République Algérienne Démocratique et Populaire
Ministère de L'Enseignement Supérieur et de la Recherche Scientifique
Université d'Oran des Sciences et de la Technologie
Mohamed Boudiaf USTO-MB

Faculté des Mathématiques et d'informatique
Département Informatique

Polycopié de cours

ARCHITECTURE DES ORDINATEURS

Dr. BEKKOUCHE Ibtissem

Le cours ARC est destiné aux étudiants de 2ème année Licence
Année universitaire
2020-2021

Plan Pédagogique du cours

Matière : Architecture des Ordinateurs
Domain : Mathématiques et Informatique
Filière : 2ème année Licence Informatique
Volume Horaire du cours : 1h30 *14 semaines
Coefficient : 3
Crédits : 5
Evaluation : Contrôle continu : 40%, Examen : 60%.

Introduction

L'architecture de l'ordinateur est le domaine qui s'intéresse aux différents composants internes des machines, en explicitant leur construction et leurs interactions. Un ordinateur est un outil complexe qui peut effectuer des tâches variées et dont les performances globales dépendent des spécifications de tous ses éléments. Comprendre son architecture permet de savoir dans quelle mesure les caractéristiques propres à chaque composant influencent la réactivité de la machine en fonction de son usage : pourquoi ajouter de la mémoire accélère-t-il l'ordinateur ? Pourquoi le temps d'accès d'un disque dur n'est-il qu'un des paramètres permettant de mesurer son efficacité ? Comment les processeurs font-ils pour aller toujours plus vite ?

Objectif général 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 :

- **Culture :** Comprendre le fonctionnement de l'ordinateur dans ses mécanismes élémentaires et identifier les rôles et l'interface des différents composants matériels d'un système informatique.
- **Technique :** Manipuler les concepts basiques récurrents en informatique.
- **Informatique :** Acquérir une connaissance de programmation en langage assembleur

Plan

Les principaux points traités dans ce cours sont :

- L'introduction à la notion de l'architecture des ordinateurs
- Les principaux composants d'un ordinateur
- Les notions sur les instructions d'un ordinateur
- Le processeur MIPS R3000

Structure du polycopié

Abréviations

ACC	Accumulateur
AGP	Accelerated Graphics Port
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
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
USB	Universal Serial Bus
UV-EPROM	Ultra Violet Erasable Programmable Read Only Memory
VGA	Video Graphics Array
VRAM	Video Random Acces Memory

Chapitre I :

Introduction à l'architecture des ordinateurs

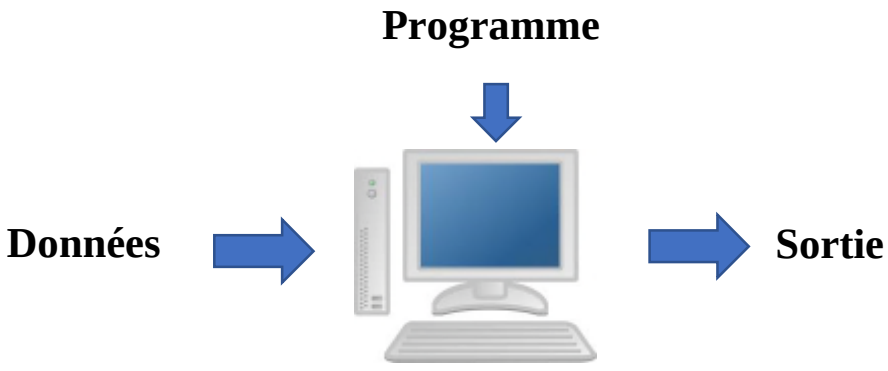
1.1. Introduction

Pour comprendre le fonctionnement d'un ordinateur, il faut d'abord connaitre sa définition, ainsi que son architecture et l'évolution de cette machine. Nous verrons également dans ce premier chapitre, les deux architectures informatiques les plus connues et utilisées dans l'ordinateur : **Von Neumann** et **Harvard.**

L'**informatique**, contraction d'**infor**mation et auto**matique**, désigne l'ensemble des sciences et des techniques en rapport avec le **traitement automatique de l'information** et ce traitement est effectué par un système, concret (machine) ou abstrait.

Qu'est-ce qu'un ordinateur ?

L'**ordinateur** est une machine électronique programmable servant au traitement de l'information codée sous forme numérique.



- Il 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* ».

L'**information** traitée par l'ordinateur peut se présenter sous forme : *numérique, texte, son, dessin ou graphique, image* ... mais aussi *instructions* composant un programme. Cette information est représentée (codée) sous forme de suites de chiffres binaires 0 et 1.

Définition ordinateur (selon le dictionnaire Hachette**)**

n. m. INFORM Machine capable d'effectuer automatiquement des opérations arithmétiques et logiques (à des fins scientifiques, administratives, comptable, etc.) à partir de programmes définissant la séquence de ces opérations.

Terminologie

1955 : Création du mot français « Ordinateur », déposé d'abord par IBM, pour désigner ce qui est en anglais un "Computer".

- **Français ➔** Ordinateur **➔** Ordre (commande et organisation)
- **Anglais ➔** Computer **➔** Calculateur
- **Arabe ➔** الكمبيوتر, الحاسوب

Qu'appelle-t-on architecture des ordinateurs ?

L'*architecture des ordinateurs* est la **discipline** qui correspond à la façon dont on conçoit les composants d'un **système informatique.**

En informatique, le terme ***architecture*** désigne l'organisation des éléments d'un système et les relations entre ces éléments. Il y a :

- L'**architecture matérielle :** concerne l'organisation des différents dispositifs physiques que l'on trouve dans un ordinateur.
  ➔ Fonctionnement logique de chaque composant et le dialogue entre les composants
- L'**architecture logicielle :** concerne l'organisation de différents programmes entre eux. ➔ Codage de l'information et jeu d'instruction de la machine càd l'ensemble des opérations que la machine peut exécuter.

1.2. Chronologie

Pour comprendre l'architecture d'un ordinateur d'aujourd'hui, il faut comprendre son évolution et comment ont fonctionné ses ancêtres et par quelles évolutions on est parvenu à l'architecture moderne des ordinateurs (voir tableau ci-dessous).

Après la machine mécanique de **Blaise Pascal** (1643), qui automatisait à l'aide de roues dentées les opérations arithmétiques et celle du Britannique **Charles Babbage** (1883), qui les enchaînait grâce à une complexe tringlerie lisant le programme sur un ruban perforé. Les « calculateurs électroniques » ont fait leur apparition dans les années 1940-1950.

Générations	*Caractéristiques*
1ère génération *1945-55 : Les ordinateurs mécaniques*	- Technologie à lampes, relais, tubes à vider, résistances. - Premiers calculateurs électroniques. **1946 EDVAC** (Electronic Discrete Variable Automatic Compute) par *Von Neumann* et mise en service en 1951 (programme et données enregistrés en mémoire)
2ème génération *1955-65 : Les ordinateurs à transistors*	- Technologie à transistors (remplacent les tubes à vides). - Apparition des langages de programmation évolués. **1955 IBM 650** 1er ordinateur fabriqué en série
3ème génération *1965-71 : Les ordinateurs à circuits intégrés*	- Technologie des Circuits Intégrés (puces) SSI/MSI (Small Scale Integration / Medium Scale Integration) qui permettent de placer un nombre important de transistors sur une même puce en silicium. - Avènement du système d'exploitation complexe. **1971 Kenback 1**1er micro-ordinateur
4ème génération *1971-77 : La micro-informatique*	- Technologie LSI (Large SI). - Avènement de réseaux de machines. **1976 Apple I** (S. Wozniak & S. Jobs,) muni de clavier
5ème génération *1977 et plus : des ordinateurs partout*	- Technologies VLSI / ULSI (Very Large / Ultra large SI) l'intégration de milliers à des milliards de transistors sur une même puce. **1981 PC (Personal Computer)** par IBM
Et depuis 1990 *Nouveaux outils* *1990 Nouveaux outils*	- Miniaturisation des composants matériels, on parle de la **nanotechnologie.** **2007 iPhone** 1er smartphone par Appel

1.3. Machine de Von Neumann et machine de Harvard

Il existe deux architectures informatiques, qui diffèrent dans la manière d'accéder aux mémoires : L'Architecture de **Von Neumann** et l'Architecture de **Harvard.**

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. Dans l'architecture de **Harvard**, les programmes et les données sont stockés et gérés par différents sous-systèmes. C'est la différence essentielle entre ces deux architectures.

1.3.1. Machine de Von Neumann

A la fin de **1946** « John Von Neumann » un physicien et mathématicien d'origine Hongroise, propose un modèle d'ordinateur qui fait abstraction du programme et se lance dans la construction d'un EDVAC (Electronic Discrete Variable Automatic Computer). Il a introduit 2 nouveaux concepts dans le traitement digital de l'information :

a. Programme enregistré

Von Neumann, a eu l'idée d'utiliser les mémoires du calculateur pour emmagasiner les programmes : d'où le nom de la machine à **programme enregistré** donné au nouveau type de calculateur.

b. Rupture de séquence

Von Neumann, a eu l'idée de rendre automatique les opérations de décision logique en munissant la machine d'une instruction appelée **branchement conditionnelle** (ou rupture de séquence conditionnelle).

❖ Description de l'architecture de Von Neumann

C'est **Von Neumann** qui a défini en **1944** l'architecture des ordinateurs modernes encore largement utilisés aujourd'hui (seules les technologies ont changé).

Dans cette architecture, on utilise une **seule mémoire** pour les programmes et les données, tandis que l'unité centrale (processeur) est composée d'unité de contrôle et d'unité arithmétique et logique. Elle est constituée d'un bus de données (programme et données) et d'un bus d'adresse (programme et données).

L'architecture de Von Neumann (fig. 1) décompose l'ordinateur en quatre parties :

- **Unité Arithmétique et Logique (UAL)** : effectue les calculs (les opérations de base).
- **Unité de contrôle (UC) :** commande les autres unités. Elle est chargée du séquençage des opérations
  - Envoie des signaux de contrôle aux autres unités
  - Supervise le fonctionnement de l'UAL
  - Envoie des signaux d'horloge aux autres unités…
- **Mémoire** : dispositif de stockage des information (données et programme).
- **Dispositifs d'Entrée-Sortie** : permettent l'échange d'informations avec les dispositifs extérieurs.

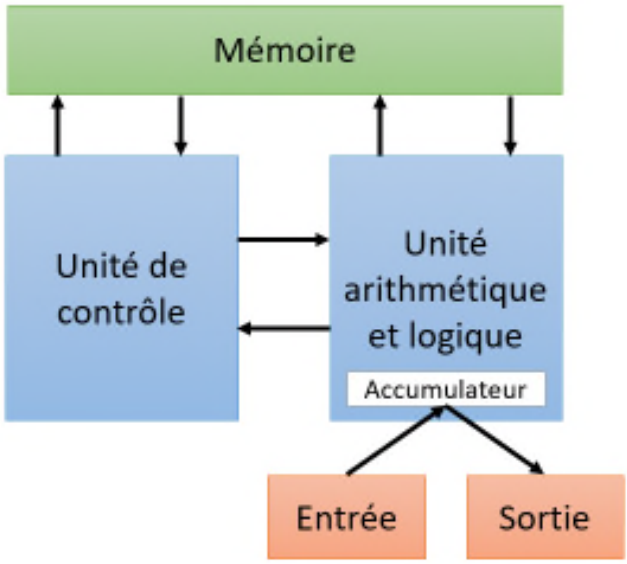


Figure 1: L'architecture de Von Neumann.

Les différents organes du système sont reliés par des voies de communication appelées **bus** (bus d'adresse et bus de données).

C'est avec cette architecture que sont construits tous les ordinateurs, du nanoprocesseur (que l'on trouve par exemple dans une machine à laver) au super-ordinateur (calcul intensif).

1.3.2. Machine de Harvard

Le nom de cette structure vient du nom de l'université de **Harvard** où une telle architecture a été mise en pratique pour la première fois avec le **Harvard Mark 1** créé par **Howard Aiken** et fut construit par **IBM** en **1944**. Également appelé par IBM **Automatic Sequence Controlled Calculator (ASCC)**. Il fut le premier ordinateur à utiliser des systèmes de mémoire séparés (des données et des instructions).

❖ Description de l'architecture de Harvard

Dans l'architecture dite de **Harvard** (mise au point dans cette université américaine en **1930**), on **sépare** systématiquement **la mémoire de programme** de **la mémoire des données** : l'adressage de ces mémoires est indépendant. Une architecture simple de **Harvard** (fig. 2), constituée d'un bus de données, d'un bus de programme et de deux bus d'adresses.

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. D'ailleurs, la mémoire de programme est également utilisée en partie comme mémoire de données pour obtenir encore plus de possibilités de traitement avec des algorithmes complexes.

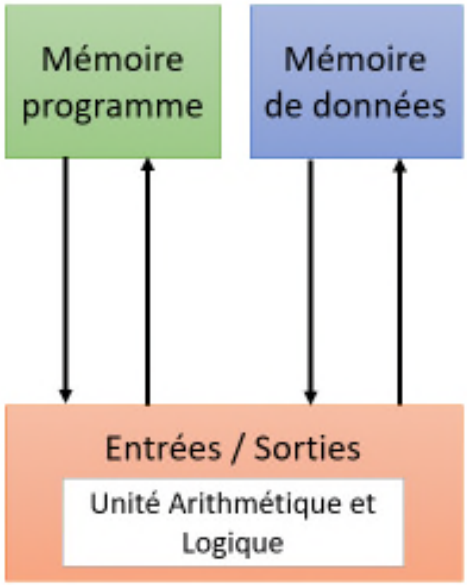


Figure 2 : L'architecture de Harvard.

Architecture Harvard n'est normalement **utilisé** que dans les deux systèmes spécialisés ou à des usages très spécifiques, elle est utilisée dans le **traitement du signal numérique spécialisé** (DSP Digital Signal Processing), typiquement pour le traitement des données vidéo et audio. Elle est également utilisée dans de nombreux petits **microcontrôleurs** dans des applications de l'électronique.

1.3.3. La différence entre l'architecture de Von Neumann et Harvard

Différence	Architecture de Von Neumann	Architecture de Harvard
Nom	Elle porte le **nom** du mathématicien et informaticien **John Von Neumann**	Le **nom** provient de « **Harvard Mark I** », un ancien ordinateur à relais, projet réalisé à l'université Harvard
Conception	La conception de l'**architecture** de Von Neumann est **simple.**	La conception de l'**architecture** de Harvard est **compliquée**.
Système de mémoire	Elle a besoin **d'une seule mémoire** pour les instructions et les données.	Elle a besoin de **deux mémoires** pour les instructions et les données
Système de bus	Ne requiert qu'**un seul bus** pour **les instructions et les données.**	Nécessite **un bus séparé** pour **les instructions et les données.**
Traitement des instructions	Le **processeur** a besoin de **deux cycles** d'horloge pour terminer une instruction.	Le **processeur** peut compléter une instruction en **un cycle.**
Performance	**Faible performance** par rapport à l'architecture de Harvard.	Plus facile à canaliser, donc de **hautes performances** peuvent être atteintes.
Coût	Coût **moins cher**.	Coût **relativement élevé**.
Utilisation	Principalement utilisée sur toutes les machines (*des ordinateurs de bureau, des ordinateurs portables et stations de travail hautes performances).*	Concept utilisé principalement dans les microcontrôleurs et le traitement du signal numérique (DSP 'Digital Signal Processor')

Remarque :

- Dans *l'architecture de Von Neumann* le processeur a besoin de **deux cycles d'horloge** pour exécuter une instruction, il lit d'abord l'instruction (mémoire programme) après il accède à la donnée (mémoire donnée) car il n'y a qu'une seule mémoire.
- Tandis que dans *l'architecture de Harvard* le processeur prend un **cycle d'horloge** pour compléter une instruction, il peut lire une instruction et accéder à la donnée en même temps car les deux mémoires sont séparées.

1.4. Conclusion

Dans ce chapitre introductif, nous avons expliqué la définition d'un ordinateur et la signification du terme architecture en informatique, son évolution par laquelle des modifications sont intervenues dans l'architecture moderne des ordinateurs d'aujourd'hui et ainsi que les architectures de *Von Neumann* et de *Harvard.* Dans le chapitre suivant, nous verrons en détail les principaux composants d'un ordinateur.

Chapitre II :

Principaux composants d'un ordinateur

2.1. Introduction

Ce deuxième chapitre s'intéresse aux éléments fondamentaux d'un ordinateur et leurs fonctionnements. Nous présenterons les principaux modules constituant l'architecture d'un ordinateur type. Nous expliquerons la fonctionnalité de chacun de ces modules et de leurs relations fonctionnelles dans l'ordinateur. Il s'agit ici uniquement de présenter de manière globale le fonctionnement de l'ordinateur.

En premier lieu, nous parlerons de la carte mère et ses caractéristiques, de l'unité de calcul (Unité Arithmétique et Logique qui se trouve dans le processeur) ainsi que des composants de transport d'informations (les registres pour le processeur et les bus pour le reste des éléments). Ensuite, nous parlerons d'un composant indispensable de l'ordinateur 'la mémoire', qui se résume souvent à une simple fonction de stockage. On distingue plusieurs catégories de mémoires différenciées par leurs caractéristiques (adressage, performances, accès…) : Mémoire interne et Mémoire cache.

Un **ordinateur** est une machine électronique capable de résoudre et traiter des problèmes en appliquant des instructions probablement définies. Donc il permet :

- D'**acquérir** des informations.
- De **conserver** des informations.
- D'**effectuer** des traitements sur les informations.
- De **restituer** des informations.

Concernant l'**organisation de base d'un ordinateur** (fig. 3), 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.

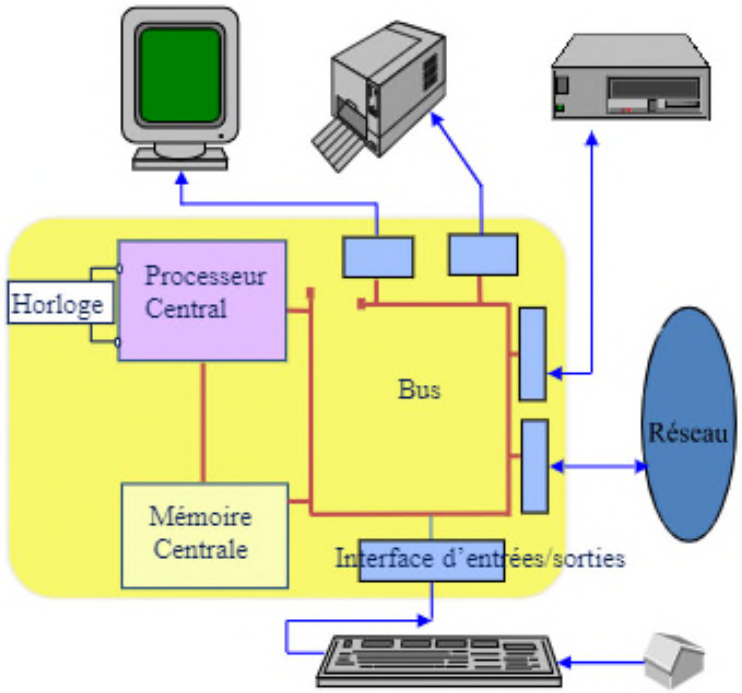


Figure 3 : La structure d'un ordinateur général.

2.2. Composants de l'ordinateur

Un ordinateur est un ensemble de **composants électroniques modulaires**, c'est-à-dire des composants pouvant être remplacés par d'autres composants ayant éventuellement des caractéristiques différentes, capables de faire fonctionner des programmes informatiques.

La mise en œuvre de ces systèmes s'appuie sur deux modes de réalisation distincts :

- ***Le matériel*** (**hardware** mot signifiant quincaillerie) correspond à l'aspect concret ou physique de l'ordinateur : unité centrale, mémoire, organes d'entrées-sorties, etc…
- ***Le logiciel*** (**software** mot fabriqué pour les besoins de la cause en remplaçant hard 'dur' par soft 'mou') désigne au contraire tout ce qui n'est pas matériel est qui correspond à un ensemble d'instructions, appelé programme, qui sont contenues dans les différentes mémoires du système d'un ordinateur et qui définissent les actions effectuées par le matériel.

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

2.2.1 Présentation de la carte mère

L'élément constitutif principal et essentiels de l'ordinateur est la ***carte* mère** (en anglais « **Mainboard** » ou « **Motherboard** », parfois abrégé en « **Mobo** »).

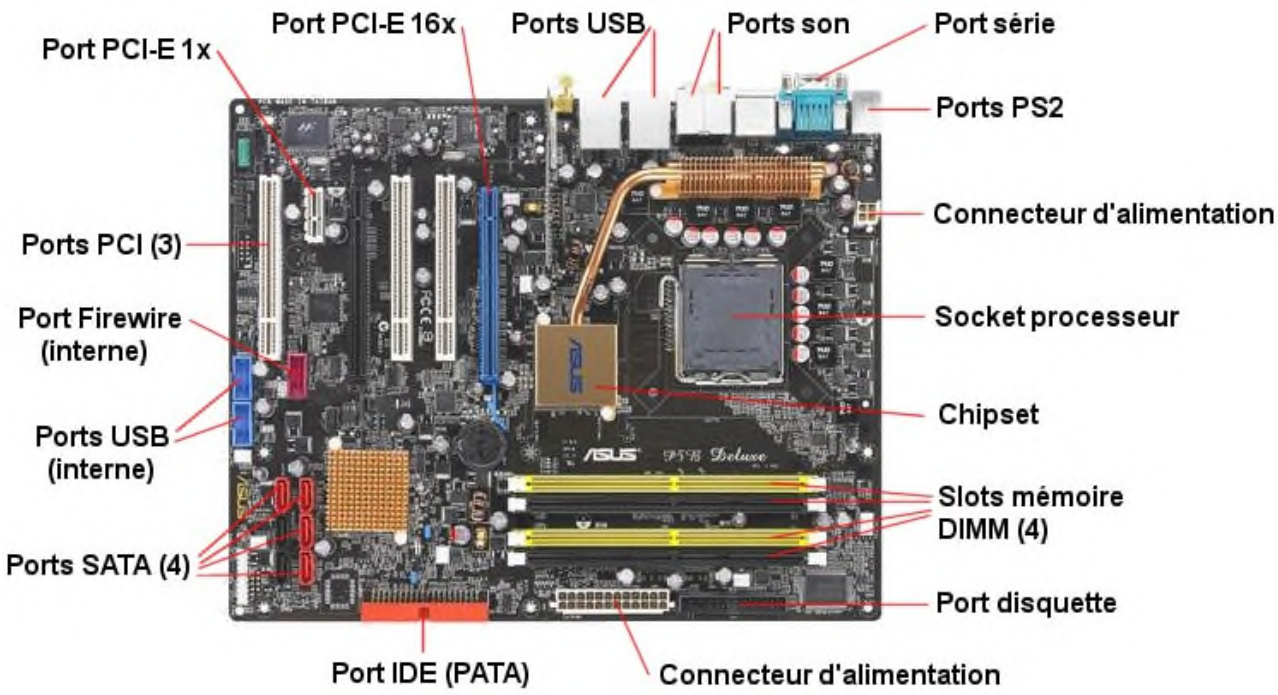


Figure 4 : Le modèle de carte mère.

La **carte mère** (fig. 4 https://lacartemere.wordpress.com/) est la plus grande carte électronique prenant la forme d'un circuit imprimé. C'est le système nerveux de l'ordinateur car elle assemble et met en relation tous les composants matériels. 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.

2.2.2 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 par ce terme la géométrie, 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,****)***, en 2009 ***ITX (****Information Technology eXtended****)***, … *etc.*

b. **Le chipset : *(traduisez jeu de composants ou jeu de circuits)* :** c'est une interface d'entrée/sortie. Elle est constituée par un jeu de plusieurs composants chargés 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 Service)* :** 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 de type EEPROM) et agit comme une interface entre le système d'exploitation et le matériel.

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

1. ***Sockets :*** un **socket** (en anglais) est le nom du connecteur destiné au processeur. Il s'agit d'un connecteur de forme carré possédant un grand nombre de petits connecteurs sur lequel le processeur vient directement s'enficher.
2. ***Slots :*** un **slot** (en anglais) est une fente rectangulaire dans laquelle on insère un composant. Selon le type de composant accueilli, on peut utiliser d'autres mots pour designer des slots :
   - Un port d'extension ou un connecteur d'extension pour enficher une ***carte d'extension***
   - Un support pour enficher une barrette de ***mémoire vive***
   - Un slot pour enficher un ***processeur***, à ne pas confondre avec un socket car certains processeurs conditionnés sous forme de cartouche.

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) :

1. ***Les connecteurs internes :*** Il existe des connecteurs internes pour connecter des cartes d'extension (***PCI*** 'Peripheral Component Interconnect', ***ISA*** 'Industry Standard Architecture', ***AGP*** 'Accelerated Graphics Port') ou des périphériques de stockage de masse (***IDE*** aussi appelé PATA 'Parallel ATA', ***SCSI*** 'Small Computer System Interface', ***SATA*** 'Serial ATA').
2. ***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**'**, ***DisplayPort***, ***audio analogiques***, ***audio numériques***, ***Firewire.***

Remarque :

- Il existe à autres éléments embarqués dans la carte mère (intégrés sur son circuit imprimé) comme l'*horloge* et la *pile du CMOS*, le *bus système* et les *bus d'extension*.
- Les cartes mères récentes embarquent généralement un certain nombre de périphériques multimédia et réseaux pouvant être désactivés : *carte réseau intégrée, carte graphique intégrée, carte son intégrée, contrôleurs de disques durs évolués.*
- Les bus de connexions filaires tendent à être remplacés par des systèmes de communications sans fils. A l'heure actuelle, il existe :
  - ***Bluetooth*** qui va servir à connecter des périphériques nécessitant des bandes passantes faibles (clavier, souris, etc…).
  - ***WIFI*** (WIreless FIdelity Network) qui permet de connecter des ordinateurs en réseau.

2.3. Processeur

Le **processeur** (**CPU**, pour *Central Processing Unit*, soit *Unité Centrale de Traitement*) 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. Sa puissance est exprimée en Hertz.

