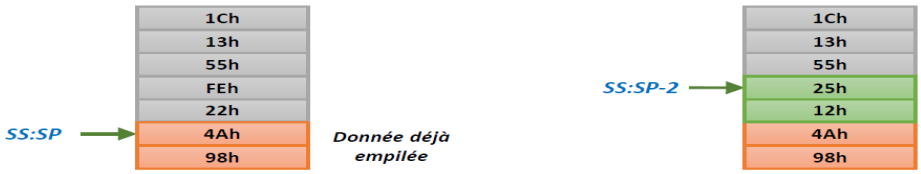
*SP←SP- 2*: on fait reculer SP de deux cases (donnée sur16bits) ,pour qu'il pointe sur le nouveau sommet susceptible de recevoir la donnée à empiler.

*[SP]←donnée:* le sommet de la pile reçoit la donnée.

Exemple:

Soit AX=1225h et soit l'instruction: PUSH AX.

• ***Avant l'instruction :*** • ***Après l'instruction :***




### *7.2. L'instruction de dépilement (POP)*

Le dépilement est l'opération qui permet de retirer le sommet de la pile et le stocker dans une destination.

- Il est réalisé par l'instruction: **POP destination**

Destination est un registre 16 bits ou un emplacement mémoire désigné par adressage direct ou indirect. Le dépilement engendre deux opérations:

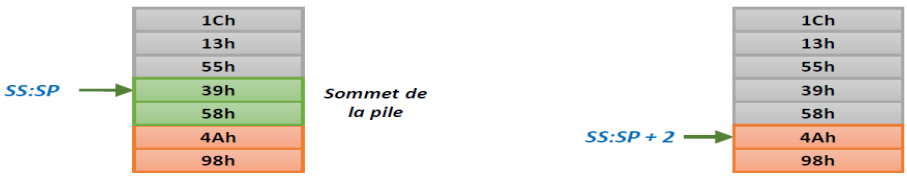
destination←[SP]: le sommet de la pile est copié dans destination.

SP←SP+2: la donnée du sommet étant retirée, on fait avancer SP pour pointer sur la donnée suivante.

Exemple:

- Considérons l'instruction: POP DX

• ***Avant l'instruction :*** • ***Après l'instruction :***




• ***Le registre DX est chargé par la valeur 5839h***

### *7.3. Autres instructions de pile*

✓ **PUSHA**(PushAll): Empile les 8 registres généraux.

• Les registres sont empilés dans l'ordre AX, CX, DX, BX, SP, BP, SI, DI.

• La valeur empilée de SP est celle avant le début de l'instruction.

✓ **POPA** (PopAll): Dépile les16 octets du sommet de la pile vers les 8 registres généraux.

•Les registres sont dépilés dans l'ordre DI, SI, BP, SP, BX, DX, CX, AX.

•En fait, le registre SP pose un cas particulier, puisqu'il n'est pas mis à jour avec la valeur dépilée qui n'est pas prise en compte.

✓ **PUSHF** (Push Flags): Empile le registre des indicateurs.
✓ **POPF** (Pop Flags): Dépile vers le registre des indicateurs.

## 8. Fonctions et sous-programmes

Un **Sous-Programme** (**procédure**) : c'est une séquence d'instructions qui possède un nom unique, et fait l'appel à l'intérieur d'un **programme principal** pour éviter la répétition d'une même séquence d'instructions plusieurs fois dans un programme et engendre un accès à la pile (mémorisation de l'adresse de retour), le branchement se fait lors de l'appel.

✓ **Instructions de sauvegarde** : si le Sous Programme utilise des registres du processeur, il est important de sauvegarder ces registres sur la pile au début du **SP**.L'utilisation du registre **SP (Optimisation de la place mémoire):** sert à stocker dans une seule zone mémoire ou l'adresse de stockage déterminée par l'assemblage

**Ecriture d'un sous-programme :**

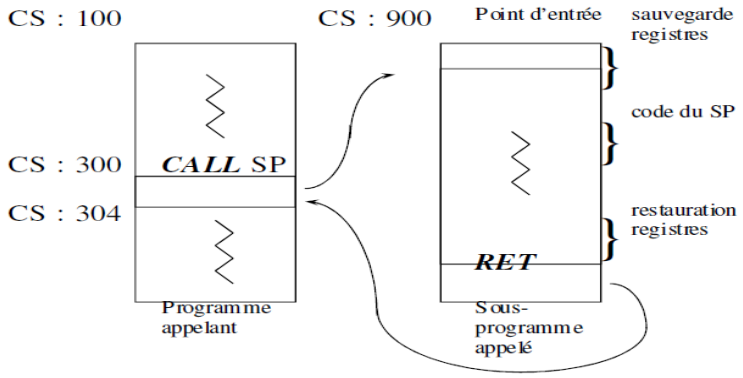
```
nom_sp    PROC
            ⋮   ← instructions du sous-programme
          ret   ← instruction de retour au programme principal
nom_sp    ENDP
```

✓ **Appel** d'un sous-programme par le programme principal : **CALL** procédure

```
      ⋮   ← instructions précédant l'appel au sous-programme
call nom_sp   ← appel au sous-programme
      ⋮   ← instructions exécutées après le retour au programme principal
```

✓ Lors de l'exécution de l'instruction **CALL**, le pointeur d'instruction **IP** est chargé avec l'adresse de la première instruction du sous-programme. Lors du retour au programme appelant, l'instruction suivant le **CALL** doit être exécutée, c'est-à-dire que **IP** doit être rechargé avec l'adresse de cette instruction.

✓ Avant de charger **IP** avec l'adresse du sous-programme, l'**adresse de retour** au programme principal, c'est-`a-dire le **contenu de IP**, est sauvegardée dans une zone mémoire particulière appelée **pile**. Lors de l'exécution de l'instruction **RET**, cette adresse est **récupérée** à partir de la **pile** et **rechargée dans IP**, ainsi le programme appelant peut se poursuivre.




**Remarque :** la **valeur de SP** doit être **initialisée par le programme principal** avant de pouvoir utiliser la pile.

✓ **Classification des paramètres :** Les paramètres peuvent être en entrée, en sortie ou en entrée-sortie, ils sont classés : **par valeur, par référence**

➢ **Passage de paramètres par valeur :**

- Recopie de la valeur à transmettre dans une zone mémoire connue du sous-programme
- Travaille sur une copie de la variable => la variable du Programme principal ne varie pas (protection)

Exemple

Prog. Principal

```
•••
MOV   AX, 4   } Paramètres
MOV   BX, 5   }
call Somme  ⟶ Appel
•••
-> Résultat dans AX
```

Sous-Prog.

```
Somme PROC NEAR
      ADD   AX, BX
      RET
Somme ENDP
```

➢ **Passage par référence : transmission de l'adresse de la variable :**

- Plus rapide
- Le sous programme travaille sur les variables du Programme principal => ces variables ont été modifiées par le sous programme.
- Les variables ne seront pas protégés dans ce cas.

Exemple

Prog. Principal

```
•••
MOV   CX, 10 ; taille de TAB
LEA   BX, TAB  ; BX<- @TAB
call Affiche
•••
```

Sous-Prog.

```
Affiche PROC NEAR
       MOV DI, 0
boucle :
       MOV AL,[BX+DI]
       call COUT
       INC DI
LOOP boucle
       RET
Affiche ENDP
```

## 9. Interruptions

Pour faire des **entrées/sorties** (essentiellement avec l'écran et le clavier), on passe par des interruptions du **BIOS** ou du **DOS**. Nous n'allons voir ici que ce dont nous avons besoin.

### *9.1 L'interruption 10h du BIOS*

Le BIOS est de relativement bas niveau et dépend fortement de la machine.
L'interruption 10h peut effectuer beaucoup de fonctions différentes, le numéro de la fonction désirée doit être place dans AH avant l'appel de l'interruption. Nous ne parlerons ici que de quelques fonctions :

✓ **Fonction 00**

Cette fonction permet de choisir un mode texte ou un mode graphique. En changeant de mode, on peut effacer l'écran, ce qui fait que l'on peut appeler cette fonction pour effacer l'écran et rester dans le même mode.
Paramètres : AH = 00

✓ **Fonction 09**

Cette fonction permet d'écrire un caractère : Permet les répétitions, Gère la couleur en mode texte et en mode graphique
Paramètres : AH = 09h

✓ **Fonction 0Eh**

Cette fonction permet d'écrire un caractère : Fonctionne en mode graphique, Gère le curseur, Gère la couleur seulement en mode graphique. Seule la couleur du caractère est gérée, la couleur du fond n'est pas gérée.
Paramètres : AH = 0Eh

✓ **Fonction 05**

Cette fonction permet de sélectionner la page active de l'affichage.
Paramètres : AH = 05h
AL = numéro de la page

### *9.2. L'interruption 21h du DOS*

Appel au BIOS qui fonctionne à un niveau plus proche de la machine. **L'interruption 21h** peut réaliser plusieurs fonctions différentes. Nous ne citerons ici que celles que nous utiliserons :

✓ **Fonction 02**

Cette fonction permet d'écrire un caractère. Le caractère est envoyé vers la sortie standard, l'écriture peut donc être redirigée dans un fichier.

Paramètres : AH = 02h DL = Caractère à écrire

✓ **Fonction 09**

Cette fonction permet en un seul appel, d'écrire une suite de caractères.

Paramètres : AH = 09h
DX = Adresse de la chaîne de caractères ; La chaîne doit être terminée par le caractère $

✓ **Fonction 07**

Cette fonction permet de lire un caractère du clavier sans qu'il n'y ait d'écho à l'écran.
Paramètre passé : AH = 07

Paramètre retourné : AL = caractère lu

Les touches fonction retourne 2 caractères, d'abord un octet nul, puis le code étendu de la touche, il faut donc faire 2 appels consécutifs de la fonction 07.

### ✓ Fonction 0Ah

Permet de saisir une chaîne de caractère au clavier. La saisie s'arrête quand on tape la touche de retour, le caractère CR (13) est mémorisé avec la chaîne

**Quelques interruptions**

| | | AH | AL | BH | BL | CX | DH | DL | commentaire |
|---|---|---|---|---|---|---|---|---|---|
| INT 10h | Mode écran | 00 | mode | | | | | | |
| | Ecrit char | 09 | car | page | coul t/g | rep | | | !Curs !CarSpec |
| | Ecrt char | 0A | car | page | | rep | | | !Curs !CarSpec |
| | Ecrt char | 0E | car | | Coul g | | | | Curs CarSpec |
| | Pos Curs | 02 | | page | | | ligne | col | |
| | Choisir page | 05 | page | | | | | | |
| | Allumer pixel | 0Ch | couleur | page | | x | y | | |
| | 50 lignes | 11h | 12h | | 30h | | | | |
| INT 21h | Ecrt char | 02 | | | | | | car | ^C^B -> int 23h |
| | Ecrit chaîne | 09 | | | | | adresse | | '$'= fin chaîne |
| | Lit car !écho | 07 | Car lu | | | | | | |
| | Lit chaine | 0Ah | | | | | adresse | | Voir cours |
| | kbhit | 0B | 0 : non<br>FF : oui | | | | | | |

# Chapitre 4 : Étude de cas : Processeurs 32 et 64 bits

## Introduction

Les deux architectures les plus populaires connues pour les microprocesseurs (puces) sont IA-32 et AMD64 ou 32 et 64 bits. L'architecture IA-32 a été développée dans les premiers temps du PC, cette architecture présentait certains inconvénients. La seconde, AMD64, est moderne et a été créée relativement récemment. Les utilisateurs qui viennent d'acquérir un ordinateur se demandent souvent ce qui est le mieux : 32 ou 64 bits ? Quelle architecture est préférable pour un PC ?

## 1. Introduction aux architectures 32 bits et 64 bits

### *1.1. Évolution des architectures (IA-32 à AMD64)*

#### 1.1.1. Qu'est-ce que le débit binaire ?

Le débit binaire de Windows fait référence à l'architecture du processeur, qui détermine le nombre de bits d'information que le processeur peut traiter en même temps. Il est courant de parler de versions 32 bits et 64 bits du système d'exploitation Windows.

- **Version 32 bits de Windows :** Les processeurs et les systèmes d'exploitation dotés d'une architecture 32 bits peuvent traiter des informations sur 32 bits à la fois. Cela signifie qu'ils peuvent adresser jusqu'à 4 giga-octets de mémoire vive et exécuter des programmes 32 bits.
- **Windows 64 bits :** les processeurs et les systèmes d'exploitation de l'architecture 64 bits peuvent traiter des informations sur 64 bits à la fois. Cela leur permet d'adresser beaucoup plus de mémoire vive (plus de 4 giga-octets) et d'offrir de meilleures performances lors de l'exécution d'applications 64 bits.

**Remarque :** Le choix entre les versions 32 bits et 64 bits de Windows dépend de vos exigences en matière d'utilisation du processeur et de la mémoire vive. Actuellement, la plupart des nouveaux ordinateurs et portables sont livrés avec Windows 64 bits en raison de sa capacité à utiliser efficacement de grandes quantités de mémoire vive et à prendre en charge les applications modernes.

#### 1.1.2. Présentation de l'architecture 32 et 64 bits

- **L'architecture 32 bits**

L'architecture 32 bits ou x86 ou IA-32 est en fait une architecture de processeur identique avec laquelle le système d'exploitation est conçu pour fonctionner. x86 a été utilisé pour la première fois pour les puces fabriquées par Intel. L'architecture a reçu le nom correspondant en raison des premiers processeurs avec lesquels elle a été utilisée - Intel 80386. Par la suite, elle a trouvé sa place chez AMD et x86 est devenu un standard pour les PC. L'architecture a été constamment améliorée et finalisée.

- **L'architecture 64 bits**

L'architecture 64 bits est apparue plus tard qu'AMD. Elle est également appelée x86-64 ou amd64. Elle fonctionne avec les puces Intel et AMD. En même temps, elle est considérée

comme entièrement compatible avec l'architecture x32. La principale différence entre les deux est le nombre de bits.

### 1.1.3. La différence entre l'architecture 32 et 64 bits

Un microprocesseur (une puce) est le composant le plus important de l'ordinateur. Il est en quelque sorte le cerveau du PC. Le processeur gère les données à traiter, contrôle les périphériques externes, leur envoie des commandes, reçoit des données et interagit avec la mémoire. Toutes les adresses et instructions exécutées par le processeur doivent être stockées dans la mémoire.

Une puce de processeur est nécessaire pour accomplir ces tâches. Quelle est la différence entre 32 bits et 64 bits ? Dans les processeurs 32 bits, la taille des cellules est de 32 bits. Dans les puces à architecture 64 bits, la taille est de 64. Plus la cellule est grande, plus elle peut contenir de données.

Les puces à architecture 32 bits sont limitées en ce sens qu'elles ne peuvent obtenir des adresses que dans la limite de $2^{32}$ degrés. Une adresse plus grande ne tient tout simplement pas dans la cellule. Vous ressentirez cette caractéristique de manière particulièrement aiguë sur la RAM. Puisque seulement $2^{32}$ bits ou 4 GB de mémoire sont dans ces limites, et si c'est plus, la puce ne sera pas capable de faire face sans une émulation spéciale par le système d'exploitation.

Une puce avec une taille de registre de 64 bits est déjà orientée pour travailler avec des adresses de $2^{64}$. Si nous traduisons cette valeur de la manière habituelle, nous verrons qu'elle est de 1 milliard de Go. Il est intéressant de noter qu'aucun système d'exploitation moderne ne peut prendre en charge une telle quantité de mémoire vive. Même le populaire Linux n'est pas conçu pour cela.

Mais les différences ne s'arrêtent pas là. Au cours d'un processus, une puce dotée d'un système x32 bits peut traiter 32 bits ou 4 octets de données, 1 octet étant égal à 8 bits. Si la taille des données est supérieure à 4 octets, la puce doit exécuter plusieurs cycles simultanément pour les traiter. Si la puce est de 64 bits, la taille des données à traiter sera multipliée par deux et sera égale à 8 octets. Et ce, bien que cette taille soit supérieure à 8 octets. La puce devra consacrer moins de temps à la tâche à accomplir, à savoir le traitement des données.

## *1.2.Différences principales : Quelle est la meilleure taille de bit ?*

La question de savoir quelle taille de bit du système d'exploitation est la meilleure dépend de vos besoins et des exigences de performance de votre ordinateur. Voici quelques facteurs qui peuvent vous aider à prendre une décision :

- ✓ ***Performances :*** les systèmes 64 bits sont capables de traiter de grandes quantités de données de manière plus efficace, ce qui peut améliorer les performances lors de l'exécution de tâches et d'applications complexes.
- ✓ ***Prise en charge de la mémoire :*** un système 64 bits peut adresser beaucoup plus de mémoire vive qu'un système 32 bits. Si vous devez utiliser de grandes quantités de mémoire vive (plus de 4 Go), un système 64 bits sera un meilleur choix.

- ✓ ***Compatibilité logicielle :*** certaines applications peuvent être optimisées pour fonctionner sur des systèmes 64 bits, ce qui peut affecter leurs performances et leurs fonctionnalités.

- ✓ ***Mises à jour futures :*** à mesure que la technologie et les logiciels évoluent, il est probable que de plus en plus d'applications fonctionneront mieux sur les systèmes 64 bits.

**Remarque :** Si vous avez la possibilité d'utiliser une version **64 bits** de Windows et que votre ordinateur prend en charge cette capacité de bits, il s'agit probablement ***d'un meilleur choix en raison de l'amélioration des performances et de la possibilité d'utiliser plus de mémoire vive***.

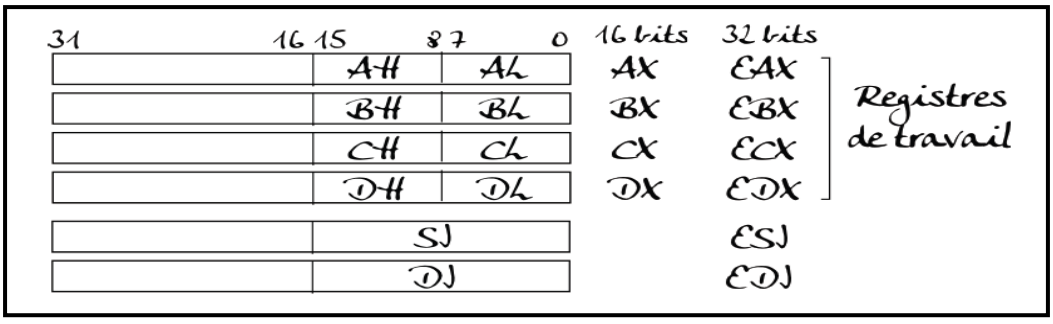
## 2. Registres dans les processeurs 32 et 64 bits

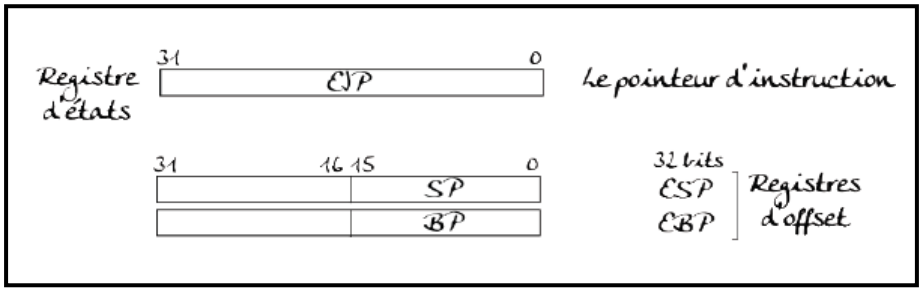
Le modèle de programmation du 80386 et des versions supérieures contient des registres étendus de 8, 16 et **32 bits**, ainsi que deux ***registres de segments*** supplémentaires de **16 bits** : **FS** et **GS**.
Certaines parties des registres sont accessibles à partir des identifiants suivants :

| | |
|---|---|
| rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp, r8, r9, ... ,r15 | 64 bits |
| eax, ebx, ecx, edx, edi, esi, ebp, esp, r8d, r9d, ... ,r15d | 32 bits |

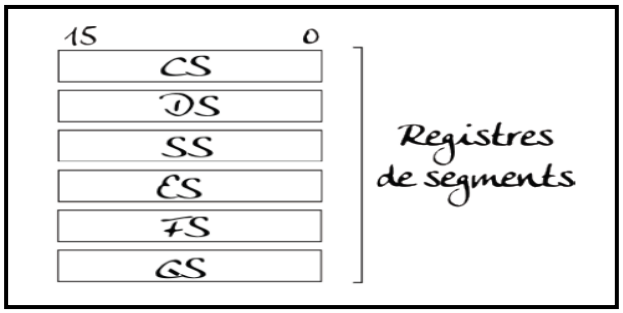
### *2.1. Les principaux registres 32 bits*

- ➢ Les processeurs 80386 et plus récents ont des registres étendus. Par exemple, le registre AX 16 bits est étendu à 32 bits.
- ➢ Les registres 16 bits sont AX, BX, CX, DX, SP, BP, DI et SI
- ➢ Les registres étendus 32 bits sont EAX, EBX, ECX, EDX, ESP, EBP, EDI et ESI.
- ➢ Pour la compatibilité ascendante :
  - AX, AL et AH existent toujours,
  - font référence à des parties de **EAX** (accumulateur),
  - pas d'accès aux 16 bits de poids fort de EAX



- Les **registres de segment** sont toujours sur **16 bits** dans le 80386, et ***2 nouveaux registres temporaires de segment*** : **FS et GS**.




### *2.2. Les principaux registres 64 bits*

- Les registres **64 bits** d'un ***Pentium 4*** avec extensions **64 bits** sont **RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI** et R8 à R15.
- La plupart des autres registres sont également étendus.
  - BP devient **EBP** ; SP devient ESP ; FLAGS devient **EFLAGS**
  - et IP devient **EIP**.
  - Mais plus d'accès aux registres 16 bits en mode protégé **32 bits**

| | |
|---|---|
| **rax** | registre général, accumulateur, contient la valeur de retour des fonctions |
| **rbx** | registre général |
| **rcx** | registre général, compteur de boucle |
| **rdx** | registre général, partie haute d'une valeur 128 bits |
| **rsi** | registre général, adresse source pour déplacement ou comparaison |
| **rdi** | registre général, adresse destination pour déplacement ou comparaison |
| **rsp** | registre général, pointeur de pile (stack pointer) |
| **rbp** | registre général, pointeur de base (base pointer) |
| **r8** | registre général |
| **r9** | registre général |
| ⋮ | |
| **r15** | registre général |
| **rip** | compteur de programme (instruction pointer) |

- **Le registre RFLAGS**
  - ✓ Le registre RFLAGS contient des informations concernant le résultat de l'exécution d'une instruction.
  - ✓ Certains bits du registre appelés drapeaux ont une signification particulière.
  - ✓ Seuls les 32 bits de la partie EFLAGS sont utilisées.
  - ✓ De plus, le microprocesseur contient un pointeur d'instruction (IP/EIP/RIP) et un registre d'indicateurs (FLAGS, EFLAGS ou RFLAGS).
  - ✓ Toutes les adresses mémoire en mode réel sont une combinaison d'une adresse de segment et d'une adresse de déplacement.

| | |
|---|---|
| CF Carry Flag (bit 0) | retenue |
| PF Parity Flag (bit 2) | |
| AF Auxiliary Carry Flag (bit 4) | |
| ZF Zero Flag (bit 6) | vaut 1 lorsque le résultat est 0 |
| SF Sign Flag (bit 7) | bit de signe du résultat |
| OF Overflow Flag (bit 11) | dépassement, le résultat contient trop de bits |
| DF Direction Flag (bit 10) | sens d'incrémentation de ESI et EDI |
| TF Task Flag (bit 8) | active la gestion de tâche en mode protégé |
| IF Interrupt Flag (bit 9) | interruption |
| IOPL I/O Privilege Level (bits 12 et 13) | |
| NT Nested Task (bit 14) | |
| RF Resume Flag (bit 16) | active le mode debug |
| VM Vitrual 8086 Mode (bit 17) | |
| AC Alignement Check (bit 18) | |
| VIF Virtual Interrupt Flag (bit 19) | |
| VIP Virtual Interrupt Pending (bit 20) | |
| ID Identification Flag (bit 21) | |

## 3. Modes d'adressage avancés

### *3.1. Passage de l'adressage segmenté à l'adressage plat*

Bien qu'un microprocesseur 64 bits puisse a priori manipuler des adresses physiques sur 64 bits, permettant d'accéder à 64 GO de mémoire vive. Le modèle de mémoire utilisé dans l'architecture 64 bits est ce qui appelé modèle plat (flat mode memory en anglais) dans lequel il n'y a pas de segmentation : l'adresse physique du premier octet de la mémoire est : 00 0000 0000H et celle du dernier est FF FFFF FFFF H, l'adresse est de 40 bits. Les registres de segment n'ont donc plus d'utilité a priori pour adresser un emplacement en mémoire. En fait le registre de segment de code CS est encore utilisé pour spécifier un descripteur précisant les droits d'accès (mais non son adresse de base et sa limite). Les données et la pile sont contenues dans le segment de code.

