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.

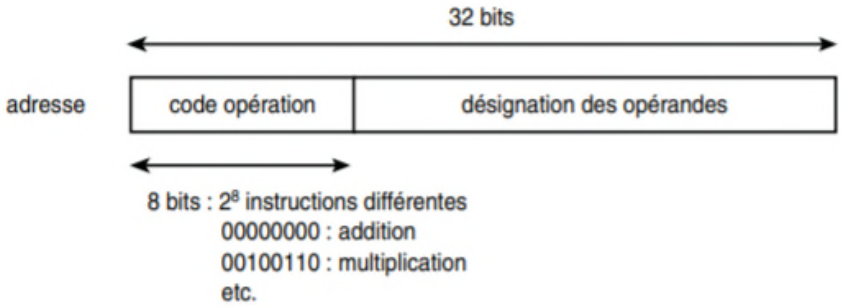


Figure 22 : L'instruction machine.

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.

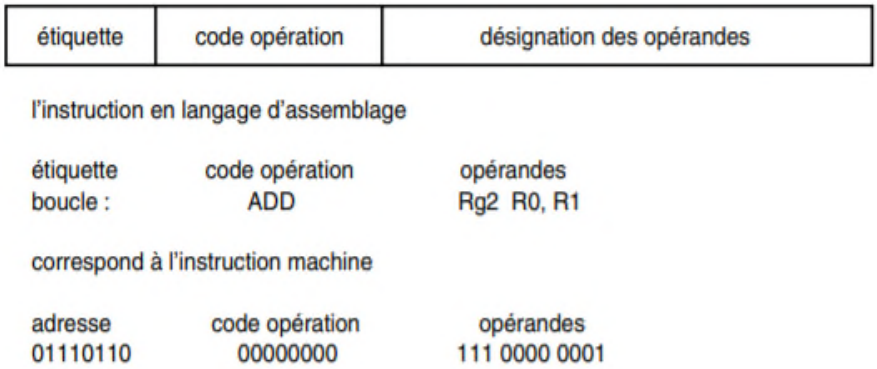


Figure 23 : L'instruction en langage d'assembleur.

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.

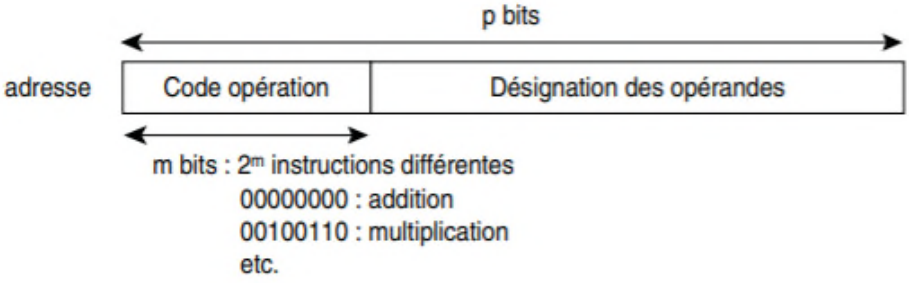


Figure 24 : Format général d'une instruction machine.

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**).

Remarque : Le tableau suivant regroupe les instructions conditionnelles de saut ainsi que la valeur des drapeaux correspondantes.

Instruction et synonymes	Signification (en anglais)	Condition nécessaire
JZ	Jump if Zero flag is set	Indicateur Zéro à 1
JNZ	Jump if No Zero flag is set	Indicateur Zéro à 0
JE	Jump if both operand were Equals	ou quand les deux opérandes A et B son égaux (A-B = 0)
JNE	Jump if both operand were Not Equals	ou quand les deux opérandes A et B ne son pas égaux (A-B != 0)
JS	Jump if Sign flag is set	Indicateur de signe à 1 (résultat négatif)
JNS	Jump if No Sign flag is set	Indicateur de signe à 0 (résultat positif)

Instruction et synonymes	Signification (en anglais)	Condition nécessaire
JO	Jump if Overflow flag is set	Indicateur de débordement à 1
JNO	Jump if No Overflow flag is set	Indicateur de débordement à 0
JNAE	Jump if Not Above or Equal	Indicateur de retenue à 1 (A < B, c'est à dire A-B < 0, retenue)
JAE	Jump if Above or Equal	Indicateur de retenue à 0 (A >= B, c'est à dire A-B >= 0, pas de retenue)
JNA	ump if Not Above	Indicateur de retenue à 1 ou indicateur Zéro à 1 (A < B ou A = B, c'est à dire A-B <= 0)
JA	Jump if Above	Indicateur de retenue à 0 et indicateur Zéro à 0 (A >= B et A != B, c'est à dire A > B, A-B > 0)

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).