Electroniquement, le processeur est une puce (circuit intégré complexe) d'environ 4cm de côté et quelques millimètres d'épaisseur en silicium regroupant quelques centaines de millions de transistors, qui chauffe beaucoup car il est très sollicité. Au-dessus du radiateur, un ventilateur va se charger d'évacuer cette chaleur (fig. 5 lien : https://cours-informatique-gratuit.fr/cours/processeur-et-carte-mere/ et https://blogue.bestbuy.ca/ordinateurs-portables-et-tablettes/ordinateurs/tout-ce-que-vous-voulez-savoir-sur-le-processeur-de-votre-ordinateur-portable).

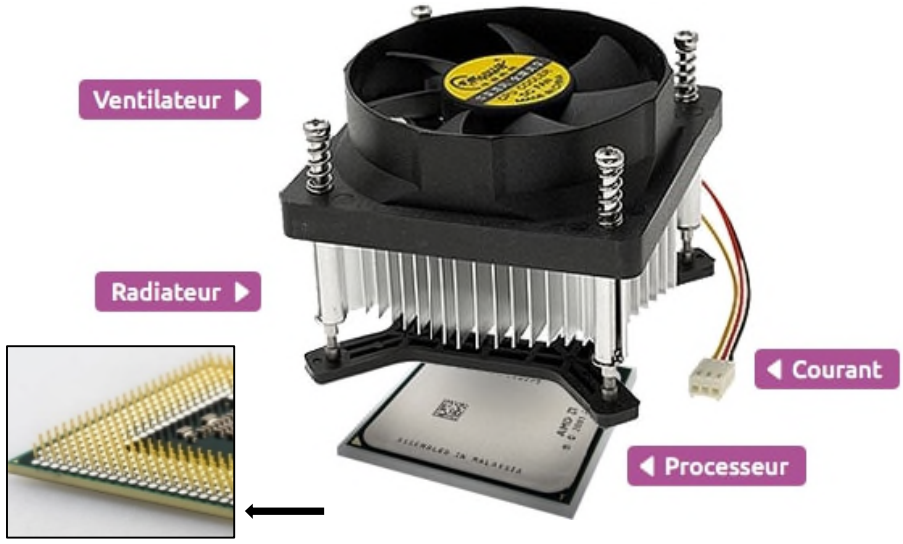


Figure 5 : Le processeur.

2.3.1 Unités d'un processeur

Le processeur est constitué d'un ensemble d'unités fonctionnelles reliées entre elles (la figure 6 ci-dessous présente son architecture générale). Les rôles des principaux éléments d'un microprocesseur sont les suivants :

1. Une **unité d'instruction** (ou ***unité de commande***, en anglais *control unit*), qui contrôle toutes les composantes et qui lit les données arrivantes, les décode puis les envoie à l'unité d'exécution. L'unité d'instruction est notamment constituée des éléments suivants :
    a. **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.
    b. **Compteur ordinal** contenant l'adresse de l'instruction en cours.
    c. **Registre d'instruction** contenant l'instruction à exécuter.
    d. **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.

2. Une **unité d'exécution** (ou ***unité de traitement***), qui accomplit les tâches que lui a donné l'unité d'instruction. L'unité d'exécution est notamment composée des éléments suivants :
3. L'**unité arithmétique et logique** (notée **UAL** ou en anglais *ALU* pour *Arithmetical and Logical Unit*) pour le traitement des données.
   a. L'**unité de virgule flottante** (notée **FPU**, pour *Floating Point Unit*), qui accomplit les calculs complexes non entiers que ne peut réaliser l'unité arithmétique et logique.
   b. Le **registre d'état**.
   c. Le **registre accumulateur**.
4. Une **unité de gestion des bus** (ou ***unité d'entrées-sorties***), qui gère les flux d'informations entrant et sortant, en interface avec la mémoire vive du système.

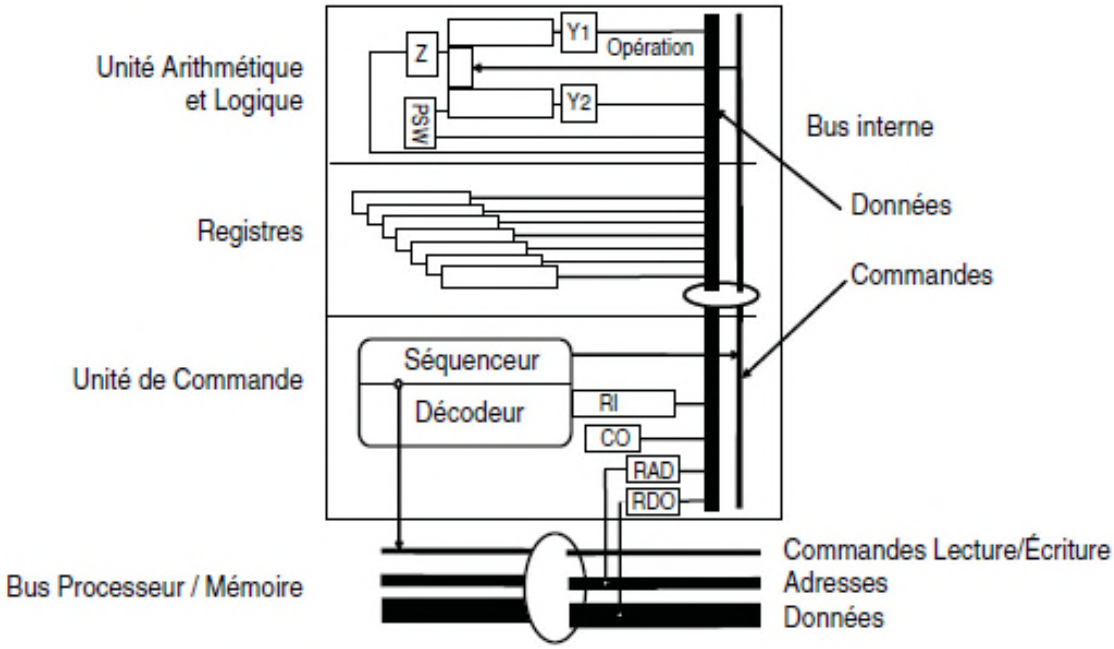


Figure 6 : L'architecture générale d'un processeur

2.3.2 Unité Arithmétique et Logique

C'est le cœur du processeur, l'**UAL** (l'abrégé de l'unité arithmétique et logique) est chargé de l'exécution de tous les calculs que peut réaliser le microprocesseur

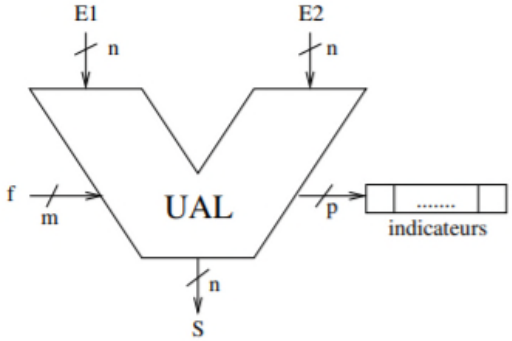


Figure 7 : L'unité arithmétique et logique.

C'est un circuit combinatoire (fig. 7) qui produit un résultat (S) sur n bits en fonction des données présentes sur ses entrées (E1 et E2) et de la fonction à réaliser (f) et met à jour les indicateurs.

➢ **Fonctions de l'UAL :**

L'UAL permet de réaliser différents types d'opérations sur des données de la forme S= f (E1, E2) :

- ➢ Des opérations arithmétiques : additions, soustractions, ...
- ➢ Des opérations logiques : ou, et, ou exclusif, ...
- ➢ Des décalages et rotations.

Elle met par ailleurs à jour **des indicateurs d'états** (ou drapeaux ou flag) en fonction du résultat de l'opération effectuée :

- **S** (Signe) : le bit de signe du résultat de la dernière opération arithmétique.
- **Z** (Zéro) : indicateur mis à 1 si le résultat de l'opération est 0.
- **N** (Négatif) : indicateur mis à 1 pour un résultat négatif (bit le plus à gauche égal à 1).
- **C** (Carry-out) : mis à 1 en cas de retenue ou débordement en contexte non signé.
- **V** (Overflow) : mis à 1 en cas de débordement en contexte signé.
- …

Exemple : Dans notre exemple, l'UAL possède deux registres d'entrée (E1 et E2) et un registre de sortie (S). Pour faire une addition :

– la première donnée est placée dans E1 via le bus interne de données.

– la seconde donnée est placée dans E2 via le bus interne de données.

– la commande d'addition est délivrée au circuit d'addition via le bus interne de commandes.

– le résultat est placé dans le registre S.

2.4. BUS

Un bus est un ensemble de fils (conducteurs électriques) qui assure la transmission des informations binaires entre les éléments 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.

2.4.1. Caractéristiques d'un bus

Un bus est caractérisé par :

a. **Sa largeur :** un bus est caractérisé par le volume d'informations qui peuvent être envoyées en parallèle (exprimé en bits) correspond au nombre de lignes physiques sur lesquelles les données sont envoyées de manière simultanée. Ainsi la **largeur** désigne le nombre de bits qu'un bus peut transmettre simultanément.

1 fil transmet un bit, 1bus à n fils = bus n bits

Exemple : une nappe de 32 fils permet ainsi de transmettre 32 bits en parallèle.

b. **Sa vitesse :** est le nombre de paquets de données envoyés ou reçus par seconde. Elle est également définie par sa **fréquence** (exprimée en Hertz).
On parle de **cycle** pour désigner chaque envoi ou réception de données. Un cycle mémoire assure le transfert d'un mot mémoire :

Cycle mémoire (s)=1 / fréquence

c. **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.

Débit (octets/s) = (nombre de transferts par seconde * largeur) / 8

Bande passante (en Mo/s) = largeur bus (en octets) * fréquence (en Hz)

Exercice : Un bus de 8 bits, cadencé à une fréquence de 100 MHz. Calculer le taux de transfert.

Solution :

Le bus possède donc un taux de transfert égal à :

Taux de transfert = largeur bus * fréquence

$= 8 * 100.10^6 = 8.\ 10^8$ bits/s $= 10^8$ octets/s $= 10^5$ K octets/s = **10^2 M octets/s**

2.4.2. Différents types de bus

Selon la nature de l'information à transporter, on retrouve trois types de bus d'information (fig. 8) en parallèle dans un système de traitement programmé de l'information:

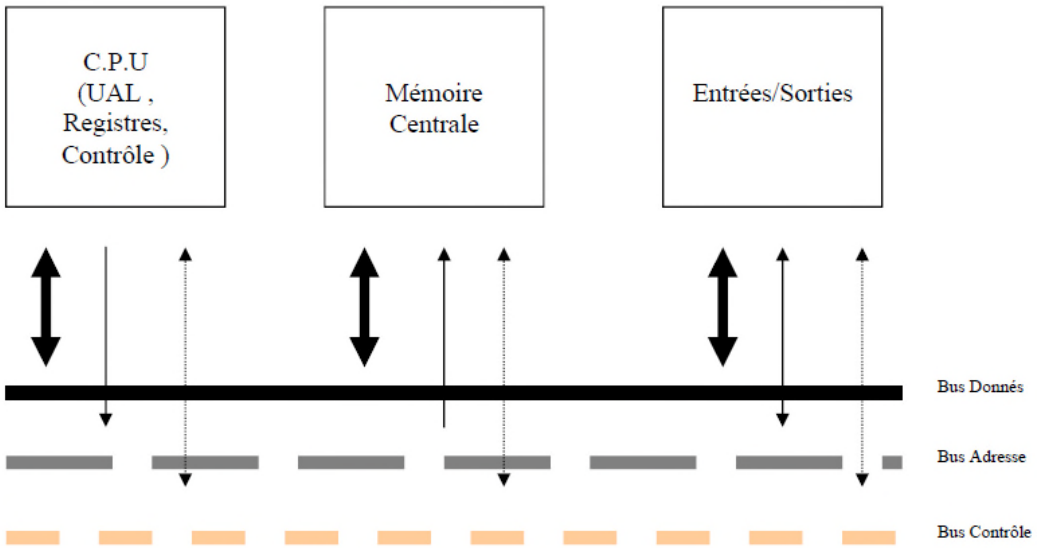


Figure 8 : Les différents types de bus.

a. **Bus de données** : c'est un bidirectionnel, il assure le transfert des informations (opérations et données) entre le microprocesseur et son environnement, et inversement. Son nombre de lignes est égal à la capacité de traitement du microprocesseur.

b. **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) selon la demande du processeur pour lire ou écrire une donnée. Il peut avoir **2^n** emplacements, avec **n** = nombre de conducteurs du bus d'adresses.

c. **Bus de commande (bus de contrôle)** : c'est un bidirectionnel, constitué par quelques conducteurs qui assurent la synchronisation des flux d'informations. Il transporte les signaux de contrôle (lecture ou écriture mémoire, opération d'entrées/ sorties, …), dont les éléments sont disponibles sur les bus donnés ou adresses.

2.4.3. Types de bus de données

Il existe deux grands types de bus de données (fig. 9 lien : https://www.commentcamarche.net/contents/770-port-serie-et-port-parallele) selon le type de transmission :

a. **Les bus séries :** ils permettent des transmissions sur de grandes distances. Ils utilisent une seule voie de communication sur laquelle les bits sont envoyés les uns à la suite des autres. Exemples : USB, SATA.
b. **Les bus parallèles :** sur un bus parallèle plusieurs bits sont transmis simultanément. Ils sont utilisés sur des distances courtes par exemple ; pour relier le processeur à la mémoire. Exemple : PATA.

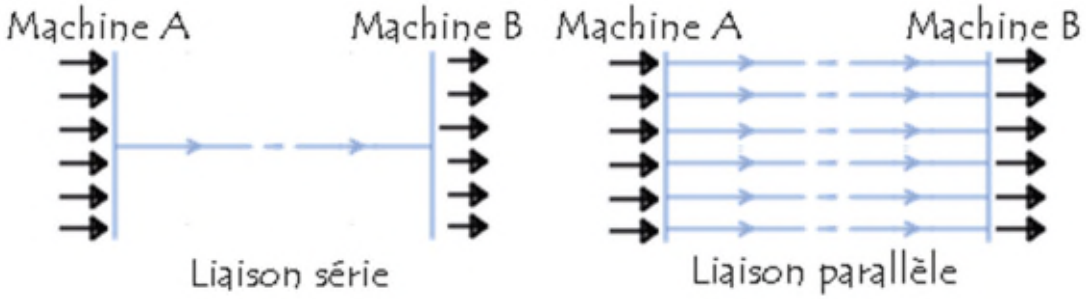


Figure 9 : Les types de bus de données.

2.4.4. Principaux bus

On distingue généralement sur un ordinateur deux principaux bus :

a. **Bus système** (*bus interne, en anglais internal bus ou front-side bus, noté FSB*) : permet au processeur de communiquer avec la mémoire centrale du système (mémoire vive ou RAM) comme les bus d'adresse et de données.
b. **Bus d'extension** (*bus d'entrée/sortie*) : permet aux divers composants liés à la carte-mère (USB, série, parallèle, cartes branchées sur les connecteurs PCI, disques durs, lecteurs et graveurs de CD-ROM, etc.) de communiquer entre eux. Il permet aussi l'ajout de nouveaux périphériques grâce aux connecteurs d'extension (appelés slots) qui lui y sont raccordés.

Remarque :

La performance d'un bus est conditionnée par sa capacité de transport simultané (16, 32, 64 bits …) et par l'électronique qui le pilote (**le chipset**).

2.5. Registres

Lorsque le processeur exécute des instructions en cours de traitement, les données sont temporairement stockées dans de petites mémoires (rapides de 8, 16, 32 ou 64 bits) que l'on appelle *registres*. Suivant le type de processeur le nombre global de registres peut varier d'une dizaine à plusieurs centaines.

Il existe deux types de registres :

a. **Registres visibles par l'utilisateur (manipulable par le programmeur) :** un registre utilisateur est un registre référençable pour les instructions exécutées par le processeur. On trouve différentes catégories :
   - **Données :** ne peuvent pas être employées pour le calcul d'adresses.
   - **Adresses :** souvent dévolues à un mode d'adressage particulier (contenant des valeurs de base ou d'index).
   - **Conditions (flags)** : constitués d'une suite de bits indépendants dont chacun est positionné en fonction du résultat d'une opération.
   - **Autres :** n'ont pas de fonction spécifique.

Les registres de l'UAL (unité arithmétique et logique), qui sont accessibles au programmeur, contrairement aux registres de l'UCC (unité de contrôle et commande). On dénombre :

- **Registre accumulateur (ACC),** stockant les résultats des opérations arithmétiques et logiques des données en cours de traitement.
- **Registres arithmétiques** : destinés pour les opérations arithmétiques (+, -, *, /, complément à 1, ...) ou logiques (NOT, AND, OR, XOR), l'accumulateur (ACC) pour stocker le résultat, ….
- **Registres d'index :** pour stocker l'index d'un tableau de données et ainsi calculer des adresses dans ce tableau.
- **Registre pointeur :** d'une pile ou de son sommet.
- **Registres généraux :** pour diverses opérations, exemple stocker des résultats intermédiaires.
- **Registres spécialisés :** destinés pour certaines opérations comme les registres de décalages, registres des opérations arithmétiques à virgule flottante, …etc

b. **Registres de contrôle et des statuts (non visible par le programmeur) :** utilisés par l'unité de commandes pour contrôler l'activité du processeur et par des programmes du système d'exploitation pour contrôler l'exécution des programmes. Quatre registres sont essentiels à l'exécution d'une instruction. Ils sont utilisés pour l'échange avec la mémoire principale :
   - **Le compteur ordinale** (CO ou PC, pour Program Counter) : contient l'adresse de la prochaine instruction à exécuter.
   - **Le registre d'instruction** (RI ou IR, pour Instruction Register) : contient l'instruction en cours de traitement.
   - **Le registre d'adresse mémoire** (MAR, pour Memory Adress Register) : contient une adresse mémoire et il est directement connecté au bus d'adresse.
   - **Le registre tampon mémoire** (MBR, pour Memory Buffy Register) **:** contient un mot de données à écrire en mémoire ou un mot lu récemment. Il est directement connecté au bus de données et il fait le lien avec les registres visibles par l'utilisateur.

Comme registre de statut, le **registre d'état** (PSW, pour Processor Status Word) contient des informations de statut. Il permet de stocker des indicateurs sur l'état du système (retenue, dépassement, etc.) et qui dépend du résultat donné par l'UAL.

2.6. Mémoire

Un ordinateur a deux caractéristiques essentielles qui sont **la vitesse** à laquelle il peut traiter un grand nombre d'informations et **la capacité de mémoriser ces informations**.

On appelle « **mémoire** » tout dispositif capable d'**enregistrer**, de **conserver** aussi longtemps que possible et de les **restituer** à la demande. Il existe deux types de mémoire dans un système informatique :

- **La mémoire centrale** (ou interne) permettant de mémoriser temporairement les données et les programmes lors de l'exécution des applications. Elle est très rapide, physiquement peu encombrante mais coûteuse. C'est *la mémoire de travail de l'ordinateur*.
- **La mémoire de masse** (ou auxiliaire, externe) permettant de stocker des informations à long terme, y compris lors de l'arrêt de l'ordinateur. Elle est plus lente, assez encombrante physiquement, mais meilleur marché. C'est *la mémoire de sauvegarde des informations*.

2.6.1. Caractéristiques d'une mémoire

Les principales caractéristiques d'une mémoire sont les suivantes :

1- **La capacité** (la taille) : représentant le volume global d'informations (en bits et aussi souvent en octet.) que la mémoire peut stocker.
2- **Le format des données** : correspondant au nombre de bits que l'on peut mémoriser par case mémoire. On dit aussi que c'est la largeur du **mot** mémorisable.
3- **Le temps d'accès** : correspondant à l'intervalle de temps entre la demande de lecture/écriture (en mémoire) et la disponibilité sur le bus de donnée T_a.
4- **Le temps de cycle** : représentant l'intervalle de temps minimum entre deux accès successifs de lecture ou d'écriture T_c. On a $T_a < T_c$ à cause des opérations de synchronisation, de rafraîchissement, de stabilisation des signaux, … etc. On $T_c = T_a +$ **temps de rafraîchissement mémoire.**
5- **Le débit** (vitesse de transfert ou bande passante) : définissant le volume d'informations échangées (lues ou écrites) par unité de temps (seconde), exprimé en bits par seconde : **Débit = n / T_c** et **n** est le nombre de bits transférés par cycle.
6- **La non volatilité :** caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement et volatile dans le cas contraire.

2.6.2. Modes d'accès

Le mode d'accès à une mémoire (fig. 10) dépendant surtout de l'utilisation que l'on veut en faire :

a. Direct ou Aléatoire :

- La recherche s'effectue via une adresse Mémoire à accès aléatoire, il s'agit du mode le plus employé.
- Le temps d'accès est identique car chaque mot mémoire est associé à une adresse unique.
- Les opérations associées à ce mode d'accès : *lecture(adr), écriture (adr, donnée).*

- Il est utilisé par : les mémoires qui composent la mémoire principale (mémoire vive) et quelques mémoires caches.

b. Associatif *(mémoire adressable par le contenu) :*

- La recherche s'effectue en parallèle sur toutes les cases mémoires via une clé et non via un index numérique. Donc un mot est retrouvé par une partie de son contenu.
- Le temps d'accès est constant.
- Les opérations associées à ce mode d'accès : *écriture (clé, donnée), lecture(clé), existe (clé), retirer(clé).*
- Il est employé principalement par les mémoires caches.

c. Semi direct ou Semi séquentiel *(Intermédiaire entre séquentiel et direct) :*

- Accès direct à un bloc de données ou cylindre (contenant la donnée recherchée) via son adresse unique puis déplacement séquentiel jusqu'à la donnée recherchée.
- Le temps d'accès est variable.
- Les opérations associées à ce mode d'accès : *lecture (bloc, déplacement), écriture (bloc, déplacement, donnée).*
- Il est employé par les disques (durs ou souple).

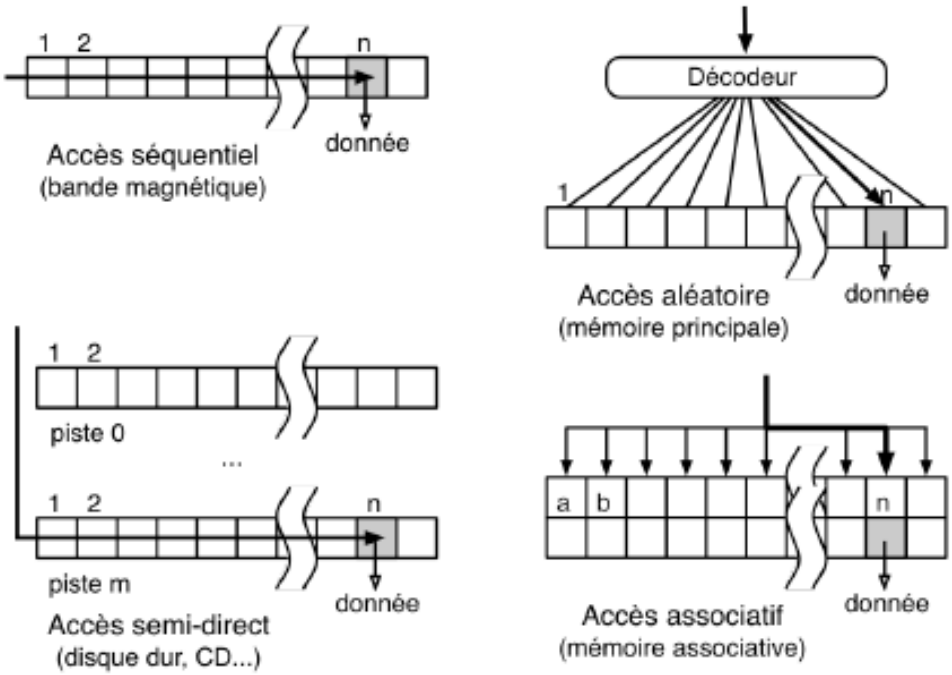


Figure 10 : Les différentes méthodes d'accès.

Remarque :

- L'accès direct est similaire à l'accès à une case d'un tableau. On accède directement à n'importe quelle case (information) directement par son indice (adresse).
- Pour un disque magnétique, l'accès à la piste est direct, puis l'accès au secteur est séquentiel. Donc c'est un accès semi-séquentiel : combinaison des accès direct et séquentiel.
- Il y a aussi un autre accès qui est **l'accès séquentiel**. C'est l'accès le plus lent il est similaire à l'accès d'une information dans une liste chainée. Pour accéder à une information, il faut parcourir toutes les informations qui la précède exemple : bandes magnétiques (K7 vidéo). Le temps d'accès est variable selon la position de l'information recherchée.

2.7. Mémoire interne

La mémoire centrale (MC) représente **l'espace de travail** de l'ordinateur car c'est l'organe principal de **rangement** des informations utilisées par le processeur.

Dans une machine (ordinateur / calculateur) pour **exécuter** un programme il faut le charger (copier) dans la mémoire centrale. Le **temps d'accès** à la mémoire centrale et **sa capacité** sont deux éléments qui influent sur le **temps d'exécution** d'un programme (performance d'une machine).

Les mémoires composant la mémoire principale sont des mémoires à base de *semi-conducteurs*, employant un mode d'accès aléatoire. Elles sont de deux types : volatiles ou non. Voici un schéma qui résume les différents types de mémoires :

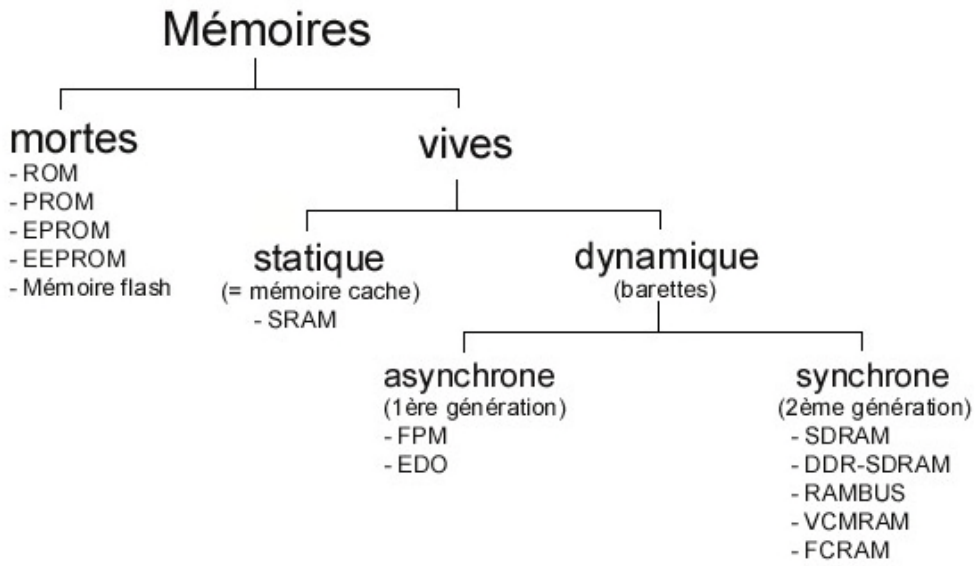


Figure 11 : Les différents types de mémoires semi-conducteurs.