Le système en mode réel n'est pas disponible si le processeur fonctionne en mode 64 bits.

- La protection et la pagination sont autorisées en mode 64 bits.
- **Remarque :** Le mécanisme de ***pagination mémoire*** permet d'attribuer n'importe quel emplacement mémoire physique à n'importe quelle adresse linéaire.
  • Une adresse linéaire est définie comme l'adresse générée par un programme.
  • L'adresse physique est l'emplacement mémoire réel auquel accède un programme.
  • Grâce à la pagination mémoire, l'adresse linéaire est traduite de manière invisible en n'importe quelle adresse physique.

- Le registre CS est toujours utilisé en mode protégé en mode 64 bits.
- La plupart des programmes actuels fonctionnent en mode compatible IA32.
  - ✓ Les logiciels actuels fonctionnent correctement, mais cela changera dans quelques années avec l'augmentation de la mémoire et l'arrivée massive d'ordinateurs 64 bits.

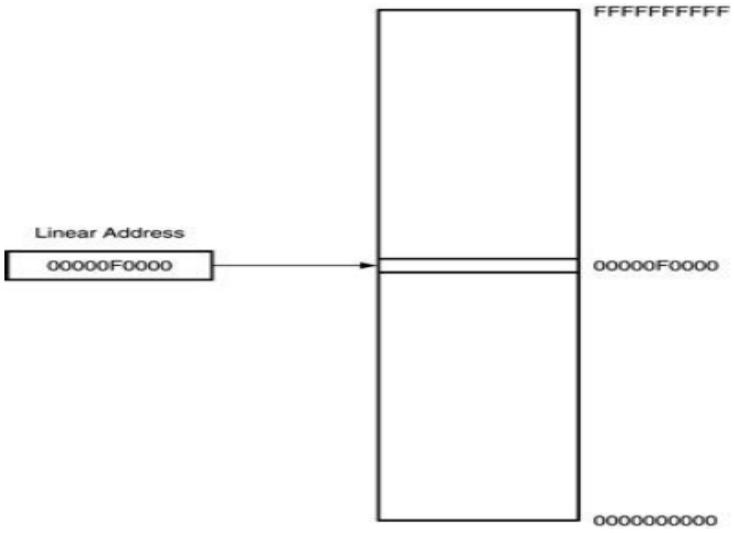



**Figure 4.1. Le modèle de mémoire en mode plat 64 bits [17].**

### *3.2. Adressage mémoire en mode réel*

Les processeurs 80286 et supérieurs fonctionnent en mode réel ou protégé :

- Le fonctionnement en mode réel permet d'adresser uniquement le premier Mo d'espace mémoire, même sur les microprocesseurs Pentium 4 ou Core2 (l'architecture 64 bits).
- Le premier Mo de mémoire est appelé mémoire réelle, mémoire conventionnelle ou mémoire système DOS.

### *3.3. Segments et Offsets (spécificités 64 bits)*

Toutes les **adresses mémoire en mode réel** doivent être composées **d'une adresse de segment et d'une adresse de déplacement (Offset).**

- ✓ **L'adresse de segment :** définit l'adresse de **début de tout segment mémoire** de **64 Ko.**
- ✓ **L'adresse de déplacement (Offset) :** sélectionne **n'importe quel emplacement** dans le segment mémoire de 64 Ko.

• La figure 4.2 illustre ***comment le schéma d'adressage segment plus déplacement sélectionne un emplacement mémoire :***

- ✓ ***Ceci montre un segment de mémoire commençant à 10 000 H et se terminant à l'emplacement IFFFFH. Longueur : 64 Ko***

- ✓ ***Montre également comment une adresse de Offset, appelée déplacement, de F000H sélectionne l'emplacement 1F000H dans la mémoire.***

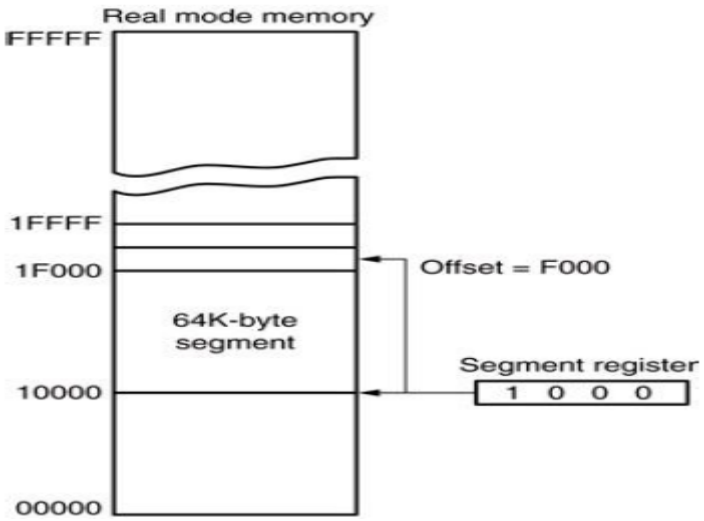



**Figure 4.2. Le schéma d'adressage de la mémoire en mode réel, utilisant une adresse de segment plus un décalage [17].**

- L'emplacement de départ **d'un segment** est défini par le nombre de **16 bits** du registre de segment, **suivi d'un zéro hexadécimal** à son extrémité droite.
- **L'adresse de déplacement (offset)** : est **un nombre de 16 bits *ajouté* à l'adresse de segment de 20 bits** pour former **l'adresse mémoire en mode réel**.
- L'adresse de déplacement est toujours ajoutée à l'adresse de début du segment pour localiser les données. L'adresse de segment et l'adresse de déplacement sont parfois écrites sous la forme 1000:2000⇨Une adresse de segment de 1000H ; un décalage de 2000H
- **L'accès à toutes les instructions (code)** se fait par **la combinaison de CS (adresse de segment)** et **IP ou EIP (adresse de déplacement)**.

## 4. Jeu d'instructions

### *4.1. Unités vectorielles*

Les unités vectorielles permettent de vectoriser le code, en d'autres termes, de le paralléliser au sein du microprocesseur. On exécutera la même instruction sur plusieurs données différentes stockées dans un registre de type MMX (64 bits), SSE (128 bits) ou AVX (256 bits).

- Par exemple avec la technologie SSE, au lieu d'écrire :

```
float v1[4], v2[4], v3[4];

void vector_sum(float *x, float *y, float *z, int size) {
    for (int i = 0; i < 4; ++i) {
        z[i] = x[i] + y[i]
    }
}

vector_sum(v1, v2, v3, 4);
```

On réalise une seule opération en parallèle sur un registre capable de contenir 4 floats, ce que l'on note :

```
z[0:3] = x[0:3] + y[0:3]
```

La notation x[0:3] symbolise x[0] à x[3], elle n'est pas utilisable en langage C, elle nous permet seulement d'exprimer de manière concise le traitement réalisé.
On parle alors de traitement SIMD « Single Instruction Multiple Data », cela signifie que la même instruction est appliquée sur des données différentes et pour que cela ait un intérêt en termes de performance, on réalise les calculs **en parallèle** et non pas de manière séquentielle.
Par la suite nous allons nous intéresser aux technologies SSE et AVX, notons également que nous allons suivre la convention de représentation Intel pour les instructions SSE qui consiste à écrire les valeurs d'un vecteur en mémoire ou d'un registre en plaçant la partie haute à gauche et la partie basse à droite.

### *4.2. Particularités et extensions modernes : SSE*

➢ **Spécificités**

Sous le sigle SSE (Streaming SIMD Extensions) nous réunissons tous les jeux d'instructions successifs SSE, SSE2, SSE3, SSSE3, SSE4A, SSE4.1 et SSE4.2.
La première version du SSE est un jeu de 70 instructions apparu en 1999 sur le Pentium III en réponse a la technologie 3DNow! D'AMD, née un an plus tôt. Ces instructions traitent des entiers ou des réels. Les versions suivantes ont apporte de nouvelles instructions de manipulation des données ou de calcul comme par exemple dpps (Dot Product of Packed Single Precision Floating-Point Values) du jeu d'instructions SSE4.1 qui réalise le produit scalaire de deux vecteurs.
Sur les Pentium III notamment, l'efficacité du SSE était nettement moindre que sur son successeur, le Pentium 4, car bien que le Pentium III disposant des registres de stockage de 128 bits, il ne possédait que de registres 64 bits pour réaliser les calculs.
De ce fait, une instruction SSE était traitée en deux fois 64 bits, on commençait par traiter la partie basse, puis la partie haute, ce qui est moins efficace que de traiter 128 bits en une seule fois.
En architecture 32 bits, il existe 8 registres SSE de 128 bits nommés **xmm0** à **xmm7**. Ce nombre de registres est doublé en architecture 64 bits avec l'ajout des registres **xmm8** à **xmm15**.
Les registres SSE possèdent des instructions qui traitent les valeurs qu'ils contiennent:

✓ soit sous forme d'entiers au format :
- 16 octets
- 8 mots
- 4 double mots (4 entiers 32 bits signés ou non)
- 2 quadruples mots (2 entiers 64 bits signés ou non)
- double quadruple mot (double quad word) soit un total de 128 bits

✓ soit sous forme de nombres a virgule flottante (32 et 64 bits)
- 4 flottants simple précision (float)
- 2 flottants double précision (double)

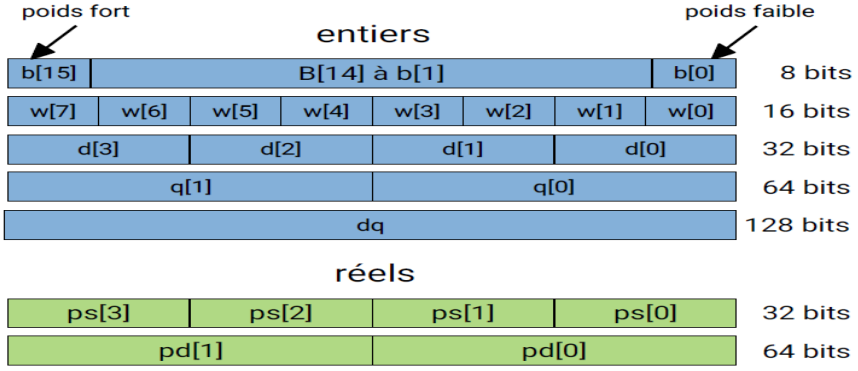



**Figure 4.3. Types de données contenues dans un registre SSE [17].**

On disposera donc de plusieurs instructions similaires mais avec des mnémoniques différents en fonction que l'on traite des entiers ou des flottants. La grande majorité de ces instructions seront suffixées par une a deux lettres (voir figure 4.4) qui correspondent au type de donnée manipulée.
Ainsi, l'instruction **paddb** réalise une addition entière en parallèle entre les 16 octets de ses deux opérandes, alors que **paddb**, **paddw**, **paddd** réalise une addition entière en parallèle sur 4 entiers 32 bits. De la même manière **addps**, **addpd** réalise une addition en parallèle sur 4 flottants en simple précision et **addpd** traite 2 flottants en double précision.
On note également pour les flottants les suffixes **ss** et **sd** qui ne traitent que la partie basse du registre SSE (respectivement 32 et 64 bits). Ces instructions liées à des flottantes simples ou doubles précisions permettent de remplacer la FPU car en 64 bits les paramètres de type float ou double sont passes dans les registres SSE et les calculs sont réalises avec ces mêmes registres.

| Type | Taille en octets | Nom | Quantité dans 128 bits | Suffixe |
|---|---|---|---|---|
| entier | 1 | byte | 16 | b |
| entier | 2 | word | 8 | w |
| entier | 4 | double word | 4 | d |
| entier | 8 | quad word | 2 | q |
| flottant | 4 | float | 4 | ps |
| flottant | 8 | double | 2 | pd |
| flottant | 4 | float | 1 | ss |
| flottant | 8 | double | 1 | sd |

**Figure 4.4. Suffixes des instructions SSE [17].**

Il n'est pas possible de manipuler un registre vectoriel en faisant directement référence à son $i^{ème}$ élément (sauf pour des instructions utilisant un masque de sélection) mais afin de simplifier la compréhension de certaines instructions et traitements nous introduisons la notation suivante qui nous permettra de décrire le comportement des instructions SSE et AVX sous forme de petits programmes C :
**xmm0.T[i]** ou : T représente le type (b, w, d, q, ps, pd, présenté figure 4.4) et i le $i^{ème}$ élément. Ainsi **xmm0.b[15]** représente le dernier octet du registre **xmm0**, donc l'octet de poids fort, l'octet de poids faible étant **xmm0.b[0]**.

### ➢ Chargement et stockage des données

Le chargement des données vers les registres SSE ou le stockage des valeurs contenues dans les registres vers la mémoire se font a l'aide des instructions de déplacement de type *mov*.
Pour les entiers, on utilisera **movdqu** (*MOV Double Quad word Unaligned*) ou **movdqa, movdqu** (*MOV Double Quad word Aligned*). Dans le cas du SSE les données sont alignées si l'adresse depuis laquelle on lit ou on écrit est un multiple de 16.
Le format des instructions de chargement de données est de la forme :

```
1    movdqa  xmm1, [ebx]            ; opérande SSE et référence
2                                   ; mémoire (Load)
3    movdqa  [edi + ecx * 4], xmm7  ; idem (Store)
4    movdqa  xmm3, xmm1             ; deux opérandes SSE
```

Pour les flottants, on utilisera les instructions **movups** ou **movaps** qui fonctionnent sur le même modèle.

Cependant, on notera que l'on peut utiliser **movdqa** (ou **movdqu**) avec des flottants et **movaps, movups** (ou **movups**)avec des entiers puisqu'il n'y a à priori aucune conversion ou modification des données, on se contente de lire les données et les stocker dans un registre ou en mémoire.
Enfin il existe des instructions qui ne traitent que la partie basse du registre SSE comme **movd** pour les entiers et **movss, movsd** pour les flottants simple et double précision :

```
mov     eax, 0x01010101
movd    xmm1, eax       ; xmm1.d[0] = 0x01010101, xmm1.d[1:3] = ?
movss   xmm2, [edi]     ; xmm2.ps[0] = [edi], xmm2.ps[1:3] = ?
movsd   xmm2, [edi]     ; xmm2.pd[0] = [edi], xmm2.pd[1] = ?
```

On charge ici la valeur héxadécimale sur 32 bits 0x01010101 dans la partie basse du registre **xmm1**, les 3 autres valeurs 32 bits ne sont pas modifiées.

➢ **Instructions arithmétiques**

Pour les entiers, on utilisera les instructions **padd** pour l'addition, **psub** pour la soustraction et **pmull** pour la multiplication, suffixées par la quantité traitée.
Notons qu'**il n'existe pas** d'instruction **pdiv** qui réaliserait une division entière.
Pour les flottants, on trouve les instructions **addps**, **subps**, **mulps**, **divps** ainsi que **addpd** et consorts.
Il existe également des instructions comme **addsubps xmm1**, **xmm2** dont le comportement est le suivant :

```
xmm1.ps[0] -= xmm2.ps[0]
xmm1.ps[1] += xmm2.ps[1]
xmm1.ps[2] -= xmm2.ps[2]
xmm1.ps[3] += xmm2.ps[3]
```

Et **haddps xmm1**, **xmm2** qui réalise une addition dite ***horizontale***.

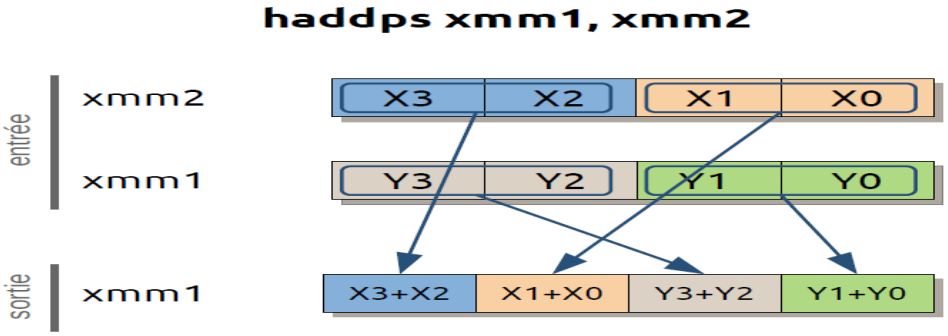



**Figure 4.5. Instruction haddps [17].**

```
; haddps xmm1, xmm2
; on utilise un registre temporaire xmmt
xmmt.ps[0] = xmm1.ps[0] + xmm1.ps[1]
xmmt.ps[1] = xmm1.ps[2] + xmm1.ps[3]
xmmt.ps[2] = xmm2.ps[0] + xmm2.ps[1]
xmmt.ps[3] = xmm2.ps[2] + xmm2.ps[3]
xmm1 = xmmt
```

L'intérêt de l'instruction **haddps** (Voir Figure. 4.5) est qu'elle permet de faire la somme des quatre valeurs flottantes simple précision contenues dans un registre SSE [2] en procédant ainsi:

On réalise deux fois l'addition horizontale d'un registre avec lui même. Au final on obtient :

```
xmm1.ps[0:3] = xmm1.ps[0] + xmm1.ps[1] + xmm1.ps[2] + xmm1.ps[3]
```

On trouve également **phaddw** et **phaddd** pour les entiers 16 et 32 bits respectivement qui réalisent l'addition horizontale de mots et double mots.

### ➢ Instructions de réarrangement

Les instructions **pshufd** pour les entiers et **shufps** pour les flottants permettent de sélectionner ou réorganiser les données au sein d'un registre SSE mais ont un comportement différent. La plupart de ces instructions utilisent un troisième opérande qualifié de masque et notée **imm8** ce qui signifie qu'il s'agit d'une constante sur 8 bits et elle est utilisée pour indiquer quels champs sélectionner.
Par exemple **pshufd xmm1, xmm2, imm8**, qui est présentée Figure 4.6, réalise une sélection et réorganisation des valeurs de **xmm2** vers **xmm1** :

```
; pshufd xmm1, xmm2, imm8
xmm1.ps[0] = xmm2.ps[ imm8 & 0x03 ];
xmm1.ps[1] = xmm2.ps[ (imm8 >> 2) & 0x03 ];
xmm1.ps[2] = xmm2.ps[ (imm8 >> 4) & 0x03 ];
xmm1.ps[3] = xmm2.ps[ (imm8 >> 6) & 0x03 ];
```

L'utilisation de cette instruction sur la même opérande avec un masque de 0 (**pshufd xmm1, xmm1, 0**) a pour effet de recopier la valeur **xmm1.d[0]** dans **xmm1.d[1:3]**. Au final on obtient donc quatre fois la même valeur dans **xmm1**.
On peut bien entendu l'utiliser pour des flottants simple précision car l'instruction **shufps**, qui possède la même syntaxe, prend en considération **xmm1** et **xmm2** pour la sélection des valeurs mais possède un comportement quelque peu différent :

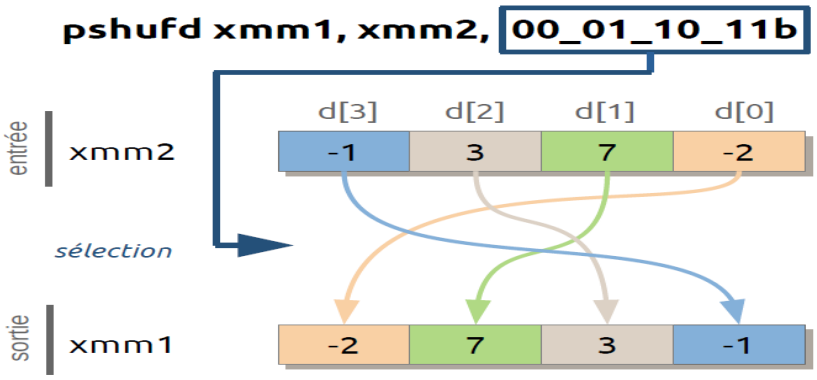



**Figure 4.6. Instruction PSHUFD [17].**

Une autre instruction intéressante est **blendps**, mais elle n'utilise que les 4 premiers bits de la constante **imm8**. Elle permet de remplacer les valeurs du registre de destination par des valeurs du registre source :

```
// blendps xmm1, xmm2, imm8
for (int index = 0; index <= 3; ++index) {
    xmm1.ps[ index ] = (imm8 & (1 << index)) == 0
        ? xmm1.ps[ index ] : xmm2.ps[ index ];
}
```

Ainsi, le code suivant remplacera **xmm1.ps[1]** par **xmm2.ps[1]** :

```
blendps xmm1, xmm2, 00000010b
```

Une instruction très utile est **pblendvb** (*Variable Blend Packed Bytes*). Elle travaille sur les octets d'un registre SSE et utilise par défaut un masque de sélection base sur le registre **xmm0** :

```
// pblendvb xmm1, xmm2
int i, byte;

for (byte = 0, i = 7; i <= 127; i += 8, ++byte) {
  xmm1.b[ byte ] = (xmm0.bits(i) == 1) ? xmm2.b[ byte ] :
    xmm1.b[ byte ];
}
```

Elle permet de sélectionner les octets de **xmm1** ou de **xmm2** en fonction des octets de poids fort de **xmm0** positionnés a 0 ou 1. Dans la même veine, mais pour les valeurs flottantes, on trouve **blenvps** (*Variable Blend Packed Single Precision*) :

```
blendvps xmm1, xmm2    <xmm0>
```

Il existe une série d'instructions **vpbroadcast(b,w,d,q)** qui permettent de recopier une valeur dans plusieurs emplacements d'un registre SSE ou AVX. Par exemple **vpbroadcastb xmm1, xmm1** recopie le premier octet du registre **xmm1** dans les 15 autres emplacements du registre:

```
// vpbroadcastb xmm1, xmm1
for (int index = 1; index <= 15; ++index) {
  xmm1.b[ index ] = xmm1.b[ 0 ];
}
```

Enfin, l'instruction **insertps xmm1, xmm2, imm8** réalise plusieurs opérations :
**1.** en premier lieu, elle sélectionne l'une des 4 valeurs de la source **xmm2** grâce aux bits 6 et 7 de la constante imm8
**2.** elle recopie ensuite cette valeur dans **xmm1** à la position indiquée par les bits 4 et 5 de imm8
**3.** elle met enfin, en fonction des bits 0 a 3 positionne a 1 de imm8, les valeurs correspondantes dans **xmm1** a 0
Le code qui suit donne comme résultat un registre **xmm1** contenant les valeurs [7.0, 3.0, 0.0, 0.0].

```
section .data
    a    dd 1.0, 2.0, 3.0, 4.0
    b    dd 5.0, 6.0, 7.0, 8.0

section .text
    movups    xmm1, [a]
    movups    xmm2, [b]
    insertps xmm1, xmm2, 10_11_0011b
```

On commence par charger dans **xmm1** le vecteur [4.0, 3.0, 2.0, 1.0], puis dans **xmm2** le vecteur [8.0, 7.0, 6.0, 5.0]. On choisit alors la valeur d'indice $10_b$ de **xmm2**, c'est à dire 7.0 et on la recopie en position $11_b$ de **xmm1**. La partie basse de la constante **imm8**, soit $0011_b$ indique que les valeurs d'indices 0 et 1 de **xmm1** doivent être mises à zéro.

### *4.3. Particularités et extensions modernes : AVX*

#### ➢ Spécificités

Sous le sigle AVX nous plaçons les jeux d'instructions AVX (*Advanced VectoreXtensions*) et AVX2 256 bits. Nous ne nous intéresserons qu'en fin de chapitre a l'AVX 512 bits. Tout comme en architecture 32 bits, il existe 8 registres AVX de 256 bits nommés **ymm0** a **ymm7**. Ce nombre de registres est doublé en architecture 64 bits avec l'ajout de **ymm8** a **ymm15**. Les principaux changements par rapport au SSE sont les suivants :

- ✓ Les instructions AVX commencent par la lettre **v** pour les distinguer des instructions SSE
- ✓ Les instructions AVX peuvent agir sur les registres **ymm** ou **xmm** et vont utiliser la même syntaxe
- ✓ Cependant, une instruction AVX peut prendre un opérande supplémentaire qui sera le registre de destination

Par exemple, en SSE, si on écrit **paddd xmm1, xmm2**, les quatre entiers de **xmm2** sont ajoutés à **xmm1**, en d'autres termes on a xmm1.d[0:3] += xmm2.d[0:3]. Les valeurs présentes dans **xmm1** sont donc perdues. On aura le même comportement si on utilise **vpaddd xmm1, xmm2**.

```
; avec deux opérandes
 paddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
vpaddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

Par contre, si on écrit **vpaddd xmm3, xmm1, xmm2**, le registre **xmm3** recevra le résultat de la somme de **xmm1** et **xmm2**. Les registres **xmm1** et **xmm2** ne seront donc pas modifies.

