DIAPOS

Architecture des ordinateurs

&

Programmation Assembleur

partie I

SMI S4

Pr. Najat Rafalia, Univ Ibn Tofail, Fac Sciences, Kénitra

Plan du cours

I. Introduction générale
II. Représentation des données
III. Architecture de base d'un ordinateur
IV. Introduction au microprocesseur
    1. Architecture interne
    2. Principe de fonctionnement
V. Introduction au langage machine
    1. Caractéristiques du langage machine
    2. Techniques de programmation en assembleur.
VI. L'assembleur 80x86
    1. L'assembleur
    2. Segmentation de la mémoire
    3. Modes d'adressage
    4. La pile
    5. Les procédures
VII. Les interruptions
    1. Présentation
    2. Interruption matériel sur PC
    3. Entrée/sorties par interruption

CHAPITRE 1 : Introduction

un ordinateur est une machine capable de résoudre des problèmes en appliquant des instructions préalablement définies.

La suite des instructions effectuées par l'ordinateur est appelée programme.

Les circuits électroniques de chaque ordinateur exécutent un nombre très limité d'instructions.

tout programme doit être converti avant son exécution.

L'ensemble des instructions exécutables directement par un ordinateur s'appèlent **langage machine (L1)**.

le langage machine dépend du processeur

le langage machine est donc très difficile à utiliser.

introduire un nouveau langage plus simple à utiliser que le langage machine : langage L2.

Introduire deux solutions permettant de convertir L2 en L1 :

a. compilation

b. interprétation

Le compilateur traduit le programme en L2 en un programme en L1

L'interprète examine chaque instruction du programme en L2 et l'interprète directement.

On peut concevoir toute une série de langages, de plus en plus pratiques à utiliser.

un ordinateur est conçu comme un empilement de couches ou de niveaux.