2.7.1. Organisation d'une mémoire centrale

Cette mémoire est constituée de **circuits élémentaires** nommés ***bits*** (*binary digit*). Il s'agit de circuits électroniques qui présentent deux états stables codés sous la forme d'un **0** ou d'un **1**. De par sa structure la mémoire centrale permet donc de **coder les informations** sur la base d'un alphabet **binaire** et toute information stockée en mémoire centrale est représentée sous la forme d'une suite de digits binaires.

Pour stocker l'information la mémoire est **découpée** en cellules mémoires : ***les mots mémoires***. Donc une mémoire peut être représentée comme une *armoire* de rangement constituée de différents *tiroirs* où chaque tiroir représente alors une *case mémoire* (mots mémoires) qui peut contenir un seul élément (exemple fig. 12).

Chaque mot est constitué par un certain nombre de bits qui définissent sa taille. On peut ainsi trouver des mots de 1 bit, 4 bits (*quartet*) ou encore 8 bits (*octet* ou *byte*), 16 bits voire 32 ou 64 bits. Chaque mot est repéré dans la mémoire par une ***adresse***, un numéro qui identifie le mot mémoire. Ainsi un mot est un contenant accessible par son adresse et la suite de digits binaires composant le mot représente le contenu ou valeur de l'information (Données ou instruction).

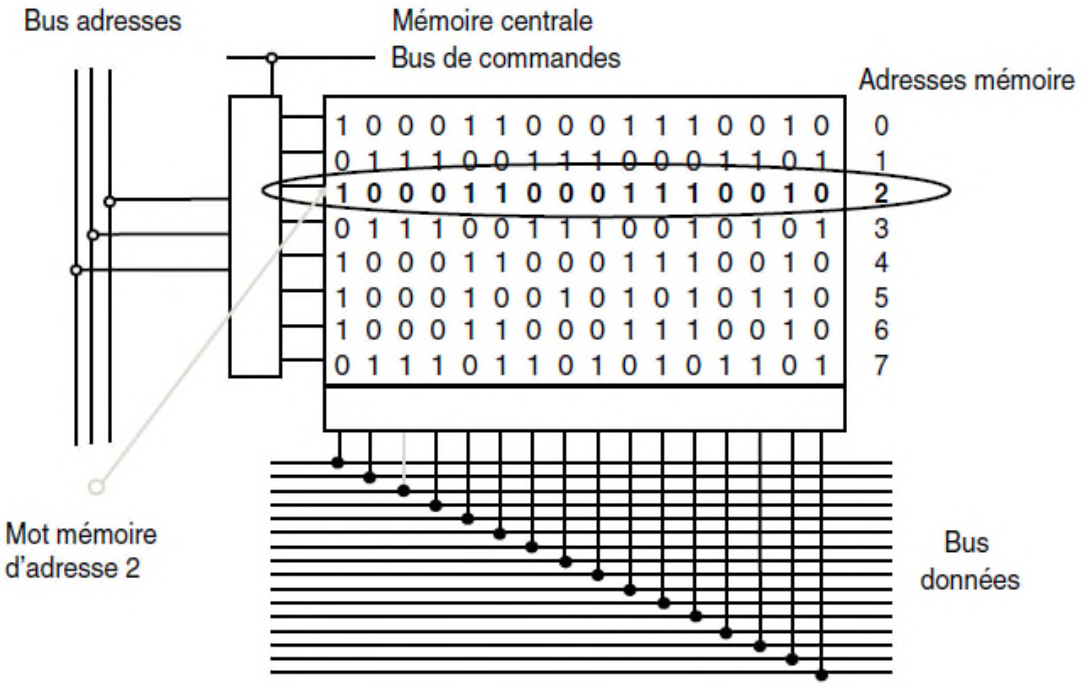


Figure 12 : L'organisation d'une mémoire centrale.

Sachant que :

- La capacité de stockage de la mémoire est définie comme étant le nombre de mots constituant.
- Avec une adresse de ***n*** bits il est possible de référencer au plus $\mathbf{2^n}$ cases mémoire
- Chaque case est remplie par un mot de données (sa longueur ***m*** est toujours une puissance de **2**).

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

- Le nombre de fils d'adresses d'un boîtier mémoire définit donc le nombre de cases mémoire que comprend le boîtier.

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

- Le nombre de fils de données définit la taille des données que l'on peut sauvegarder dans chaque case mémoire.

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

<u>Exercice1</u> : (de la figure 12)

Notre mémoire a une capacité de 8 mots de 16 bits chacun. On exprime également cette capacité en nombre d'octets ou de bits.

Solution :

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

Notre mémoire a donc une capacité de (8*2 octets) **16 octets** ou de (8*16 bits) **128 bits.**

Exercice 2 :

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 ?

Solution :

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

On a **Tailla de bus d'adresse = Nombre de lignes d'adresses** donc **Nombre de mots =** $2^{\text{nombre de lignes d'adresses}}$

Et aussi **Tailla de bus de données = Taille du mot**

Alors Capacité=2^{16} mots de 8 bits ➔ Capacité = $2^{16} * 2^{3} =$ **2^{19} bits** = **2^{16} octets** = **2^{13} K octets**

Remarque :

- Un mot de **n** bits peu avoir **2^{n}** combinaisons différentes.
- La capacité est exprimée aussi en octet (ou byte) ou en mot de 8, 16 ou 32 bits. On utilise des puissances de deux, avec les unités suivantes :

Kilo 1K = 2^{10} Méga 1M = 2^{20} Giga 1G = = 2^{30} Téra 1T = 2^{40} Péta 1P = 2^{50}…

2.7.2. Mémoire vive

Une mémoire vive ou **RAM** *(Random Acces Memory*, la traduction est *Mémoire à accès aléatoire*). Son contenu est modifiable car elle sert au stockage temporaire des données et des programmes nécessaires fonctionnement du matériel. Elle doit avoir un temps de cycle très court pour ne pas ralentir le microprocesseur. Les mémoires vives sont en général volatiles car elles perdent leurs informations en cas de coupure d'alimentation.

Il existe deux grands types de mémoires RAM :

a. Les mémoires statiques

- Dans la mémoire vive statique ou SRAM (Static Random Access Memory), la cellule de base est constituée par une **bascule** de transistors (1bit = 4 transistors = 2 portes NOR).
- Elle ne nécessite quasiment pas de rafraichissement.
- Le terme statique, fait référence au fonctionnement interne de la bascule.
- Dans la mesure où ce rafraichissement à un coût en temps, cela explique pourquoi ce type de mémoire est très rapide, entre 6 et 15 ns, mais assez chère.
- Elle est plus coûteuse qu'une DRAM et utilisée essentiellement pour des mémoires de faibles capacités comme dans la mémoire cache pour les microprocesseurs.
- Elle est un type de mémoire informatique spéciale utilisée dans certaines applications de recherche à très haute vitesse. Elle est aussi connue sous le nom de mémoire adressable.

b. Les mémoires dynamiques

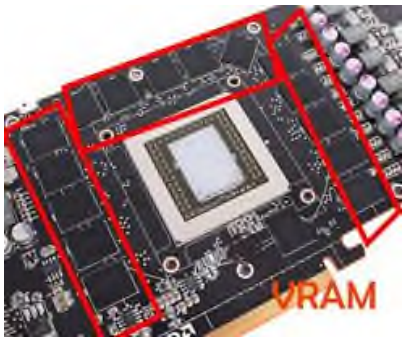
- Dans la mémoire vive dynamique ou DRAM (Dynamic Random Access Memory), la cellule de base est constituée par un **condensateur** et un transistor (1 bit = 1 transistor + 1 condensateur) et le condensateur est utilisé pour stocker l'information.

- Mémoire électronique à réalisation très simple mais le problème c'est que les condensateurs ont le défaut de se décharger (perdre lentement sa charge) et ils doivent être rechargés fréquemment (rafraichissement).
- Durant ces temps de rechargement, la mémoire ne peut être ni lue, ni écrite, ralentissant donc son fonctionnement (d'où le terme de Dynamique).
- Peu coûteuse elle est principalement utilisée pour la mémoire centrale de l'ordinateur.

Il y a aussi :

- **SDRAM** (Synchrone DRAM) : est une mémoire dynamique DRAM qui fonctionne à la vitesse du bus mémoire, elle est donc synchrone avec le bus (processeur) (lien image : https://pc4you.pro/composants-memoire-ram/724-sdram-pc100-64mb-hyundai-barrette-memoire-ram-0000000000000.html).

- **DDR SDRAM** (**D**ouble **D**ata **R**ate SDRAM) : est une SDRAM à double taux de transfert pouvant expédier et recevoir des données deux fois par cycle d'horloge au lieu d'une seule fois.
- **VRAM** (**V**ideo RAM) **:** elle a 2 ports pour pouvoir être accédée simultanément en lecture et en écriture (lien image : https://forums.tomshardware.com/threads/installed-a-kraken-g10-on-my-gtx-980-do-i-need-to-get-vrm-heatsinks.2780287/).



- **DIMM** (Dual In-line Memory Module) : groupe de puces RAM fonctionnant en 64 bits et généralement montées sur un circuit imprimé de forme rectangulaire, appelé barrette, que l'on installe sur la carte mère d'un ordinateur.
- **SIMM** (Single In-line Memory Module): idem à DIMM mais en 32 bits.
- **Mémoire flash** : est une mémoire RAM basée sur une technologie EEPROM. Le temps d'écriture est similaire à celui d'un disque dur (ex. mémoire d'appareils photos, téléphone, USB (flash disk), MemoryStick, ...). (lien image : https://www.macfix.fr/recuperation-donnees/r%C3%A9cup%C3%A9ration-de-donn%C3%A9es-cl%C3%A9-usb-ou-m%C3%A9moire-flash-detail)

Remarque :

Les performances des mémoires s'améliorent régulièrement. Le secteur d'activité est très innovant, le lecteur retiendra que les mémoires les plus rapides sont les plus chères et que pour les comparer en ce domaine, il faut utiliser un indicateur qui se nomme le cycle mémoire.

2.7.3. Mémoire morte

Les mémoires mortes ou mémoires à lecture seule (ROM : Read Only Memory) sont utilisées pour stocker des informations permanentes (programmes systèmes, microprogrammation). Ces mémoires, contrairement aux RAM, ne peuvent être que lue

(l'exécution des programmes) et les conservent en permanence même hors alimentation électrique (c.à.d. non volatile). Suivant le type de ROM, la méthode de programmation changera. Il existe donc plusieurs types de ROM :

a. **ROM :** information stockée au moment de la conception du circuit.
b. **PROM :** (Programmable ROM) mémoire programmable une seule fois et elle est réalisée à partir d'un programmateur spécifique.
c. **EPROM *ou* *UV-EPROM :*** L'EPROM (Erasable Programmable ROM) mémoire (re)programmable et effaçable par ultraviolet (lien image : https://www.reichelt.com/ch/fr/eprom-uv-c-mos-c-dil-42-2-mx8-1-mx16-100-ns-27c160-100-p40037.html).

d. **EEPROM :** (Electically EPROM) mémoire (re)programmable et effaçable électriquement.
e. **FLASH EPROM :** La mémoire Flash est programmable et effaçable électriquement comme les EEPROM. Exemple : appareil photo numérique - lecteur MP3 (lien image : http://www.industrialautomation-products.com/sale-11136421-6es7952-1as00-0aa0-siemens-memory-card-ram-s7-400-flash-memory-card.html).

2.7.4. Structure physique d'une mémoire centrale

Concernant la **structure physique d'une mémoire centrale** (fig. 13), elle contient les composants suivants :

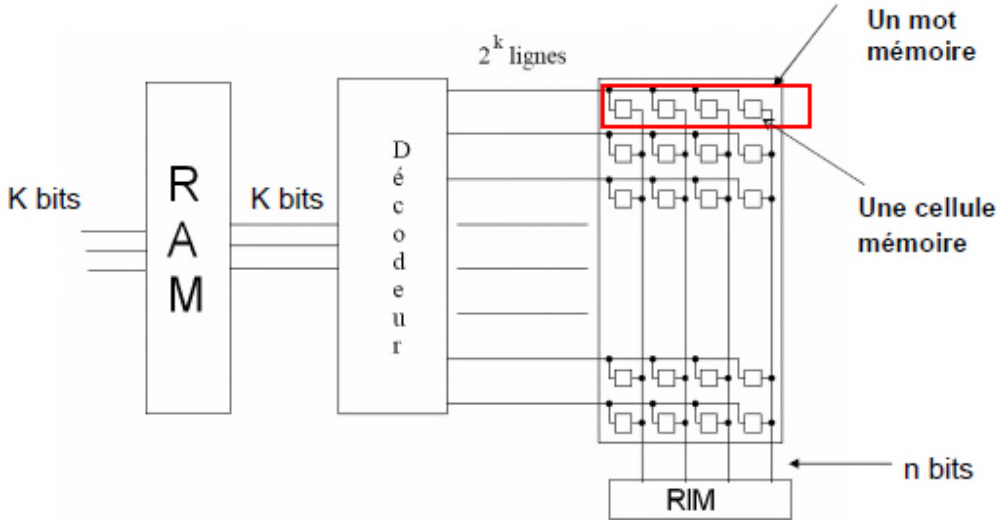


Figure 13 : La structure physique d'une mémoire centrale.

•**RAM** (Registre d'adresse Mémoire) : ce registre stock l'adresse du mot à lire ou à écrire.

•**RIM** (Registre d'information mémoire) : stock l'information lu à partir de la mémoire ou l'information à écrire dans la mémoire.

•**Décodeur** : permet de sélectionner un mot mémoire.

•**R/W** : commande de lecture/écriture, cette commande permet de lire ou d'écrire dans la mémoire (si R/W=1 alors lecture sinon écriture)

•**Bus d'adresses** de taille **k** bits

•**Bus de données** de taille **n** bits

Pour le principe de **sélection d'un mot mémoire**, lorsqu'une adresse est chargée dans le registre RAM, le décodeur va recevoir la même information que celle du RAM. A la sortie du décodeur nous allons avoir une seule sortie qui est active, donc cette sortie va nous permettre de sélectionner un seul mot mémoire.

2.7.5. Lecture et écriture de l'information

Les seules opérations possibles sur la mémoire sont :

- **Ecriture dans un emplacement** *(récupérer ou restituer)* : le processeur donne une valeur et une adresse et la mémoire range la valeur à l'emplacement indiqué par l'adresse.
- **Lecture d'un emplacement** *(enregistrer ou modifier)* : le processeur demande à la mémoire la valeur contenue à l'emplacement dont il indique l'adresse. Le contenu de l'emplacement lu reste inchangé.

- **Algorithme de lecture**

Pour lire une information en mémoire centrale, Il faut effectuer les opérations suivantes :

1. L'unité centrale commence par charger dans le registre RAM l'adresse mémoire du mot à lire.
2. Elle lance la commande de lecture à destination de la mémoire (R/W=1)
3. L'information est disponible dans le registre RIM au bout d'un certain temps (temps d'accès) où l'unité centrale peut alors le récupérer.

- **Algorithme d'écriture**

Pour écrire une information en MC il faut effectuer les opérations suivantes :

1. L'unité centrale commence par placer dans le RAM l'adresse du mot où se fera l'écriture.
2. Elle place dans le RIM l'information à écrire.
3. L'unité centrale lance la commande d'écriture pour transférer le contenu du RIM dans la mémoire centrale.

Remarque :

- Il y a écriture lorsqu'on enregistre des informations en mémoire et lecture lorsqu'on récupère des informations précédemment enregistrées.
- Dans l'étape N°1, puisque les 2 opérations (lecture et écriture) sont indépendantes et qu'elles utilisent des bus différents, alors elles peuvent être effectuées en parallèle (gain de temps).

2.8. Mémoire cache

La mémoire cache ou ***antémémoire*** (fig. 14) est une mémoire très rapide d'accès pour le microprocesseur. On la réalise à partir de cellule SRAM de taille réduite (à cause du coût) car SRAM est 8 à 16 fois plus rapide que DRAM mais 4 à 8 fois plus volumineuse. Elle agit comme un tampon entre le processeur et la mémoire principale. Sa capacité mémoire est donc très inférieur à celle de la mémoire principale.

Elle est utilisée pour maintenir les parties de données et programmes qui sont le plus fréquemment utilisés par les CPU. Les parties de données et les programmes sont transférés du disque vers la mémoire cache par le système d'exploitation. Les données stockées dans une mémoire cache pourraient être les résultats d'un calcul plus tôt, ou les doublons de données stockées ailleurs.

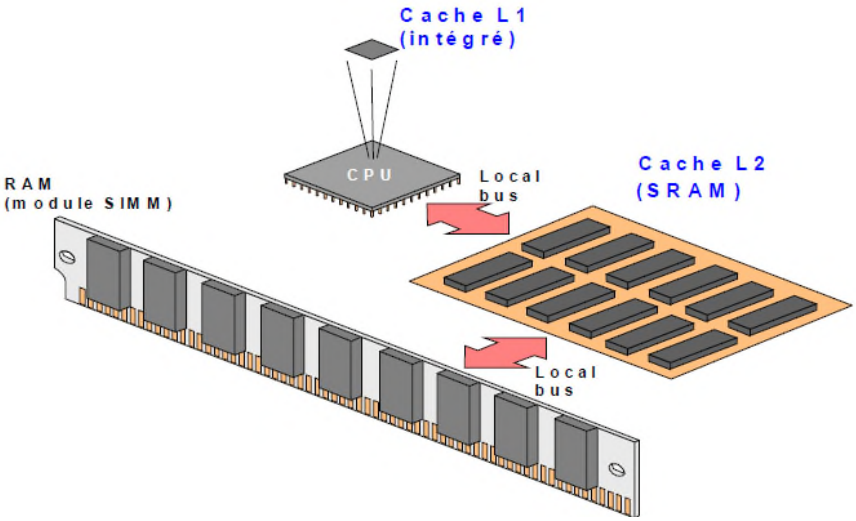


Figure 14 : Exemple de mémoire cache à deux niveaux.

Au départ cette mémoire était intégrée en dehors du microprocesseur mais elle fait maintenant partie intégrante du microprocesseur et se décline même sur plusieurs niveaux.

2.8.1. Principe

Le principe de cache est très simple (fig. 15) : le microprocesseur n'a pas conscience de sa présence et lui envoie toutes ses requêtes comme s'il agissait de la mémoire principale :

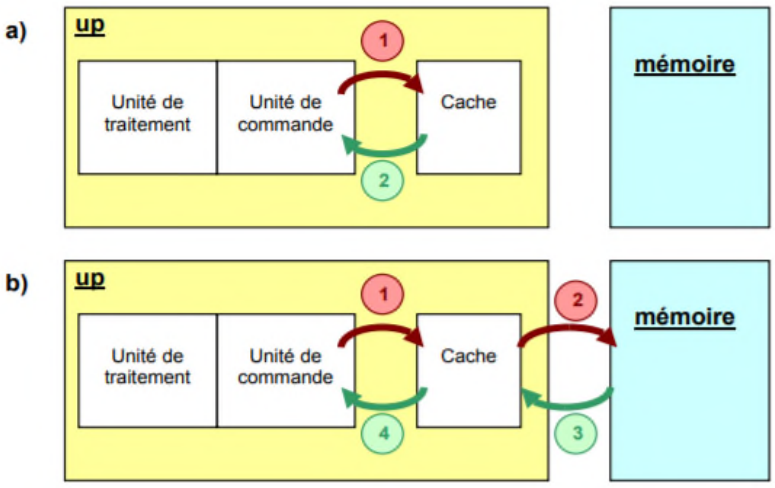


Figure 15 : Le principe de la mémoire cache.

- Soit la donnée ou l'instruction requise est présente dans le cache et elle est alors envoyée directement au microprocesseur. On parle de **succès de cache (a) (**en anglais **Hit).**
- Soit la donnée ou l'instruction n'est pas dans le cache et le contrôleur de cache envoie alors une requête à la mémoire principale. Une fois l'information récupérée, il la renvoie au microprocesseur tout en la stockant dans le cache. On parle de **défaut de cache (b) (**en anglais **Miss).**

Remarque :

Le cache mémoire n'apporte un gain de performance que dans le premier cas. Sa performance est donc entièrement liée à son taux de succès. Il est courant de rencontrer des taux de succès moyens de l'ordre de 80 à 90%.

2.8.2. Fonctionnement

Actuellement le cache des micro-processeurs récents sur le marché est composé de deux niveaux de mémoires de type SRAM la plus semblable à celle des registres : le *cache de niveau un* est noté **L1** et le *cache de niveau deux* est noté **L2**.

Sachant que la mémoire ***cache de niveau L1*** est dans le processeur (cache interne/ on-chip), *unifié :* contient instructions et données (ex. : Intel 486), mais actuellement au moins *2 caches* : 1 cache de données et 1 cache d'instructions (ex. Pentium : 2 caches L1 de 8 ko, Pentium III : 2 caches L1 de 16 ko, actuellement cache L1 : 128 ko). Avantage des caches séparés : les opérations mémoires sur des instructions et données indépendantes peuvent être simultanées.

Pour la mémoire ***cache de niveau L2***, elle est à côté du processeur (cache externe / out-chip), généralement de 256 ko. Il y a aussi le ***cache de niveau trois L3*** à l'extérieur comme **L2**, rarement utilisé (ex : Intel Core i7).

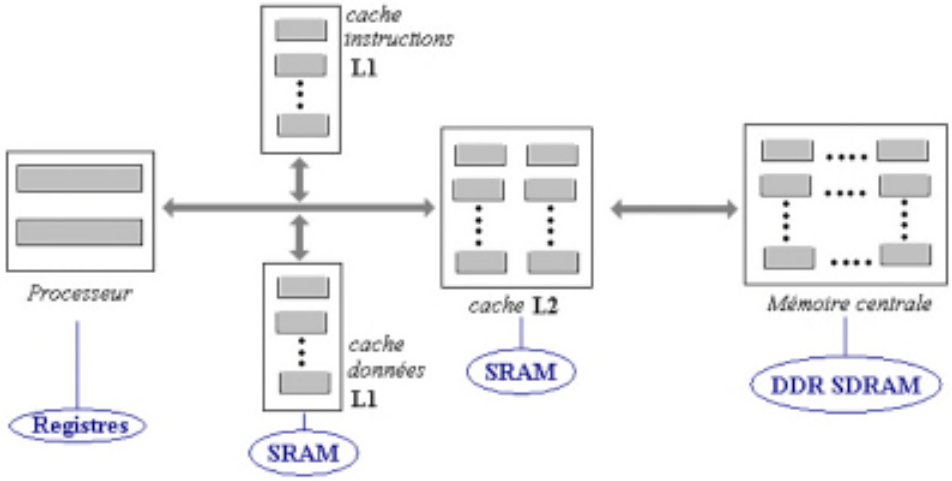


Figure 16 : Le fonctionnement de la mémoire cache.

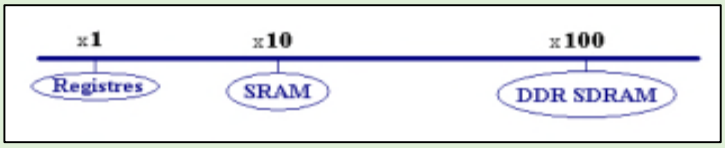
Le fonctionnement est le suivant (fig. 16) :

- **Si** un étage du *processeur cherche une donnée*, elle va être d'abord **recherchée** dans le cache de donnée **L1** et **rapatriée** dans un *registre* adéquat, **sinon si** la donnée **n'est pas présente** dans le **cache L1**, elle sera **recherchée** dans le **cache L2**.

- **Si** la *donnée* est **présente** dans **L2**, elle est alors **rapatriée** dans un *registre* adéquat et **recopiée** dans le **bloc de donnée** du **cache L1**. Il en va de même lorsque la *donnée* **n'est pas présente** dans le **cache L2**, elle est alors **rapatriée** depuis la **mémoire centrale** dans le *registre* adéquat et **recopiée** dans le **cache L2.**

Remarque :

Le facteur d'échelle (d'un coefficient de multiplication des temps d'accès à une information) relatif entre les différents composants mémoires du processeur et de la mémoire centrale.



Les registres, mémoires les plus rapides se voient affecter la valeur de référence 1. L'accès par le processeur à une information située dans la DDR SDRAM de la mémoire centrale est 100 fois plus lente qu'un accès à une information contenue dans un registre.

2.8.3. Gestion de la mémoire cache

a. Définitions

- **Ligne :** est le plus petit élément de données qui peut être transféré entre la mémoire cache et la mémoire de niveau supérieur.
- **Mot :** est le plus petit élément de données qui peut être transféré entre le processeur et la mémoire.

b. Localité

Le principe de localité affirme que les informations auxquelles va accéder le processeur ont une forte probabilité d'être localisées dans une ***fenêtre spatiale*** et une ***fenêtre temporelle***.

- **Localité spatiale :** indique que l'accès à une instruction située à une adresse X va probablement être suivie d'un accès à une zone toute proche de X
  Exemple : tableaux, structures.
  La localité spatiale, suggère de copier des blocs de mots dans le cache plutôt que des mots isolés.
- **Localité temporelle** : indique que l'accès à une zone mémoire à un instant donné a de fortes chances de se reproduire dans la suite du programme.
  Exemple : structures itératives.
  La localité temporelle suggère de conserver pendant quelque temps dans le cache les informations auxquelles on vient d'accéder.

c. Nombre de cache et localisation

Actuellement, la norme est à l'utilisation de multiple caches, organisés en niveau (level).