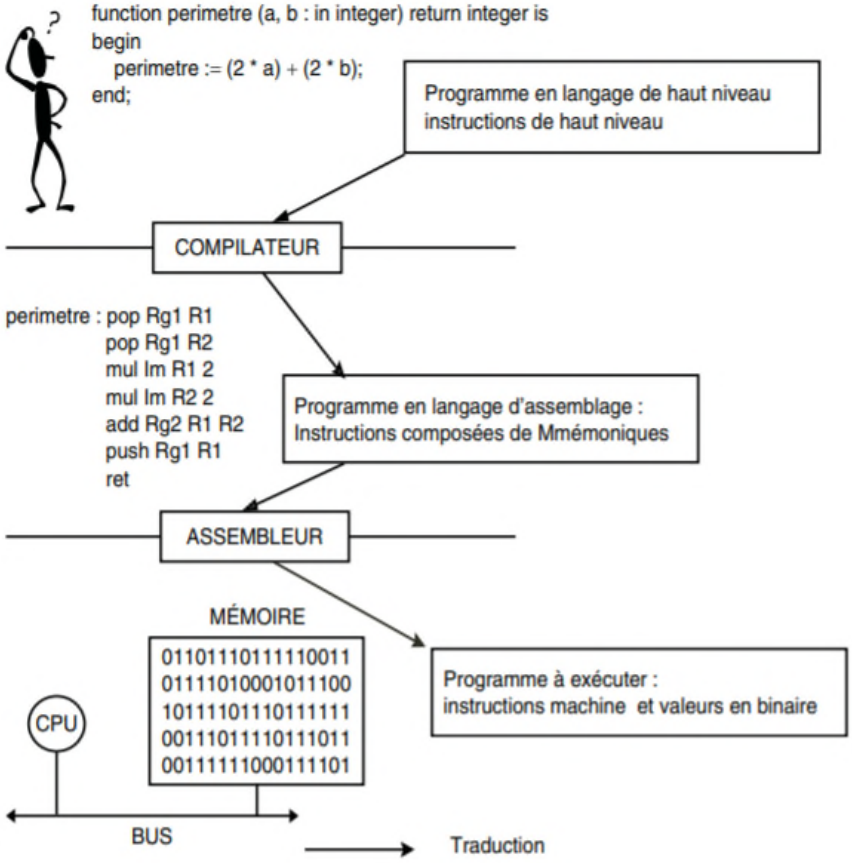


Figure 25 : Exemple de principe de compilation et d'assemblage

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).

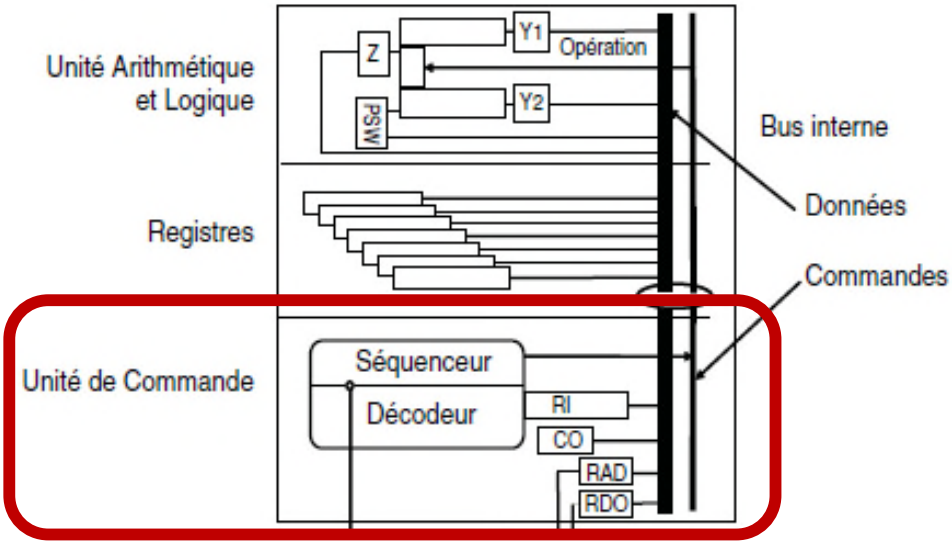


Figure 26: L'unité de commande.