Les six couches de la plupart des ordinateurs actuels

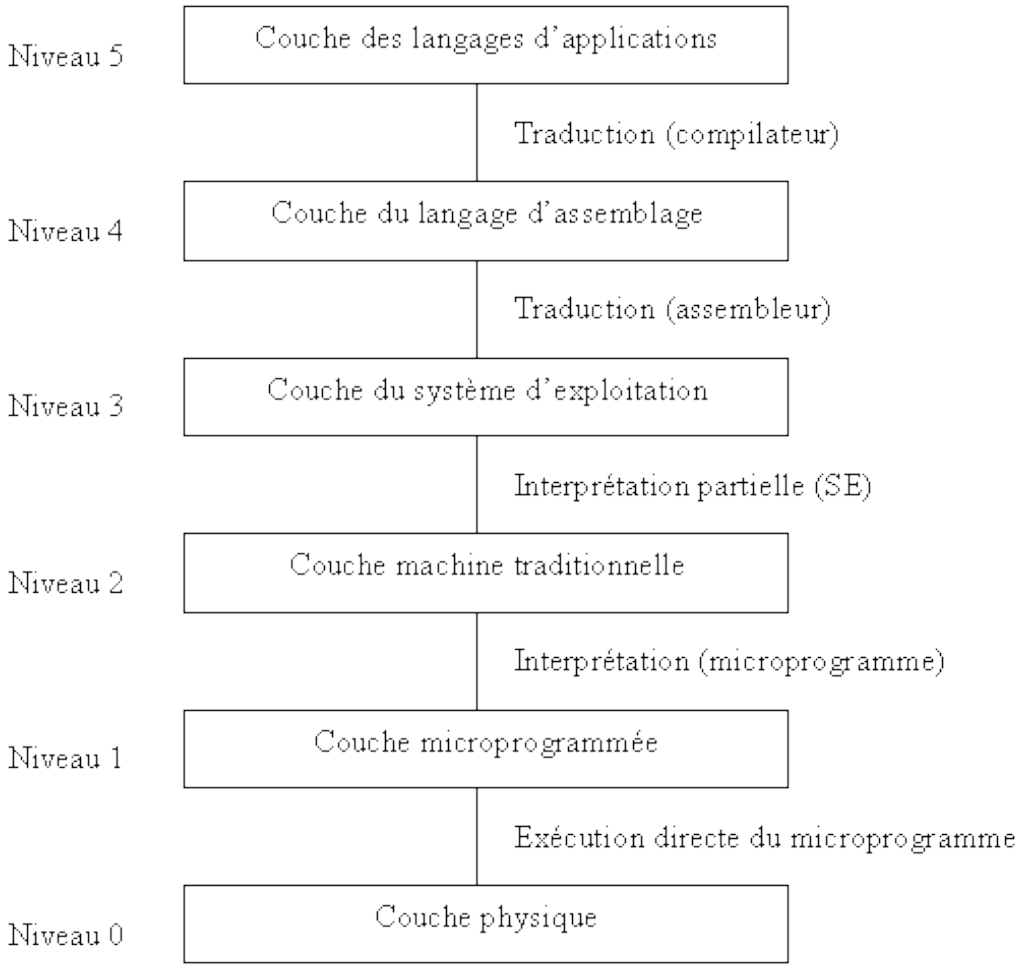


Figure I.1 Les six niveaux d'un ordinateur

1. Le niveau physique : Les objets manipulés sont les circuits composés de portes logiques.
2. Niveau micro-programmé : microprogramme, qui interprète les instructions de niveau 2.
3. Niveau machine conventionnel

4. Niveau du système d'exploitation : gestion des E/S, la mémoire, les fichiers, les processus
5. Niveau d'assemblage : une forme symbolique des langages sous-jacents de bas niveaux.

Niveau des langages d'applications : langages utilisés par les programmeurs (C, Pascal...).

CHAPITRE II : Architecture de base d'un ordinateur

II.1 Principe de fonctionnement

Un ordinateur est une machine de traitement de l'information.

Une information c'est tout ensemble de données :

- textes
- nombres
- sons
- images
- instructions composant un programme

Un programme est une suite d'instructions élémentaires exécutée dans l'ordre par un ordinateur.