```
; avec trois opérandes
vpaddd  xmm3, xmm1, xmm2  ; xmm3.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

#### ➢ Partie haute

Certaines instructions, comme **insertps**, dont nous avons parle précédemment, travaillent uniquement avec la partie basse des registres AVX. Cela est du à la constante **imm8** qui interagit avec l'un des quatre flottants simple précision d'un registre SSE. L'extension AVX de cette instruction **vinsertps** ne permet pas d'identifier les flottants dans la partie haute d'un registre AVX.

Il est donc nécessaire pour transposer l'utilisation du SSE vers l'AVX de travailler sur la partie basse du registre AVX puis de déplacer la partie basse vers la partie haute. On dispose par exemple des instructions **vinsertf128**, **vextractf** ou **vpbroadcast** qui réalisent ces manipulations.

En particulier, l'instruction **vinsertf128 ymm3, ymm2, xmm1, 0/1** copie **ymm2** dans **ymm3** puis remplace la partie haute (1) ou la partie basse (0) de **ymm3** par les valeurs de **xmm1**.

**vextractf128 xmm1, ymm2, 0/1**, copie la partie basse (0) ou la partie haute (1) de **ymm2** dans **xmm1**.

La série d'instructions vpbroadcast(b/w/d/q) x/ymm, reg recopie les 8/16/32 ou 64 bits d'un registre général respectivement vers tous les octets, mot, double mots ou quadruples mots d'un registre SSE ou AVX.

Ainsi pour recopier 8 fois l'octet 0x85 dans le registre **ymm1**, on écrira :

```
mov          eax, 0x85    ; ou mov al, 0x85
vpbroadcastb ymm1, eax
```

#### ➢ Instructions singulières

Certaines instructions n'ont pas le même comportement en AVX et en SSE.

C'est le cas de **haddps** dont nous avons parlé déjà. Nous avons vu que l'utilisation de deux fois cette instruction sur le même registre permet de calculer la somme des quatre valeurs

qu'il contient. Malheureusement cela ne fonctionne pas avec les 8 valeurs 32 bits qui contient un registre **ymm** lorsque l'on utilise **vhaddps**. En effet, le code suivant :

```
section .data
    v    dd 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0

section .text

    vmovups     ymm0, [v]
    vhaddps     ymm0, ymm0
    vhaddps     ymm0, ymm0
    vhaddps     ymm0, ymm0
```

Produira successivement les résultats :

| Instruction | ymm0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| vmovups | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| vhaddps | 15 | 11 | 15 | 11 | 7 | 3 | 7 | 3 |
| vhaddps | 26 | 26 | 26 | 26 | 10 | 10 | 10 | 10 |
| vhaddps | 52 | 52 | 52 | 52 | 20 | 20 | 20 | 20 |

Or on aimerait obtenir la somme des valeurs c'est à dire 36. Il faut alors procéder comme suit:

```
    vhaddps ymm0, ymm0           ; ymm0 = [15, 11, 15, 11, 7, 3, 7, 3]
    vhaddps ymm0, ymm0           ; ymm0 = [26, ... , 26, 10, ..., 10]
    vextractf128  xmm1, ymm0, 1  ; xmm1 = [26, 26, 26, 26]
    addps   xmm0, xmm1           ; xmm0 = [36, 36, 36, 36]
```

On réalise la somme horizontale deux fois comme en SSE, puis on transfert la partie haute de **ymm0** vers **xmm1**. Il suffit alors d'additionner les deux registres **xmm0** et **xmm1** pour avoir dans **xmm0** le résultat escompte.

## 5. Cycle d'exécution d'une instruction

### *5.1. Étapes du pipeline pour une instruction donnée*

Afin d'améliorer la vitesse de traitement des instructions un mécanisme de pipeline a été mis en place. Il consiste à ne pas attendre que l'ensemble des étapes de traitement aient été réalisées avant de passer à l'instruction suivante. Pour cela on rend chaque étape de traitement indépendante. Une première instruction passe dans l'étape de chargement au temps t = 0, puis au temps t + 1, elle passe dans l'étape de décodage, pendant que l'instruction suivante passe dans l'étape de chargement et ainsi de suite.

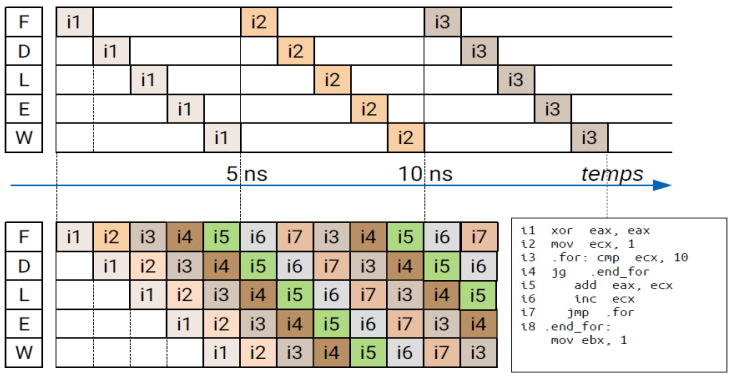



**Figure 4.7. Pipeline d'instructions [17].**

La question que l'on peut légitimement se poser est : quel gain apporte le pipeline ? Pour répondre à cette question il suffit de comparer les temps d'exécution avec et sans pipeline pour traiter n instructions :
• *sans pipeline* une instruction est exécutée toutes les 5 ns, si on à n instructions à exécuter il faut donc 5 × n ns.
• *avec pipeline*, il faut 5 ns pour que la première instruction soit exécutée, puis n − 1 ns pour exécuter les n − 1 instructions restantes
Le gain obtenu est donné par le rapport du temps d'exécution sans pipeline par le temps d'exécution avec pipeline :

$$\text{gain} = \lim_{n\to\infty}\left(\frac{5n}{5+n-1}\right) \simeq \frac{5n}{n} \simeq 5$$

| Micro architecture | Pipeline | Micro architecture | Pipeline |
|---|---|---|---|
| P5 (Pentium) | 5 | NetBurst (Cedar Mill) | 31 |
| P6 (Pentium 3) | 10 | Core | 14 |
| P6 (Pentium Pro) | 14 | Sandy Bridge | 14 |
| NetBurst (Willamette) | 20 | Haswell | 14 |
| NetBurst (Northwood) | 20 | Skylake | 14 |
| NetBurst (Prescott) | 31 | Kabylake | 14 |

**Figure 4.8. Nombre d'étages de pipeline pour différentes architectures Intel [17].**

Un pipeline de k étapes (on parle également d'étages ou *stages* en anglais), permet théoriquement de diviser le temps de traitement par k. Cependant, le nombre d'étages de traitement est limité par le nombre d'étapes élémentaires à réaliser mais il est influencé par les accès à la mémoire et le nombre d'unités de traitement (cf. sections suivantes). Plus le pipeline est long, plus il est couteux de le vider et le réalimenter, c'est ce qui arrive lors de l'exécution des instructions conditionnelles ou lors du traitement des boucles. Il se limite à une quinzaine d'étages sur la plupart des microprocesseurs actuels. Voyons par la suite comment les différentes étapes de traitement des instructions s'enchainent.

➢ **Etape 1 ; Frontal : chargement et décodage**

Sur le schéma de la Figure 4.9 on a fait apparaitre les différentes étapes liées au frontal. A partir de **cs:eip** on obtient l'adresse de la prochaine instruction à exécuter.
Cependant comme nous allons le voir et comme cela a déjà été évoqué, certaines instructions assembleur modifient **eip** et il est nécessaire d'utiliser un mécanisme de prédiction de branchement, représente sur la figure par **BPU** pour ***Branch Prediction Unit***, afin de savoir si l'on devra lire l'instruction suivante ou si on devra se déplacer à une autre adresse du code.
Une fois que l'on dispose de la bonne adresse, on récupère l'instruction à exécuter dans le cache L1 d'instructions (L1i). Il se peut que l'instruction ne soit pas présente dans le cache L1i, il faudra alors chercher si elle est dans le cache L2, puis dans le cache L3 et finalement, si elle n'est présente dans aucun cache, il faudra lancer une requête d'accès en mémoire pour récupérer les octets situés à l'adresse à lire et les charger dans les différents caches ou dans le cache L1i uniquement.

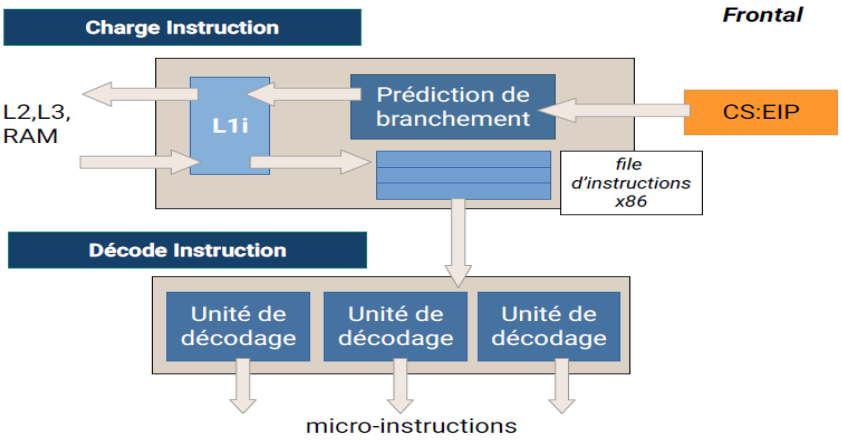



**Figure 4.9. Chargement et décodage [17].**

➢ **Etape 2 ; Chargement et prédiction de branchement**

Comme nous venons de le dire, le chargement d'une instruction fait appel à plusieurs mécanismes dits de *prediction de branchement* qui permettent de prédire à quelle adresse le pointeur d'instruction (**eip**) doit se placer. Généralement il s'agit de l'instruction suivante. Mais dans le cas de branchements, d'une boucle **for** par exemple, il faut revenir au début de la boucle après avoir exécuté son corps ou sortir de la boucle lorsque la condition d'arrêt est atteinte. On dit alors qu'il existe plusieurs chemins d'exécution.

Considérons le code C de la **Figure 4.10** pour lequel on calcule la somme des entiers de 1 à 10. On voit sur l'organigramme de gauche qu'il existe deux chemins :

- Le premier est pris lorsque i ≤ 10 et le second lorsque i > 10.
- Le registre **eax** contient la somme des valeurs et le registre **ecx** représente la variable de boucle (**i**).
- Après l'utilisation de l'instruction cmp ecx, 10 qui compare le registre **ecx** à la constante 10, on place une instruction de branchement conditionnel jg .end_for, qui signifie *jump on greater*.
- Ces instructions de branchement conditionnel sont source de ralentissement au sein du pipeline puisqu'il est nécessaire de vider le pipeline si le chemin d'exécution suivi n'est pas le bon. Si **ecx** est supérieur à 10 il faut sortir de la boucle et modifier **eip** pour qu'il pointe sur l'instruction après le label .end_for, c'est à dire l'instruction i8. Cependant les instructions suivant la comparaison (*i5, i6, i7*) ont déjà été chargées dans le pipeline pendant le traitement de *i3* et *i4*.

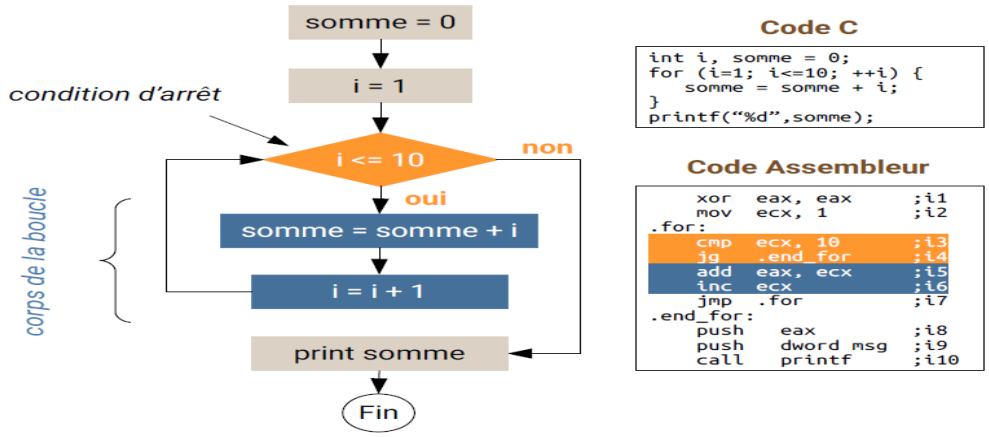



**Figure 4.10. Exemple de boucle *for* [17].**

On doit donc invalider leur traitement en vidant le pipeline et recommencer à partir de l'instruction i8.
Afin d'éviter le plus possible de vider le pipeline, la prédiction de branchement, comme son nom l'indique, permet de prédire des lors qu'une instruction de type branchement est présente, si le branchement sera emprunté ou non. De son efficacité découle une vitesse de traitement accrue.
Notons également que les conditionnelles de type **if then** ou **if then else** a l'intérieur d'une boucle (**for** ou **while**) sont les plus pénalisantes et le sont d'autant plus qu'on ne peut prédire la condition du **if.**

➢ **Etape 3 ; Décodage d'instructions**

Les instructions assembleur peuvent être qualifiées de macro-instructions car elles définissent des traitements parfois très complexes. Au sein du microprocesseur, ces macro-instructions sont décomposées en une série d'instructions plus simples appelées micro-opérations et notées **µ-ops**.
L'instruction **add** [**ebx** + **ecx** * **4** + **8**], **eax**. Cette instruction sera décomposée en plusieurs micro-opérations beaucoup plus simples afin d'être exécutée :

1. µ-op1 : calcul de l'adresse **A = ebx + ecx * 4 + 8**

2. µ-op2 : chargement de la donnée à l'adresse mémoire A dans le registre R

3. µ-op3 : exécution de l'addition R + **eax** et stockage dans R

4. µ-op4 : stockage de R à l'adresse mémoire A

De retour à la **Figure 4.9**, nous voyons qu'une fois chargée dans une file d'instructions x86, la prochaine instruction à exécuter doit être décodée en micro-instructions. Généralement, il existe un mécanisme de cache de traduction représente sur la figure par le µ-Ops Cache. Ce cache a pour objectif de stocker la série de micro-instructions générées par le décodage d'une instruction x86 précédemment décodée. Si l'instruction x86 est présente dans ce cache, on approvisionnera la file de µ-ops avec les données du cache, sinon on utilisera le décodeur qui est le circuit dédié à la traduction d'une instruction x86 en µ-ops.
De nos jours la partie décodage est capable de décoder plusieurs instructions à la fois, généralement de l'ordre de 3 à 5 sur les microprocesseurs récents.

➢ **Etape 4 ; Exécution des instructions**

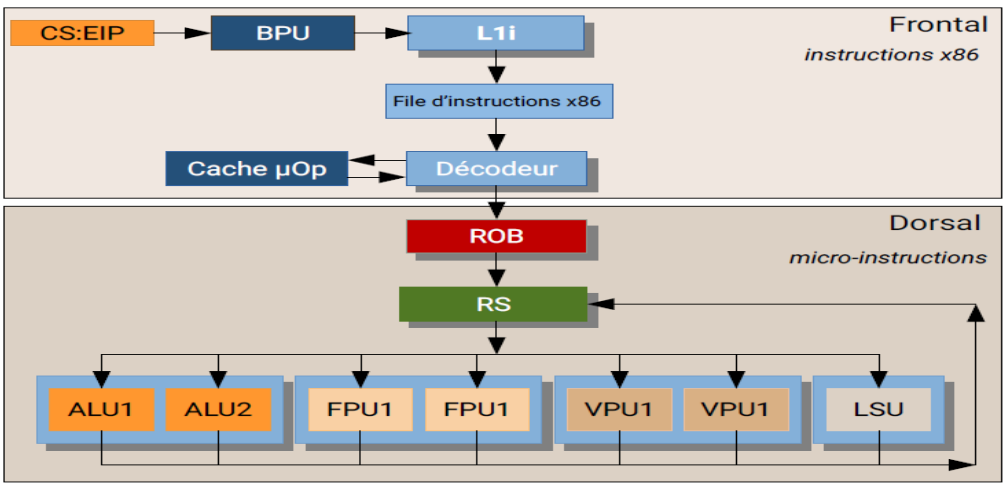



**Figure 4.11. Traitement instruction [17].**

Au niveau du dorsal (Figure 4.11), c'est un ensemble de µ-ops associées à des instructions x86 que l'on doit traiter. Afin de diminuer les temps d'attentes et ne pas ralentir l'exécution du traitement on utilise un mécanisme d'exécution dans le désordre (*Out Of Order*) qui consiste à traiter les µ-ops des lors qu'elles disposent de toutes les ressources nécessaires pour être traitées. Cependant, cette exécution dans le désordre pose un problème crucial à résoudre : faire en sorte qu'au final les instructions x86 soient traitées dans l'ordre dans lequel elles sont entrées dans le pipeline de traitement.
Pour ce faire, on utilise deux tampons (*buffers*) appelés *Reservation Station* et *ReOrder Buffer* notes respectivement **RS** et **ROB**. Nous ne détaillerons pas leur fonctionnement afin de rester le plus concis possible et ne pas désorienter le lecteur, mais ces deux tampons assurent les fonctionnalités suivantes :
• **ROB**, comme son nom l'indique est chargé de garder la cohérence et maintenir l'ordre d'exécution, il est également chargé de l'*allocation de registres*
• **RS** est charge de stocker les instructions et de les garder jusqu'a ce qu'elles soient exécutées
L'*allocation avec renommage de registres* est une technique essentielle pour traiter les instructions dans le désordre. En interne le microprocesseur dispose de plusieurs registres et lorsqu'il traite une instruction x86 il établit une correspondance entre les registres visibles par le programmeur (**eax**, **ebx**, etc...) et ses registres internes de manière à pouvoir traiter chaque instruction de manière indépendante.

### *5.2. Exemple d'instructions en 32 bits et 64 bits*

La grande majorité des instructions x86 sont de la forme :

**Opération** *destination*, *source*

• **opération** est un mnémonique, c'est à dire un symbole court de quelques lettres facilement compréhensible et mémorisable qui représente l'opération à exécuter, par exemple **add** pour l'addition
• *source* est une donnée en lecture qui ne sera donc pas modifiée, ce peut être une constante, un registre ou une adresse mémoire
• *destination* est une donnée en écriture qui peut être un registre ou une adresse mémoire
• les deux opérandes *source* et *destination* sont séparées par une virgule
En fait la syntaxe de l'assembleur permet d'écrire plus simplement le traitement qui est effectivement réalise, à savoir :

**Destination = destination opération source**

**Remarque:** Même principe en utilisant l'architecture **64 bits avec les registres de 64 bits rax, rbx, …, puisque l'utilisation des registres de 32 bits avec l'architecture 64 bits signifie le reste des bits sera remplis est complété par des zéros**

Ainsi, **add eax**, **ebx** signifie que l'on doit réaliser le calcul **eax** = **eax** + **ebx** : opération d'addition en 32 bits, et **add rax**, **rbx** opération d'addition en 64 bits
Nous verrons qu'il existe d'autres variantes de format d'instruction comme pour les instructions **neg**, **not**, **cmp**, **test**, **div**, **mul**, etc.
Il existe toutefois une contrainte imposée par le format de codage des instructions x86 qui nous force à n'avoir qu'une seule référence mémoire cela implique que l'on ne peut pas écrire :

```
1    operation    [adresse1], [adresse2]    ; !!! non autorisé !!!
```

Il faudra alors passer par un registre et écrire :

```
1    mov          registre, [adresse2]
2    operation    [adresse1], registre
```

➢ **L'instruction de chargement et stockage des données : Mov**

L'instruction **mov** dont nous allons reparler ci-après déplace la donnée située à l'adresse dans un registre.

On remarque également que lorsque l'on fait référence à une donnée en mémoire identifiée par son adresse, on place l'adresse entre crochets [ ]. Ainsi :

• **mov eax**, [**addr**] signifie placer la valeur 32 bits située a l'adresse **addr** dans le registre **eax**

• **mov eax**, **val** signifie placer la valeur constante codée sur 32 bits **val** dans le registre **eax**

**Exemples :**

- **mov eax**, **0** : affecter la valeur 0 au registre **eax**
- **mov eax**, **ebx** : affecter le contenu de **ebx** au registre **eax**
- **mov al**, **bh** : affecter le contenu de **bh** au registre **al**
- **mov eax**, [**ebx** + **ecx** * **4**] : affecter au registre **eax** la valeur située à l'adresse mémoire indiquée, il s'agit d'une référence mémoire comme **eax** est un registre 32 bits on lit le double mot situe a l'adresse indiquée
- **mov** [**edi** + **esi**], **edx** : stocker a l'adresse **edi** + **esi** la valeur contenue dans le registre **edx**

On trouve deux variantes de l'instruction **mov** :

✓ **movsx** (*Mov with Sign eXtension*) qui transforme une valeur sur **8 (respectivement 16 bits)** en une valeur **16 (respectivement 32 bits)** en préservant le fait que la valeur soit négative ou positive

✓ **movzx** (*Mov with Zero eXtend*) qui transforme une valeur sur **8 ou 16 bits** en une valeur **16, 32 ou 64 bits** en la complétant avec des 0. L'instruction **movzx** est parfois plus rapide que mov 1. Il est donc préférable d'écrire, afin de lire l'octet à l'adresse **edi** en mémoire et le stocker dans le registre **al** :

```
1       movzx eax, byte [edi]
```

Plutôt que :

```
1       mov   al, [edi]
```

La différence est que **movzx** va modifier **eax** en mettant dans **al** l'octet pointe par **edi** et en mettant à 0 les 24 autres bits. On remarque que dans ce cas il faut préciser la quantité chargée : **byte** pour un octet, **word** pour un mot et dans d'autres instructions **dword** pour un double mot.

➢ **Instructions arithmétiques**

✓ **Instructions add, sub, inc et dec**

Les instructions **add** et **sub** réalisent respectivement l'addition et la soustraction de deux valeurs entières signées ou non signées et prennent deux opérandes.

Les instructions **inc** et **dec** réalisent respectivement l'incrémentation et la décrémentation de leur unique opérande. On peut les utiliser pour réaliser les opérations du C comme ++i qui correspond a l'incrémentation d'une variable de boucle **for**.

Les deux instructions suivantes sont donc équivalentes :

```
1       add   eax, 1
2       inc   eax
```

On trouve également l'instruction **sbb** (*SuBstract with Borrow*) pour faire des soustractions si on utilise deux registres **16** ou **32 bits**.

Concernant les instructions **inc** et **dec**, elles modifient les flags OF, SF, ZF, AF et PF, mais pas le Carry Flag. Il est de plus conseillé de ne pas les utiliser car elle peuvent produire dans certaines situations des *false dependencies*.