- Un cache peut être situé sur la même puce que le processeur (**on-chip/internal cache**).
- Ou n'être accessible que via un bus externe au processeur (**external cache**).

L'utilisation d'un cache interne permet d'augmenter les performances de laisser le bus externe disponible.

Une organisation typique est : un cache interne (de niveau 1) et un cache externe (de niveau 2). Le cache de niveau 2 doit être de 10 à 100 fois plus grand que le/les caches de niveau 1 pour être intéressant.

d. Correspondance cache et mémoire (le mapping)

La taille du cache est beaucoup plus petite que la taille de la mémoire. Il faut définir une stratégie de copie des blocs de données dans le cache. Cette méthode s'appelle le mapping. Trois stratégies sont possibles :

- **Correspondance directe (direct mapped cache) :** le bloc **n** de la mémoire principale peut se retrouver seulement dans le bloc **m = (n modulo sb)** de la mémoire cache, sachant que **sb** est la taille en nombre de blocs de la mémoire cache (fig. 17).

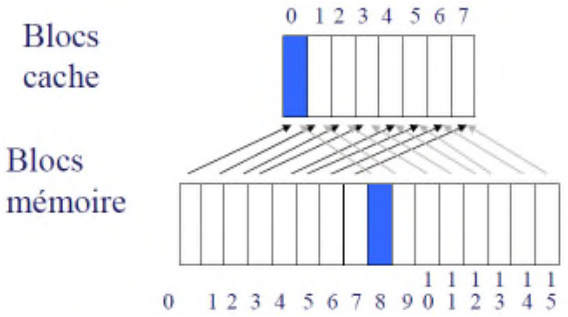


Figure 17 : La correspondance cache directe.

- **Correspondance totalement associatif (fully associative cache)** : chaque bloc mémoire peut être placé dans n'importe quel bloc du cache (fig. 18).

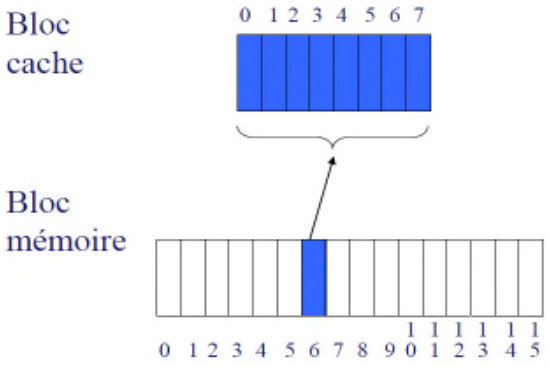


Figure 18 : Correspondance cache totalement associatif.

- **Correspondance associative par ensemble (set associative cache)** : séparation de la mémoire cache en groupes de blocs et associativité complète dans un groupe, c.à.d. le bloc **n** de la mémoire principale peut se retrouver dans n'importe quel bloc du groupe **g = (n modulo sg**) de la mémoire cache, sachant que **sg** est le nombre total de groupes de blocs dans la mémoire cache (fig. 19).

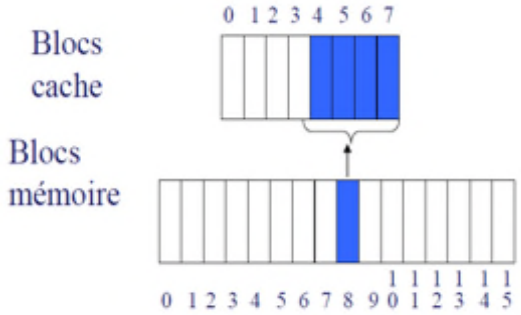


Figure 19 : La correspondance cache associative par ensemble.

Remarque :

La différence entre les correspondances cache et mémoire

Fonction de correspondance	*Avantages*	*Désavantages*
Placement direct	- Simple, - Peu - couteux - Bon choix pour les caches larges	- Très restrictive
Cache totalement associatif	- Meilleur taux de succès - Bon choix pour les caches petits	- Très couteux - Demande matériel - Demande une étiquette plus large
Cache associatif par ensemble de bloc	- Compromis - Préféré souvent	- Demande une étiquette large

De nos jours, la grande majorité des caches sont à correspondance directe ou a correspondance associative par ensemble de 2 ou 4 blocs.

Exercice : *Pentium 4 Prescott* ayant les caractéristiques de mémoire cache suivantes :

- L1 (données) : 16 Kbits ; lignes de 64bits ; associative par ensembles de 8
- L2 : 1 Mbits ; lignes de 128bits ; associative par ensembles de 8

1- Combien y-a-t-il de lignes dans cette mémoire cache ?

2- Combien y-a-t-il de blocs associatifs dans cette mémoire cache ?

Solution :

1- Combien y-a-t-il de lignes dans cette mémoire cache ?

Nombre de lignes L1 = Taille cache / Taille de la ligne = 16 Kbits / 64 bits

$= 2^4 * 2^{10} / 2^6 =$ **2^8 Lignes**

Nombre de lignes L2 = Taille cache / Taille de la ligne = 1 Mbits / 128 bits

$= 2^{20} / 2^7 =$ **2^{13} Lignes**

2- Combien y-a-t-il de blocs associatifs dans cette mémoire cache ?

Nombre de blocs L1= Nombre de lignes/ Nombre de lignes par bloc = $2^8 / 8 =$ **2^5 blocs**

Nombre de blocs L2= Nombre de lignes/ Nombre de lignes par bloc = $2^{13} / 8 =$ **2^{10} blocs**

e. Accès à un bloc du cache

Les adresses mémoires peuvent être construites en fonction de la correspondance entre mémoire principale et cache (fig. 20). Dans ce cas, l'adresse mémoire d'un mot contient des informations sur sa présence dans un bloc et sa présence éventuelle dans le cache. Elle se décompose en deux parties :

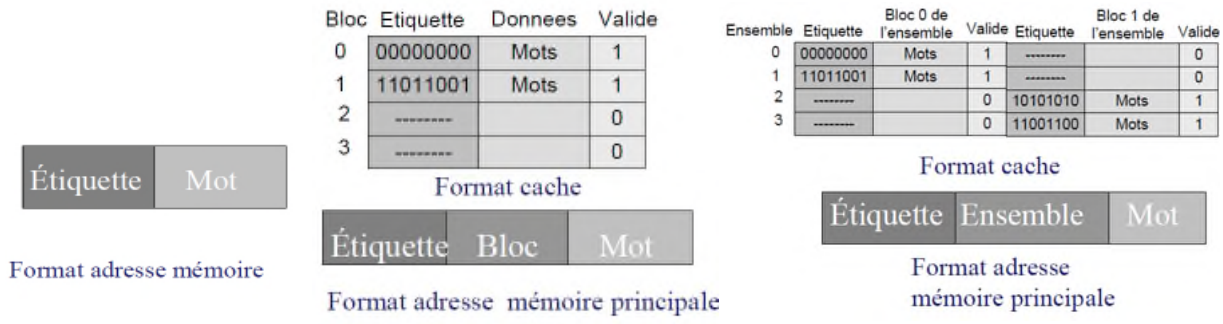
- Un **numéro de bloc**, qui se décompose en

  - un *index*, correspondant à l'emplacement de e bloc dans le cache

  - une *étiquette* permettant d'identifier le bloc mémoire correspondant au bloc placé dans le cache

- Un **déplacement** dans le bloc (le numéro du mot dans le bloc).

Ainsi, une table d'étiquette est maintenue, ce qui donne pour chaque bloc du cache l'étiquette du bloc mémoire placé dans ce bloc, ou le fait qu'aucun bloc mémoire n'a été copié dans ce bloc.



Cache totalement associatif ***Placement direct*** ***Cache associatif par ensemble***

Figure 20 : Format d'adresse mémoire cache.

f. Algorithme de remplacement

Si le cache est plein et que le processeur a besoin d'un bloc qui n'est pas dans le cache, il faut remplacer un des blocs du cache. Diverses stratégies sont employées, principalement :

- Choisir un bloc candidat de manière aléatoire (Random)
- Choisir le plus ancien bloc du cache (FIFO pour First In First Out)
- Choisir le bloc le moins récemment utilisé (LRU pour Least Recently Used)
- Choisir le bloc le moins fréquemment utilisé (LFU pour Least Frequently Used)

Remarque :

- Dans un cache à accès direct le problème ne se pose évidemment pas. En revanche dans les caches associatifs, ou associatifs par ensemble, une stratégie doit être mise en œuvre.
- Les stratégies concernant l'utilisation (LFU, LRU) sont les plus efficaces, vient ensuite la stratégie aléatoire.
- Les stratégies aléatoires et FIFO sont plus faciles à implanter.

g. Interaction avec la mémoire centrale (lecture / écriture)

La lecture est l'opération la plus courante dans les caches. Toutes les instructions sont lues et la plupart d'entre elles ne provoquent pas d'écriture.

Les ***politiques de lecture*** lors d'un *échec* dans le cache sont :

- **Lecture immédiate (**en anglais **Read Through)** : la lecture se fait directement de la mémoire centrale vers le CPU.
- **Lecture non immédiate (**en anglais **No Read Through)** : la lecture se fait de la mémoire centrale vers le cache puis du cache vers le CPU.

Deux ***politiques d'écriture*** sont employées pour traiter le cas d'un *succès* dans le cache :

- **Ecriture immédiate ou simultanée (**en anglais **Write Through)** : l'information est écrite dans le cache et dans la mémoire centrale.
- **Ecriture remplacement ou réécriture (**en anglais **Write Back)** : l'information est écrite uniquement dans le cache. Elle est écrite dans la mémoire centrale seulement lors d'un remplacement. Un bit, appelé **dirty bit**, indique pour chaque voie s'il est nécessaire de mettre à jour la voie en mémoire centrale.

Il y a également deux politiques lors d'une écriture sur une information non présente dans le cache :

- **Ecriture allouée (**en anglais **Write Allocate)** : l'information est d'abord chargée dans le cache puis modifiée.
- **Ecriture non allouée (**en anglais **No Write Allocate)** : l'information est directement modifiée dans la mémoire centrale et n'est pas chargée dans le cache.

Résume des politiques de lecture et d'écriture :

<table>
<tr><th colspan="2">En cas d'échec d'écriture</th><th>Ecriture dans le cache</th><th></th></tr>
<tr><td colspan="2"></td><td>Oui</td><td>Non</td></tr>
<tr><td rowspan="3">Ecriture dans la mémoire</td><td>Oui</td><td rowspan="2">Ecriture immédiate + Ecriture allouée</td><td>Ecriture immédiate + Ecriture non allouée</td></tr>
<tr><td></td><td>Réécriture + Ecriture non allouée</td></tr>
<tr><td>Non</td><td>Réécriture + Ecriture allouée</td><td></td></tr>
</table>

h. Performance

On peut évaluer la performance d'une mémoire utilisant un cache par le calcul du temps d'accès mémoire moyen :

Temps D'accès Mémoire Moyen = Temps D'accès Succès + Taux D'échec * Pénalité D'échec

Temps D'accès Succès = Temps D'accès A Une Donnée Résidant Dans Le Cache

Taux D'échec = Nombre De Défauts De Cache / Nombre D'accès Cache Ou = 1 - Taux De Succès

Taux De Succès = Nombre De Succès / Nombre D'accès Cache

Sachant que : **Temps D'accès Succès << Pénalité D'échec**

Exercice :

A partir des performances du tableau ci-dessous, calculer le temps d'exécution moyen d'une instruction pour chaque niveau sachant que la durée d'un cycle horloge est **T.**

Niveau	Temps d'accès succès (ns)	Taux de succès (ns)	Pénalité d'échec (Cycles)	Taille
Cache L1	3	80%	5	128 Ko
Cache L2	5	90%	10	512 Ko

Solution :

On a le **Temps d'accès mémoire moyen = temps d'accès succès + taux d'échec * pénalité d'échec**

Et **taux d'échec = 1 - taux de succès**

Donc **temps d'accès mémoire moyen Cache L1** = 3 + (1-80%)*5= **4T**

Et **temps d'accès mémoire moyen Cache L2 =** 5 + (1-90%)*10**= 6T**

2.8.4. Avantages et inconvénients

a. Avantages de la mémoire cache

- Elle est très rapide d'accès plus que la mémoire principale.
- Elle consomme moins de temps d'accès par rapport à .la mémoire.
- Elle stocke le programme qui peut être exécuté dans un temps court.
- Elle stocke les données pour une utilisation temporaire.

b. Inconvénients de la mémoire cache

- Elle a une capacité limitée.
- Elle est très coûteuse.

2.9. Hiérarchie de mémoires

Les différents éléments de la mémoire d'un ordinateur sont ordonnés en fonction des critères : temps d'accès, capacité et coût par bit (fig. 21).

1- Les éléments de mémoire situés dans l'unité centrale de traitement (CPU) sont les **registres** qui sont caractérisés par une grande vitesse et servent principalement au stockage d'opérandes et des résultats intermédiaires.
2- La **mémoire cache (**ou **l'antémémoire) :** est une mémoire rapide de faible capacité (par rapport à la mémoire centrale). Utilisée comme mémoire tampon entre le CPU et la mémoire centrale. Cette mémoire permet au CPU de faire moins d'accès à la mémoire centrale et ainsi de gagner du temps.
3- La **mémoire centrale :** est l'organe principal de rangement des informations utilisées par le CPU. Pour exécuter un programme, il faut le charger (instructions + données)

en mémoire centrale. Cette mémoire est une mémoire à semi-conducteur, mais son temps d'accès est beaucoup plus grand que celui des registres et du cache.

4- La **mémoire d'appui :** sert de mémoire intermédiaire entre la mémoire centrale et les mémoires auxiliaires. Elle est présente dans les ordinateurs les plus évolués et permet d'augmenter la vitesse d'échange des informations entre ces deux niveaux.

5- La **mémoire de masse (**ou **mémoire auxiliaire) :** est une mémoire périphérique de grande capacité et de coût relativement faible, utilisée pour le stockage permanent des informations. Elle est utilisée pour le stockage, la sauvegarde ou l'archivage à long terme des informations. Elle utilise pour celà des supports magnétiques (disques, cartouches, bandes), magnéto-optiques(disques) ou optiques (disques optiques).

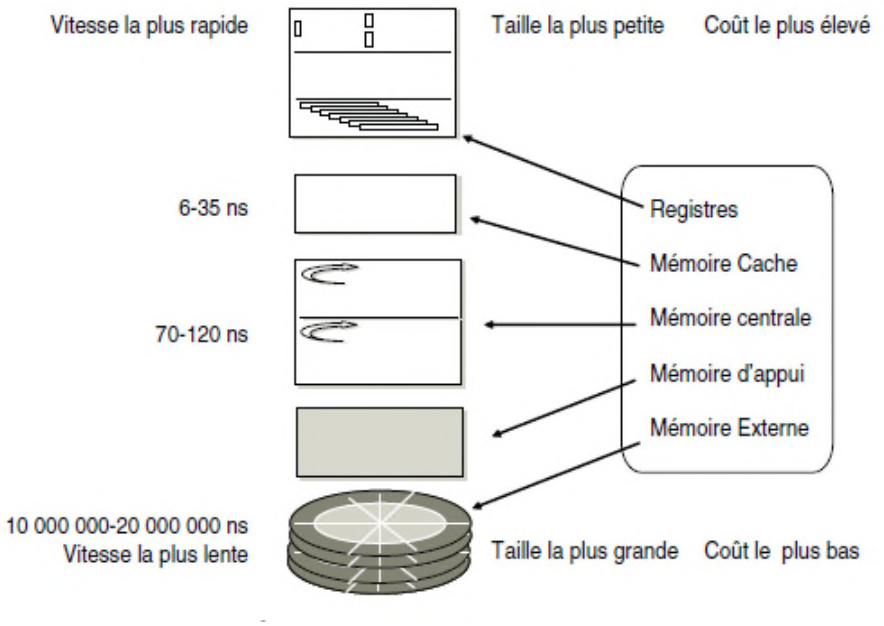


Figure 21: La hiérarchie de mémoires.

Exercice : Classez les mémoires suivantes par taille, par rapidité : CD-ROM, Registre d'Instruction, Disques durs, ROM, Cache L1, USB, Cache L2.
Solution :
Par taille : RI < L1 < L2 < ROM < CD <USB < DD.
Par vitesse : RI > L1 > L2 > ROM > DD > USB > CD.

2.10. Conclusion

Dans ce deuxième chapitre, nous avons expliqué le fonctionnement des principaux composants d'une architecture d'ordinateur en général. Nous avons parlé en première partie de la carte mère, de l'Unité Arithmétique et Logique UAL, des registres et des bus. Ensuite, nous avons parlé des Mémoires : *Interne* utilisée pour le rangement des informations utilisées par le CPU et *Cache* utilisée comme mémoire tampon entre le CPU et la mémoire centrale, ainsi que la hiérarchie de mémoires. Dans le prochain chapitre, nous expliquerons les notions des instructions d'un ordinateur.

Chapitre III :

Notions sur les instructions d'un ordinateur

3.1. Introduction

Le processeur exécute une par une les instructions stockées sous forme numérique en mémoire, écrites par le programmeur, en utilisant ses éléments internes : unité de contrôle et de commande (UCC) et Unité Arithmétique et Logique (UAL). Nous découvrirons dans ce chapitre les composants d'une instruction machine et de quelle manière les instructions sont exécutées mais avant nous parlerons des langages de programmation (Haut niveau, assembleur et machine), le principe de compilation et d'assemblage de base pour le traduire une instruction en langage de haut niveau en assembleur puis la convertir en code machine.

L'**architecture d'un ordinateur** en général est fondée sur le **modèle de Von Neumann** : la machine est construite autour d'un processeur, véritable tour de contrôle interne, d'une mémoire stockant les données et le programme et d'un dispositif d'entrées/sorties nécessaire pour l'échange avec l'extérieur. Le déroulement d'un programme au sein de l'ordinateur est le suivant:

1. L'UCC extrait une instruction de la mémoire
2. Analyse l'instruction
3. Recherche dans la mémoire les données concernées par l'instruction
4. Déclenche l'opération adéquate sur l'UAL ou l'E/S
5. Range au besoin le résultat dans la mémoire

3.2. Langages de programmation

La programmation est donc l'activité qui consiste à traduire par un programme un algorithme dans un langage assimilable par l'ordinateur.

Cette activité de programmation peut s'effectuer à différents niveaux : la programmation de bas niveau en **langage machine**, la programmation de bas niveau en **langage d'assemblage**, la programmation de haut niveau à l'aide d'un **langage de haut niveau** ou langage évolué.

3.2.1. Langage machine

C'est le seul langage exécutable directement par le microprocesseur. Ce langage est difficile à maîtriser puisque chaque instruction est codée par une séquence propre de bits (Binary Digit). Afin de faciliter la tâche du programmeur, on a créé différents langages plus ou moins évolués.

Une **instruction machine** (fig. 22) est une chaîne binaire composée essentiellement de deux parties : le **code opération** et les **opérandes**. Ces opérandes sont soit des mots mémoires, soit des registres du processeur ou encore des valeurs immédiates.

Chaque instruction est par ailleurs repérée par une adresse qui mémorise la position de l'instruction dans le programme.

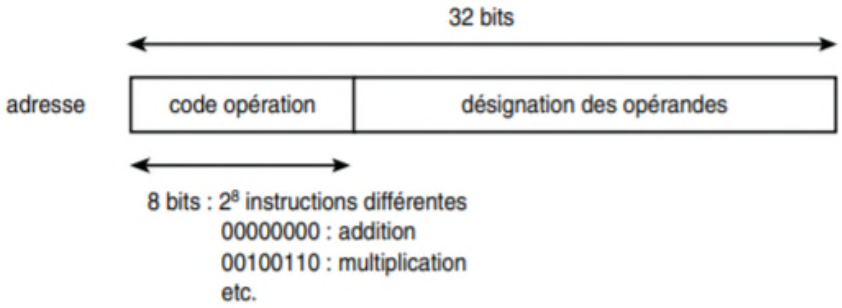


Figure 22 : L'instruction machine.

3.2.2. Langage assembleur

C'est le langage le plus proche du langage machine. Il est composé par des instructions en général assez rudimentaires que l'on appelle des **mnémoniques**.

Ce sont essentiellement des opérations de transfert de données entre les registres et l'extérieur du microprocesseur (mémoire ou périphérique), ou des opérations arithmétiques ou logiques. Chaque instruction représente un code machine différent et chaque microprocesseur peut posséder un assembleur différent.

Une **instruction du langage d'assemblage** (fig. 23) est composée de champs séparés par un ou plusieurs espaces. On identifie :

- Un champ **étiquette** : non obligatoire, qui correspond à l'adresse de l'instruction machine.
- Un champ **code opération** : qui correspond à la chaîne binaire code opération de l'instruction machine ;
- Un champ **opérandes :** pouvant effectivement comporter plusieurs opérandes séparés par des virgules qui correspondent aux registres, mots mémoires ou valeurs immédiates apparaissant dans les instructions machine.

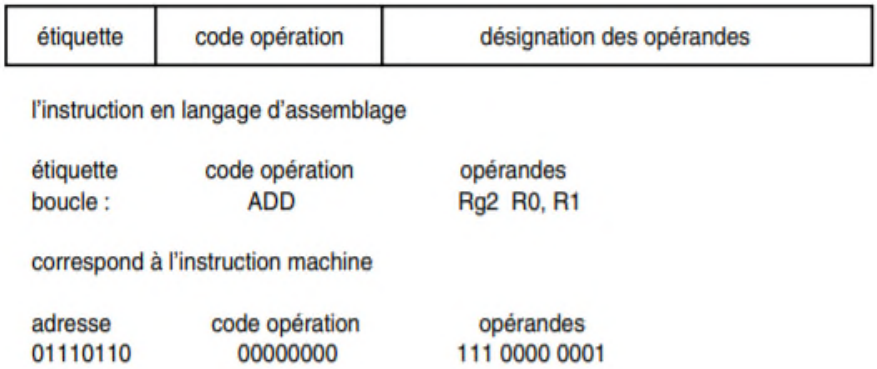


Figure 23 : L'instruction en langage d'assembleur.

3.2.3. Langage haut niveau (ou évalué)

La difficulté de mise en œuvre de ce type de langage (machine et assembleur) et leur forte dépendance avec la machine a nécessité la conception de **langages de haut niveau**, plus adaptés à l'homme, et aux applications qu'il cherchait à développer. Le langage de haut niveau est le niveau de programmation le plus utilisé aujourd'hui. C'est un niveau de programmation indépendant de la structure physique de la machine et de l'architecture du processeur qui permet l'expression d'algorithmes sous une forme plus facile à apprendre et à dominer.

Deux familles de langages importants et courants sont :

- **Le langage procédural** : l'écriture d'un programme est basée sur les notions de procédures et de fonctions, qui représentent les traitements à appliquer aux données du problème, de manière à aboutir à la solution du problème initial. **Les langages C et Pascal sont deux exemples de langages procéduraux** ;
- **Le langage objet** : l'écriture d'un programme est basée sur la notion d'objets, qui représentent les différentes entités entrant en jeu dans la résolution du problème. À chacun de ces objets sont attachées des méthodes, qui lorsqu'elles sont activées, modifient l'état des objets. **Les langages Java et Eiffel sont deux exemples de langages objets.**

Exemple de programme :

Code machine (68HC11)	Assembleur (68HC11)	Langage C
@00 C6 64 @01 B6 00 @03 1B @04 5A @05 26 03	LDAB #100 LDAA #0 ret ABA DECB BNE ret	A=0 ; for (i=1 ; i<101 ; i++) A=A+i ;

3.3. Instruction machine

Une instruction (en langage machine ou assembleur) désigne un ordre donné au processeur et qui permet à celui-ci de réaliser un traitement élémentaire. L'instruction machine (fig. 24) est une chaîne binaire de **p** bits composée principalement de deux parties :

- Le champ **code opération** composé de **m** bits :

- Il indique au processeur le type de traitement à réaliser (addition, lecture d'une case mémoire, etc.).
- Un code opération de **m** bits permet de définir $\mathbf{2^m}$ opérations différentes pour la machine.
- Le nombre d'opérations différentes autorisées pour une machine définit **le jeu d'instructions** de la machine.

➢ Le champ **opérandes** composé de **p – m** bits :

- Il permet d'indiquer la nature des données sur lesquelles l'opération désignée par le code opération doit être effectuée.
- La façon de désigner un opérande dans une instruction peut prendre différentes formes : on parle alors de **mode d'adressage** des opérandes.

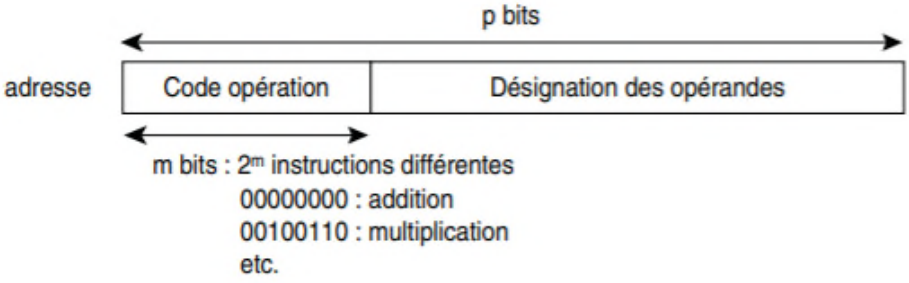


Figure 24 : Format général d'une instruction machine.