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*

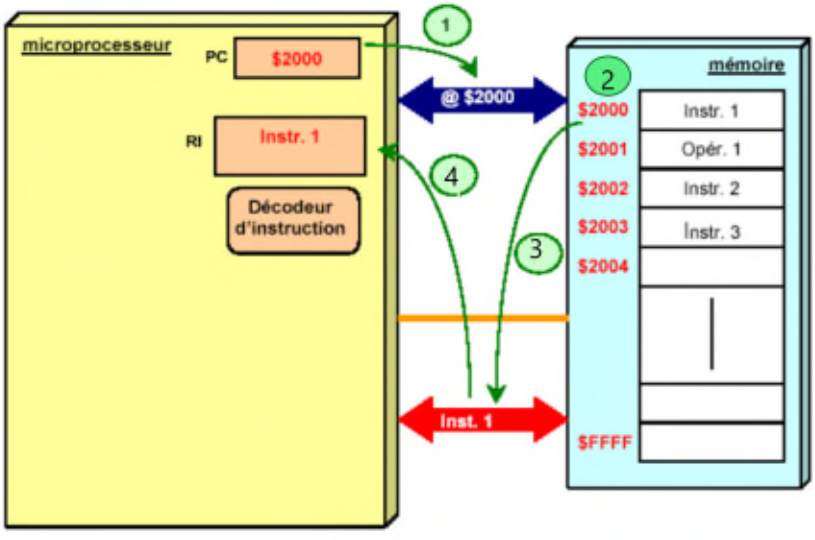


Figure 27 : La première phase d'exécution d'une instruction.

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

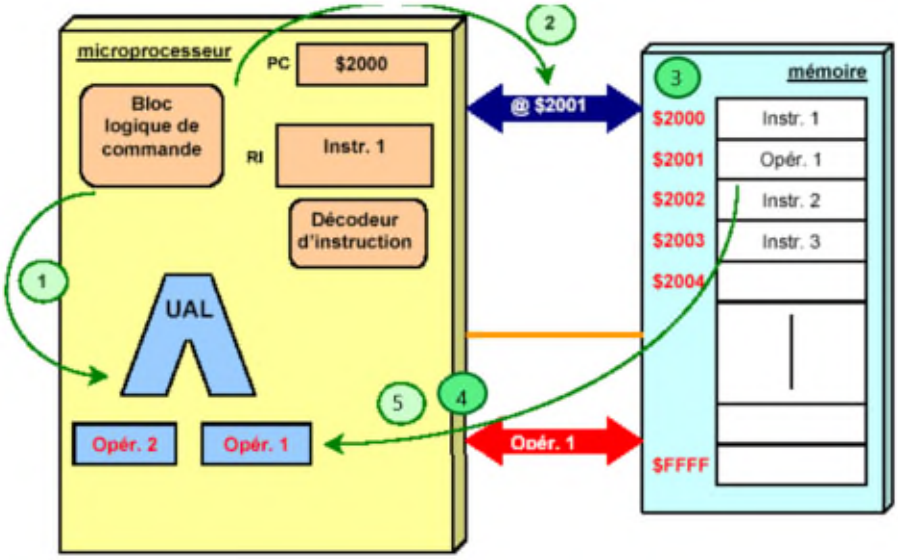


Figure 28 : La deuxième phase d'exécution d'une instruction.