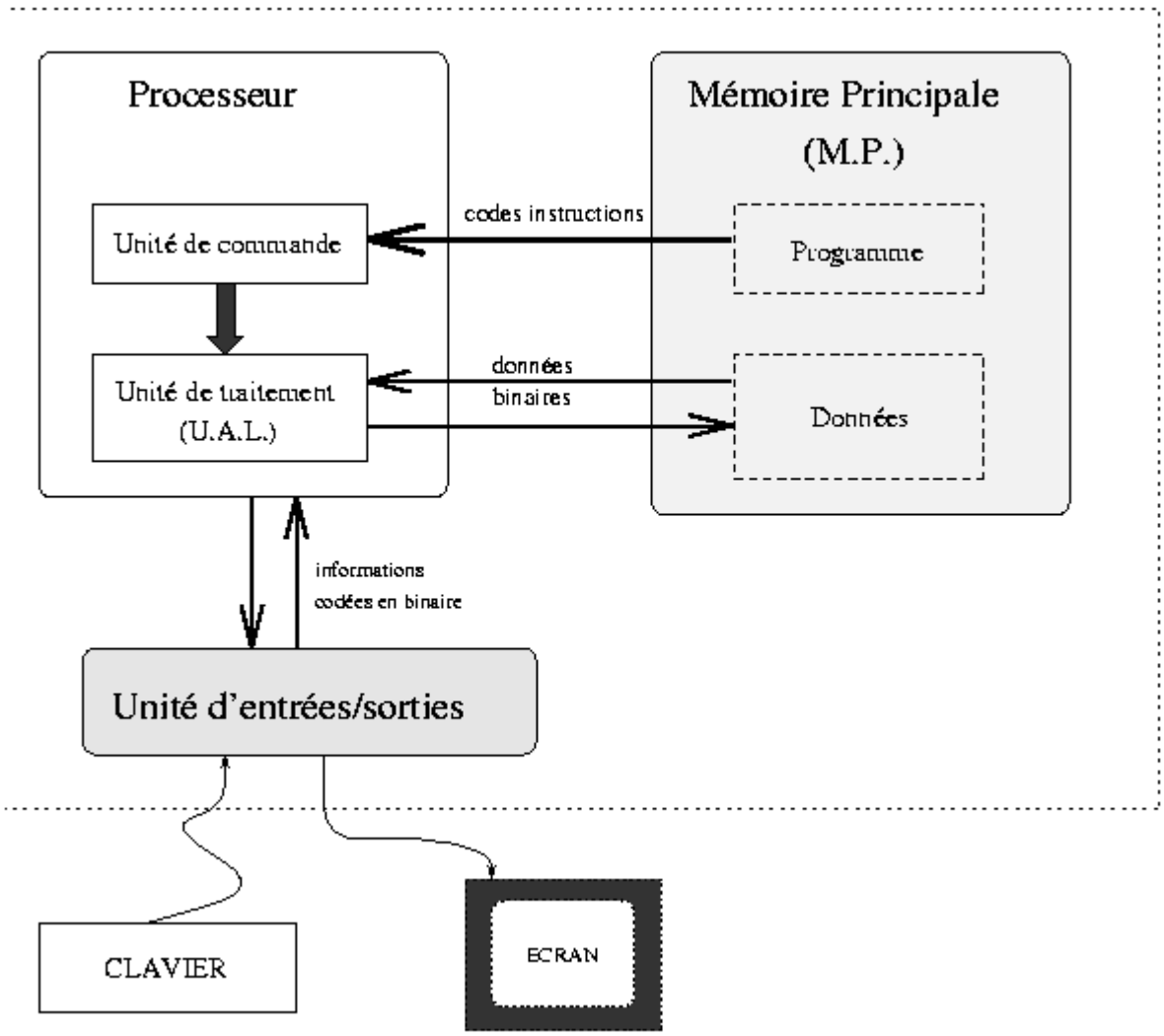


Figure II.1 Architecture schématique d'un ordinateur

La structuration d'un ordinateur est constituée de :

1. Mémoire principale
2. Processeur.

3. Interfaces d'E/S

Reliés entre eux par :

1. Un bus de données
2. Un bus d'adresses

3. Un bus de commandes

4. Signaux de contrôle

La mémoire principale (MP en abrégé) permet de stocker de l'information (programmes et données)

le processeur est un circuit électronique qui exécute dans l'ordre les instructions composant un programme.

Le processeur contrôle aussi les mémoires et les interfaces d'E/S.

Le processeur est composé de :

- l'unité de commande : lecture en mémoire et décodage des instructions;

- l'unité de traitement (*Unité Arithmétique et Logique* (U.A.L.)) : exécute les instructions qui manipulent les données.

L’unité de transfert (ou interfaces d’E/S) : dialogue entre le µp et les périphériques externes.

Les interfaces d’E/S sont indispensable pour les raisons suivantes :

- différence de vitesse de traitement entre µp et les périphériques externes.
- Diversité des périphériques externes

périphériques externes :

- Consol de visualisation
- Imprimante
- Disque dur
- Disquette

II.2 La mémoire principale

La mémoire stocke les programmes et les données.

Il existe plusieurs types de mémoires :

- disque dur

- RAM : Random Acces Memory (Mémoire à accès aléatoire)
- ROM : Read Only Memory (mémoire à lecture
- registres…

la mémoire est un ensemble de **bits**.

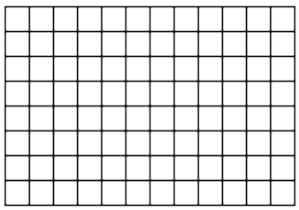


Figure II.2 Représentation d'une mémoire comportant 96 bits

Un bit peut être égal à 0 ou à 1

Toutes les données stockées dans la mémoire
sont codées à l'aide de suites de 0 et de 1.

Les adresses mémoire

Les adresses permettent à un programme d'accéder facilement aux données.