Remarque :

- Chaque architecture de la machine physique correspond à une forme symbolique du langage machine associé au processeur.
- Pour éviter d'avoir affaire au langage machine difficilement manipulable par l'homme on utilise un langage symbolique équivalent appelé langage assembleur.
- Les codes opérations d'une instruction machine sont représentés par des abréviations appelés **mnémonique** indiquant les opérations

Exemple :

- ADD	pour une opération d'addition
- SUB	pour une opération de soustraction
- MPY (ou MUL)	pour une opération de multiplication
- DIV	pour une opération de division
- LOAD (ou LD)	charger une donnée de la mémoire
- MOVE	Transfert d'une donnée
- JUMP	branchement à une adresse donnée
- STORE (ou ST)	stocker une donnée dans la mémoire
- STA	stocker une donnée dans une adresse
- STR	stocker une donnée dans une registre

Les opérandes sont aussi représentés symboliquement

Exemple :

ADD R, y ➔ Additionner la valeur contenue dans position y au contenu du registre R

3.3.1. Classification des machines par le nombre d'opérandes

Il est parfois fait référence à une machine par le **nombre de champs opérandes** contenus dans ses instructions. Il existe des **machines à 4, 3, 2, 1 ou 0 adresse (s).**

a- Machine à 4 adresses

A1 : adresse du 1er opérande ou la donnée elle-même
A2 : adresse du 2ème opérande ou la donnée elle-même
A3 : adresse où doit être rangé le résultat
A4 : adresse de l'instruction suivante

Effet : A3←(A1) + (A2) **Inst Suiv = A4**
Exemple : ADD 19,13,100,110 Effet : 100←(19) + (13) Inst Suiv = 110

b- Machine à 3 adresses

Le champ A4 n'existe pas. L'adresse de l'instruction suivante étant indiquée par le compteur ordinal (CO).

Effet : A3←(A1)+(A2) **CO ←CO+1**

Exemple : ADD 19,13,100 Effet : 100←(19) + (13)

c- Machine à 2 adresses

Les champs A4 et A3 n'existent pas. Le résultat est rangé dans le mot dont l'adresse est contenue dans le champ A2.

Effet : A2←(A1)+(A2) **CO ←CO+1**

Exemple : ADD 19,13 Effet : 19←(19) + (13)

d- Machine à 1 adresse « Machine à Accumulateur »

A1 : désigne l'emplacement du 1er opérande. Le 2ème opérande se trouve dans un registre, l'opération terminée, le résultat est rangé dans ce registre. Sur certaines machines, ce registre est appelé **accumulateur.**

Effet : ACC←(ACC)+(A1)

Exemple : ADD 19 Effet : ACC←100+(19) / Si ACC contient la valeur 100

e- Machine à 0 adresse « machine à pile »

Appelée machine à pile, ces machines n'ont pas de vrai registre, toutes les opérations effectuent un transfert entre la mémoire et une pile.

Exemple : ADD Effet : si contient val0 et val1 → pile={val0+val1}

3.3.2. Modes d'adressage des opérandes

Par **mode d'adressage**, on désigne le chemin que doit emprunter l'unité centrale (processeur) pour accéder à l'opérande. Il désigne comment le champ adresse de l'instruction est utilisé pour déterminer l'opérande.

Remarque :

- **Le mode d'adressage**, se trouve indiqué au sein de l'instruction dans le code opération ou dans un champ séparé réservé à cet effet, appelé conditions d'adressage.
- **L'adresse effective**, correspond à l'adresse finale envoyée dans le RAM après des transformations du contenu de la partie adresse de l'instruction.

a- Adressage Immédiat

L'opérande contenue dans un champ de l'instruction ne désigne pas l'adresse (emplacement dans un registre ou une mémoire) où se trouve la valeur. Il désigne la valeur elle-même.

Exemple : LOAD nbr ou LOAD #nbr **!ACC← nbr**

b- Adressage Direct

L'adresse de l'opérande est donnée dans l'instruction sous forme d'adresse mémoire.

Exemple : LOAD adr **!ACC←M[adr]**

Adresse effective : adr

c- Adressage Indirect

C'est l'emplacement où se trouve l'adresse qui est désigné et non l'adresse elle-même.

Exemple : LOAD [adr] ou LOAD @adr **!ACC←M[M[adr]]**

Adresse effective : M[adr]

Il y a aussi :

-Adressage registre : L'adresse de l'opérande est donnée dans l'instruction sous forme d'identifiant ou n° de registre.

Exemple : LOAD R1 !ACC←R1

-Adressage registre indirect : c'est l'emplacement où se trouve l'adresse qui est désigné et non l'adresse elle-même.

Exemple : LOAD (R1) !ACC←M[R1] si R1 contient une adresse mémoire

!ACC←R1 si R1 contient un n° de registre

-Adressage relatif : l'adresse réelle, c'est l'adresse indiquée dans l'instruction, ajoutée à une adresse de référence contenue dans un registre le plus souvent ou dans une case mémoire particulière.

Adresse effective = (base) + déplacement.

Exemple : LOAD 100(R1) !ACC←M[100+R1]

-Adressage indexée : on obtient l'adresse effective en ajoutant à la partie adresse de l'instruction le contenu d'un registre appelé registre d'index.

Exemple : LOAD (adr+R1) !ACC←M[adr+R1]

-Adresse auto-incrémenté et auto-décrémenté : pour parcourir une série de positions de mémoires successives (ex accès à un tableau), il est utile d'incrémenter (décrémenter) une adresse avant ou après chaque accès.

Exemple : ADD R1, (R2)+ !R1←R1 + M[R2]

!R1←R1 + d d : facteur de déplacement

Exercice 1 : Soit l'instruction suivante : A ← 3 + 5. Réécrire cette instruction en une suite d'instructions de format à deux (02) adresses, de format à une (01) adresse et format à zéro (0) adresse.

Solution :

Machine à 0 adresse (Machine à pile)	**Machine à 1 adresse** (Machine à accumulateur)	**Machine à 2 adresses** (Machine a registres)	
PUSH X Top (pile) ← X	LOAD X Acc ← X	LOAD R_i, X R_i ← X	
POP X X← Top (pile)	STORE X X← Acc	STORE R_i, X X← R_i	
ADD POP t_1 ; POP t_2 PUSH $t_1 + t_2$	ADD X Acc ← Acc + x	ADD R_i, R_j R_j ← R_i + R_j	ADD X, R_i R_i ← X + R_i
PUSH 3 PUSH 5 ADD POP A	LOAD 3 ADD 5 STORE A	LOAD R1, 3 LOAD R2, 5 ADD R1, R2 STORE R2, A	LOAD R1, 3 ADD 5, R1 STORE R1, A

Exercice 2 :

Trouvez les résultats du fragment de programme suivant pour les trois modes d'adressage que voici : Immédiat, Direct et Indirect.

ADD 10
SUB 20
MPY 30
DIV 10

Sachant que [acc]=50 ; [10] =30 ; [20] = 10 ; [30] =20.

Solution :

	Immédiat	**Direct**	**Indirect**
ADD 10	[Acc] = 60	[Acc] = 80	[Acc] = 70
SUB 20	[Acc] = 40	[Acc] = 70	[Acc] = 40
MPY 30	[Acc] = 1200	[Acc] = 1400	[Acc] = 400
DIV 10	[Acc] = 120	[Acc] = 46.66	[Acc] = 20

3.3.3. Différents types d'instructions

Pour être complet, un ensemble d'instructions doit contenir assez d'instructions dans chacune des catégories suivantes **:**

a. les instructions arithmétiques et logiques : Ces deux groupes d'instructions mettent en jeu les circuits de l'UAL.
Add, Sub, Mul, Div et ***And, Or, Not, Xor,*** etc….
Exemple : adressage immédiat
ADD 3 R1 effectue l'addition de la valeur immédiate 3 avec le contenu du registre R1 et stocke le résultat dans le registre R1.

b. Instructions pour déplacer l'information : pour charger la mémoire, sauvegarder en mémoire, effectuer des transferts de registres à la mémoire, registre à registre, de mémoire à mémoire, etc…
Load, Move, Store
Exemple : adressage direct
LOAD 3 range la valeur contenue à l'adresse 3 en mémoire centrale dans le registre ACC.
MOVE 3 R1 transfert la valeur contenue à l'adresse 3 en mémoire centrale dans le registre R1.
STORE R1 3 écrit le contenu du registre R1 à l'adresse mémoire 3.

c. Instructions de contrôle du programme (sauts et branchement) **et instructions** qui vérifient certaines **conditions d'état** (comparaison).
Exemple :
JMP 128 effectue toujours un branchement dans le code du programme à l'adresse 128.
JMPO 128 effectue ce même branchement si seulement il y a un dépassement de capacité qui est positionné dans le registre d'état de l'UAL (indicateur O).

d. les instructions d'entrées-sorties : ce sont les instructions qui permettent au processeur de lire une donnée depuis un périphérique (par exemple le clavier) ou d'écrire une donnée vers un périphérique (par exemple l'imprimante).

e. les instructions particulières permettent par exemple d'arrêter le processeur (**HALT**) ou encore de masquer/démasquer les interruptions (**DI/EI**).

3.4. Principe de compilation et d'assemblage

Pour pouvoir exécuter un programme écrit en langage d'assemblage, il faut donc traduire les instructions de celui-ci vers les instructions machines correspondantes. Cette phase de traduction est réalisée par un outil appelé l'**assembleur** (Par bus de langage et le terme assembleur désigne tout à la fois le langage d'assemblage lui-même et l'outil de traduction).

Mais en langage de haut niveau, chaque instruction correspondra à une succession d'instructions en langage assembleur. Une fois développé, le programme en langage de haut niveau n'est donc pas compréhensible par le microprocesseur. Il faut le **compiler** pour le traduire en **assembleur** puis l'**assembler** pour le convertir en **code machine** compréhensible par le microprocesseur. Ces opérations sont réalisées à partir de logiciels spécialisés appelés **compilateur et assembleur** (fig. 25).

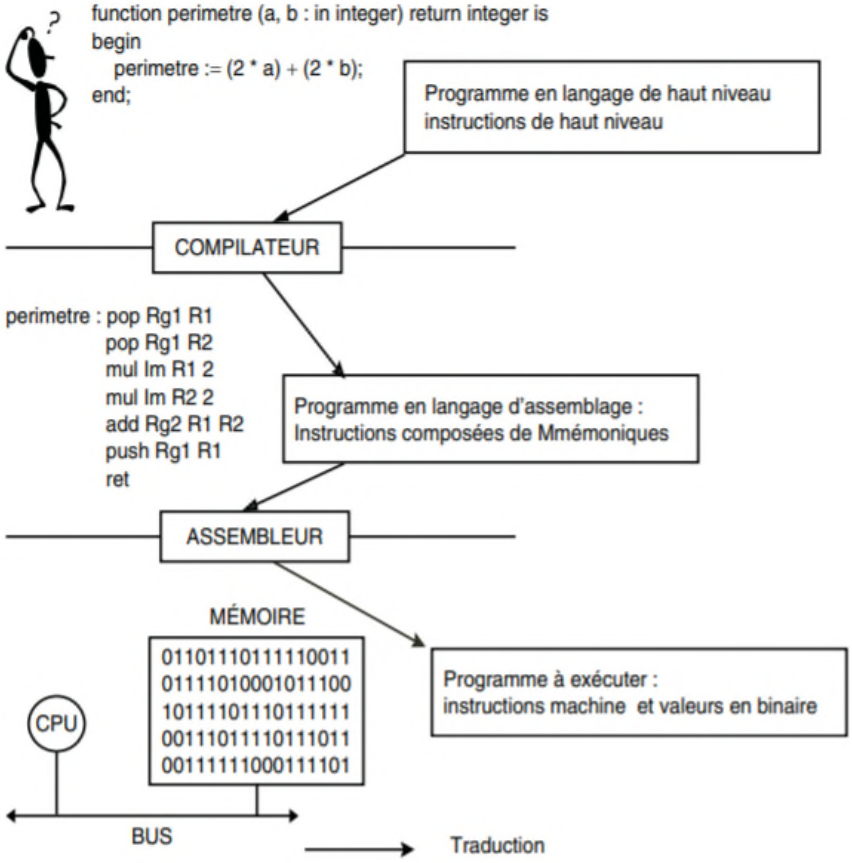


Figure 25 : Exemple de principe de compilation et d'assemblage

3.5. Unité de contrôle et de commande

Elle exécute les instructions machines et pour cela elle utilise les registres et l'UAL du microprocesseur. On y trouve **deux registres** pour la manipulation des instructions *(le **compteur** ordinal CO, le **registre d'instruction** RI)*, le **décodeur**, le **séquenceur** et **deux registres** *(le **registre d'adresses** RAD et le **registre de données** RDO)* permettant la communication avec les autres modules via le bus. Enfin, via le bus de commandes, elle commande la lecture et/ou l'écriture dans la mémoire centrale (fig. 26).

Pour le déroulement des instructions, elle est composée par :

- **Compteur de programme CO (**ou **Compteur de Programme CP) :** C'est un registre d'adresses dont le contenu est initialisé avec l'adresse de la première instruction du programme. À chaque instant il contient l'adresse de la prochaine instruction à exécuter. Ainsi en fin d'exécution de l'instruction courante le compteur ordinal pointe sur la prochaine instruction à exécuter et le programme machine peut continuer à se dérouler.
- **Registre d'instruction RI :** C'est un registre de données. Il contient l'instruction à exécuter puis elle sera décodée par le décodeur d'instruction.
- **Décodeur :** Il s'agit d'un ensemble de circuits dont la fonction est d'identifier l'instruction à exécuter qui se trouve dans le registre RI parmi toutes les opérations

possibles, puis d'indiquer au séquenceur la nature de cette instruction afin que ce dernier puisse déterminer la séquence des actions à réaliser.

- **Bloc logique de commande (ou séquenceur) :** Il organise l'exécution des instructions au rythme d'une horloge. Il élabore tous les signaux de synchronisation internes ou externes (bus de commande) du microprocesseur en fonction des divers signaux de commande provenant du décodeur d'instruction ou du registre d'état par exemple. Il s'agit d'un automate réalisé soit de façon câblée (obsolète), soit de façon micro-programmée, on parle alors de microprocesseur.
- **Registre RAD :** C'est un registre d'adresses. Il est connecté au bus d'adresses et permet la sélection d'un mot mémoire via le circuit de sélection. L'adresse contenue dans le registre RAD est placée dans le bus d'adresses et devient la valeur d'entrée du circuit de sélection de la mémoire centrale qui va à partir de cette entrée sélectionner le mot mémoire correspondant.
- **Registre RDO :** C'est un registre de données. Il permet l'échange d'informations (contenu d'un mot mémoire) entre la mémoire centrale et le processeur (registre).

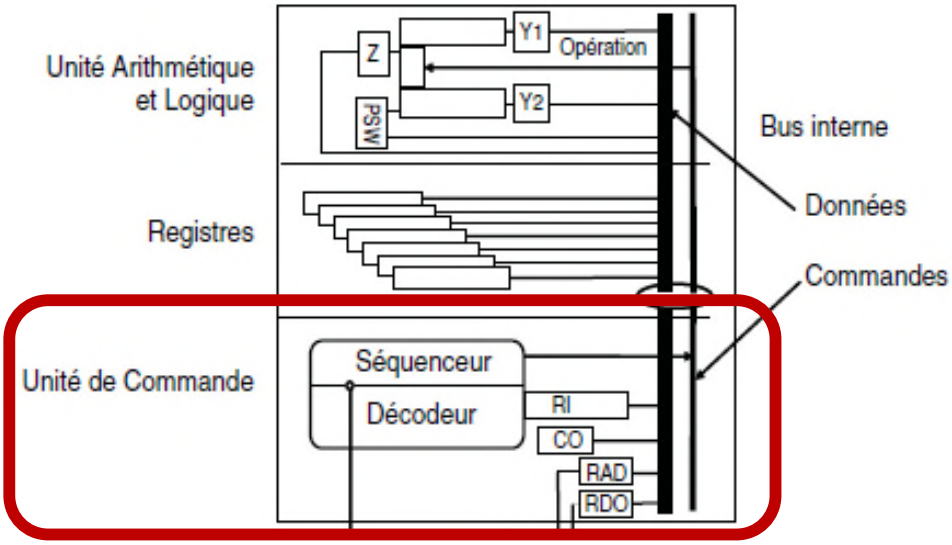


Figure 26: L'unité de commande.

Ainsi lorsque le processeur doit exécuter une instruction il :

- Place le contenu du registre CO dans le registre RAD via le bus d'adresses et le circuit de sélection.
- Déclenche une commande de lecture mémoire via le bus de commandes.
- Reçoit dans le registre de données RDO, via le bus de données, l'instruction.
- Place le contenu du registre de données RDO dans le registre instruction RI via le bus interne du microprocesseur.

Pour lire une donnée le processeur :

- Place l'adresse de la donnée dans le registre d'adresses RAD.
- Déclenche une commande de lecture mémoire.
- Reçoit la donnée dans le registre de données RDO.
- Place le contenu de RDO dans un des registres du microprocesseur (registres généraux ou registres d'entrée de l'UAL).

Remarque :

- Le ***rôle de l'unité de contrôle de commande*** est de :
- **Coordonner** le travail de toutes les autres unités (UAL, mémoire, ...)
- Assurer la **synchronisation** de l'ensemble.
- Elle assure :
- La r**echerche** (lecture) de l'instruction et des données à partir de la mémoire,
- Le **décodage** de l'instruction et l'exécution de l'instruction en cours
- La **préparation** de l'instruction suivante.
- On dit que pour transférer une information d'un module à l'autre le microprocesseur établit un ***chemin de données*** permettant l'échange d'informations.
  Par exemple pour acquérir une instruction depuis la mémoire centrale, le chemin de données est du type : CO, RAD, commande de lecture puis RDO, RI.

3.6. Phases d'exécution d'une instruction

Le déroulement d'une instruction peut être décomposé en trois phases :

1. Phase de recherche de l'instruction
2. Phase d'analyse de l'instruction et de recherche de l'opération et des opérandes
3. Phase traitement effectif de l'instruction et de préparation de l'instruction suivante

Chaque phase comporte un certain nombre d'opération élémentaires exécutées dans un ordre précis par l'unité de commande.

1er Cas : une instruction arithmétique à un seul opérande (mode direct)

Phase 1 : *Recherche de l'instruction à traiter*

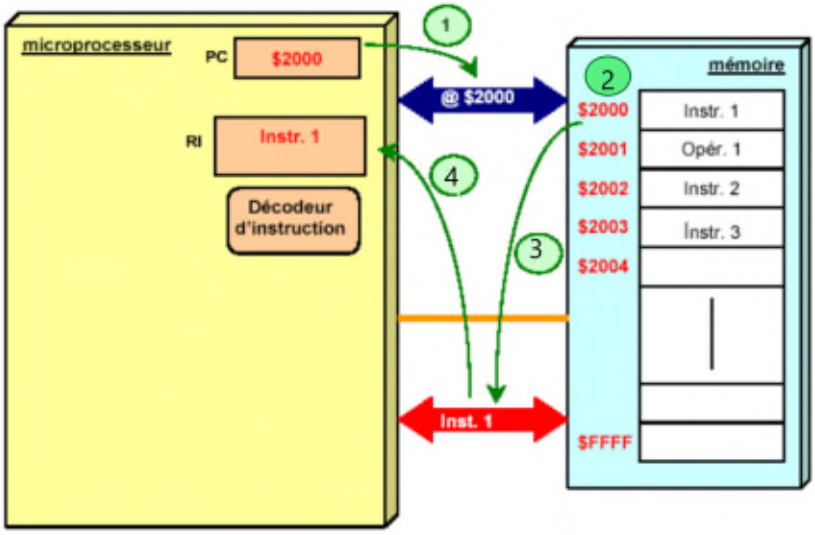


Figure 27 : La première phase d'exécution d'une instruction.

1.1 Mettre le contenu du **CO** dans le **registre d'adresse mémoire** (R@M). *// CO qui contient l'adresse de l'instruction suivante.*

(CO) → R@M

1.2 Commande de **lecture** à partir de la mémoire
1.3 Sélection de l'instruction (Adresse d'instruction) et **son contenu** est transféré vers le **registre de Données Mémoire** (RDM). *// au bout d'un certain temps (temps d'accès à la mémoire)*

(@inst)→ RDM

1.4 Transfert du contenu du RDM dans le **registre instruction** (RI) du processeur.

(RDM) → RI

Phase 2 : ***Décodage de l'instruction et recherche de l'opérande***

2.1 Analyse et **décodage** du **code opération.** // *L'unité de commande transforme l'instruction en une suite de commandes élémentaires nécessaires au traitement de l'instruction.*
2.2 Transfert de l'ADresse de l 'OPérande (ADOP) dans le **R@M**.

(ADOP) → R@M

2.3 Commande de **lecture.**
2.4 Sélection de l'opérande (Adresse opérande) et **son contenu** est transféré vers le **RDM.**

(N°ope) → RDM

2.5 Transfert du contenu du **RDM** vers l'**UAL**.

(RDM) → UAL

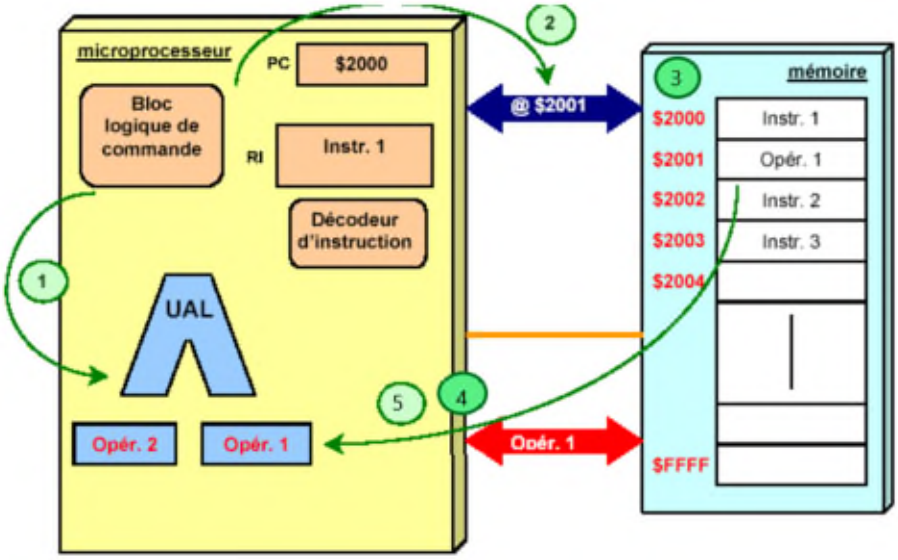


Figure 28 : La deuxième phase d'exécution d'une instruction.

Phase 3 : ***Exécution de l'instruction et passer à l'instruction suivante***

3.1 Commande de **l'exécution de l'opération** (ou exécution de l'instruction).
3.2 Les **drapeaux** sont positionnés (registre d'état).
3.3 L'**unité de commande** positionne le **CO** pour **l'instruction suivante.**

(CO) +1 → CO

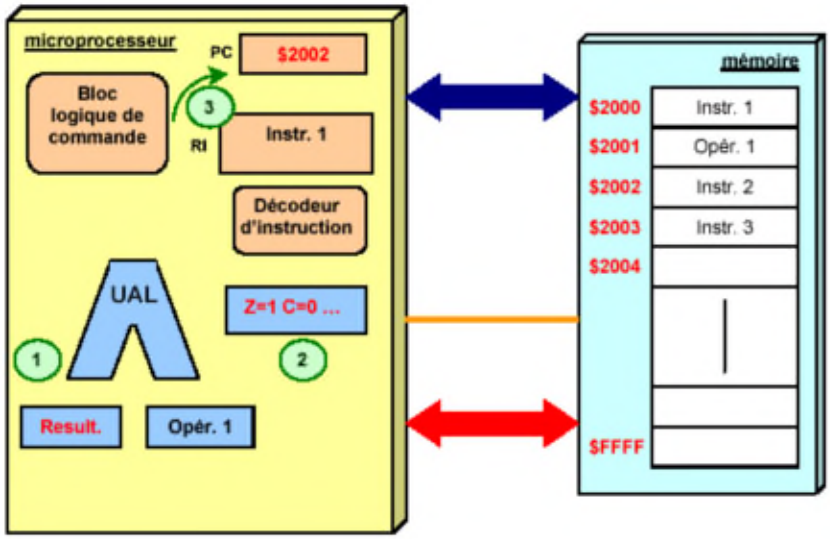


Figure 29 : La troisième phase d'exécution d'une instruction.

Remarque :

- L'étape **3.3** peut être effectuée en même temps que l'étape **1.2**
- La **phase 1** ne change pas pour l'ensemble des instructions, par contre la **phase 2** et **3** changes selon l'**instruction** et le **mode d'adressage**

2ème Cas : une instruction arithmétique à un seul opérande (mode immédiat)

Phase 2 : ***Décodage de l'instruction et recherche de l'opérande***

2.1 Analyse et décodage du code opération.
2.2 Transfert l 'Opérande (valeur contenue dans le RI) dans le UAL.

(RI) → UAL

3ème Cas : une instruction arithmétique à un seul opérande (mode indirect)

Phase 2 : ***Décodage de l'instruction et recherche de l'opérande***

2.1 Analyse et décodage du code opération.
2.2 Transfert de l'ADresse de l 'OPérande (ADOP) dans le R@M.

(ADOP) → R@M

2.3 Commande de lecture de l'adresse.
2.4 Transfert du contenu du RDM vers le R@M

(RDM) → R@M

2.5 Commande de lecture de l'opérande contenu dans l'adresse
2.6 Sélection de l'opérande (Adresse opérande) et son contenu est transféré vers le RDM.