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

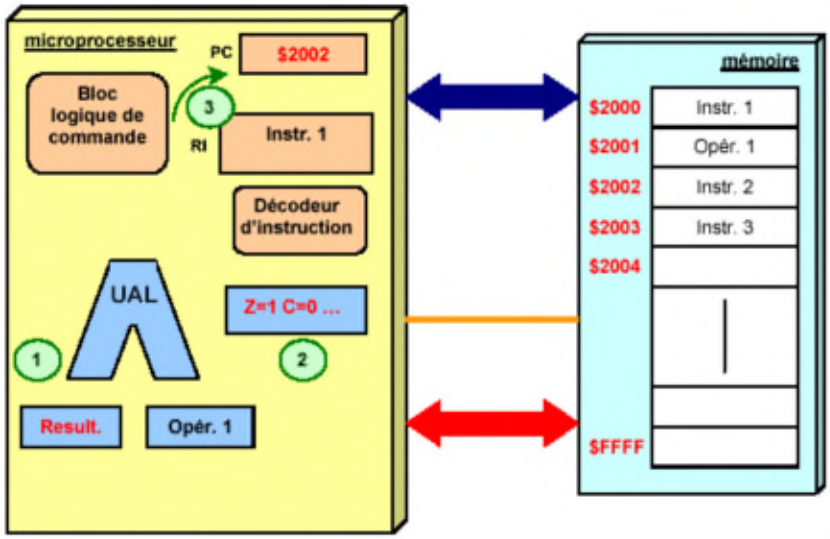


Figure 29 : La troisième phase d'exécution d'une instruction.

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.

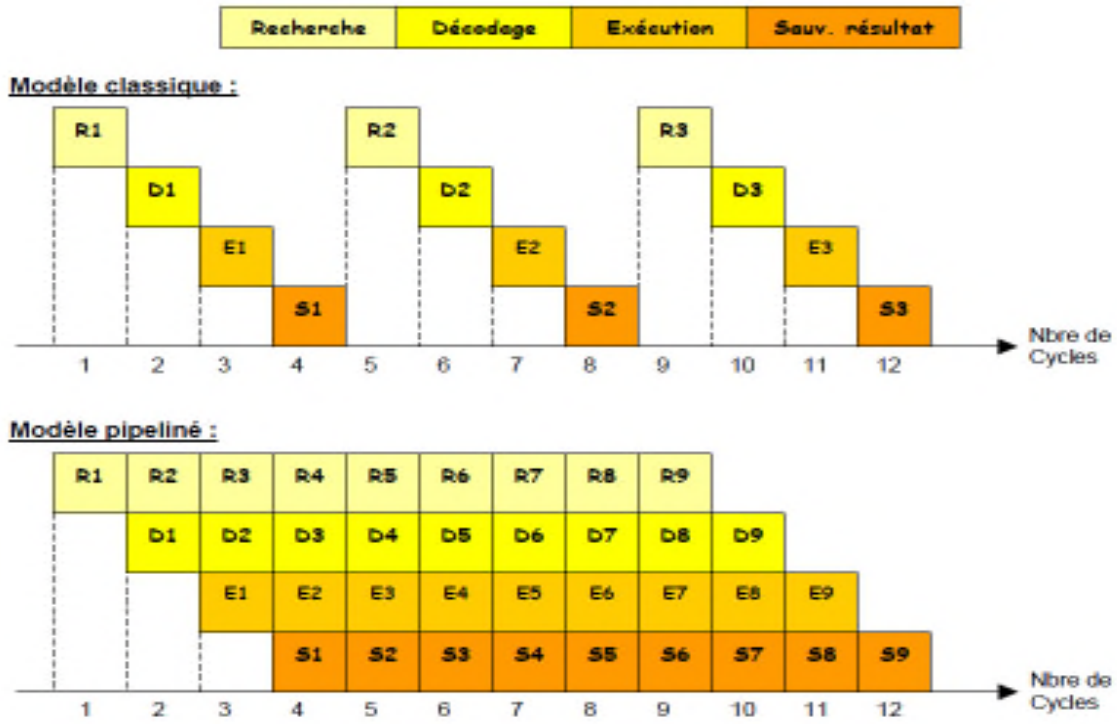
(RDM) → UAL

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**.

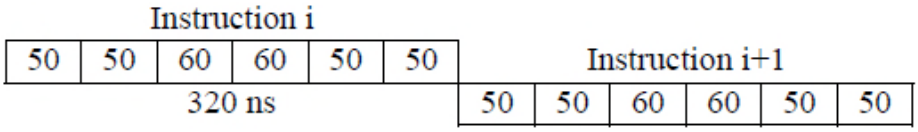
Exemple (de la figure précédant) : la machine débute l'exécution d'une instruction à chaque cycle et le pipeline est pleinement occupé à partir du quatrième cycle (Nombre de cycles = 4).