Pour adresser une mémoire, on la découpe en cellules formées de plusieurs bits.

Toutes les cellules d'une mémoire comportent le même nombre de bits.

On attribue ensuite un numéro à chacune de ces cellules : adresse

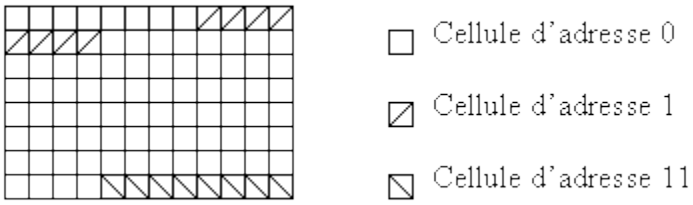


Figure II.3 Exemple d'adressage d'une mémoire de 96 bits en cellules de 8 bits

La taille de cellules varie d'un ordinateur à l'autre.

Les tailles de cellules les plus rencontrées sont 8 octet), 16 (mot) et 32 (double mot).

1024 octets = 1Ko.
1 Mo = 1024 Ko
1 Go = 1024 Mo

Opérations sur la mémoire

Seul le processeur peut modifier l'état de la mémoire.

Chaque emplacement mémoire conserve son contenu jusqu'à coupure de l'alimentation électrique.

Les mémoires externes (disquettes et disques durs) gardent leur contenus.

Les seules opérations possibles sur la mémoire sont :

- *écriture* dans un emplacement
- *lecture* d'un emplacement

II.3 Liaisons Processeur-Mémoire : les bus

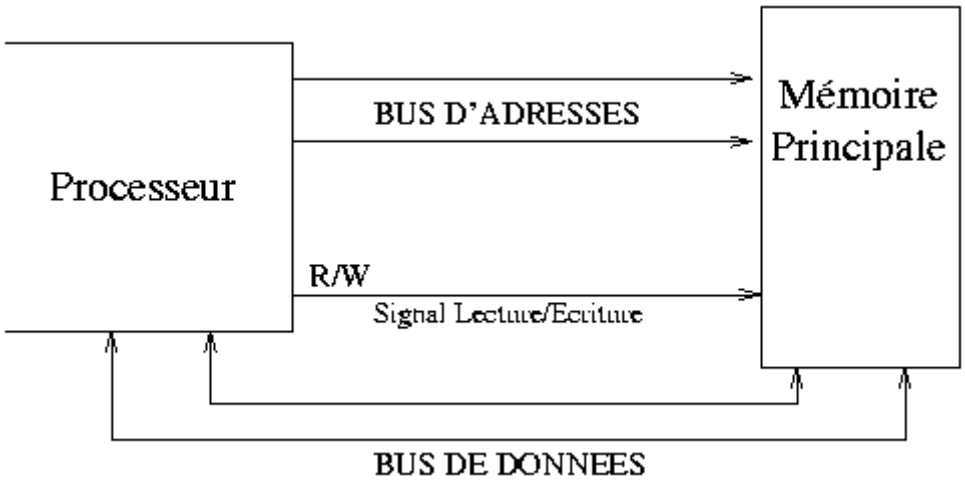


Figure II.4 Connexions Processeur-Mémoire : bus de données, bus d'adresse et signal lecture/écriture.

Les informations échangées entre la mémoire et le processeur circulent sur des *bus* .

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

Bus d’adresses

C’est un ensemble de broches qui permet au microprocesseur d’adresser les différentes cases mémoire et les interfaces d’E/S.

Ce bus est unidirectionnel (du microprocesseur vers la mémoire centrale ou les interfaces E/S).

Chaque case mémoire est repérée par une adresse et stocke une information binaire.

n broches d’adresses permettent l’adressage de 2^n cases mémoires.

Exemple :

1 µp de 8 bits et un bus d’adresses de 16 broches permet de d’adresser 2^{16} cases mémoire soit 65535 bits càd 64 ko.

Bus de données

C’est un ensemble de broches par lesquelles transitent les instructions et les données.

Le bus de données est bidirectionnel.