(N°ope) → RDM

2.7 Transfert du contenu du RDM vers l'UAL.

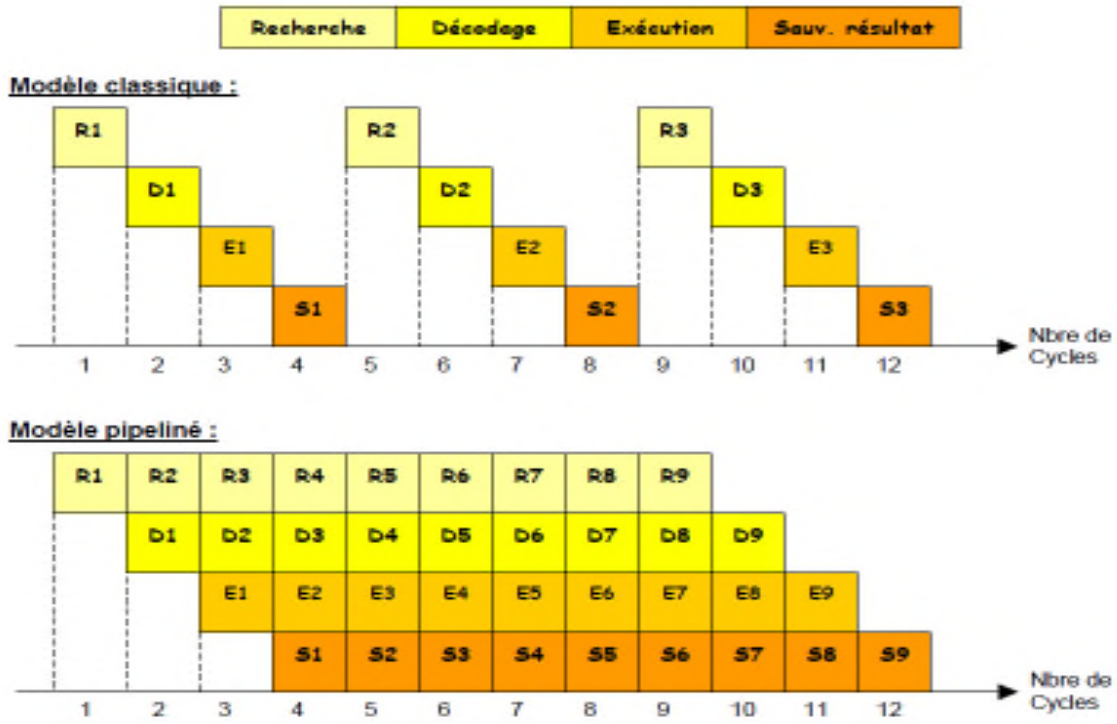
(RDM) → UAL

3.7. Pipeline

La **technique du pipeline** a été créée au début des année 1990. Elle permet d'améliorer l'efficacité du processeur.

- C'est une technique de mise en œuvre qui permet à **plusieurs instructions** de se **chevaucher** pendant l'**exécution**.
- Une **instruction** est **découpée** dans un pipeline en petits morceaux appelés ***étage de pipeline***.
- La technique du pipeline **améliore le débit** des instructions plutôt que le temps d'exécution de chaque instruction.
- Elle exploite le **parallélisme** entre instructions d'un flot séquentiel d'instructions
- Elle présente l'avantage de pouvoir, contrairement à d'autres techniques d'accélération, à être rendue **invisible du programmeur**.

Exemple de l'exécution en 4 phases d'une instruction :



3.7.1. Performance d'un pipeline

a. Première méthode (selon le nombre d'instructions)

Pour exécuter **n** instructions, en supposant que chaque instruction s'exécute en **k** cycles d'horloge, il faut :

- **n*k cycles d'horloge** pour une **exécution séquentielle**.
- **k cycles d'horloge** pour exécuter la **première instruction** puis **n-1 cycles** pour les **n-1 instructions** suivantes si on utilise un pipeline de **k étages**

Le **gain** obtenu est donc de : **G= n*k/k+n-1**

Lorsque le nombre **n** d'instructions à exécuter est grand par rapport à **k**, on peut admettre qu'on divise le temps d'exécution par **k**.

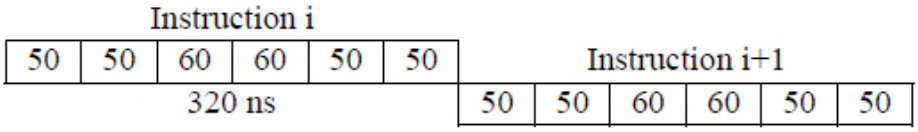
Exemple (de la figure précédant) : la machine débute l'exécution d'une instruction à chaque cycle et le pipeline est pleinement occupé à partir du quatrième cycle (Nombre de cycles = 4).

Le gain du pipeline par rapport à une exécution d'un modèle classique (pour le même nombre d'instructions) : G = (3*4) + (4+3-1) = 12/6 = 2

➔ Le gain obtenu dépend donc du nombre d'étages du pipeline.

b. Deuxième méthode (selon le temps exécution)

Considérons un pipeline à **6 étages** de temps : *50 ns, 50 ns, 60 ns, 60 ns, 50 ns et 50 ns*. Dans un premier temps, on n'utilise pas le pipeline. Combien de temps prennent l'exécution de 100 instructions ?

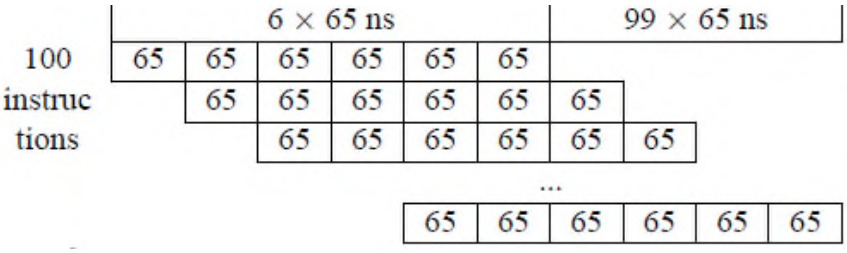


Temps d'exécution des instructions = **Nombre d'instruction * Temps d'exécution d'une instruction**, donc **Temps d'exécution de 100 instructions** = 100 * 32 = **32 000 ns.**

Utilisons le pipeline :

- **Le temps de chaque étage** est alors le même (celui du plus lent car les autres étages doivent attendre).
- **Le temps de passage entre deux étages** ne peut pas être instantané. Il faut attendre une stabilisation des registres de pipeline, prenons ici *5 ns*.

Temps d'un étage = MAX (temps des étages) + temps de stabilisation = 60 + 5 = 65 ns.



Temps d'exécution des instructions (avec pipeline) = **Temps d'un étage + ((Nombre d'instruction -1)* Temps d'exécution d'une instruction)**

Donc **Temps d'exécution de 100 instructions (avec pipeline)** = 65*6 + 99*65 = 6 825 ns.

Temps moyen traitement d'une instruction non pipeliné = 320 ns.

Temps moyen traitement d'une instruction pipeliné = 65 ns (dans notre exemple : 68,25 ns).

Donc **Accélération** = 320/65 = 4,92.

➔ Un pipeline n'accélère pas le temps de traitement d'une instruction mais augmente le débit des instructions : dans un intervalle de temps on traite plus d'instructions.

Exemples du nombre d'étages de pipeline dans les processeurs :

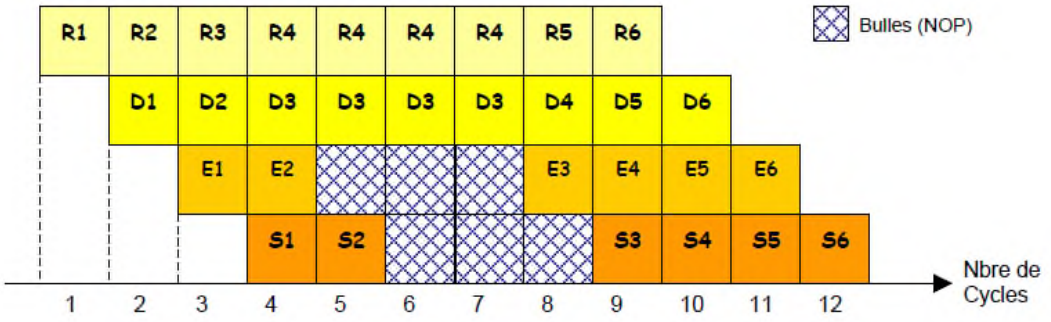
L'Athlon d'AMD comprend un pipeline de **11** étages.

Les Pentium 2, 3 et 4 d'Intel comprennent respectivement un pipeline de **12, 10** et **20** étages.

3.7.2. Les aléas dans le pipeline (Problèmes)

Il existe 3 principaux cas où la performance d'un processeur pipeliné peut être dégradé, ces cas de dégradations de performances sont appelés des **aléas** :

- **Aléa structurel** qui correspond au cas où deux instructions ont besoin d'utiliser la même ressource du processeur (conflit de dépendance),
- **Aléa de données** qui intervient lorsque le résultat d'une instruction dépend de celui d'une instruction précédente qui n'est pas encore terminée.
- **Aléas de contrôle** qui se produit chaque fois qu'une instruction de branchement est exécutée. L'exécution d'un saut conditionnel ne permet pas de savoir quelle instruction il faut charger dans le pipeline puisque deux choix sont possibles (l'instruction suivante ou les instructions qui suivent le saut).



Lorsqu'un aléa se produit, cela signifie qu'une instruction ne peut continuer à progresse dans le pipeline. Pendant un ou plusieurs cycles. L'instruction va rester bloquée dans un étage du pipeline, mais les instructions situées plus en avant pourront continuer à s'exécuter jusqu'à ce que l'aléa ait disparu. Plus le pipeline possède d'étages, plus la pénalité est grande. Les étages vacants du pipeline sont appelés des « **bulles** » de pipeline, en pratique une bulle correspond en fait à une instruction **NOP (No OPeration)** émise à la place de l'instruction bloquée.

Remarque :

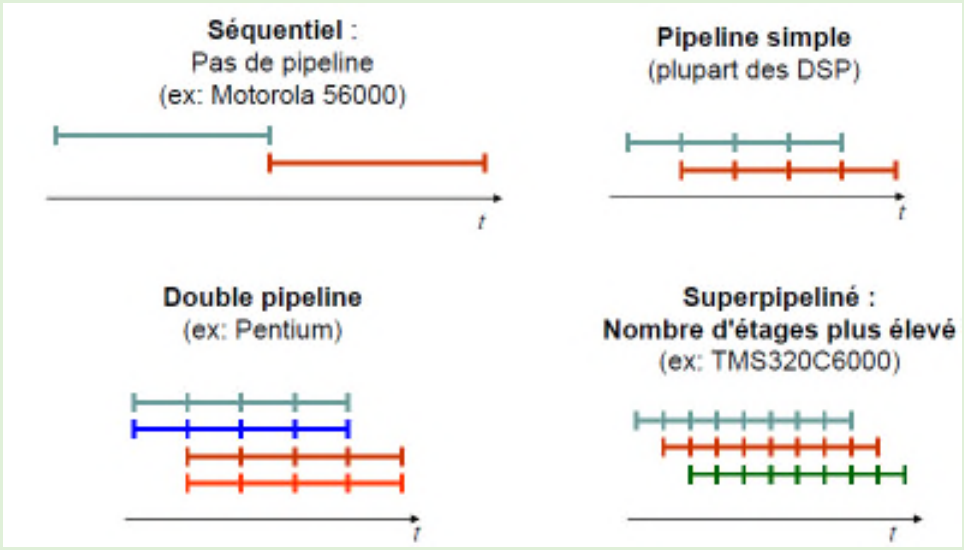
- *Avantages du pipeline* :

- Les ressources de l'unité d'exécution sont exploitées au maximum. À chaque coup d'horloge, on peut produire le résultat d'une instruction (dans un monde idéal).
- Si on découpe l'exécution d'une instruction en étapes très petites, on peut augmenter la fréquence d'horloge.

- *Limites du pipeline* :

- Les étapes d'une instruction n'ont pas toutes la même durée. Dans un pipeline, les durées des étapes sont forcément égales et déterminées par l'étape la plus longue.
- Des dépendances entre les instructions, des branchements, ou l'accès de plusieurs instructions aux mêmes ressources diminuent le gain apporté par les pipelines
- Il faut du temps, du matériel supplémentaire et des algorithmes plus complexes pour propager les informations d'un étage à l'autre du pipeline.

- *Types de pipelining*



3.8. Horloge et séquenceur

3.8.1. Horloge

L'**horloge** (base de temps) divise le temps en battements de même durée appelés **cycles**. Elle distribue des impulsions régulièrement pour synchroniser les différentes opérations élémentaires à effectuer pendant le déroulement d'une instruction.

A chaque cycle d'horloge et en fonction des ordres données par l'unité de contrôle, certaines portes (principe des vannes) se ferment d'autres s'ouvrent pour laisser circuler les informations.

Cycle machine : *cycle de base ou élémentaire égal à l'inverse de la fréquence*. Il est utilisé pour synchroniser chaque étape des cycles de recherche et d'exécution.

Cycle instruction : *cycle recherche + cycle exécution* : chacun d'eux nécessite plusieurs cycles machine (dépendant de l'instruction)

Cycle CPU : *temps d'exécution de l'instruction la plus courte* (recherche + exécution)

3.8.2. Séquenceur

Le **séquenceur** est un automate distribuant des signaux de commandes aux divers unités participantes à l'exécution d'une instruction selon un chronogramme précis en tenant compte des temps de réponse des circuits sollicités. Il peut être câblé ou microprogrammé

a. **Séquenceur câblé :** C'est est un circuit séquentiel (synchrone) complexe comprenant un sous circuit pour chacune des instructions à commander. Ce sous circuit est activé par le décodeur.
b. **Séquenceur microprogramme :** Il est possible de remplacer un circuit logique par **une suite de micros instructions (microprogramme)** stockées dans une mémoire (ROM) de microprogrammation. Le code de l'opération à exécuter dans l'instruction est utilisé comme étant l'adresse de la 1ère micro instruction du microprogramme.

Ce microprogramme est capable de générer une suite de signaux de commande équivalent à celle produite par un séquenceur câble.

Remarque :

- La vitesse de fonctionnement d'un ordinateur ne dépend pas seulement de sa fréquence d'horloge mais aussi du cycle mémoire et de la vitesse du bus.
- Un séquenceur microprogramme est plus lent qu'un séquenceur câblé.

3.9. Conclusion

Dans ce chapitre, nous avons vu que l'objet de la fonction d'exécution d'un ordinateur est d'exécuter une suite d'instructions sur un ensemble de données. Pour cela, un problème est traduit en une suite d'instructions machines caractéristiques d'un processeur capable de les exécuter en utilisant un ensemble de circuits électroniques. Nous avons été amenés à indiquer que le programme machine exécutable doit être placé dans la mémoire principale (mémoire centrale) et que le microprocesseur exécute ce programme instruction après instruction selon les notions liées aux microcommandes et aux séquenceurs. Nous avons également donné un aperçu d'une méthode, nommée *pipeline*, permettant une amélioration des performances des microprocesseurs

En présentant ce chapitre, nous avons présenté les fonctions essentielles de l'exécution des instructions reposant sur une architecture de machine dite *architecture de Von Neumann* qui correspond encore à l'architecture la plus répandue.

Chapitre IV :

Microprocesseur MIPS 3000

4.1. Introduction

Le **processeur** est le véritable **cerveau** de l'ordinateur. 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, en calculant le CPI (Cycle Per Instruction). Nous allons étudier la programmation en assembleur d'un microprocesseur. Pour cette partie nous verrons une présentation d'une version légèrement simplifiée de l'architecture externe et interne du processeur **MIPS R3000**, puis nous décrirons le langage d'assemblage du processeur MIPS, ainsi que différentes conventions relatives à l'écriture des programmes en langage d'assemblage.

4.2. Processeur

Aussi appelé CPU (*Central Processing Unit*), c'est un circuit électronique cadencé au rythme d'une horloge interne, grâce à un cristal de quartz qui, soumis à un courant électrique, 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.

Autrefois agglomérat de circuits physiquement séparés, le microprocesseur est né en 1971 (là encore, le préfixe « micro », à l'origine synonyme de petite taille par rapport aux processeurs sur les gros systèmes, a disparu des dénominations courantes). On est passé de deux mille trois cents transistors (les composants de base des circuits informatiques) pour le premier microprocesseur à plusieurs centaines de millions actuellement.

4.2.1. 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) de ce qui

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.

4.2.2. Calcul de CPI (Cycle Per Instruction)

A chaque **top d'horloge**, le processeur exécute une action correspondant à une instruction ou une partie d'instruction. Donc chaque **instruction** nécessite un certain **nombre de cycles** d'horloge pour s'effectuer :

- Le **nombre de cycles** dépend de la **complexité de l'instruction.**
- La **durée d'un cycle** dépend de la fréquence d'**horloge du séquenceur.**

On peut caractériser la **puissance d'un microprocesseur** par le nombre d'instructions qu'il est capable de traiter par seconde. Pour cela, on définit :

- Le **CPI** (*Cycle Par Instruction*) qui représente le **nombre moyen de cycles d'horloge** nécessaire pour **l'exécution d'une instruction** pour un microprocesseur donné.
- Le **MIPS** (*Millions d'Instructions Par Seconde*) qui représente **la puissance de traitement** du microprocesseur.

L'indicateur appelé **CPI** (Cycles Par Instruction) permet de représenter le **nombre moyen de cycles d'horloge** nécessaire à **l'exécution d'une instruction** sur un microprocesseur. La moyenne des cycles par instruction dans un processus donné est définie comme suit :

$$CPI = \frac{\sum_i (IC_i)(CC_i)}{IC}$$

Ou $\mathbf{IC_i}$ est le ***nombre d'instructions*** pour un type d'instruction donné i, $\mathbf{CC_i}$ est les ***cycles d'horloge*** pour ce type d'instruction et $\mathbf{IC=\sum_i (IC_i)}$ est le ***nombre total d'instructions***. La sommation résume tous les types d'instructions pour un processus d'analyse comparative donné.

La **puissance du processeur** peut ainsi être caractérisée par le nombre d'instructions qu'il est capable de traiter par seconde. L'unité utilisée est le MIPS (Millions d'Instructions Par Seconde) correspondant à la fréquence (en MHz) du processeur que divise le CPI et il est défini comme suit :

$$MIPS = \frac{Fréquence}{PCI}$$

Remarque :

Pour augmenter les performances d'un microprocesseur, on peut donc soit augmenter la fréquence d'horloge (limitation matérielle), soit diminuer le **CPI** (choix d'un jeu d'instruction adapté).

Exercice 1 : Pour une processeur MIPS multicycle, il existe cinq types d'instructions :

- Load (5 cycles)
- Store (4 cycles)
- R-type (4 cycles)
- Branch (3 cycles)
- Jump (3 cycles)

Si un programme a :

- 50% load instructions
- 25% store instructions
- 15% R-type instructions
- 8% branch instructions
- 2% jump instructions

Calculer CPI nécessaire pour l'exécution d'une instruction.

Solution :

CPI = (5*50+4*25+4*15+3*8+3*2) / (50+25+15+8+2) = **4.4**
Alors le CPI est égal à 4.4

Exercice 2 : Un processeur à 400 MHz a été utilisé pour exécuter un programme de référence avec le mélange d'instructions et le nombre de cycles d'horloge suivants :

Type instruction	Nombre instruction	Nombre de cycle d'horloge
Arithmétique entière	45000	1
Transfert de données	32000	2
Point flottant	15000	2
Transfert de contrôle	8000	2

Déterminer l'effectif CPI et le taux MIPS pour ce programme

Solution :

CPI = (45000*1+32000*2+15000*2+8000*2) / (45000+32000+15000+8000) =**1.55**

Alors le CPI est égal à 1.55

MIPS = 400/1.55= **258 MIPS**

Alors le MIPS est égal à 258

4.2.3. Processeur CISC et RISC

Actuellement l'architecture des microprocesseurs se compose de deux grandes familles:

- N'implanter en machine que les mécanismes réellement utiles (dont on a statistiquement montré l'utilité), c'est l'approche **RISC** (Reduced Instruction Set Computer).
  Exemple : PowerPC, MIPS, Sparc

- Faire correspondre à chaque structure de données exprimées dans le langage de haut niveau un mode d'adressage adapté dans le langage machine, c'est l'approche **CISC** (Complex Instruction Set Computer)
  Exemple : Pentium

Si dans l'approche **CISC**, la volonté était de séparer la machine matérielle de l'implantation, dans l'approche, c'est la volonté d'une optimisation globale (matérielle et logicielle) qui en est le moteur : on souhaite réaliser des architectures efficaces pour l'exécution des programmes.

a- Caractéristiques :

Leurs caractéristiques sont les suivantes :

RISC	CISC
• Jeu d'instructions de taille limitée • Instructions simples ne prenant qu'un seul cycle • Format des instructions petit et fixé • Modes d'adressage réduits • Décodeur simple (câble) • Beaucoup de registres • Seules les instructions LOAD et STORE ont accès à la mémoire • Compilateur complexe	• Jeu d'instructions de taille importante • Instructions pouvant être complexes prenant plusieurs cycles • Format d'instructions variables (de 1 à 5 mots) • Modes d'adressages complexes. • Décodeur complexe (microcode) • Peu de registres • Toutes les instructions sont susceptibles d'accéder à la mémoire • Compilateur simple

b- Avantages et inconvénients :

Ces deux types ont leurs avantages et leurs inconvénients :

	Avantages	Inconvénients
RISC	+ programmation de plus haut niveau. + programmation plus compacte (écriture plus rapide et plus élégante des applications) + moins d'occupation en mémoire et à l'exécution	- complexifie le processeur - taille des instructions élevée et variable : pas de structure fixe - exécution des instructions : complexe et peu performante.
CISC	+ instructions de format standard + traitement plus efficace + possibilité de pipeline plus efficace	- programmes plus volumineux - compilation plus compliquée

4.3. Microprocesseur MIPS R3000

MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) est une architecture de microprocesseur de type **RISC** développée par la compagnie MIPS Computer Systems Inc. Les processeurs fabriqués selon cette architecture on les retrouve dans plusieurs systèmes embarqués, comme les ordinateurs de poche, les routeurs Cisco et les consoles de jeux vidéo (Nintendo 64 et Sony PlayStation, PlayStation 2 et PSP).

Cette architecture est suffisamment **simple** pour présenter les principes de base de l'architecture des processeurs et suffisamment puissante pour supporter un système d'exploitation multi-tâches tel qu'UNIX.

Le processeur **MIPS R3000 (**la deuxième génération de processeur de la société MIPS**)** est industriel, conçu dans les années 80 et utilisé jusqu'au début 2000 (PlayStation 2). Les premières implémentations de l'architecture MIPS étaient de **32 bits** (autant au niveau des registres que des chemins de données), mais par la suite, on a développé des implémentations de 64 bits.

Vers la fin des années 1990, on estimait que les processeurs dérivés de l'architecture MIPS occupaient le tiers des processeurs RISC produits. Le MIPS R4000 sorti en 1991 serait le premier processeur 64 bits. Il a été supporté par Microsoft de Windows NT 3.1 jusqu'à Windows NT 4.0

Il existe plusieurs jeux d'instructions MIPS qui sont ***rétro-compatibles*** (*backward compatible*) : MIPS I, MIPS II, MIPS III, MIPS IV, et MIPS V ainsi que MIPS32 et MIPS64. MIPS32 et MIPS64, qui se basent sur MIPS II et MIPS V et ont été introduits comme jeux d'instructions normalisés.

4.4. Structure externe du processeur MIPS R3000

L'architecture externe représente ce que doit connaitre un programmeur souhaitant programmer en assembleur ou la personne souhaitant écrire un compilateur pour ce processeur:

- Les registres visibles.
- L'adressage de la mémoire.
- Le jeu d'instructions.
- Les mécanismes de traitement des interruptions et exceptions.

4.4.1. Registres visibles du logiciel

Tous les **registres visibles du logiciel**, c'est à dire ceux dont la valeur peut être **lue ou modifiée** par les instructions, sont des registres **32 bits**.

Afin de mettre en œuvre les mécanismes de protection nécessaires pour un **système d'exploitation multi-tâches**, le processeur possède deux modes de fonctionnement : **utilisateur** et **superviseur**. Ces deux modes de fonctionnement imposent d'avoir deux catégories de registres.

a- Registres non protégés :

Le processeur MIPS possède **35** registres manipulés par les instructions standard (c'est à dire les instructions qui peuvent s'exécuter aussi bien en **mode utilisateur** qu'en **mode superviseur).**

Ri	• **32** registres généraux ($0 \leq i \leq 31$) • Ces registres sont directement adressés par les instructions, et permettent de stocker des résultats de calculs intermédiaires. • Le registre **R0** est un registre particulier : - La lecture fournit la valeur constante "0x00000000" - L'écriture ne modifie pas son contenu. • Le registre **R31** est utilisé pour les instructions d'appel de procédures (instructions **BGEZAL, BLTZAL, JAL et JALR**) et pour sauvegarder l'adresse de retour.
PC	• Registre compteur de programme (Program Counter) • Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.
HI et LO	• Registres pour la multiplication ou la division • Ces deux registres **32** bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division, qui est un mot de **64** bits.

b- Registres protégés

L'architecture **MIPS** définit 32 registres (**numérotés de 0 à 31**), qui ne sont accessibles, en lecture comme en écriture, que par des instructions privilégiées (c'est à dire des instructions qui ne peuvent être exécutées qu'en mode superviseur). En pratique, cette version de processeur **MIPS R3000** en utilise 4 pour la gestion des interruptions et des exceptions (voir chapitre 5).

SR	• **Registre d'état** (Status Register). • Il contient en particulier le bit qui définit le mode : superviseur ou utilisateur, ainsi que les bits de masquage des interruptions. (Ce registre possède le **numéro 12**)
CR	• **Registre de cause** (Cause Register). • En cas d'interruption ou d'exception, son contenu définit la cause pour laquelle on fait appel au programme de traitement des interruptions et des exceptions. (Ce registre possède le **numéro 13**)
EPC	• **Registre d'exception** (Exception Program Counter). • Il contient l'adresse de retour (**PC + 4**) en cas d'interruption. • Il contient l'adresse de l'instruction fautive en cas d'exception (PC). (Ce registre possède le **numéro 14**)
BAR	• **Registre d'adresse illégale** (Bad Address Register). • En cas d'exception de type "adresse illégale", il contient la valeur de l'adresse mal formée. (Ce registre possède le **numéro 8**)

4.4.2. Adressage de la mémoire

a- Adresse octet

Toutes les **adresses** émises par le processeur sont des **adresses octets**, ce qui signifie que la **mémoire** est vue comme un **tableau d'octets**, qui contient aussi bien les données que les instructions.

- Les **adresses** sont codées sur **32 bits.**
- Les **instructions** sont codées sur **32 bits.**
- Les **échanges de données avec la mémoire** se font par **mot** (**4** octets consécutifs), **demi-mot** (**2** octets consécutifs), ou par **octet**. Pour les **transferts** de mots et de demi-mots, le processeur respecte la **convention "*little endian*"**.

L'**adresse d'un mot** de donnée ou d'une instruction doit être un **multiple de 4**. L'**adresse d'un demi-mot** doit être un multiple de **2** (On dit que les adresses doivent être "alignées").