### ✓ L'instruction mul

L'instruction **mul** n'accepte qu'un seul opérande source et réalise la multiplication non signée entre un registre 8, 16 ou 32 bits et respectivement **al**, **ax**, **eax** comme indique Figure 4.12. Les notations **reg8, reg16, reg32** signifient respectivement un registre général **8, 16 ou 32 bits.**

| Opération | Source | Résultat |
|---|---|---|
| mul reg8 | al | ax |
| mul reg16 | ax | dx:ax |
| mul reg32 | eax | edx:eax |

**Figure 4.12. Modes d'utilisation de mul [17].**

Par exemple, si on écrit **mul bh**, c'est le registre **al** qui est multiplie par **bh** et le résultat est place dans **ax**.
En architecture 32 bits, on notera qu'avec un opérande source de 16 (resp. 32 bits), le registre **dx** (resp. **edx**) est modifie. Il ne faudra donc pas stocker de donnée dans **edx**, ou alors, sauvegarder cette donnée avant la multiplication en la plaçant dans la pile, puis après la multiplication, la récupérer depuis la pile. Pour calculer 7 × 5 ou 5 × 7, on écrira donc :

```
1    push   edx        ; on sauvegarde edx
2    mov    eax, 5
3    mov    ebx, 7
4    mul    ebx       ; edx:eax= 0:35
5    pop    edx       ; on restaure edx
```

### ✓ L'instruction div et le modulo

L'instruction **div** réalise la division entière **non signée** entre une valeur 64, 32 ou 16 bits par un diviseur sur 32, 16 ou 8 bits respectivement, le reste de la division est également calculé. L'instruction **div** permet donc également de réaliser le modulo (voir Figure 4.13).
Par exemple en architecture 32 bits, c'est en fait **une valeur sur 64 bits contenue dans deux registres 32 bits edx:eax** que l'on divise par un opérande 32 bits contenu dans un autre registre. Si on désire travailler avec des valeurs 32 bits, il faut mettre **edx** à 0 avant de faire la division. Pour diviser 1024 par 3, on écrira donc :

```
1    mov    eax, 1024
2    xor    edx, edx    ; mise à zéro de edx pour rester
3                       ; en 32 bits
4    mov    ebx, 3
5    div    ebx
```

Le registre **eax** contiendra alors la valeur 341 et **edx** sera egal a 1 car 1024 = 3 × 341 + 1

| Dividende | Diviseur | Quotient | Reste |
|---|---|---|---|
| edx:eax | div reg32 | eax | edx |
| eax | div reg16 | ax | dx |
| ax | div reg8 | al | ah |

**Figure 4.13. Comportement de l'instruction div [17].**

✓ **Comparaison : L'instruction cmp**

L'instruction **cmp** (*CoMPare*) réalise la comparaison non signée de deux opérandes en calculant leur différence. Les opérandes ne seront pas modifiés seule la différence sera utilisée pour mettre à jour le registre des **flags**. Voici quelques utilisations de cette instruction :

• **cmp eax**, **10** : compare **eax** à la constante 10
• **cmp eax**, **edx** : compare **eax** à **edx**
• **cmp ecx**, [**ebp**-**12**] : compare **ecx** a l'entier 32 bits situe en mémoire à l'adresse [ss:ebp-12]
La Figure 14 montre comment sont modifiés les principaux bits du registre **flags** lors de la comparaison en fonction des opérandes de la commande **cmp**.

| cmp eax, ebx | CF | SF | ZF |
|---|---|---|---|
| eax == ebx | 0 | 0 | 1 |
| eax < ebx | 1 | 1 | 0 |
| eax > ebx | 0 | 0 | 0 |

**Figure 4.14. Influence sur les bits du registre flags de la comparaison de valeurs [17].**

✓ **Instructions de branchement conditionnel et inconditionnel**

- **Loop (conditionnel)**

Il existe également une instruction spécifique **loop** ***address*** qui décrémente le registre **ecx** et, si celui-ci n'est pas égal à 0, se branche à l'adresse indiquée. Elle est donc équivalente aux deux instructions suivantes :

```
.begin:
    ...
    dec   ecx     ; à remplacer par
    jnz   .begin  ; loop .begin
```

- **Autres instructions de branchement (inconditionnel)**

Il s'agit des instructions de branchement comme :
• **jmp address** : modifie (*JuMP*) le pointer d'instruction pour qu'il soit égal à l'opérande ***address***
• **call address** : réalise un appel de sous-programme
• **ret** : réalise le retour de sous-programme

L'instruction **call** empile l'adresse de l'instruction qui lui succède puis modifie le registre **eip** pour qu'il soit égal a **address**. L'instruction **ret**, utilisée lors du retour de sous-programme, dépile l'adresse en sommet de pile (placée par **call**) et l'affecte à **eip**.

## Résumé

En architecture **32 bits**, on dispose de huit registres généraux, cependant seuls 6 sont utilisés pour faire des calculs ou stocker des données (**eax**, **ebx**, **ecx**, **edx**, **edi**, **esi**) ; le registre **esp** contient le sommet de pile et ne doit pas être modifie directement alors que **ebp** est utilisé afin de récupérer les arguments d'un sous-programme :

- En architecture **32 bits**, si l'on doit réaliser des multiplications ou des divisions les registres **eax** et **edx** seront impactes, ce qui ne laisse plus que 4 registres pour faire les calculs
- En architecture **64 bits**, on dispose de 8 registres généraux supplémentaires (**r8** a **15**), ce qui permet de lever le verrou des limitations du **32 bits**
- L'assembleur ne dispose pas de structures de contrôle comme la conditionnelle **if**, les boucles **for**, **while**. Ecrire en assembleur est donc une tâche difficile.
- Les techniques de dépliage de boucle ou de tuilage permettent d'améliorer l'efficacité du traitement des boucles
- Positionner un **if** a l'intérieur d'une boucle (**for** ou **while**) conduit à ralentir l'exécution du traitement, il faut alors être en mesure de pouvoir éliminer le **if** soit en le remplaçant par des instructions spécifiques, soit en le vectorisant
-

### ✓ Compétences à acquérir

On doit être capable de traduire :

- Une instruction arithmétique : addition soustraction, multiplication, une division
- Une instruction de branchement conditionnelle ou inconditionnelle

# Conclusion

Si vous utilisez des applications très lourdes dans votre travail, il est peu probable que vous constatiez une amélioration des performances. En conclusion, nous devrions dire que la différence entre les systèmes 32 et 64 bits est significative. Cet article peut être poursuivi, mais il est déjà suffisant en tant qu'introduction.

Nous pouvons donc constater que les architectures sont très différentes les unes des autres, et tout d'abord par le fait que les systèmes 64 bits sont plus optimisés, conçus pour le matériel le plus récent, le multitâche, le travail productif et productif. Aujourd'hui, toutes les puces fonctionnent en mode 64 bits mais supportent le 32 bits pour la compatibilité en mode émulation. Vectoriser son code est une opération peu couteuse et qui permet d'obtenir une amélioration importante en termes de diminution du temps de calcul. Les unités vectorielles sont qualifiées de SIMD (*Single Instruction Multiple Data*) et permettent de paralléliser les calculs en effectuant la même opération sur des données différentes :

- Les unités SSE permettent de traiter 128 bits alors que les unités AVX sont capables de traiter 256 bits de différents formats allant de l'octet au nombre à virgule flottante double précision
- La technologie AVX apporte généralement, dans la plupart des traitements, un gain négligeable mais peut, dans certains cas, se révéler deux fois plus efficace que le SSE

A la fin, on a abordé les notions liées à la mémoire (comme l'alignement, l'adressage mémoire et le *dual channel*) et les notions relatives au fonctionnement du microprocesseur (chargement des instructions, décodage et exécution, pipeline, etc), et on a introduire les instructions assembleur de base.

# Chapitre 5 : Architectures des processeurs récents

## Introduction

Le domaine du traitement parallèle concerne les méthodes architecturales et algorithmiques permettant d'améliorer les performances des ordinateurs numériques ou d'autres critères tels que la rentabilité et la fiabilité grâce à diverses formes de concurrence. Même si le calcul concurrent existe depuis les premiers jours des ordinateurs numériques, ce n'est que récemment qu'il a été appliqué d'une manière permettant d'obtenir de meilleures performances, ou un meilleur rapport coût-efficacité, par rapport aux superordinateurs habituels. Comme tout autre domaine scientifique ou technologique, l'étude des architectures et algorithmes parallèles nécessite une motivation, une vue d'ensemble montrant les relations entre les problèmes et les différentes approches pour les résoudre, ainsi que des modèles pour comparer, connecter et évaluer les nouvelles idées, dont l'objectif de ce chapitre.

Depuis le début de l'informatique s'est posée la question de résoudre rapidement des problèmes (le plus souvent numérique) coûteux en matière de temps de calcul : Simulation numérique, cryptographie, imagerie, SGBD,…etc.
Pour résoudre plus rapidement un problème donné, une idée naturelle consiste à faire coopérer simultanément Multi-cœurs au lieu plusieurs agents à sa résolution, qui travailleront donc en parallèle. Au fil des années les mémoires caches se sont développées et sont devenues de plus en plus volumineuses en raison notamment de l'apparition des microprocesseurs multi-cœurs. Initialement absentes, elles ont commencé à être disponibles sur la carte mère, puis ont été progressivement intégrées au microprocesseur.
Ce chapitre est organisé comme suit : tout d'abord la définition des architectures parallèles (Multi-cœurs et parallélisme) leurs principales caractéristiques et leurs classifications, dans section 1, Mémoire cache et hiérarchie mémoire dans section 2

## 1. Introduction aux processeurs modernes : tendances et évolutions technologiques

Dans les architectures des ordinateurs, on peut définir une **machine parallèle** comme étant, essentiellement, une machine qui possède **un ensemble de processeurs** qui coopèrent et communiquent ensemble et qui concourent dans **le but d'exécuter un programme parallèle**. Par conséquent, on peut définir les ordinateurs parallèles comme étant des machines qui comportent une architecture parallèle.
Un **programme parallèle** est un ensemble fini de processus séquentiels communiquant entre eux par échange de messages.
**La performance** d'une architecture parallèle est la **combinaison des performances** de ses ressources tels que la latence, le débit, leur coût de mise en œuvre, etc.

### *1.1.Multi-cœurs et parallélisme*

#### 1.1.1. Concept de parallélisme et multi-cœur

Beaucoup d'applications présentent des propriétés de parallélismes, autrement dit elles peuvent être l'objet d'une éventuelle exécution parallèle en cas de présences de ressources matérielles suffisantes. Le parallélisme d'une application consiste à faire exécuter simultanément plusieurs parties indépendantes de cette application sur plusieurs ressources matérielles. Selon les structures des applications, il existe deux formes de parallélismes :

- ✓ Parallélisme de contrôle
- ✓ Parallélisme de données

➢ **Parallélisme de données**

C'est un type de parallélisme pour lequel la même opération est réalisée simultanément par de nombreux processeurs sur des données différentes.
Exemple : l'addition de deux matrices A et B. elle consiste, pour tous les éléments de mêmes indices des deux matrices opérandes à les additionner et à stocker le résultat dans l'élément de même indice d'une troisième matrice résultat C.

```
Pour i = 1 à N faire
  Pour J = 1 à M faire
    C[i,j] ← A[i,j] + B[i,j]
  Fin_pour
Fin_pour
```

Les N*M itérations de cette boucle sont indépendantes, on peut donc faire N*M opérations simultanées. L'ampleur du potentiel de parallélisme exploitable dépend directement de la taille des structures de données manipulées.
***Exemple d'application de ce parallélisme*** :

- ✓ Applications numériques
- ✓ Les traitements de base de données (avec des milliers d'entées)
- ✓ Le traitement du signal et de l'image (matrices de plusieurs milliers d'éléments : pixels).

Il s'agit dans ce type de parallélisme, d'utiliser de nombreux processeurs simultanément et de leur faire exécuter la même opération. Généralement le nombre de processeurs est beaucoup plus petit que le parallélisme potentiel et chaque processeur devra donc traiter plusieurs données.

➢ **Parallélisme de contrôle**

C'est un type de parallélisme pour lequel des opérations différentes sont réalisées simultanément. Ce parallélisme peut provenir de l'existence dans le programme de fonctions indépendantes (parties indépendantes) ou d'opérations indépendantes dans une suite d'opérations. C'est l'absence de dépendance entre différentes parties de programme qui est la source de parallélisme.
Ce parallélisme ne dépend donc pas des données mais de la structure du programme à exécuter.
***Les systèmes d'exploitation des ordinateurs modernes sont des exemples de ce type de parallélisme.***
Un exemple d'exploitation du parallélisme de contrôle est ***le pipeline***. Il repose sur l'organisation des données à traiter sous la forme d'un flux d'informations sur l'application de plusieurs traitements consécutifs sur chaque élément de flux.

### 1.1.2. Les plateformes de calcul Parallèle

Le calcul parallèle ne peut avoir lieu que sur des **plateformes de calcul Parallèle.**
Parmi les plateformes de calcul parallèle actuelles, on distingue :

- ✓ **Les machines multi-cœurs** : Processeurs possédant plusieurs cœurs physiques (unités de calcul) gravés au sein de la même puce.

Ces plateformes sont largement disponibles à des coûts de plus en plus faibles.
De nos jours, le nombre de cœurs d'un processeur type pourrait atteindre des centaines
**Exemple :** (**Intel® Xeon Phi™7290** 16GB, 1.50 GHz, 72 cœurs)

- ✓ **Les supercalculateurs** : Ordinateurs conçus pour atteindre les plus hautes performances possibles avec les technologies disponibles au moment de leur conception. Ces machines ne sont généralement pas à la portée de tout le monde.
- ✓ **Autres plateformes de calcul parallèle :** peuvent être élaborées en faisant collaborer des ordinateurs ordinaires interconnectés, telles que :

  - **Les clusters** (ensemble d'ordinateurs reliées au sein d'un réseau local).
  - **Les grilles de calcul** qui sont des agrégations de ressources de calcul hétérogènes et distribuées sur des sites distants.
  - **Cloud Computing**
  - **Les Processeurs graphiques (GPGPU) :** les processeurs graphiques modernes contiennent des milliers de cœurs (exp : 5120 cœurs dans la Tesla V100)

### 1.1.3. Pourquoi le parallélisme ? (Avantages et défis)

La réponse à cette question se résume dans les trois points suivants :

- ✓ Les besoins croissants en hautes performances,
- ✓ L'évolution de la technologie, et
- ✓ L'évolution des applications.

Deux facteurs majeurs contribuent à l'amélioration des performances des processeurs modernes :

- ➢ La technologie rapide des circuits et les caractéristiques architecturales, à savoir des caches de plus en plus grands, des bus multiples de plus en plus rapides, le pipelining, les architectures superscalaires, etc.

- ➢ Le deuxième facteur concerne le fait que les calculateurs avec une seule unité centrale de traitement (*central processing unit* CPU) sont souvent incapables de répondre à certains besoins comme : l'analyse des flots de liquides et aérodynamique, la simulation de systèmes complexes (physique, économie, biologie, météo, technique, etc.), l'analyse de données massives pour la prise de décisions stratégiques, la conception assistée par ordinateurs (CAO) et le Multimédia, etc. Du fait que ces applications sont caractérisées par une très grande quantité de calculs numériques et/ou une grande quantité de données en entrée (Figure 5.1).

L'une des solutions à ce besoin en performances est la création des architectures dans lesquelles plusieurs CPUs fonctionnent dans le but de résoudre une application donnée. De tels calculateurs ont été organisés de différentes manières selon certaines caractéristiques clés, notamment le nombre et complexité des CPU individuels, la disponibilité de mémoires partagées ou communes, la topologie d'interconnexion, les performances des réseaux d'interconnexion, les dispositifs d'Entrées/Sorties, etc.

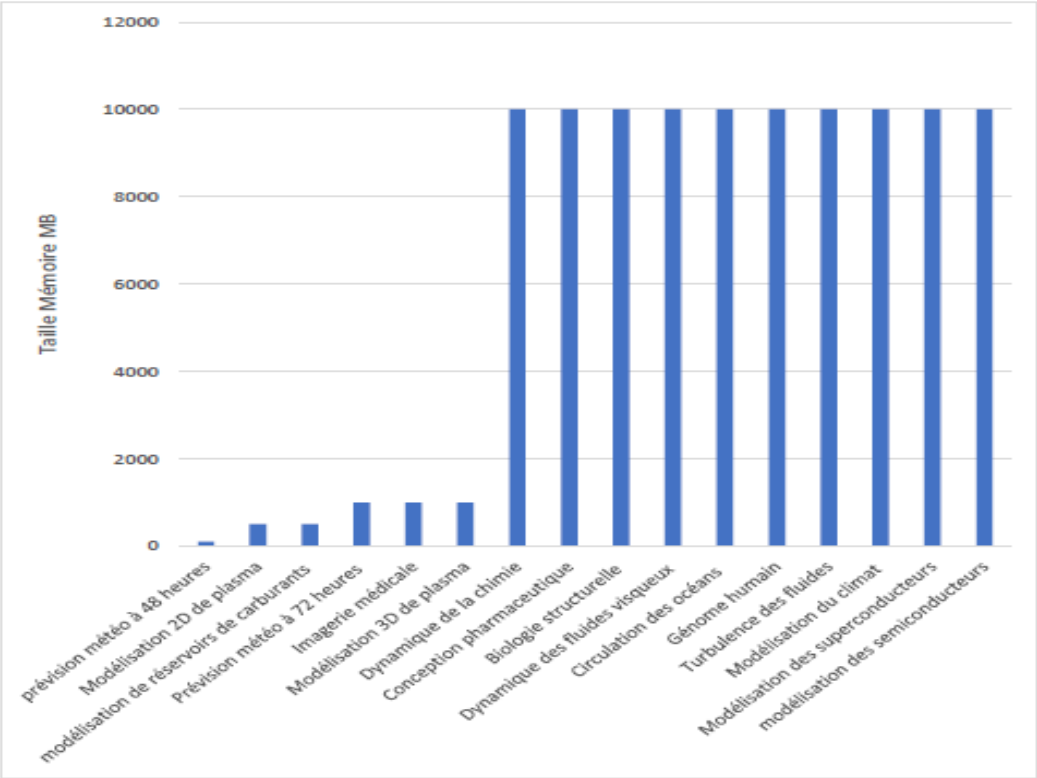



**Figure 5.1. Les besoins en performances [8].**

### 1.1.4. Les caractéristiques des architectures parallèles

Les architectures parallèles sont caractérisées par une ou plusieurs caractéristiques parmi les suivantes :

- ✓ Espace mémoire étendu.
- ✓ Plusieurs processeurs.
- ✓ Traitement partiel indépendant.
- ✓ Accélération des calculs complexes ou coûteux en temps d'occupation CPU.
- ✓ Calcul parallèle répétitif sur un large ensemble de données structurées.

Alors que les performances architectures parallèles sont limités lorsqu'il existe dans le programme parallèle :

- ➢ **Des dépendances fonctionnelles :** que ce soit des dépendances de données, ou des dépendances de contrôle.
- ➢ **Des dépendances de ressources :** si le nombre de processeurs est insuffisant par rapports le nombre des taches parallèles indépendantes.
- ➢ **Des délais de communication élevés :** les communications peuvent dans ce cas augmenter le temps d'exécution et donc il ne sera pas avantageux de paralléliser une application dans ce cas.

### 1.1.5. Paradigme de la programmation parallèle

La programmation parallèle consiste, en général, à ***décomposer un seul problème*** de taille N en deux ou plusieurs sous-problèmes de petites tailles qui ne présentent ***pas le même degré de***

***parallélisme***. Aussi, elle consiste à utiliser la concurrence, la coordination et la communication dans un programme, afin d'augmenter la taille des problèmes qui peuvent être résolus ou de réduire le temps de réponse pour résoudre un problème donné.
***Le modèle de programmation parallèle dépend fortement de l'architecture parallèle utilisée*** car nous ne pouvons pas programmer, par exemple, une machine avec une mémoire distribuée de la même manière qu'une machine avec une mémoire partagée.

Aussi, nous ne pouvons pas appliquer les principes de la programmation séquentielle en programmation parallèle. Parce qu'en programmation parallèle, le programmeur doit décider s'il applique le parallélisme de données (dans le cas d'une architecture SIMD par exemple) : L'architecture SIMD est une architecture dont toutes les unités de traitement exécutent le même flux d'instructions qui opèrent sur des flux de données différents. Cependant, ce type de programmation pose quelques ***problèmes liés à la communication et la synchronisation de tâches parallèles*** suite à la charge d'un réseau par exemple ou encore à la différence des distances entre les lieux d'exécution des tâches parallèles.

Les deux modèles de la programmation parallèles les plus connus sont :
- **Paradigme de programmation Multithreading** : La programmation parallèle la plus simple consiste à avoir plusieurs threads ou flots d'instruction qui s'exécutent en manipulant des données stockées dans une mémoire commune, ***Un thread*** est défini comme étant une courte séquence d'instructions pouvant être exécutée en tant qu'une seule unité par un processeur.
- **Paradigme message-passing** : Ce modèle de programmation est naturellement adéquat aux architectures parallèles dans lesquels chaque processeur possède son propre espace mémoire.

#### 1.1.6. Organisation mémoire

Les méthodes de synchronisation et de coordination en calcul parallèle sont fortement liées à la façon dont les informations sont échangées entre les **processus** ou **threads**.
- ✓ La façon dont les informations sont échangées dépend de l'organisation de la mémoire du hardware.
- ✓ Une classification grossière de l'organisation mémoire distingue entre les machines à **mémoire partagée** et celles à **mémoire distribuée.**

Souvent, le terme ***thread*** est lié à la mémoire **partagée** et le terme ***processus*** est lié à la mémoire **distribuée**.

### *1.2. Mémoire cache et hiérarchie mémoire*

Le progrès technologique des ordinateurs a fait en sorte que la mémoire principale n'arrive plus à suivre le rythme croissant des fréquences d'horloge des processeurs. Donc, pour faire face à cette lenteur, il a été possible d'intercaler entre la mémoire principale et le processeur une mémoire cache plus rapide (jusqu'à 10 fois plus rapide que la RAM) et de faible taille (jusqu'à 100 fois plus petite que la RAM). Cela constitue un premier niveau de hiérarchisation de la mémoire. L'objectif des mémoires caches est de permettre au processeur de mémoriser les informations les plus utilisées afin d'être en mesure de les récupérer le plus rapidement possible en cas de besoin. De cette façon, le processeur n'a pas besoin d'aller chercher les informations auprès de la mémoire principale, ce qui constitue un gain de temps très

important. Bien évidemment, l'efficacité d'un tel système dépend des caractéristiques de la mémoire cache, à savoir son efficacité et sa taille.

### 1.2.1. Principe de la mémoire cache et hiérarchie mémoire

Il est possible de hiérarchiser la mémoire en plaçant plusieurs autres niveaux de cache entre la mémoire principale et le processeur. Dans les systèmes actuels, nous n'avons pas une seule mémoire, mais plutôt plusieurs mémoires organisées en hiérarchie l'objectif des mémoires caches, elles doivent être rapides pour alimenter le processeur en instructions et en données, et de grande capacité pour stocker l'intégralité des informations disponibles. Or, plus la mémoire et de grande capacité, plus son accès devient lent. Il faut donc utiliser tous les types de mémoires en hiérarchisant et en répartissant les informations en fonction de leur fréquence d'utilisation.
La figure 5.2 montre comment les mémoires sont organisées selon leur capacité et temps d'accès.

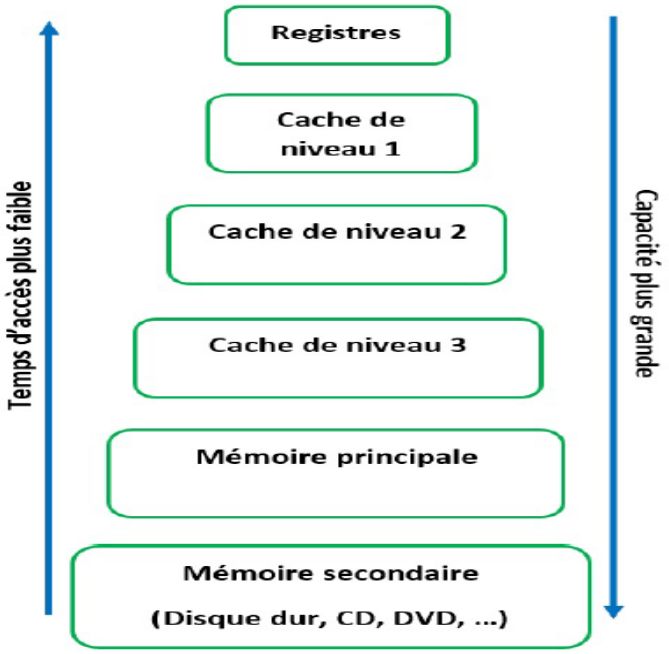



**Figure 5.2. Hiérarchie des mémoires [19].**

Toutes les opérations mémoires de lecture et d'écriture sont dirigées d'abord vers la mémoire cache afin de vérifier la présence ou absence de la donnée. Deux scénarios possibles :

- ✓ **La donnée est présente :** ce cas s'appelle succès ou (hit, en anglais). Dans ce cas, la donnée est transmise au processeur.
- ✓ **La donnée est absente :** ce cas s'appelle défaut ou (miss, en anglais). Dans ce cas, la donnée est transmise d'abord de la mémoire principale vers la mémoire cache pour être délivrée ensuite au processeur.

### 1.2.2. Caractéristiques de la mémoire cache

Nous avons vu dans la section précédente le principe général de la mémoire cache et comment elle est utilisée par le processeur. Dans cette section, nous allons découvrir les caractéristiques de la mémoire cache en ce qui a traits à la taille, à l'adressage, et aux algorithmes de remplacement.

- **Taille d'une mémoire cache**

La technologie utilisée dans la mémoire cache met en évidence que plus la mémoire cache est de grande capacité, plus le processeur a de chance d'y trouver de l'information dont il cherche. Par conséquent, pour arriver à des meilleures performances, on cherche toujours à avoir la mémoire cache avec la plus grande capacité possible.
Cependant, la technologie utilisée dans la mémoire cache fait en sorte que son prix est beaucoup plus élevé que celui de la mémoire principale. Donc, plusieurs considérations doivent être prises en compte pour arriver à un compromis entre la taille du cache et les performances du processeur :

- ✓ La mémoire cache est très rapide mais très chère aussi comparativement à la mémoire principale.
- ✓ La mémoire cache occupe plus d'espace si sa capacité est plus grande. La taille d'une mémoire cache varie généralement entre quelques centaines de Ko à quelques Mo. La mémoire cache ne peut donc pas y stocker tout un programme et ses données.
- ✓ Le temps d'accès à la mémoire cache est plus rapide si la capacité et faible et vice versa. Donc, cette proportionnalité affecte les performances si la mémoire cache est de grande capacité.