Le bus de données correspond à la capacité de traitement et de codage d'un microprocesseur.

Exemple : 1 microprocesseur de 8 bits $\Leftrightarrow$ un bus de données de 8 broches.

1 microprocesseur de 16 bits $\Leftrightarrow$ un bus de données de 16 broches.

CHAPITRE IV : Introduction au microprocesseur

IV.1 Architecture interne d'un microprocesseur

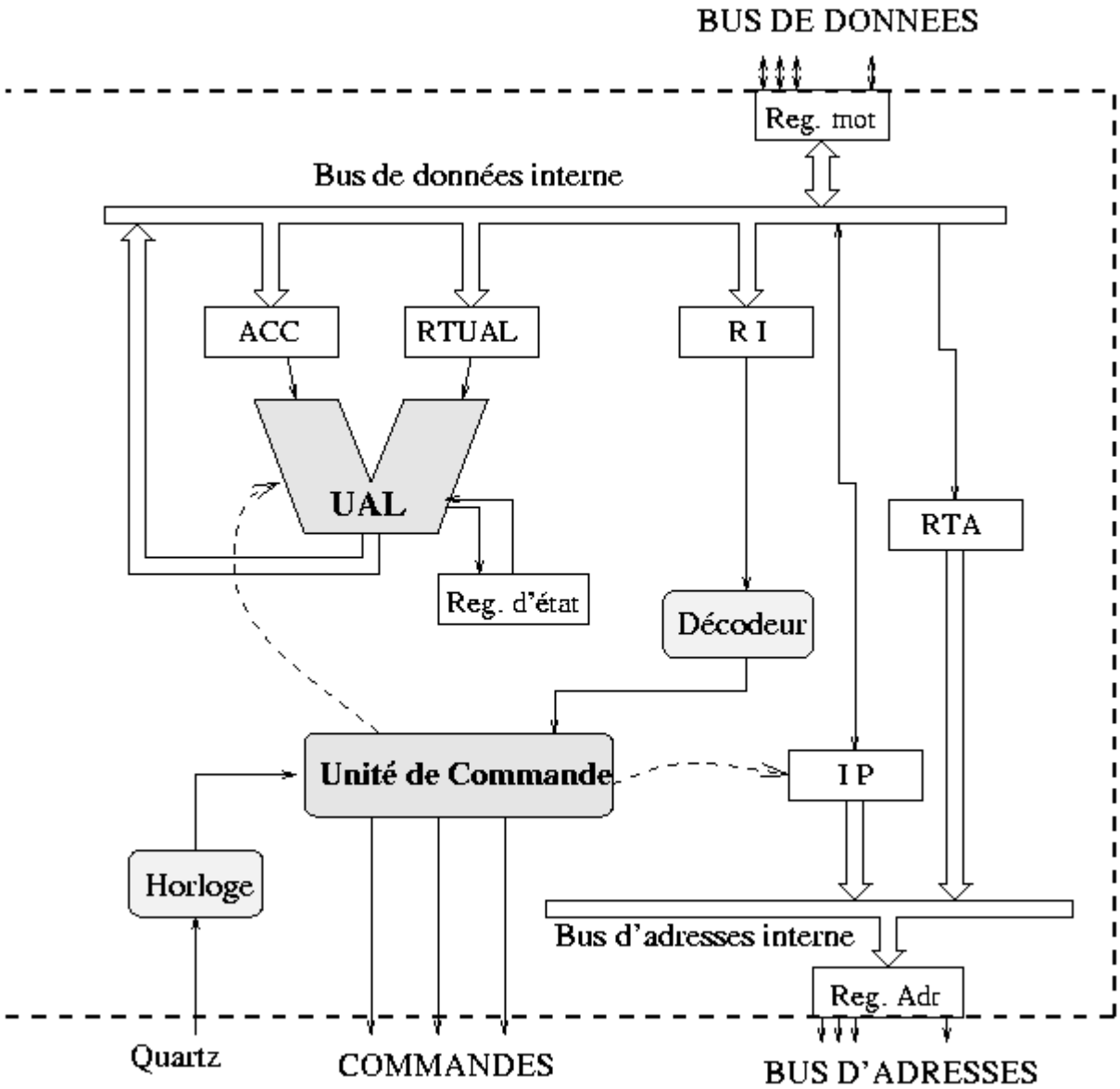


Figure VI.1 Schéma simplifié d'un processeur.