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 ?

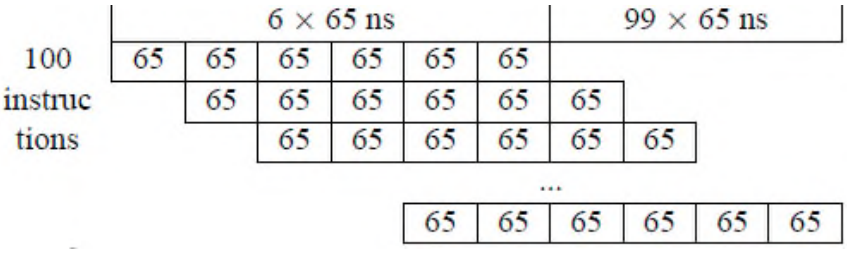


Temps d'exécution des instructions = **Nombre d'instruction * Temps d'exécution d'une instruction**, donc **Temps d'exécution de 100 instructions** = 100 * 32 = **32 000 ns.**

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 :

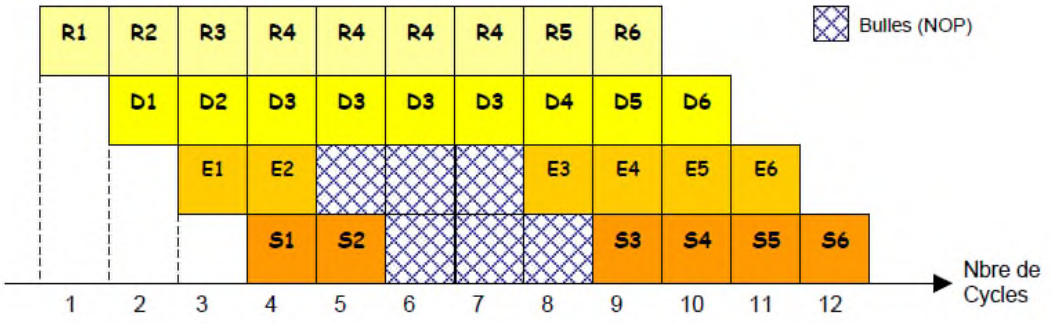
L'Athlon d'AMD comprend un pipeline de **11** étages.

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 :

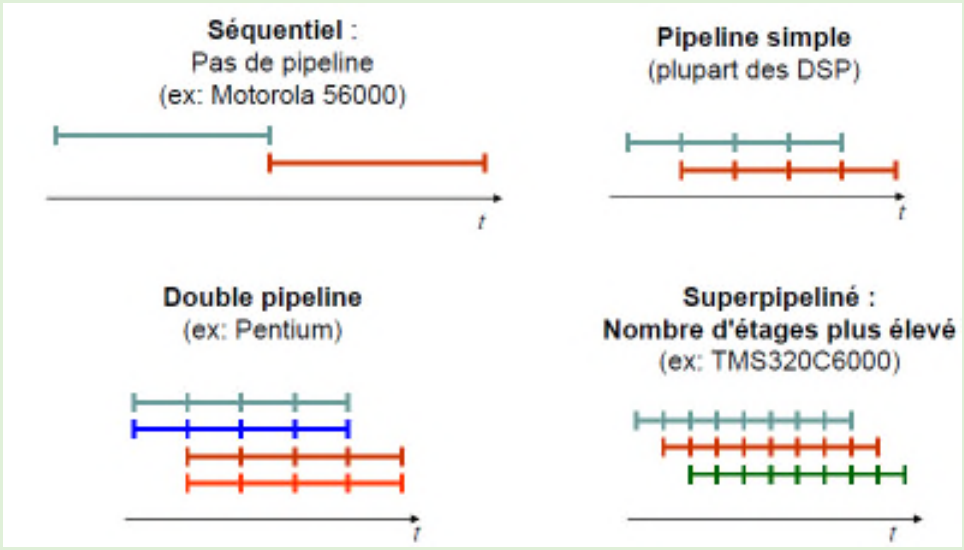
- *Avantages du pipeline* :

- 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.