Le constructeur de la mémoire cache doit prendre en compte toutes ces considérations et même d'autres pour arriver à une bonne conception des systèmes

- **Adressage**

Comme nous l'avons vu dans le cadre de la mémoire principale, la mémoire cache possède également différents types d'adressage. C'est-à-dire, lorsqu'une information est transférée de la mémoire principale à la mémoire cache, il faut décider où placer cette information.

- **Algorithmes de remplacement**

Comme nous l'avons vu dans les sections précédentes, le cache contient une partie des lignes de la mémoire centrale. Si le processeur effectue une opération de lecture ou d'écriture dans une ligne qui n'existe pas dans le cache, à ce moment là, on doit charger en cache cette ligne manquante. Vu que le cache est plein, le processus de chargement des nouvelles lignes dans le cache nécessite d'enlever des lignes pour y mettre d'autres lignes. Il existe principalement 5 méthodes pour choisir la ligne à remplacer :

- ✓ Aléatoire
- ✓ FIFO (First In First Out)
- ✓ LRU (Least Recently Used)
- ✓ LFU (Least Frequently Used)
- ✓ NMRU (Non-Most-Recently-Used)

- **Politique de réécriture**

Outre les opérations de lecture en mémoire principale, le processeur effectue également des écritures, par exemple lorsqu'il modifie des données. Il existe deux méthodes pour la réécriture des données présentées ci-dessous :

- ✓ **Cache write-through**

Le principe de cette méthode est simple : Écriture simultanée (write-through). C'est-à-dire, lorsqu'on écrit un mot d'une ligne du cache, on écrit simultanément le mot de la même ligne en mémoire centrale. De cette façon l'écriture est propagée du cache vers la mémoire centrale.

- ✓ **Cache write-back**

Contrairement à la méthode write-through, la méthode write-back (à écriture différée) ne modifie l'information en mémoire principale que lorsque la ligne correspondante disparaît du cache. Cela permet de ne pas perdre trop de temps à accéder à la mémoire principale.

### *1.3. Gestion de la mémoire cache dans les architectures récentes*

La gestion de la mémoire cache est l'un des domaines où l'évolutivité présente des défis importants. Traditionnellement, la mise en cache était considérée comme un moyen simple d'améliorer les performances en stockant les données fréquemment consultées dans une zone d'accès rapide. Cependant, à mesure que les systèmes gagnent en complexité et en échelle comme nous le voyons souvent dans les architectures composables, la gestion de la mémoire cache devient une tâche complexe, nécessitant des stratégies sophistiquées pour garantir à la fois les performances et la cohérence dans les systèmes distribués.

- **Cohérence entre les systèmes distribués**
  Dans un système distribué, s'assurer que tous les services accèdent de manière cohérente aux données mises en cache est un défi important. Toute mise à jour du cache doit être propagée à tous les nœuds, ce qui peut entraîner des temps de latence et une certaine complexité.
- **Invalidation du cache**

La détermination du moment où il faut invalider ou rafraîchir le cache est un autre problème critique. Des données périmées peuvent entraîner des réponses incorrectes, tandis qu'une invalidation excessive peut dégrader les performances. Politiques d'éviction du cache : Le choix de la bonne politique d'éviction est crucial pour équilibrer l'utilisation de la mémoire et la fraîcheur des données. Parmi les politiques les plus courantes, on peut citer la méthode LRU (Least Recently Used), la méthode FIFO (First In, First Out) et la méthode TTL (Time-To-Live), chacune d'entre elles présentant des avantages et des inconvénients.

- **Partage de la mémoire cache**

Lorsque le volume de données augmente, il devient nécessaire de répartir le cache sur plusieurs nœuds. Cependant, cela introduit une complexité dans la gestion et l'accès aux données partagées. Solutions pour surmonter les dilemmes de la gestion du cache

### *1.4. Les problèmes de la gestion des caches dans les systèmes évolutifs*

#### 1.4.1. Frameworks de gestion de cache distribué

L'utilisation de cadres de gestion de cache distribués peut simplifier la gestion du cache dans les systèmes évolutifs. Ces frameworks fournissent des mécanismes intégrés pour la réplication des données, le sharding et les politiques d'éviction, réduisant ainsi la complexité de la gestion du cache dans les systèmes distribués. Cela permet de répartir la charge de la mise en cache sur plusieurs nœuds et d'éviter les goulets d'étranglement.

Un système de cache distribué consiste à répartir l'infrastructure de mise en cache sur plusieurs nœuds ou serveurs d'un réseau. Cette approche présente plusieurs avantages lorsqu'il s'agit de gérer les complexités de l'évolutivité :

- **Répartition de la charge :** En divisant le cache sur plusieurs nœuds, la charge de la mise en cache est uniformément répartie, ce qui évite qu'un seul nœud ne devienne un goulot d'étranglement. Cela garantit que les opérations de mise en cache peuvent s'adapter horizontalement à la demande croissante de votre système.

- **Tolérance aux pannes :** Les systèmes de cache distribués améliorent la tolérance aux pannes. Si un nœud de cache tombe en panne, les autres peuvent toujours répondre aux demandes, ce qui permet de résister aux défaillances matérielles ou aux pannes temporaires.
- **Évolutivité :** Au fur et à mesure que votre application se développe, vous pouvez facilement faire évoluer un système de cache distribué en ajoutant des nœuds supplémentaires. Cette évolutivité vous permet de gérer des charges de travail plus importantes et de maintenir des performances optimales.
- **Réduction de la latence :** La mise en cache distribuée peut réduire la latence en rapprochant les données des instances d'application qui en ont besoin. Ceci est particulièrement avantageux dans les systèmes distribués géographiquement où l'accès à un cache centralisé peut introduire un temps de latence important.
- **Cohérence et homogénéité :** La coordination de la cohérence du cache devient plus facile à gérer avec un système distribué. Des techniques avancées de gestion du cache, comme l'invalidation du cache distribué ou les protocoles de cohérence, peuvent être mises en œuvre pour garantir que tous les nœuds disposent de données actualisées et cohérentes.
- **Architecture décentralisée :** La nature décentralisée de la mise en cache distribuée s'aligne bien sur les principes des micro-services et des systèmes distribués. Chaque micro-service ou composant peut avoir son propre cache, ce qui minimise les dépendances entre services.

Les solutions de cache distribué les plus populaires sont Apache Ignite, Redis Cluster et Memcached. Ces systèmes fournissent des mécanismes de distribution, de réplication et de gestion des données de cache sur plusieurs nœuds, ce qui permet une mise en cache efficace et évolutive dans le contexte d'applications à grande échelle.

### 1.4.2. Techniques d'invalidation des caches

Les techniques d'invalidation de la mémoire cache sont essentielles au maintien de la cohérence des données dans un système de mise en cache. Lorsqu'il s'agit de gérer les complexités de scalabilité, la mise en œuvre de stratégies efficaces d'invalidation de la mémoire cache devient essentielle. Voici un aperçu des techniques d'invalidation de la mémoire cache :

- **Expiration basée sur le temps :** Fixer des limites de temps pour la validité des données mises en cache. Après une période prédéfinie, le contenu mis en cache est considéré comme expiré et doit être actualisé. Cela garantit que le cache est régulièrement mis à jour, réduisant ainsi la probabilité de servir des informations obsolètes. L'adoption de politiques dynamiques d'éviction du cache qui s'adaptent à la charge actuelle et aux schémas d'accès peut améliorer l'efficacité du cache. Par exemple, l'utilisation d'un TTL à fenêtre glissante qui s'ajuste en fonction de la taille actuelle du cache et des schémas d'accès peut aider à maintenir un équilibre entre la fraîcheur des données et l'utilisation de la mémoire.
- **Invalidation en fonction des événements :** Invalider les entrées du cache en réponse à des événements spécifiques ou à des changements dans les données sous-jacentes. Lorsque des données sont mises à jour, supprimées ou ajoutées, les entrées correspondantes du cache sont invalidées, ce qui garantit que le cache reflète l'état le plus récent des données.
- **Versioning :** Attribuer des versions aux données mises en cache et mettre à jour la version lorsque les données changent. Cela permet aux clients de vérifier la version avant d'utiliser les

données mises en cache. Si la version a changé, le client sait qu'il doit récupérer les données les plus récentes, ce qui maintient la cohérence.

- **Mise en cache par écriture :** Dans une stratégie de mise en cache par écriture, les mises à jour ou les écritures dans le magasin de données sous-jacent déclenchent des mises à jour simultanées dans le cache. Cette approche garantit que le cache est toujours synchronisé avec les données les plus récentes, minimisant ainsi le risque de servir des informations périmées.
- **Mise en cache en aval de l'écriture :** La mise en cache en aval de l'écriture consiste à mettre à jour le cache de manière asynchrone après avoir mis à jour le magasin de données. Bien que cela puisse entraîner un léger retard dans les mises à jour du cache, cela peut améliorer les performances globales du système en découplant les opérations d'écriture des mises à jour du cache.
- **Jetons ou clés d'invalidation :** Utilisez des jetons ou des clés associés aux données mises en cache qui peuvent être explicitement invalidées en cas de modification. Cette approche ciblée permet d'invalider des entrées spécifiques du cache sans affecter l'ensemble du cache, ce qui réduit le risque de goulots d'étranglement.
- **Invalidation basée sur des modèles :** Identifier des modèles ou des règles pour l'invalidation des caches en fonction des modèles d'utilisation ou de la logique d'entreprise. Par exemple, l'invalidation des caches associés à des rôles d'utilisateurs spécifiques ou à des sous-ensembles de données peut aider à adapter la gestion du cache aux besoins spécifiques de l'application.
- **Diffusion globale des invalidations :** Mettre en œuvre un mécanisme de diffusion des messages d'invalidation du cache au niveau global dans le système distribué. Cela permet de s'assurer que tous les nœuds sont au courant des changements, ce qui facilite la coordination des mises à jour du cache et le maintien de la cohérence.

En combinant ces techniques d'invalidation de cache, vous pouvez concevoir un système de mise en cache résilient qui gère efficacement les complexités introduites par la scalabilité. L'essentiel est de choisir ou de combiner les techniques en fonction des exigences et des caractéristiques spécifiques de votre application et de vos données.

### 1.4.3. Protocoles de cohérence de la mémoire cache

Les protocoles de cohérence de la mémoire cache sont des mécanismes essentiels utilisés dans les systèmes distribués pour garantir que les données mises en cache restent cohérentes entre les différents nœuds ou caches. Au fur et à mesure que les systèmes s'étendent et deviennent distribués, le maintien de la cohérence devient un défi en raison de la possibilité pour plusieurs nœuds de mettre en cache les mêmes données de manière indépendante. Voici une explication plus détaillée des protocoles de cohérence du cache :

- **Mise en cache par écriture et mise en cache en aval de l'écriture :** Ces techniques d'invalidation de cache permettent aussi, par nature, de garder une cohérence. Bien que l'approche par écriture maintienne la cohérence, elle peut entraîner un temps de latence plus élevé pour les opérations d'écriture. À contrario, la méthode en aval de l'écriture peut améliorer les performances des opérations d'écriture en découplant les mises à jour du cache de l'écriture immédiate dans le stockage sous-jacent. Cependant, il nécessite une gestion minutieuse pour maintenir la cohérence des données.
- **Cohérence basée sur l'invalidation :** Les protocoles de cohérence basés sur l'invalidation se concentrent sur l'invalidation explicite des données mises en cache lorsque des changements

se produisent dans le magasin de données. Lorsque des données sont modifiées ou mises à jour, les entrées correspondantes du cache sont marquées comme non valides et les requêtes ultérieures déclenchent la récupération des données les plus récentes. Cette approche permet de maintenir la cohérence, mais nécessite des mécanismes efficaces d'invalidation.

- **Écriture unique, lecture multiple (WORM) :** Les protocoles d'écriture unique et de lecture multiple conviennent aux scénarios dans lesquels les données sont principalement lues et rarement modifiées. Une fois que les données sont écrites, elles sont considérées comme immuables, ce qui réduit le besoin de mises à jour fréquentes du cache. Cette approche simplifie la gestion de la cohérence mais peut ne pas convenir à des données très dynamiques.
- **Validation en deux phases :** Dans les systèmes distribués, un protocole de validation en deux phases est parfois utilisé pour garantir l'atomicité des opérations d'écriture sur plusieurs nœuds. Ce protocole implique qu'un nœud coordinateur interagisse avec tous les nœuds pour confirmer qu'ils sont prêts avant de valider une écriture. Bien qu'il soit efficace pour maintenir la cohérence, il introduit une surcharge de coordination supplémentaire.
- **Systèmes basés sur le quorum :** Certaines bases de données distribuées utilisent des systèmes basés sur le quorum pour les lectures et les écritures. Dans cette approche, un certain nombre de nœuds doivent s'accorder sur la validité d'une opération pour qu'elle soit considérée comme réussie. Cela permet de maintenir la cohérence dans des scénarios où tous les nœuds ne sont pas disponibles ou réactifs.
- **Horloges vectorielles :** Les horloges vectorielles sont utilisées pour suivre la causalité entre les événements distribués. Elles sont particulièrement utiles dans les systèmes où l'ordre des événements est essentiel au maintien de la cohérence. Les horloges vectorielles permettent d'identifier l'ordre relatif des événements entre les nœuds.

La mise en œuvre d'un protocole de cohérence de cache approprié dépend des exigences et des caractéristiques spécifiques du système distribué. Il est essentiel de trouver un équilibre entre la cohérence et les performances, et le choix du protocole implique souvent des compromis basés sur le cas d'utilisation et la charge de travail de l'application.

### 1.4.4. Monitoring et Metrics

L'utilisation d'outils de surveillance et d'optimisation peut fournir des informations sur les performances du cache et aider à identifier les goulots d'étranglement ou les inefficacités. Ces outils peuvent aider à ajuster les paramètres et les politiques du cache en temps réel, garantissant ainsi des performances optimales. Voici une explication plus détaillée de l'importance et de la mise en œuvre de la surveillance et des mesures dans un environnement évolutif :

- **Visibilité en temps réel :** La mise en œuvre de systèmes de surveillance permet d'obtenir une visibilité en temps réel sur divers aspects de votre système distribué. Cela comprend l'utilisation des ressources, les temps de réponse, les taux d'erreur et d'autres indicateurs clés de performance. Les informations en temps réel permettent d'identifier et de résoudre rapidement les problèmes.
- **Identification des goulots d'étranglement :** Les métriques permettent d'identifier les goulots d'étranglement en suivant l'utilisation des ressources et le comportement du système. En surveillant l'utilisation de l'unité centrale, la consommation de mémoire, la latence du réseau et d'autres mesures pertinentes, vous pouvez identifier les domaines qui nécessitent une optimisation ou une mise à l'échelle.

- **Planification de la scalabilité :** La surveillance et les mesures contribuent à la planification de l'évolutivité en fournissant des données sur la croissance du système et les tendances d'utilisation des ressources. Ces informations permettent de prendre des décisions éclairées sur l'évolution des ressources, qu'il s'agisse d'ajouter des serveurs, d'ajuster les configurations ou d'optimiser le code.
- **Mécanismes d'alerte :** Mettre en place des mécanismes d'alerte en fonction de seuils prédéfinis ou d'anomalies détectées dans les mesures. Cela permet d'apporter des réponses proactives aux problèmes potentiels, de minimiser les temps d'arrêt et d'assurer la fiabilité de votre système distribué.
- **Tracing distribué :** Mettre en œuvre le traçage distribué pour suivre le flux des requêtes et identifier les goulots d'étranglement des performances à travers plusieurs composants ou services. Ceci est particulièrement utile dans les architectures microservices où la compréhension du flux de bout en bout d'une requête est essentielle pour l'optimisation.
- **Surveillance de l'expérience utilisateur :** Les mesures liées à l'expérience utilisateur, telles que les temps de réponse et les taux d'erreur au niveau de l'application, donnent un aperçu de la manière dont les performances du système ont un impact direct sur les utilisateurs finaux. Le suivi de l'expérience utilisateur permet de hiérarchiser les améliorations qui renforcent la satisfaction générale.
- **Mesures d'utilisation des ressources :** Suivre les mesures d'utilisation des ressources pour chaque nœud du système distribué, y compris l'utilisation de l'unité centrale, de la mémoire, du disque et du réseau. Ces informations permettent de s'assurer que les ressources sont allouées de manière appropriée et peuvent guider les décisions relatives à la mise à l'échelle ou à l'optimisation de composants spécifiques.
- **Analyse historique :** Analyser l'historique des données métriques permet un examen rétrospectif du comportement du système. Elle permet d'identifier des modèles, de comprendre les tendances à long terme et de prendre des décisions fondées sur des données pour améliorer le système en permanence.
- **Optimisation des coûts :** La surveillance peut contribuer à l'optimisation des coûts en identifiant les ressources sous-utilisées ou les domaines dans lesquels l'allocation des ressources peut être ajustée en fonction de la demande réelle. Ceci est particulièrement pertinent dans les environnements cloud où les coûts des ressources sont souvent liés à l'utilisation.
- **Tableaux de bord complets :** Utiliser des tableaux de bord complets pour visualiser les mesures clés et les indicateurs de performance. Les tableaux de bord fournissent une vue d'ensemble de l'état et des performances du système, ce qui facilite la prise de décision et le dépannage.

En investissant dans des systèmes de surveillance et de mesure robustes, vous permettez à votre équipe d'adopter une attitude proactive vis-à-vis de la santé du système, d'identifier les problèmes à un stade précoce et d'optimiser en permanence les performances de votre système distribué évolutif.

## Résumé

- Les processeurs actuels (**x86**, ARM, RISC-V) optimisent le compromis performance/consommation via l'exécution hors-ordre, la prédiction de branches et les jeux d'instructions vectoriels (**SIMD**).
- **Multi-cœurs et parallélisme** : Face aux limites de fréquence, l'augmentation du nombre de cœurs (2 à 128+) permet d'exécuter plusieurs tâches ou threads simultanément, en exploitant le parallélisme au niveau instructions, données et threads.
- **Mémoire cache et hiérarchie** : Une hiérarchie de caches (L1 rapide mais petite, L2, L3 plus lente mais grande) réduit les accès à la RAM lente. Des protocoles de cohérence garantissent la synchronisation entre cœurs.

## Conclusion

Le domaine du traitement parallèle concerne les méthodes architecturales et algorithmiques permettant d'améliorer les performances des ordinateurs numériques ou d'autres critères tels que la rentabilité et la fiabilité grâce à diverses formes de concurrence. Même si le calcul concurrent existe depuis les premiers jours des ordinateurs numériques, ce n'est que récemment qu'il a été appliqué d'une manière permettant d'obtenir de meilleures performances, ou un meilleur rapport coût-efficacité, par rapport aux superordinateurs vectoriels.

Ce chapitre représente d'une manière simplifiée les principes de bases liées aux architectures parallèles en abordant brièvement leurs concepts, leurs caractéristiques, la classification de ce type d'architectures ainsi que l'approche de programmation de chacune des classes.

Face aux défis de la scalabilité, la gestion de la mémoire cache est passée d'une simple amélioration des performances à un problème complexe aux multiples facettes. En s'appuyant sur des frameworks de gestion de cache distribués, en adoptant l'invalidation de cache pilotée par les événements, en mettant en œuvre des politiques d'éviction de cache dynamiques et en employant des stratégies de partage de cache efficaces, les équipes IT peuvent résoudre les dilemmes de la gestion de cache dans les systèmes évolutifs. À mesure que les systèmes continuent de croître et d'évoluer, l'importance de stratégies robustes de gestion de la mémoire cache ne fera qu'augmenter, soulignant la nécessité d'une innovation et d'une optimisation continues dans ce domaine.

# Conclusion générale

À travers ces différents chapitres, nous avons parcouru l'évolution progressive de l'unité centrale et des processeurs, depuis les principes de base de l'organisation matérielle d'un ordinateur jusqu'aux architectures les plus récentes. Le premier chapitre a permis de comprendre la structure générale de l'unité centrale et le rôle essentiel de ses principaux composants dans le traitement des informations. Le deuxième chapitre a ensuite présenté l'architecture interne des processeurs, en mettant en évidence les mécanismes fondamentaux qui assurent l'exécution des instructions et la coordination des opérations.

L'étude du processeur Intel 8086, dans le troisième chapitre, a offert un exemple concret d'architecture historique marquante, à la fois simple dans ses principes et déterminante dans l'évolution des microprocesseurs. Le quatrième chapitre a montré la transition vers les processeurs 32 bits et 64 bits, illustrant l'augmentation des performances, de la capacité de traitement et de l'espace adressable. Enfin, le cinquième chapitre a permis d'ouvrir la réflexion sur les architectures des processeurs récents, caractérisées par la miniaturisation, le multicœur, la parallélisation et l'optimisation énergétique.

En définitive, cette progression met en évidence que l'évolution des processeurs répond constamment à un double objectif : augmenter la puissance de calcul et améliorer l'efficacité de traitement. Elle montre aussi que les architectures modernes ne sont pas seulement plus rapides, mais aussi plus intelligentes, plus flexibles et mieux adaptées aux besoins actuels de l'informatique, qu'il s'agisse du multimédia, de l'intelligence artificielle, des systèmes embarqués ou du traitement massif des données.

# Séries TD avec corrections

# Série de TD N°1 :

**Exercice 1 :**1. Donnez la différence entre le mot « information » et « Informatique »,

2. Reliez entre les mots suivants :

| | |
|---|---|
| *Horloge* | *Unité Centrale de Traitement* |
| *Circuit intégré* | *RAM* |
| *Registre* | *Unités d'entrée sorties* |
| *Carte Mère* | *Microprocesseur* |
| *Bus* | *Périphériques* |

2. Quelle est la signification des acronymes suivants :

- CPU: ……
- UAL: ……
- RAM: ……
- ROM: ……
- UC: ……
- BIOS: ……

**Exercice 2 :** Répondez aux questions suivantes :

1- Qu'est ce qui caractérise la machine de Von Neumann par rapport aux machines qui existaient avant ? même question pour l'architecture Harvard ?
2- Donnez le schéma des deux architectures ?
3- Quels sont les éléments de base d'un système informatique ?
4- Quel est le rôle d'un système d'exploitation « logiciel de système » dans un système informatique ?
5- Citez les composants principaux d'un ordinateur ?
6- Citez les différences entre les différents types de bus ?

**Exercice 3:** Répondez aux questions suivantes :

**Etant donné :**

**Capacité = $2^n$ Mots mémoire = $2^n$ * m Bits**

**Capacité d'une mémoire = Nombre de mots * Taille du mot**

**Nombre de mots = $2^{\text{nombre de lignes d'adresses}}$**

**Taille du mot (en bits) = nombre lignes de données**

1. Notre mémoire a une capacité de 8 mots de 16 bits chacun. On exprime également cette Capacité en nombre d'octets ou de bits.
2. Notre mémoire a 25 lignes d'adresse et 1200 lignes de données, trouvez le nombre des mots mémoire ainsi la taille de chaque mot en bits et en octets
3. Dans une mémoire la taille du bus d'adresses K=16 et la taille du bus de données N=8. Calculer la capacité de cette mémoire ?

# Correction de TD N°1 :

**Exercice 1 :**1. Donnez la différence entre le mot « information » et « Informatique »,

Information : les données à lesquelles en ajoute de sens

Informatique : traitement automatique des informations

2. Reliez entre les mots suivants :

| | |
|---|---|
| *1.Horloge ( D)* | *A. Unité Centrale de traitement* |
| *2. Circuit intégré (A)* | *B. RAM* |
| *3. Registre (D)* | *C. Unités d'entrée sorties* |
| *4. Carte Mère* (A, B, C, D, E) | *D. Microprocesseur* |
| *5. Bus* (A, B, C, D, E) | *E. Périphériques* |

3. Quelle est la signification des acronymes suivants :

- CPU: Central Processing Unit
- UAL: Unit Arithmétique et logique…
- RAM: Random Access Memory……
- ROM: Read Only Memory……
- UC: …Unité Centrale……
- BIOS: …Basic Input Output System……

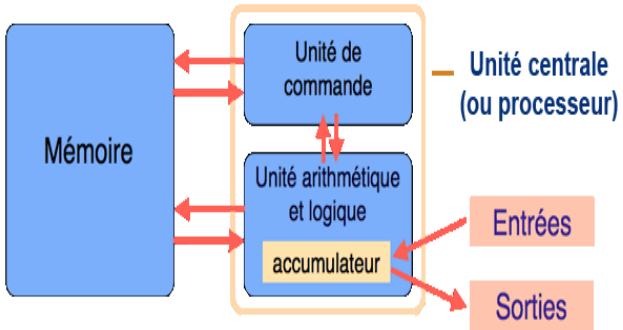
**Exercice 2 :** Répondez aux questions suivantes :

1-La machine de Von Neumann par rapport aux machines qui existaient avant : La machine de Von Neumann a introduit la notion de mémoire (Programmes et données sont stockés dans la même mémoire)

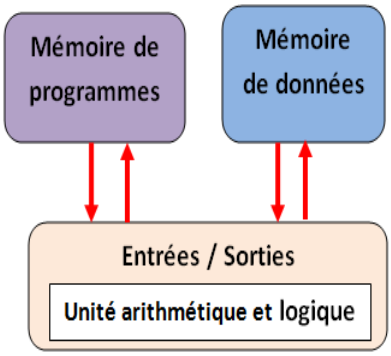
✓ Pour l'architecture Harvard : Mémoire de programme séparées de la mémoire des données **(Traitement des données vidéo et audio)**

2- Le schéma des deux architectures :

**L'architecture Von Neumann**




**L'architecture Harvard**




3- Les éléments de base d'un système informatique ?