La figure représente l'architecture interne simplifiée d'un microprocesseur, elle comporte:

- Les regitres
- L'*unité de commande*,
- L'*UAL*, les registres
- Le *décodeur* d'instructions, qui, à partir du code de l'instruction lu en mémoire actionne la partie de l'unité de commande nécessaire.

Les informations circulent à l'intérieur du processeur sur deux *bus internes*, l'un pour les données, l'autre pour les instructions.

Le µp est relié à l'extérieur par les bus de données et d'adresses, le signal d'horloge et les signaux de commandes.

IV.1.1 Les registres

Le processeur utilise toujours des *registres*.

Ce sont des petites mémoires internes d'accès très rapides.
Elles sont utilisées pour stocker temporairement une donnée, une instruction ou une adresse.

Chaque registre stocke 8, 16 ou 32 bits.

Parmi les registres, le plus important est le registre *accumulateur*.

L'accumulateur est utilisé pour stocker les résultats des opérations arithmétiques et logiques.

Un accumulateur est un registre lié d'un coté à un registre temporaire et au bus interne de données et de l'autre coté à la sortie de l'UAL.

Déroulement d'une opération traitant deux données :

1. Une donnée sera stockée tout d'abord dans l'accumulateur
2. Cette donnée passera en suite au premier registre temporaire
3. La deuxième donnée passera directement au deuxième registre temporaire.