Remarque :

Le processeur part en exception si une instruction calcule une adresse qui ne respecte pas cette contrainte.

b- Calcul d'adresse

Il existe **un seul mode d'adressage**, consistant à effectuer la somme entre le contenu d'un registre général **Ri**, défini dans l'instruction et d'un déplacement qui est une valeur immédiate signée, sur **16 bits**, contenue également dans l'instruction :

Adresse = Ri + Déplacement

c- Mémoire virtuelle

Pour des raisons de simplicité, cette version du processeur R3000 ne possède pas de mémoire virtuelle. C'est à dire que le processeur ne contient aucun mécanisme matériel de traduction des adresses logiques en adresses physiques. Les adresses calculées par le logiciel sont donc transmises au système mémoire sans modifications.

d- Segmentation

L'espace mémoire est découpé en 2 segments identifiés par le bit de poids fort de l'adresse :

adr 31 = 0 ➔ segment utilisateur

adr 31 = 1 ➔ segment système

- Quand le processeur est en **mode superviseur**, les **2 segments** sont accessibles.
- Quand le processeur est en **mode utilisateur**, **seul** le segment utilisateur est accessible.

Remarque :

Le processeur part en exception si une instruction essaye d'accéder à la mémoire avec une adresse correspondant au segment système alors que le processeur est en mode utilisateur.

4.5. Structure interne du processeur MIPS R3000

L'architecture externe du processeur et en particulier le **jeu d'instructions** sont décrits dans le titre suivant (4.6.), dont la lecture est indispensable pour la compréhension de celui-ci.

4.5.1. Interface

L'interface entre le processeur et la mémoire est réalisé par les signaux **ADR [31:0], DATA[31:0], RW[2:0], FRZ, BERR** (fig. 30). Les requêtes possibles vers la mémoire sont les suivantes :

RW	REQUETE	
000	**NO**	ni écriture, ni lecture
001	**WW**	écriture d'un mot
010	**WH**	écriture d'un demi-mot
011	**WB**	écriture d'un octet
1**	**RW**	lecture d'un mot

Dans le cas **WW,** les **4 octets** du **bus DATA** sont écrits en mémoire à une adresse alignée sur les **mots** (les 2 bits de poids faible de ADR ne sont pas pris en compte).

Dans le cas **WH**, les **2 octets** de poids faibles du **bus DATA** sont écrits à une adresse alignée sur les **demi-mots** (le bit de poids faible de ADR n'est pas pris en compte).

Dans le cas **WB**, l'**octet** de poids faible du bus **DATA** est écrit à l'adresse ADR.

Dans le cas **RW**, les **deux bits** de poids faible de ADR ne sont pas pris en compte, et la mémoire doit fournir sur le bus **DATA** un mot aligné.

Dans le cas des **instructions LBU** et **LHU**, c'est le processeur qui effectue le recadrage et l'extension de signe.

Dans le cas **où le système mémoire ne peut pas** satisfaire en **un cycle la requête d'écriture** ou de **lecture** (par exemple en cas de MISS dans le cache), le signal **FRZ** doit être activé. Le processeur maintient sa requête tant que le signal **FRZ** est actif.

Dans le cas d'une **erreur matérielle** lors d'un accès à la mémoire, cette erreur peut être signalée au processeur par le signal **BERR.**

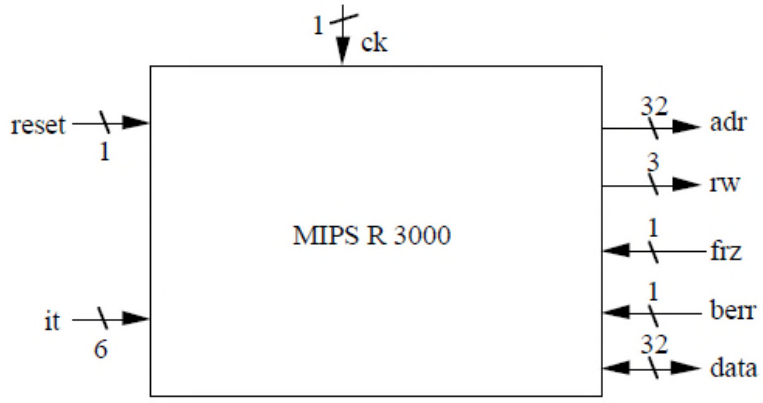


Figure 30 : L'interface du processeur MIPS R3000.

4.5.2. Architecture interne

L'architecture interne du processeur (fig. 31) se décompose en une partie opérative et une partie contrôle.

a- La partie opérative (PO) :

- Elle contient les **registres** et les **opérateurs**
- Elle réalise des **transferts élémentaires de données** entre un ou plusieurs registres sources et un registre destination.
- Un **transfert élémentaire** est exécuté en un cycle.
- La partie opérative est commandée par la partie contrôle.

b- La partie contrôle (PC) :

- Elle est chargée de définir, pour chaque **cycle d'horloge**, les transferts élémentaires qui doivent être réalisés par la **partie opérative**.
- La partie contrôle contient principalement un **séquenceur** décrit comme un automate d'états finis (automate de MOORE).

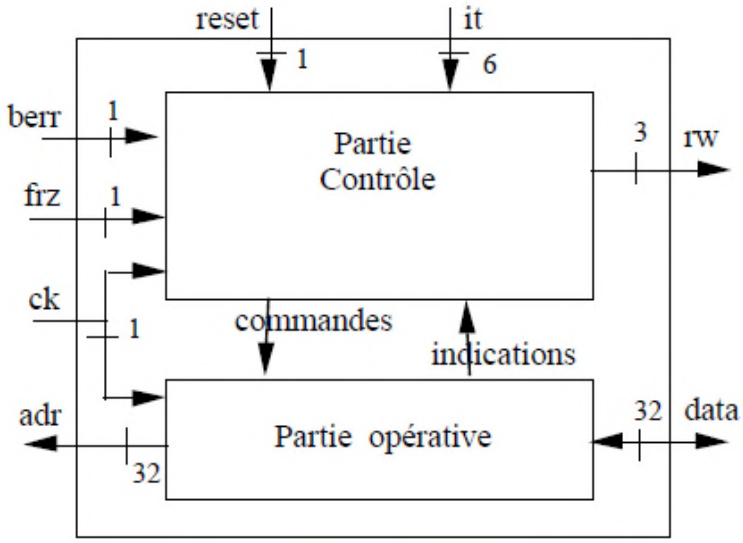


Figure 31: L'architecture interne du processeur MIPS R3000.

4.6. Jeu d'instructions du MIPS R3000

Le processeur MIPS possède **57 instructions** qui se répartissent en 4 classes :

- **33** instructions **arithmétiques/logiques** entre registres
- **12** instructions de **branchement**
- **7** instructions de **lecture/écriture** mémoire
- **5** instructions **systèmes**

4.6.1. Formats

Toutes les instructions ont une longueur de 32 bits et possèdent un des trois formats suivants :

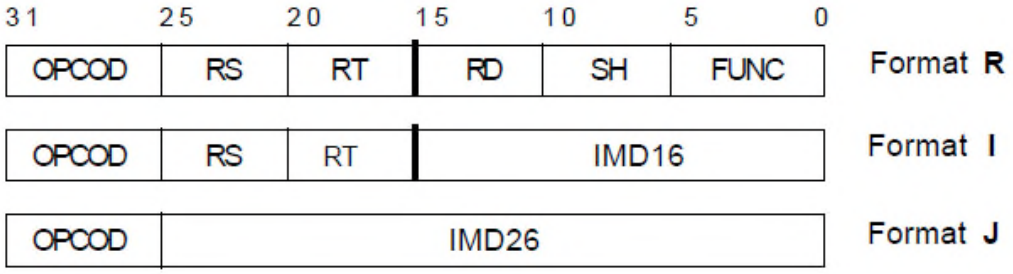


Figure 32 : Les différents formats d'instructions en MIPS

- Le format **J** n'est utilisé que pour les branchements à longue distance (inconditionnels).
- Le format **I** est utilisé par les instructions de lecture/écriture mémoire, par les instructions utilisant un opérande immédiat, ainsi que par les branchements courte distance (conditionnels).
- Le format **R** est utilisé par les instructions nécessitant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD.

4.6.2. Codage des instructions

Le **codage des instructions** est principalement défini par les **6 bits** du champs **code opération** de l'instruction (**INS 31:26**).

Cependant, trois valeurs particulières de ce champ définissent en fait une famille d'instructions : il faut alors analyser d'autres bits de l'instruction pour décoder l'instruction. Ces codes particuliers sont : SPECIAL (valeur "000000"), BCOND (valeur "000001") et COPRO (valeur "010000")

Ce tableau (Figure ci-dessous) exprime que l'instruction LHU (par exemple) possède le code opération "100101".

INS 28 : 26

INS 31 : 29	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Figure 33 : Le décodage du code opération.

Lorsque le code opération a la valeur SPECIAL ("000000"), il faut analyser les 6 bits de poids faible de l'instruction (**INS 5:0**) (fig. 34).

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Figure 34 : Le décodage du code opération = SPECIAL.

Lorsque le code opération a la valeur BCOND, il faut analyser les bits 20 et 16 de l'instruction.
Lorsque le code opération a la valeur COPRO, il faut analyser les bits 25 et 23 de l'instruction.
Les trois instructions de cette famille COPRO sont des instructions privilégiées (fig. 35).

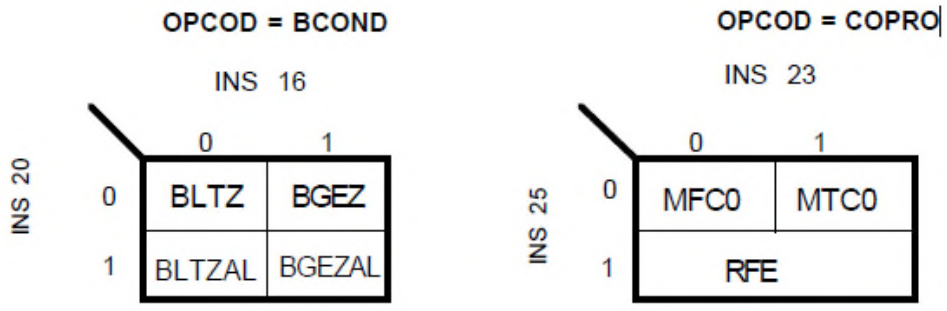


Figure 35 : Le décodage du code opération = BCOND ou = COPRO

4.6.3. Jeu d'instructions

Le **jeu d'instructions** est "*orienté registres*". Cela signifie que les instructions **arithmétiques et logiques** prennent leurs opérandes dans des registres et rangent le résultat dans un registre. Les seules instructions permettant de lire ou d'écrire des données en mémoire effectuent un simple transfert entre un registre général et la mémoire, sans aucun traitement arithmétique ou logique.

- La plupart des instructions **arithmétiques et logiques** (fig. 36 et fig. 37) se présentent sous les **2 formes** *registre-registre* et *registre-immédiat* :

ADD : R(rd) ← R(rs) op R(rt) format R

ADDI : R(rt) ← R(rs) op IMD format I

L'opérande immédiat 16 bits est signé pour les opérations arithmétiques et non signé pour les opérations logiques.

- Le **déplacement** est de **16 bits** pour les **instructions de branchement conditionnelles** (Bxxx) et de **26 bits** pour les **instructions de saut inconditionnelles** (Jxxx) (fig. 38). De plus les instructions **JAL, JALR, BGEZAL,** et **BLTZAL** sauvegardent une adresse de retour dans le registre **R31**. Ces instructions sont utilisées pour les appels de sous-programmes.
  Toutes les instructions de **branchement conditionnel** sont relatives au **compteur ordinal** (CO) pour que le code soit translatable. L'adresse de saut est le résultat d'une addition entre la valeur du compteur ordinal et un déplacement signé.

- Les instructions **MTC0** et **MFC0** permettent de transférer le contenu des registres **SR, CR, EPC** et **BAR** vers un registre général et inversement. Ces **2 instructions** ne peuvent être exécutées qu'en mode superviseur, de même que l'instruction **RFE** qui permet de restaurer l'état antérieur du registre d'état avant de sortir du gestionnaire d'exceptions (fig. 40).

Instructions Arithmétiques/Logiques entre registres				
Assembleur		Opération		Format
Add	Rd, Rs, Rt	Add overflow detection	Rd <- Rs + Rt	R
Sub	Rd, Rs, Rt	Subtract overflow detection	Rd <- Rs - Rt	R
Addu	Rd, Rs, Rt	Add no overflow	Rd <- Rs + Rt	R
Subu	Rd, Rs, Rt	Subtract no overflow	Rd <- Rs - Rt	R
Addi	Rt, Rs,I	Add Immediate overflow detection	Rt <- Rs + I	I
Addiu	Rt, Rs,I	Add Immediate no overflow	Rt <- Rs + I	I
Or	Rd, Rs, Rt	Logical Or	Rd <- Rs or Rt	R
And	Rd, Rs, Rt	Logical And	Rd <- Rs and Rt	R
Xor	Rd, Rs Rt	Logical Exclusive-Or	Rd <- Rs xor Rt	R
Nor	Rd, Rs, Rt	Logical Not Or	Rd <- Rs nor Rt	R
Ori	Rt, Rs, I	Or Immediate unsigned immediate	Rt <- Rs or I	I
Andi	Rt, Rs, I	And Immediate unsigned immediate	Rt <- Rs and I	I
Xori	Rt, Rs, I	Exclusive-Or Immediate unsigned immediate	Rt <- Rs xor I	I
Sllv	Rd, Rt, Rs	Shift Left Logical Variable 5 lsb of Rs is significant	Rd <- Rt << Rs	R
Srlv	Rd, Rt, Rs	Shift Right Logical Variable 5 lsb of Rs is significant	Rd <- Rt >> Rs	R
Srav	Rd, Rt, Rs	Shift Right Arithmetical Variable 5 lsb of Rs is significant	Rd <- Rt >>* Rs	R
Sll	Rd, Rt, sh	Shift Left Logical	Rd <- Rt << sh	R
Srl	Rd, Rt, sh	Shift Right Logical	Rd <- Rt >> sh	R
Sra	Rd, Rt, sh	Shift Right Arithmetical * : with sign extension	Rd <- Rt >>* sh	R
Lui	Rt, I	Load Upper Immediate 16 lower bits of Rt are set to zero	Rt <- I || "0000"	I

Figure 36 : Les instructions Arithmétiques/Logiques entre registres.

Instructions Arithmétiques/Logiques (suite)

Assembleur	Opération		Format
Slt Rd, Rs, Rt	Set if Less Than	Rd <- 1 if Rs<Rt else 0	R
Sltu Rd, Rs, Rt	Set if Less Than Unsigned	Rd <- 1 if Rs<Rt else 0	R
Slti Rt, Rs, I	Set if Less Than Immediate sign extended Immediate	Rt <- 1 if Rs < I else 0	I
Sltiu Rt, Rs, I	Set if Less Than Immediate unsigned immediate	Rt <- 1 if Rs < I else 0	I
Mult Rs, Rt	Multiply	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Multu Rs, Rt	Multiply Unsigned	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Div Rs, Rt	Divide	Rs / Rt LO <- Quotient HI <- Remainder	R
Divu Rs, Rt	Divide Unsigned	Rs / Rt LO <- Quotient HI <- Remainder	R
Mfhi Rd	Move From HI	Rd <- HI	R
Mflo Rd	Move From LO	Rd <- LO	R
Mthi Rs	Move To HI	HI <- Rs	R
Mtlo Rs	Move To LO	LO <- Rs	R

Figure 37: Les instructions Arithmétiques/Logiques (Suite).

Instructions de Branchement			
Assembleur	Opération		Format
Beq Rs, Rt, Label	Branch if Equal	PC <- PC+4+(I*4) if Rs = Rt PC <- PC+4 if Rs ≠ Rt	I
Bne Rs, Rt, Label	Branch if Not Equal	PC <- PC+4+(I*4) if Rs ≠ Rt PC <- PC+4 if Rs = Rt	I
Bgez Rs, Label	Branch if Greater or Equal Zero	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0	I
Bgtz Rs, Label	Branch if Greater Than Zero	PC <- PC+4+(I*4) if Rs > 0 PC <- PC+4 if Rs ≤ 0	I
Blez Rs, Label	Branch if Less or Equal Zero	PC <- PC+4+(I*4) if Rs ≤ 0 PC <- PC + 4 if Rs > 0	I
Bltz Rs, Label	Branch if Less Than Zero	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0	I
Bgezal Rs, Label	Branch if Greater or Equal Zero and link	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0 R31 <- PC+4 in both cases	I
Bltzal Rs, Label	Branch if Less Than Zero and link	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0 R31 <- PC+4 in both cases	I
J Label	Jump	PC <- PC 31:28 || I*4	J
Jal Label	Jump and Link	R31 <- PC+4 PC <- PC 31:28 || I*4	J
Jr Rs	Jump Register	PC <- Rs	R
Jalr Rs	Jump and Link Register	R31 <- PC+4 PC <- Rs	R
Jalr Rd, Rs	Jump and Link Register	Rd <- PC+4 PC <- Rs	R

Figure 38 : Les instructions de Branchement.

Instructions de lecture/écriture mémoire			
Assembleur		Opération	Format
Lw	Rt, I (Rs)	Load Word sign extended Immediate — Rt <- M (Rs + I)	I
Sw	Rt, I (Rs)	Store Word sign extended Immediate — M (Rs + I) <- Rt	I
Lh	Rt, I (Rs)	Load Half Word — Rt <- M (Rs + I) sign extended Immediate. Two bytes from storage is loaded into the 2 less significant bytes of Rt. The sign of these 2 bytes is extended on the 2 most signficant bytes.	I
Lhu	Rt, I (Rs)	Load Half Word Unsigned — Rt <- M (Rs + I) sign extended Immediate. Two bytes from storage is loaded into the the 2 less significant bytes of Rt, other bytes are set to zero	I
Sh	Rt, I (Rs)	Store Half Word — M (Rs + I) <- Rt sign extended Immediate. The Two less significant bytes of Rt are stored into storage	I
Lb	Rt, I (Rs)	Load Byte — Rt <- M (Rs + I) sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt. The sign of this byte is extended on the 3 most signficant bytes.	I
Lbu	Rt, I (Rs)	Load Byte Unsigned — Rt <- M (Rs + I) sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt, other bytes are set to zero	I
Sb	Rt, I (Rs)	Store Byte — M (Rs + I) <- Rt sign extended Immediate. The less significant byte of Rt is stored into storage	I

Figure 39: Les instructions de lecture/écriture mémoire.

Instructions Systèmes		
Assembleur	Opération	Format
Rfe	Restore From Exception SR <- SR 31:4 II SR 5:2 Privileged instruction. Restore the prevous IT mask and mode	R
Break n	Breakpoint Trap SR <- SR 31:6 II SR 3:0 II "00" Branch to exception handler. PC <- "8000 0080" n defines the breakpoint number CR <- cause	R
Syscall	System Call Trap SR <- SR 31:6 II SR 3:0 II "00" Branch to exception handler PC <- "8000 0080" CR <- cause	R
Mfc0 Rt, Rd	Move From Control Coprocessor Rt <- Rd Privileged Instruction . The register Rd of the Control Coprocessor is moved into the integer register Rt	R
Mtc0 Rt, Rd	Move To Control Coprocessor Rd <- Rt Privileged Instruction . The integer register Rt is moved into the register Rd of the Control Coprocessor	R

Figure 40: Les instructions Systèmes.

4.7. Programmation du MIPS R3000

Le but d'un *programme X* écrit en langage d'assemblage est de fournir à un programme particulier (appelé **assembleur**) les directives nécessaires pour générer le code binaire représentant les instructions et les données qui devront être chargées en mémoire pour permettre au *programme X* de s'exécuter sur du matériel.

4.7.1. Organisation de la mémoire

Dans **l'architecture MIPS R3000**, **l'espace adressable** est divisé en **deux segments** : le segment **utilisateur** et le segment **noyau** (fig. 41).

Un **programme utilisateur**, utilise généralement trois sous-segments (appelés **sections**) dans le segment utilisateur :

- La section **text**, contient le **code exécutable en mode utilisateur**. Elle est implantée conventionnellement à l'adresse **0x00400000**. Sa taille est fixe et est calculée lors de l'assemblage. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé dans cette section ;
- La section **data**, contient les **données globales manipulées par le programme utilisateur**. Elle est implantée conventionnellement à l'adresse **0x10000000**. Sa taille

est fixe et est calculée lors de l'assemblage. Les valeurs contenues dans cette section peuvent être initialisées grâce à des directives contenues dans le programme source en langage d'assemblage ;

- La section **stack** contient **la pile d'exécution du programme**. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **0x7FFFEFFF**. Contrairement aux sections **data** et **text**, la pile s'étend vers les adresses décroissantes.

Deux autres sections sont définies dans le segment noyau :

- La section **ktext** contient le **code exécutable en mode noyau**. Elle est implantée conventionnellement à l'adresse **0x80000000**. Sa taille est fixe et est calculée lors de l'assemblage ;
- La section **kdata** contient les **données globales manipulées par le système d'exploitation** en mode noyau. Elle est implantée conventionnellement à l'adresse **0xC0000000**. Sa taille est fixe et est calculée lors de l'assemblage ;
- La section **kstack** contient la **pile d'exécution du programme**. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **0xFFFFEFFF**. Contrairement aux sections **data** et **text**. La pile s'étend vers les adresses décroissantes.

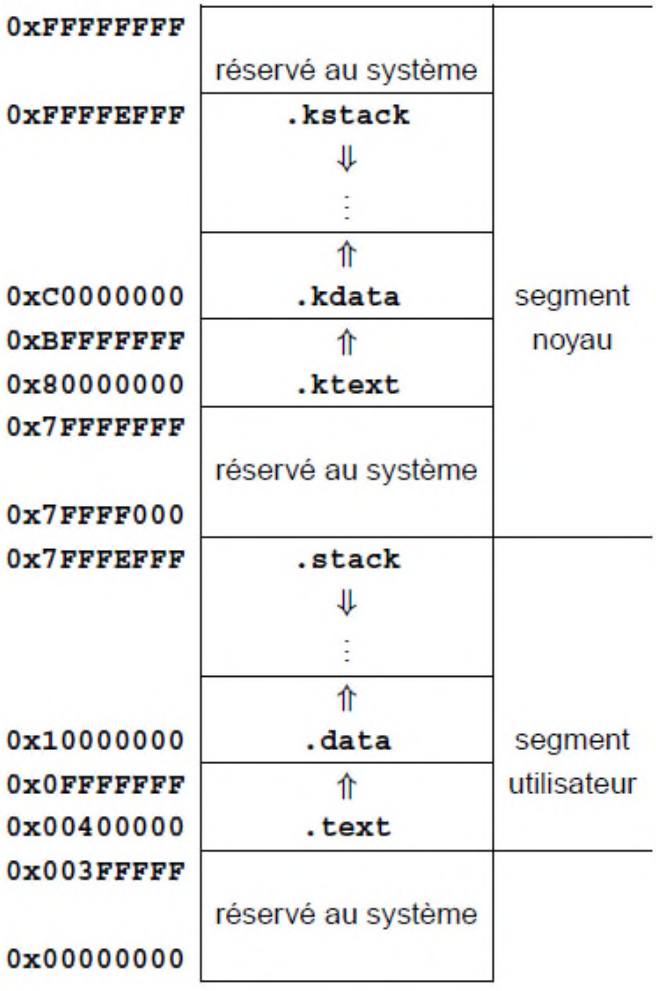


Figure 41: L'organisation de la mémoire du MIPS R3000.

4.7.2. Règles syntaxiques

a- **Les noms de fichiers** : Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixé par << **.s** >>.
Exemple : monprogramme.s

b- **Les commentaires** : Ils commencent par un **#** ou un **;** et s'achèvent à la fin de la ligne courante.
Exemple :

```
################################################
# Source Assembleur MIPS de la fonction memcpy
################################################
.... ; sauve la valeur copiée dans la mémoire
```

c- **Les entiers** : une valeur entière décimale est notée **250**, une valeur entière octale est notée **0372** (préfixée par un zéro), et une valeur entière hexadécimale est notée **0xFA** (préfixée par zéro suivi de x).
En hexadécimal, les lettres **A** à **F** peuvent être écrites en majuscule ou en minuscule.

d- **Les chaines de caractères** : elles sont simplement entre guillemets et peuvent contenir les caractères d'échappement du langage C.
Exemple : `"Oh la jolie chaîne avec retour à la ligne\n"`

e- **Les labels :** ce sont des mnémoniques correspondant à des adresses. Ces adresses peuvent être soit des **adresses de variables**, soit des **adresses de saut**. Ce sont des chaînes de caractères qui commencent par une lettre majuscule ou minuscule, un **$**, un**_**, ou un **.** . Ensuite, un nombre quelconque de ces mêmes caractères auxquels on ajoute des chiffres sont utilisés. Pour la déclaration, le label doit être suffixé par << **:** >>. **Exemple :** `$LC12:` Pour y référer, on supprime le << **:** >>.
Exemple :

```
message: .asciiz "Ceci est une chaîne de caractères...\n"
.text
__start:
la $4, message ; adresse de la chaine dans $4
ori $2, $0, 4 ; code du 'print_string' dans $2
syscall
```

Attention : sont illégaux les labels qui ont le même nom qu'un mnémonique de l'assembleur ou qu'un nom de registre.

f- **Les immédiats :** sont les **opérandes** contenus dans l'instruction. Ce sont aussi des **constantes.** Ils sont soit des entiers, soit des labels. Ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui l'utilise : **16** ou **26** bits.

g- **Les registres :** le processeur MIPS possède **32** registres accessibles au programmeur. Chaque registre est connu par **son numéro**, qui varie entre **0 et 31**, et est préfixé par un**$**.
Par exemple : le registre 31 sera noté **$31** dans l'assembleur.

En dehors du registre **$0**, tous les registres sont identiques du point de vue de la machine. Lorsque le registre **$0** est utilisé comme registre source dans une instruction, la valeur lue est toujours 0. Utiliser ce registre comme registre destination dans une instruction ne modifie pas sa valeur.