- ✓ **HARDWARE : Matériels (Périphériques d'entrée/sortie)**
- ✓ **SOFTWARE : Logiciels System (Windows, MSDOS)+ Logiciels Application (Word, Excel……)**

4- Le rôle d'un système d'exploitation «  logiciel de système » dans un système informatique : Il joue le rôle d'interpréteur entre la machine et l'utilisateur.
5- Les composants principaux d'un ordinateur : L'unité de traitement, l'unité de stockage « la mémoire centrale (RAM) », les unités d'entrées/ sorties (Les périphériques)
6- Citez les différences entre les différents types de bus : Les bus de données «  bidirectionnel », ( transportent les données) entre « microprocesseur et mémoire centrale »  et d'adresses « unidirectionnel », ( transportent les adresses) entre « microprocesseur et mémoire centrale » et les bus de commande «  bidirectionnel » ( transportent les commandes entre les différents unités )  en **synchronisation** des flux d'informations sur les bus donnés ou adresses.

**Exercice 3:** Répondez aux questions suivantes :
Etant donné :

2- Notre mémoire a une capacité de 8 mots de 16 bits chacun. On exprime également cette Capacité en nombre d'octets ou de bits.
Notre mémoire a donc une capacité de (8*2 octets) **16 octets** ou de (8*16 bits) **128 bits.**

3- Notre mémoire a 25lignes d'adresse et 1200 lignes de données, trouvez le nombre des mots mémoire ainsi la taille de chaque mot en bits et en octets :

**Nombre de mots = $2^{\text{nombre de lignes d'adresses}}$**

**Nombre de mots= $2^{25}$ = 33554432 mots de 8bits**

**Taille du mot (en bits) = 1200 bits = 1200/8= 150 octets**

4- Dans une mémoire la taille du bus d'adresses K=16 et la taille du bus de données N=8. Calculer la capacité de cette mémoire ?
**Taille du mot (en bits) =** la taille du bus de données N=**8 bits**
**Nombre de mots = $2^{16}$ = 65536 bits**
**Capacité d'une mémoire = Nombre de mots * Taille du mot = $2^{16}$ * 8= $2^{16}$ * $2^3$ =$2^{19}$bits**
**$2^{19}$bits= $2^{16}$ octets =$2^{13}$ K octets**

# Série de TD N°2

**Exercice 1 :**1. Trouver l'équivalent binaire de chacun des nombres suivants :

- $(312)_{10}$, $(39)_{10}$, $(22)_{10}$, $(932)_{10}$, $(432)_{10}$, $(1034)_{10}$, $(5230)_{10}$, $(4390)_{10}$
- $(352)_8$, $(1462)_8$, $(5232)_8$, $(1045)_8$, $(53)_8$, $(3650)_8$, $(15,54)_8$, $(447)_8$
- $(F560)_{16}$, $(DD8)_{16}$, $(1A0B)_{16}$, $(1F2)_{16}$, $(77C9)_{16}$, $(EB77)_{16}$, $(5AB01)_{16}$

2. Quelle est la valeur décimale des nombres suivants :

- $(1\ 0011)_2$, $(1101)_2$, $(0000\ 0111)_2$, $(111)_2$, $(10110)_2$, $(100101011)_2$, $(11100100)_2$ , $(1001,101)_2$
- $(333)_8$, $(175)_8$, $(627)_8$ , $(4721)_8$ , $(342)_8$ , $(745,05)_8$ , $(530, 123)_8$ , $(3412, 87)_8$ , $(678,44)_8$
- $(A4B)_{16}$ , $(5AC)_{16}$ , $(EF1)_{16}$ , $(59D)_{16}$ , $(BB30)_{16}$, $(4F26)_{16}$, $(90D8)_{16}$, $(5CE7)_{16}$ , $(AB62E)_{16}$

3. Convertir en décimale puis en hexadécimale les valeurs suivantes :

- $(11101010)_2$, $(1100110010)_2$, $(101010011010)_2$, $(001000100101)_2$, $(110110001011)_2$
- $(342)_8$, $(745,05)_8$, $(322)_8$, $(2734)_8$, $(226)_8$, $(3742)_8$, $(4266)_8$, $(2025)_8$
- $(2011)_{10}$, $(342)_{10}$, $(2478)_{10}$, $(5634)_{10}$, $(449)_{10}$, $(5266)_{10}$, $(24567)_{10}$

**Exercice 2 :**1. Calculer les opérations suivantes :

1010+0101= ? 10110-1100= ? 1011*11= ?

1101001 + 10101= ? 10111000 + 1001000= ? 1001111 x 110100= ?

2- Transformez les résultats des opérations précédentes en décimale et en hexadécimale.

Etant donné les tableaux suivants :

| Décimal | Binaire | Octal |
|---|---|---|
| 0 | 000 | 0 |
| 1 | 001 | 1 |
| 2 | 010 | 2 |
| 3 | 011 | 3 |
| 4 | 100 | 4 |
| 5 | 101 | 5 |
| 6 | 110 | 6 |
| 7 | 111 | 7 |

| Décimal | Binaire | Hexadécimal |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |

| | | |
|---|---|---|
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

# Correction série de TD N°2 :

**Exercice 1 :**1. Ecrivez en binaire.

1) La base décimale : On obtient avec une division successive sur 2

- $(312)_{10}$= 10011 $1000_2$
- $(39)_{10}$ =10 $0111_2$
- $(22)_{10}$= $(10110)_2$ $(1034)_{10}$= $(10000001010)_2$
- $(932)_{10}$=$(1110100100)_2$ $(5230)_{10}$ = $(1010001101110)_2$
- (432)10= (110110000)2 $(4390)_{10}$ = $(1000100100110)_2$
-

2) La base octale : En utilisant le tableau de vérité

- $(352)_8$=$(011101010)_2$, $(1462)_8$= $(001100110010)_2$, $(5232)_8$= $(101010011010)_2$, $(1045)_8$= $(001000100101)_2$ $(53)_8$ =$(100011)_2$ $(3650)_8$=$(011110101000)_2$ $(15,54)_8$ =$(001101, 101100)_2$ $(447)_8$= $(100100111)_2$

3) La base héxadécimale : En utilisant le tableau de vérité

- $(F560)_{16}$=$(1111010101100000)_2$
- $(DD8)_{16}$=( $110111011000)_2$
- $(1A0B)_{16}$= $(0001101000001011)_2$
- $(1F2)_{16}$=$(000111110010)_2$
- $(77C9)_{16}$, =$(0111011111001001)_2$
- $(EB77)_{16}$=$(1110101101110111)_2$
- $(5AB01)_{16}$=$(01011010101100000001)_2$

2- Quelle est la valeur décimale des nombres binaire suivants

////La base binaire vers décimale

- $(1001,101)_2 = 1x2^3 + 0x2^2 + 0x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} = 9 + 0,5 + 0,125$=(1001,101)2 = $(9,625)_{10}$
- 1101= $(13)_{10}$ 10011= $(19)_{10}$

- $(111)_2$=$1*2_0$+$1*2_1$+$1*2_2$=$(7)_{10}$ $(00000111)_2$ = $(7)_{10}$
- $(10110)_2$=$(22)_{10}$ $(100101011)_2$=$(299)_{10}$ $(11100100)$=$(228)_{10}$

////// La base octale vers décimale

$X_1 = 3x8^2 + 4x8^1 + 2x8^0 = 192 + 32 + 2 = 226$

$X_1 = (342)_8 = (226)_{10}$

- $(333)_8$= $(21)_{10}$ $(175)_8$=$(125)_{10}$ $(627)_8$=$(407)_{10}$ $(4721)_8$=$(2513)_{10}$
- $(745,05)_8 = 7x8^2 + 4x8^1 + 5x8^0 + 0x8^{-1} + 0x8^{-2} = 448 + 32 + 5 + 0 + 0,078125$ $(745,05)_8 = (485,078125)_{10}$ $(530,123)_8 = (344, 162)_{10}$
- $(3412, 87)_8$ , $(678,44)_8$ Impossible
-
-
- //////// La base hexadécimale vers décimale

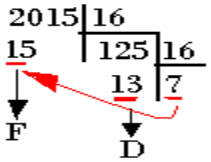
- $(A4B)_{16}$= (101001001011)2 =$(2635)_{10}$ $(5AC)_{16}$= $(10110101100)_2$ = $(1452)_{10}$ $(EF1)_{16}$= (111011110001)2= $(3825)_{10}$ $(59D)_{16}$= (10110011101)2=$(1437)_{10}$

- $(BB30)_{16}$= $(1011101100110000)_2$ = $(47920)_{10}$
- $(4F26)_{16}$= $(100111100100110)_2$ =$(20262)_{10}$
- $(90D8)_{16}$ =$(1001000011011000)_2$= $(37080)_{10}$
- $(5CE7)_{16}$ =$(101110011100111)_2$= $(23783)_{10}$
- $(AB62E)_{16}$= $(10101011011000101110)_2$=$(701998)_{10}$

3. Convertir en **décimale** puis en hexadécimale les valeurs suivantes

**$(342)_8$** = $X_1 = 3x8^2 + 4x8^1 + 2x8^0 = 192 + 32 + 2 = (226)_{10} = (E2)_{16}$

**Exemple3: convertir le nombre 2015 en base 16**




$(2015)_{10} = (7DF)_{16}$

$(742,05)_8$ = $(482,078125)_{10}$ = $(1E5, 14)_{16}$

$(322)_8$=$(210)_{10}$= $(D2)_{16}$ $(2734)_8$=$(1500)_{10}$= $(5DC)_{16}$ $(226)_8$= $(150)_{10}$=$(96)_{16}$

$(3742)_8$=$(2018)_{10}$=$(7E2)_{16}$ $(4266)_8$=$(2230)_{10}$=( $8B6)_{16}$
$(2025)_8$=$(1045)_{10}$=( $415)_{16}$

////////////////////////////////////////

- $(1110/1010)_2$=$(EA)_{16}$ = $(234)_{10}$ $(11/0011/0010)_2$=$(332)_{16}$ = $(818)_{10}$
- $(1010/1001/1010)_2$=$(A9A)_{16}$= $(2714)_{10}$
- $(10/0010/0101)_2$ =$(225)_{16}$ = $(549)_{10}$ ( $1101/1000/1011)_2$= $(D8B)_{16}$=$(3467)_{10}$

**Exercice 2 :**

1010+0101= 1111 = $(15)_{10}$ = $(F)_{16}$ 10110-1100= 01010 =$(10)_{10}$= $(A)_{10}$

1011*11= 100001= $(33)_{10}$=$(21)_{16}$

1101001 + 10101=1111110 =(126) 10=(7E)16
10111000+ 1001000= 100000000 =(256)10=(100)16
1001111 x110100= 1000000001100= (4108) 10 =(100C)16

# Série de TD N°3

**Exercice 1 :** On considère une machine à 4 adresses, on veut exécuter l'opération suivante :

ADD A, B avec A et B deux nombres quelconques.

Sachant que :

- ✓ L'adresse en mémoire centrale de l'opération ADD est 100.
- ✓ Celle du 1er opérande est 120.
- ✓ Celle du 2ème opérande est 150.
- ✓ Celle du résultat est 200.

1. Donnez le contenu du registre instruction ?
2. Donnez le contenu de la mémoire centrale pour A= 36 et B=33 ?
3. Reprendre les mêmes données en utilisant une machine à deux adresses, donnez le contenu du registre instruction ? Donnez le contenu de la mémoire centrale avant et après l'exécution de l'instruction ?

**Exercice 2 : 1-** On considère qu'une pile initialement vide, écrivez le programme qui fait le calcul suivant :

5+7*4

**2-** En utilisant le jeu d'instruction de la machine à zéro adresse donnée en cours :

1. Donnez la suite d'instructions pour l'exécution de l'opération suivante : Y= a- (b*c)
2. Donnez le contenu de la pile pour les valeurs suivante : a= 10, b= 2, c=3

**Exercice 3 :** Donnez le contenu du registre de destination dans chaque cas qui suit :

- ✓ **Cas 1 : adressage immédiat**

  MOVE R1, #200

| MOVE | R1 | 200 |
|---|---|---|

- ✓ **Cas 2 : adressage direct**

  MOVE R1, 200

| | |
|---|---|
| **200** | 230 |

- ✓ **Cas 3 : adressage indirect**

  MOVE R2, (R1)

| MOVE | R1 | (R1) |
|---|---|---|

| | |
|---|---|
| **200** | 230 |
| **230** | 400 |
| | ..... |

# Correction série de TD N°3

**Exercice 1 : 1) Contenu du registre RI :**

| ADD | 120 | 150 | 200 | 101 |
|---|---|---|---|---|

**2) Contenu de la mémoire centrale :**

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| 100 | ADD | 120 | 150 | 200 | 101 |
| 101 | Instruction suivante | | | | |
| | | | | | |
| 120 | 36 | | | | |
| | | | | | |
| 150 | 33 | | | | |
| | | | | | |
| 200 | 69 | | | | |

3) Le contenu du registre instruction,

| ADD | 120 | 150 |
|---|---|---|

Le contenu de la mémoire centrale avant et après l'exécution de l'instruction une machine à deux adresses

**Avant l'exécution**

| | | | |
|---|---|---|---|
| | | | |
| 100 | ADD | 120 | 150 |
| | | | |
| 120 | 36 | | |
| | | | |
| 150 | 33 | | |
| | | | |

**Après l'exécution**

| | | | |
|---|---|---|---|
| | | | |
| 100 | ADD | 120 | 150 |
| | | | |
| 120 | 36 | | |
| | | | |
| 150 | 69 | | |
| | | | |

Le contenu du compteur ordinal : 101

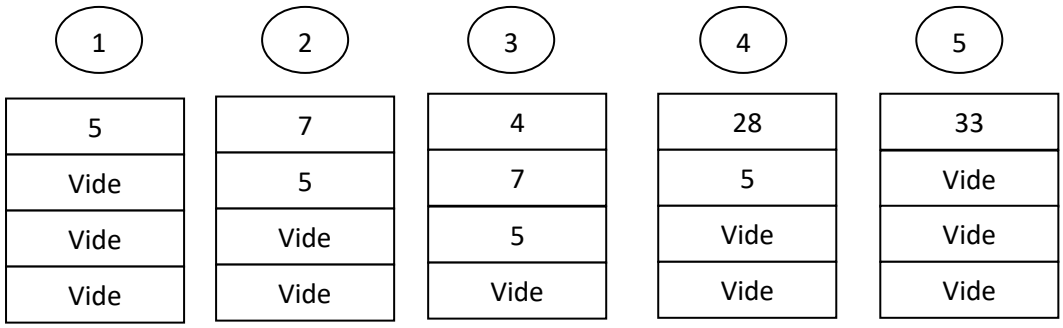
**Exercice 2 : 1-** On considère qu'une pile initialement vide, écrivez le programme qui fait le calcul suivant : 5+7*4

**Programme :**

1. **PUSH #5**
2. **PUSH #7**
3. **PUSH #4**
4. **MUL /* Multiplier 7 par 4 et écrit 28**
5. **ADD /* Additionne 28 et 5 et écrit**

PUSH , POP avec les valeurs constantes

**Contenu de la pile :**




**2-** En utilisant le jeu d'instruction de la machine à zéro adresse donnée en cours :
Y= a- (b*c) ou : a= 10, b= 2, c=3

**Programme :**

1. **LOAD a**
2. **LOAD b**
3. **LOAD c**
4. **MUL /* Multiplier b par c**
5. **SUB /* soustraire le résultat de multiplication de a**
6. **STORE Y /* Charger et récupérer le résultat final**

LOAD, STORE avec les valeurs variables

**Exercice 3 :**

- ✓ **Cas 1 : adressage immédiat**
  **R1 contient la valeur 200**
- ✓ **Cas 2 : adressage direct**
  **R1 contient la valeur 230**
- ✓ **Cas3 : adressage indirect**
  **R2 contient la valeur 400**

# Série de TD N°4

**Exercise 1: 1**- write a program that performs the following calculation Équation : x = \frac{b - c}{a}

(variables en mémoire : A, B, C ; résultat en X)

2-Équation : Delta = b^2 - 4ac

(variables A, B, C en mémoire ; résultat DELTA)

3- Moyenne pondérée

Équation : M = \frac{a \cdot x + b\cdot y}{a + b} (\frac= / ) , (\cdot= * )

(variables A, B, X, Y en mémoire ; résultat MOY)

**Exercice 2 :**

Rappelons qu'un segment de mémoire est une zone mémoire adressable avec une valeur fixée de su segment (les 16 bits de poids fort pour un processeur 32 bits). Dans ce cas un segment a donc une taille maximale de 64 Ko. Mieux, dans un adressage indirect, l'adresse physique sur 32 bits est trouvée de la manière suivante :

**Adresse physique= 16* segment+déplacement**

1. Trouvez l'adresse physique référencée par 047C : 003B, ou 047C représente l'adresse du segment et 003B représente l'adresse de déplacement ?
2. On souhaite stocker le contenu du registre AH en mémoire en utilisant : MOV[1200h], AH sachant que (DS=720h) , trouvez l'adresse physique ?

**Exercice 3 :** Indiquer pour chacune des instructions suivantes le mode d'adressage :

```
MOV AL, [000BH]
ADD AL, C4H
MOV [BX], 00H
SUB AL, [BX+SI]
MOV AX, 9
MOV 9, AX
MOV AX, BX
MOV AX, N1 ( N1 est une variable)
MOV AX, [BX]
MOV AX, [BX+2]
```

# Correction Série de TD N°4

## Exercise 1

**-1-**

LOAD B ; empile b

LOAD C ; empile c

SUB ; b - c → sommet

LOAD A ; empile a

DIV ; (b-c) / a → sommet

STORE X

**-2-**

LOAD B ; empile b

LOAD B ; empile b à nouveau

MULT ; b * b = $b^2$ → sommet

PUSH 4 ; empile 4

LOAD A ; empile a

MULT ; 4 * a → sommet

LOAD C ; empile c

MULT ; (4*a) * c = 4ac → sommet

SUB ; $b^2$ - 4ac → sommet

STORE DELTA

**-3-**

LOAD A ; empile a

LOAD X ; empile x

MULT ; a*x → sommet

LOAD B ; empile b

LOAD Y ; empile y

MULT ; b*y → sommet

ADD ; a*x + b*y → sommet

LOAD A ; empile a

LOAD B ; empile b

ADD ; a + b → sommet

DIV ; (a*x + b*y) / (a+b) → sommet

STORE MOY

**Exercice 2 :**

Rappelons qu'un segment de mémoire est une zone mémoire adressable avec une valeur fixée de su segment (les 16 bits de poids fort pour un processeur 32 bits). Dans ce cas un segment a donc une taille maximale de 64 Ko. Mieux, dans un adressage indirect, l'adresse physique sur 32 bits est trouvée de la manière suivante :

**Adresse physique= 16* segment+déplacement**

1. Trouvez l'adresse physique référencée par 047C : 003B, ou 047C représente l'adresse du segment et 003B représente l'adresse de déplacement ?
   **Adresse physique= (16*** 047C)+ 003B=

2. On souhaite stocker le contenu du registre AH en mémoire en utilisant : MOV[1200h], AH sachant que (DS=720h) , trouvez l'adresse physique ?

**Adresse physique= (16*** 720h)+ 1200h= 08400H

**Exercice 3 :** Indiquer pour chacune des instructions suivantes le mode d'adressage :

MOV AL, [000BH] **Mode direct**
ADD AL, C4H **Mode immédiat**
MOV [BX], 00H **Mode indirect basé**
SUB AL, [BX+SI] **Mode indirect basé indéxé**
MOV AX, 9 **Mode immédiat**
MOV 9, AX **Impossible**
MOV AX, BX **Mode registre**
MOV AX, N1 ( N1 est une variable) **Mode direct**
MOV AX, [BX] **Mode indirect basé**
MOV AX, [BX+2] **Mode indirect basé**

# Série de TD N°5

**Exercice 1 : (Les instructions de branchement)**

1. Ecrivez un code assembleur qui permet de calculer la somme des chiffres de 1 à 9.

2. Ecrivez un code assembleur qui permet d'effectuer un saut si la somme de deux nombres est supérieure à 10.
3. Ecrivez un code assembleur qui permet d'effectuer un saut si un nombre est pair.

**Exercice 2 : (Boucles et conditions en assembleur)**

1. On veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul. Donnez l'organigramme qui présente le problème donné et le programme en langage assembleur associé

2. Ecrire le code (uniquement les suites d'instructions) Assembleur pour faire la somme de dix (10) éléments d'un tableau. Le premier étant rangé à la case mémoire ayant l'adresse "0F68H" ? (Chaque nombre est sur un octet et la somme peut dépasser 255)

**Exercice 3 : (Appel des procédures, adressage segmenté et gestion de la pile)**

- **Dans le programme principal :**
  1. Ecrivez un programme en utilisant les 02 instructions PUSH et POP pour la gestion d'une pile de 4 éléments.
- **Dans le sous-programme:**
  2. Ecrivez le programme Assembleur 8086 contenant les deux procédures suivantes :
  - ✓ L'affichage des éléments d'une pile
  - ✓ L'addition d'un nombre quelconque pour chaque élément

**Exercice 4: (Gestion des entrées/sorties)**

Expliquez les interruptions 21h les plus utilisés suivantes en donnant des exemples avec les instructions de l'assembleur 8086 :

- La fonction 09
- La fonction 02
- La fonction 01
- La fonction 0AH

# Correction Série de TD N°5

**Exercice 1 :** 1. Ecrivez un code assembleur qui permet de calculer la somme des chiffres de 1 à 9.

1. Ecrivez un code assembleur qui permet de calculer la somme des chiffres de 1 à 9.

**L'une des solutions pour additionner les chiffres de 1 _a 9 pourrait être la suivante:**

```
MOV AX,0
MOV BX,1
MOV CX,10
BOUCLE: ADD AX, BX     ; résultat de l'addition dans AX 1,2, 3……..9
INC BX
CMP BX, 0              comparaison jusqu'à indicateur TF achevé « boucle LOOPNE il faut une condition »
LOOPNE BOUCLE      ; do    while
```

1. Ecrivez un code assembleur qui permet d'effectuer un saut si la somme de deux nombres est supérieure à 10.

**Le code assembleur pour effectuer un saut si la somme de deux nombres est supérieure _a 10 est le suivant:**

```
 MOV AX,#nombre1
MOV BX,#nombre2
ADD AX, BX
CMP AX, 10  ; résultat dans AX
JG saut     ;  jump ou saut si  valeur de AX elle est supérieur à 10
   saut: ...
```

2. Ecrivez un code assembleur qui permet d'effectuer un saut si un nombre est pair.

**Pour vérifier si un nombre est pair, nous devons le diviser sur 2 et vérifier le reste de la division. Si le reste est 0 le nombre est pair sinon, il est impair.**

**Le code assembleur pour faire ca est le suivant:**

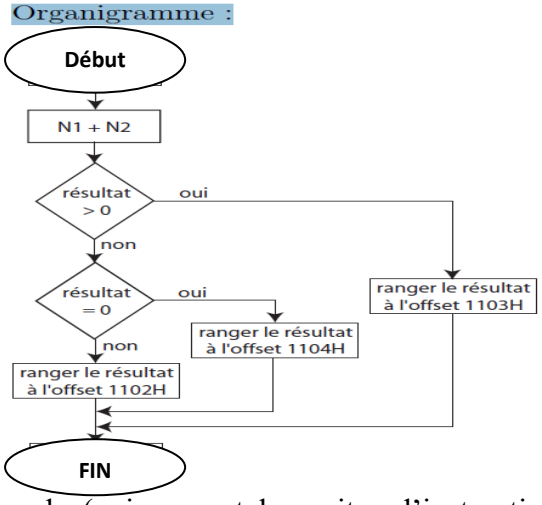
```
  MOV AX, #nombre    ; le contenu du rY + la valeur 5.
  MOV BX, 2          ; 2   présenter sur un octet le reste dans AH
  DIV  BX            ; diviser AX sur 2. Le reste est stocker dans AH.
  CMP AH, 0
 JE SAUT      ; effectuer un saut si  le reste de division égal à zero « est nb paire ».
...  ;reste du code.
```

- **DIV :** Si **l'opérande est un octet**, alors **on récupère le quotient** dans le registre **AL** et **le reste dans le registre AH**.
  Si **l'opérande est un mot**, alors **on récupère le quotient** dans le registre **AX** et **le reste dans le registre DX**.