4. Le résultat sera stocké dans l'accumulateur par l'intermédiaire de la sortie de l'UAL.

Exemple : addition de deux données : D1 et D2

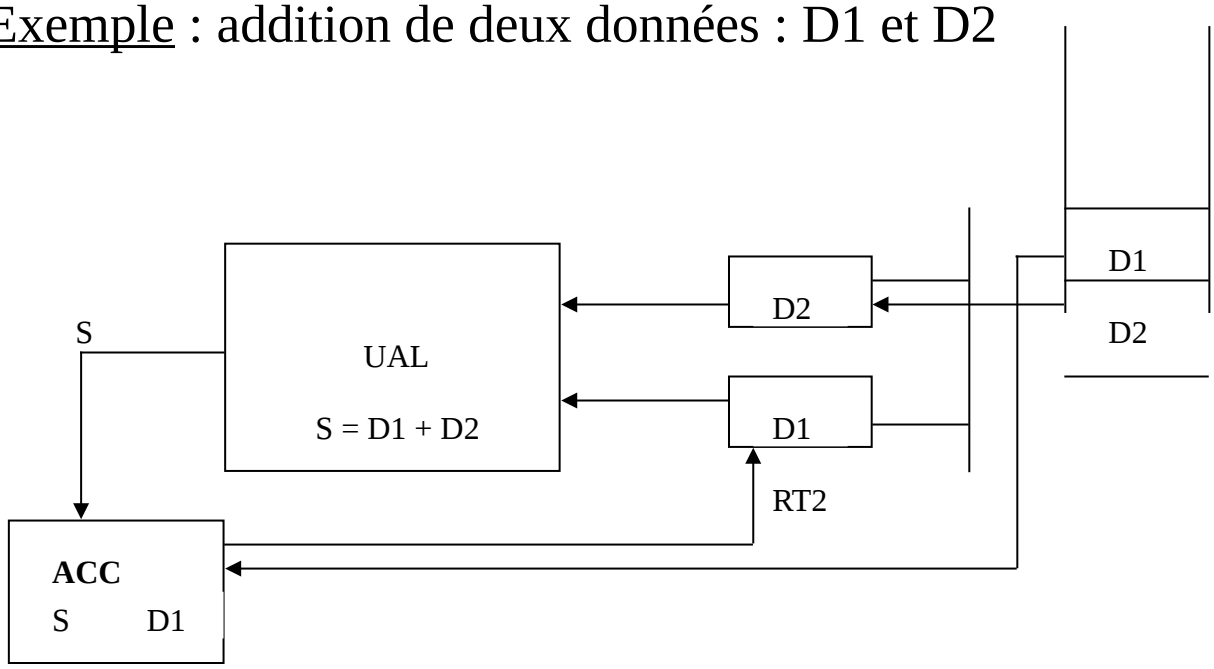
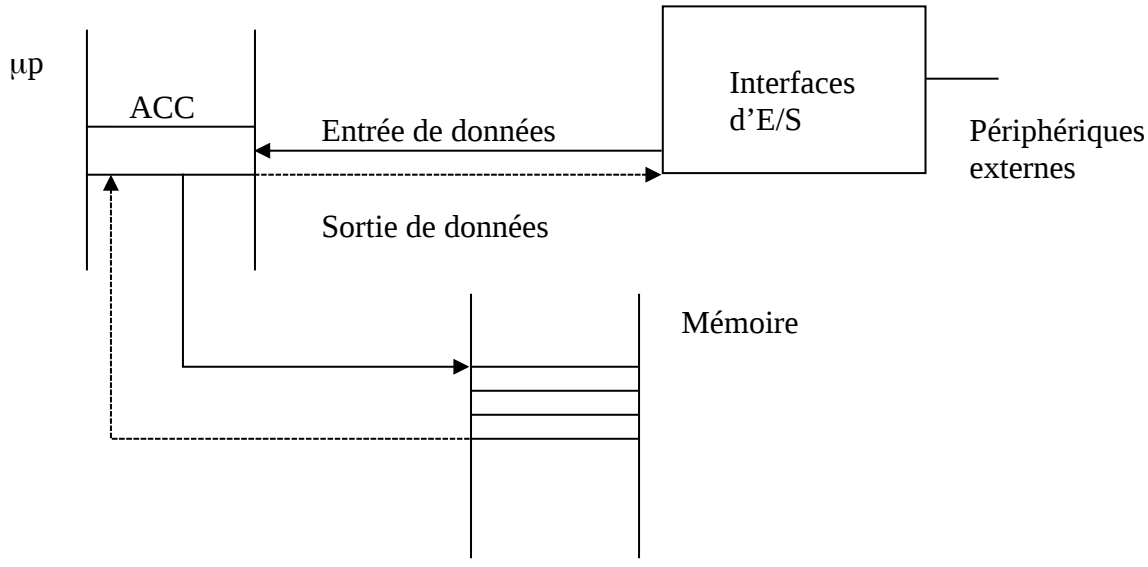


Figure IV.4 Addition de deux données

Un accumulateur représente également une intermédiaire entre la mémoire et les interfaces d'E/S



Le nombre de bits de l'ACC est égal en général au nombre de broches du bus de données.

Le nombre d'accumulateur varie d'un µp à l'autre.

Exemple :

première génération : 8088, 8085 et 6800 ce sont des µp de 8 bit ils possèdent 1 accumulateur de 8 bits.

6809 et 8088 sont appelés des µps faux 16 bits :

- ce sont des µps de 8 bits ;
- Ils possèdent deux accumulateur de 8 bits qui peuvent être regroupés pour former un seul accumulateur de 16 bits.

Le 8086, 80286, 68000 sont de vrais 16 bits : Ce sont des µps de 16 bits avec 1 accumulateur de 16 bits.

Les µps vrais 16 bits sont plus rapides que les µps faux 16 bits.

L'accumulateur intervient dans une proportion importante des instructions.

Exemple : Exécution d’une instruction comme ``*Ajouter 5 au contenu de la case mémoire d'adresse 180*" :

1. Le processeur lit et décode l'instruction;
2. le processeur demande à la mémoire le contenu de l'emplacement 180;
3. la valeur lue est rangée dans l'accumulateur;
4. l'unité de traitement (UAL) ajoute 5 au contenu de l'accumulateur;
5. le contenu de l'accumulateur est écris en mémoire à l'adresse 180.