Afin de normaliser et de simplifier l'écriture du logiciel, des conventions d'utilisation des registres sont définies. Ces conventions sont particulièrement nécessaires lors de l'utilisation des fonctions.

`$0`	vaut zéro en lecture, non modifié par écriture
`$1`	réservé à l'assembleur. Ne doit pas être employé dans les programmes utilisateur
`$2`	valeur de retour des fonctions
`$3, $4`	argument des *syscall*
`$5, ..., $26`	registres de travail à sauver
`$27, ..., $28`	registres réservés aux procédures noyau. Ils ne doivent pas être employés dans les programmes utilisateur
`$29`	pointeur de pile
`$30`	pointeur sur les variables globales
`$31`	adresse de retour d'appel de fonction

h- **Les registres :** si une instruction nécessite plusieurs arguments, comme **par exemple l'addition** entre deux registres, ces arguments sont séparés par des virgules. Dans une instruction assembleur, on aura en général comme argument en premier le registre dans lequel est mis le résultat de l'opération, puis ensuite le premier registre source, puis enfin le second registre source ou une constante.

Exemple : `add $3, $2, $1`

i- **L'adressage mémoire :** le MIPS ne possède qu'un **unique mode d'adressage** : **l'adressage indirect registre** avec déplacement. Ainsi l'accès à une case mémoire à partir de l'adresse présente dans un registre se note par le déplacement, c.-à-d. un entier comme défini précédemment, suivi du registre entre parenthèses.

Exemples : `11($12), 013($12), 0xB($12).`

Ces trois exemples indiquent la case mémoire ayant comme adresse le contenu du registre **$12** plus 11.

S'il n'y a pas d'entier devant la parenthèse ouvrante, le déplacement est nul. En revanche, il n'est pas possible d'écrire des sauts à des adresses absolues ou relatives *constantes*, comme par exemple un `j 0x400000` ou un `bnez $3, -12`. Il faut nécessairement utiliser des labels.

4.7.3. Directives supportées par l'assembleur MIPS

Les directives ne sont pas des instructions exécutables par la machine, mais permettent de donner des ordres à l'assembleur. Toutes les pseudo-instruction commencent par le caractère << **.** >> ce qui permet de les différencier clairement des instructions.

a- Déclaration des sections : text, data et stack

Six directives permettent de spécifier quelle section de la mémoire est concernée par les instructions, macro-instructions ou directives qui les suivent. Sur ces six directives, **deux sont dynamiquement** gérées à l'exécution : ce sont celles qui concernent la pile utilisateur, **stack**, et la pile système, **kstack**. Ceci signifie que l'assembleur gère quatre compteurs d'adresse indépendants correspondants aux quatre sections **text**, **data**, **ktext** et **kdata**.

Syntaxe	Action	Description
`.text`	Passage dans la section **text**	Toutes les instructions et directives qui suivent concernent la section **text** dans le segment utilisateur.
`.data`	Passage dans la section **data**	Toutes les instructions et directives qui suivent concernent la section **data** dans le segment utilisateur.
`.stack`	Passage dans la section **stack**	Toutes les instructions et directives qui suivent concernent la section **stack** dans le segment utilisateur.
`.ktext`	Passage dans la section **ktext**	Toutes les instructions et directives qui suivent concernent la section **ktext** dans le segment noyau.
`.kdata`	Passage dans la section **kdata**	Toutes les instructions et directives qui suivent concernent la section **kdata** dans le segment noyau.
`.kstack`	Passage dans la section **kstack**	Toutes les instructions et directives qui suivent concernent la section **stack** dans le segment noyau.

Exemple 1:

```
     .data
vars:.word 5
     .word 10
     .text
__start:la $t0, vars
     lw $t1, 0($t0)
     lw $t2, 4($t0)
saut: bge $t1, $t2, exit
     move $a0, $t1
     li $v0, 1
     syscall
     addi $t1, $t1, 1
     j saut
exit: li $v0, 10
     syscall
```

On y trouve :

- Des mots clefs: .data, .word, .text
- Des instructions : lw, la, add, addi, bge ...
- Des registres : $t0, $t1, $t2
- Des étiquettes qui correspondent à des adresses : vars, saut,

b- Déclaration et initialisation de variables

Les directives suivantes permettent de d'initialiser certaines zones dans les sections **text** ou **data** de la mémoire.

Syntaxe	Action	Description
align num	Aligne le compteur d'adresse courant afin qu'expression bits de poids faible soient à zéro.	Cet opérateur aligne le compteur d'adresse sur une adresse telle que les n bits de poids faible soient à zéro. Cette opération est effectuée implicitement pour aligner correctement les instructions, demi-mots et mots.
.ascii *chaîne,* ***[chaîne,]*** **: : :**	Déclare et initialise une chaîne de caractères	Cet opérateur place à partir de l'adresse du compteur d'adresse correspondant à la section active la suite de caractères entre guillemets. S'il y a plusieurs chaînes, elles sont placées à la suite. Cette chaîne peut contenir des séquences d'échappement du langage C, et doit être terminée par un zéro binaire si elle est utilisée avec un appel système.
.asciiz *chaîne,* ***[chaîne,]*** **: : :**	Déclare et initialise une chaîne de caractères, en ajoutant un zéro binaire à la fin.	Cet opérateur est strictement identique au précédent, la seule différence étant qu'il ajoute un zéro binaire à la fin de chaque chaîne.
.byte *expression,* ***[expression,]*** **: : :**	Positionne des octets successifs aux valeurs des expressions.	La valeur de chacunes des expressions est tronquée à 8 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.
.half *expression,* ***[expression,]*** **: : :**	Positionne des demi-mots successifs aux valeurs des expressions.	La valeur de chacunes des expressions est tronquée à 16 bits, et les valeurs ainsi obtenues sont placées dans des adresses successives de la section active.
.word *expression,* ***[expression,]*** **: : :**	Positionne des mots successifs aux valeurs des expressions.	La valeur de chaque expression est placée dans des adresses successives de la section active.
.space *expression*	Reserve *expression* octets, et les mets à zéro	Un espace de taille *expression* octets est réservé à partir de l'adresse courante de la section active.

Exemples :

```
.align 2
.byte 12
.align 2
.byte 24
message: .ascii "Bonjour, Maître!\n\0"
message: .asciiz "Bonjour, Maître!\n"
```

```
table: .byte 1, 2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 1
coordonnées: .half 0 , 1024
             .half 968, 1024
entiers: .word -1, -1000, -100000, 1, 1000, 100000
nuls: .space 1024 ; initialise 1 kilo de mémoire à zéro
```

4.7.4. Conventions pour les appels de fonction

L'exécution de fonctions, nécessite une **pile en mémoire**. Cette pile correspond à la section **stack**. L'utilisation de cette pile fait l'objet de conventions qui doivent être respectées par la **fonction appelée** et par la **fonction appelante**.

- La **pile** s'étend vers les adresses décroissantes ;
- Le **pointeur de pile** pointe *toujours* sur la dernière case occupée dans la pile. Ceci signifie que toutes les cases d'adresses inférieures au pointeur de pile sont libres ;
- Le **R3000** ne **possède pas d'instructions spécifiques à la gestion de la pile**. On utilise les instructions **lw** et **sw** pour y accéder.

Les **appels de fonction**, utilisent un pointeur particulier, appelé **pointeur de pile**. Ce pointeur est stocké conventionnellement dans le registre **$29**. On le désigne aussi par la notation **$sp**. La valeur de retour d'une fonction est conventionnellement présente dans le registre **$2**.

Par ailleurs, l'architecture du processeur MIPS R3000 impose l'utilisation du registre **$31** pour stocker l'adresse de retour lors d'un appel de fonction (instructions de type **jal** ou **bgezal**).

À chaque **appel de fonction** est associée une zone dans la pile constituant le ' **contexte d'exécution '** de la fonction. Dans le cas des fonctions récursives, une même fonction peut être appelée plusieurs fois et possèdera donc plusieurs contextes d'exécution dans la pile. Lors de l'entrée dans une fonction, les registres **$5** à **$26** sont disponibles pour tout calcul dans cette fonction. Dans le cas général, un contexte d'exécution d'une fonction est constitué de quatre zones qui sont, dans l'ordre d'empilement :

1. ***La zone de sauvegarde* des registres de travail de la fonction appelante**. La fonction, qui a appelé la fonction courante, a dû sauvegarder les registres qu'elle utilise et qu'elle ne veut pas voir modifiés. Dans la suite, nous nommerons ces registres << persistants >> par opposition aux registres << temporaires >> dont la modification n'a pas d'influence dans la suite de la fonction appelante. Ces derniers n'ont pas à être sauvegardés. Par convention, on positionne toujours le registre persistant d'index le plus petit à l'adresse la plus petite ;
2. **La zone de sauvegarde de l'adresse de retour à la fonction appelante** (adresse à laquelle la fonction appelée doit revenir lors de sa terminaison) ;
3. **La zone des arguments de la fonction appelée**. Les valeurs des arguments sont écrites dans la pile par la fonction appelante et lues dans la pile par la fonction appelée. Par convention, on positionne toujours le premier argument de la fonction appelée à l'adresse la plus petite ;
4. **La zone des variables locales à la fonction appelé**.

Exemple 1 :
A partir des possibilités d'appels système sur les entrées-sorties, écrire un programme qui lit répétitivement un entier au clavier et l'ajoute aux entiers précédemment lus. Le programme s'arrête quand l'entier lu est 0 et affiche alors le message "Somme = ", suivi du résultat.

Solution :
Le programme assembleur est le suivant (**sans optimisations**) : **main** n'est pas une fonction comme les autres puisqu'elle n'est appelée par personne : On ne se préoccupe pas de sauver d'adresse de retour.

```
          .data
str1:     .asciiz "Entrez une suite d'entiers, terminez par 0 : \n"
str2:     .asciiz "Somme = "

          .text
main:     ori $v0, $zero, 4        # v0 <- 4
          la $a0, str1             # a0 <- str1
          syscall                  # "Entrez une suite d'entiers, terminez par 0 : \n"
          or $t0, $zero, $zero     # t0 <- 0
lecture:  ori $v0, $zero, 5        # v0 <- 5
          syscall                  # lecture
          add $t0, $v0, $t0        # t0 <- v0 + t0
          bne $v0, $zero, lecture  # if (vo != 0) goto lecture
          ori $v0, $zero, 4        # v0 <- 4
          la $a0, str2             # a0 <- str2
          syscall                  # "Somme = "
          or $a0, $zero, $t0       # a0 <- t0
          ori $v0, $zero, 1        # v0 <- 1
          syscall                  # affichage de la somme
          ori $v0, $zero, 10       # v0 <- 10
          syscall                  # return 0
```

Exemple 2 : Le programme C++ suivant permet d'afficher le contenu d'un tableau T :

```
#include <iostream>
#include <iomanip>

using namespace std;

int main (void)
{
  const unsigned N=5;
  unsigned T[]={1, 2, 3, 4, 5};

  cout << "T :\n";
  for (unsigned i = 0 ; (i < N) ; ++i)
    cout << setw(3) << T[i];

  return 0;
} // main()
```

En respectant la structure et la sémantique de ce programme, le traduire en assembleur MIPS en optimisant au mieux les échanges entre mémoire et registres, et en commentant chaque instruction en assembleur.

Solution :

```
                .data
N:              .word 5                 # unsigned N
T:              .word 1, 2, 3, 4, 5     # unsigned T[]
str:            .asciiz "T : \n"

                .text
main:           ori $v0, $zero, 4       # affichage d'une chaine
                la $a0, str             # $a0 <- @chaine
                syscall                 # "T : \n"

                la $t0, N               # $t0 <- @N
                lw $s0, 0($t0)          # $s0 <- N

                la $s1, T               # T:$s1 <- @T

                or $t0, $zero, $zero    # i:$t0 <- 0
for:            beq $t0, $s0, ExitFor   # if (i==N) goto exitfor
                sll $t1, $t0, 2         # $t1 <- i:$t0 * 4
                add $t1, $s1, $t1       # $t1 <- @T + deplacement physique
                lw $a0, 0($t1)          # $a0 <- T[i]
                ori $v0, $zero, 1       # affichage d'un entier
                syscall                 # affiche le contenu de $a0
                addi $t0, $t0, 1        # ++i
                j for                   # retour au debut de boucle

exitfor:        or $v0, $zero, 10       # return 0
                syscall
```

4.8. Conclusion

Ce document décrit l'architecture externe et interne ainsi que le jeu d'instructions du processeur MIPS R3000. Nous avons vu dans la partie architecture externe du processeur MIPS les **registres visibles du logiciel**, les règles d'adressage de la mémoire, le codage des instructions machine. Par contre dans la partie architecture interne, nous avons vu l'interface processeur mémoire et l'architecture interne du processeur qui se décompose en une partie opérative et une partie contrôle.

Et dans la partie de programmation en MIPS, nous avons présenté successivement l'organisation de la mémoire, les principales règles syntaxiques du langage, les directives acceptées par l'assembleur, les quelques appels système disponibles, ainsi que conventions imposées pour les appels de fonctions avec deux exemples.

En exposant ce chapitre, nous avons présenté les notions essentielles pour comprendre la programmation en processeur MIPS R3000. Dans le prochain et le dernier chapitre, nous allons voir les mécanismes de traitement des interruptions et exceptions.

Chapitre V :

Instructions spéciales

5.1. Introduction

Les instructions spéciales sont des évènements qui peuvent interrompre l'exécution d'un programme, comme les mécanismes de traitement des interruptions et des exceptions. Ce dernier chapitre s'intéresse à présenter les notions sur les interruptions plus précisément les entrées-sortir et les interruption systèmes.

5.2. Notions sur les interruptions

Il existe quatre types d'évènements qui peuvent interrompre l'exécution "***normale***" d'un programme :

- Les exceptions
- Les interruptions
- Les appels système (instructions **SYSCALL** et **BREAK**)
- Le signal **RESET**

Dans tous ces cas, le principe général consiste à passer la main à une procédure logicielle spécialisée qui s'exécute en **mode superviseur**, à qui il faut transmettre les informations minimales lui permettant de traiter le problème.

5.2.1 Exceptions

Les **exceptions** sont des évènements "***anormaux***", le plus souvent liés à une erreur de programmation, qui empêchent l'exécution correcte de l'instruction en cours. La **détection d'une exception** entraîne **l'arrêt immédiat de l'exécution de l'instruction** fautive. Ainsi, on assure que l'instruction fautive ne modifie pas la valeur d'un registre visible ou de la mémoire.

Les exceptions ne sont évidemment pas masquées. Il y a **7 types d'exception** dans cette version du processeur **R3000** :

ADEL	**Adresse illégale en lecture** : adresse non alignée ou se trouvant dans le segment système alors que le processeur est en mode utilisateur.
ADES	**Adresse illégale en écriture** : adresse non alignée ou accès à une donnée dans le segment système alors que le processeur est en mode utilisateur.
DBE	**Data bus erreur** : le système mémoire signale une erreur en activant le signal **BERR** à la suite d'un accès de donnée.
IBE	**Instruction bus erreur** : le système mémoire signale une erreur en activant le signal **BERR** à l'occasion d'une lecture instruction.
OVF	**Dépassement de capacité** : lors de l'exécution d'une instruction arithmétique (**ADD, ADDI ou SUB**), le résultat ne peut être représenté sur **32** bits.
RI	**Codop illégal** : le codop (Code opération) ne correspond à aucune instruction connue (il s'agit probablement d'un branchement dans une zone mémoire ne contenant pas du code exécutable.
CPU	**Coprocesseur inaccessible** : tentative d'exécution d'une instruction privilégiée (**MTC0, MFC0, RFE**) alors que le processeur est en mode utilisateur.

Le processeur doit alors passer en mode superviseur et se brancher au **gestionnaire d'exceptions** qui est une routine logicielle implantée conventionnellement à l'adresse "0x80000080".

Toutes les **exceptions** étant fatales dans cette version du processeur R3000, il n'est pas nécessaire de sauvegarder une adresse de retour car il n'y a pas de reprise de l'exécution du programme contenant l'instruction fautive. Le processeur doit cependant transmettre au **gestionnaire d'exceptions** l'adresse de l'instruction fautive et indiquer dans le registre de cause le type d'exception détectée.

Lorsqu'une exception est détectée, le processeur :

- Sauvegarde l'adresse de l'instruction fautive dans le registre **EPC**
- Sauvegarde l'ancienne valeur du registre d'état **SR**
- Passe en mode superviseur et masque les interruptions dans **SR**
- Écrit le type de l'exception dans le registre **CR**
- Branche à l'adresse "0x80000080".

5.2.2 Interruptions

Les **requêtes d'interruption** matérielles sont des évènements **asynchrones** provenant généralement de périphériques externes. Elles peuvent être masquées. Le processeur **MIPS** possède **6 lignes d'interruptions externes** qui peuvent être masquées globalement ou individuellement. L'activation d'une de ces lignes est une **requête d'interruption**.

Elles sont inconditionnellement écrites dans le registre **CR** et sont prises en compte à la fin de l'exécution de l'instruction en cours si elles ne sont pas masquées. Cette requête doit être maintenue active par le périphérique tant qu'elle n'a pas été prise en compte par le processeur.

Le processeur passe alors en **mode superviseur** et se branche ici encore au **gestionnaire d'exceptions**. Comme il faut reprendre l'exécution du programme en cours à la fin du traitement de l'interruption, il faut sauvegarder une adresse de retour.

Lorsqu'une **requête d'interruption non-masquée** est détectée, le processeur MIPS :

- Sauvegarde l'adresse de retour (PC + 4) dans le registre **EPC**
- Sauvegarde l'ancienne valeur du registre d'état **SR**
- Passe en mode superviseur et masque les interruptions dans **SR**
- Écrit qu'il s'agit d'une interruption dans le registre **CR**
- Branche à l'adresse "0x80000080".

En plus des **6 lignes d'interruption matérielles**, le processeur R3000 possède un mécanisme **d'interruption logicielle** : Il existe 2 bits dans le registre de cause **CR** qui peuvent être écrits par le logiciel au moyen de l'instruction privilégiée **MTC0**. La mise à **1** de ces bits **déclenche** le **même traitement** que les **requêtes d'interruptions externes**, s'ils ne sont pas masqués.

5.2.3 Signal RESET

Le processeur MIPS possède également une ligne **RESET** dont l'activation, pendant au moins un cycle, entraîne le branchement inconditionnel au logiciel d'initialisation.

Cette **requête** est très semblable à une **septième ligne d'interruption externe** avec les différences importantes suivantes :

- Elle n'est pas masquée.
- Il n'est pas nécessaire de sauvegarder une adresse de retour.
- Le gestionnaire de reset est implanté à l'adresse "0xBFC00000".

Dans ce cas, le processeur :

- Passe en mode superviseur et masque les interruptions dans **SR**
- Branche à l'adresse "0xBFC00000".

5.3. Entrées-sorties

Pour exécuter certaines fonctions système, typiquement les **entrées/sorties** (lire ou écrire un nombre, ou un caractère), il faut utiliser des **appels système**.

Par convention, le numéro de l'appel système est contenu dans le registre **$2** et son unique argument est dans le registre **$4**.

Cinq appels système sont actuellement supportés dans l'environnement de simulation :

a- **Ecrire un entier :** Il faut mettre l'entier à écrire dans le registre **$4** et exécuter l'appel système numéro 1.

Typiquement, on aura par **exemple** :

```
li $4, 1234567      ; met 1234567 dans l'argument
ori $2, $0, 1       ; code de 'print_integer'
syscall             ; affiche 1234567
```

b- **Lire un entier :** La valeur de retour d'une fonction — système ou autre — est positionnée dans le registre **$2**. Ainsi, lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre**$2**.

Exemple :

```
  ori $2, $0, 5       ; code de 'read_integer'
  syscall             ; $2 contient ce qui a été lu
```

c- **Ecrire une chaîne de caractères :** Une chaîne de caractères étant identifiée par un pointeur, il faut passer ce pointeur à l'appel système numéro 4 pour l'afficher.

Exemple :

```
str: .asciiz "Chaîne à afficher\n"
       la $4, str          ; charge le pointeur dans $4
       ori $2, $0, 4       ; code de 'print_string'
       syscall             ; affiche la chaine pointée
```

d- **Lire une chaîne de caractères :** Pour lire une chaîne de caractères, il faut un pointeur et une longueur maximum. Il faut passer ce pointeur dans **$4** et cette longueur dans **$5** et exécuter l'appel système numéro 8. Le résultat sera mis dans l'espace pointé.

Exemple :

```
read_str: .space 256
       la $4, read_str     ; charge le pointeur dans $4
       ori $5, $0, 255     ; charge longueur max dans $5
       ori $2, $0, 8       ; code de 'read_string'
       syscall             ; lit la chaîne dans l'espace
                           ; pointé par $4
```

e- **Quitter :** L'appel système numéro 10 effectue l'**exit** du programme au sens du langage C.

Exemple :

```
  ori $2, $0, 10 ; indique l'appel à exit
  syscall ; quitte pour de bon!
```

Exemple : Programme assembleur

```
.data                    # Des données suivent.
hello :                  # L ' adresse de la donnée qui suit.
.asciiz " hello world\n" # Chaîne terminée par un zéro.
.text                    # Des instructions suivent.
main:                    # Adresse de l ' instruction qui suit.
     li $v0, 4           # Code de l'appel print_string.
     la $a0, hello       # Adresse de la chaine.
     syscall             # Appel au noyau.
     jr $ra              # Fin de la procédure main.
```

5.4. Instructions systèmes

5.4.1. Appels Système : Instructions SYSCALL et BREAK

L'instruction **SYSCALL** permet à une tâche (utilisateur ou système) de demander un service au système d'exploitation, comme par exemple effectuer une entrée-sortie. Le code définissant le type de service demandé au système et un éventuel paramètre doivent avoir été préalablement rangés dans des registres généraux.

L'instruction **BREAK** est utilisée plus spécifiquement pour poser un point d'arrêt (dans un but de déverminage du logiciel) : on remplace brutalement une instruction du programme à déverminer par l'instruction **BREAK**.

Dans les deux cas, le processeur passe en mode superviseur et se branche au **gestionnaire d'exceptions.** Ces deux instructions sont exécutables en **mode utilisateur**. Elles effectuent les opérations suivantes :

- Sauvegarde de l'adresse de retour (PC + 4) dans le registre **EPC**
- Sauvegarde de l'ancienne valeur du registre d'état **SR**
- Passage en mode superviseur et masquage des interruptions dans **SR**
- Écriture de la cause du déroutement dans le registre **CR**
- Branchement à l'adresse "0x80000080".

5.4.2. Retour d'interruption

Avant de reprendre l'exécution d'un programme qui a effectué un appel système (instructions **SYSCALL** ou **BREAK**) ou qui a été interrompu, il est nécessaire d'exécuter l'instruction **RFE**.

Cette instruction effectue la restitution de l'état précédent dans le registre **SR**

5.5. Conclusion

Dans ce chapitre, nous avons vu qu'il existe quatre types d'évènements qui peuvent interrompre l'exécution d'un programme : Les **exceptions** provoquées par une erreur de programmation, les **requêtes d'interruption** matérielles, les **appels système** pour lire ou écrire un nombre ou un caractère et le signal **RESET** dont l'activation entraîne le branchement inconditionnel au logiciel d'initialisation.

Références bibliographies

Cours web

- **Adam J.M.,** 2015, *La gestion de la mémoire*, Université de Lausanne.
- **Beltrame G.,** 2018, *INF1600 : Architecture des micro-ordinateurs (Mémoires – Mémoire cache)*, Polytechnique Montréal.
- **Cazes A., Delacroix J.**, 2011, *Architecture des machines et des systèmes informatiques 4ème édition*, Collection : Informatique, Dunod.
- **Dumartin T.,** 2004-2005, *Architecture des ordinateur – Note de cours*, Informatique Industrielle.
- **Essaddouki M.,** 2005, *Généralités et algorithmique de base – Structure et fonctionnement d'un ordinateur, développement informatique.*
- **Errami A.,** 2010-2011, *Support du cours architecture des ordinateurs*, Sup'Technology.
- **Ghalouci L.**, 2015, *Architecture de l'ordinateur-Voyage au centre de votre unité centrale*, Université d'Oran des Sciences et de la Technologie - Mohamed Boudiaf.
- **Haymen S.** 2007, *Cours informatique* : 1ère année tronc commun.
- **Lazard E.**, 2011, *Architecture de l'ordinateur*, Université Paris-Dauphine.
- **Marcel P**., 2001, *Architecture des ordinateurs*, informatique & télécommunications 1ère année.
- **Mathieu P.,** 2009-2010, *Structure des ordinateurs*, Edition n°1, Haute ecole Louvain en Hainaut.
- **Merazgui A.**, 2004-2005, *Architecture des ordinateurs I*, Université Larbi Ben M'hidi Oum El-Bouaghi.
- **Montagny S.,** 2013, Architecture des ordinateurs, Université de Savoie.
- **Tanenbaum A. S.**, 2012, *Structured Computer Organization Fifth edition*, Pearson.
- **Viennent E.,** 1999-2000, *Architecture des ordinateurs*, IUT de Villetaneuse.
- **Vivien F.,** 2002, *Architecture des ordinateurs*, ENS Lyon.
- **Zanella P., Ligier Y., Lazard E.**, 2013, *Architecture et technologie des ordinateurs : Cours et exercices* 5ème édition - Collection : Sciences Sup, Dunod.

Sites web

- Cours de la Spécialité **N**umérique et **S**ciences **I**nformatiques – Thème6 : Architecture matérielles et systèmes d'exploitation, Consulter en 2020: http://portail.lyc-la-martiniere-diderot.ac-lyon.fr/srv1/co/Div_6_Archi_OS.html
- Cours Architecture de l'ordinateur, Consulter en 2019 : https://rmdiscala.developpez.com/cours/LesChapitres.html/Cours1/Chap1.5.htm#1.4
- Histoire des machines, Consulter en 2019 : http://aconit.inria.fr/omeka/exhibits/show/histoire-machines.1.html

Liens vers le microprocesseur MIPS R3000, Consulter en 2020 :

- ftp://132.227.86.9/pub/mips/mips.asm.pdf
- ftp://asim.lip6.fr/pub/mips/mips.externe.pdf
- ftp://asim.lip6.fr/pub/mips/mips.interne.pdf