**Exercice 2 :**

1. On veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul. Donnez l'organigramme qui présente le problème donné et le programme en langage assembleur associé

Organigramme :




Programme :

```
                mov     al,[1100H]
                add     al,[1101H]
                js      negatif
                jz      nul
                mov     [1102H],al
                jmp     fin
negatif :       mov     [1103H],al
                jmp     fin
nul :           mov     [1104H],al
fin :           hlt
```

2. Ecrire le code (uniquement les suites d'instructions) Assembleur pour faire la somme de dix (10) éléments d'un tableau. Le premier étant rangé à la case mémoire ayant l'adresse "0F68H" ? (chaque nombre est sur un octet et la somme peut dépassée 255)

```
mov bx, 0F68H    ;BX contient l'adresse du 1er élément
mov cx, 10       ;CX (compteur) contient le nombre d'élément du tableau
```

```
mov ax, 0        ;AX contient la somme et initialisé à 0
mov dx, 0        ;initialiser DX à zéro, le sous-registre DL va contenir
                 ;l'élément lu de la mémoire, DH restera toujours zéro et sera
                 ;utile pour l'addition des 2 registres AX et DX

repeat:
     mov dl, [bx]     ;pour lire 1 seul octet de la mémoire
     add ax, dx       ;DH étant à zéro faire la somme de 2 registre 16bits
     inc bx           ;incrémenter BX pour lire l'elt suivant du tableau
     loop repeat
```

**Exercice 3 : (Appel des procédures, adressage segmenté et gestion de la pile)**

**Dans le programme principal :**

3. Ecrivez un programme en utilisant les 02 instructions PUSH et POP pour la gestion d'une pile de 4 éléments.

**Dans le sous programme:**

4. Ecrivez le programme Assembleur 8086 contenant les deux procédures suivantes :

- ✓ L'affichage des éléments d'une pile
- ✓ L'addition d'un nombre quelconque pour chaque élément

**Solution :**

```
Data segment
        N1  dw  3      ; déclaration des 4 éléments
        N2 dw 4
        N3 dw 5
        N4 dw 6
ends
stack segment
        dw 128  dup (0)     ;  déclarartion de la pile ou chaque element est de 16 bits
ends
start:
 mov   ax, data
mov  ds, ax                  ; stocker les données dans le registre segment données ds pour les empliler
mov   ax , stack
mov    ss, ax                 ; stocker la pile dans le registre de segment ss
push    N1
push    N2                  ; empiler les éléments dans la pile
push    N3
push    N4
call    fct_affichage                ; Appel de la fonction d'affichage
call  fct_addition                   ; Appel de la fonction d'addition
fct _affichage   proc  near    ;  nom_fonction    type « procédure »    near «  appel dans le même programme »
mov   cx, 4      ; initialiser le compteur de la boucle avec 4 itérations ( 4 éléments)
boucle:
      mov    ah, 02h      ; Affichage (en code ASCII)
       POP  dx        ; stocker les données  empilées dans le registre général  dx
       Int    21h
       Loop  boucle
Ret    ; retour au prgm principal
Endp
```

```
call fct_addition    proc near    ; nom_fonction   type « procédure »   near « appel dans le même programme »
mov   cx, 4      ; initialiser le compteur de la boucle avec 4 itérations ( 4 éléments)
boucle:
     ADD    dx, 10      ; Ajouter 10 pour chaque élément
     mov    ah, 02h      ; Affichage (en code ASCII)
     POP  dx         ; stocker les données  empilées dans le registre général  dx
      Int    21h
      Loop  boucle
Ret    ; retour au prgm principal
Endp

Mov   ax, 4cOOh              ; Vider le registre AX à la fin du programme
Int 21h                      ; fin de programme
ends
```

## Exercice 4: (Gestion des entrées/sorties)

Expliquez les interruptions 21h les plus utilisés suivantes en donnant des exemples avec les instructions de l'assembleur 8086 :

- La fonction 09
- La fonction 02
- La fonction 01
- La fonction 0AH

**Solution**

- La fonction 09 : Affiche le contenu de registre Dx et arrete l'affichage jusqu'à quand le caractère $, sera trouvé (signe arrêt d'affichage)
- Exemple : CH DB ' HELLO', $ ; CH « label »

```
Mov  dx, offset   CH
MOV   AH, 09   ; activer l'interruption
Int  21h
```

- La fonction 02 : Affiche le contenu de registre Dl « un seul caracter ou nb »

Exemple :
```
mov  DL, 'j'
Mov  AH, 02          ; afficher:   j    affiche code ASCII
Int  21h
```

- La fonction 01 : Affiche le premier caractère taper sur clavier, qu'est stocker dans le registre AL
  Exemple : mov AH, 01 ; AL= 3

  Int 21h

- La fonction 0AH : Affiche les caractères taper sur clavier nbrs fixés par l'utilisateur. La saisie s'arrête quand on tape la touche de retour, le caractère CR (13) est mémorisé avec la chaîne
  Exemple : ORG 100h

  *Data :*
  *Nb_max DB 6 DUP(FF)*

  Mov AH, 0AH

  Mov DX, offset *Nb_max*

  Int 21h

Explications:

| 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | |
|---|---|---|---|---|---|---|
| FF | FF | FF | FF | FF | FF | :Avant |
| 6 | A | B | C | CR | FF | :Après |

# Série de TD N°6

**Exercice 1 :** Expliquer les instructions suivantes et indiquez le mode d'adressage :

MOV EAL, [E58]
ADD EAL, F632
MOV [EBX], [ebx + ecx * 4 + 8]
SUB RAL, [RBX+ESI]
MOVZX EAX, Word [edi]
MOV EAX, RBX
MOV [Rdi + Rsi], Edx
MOV EAX, [EBX]
MOV RAX, [RBX+4]

**Exercice 2 :** 1- Ecrire un programme en assembleur 8086 qui permet de :

- Trouvez le maximum entre deux nombre entiers
- Ajoutez le code qui fait la recherche du maximum pour 04 nombres entiers
- Effectuer un JUMP dans votre code vers une instruction qui stocke le résultat dans un nouveau registre
- Réalisez le programme avec l'utilisation des registres de l'architecture 32 et 64 bits

# Correction Série 6

**Exercice 1 :** Expliquer les instructions suivantes et indiquez le mode d'adressage :

MOV EAL, [E5**8**] **Mode direct : copier le contenu de l'adresse mémoire E58 dans le registre EAL, en utilisant l'architecture 32 bits**
ADD EAL, F632 **Mode immédiat : Copier la valeur en hexadécimal F632 dans le registre EAL, en utilisant l'architecture 32 bits**
MOV [EBX], [**ebx** + **ecx * 4** + **8]** **Incorrect : Impossible de copier une adresse vers une autre adresse**
SUB RAL, [RBX+ESI] **Mode indirect basé indexé : soustraction de contenu de l'adresse mémoire des registres RBX+ ESI de la valeur contenue dans le registre RAL en utilisant l'architecture 64 bits**

MOVZX EAX, Word [edi] **Mode direct : copier un mot mémoire sur 16 bits et** modifier **eax** en mettant dans **al** le mot mémoire pointé par **edi** et en mettant à 0 les 16 autres bits. **, en utilisant l'architecture 32 bits**

MOV EAX, RBX **Incorrect : Impossible de copier le contenu de registre 64 bits dans un registre 32 bits**
MOV [**Rdi** + **Rsi**], **Edx** **Mode indirect indexé : stocker dans l'adresse Rdi + Rsi la valeur contenue dans le registre edx en utilisant l'architecture 64 bits**
MOV EAX, [EBX] **Mode indirect basé : stocker dans le registre Eax le contenu de registre EBX en utilisant l'architecture 32 bits**
MOV RAX, [RBX+4] **Mode indirect basé : stocker dans le registre RAX le contenu de l'adresse mémoire RBX +4 en utilisant l'architecture 64 bits**

**Exercice 2 :** 1- Ecrire un programme en assembleur 8086 qui permet de :

- Trouvez le maximum entre deux nombres entiers
- Ajoutez le code qui fait la recherche du maximum pour 4 nombres entiers
- Effectuer un JUMP dans votre code vers une instruction qui stocke le résultat dans un nouveau registre
- Réalisez le programme avec l'utilisation des registres de l'architecture 32 et 64 bits

1. Le maximum entre deux nombres entiers (en utilisant l'architecture 32 bits) :

```
MOV EAX, #nombre1
MOV EBX, #nombre2
Cmp EAX, EBX
JG  stocker
Mov  EDX, EBX

Stocker:
Mov  EDX, EAX

JMP Fin

Fin int 21H
```

**2.** Le maximum entre quatre nombres entiers (en utilisant l'architecture 64 bits) :

**Solution 1**

```
MOV RAX, #nombre1
MOV RBX, #nombre2
MOV RDX, #nombre3
MOV RCX, #nombre4

CMP RAX, RBX
JG   Max1
Mov  R1, RBX

Max 1:
Mov  R1, RAX

CMP RDX, RCX
 JG   Max 2

  Mov  R2, RCX


Max 2 :

Mov  R2, RDX

     Cmp R1, R2

     JG Max

Mov  R3, R2

Max :

Mov R3, R1

JMP FIN
```

**Solution 2 « architecture 16 bits » (même principe)**

```
.MODEL SMALL
.STACK 100H
.DATA
   nombres DW 12, 45, 7, 23 ; Les 4 nombres à comparer (peuvent être
modifiés)
   count DB 3 ; Nombre de comparaisons à faire (4 nombres = 3
comparaisons)
   max_msg DB 'Le maximum est: $'

.CODE
Fonction Max  PROC near
   MOV AX, @DATA
   MOV DS, AX

   ; Initialisation avec le premier nombre
   MOV AX, nombres[0]
      ; Initialisation pour la boucle

   MOV SI, 2 ; Décalage pour accéder aux nombres suivants
   MOV CX, count ; Compteur de boucle

comparaison
   ; Charger le nombre suivant dans BX
   MOV BX, nombres[SI]

   ; Comparaison
   CMP AX, BX
   JGE pas_plus_grand
   MOV AX, BX ; AX = nouveau maximum

pas_plus_grand:
   ; Préparation pour la prochaine itération
   ADD SI, 2 ; Passer au nombre suivant (chaque nombre = 2 octets)
   DEC CX
    JNZ
Loop comparaison ; Continuer tant qu'il reste des nombres

   ; Affichage du résultat
   MOV AH, 02H
   LEA DX, max_msg
   INT 21H

   ; Conversion et affichage du nombre (supposé < 100)
   MOV BX, AX ; Sauvegarde du maximum

   ; Affichage des dizaines
   MOV AL, BL
   MOV AH, 0
    MOV DL, 10
   DIV DL ; AL = dizaines, AH = unités

   MOV DL, AL
    ADD DL, '0'
   MOV AH, 02H
   INT 21H

   ; Affichage des unités
   MOV DL, AH
   ADD DL, '0'
   MOV AH, 02H
   INT 21H

   ; Fin du programme
   MOV AH, 4CH
   INT 21H
ENDP
ENDS
```

# Séries TP

# TP 1 : Manipulation du BIOS/UEFI

## 1) Le BIOS

Le BIOS (Basic Input/Output System) est le premier programme basique en langage assembleur qui s'exécute servant d'interface entre le système d'exploitation et la carte mère.

La majorité des BIOS ont un « setup » (programme de configuration) qui permet de modifier la configuration basique du système. Lorsque tous les tests ont été effectués, le BIOS affiche un message invitant l'utilisateur à appuyer sur une ou plusieurs touches afin d'entrer dans le setup du BIOS (F2, F10, DEL, shift, …). Il fournit de nombreuses fonctions d'aide qui permettent de lire les secteurs de démarrage du stockage connecté et d'imprimer des éléments à l'écran. Le BIOS fonctionne en mode 16 bits


**Figure 1. Capture d'écran pour une fenêtre de BIOS.**

## 2) L'UEFI

Le standard UEFI (Unified Extensible Firmware Interface) signifiant « Interface micrologicielle extensible unifiée » définit une interface entre le micrologiciel (firmware) et le système d'exploitation d'un ordinateur. Cette interface succède sur certaines cartes mères au BIOS. L'UEFI offre de nombreux avantages sur le BIOS : fonctionnalités réseau en standard, interface graphique de bonne résolution, gestion intégrée d'installations multiples de systèmes d'exploitation et affranchissement de la limite des disques.

Le BIOS, écrit en assembleur, limitait les modifications, l'UEFI est écrit en C, ce qui rend sa maintenance plus souple et reste acceptable en raison des coûts décroissants de la mémoire. Développé pour assurer l'indépendance entre système d'exploitation et plate-forme matérielle sur laquelle il fonctionne, l'UEFI est disponible sur les plates-formes x86 (32 et 64 bits). Microsoft Windows, depuis la version 8, supporte de façon optionnelle le secure boot grâce à

une signature numérique transmise aux constructeurs de cartes mères. Depuis Windows 11, Secure Boot est obligatoire. L'UEFI est donc capable de fournir une interface graphique (navigation avec la souris) contrairement au BIOS qui permet la navigation uniquement à l'aide du clavier. Si un problème est rencontrer lors d'accéder au BIOS « BOOSER », il faut entrer dans le setup du l'UEFI il faut suivre les étapes suivantes : (Menu Démarrer, Marche/Arrêt, avant de cliquer sur Démarrer cliquer en foncé sur la touche Maj, puis cliquer sur redémarrer, l'écran de téléchargement apparait relâché Maj, dans le petit menu apparait aller au Dépannage, Options avancées, un autre petit menu apparait il faut aller dans l'option « changer les paramètres du microprogramme UEFI », à la fin il suffit cliquer sur le bouton Redémarrer qui apparait) et en serra dans l'UEFI (Notre interface graphique pour la configuration des paramètres de notre ordinateur en utilisant seulement la souris sans l'utilisation de clavier).

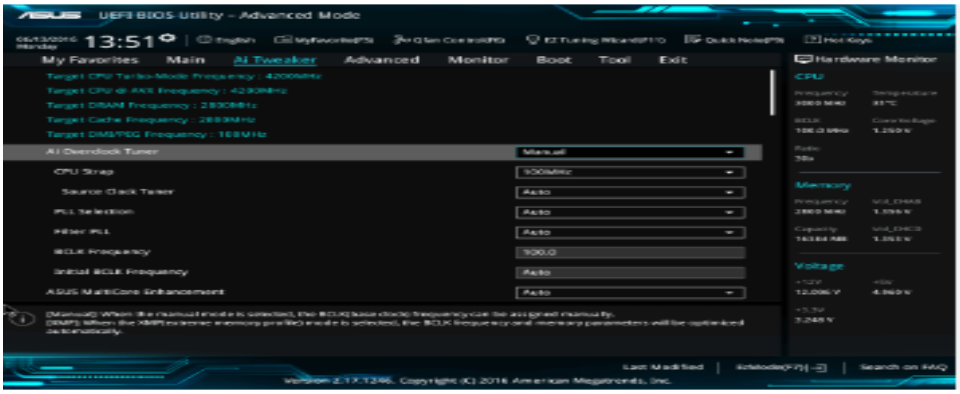



**Figure 2. Capture d'écran pour une fenêtre de l'UEFI.**

Manipulation du BIOS/UEFI :

**Avec BIOS :** Découvrir et configurer les paramètres fondamentaux :

- ✓ L'ordre de démarrage (Boot Options, Boot Order : modifier l'ordre en utilisant le menu indicateur à droite), pour réinitialiser cliquer sur Exit, Load setup Default (l'ordre par défaut) ou Exit sans enregistrer les modifications (Without Save changes).
- ✓ La gestion des périphériques (menu Storage, Device configuration) Et les réglages de sécurité (menu Security, Device security).
  - ➢ ***La touche ESC « échappe » pour sortir des menus sans la sauvegarde.***

**Avec UEFI :** Découvrir et configurer les paramètres fondamentaux :

Pour améliorer les performances de l'ordinateur et accélérer le temps de redémarrage

- ✓ Activer le profil « XMP pour Intel et DECP pour DELL » : (Profil de la RAM) dans la page d'accueil.
- ✓ Dans le menu BOOT (Launch CSM ) dans le Window 10 il faut le désactiver (Launch CSM ).

# TP 2 : Programmation assembleur avec EMU8086

**BUT DU TP :** Se familiariser avec l'EMU8086, maitriser les instructions de transfert des données et les calculs d'adresses.

## 1. Définition de l'assembleur

L'assembleur est un langage de programmation transformant un fichier texte contenant des instructions, en un programme que le processeur peut comprendre (programme en langage machine).

Ce langage machine à la particularité d'être difficile à programmer car il n'est composé que des nombres en hexadécimal.

## 2. Présentation d'EMU 8086

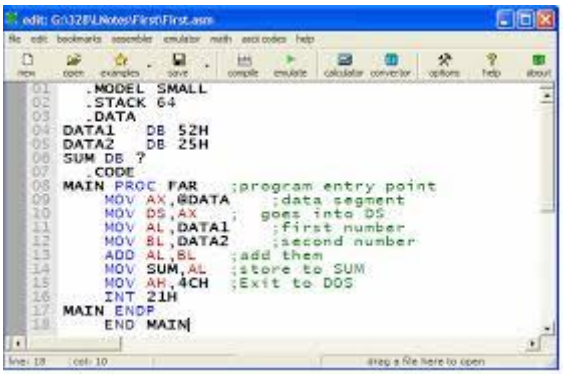
**EMU8086** est un émulateur du microprocesseur **8086** (compatible Intel et AMD). Dans un premier temps, nous allons utiliser **EMU8086** pour consulter/modifier les registres, pour afficher/modifier le contenu de la mémoire et enfin pour éditer /exécuter des programmes élémentaires.

Les instructions peuvent être exécutées en arrière et en avant. L'interface visuelle est très facile à utiliser. Vous pouvez regarder les registres, les drapeaux et la mémoire pendant l'exécution de votre programme.

Au lancement de l'émulateur, une fenêtre contenant la boite de dialogue suivante s'affiche (figure 1), cette dernière permet de choisir entre plusieurs options et de créer un nouveau fichier.


**Figure 1. Boite de dialogue.**

En cliquant sur le bouton New, une boite de dialogue s'ouvre, cette fenêtre permet de choisir un type de fichier à créer. Dans ce TP en utilisant des fichiers binaire (d'extension.bin) on sélectionne le format **BIN** et on clique sur **OK**. La fenêtre qui s'affiche présente l'éditeur de programme de l'Emu 8086.



```
.MODEL SMALL
.STACK 64
.DATA
DATA1   DB 52H
DATA2   DB 25H
SUM DB ?
.CODE
MAIN PROC FAR   ;program entry point
    MOV AX,@DATA    ;data segment
    MOV DS,AX   ;   goes into DS
    MOV AL,DATA1    ;first number
    MOV BL,DATA2    ;second number
    ADD AL,BL   ;add them
    MOV SUM,AL  ;store to SUM
    MOV AH,4CH  ;Exit to DOS
    INT 21H
MAIN ENDP
    END MAIN
```

**Figure 2. Editeur de programme de l'EMU8086.**

Une fois la saisie du programme est terminée on peut soit le compiler ou l'émule, on cliquant sur le bouton **Emulate,** et après les corrections des erreurs syntaxiques les deux fenêtres suivantes s'ouvrent :

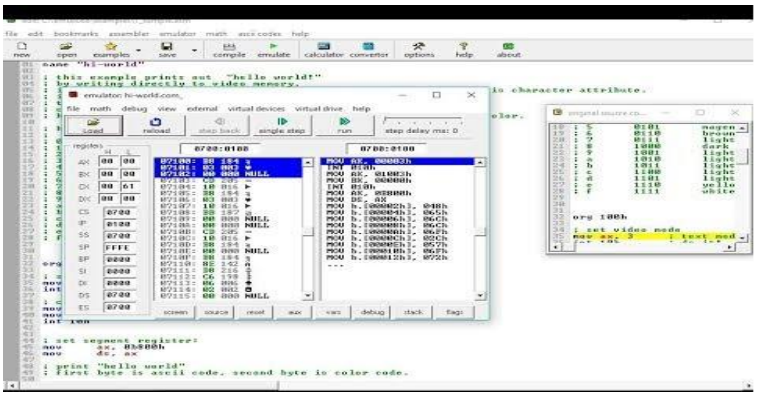


**Figure 3. Les 2 fenêtres de l'EMU8086**.

La petite fenêtre contient le code source, et la grande représente l'émulateur. Ce dernier permet d'exécuter le programme tout entier instruction par instruction. Il permet aussi de visualiser les contenus de tous les registres de microprocesseur ainsi que les différents segments de la mémoire et différents composants de système.

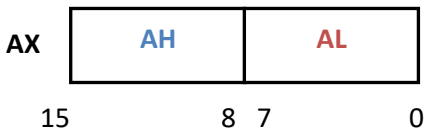
Les instructions peuvent être exécutées en arrière et en avant. L'interface visuelle est très facile à utiliser. Vous pouvez regarder les registres, les drapeaux et la mémoire pendant l'exécution de votre programme.

**a. Registres sur 8086 :**

- 8 registres généraux (16 bits) : **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **DP** et **SP**
- 4 registres de segments (16 bits) : **CS**, **DS**, **ES**, **SS**
- 2 registres spéciaux (16 bits) : **IP** et **FLAGS**

**b. Décomposition des registres :**

- Les registres A, B, C et D se décomposent en deux sous registres de 8 bits chacun :
- H (partie haute) et L (partie basse)
- Par exemple, **AX** se compose en **AH** et **AL**.




**c. Instruction de base :**

**MOV :** transfert d'information

L'instruction la plus utilisée est l'instruction MOV, qui copie la valeur d'un opérande ***source*** dans un opérande ***destination***

- **Format :** **MOV** destination, source

  Cinq formes sont possibles

  **MOV** reg, reg — reg : registre
  **MOV** reg, mem — mem : adresse mémoire
  **MOV** mem, reg
  **MOV** reg, immed — immed : valeur immediate(binaire, décimale, hexadecimal)
  **MOV** mem, imme

Notez qu'il n'existe pas de transfert d'une adresse mémoire à une adresse mémoire.

**Exemple :**

**MOV** AX, 1h ➡ AX= 1h
**MOV** AX, BX ➡ AX= BX

**MOV** AL, [BX] ➡ AL = Le contenu de l'adresse mémoire pointée par DS : BX

3. **Structure d'un programme EMU8086 :** Il existe deux structures**,** sa dépend de la version d'EMU 8086 utilisé :

**//Structure 1//**

```
TITLE nom  ; donner un nom au programme
   PILE  SEGMENT STACK  ; déclaration d'un segment de pile
                        ; dont le nom  est pile
    .....
    .....
   PILE  ENDS           ; fin de la déclaration de la pile

   DONNEE  SEGMENT ; déclaration d'un segment de données qui va
                   ; contenir les variables
    .....
   DONNEE ENDS      ; fin de la déclaration de données

   LECODE  SEGMENT ; déclaration du segment de code  qui va contenir le
   code
        Debut:          ; étiquette d'indication du point d'entrée du code
                MOV AX,DATA
                MOV DS, AX
    .....
    .....
                MOV   AH, 4Ch
                INT   21h
   LECODE ENDS      ; fin de la déclaration du code
        END  Debut  ; fin du point d'entrée du code
```

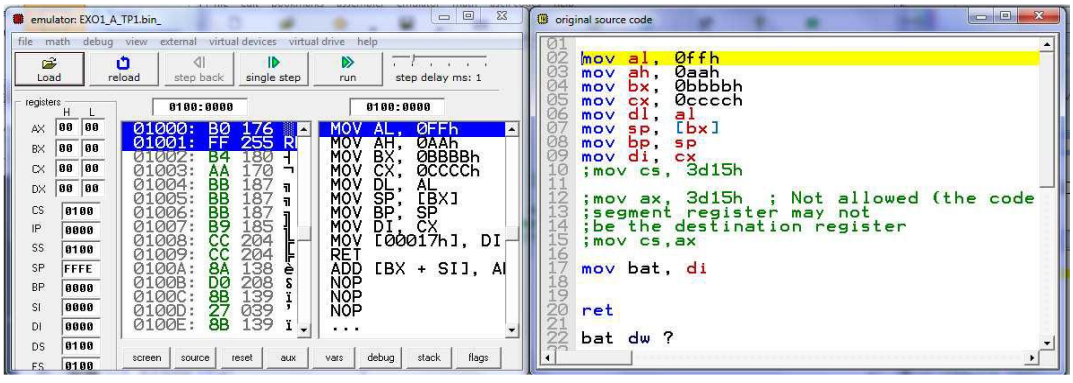
**//Structure 2//**

```
ORG 100h            ; Début de pgm

 include emu 8086 inc     ; Appel de biblio 8086
 .data          ; Partie de déclaration
 .code          ; début du pgm ou le code

 INT 21h        ; Appel de l'interruption 21h
 END
 RET
```

**Exercice 1 :**

**1-1)** Lancez emu8086, appuyez sur 'new' et 'cancel', puis 'save' or sauvegardez cette session sous: EXO1_A_TP1.asm dans le répertoire de travail.
Copiez les lignes suivantes dans de l'éditeur, corrigez les par l'exécution de ce fragment (signle step==> pas à pas):

```
mov al, ffh
mov ah, aah
mov bx, bbbbh
mov cx, cccch
mov dl, al
mov sp, [al+bx]
mov bp, sp
mov di , cx
mov cs, 3d15h
mov bat, di
```




- Vérifiez/confirmez le contenu des différents registres du mp8086.
- Est ce que la variable '**bat**' contient la valeur '**cccch**'? Montrez-le.

**1-2) -** Identifiez les fenêtres suivantes: code source original, code source en exécution, émulateur en exécution.
- Identifiez les zones suivantes: zone registres, zone mémoire et zone code désassemblé.

**Exercice 2 :**

1. Ecrire un programme Assembleur 8086 qui affiche le texte suivant « *Voici mon premier programme Assembleur du microprocesseur 8086* »
2. Soit le programme suivant, remplir les vides

```
ORG100h
MOV AX, 243 ;…….
ADD AX, 243h ; .....
;  ……..
MOV AX, 0xA243; .....
MOV AX, 0A243h ; .....
MOV AX, 1010001001000011b; .....
MOV AL, 'a' ; .....ASCII du caractère 'a'
MOV AX, 'a' ; ..... ASCII du caractère 'a'
MOV AX,'ab' ; .....
```

Emulator du mp 8086 :dispositif qui émet le mp8086 => comprendre l'architecture 8086
Emulator <> simalator
Emulator = materiel (carte = kit) + logiciel
Simulator = Logiciel

# TP 3 : Programmation assembleur avec EMU8086

**BUT DU TP :** Se familiariser avec les registres du 8086 et certaines instructions de base, maitriser la notion de la segmentation de la mémoire et le mode d'adressage.

## 1. Instructions de transfert des données et modes d'adressage

C'est principalement l'instruction **MOV**, elle permet d'écrire une valeur directement dans l'un des registres du processeur ou dans un emplacement en mémoire. Elle permet aussi le transfert des données entre les registres ou entre registre et mémoire, le transfert entre mémoire - mémoire n'est pas possible. Le syntaxe de l'instruction MOV est comme suit : **MOV** ***destination, source***

L'instruction MOV permet le transfert du contenu de la **source** vers la **destination.** Plusieurs modes d'adressage sont possibles :

- ✓ **1. Mode immédiat :** Utilisé pour charger (écrire) une valeur dans un registre. La donnée est spécifiée immédiatement après l'instruction.

  **Exemple:** MOV SI, 1240h
  MOV AH, 10h

- ✓ **2. Mode registre :** Utile lors de transfert du contenu d'un registre vers un autre.

  **Exemple:** MOV AX, BX

- ✓ **3. Mode direct :** Permet d'écrire une valeur directement à une adresse effective donnée. Ou de transférer le contenu d'un registre vers un emplacement mémoire ou vise versa. Un opérande est donnée en mode direct s'il est désigné par son adresse effective dans la mémoire.

  **Exemple :** MOV 100h, 24h
  MOV AH, 150b

- ✓ **4. Mode indirect basé:** Permet les mêmes opérations de transfert que le mode direct, mais dans ce cas l'adresse effective est le contenu d'un registre de base (BX ou BP) avec éventuellement d'un déplacement contant. **Exemple :** MOV [BX], 64h
  MOV AL, [BX]

- ✓ **5. Mode indirect indexé :** L'adresse effective est fournie par l'un des registres d'index **SI** ou **DI,** avec éventuellement d'un déplacement constant.

  **Exemple :** MOV [SI], 33h
  MOV [DI+82h], 5h

- ✓ **6. Mode indirect basé-indexé :** Les modes basé et indexé sont combinés. L'adresse logique est donnée par **BX/BP**, et **SI/DI.**

  **Exemple :** MOV AL, 2[BX, SI]
  MOV [BP+DI], AH

✓ **Opérations arithmétiques :**

➢ **ADD :** Addition

Réalise l'opération d'addition standard, possède les mêmes formes possibles que MOV

- **Format : ADD** destination, source
- **Signification :** destination= destination+ source
- **Exemple :**
  **MOV** AL, 12h ➡ AL=12h
  **MOV** BL, 35h ➡ BL=35h
  **SUB** AL, BL ➡ AL= AL – BL = 12h -35h =DDH
  **SUB** BL, 56h ➡ BL= BL -56h =35h-56h=DFH
  **SUB** CX, [SI] ➡ CX=CX-[DS :SI]

➢ **MUL :** Multiplication non signée

L'instruction **MUL** réalise la multiplication non signée de deux opérandes dont l'un est par défaut **AL** (ou **AX**) l'autre est spécifié en instruction. **MUL** possède mêmes formes qu'**INC** « Incrémentation ». Selon la taille de l'opérande, deux cas sont possibles : multiplication sur l'octet ou sur l'octet ou sur 2 octets.

✓ Sur 1 octet : MUL opérande ➡ AX=AL opérande
✓ Sur 2 octets : MUL opérande ➡ DX, AX=AX× opérande

- **Format : MUL** opérande

**Exemple :**

**MOV** AL, 12h ➡ AL=12h
**MOV** BL, 35h ➡ BL = 35h
**MUL** BL ➡ AX= AL×BL = 12h ×35h =3BAh
**MOV** CX, 1256h ➡ CX=1256h
**MUL** CX ➡ AX= 547Ch, DX= 0044h

➢ **DIV:** Division non signée

L'instruction **DIV** réalise la division non signée de deux opérandes dont l'un est par défaut AX (ou DX, AX) l'autre est spécifié en instruction , DIV possède mêmes formes qu'INC. Selon la taille de l'opérande, deux cas sont possibles : division sur l'octet ou sur 2 octets .

✓ Sur 1 octet : DIV opérande AX/ opérande ➡ AL= quotient, AH=Reste
✓ Sur 2octets : DIV opérande DX, AX/ opérande ➡ AX=quotient, DX=Reste

- **Format : DIV** opérande
  Si une division par zéro est tentée, une interruption est produite. Le quotient et le reste sont dans ce cas, non définis.

**Exemple :**

**MOV** AX, 1345h ➡ AX=1345h
**MOV** BL, 85h ➡ BL=85h
**DIV BL** ➡ AX/BL= 1345h/ 85h AL=25h, AH=0Ch

**MOV** DX, 02B6h ➡ DX=02B6h
**MOV** CX, 9256h ➡ CX=9256h
**DIV** CX ➡ DX, AX / CX=02B6h0C25h / 9256h
AX= 04BEh, DX=1851h

**EXERCICE 1 :**

1. Saisir le programme ci-dessus dans l'éditeur de l'émulateur 8086

**MOV AX, 600H**
**MOV DS, AX**
**MOV AX, 700H**
**MOV SS, AX**
**MOV AX, 800H**
**MOV ES, AX**
**MOV ES, AX**
**MOV AX, 1234H**
**MOV BX, AX**
**MOV CX, BX**
**MOV DL, CH**
**MOV [100H], DL**
**MOV [102H], AX**
**MOV CH, [102H]**

2. Indiquez le mode d'adressage pour chaque instruction du programme précédent et les instructions suivantes :
   MOV AX, 1h
   mov al, 1h
   mov ax, 1000h
   mov AH, 10h
   MOV bx, 1234h
3. Commenter chaque instruction, donnez le résultat après l'exécution de chaque instruction.

**EXERCICE 2:**

1. Utiliser l'**emu8086** pour écrire un programme permettant d'additionner deux nombre de votre choix, et affiche le résultat. Modifier le programme pour faire la multiplication.
2. Utiliser l'**emu8086** pour écrire un programme de soustraction de deux registres, stocker le résultat sur le data segment.
3. Ecrire un programme de multiplication de deux registres, stocker le résultat sur le registre DX
4. Ecrire un programme de division de deux registres
5. Convertir les équations mathématiques suivantes en langage assembleur :
   Z= X-Y+5 avec X= 4h, Y= 1h
   Z= X+ Y+3 avec X= 2h, Y= 1h
6. Ecrire un programme pour additionner 5 à BX 5 fois avec BX= 80h.

# TP 4 : Programmation assembleur avec EMU8086

**BUT DU TP :** Travailler sur les instructions arithmétiques et logiques, Boucles et conditions, Gestion de la pile, Appliquer des sous-programmes simples, Gestion des entrées/sorties avec Emu8086

## 1. Programmation en Assembleur 8086

### 1.1. Syntaxe d'une instruction en assembleur

| Label (facultatif) | Opération (obligatoire) | opérande (selon opération) | Commentaire (facultatif) |
|---|---|---|---|

**Figure 1 : Syntaxe d'une instruction en assembleur.**

- **Label :** étiquette attribuée à une ligne du programme utile pour les branchements. Il comporte au maximum 31 caractères et se termine par (:) (sauf pour les directives)
- **Opération :** un mnémonique qui définit l'opération à exécuter et qui doit être rigoureusement respecté. ex JUMP, ADD
- **Opérande :** champ qui présente les paramètres éventuels de l'opération (0 ou plusieurs données).
- **Commentaire :** champ commençant par ( ;) ou (#) sans valeur pour le programme assembleur

### 1.2. Structure d'un programme en Assembleur :

Un programme en langage assembleur contient deux types de commandes :
• **Les instructions proprement dites**, qui appartiennent du jeu d'instruction du microprocesseur et seront traduites par le programme assembleur en instructions en langage machine. (Destinées au microprocesseurs)
• **Les pseudo-instructions ou directives**, mots clefs spéciaux qui consistent en des ordres destinés au programme assembleur sur la façon dont il doit traduire le programme. (ne génèrent généralement pas d'instructions en langage machine).

#### a. Principales Directives pour l'assembleur 8086 -pseudo instructions-

- **Déclaration des segments :**
  seg_pile **SEGMENT stack** ; mot clef stack (pile)
  DW 100 dup (?) ; reserve espace
  seg_pile **ENDS**
- **Définitions de constantes :**
  nom constante **EQU** valeur
  Exemple : escape equ 1BH

- **Déclaration des variables (Réservation de cases mémoires )**
  **DB :** Define Byte, (1 octets)
  **DW :** Define Word, (2 octets)
  **DD** : Define Double, (4 octets)
  **DQ** : Define Quadruple, (8 octets)

**Exemple**

```
data  SEGMENT
truc  DW  0F0AH      ; en hexa
masque  DB  01110000b     ; en binaire
data ENDS
```

- **Déclaration des tableaux**

```
data SEGMENT
machine   db 10, 0FH       ;  tableau de 10 éléments 2 fois 1 octet
chose     db 2, 'ALORS'     ; tableau de 2 éléments valant « Alors »
data ENDS
```

- **Déclaration d'une procédure**

```
Calcul  PROC near      ; procedure nommée Calcul
... ; instructions
Calcul ENDP          ; fin de la procedure
CALL Calcul          ; Appel de la procédure
```

### b. Jeu d'instructions du microprocesseur 8086

- Instructions de transfert de données (affectation)
- Instructions Arithmétiques et logiques
- Instructions de branchement (saut de programme)

- **Type d'opérandes**

| opérande 1 : destination | opérande 2 : source |
|---|---|
| registre | registre |
| registre | immédiat |
| registre | mémoire |
| mémoire | registre |
| mémoire | immédiat |

**Figure 2 : Type d'opérandes avec 8086.**

| Symbole | Code Op. | Octets | |
|---|---|---|---|
| MOV AX, *valeur* | B8 | 3 | AX ← *valeur* |
| MOV AX, [ *adr* ] | A1 | 3 | AX ← contenu de l'adresse *adr*. |
| MOV [ *adr* ], AX | A3 | 3 | range AX à l'adresse *adr*. |
| ADD AX, *valeur* | 05 | 3 | AX ← AX + *valeur* |
| ADD AX, [ *adr* ] | 03 06 | 4 | AX ← AX + contenu de *adr*. |
| SUB AX, *valeur* | 2D | 3 | AX ← AX - *valeur* |
| SUB AX, [ *adr* ] | 2B 06 | 4 | AX ← AX - contenu de *adr*. |
| SHR AX, 1 | D1 E8 | 2 | décale AX à droite. |
| SHL AX, 1 | D1 E0 | 2 | décale AX à gauche. |
| INC AX | 40 | 1 | AX ← AX + 1 |
| DEC AX | 48 | 1 | AX ← AX - 1 |
| CMP AX, *valeur* | 3D | 3 | compare AX et *valeur*. |
| CMP AX, [ *adr* ] | 3B 06 | 4 | compare AX et contenu de *adr*. |
| JMP *adr* | EB | 2 | saut inconditionnel (adr. relatif). |
| JE *adr* | 74 | 2 | saut si = |
| JNE *adr* | 75 | 2 | saut si $\neq$ |
| JG *adr* | 7F | 2 | saut si $>$ |
| JLE *adr* | 7E | 2 | saut si $\leq$ |
| JA *adr* | | | saut si CF = 0 |
| JB *adr* | | | saut si CF = 1 |
| *Fin du programme (retour au DOS) :* | | | |
| MOV AH, 4C | B4 4C | 2 | |
| INT 21 | CD 21 | 2 | |

**Figure 3 : Exemples des instructions différentes avec 8086.**

**Exercice 1 :**

1. Ecrivez le programme Assembleur 8086 calculant la somme de 6 et 3 et l'affiche.
2. Ecrivez le programme Assembleur 8086 qui calcule la longueur d'une chaine de caractère.
3. Ecrire un programme Assembleur 8086 qui cherche un élément dans un tableau et affiche l'élément s'il est trouvé.
4. Ecrire un programme Assembleur 8086 qui lit 5 entiers et les insère dans un tableau de dimension 5. Puis il retournera la somme de ces entiers.

**Exercice 2 :**

1 Ecrire le programme Assembleur 8086 contenant les deux procédures suivantes :

- ✓ L'affichage des éléments d'une pile
- ✓ L'addition d'un nombre quelconque pour chaque élément

**Programme principal :**

Ecrivez un programme en utilisant les 02 instructions PUSH et POP pour la gestion d'une pile de 4 éléments.

# Solutions des exercices :

**Exercice 1 :**

1. Ecrivez le programme calculant la somme de 6 et 3 et l'affiche

```
; Définition du segment de donnée ------------------------------------------------------------
data SEGMENT
A DB 6 ; A = 6
B DB 3 ; B = 3
Somme DB ?    ; Résultat
data ENDS
; Définition du segment de code et début du code ----------------------------------------
code SEGMENT
Addition PROC
ASSUME DS:data ; Attribution du registre de segment de donnée
ASSUME CS:code ; Attribution du registre de segment de code
debut :
; Pointage des données-------------------------------------------------------------
MOV AX, data ; première instruction
MOV DS, AX ; initialisation de DS
; Le programme d'addition ----------------------------------------------------
MOV AL, A
ADD AL, B
MOV result, AL ; rangement résultat
; Conversion ASCII puis Affichage-----------------------------------------
Mov DL, Result
Add DL, 48 ;48 le code ASCII de 0
MOV AH, 2 ; fonction du DOS qui affiche
INT 21H ; à l'écran le caractère se trouvant dans DL
; Retour au DOS-----------------------------------------------------------------------
MOV AH, 4CH ; fin de traitement et retour au DOS
INT 21H
Addition ENDP ; fin de procédure
code ENDS ; fin de segment
END debut ; fin des instructions.
```

2. Programme qui calcule la longueur d'une chaine de caractère

```
;-------------------------------------------------------------------------------------------------
data SEGMENT ; Segment de données
chaine DB 'Hello world\0'
data ENDS
code SEGMENT ; Segment de code
ASSUME DS:data, CS:code
main:
MOV AX,data ; Initialisation du registre de
MOV DS,AX ; segment de données
MOV BX,offset chaine ; Chargement de l'adresse de la
```

; chaîne dans BX Il s'agit de
; l'adresse relative au début du
; segment
MOV CX,0 ; Initialisation compteur
loop: MOV AL,[BX] ; transfert des caractères dans AL
; Adressage indirect
CMP AL,0 ; Test de fin de chaîne
JZ fini
INC BX ; Incrémentation pointeur
INC CX ; Incrémentation compteur
JMP loop
fini:
MOV AH,4CH ; retour au Dos
INT 21H
code ENDS
END main

3.Ecrire un programme qui cherche un élément dans un tableau et affiche l'élément s'il est trouvé

```
1.  Data SEGMENT
2.    C         db "H"
3.              Tab      db "B","C","D","G","H","J","L","$"
4.  Data ENDS
5.  Code SEGMENT
6.        assume CS:Code, DS:Data
7.  main :      Mov AX, Data
8.              Mov DS,AX
9.              Mov SI, 0
10.  boucle: mov AH, Tab[SI]
11.             cmp AH,"$"
12.             je fini
13.  aff:       cmp AH,C
14.             jne suite
15.             mov DL,AH
16.             mov AH,2
17.             int 21h
18.             jmp fini
19.  suite:     INC SI
20.             jmp boucle
21.  pas :      mov DL,AH
22.             mov AH,2
23.             int 21H
24.  fini:      Mov AH, 4Ch
25.             Int 21h
26.  Code ENDS
27.  END main
```

4. Ecrire un programme Assembleur 8086 qui lit 5 entiers et les insère dans un tableau de dimension 5. Puis il retournera la somme de ces entiers.

. data

Tab1 db 5 dup(0) ; déclarer un tableau de 5 dimension dont toutes les cases sont initialisé à 0

. Stack

; contenir un bal de pile

db 128 dup (0)

.code

```
Mov   ax, @data
Mov   ds, ax
Mov   ax, 00h
Mov   si, 00h
Mov  cx, 05h
i:
mov  ah, 09h      ;lecture
int  21h
mov   tab1[si], al
inc  si
loop i
mov   ah, 02h
mov   dl, 0Ah
int  21h
mov  si, 00h
mov  cx, 00h
o:
mov    dl, tab1[si]
mov ah, 02h   ; lecture
int  21h
inc   si
loop  0
mov  ax, 4c0h
int  21h
09h    ; afficher
```

## Exercice 2 :

1 Ecrire le programme Assembleur 8086 contenant les deux procédures suivantes :

- ✓ L'affichage des éléments d'une pile
- ✓ L'addition d'un nombre quelconque pour chaque élément

## Programme principal :

Ecrivez un programme en utilisant les 02 instructions PUSH et POP pour la gestion d'une pile de 4 éléments.

### Solution :

```
Data segment

        N1  dw  3       ; déclaration des 4 éléments
        N2 dw 4
        N3 dw 5
        N4 dw 6

ends

stack segment

        dw 128  dup (0)      ;  déclararion de la pile ou chaque element est de 16 bits

ends

start:

 mov   ax, data
mov  ds, ax                 ; stocker les données dans le registre segment données ds pour les empliler

mov   ax , stack

mov    ss, ax                ; stocker la pile dans le registre de segment ss

push    N1

push    N2                  ; empiler les éléments dans la pile

push    N3

push    N4

call    fct_affichage               ; Appel de la fonction d'affichage

call  fct_addition                  ; Appel de la fonction d'addition

fct _affichage   proc  near    ;  nom_fonction    type « procédure »    near «  appel dans le même programme »

mov   cx, 4      ; initialiser le compteur de la boucle avec 4 itérations ( 4 éléments)
```

```
boucle:
    mov   ah, 02h      ; Affichage (en code ASCII)
     POP  dx        ; stocker les données  empilées dans le registre général  dx
     Int   21h
     Loop  boucle
Ret    ; retour au prgm principal
Endp
call  fct_addition    proc  near    ;  nom_fonction    type « procédure »    near « appel dans le même programme »
mov   cx, 4      ; initialiser le compteur de la boucle avec 4 itérations ( 4 éléments)
boucle:
     mov   dx, 10      ; Ajouter 10 pour chaque élément
    mov   ah, 02h      ; Affichage (en code ASCII)
    POP  dx        ; stocker les données  empilées dans le registre général  dx
     Int   21h
     Loop  boucle
Ret    ; retour au prgm principal
Endp

Mov   ax, 4cOOh              ; fin de programme
Int 21h
ends
```

# Références


- **Livres**

  [1] Architecture des ordinateurs, Cours et exercices corrigés. Auteurs : Paolo Zanella, Yves Ligier, Emmanuel Lazard. 5ème édition. DUNOD (2013).
  [2] Architecture de l'ordinateurs, Cours et exercices. Auteur : Andrew Tanenbaum. 2ème cycle école d'ingénieurs. 4ème édition. DUNOD (2018).
  [3] Architecture des ordinateurs : Fonctionnement et Technologie, les manuels de l'étudiant : Cours et exercices corrigés. Auteur : Mc BELAID. Pages Bleues (Mars 2008)

- **Polycopies de cours :**

  [4] Support de cours : Architecture des ordinateurs, note de cours et exercices, préparé par : Dr. BETAOUAF Talib Hicham, 2eme année licence en productique, département de génie électrique et électronique, Université de Tlemcen (2020/2021)
  [5] Support de cours : Architecture des Ordinateurs (AO), préparé par : Dr. BENDJIMA Mostefa, Département des Maths et Informatique, Université Tahri Mohamed de Bechar (2020/2021)
  [6] Support de cours : Architecture des ordinateurs, préparé par : Dr. BEKKOUCHE Ibtissem, département Informatique, Université d'Oran des Sciences et de la Technologie Mohamed Boudiaf USTO-MB (2020/2021)
  [7] Support de cours : Système d'exploitation, L1-INFO, préparé par : Pr. AMROUN Kamel, Département d'Informatique, Université de Béjaia (2021/2022)
  [8] Support de cours : Architectures parallèles, préparé par : Dr. BENHAMIDA Nadjette, Département d'Informatique, Université 8 mai 1945- Guelma (2021/2022).
  [9] Support de cours : Calcul Parallèle, Master I Informatique, Spécialité : Intelligence Artificielle et Multimédia, préparé par : Pr. Rachid Seghir, Département d'Informatique, Université de Batna (2020/2021).
  [10] Support de cours : Architecture des ordinateurs, L1-INFO, préparé par : Dr. Abdelkader SAADAOUI, Département Technologies de l'Informatique, Institut Supérieur des Etudes Technologiques de Radès (ISET Radès), Tunisie (2012/2013).
  [11] Polycopie de cours : Algorithmique et Structure de Données, L1-MI : Partie 1 préparé par : Dr. Slimane BELLAOUAR, Département de Mathématiques et de l'Informatique, Université de Ghardaia (2020/2021).

- **PDF :**

  [12] https://TP Centre universitaire Tipaza / ST/ 2eme Licence Electronique. ilide.info-chapitre3-programmation-en-assembleur-8086-pdf pr_079b60484de70060df2e2022ee8947a7.
  [13] https://adn56.net/pres/raspberry/20190717_projet_raspberry.pdf
  [14] https://Cours MICROPROCESSEURS DE LA FAMILLE 8086 Intel 8086 _ oumnad.pdf
  [15]https://fr.scribd.com/document/730999417/5fdd05124a1401322eec981e93f1f0dbc29a5f. Architecture et technologie des ordinateurs.pdf, Cours et exercices corrigés - 6eme édition, Paolo Zanella, Yves Ligier, Emmanuel Lazard, 2018.

[16] https://Architecture et programmation des ordinateurs.pdf, Jacques Lonchamp, IUT Nancy Charlemagne –DUT Informatique 1A.
[17] https://Programmation Assembleur x86 ; 32 et 64 bits sous Linux Ubuntu, Jean-Michel RICHER, 5ème Edition, version 2024. Destinée au cours à partir de la rentrée de Septembre 2024.
[18] https://www.univ-usto.dz/images/coursenligne/AO_BK.pdf
[19]https://inf1427.teluq.ca/teluqDownload.php?file=2018/10/Cours_INF1427_module_6.pdf. Module 6: Mémoire cache.

- **Sites web :**

[20] https://tomi.digital/es/preview/88770.
[21]Les bus, d'après le site : https://kb.planethoster.com/glossaire/bus-informatique/. Visité le :15 Février 2025.
[22] Améliorer la scalabilité grâce à une gestion efficace de la mémoire cache, by Damien Tivelet, d'après le site : https://www.kaliop.com/fr/gestion-du-cache-informatiques-evolutifs. Visité le :29 Avril 2025.
[23]https://fr.slideshare.net/slideshow/architecture-et-fonctionnements-des-ordinateurs/283113541