C'est l'unité de commande qui déclenche chacune de ces actions dans l'ordre.

L'addition proprement dite est effectuée par l'UAL.

Il existe aussi six registres fondamentaux qu’on trouve dans tous les µp :

- Compteur d’instructions ou compteur ordinal
- Registre d’adresse
- Registre d’instruction
- Registre de données (accumulateur)
- Registre d’Etat

- Registre temporaire

Selon le type du µp utilisé on peut trouver en plus de ces registres d'autres registres pour faciliter la programmation d'un µp.

IV.1.1.1 Compteur ordinal (CO)

Format d'une instruction

Chaque instruction est représentée généralement par deux champ :

code opération : spécifie au µp la nature de l'opération effectuée.

Cette opération peut être une +, -, *, un transfert de données, ….

Le code opération est codé sur une cellule mémoire.

Code opérande : indique au µp les données qui vont être traitées par le code opération.

La taille du code opérande dépend de la nature des données mises en jeu.

Un certain nombre de cellules mémoire est nécessaire pour stocker le code opérande.

La structure d'un programme stocké en mémoire a la forme suivante :

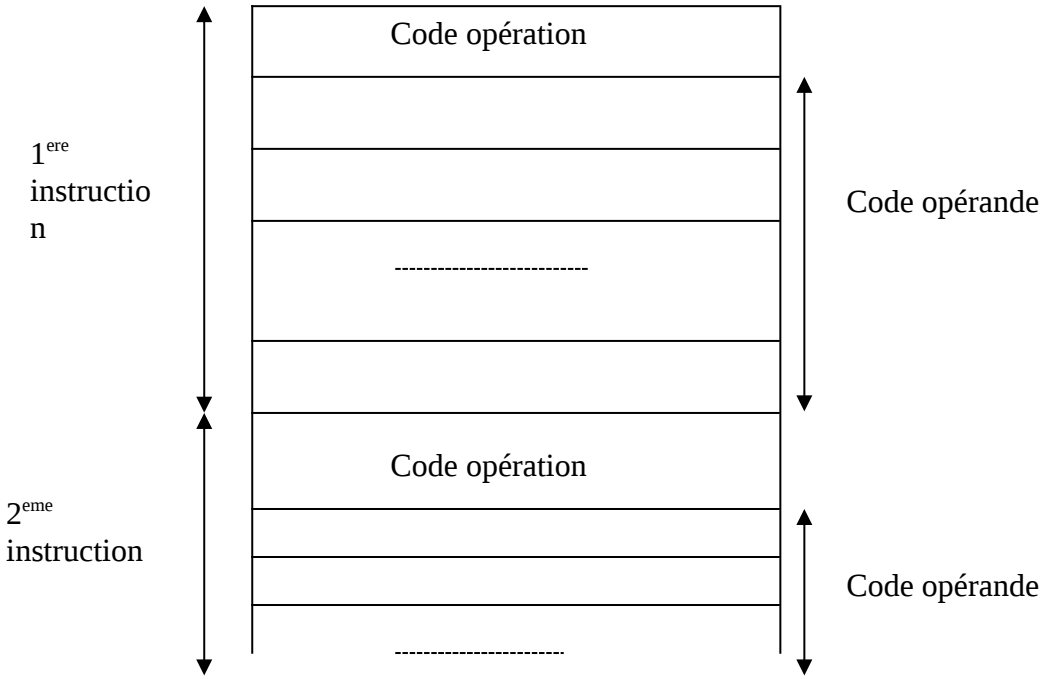


Figure IV.2 structure d'un programme en mémoire

Le compteur ordinal est un registre qui permet de localiser les instructions stockées en mémoire.

Le CO est chargé au début de l'exécution d'un programme à partir de l'adresse de la première case mémoire où se trouve le premier code opération de la première instruction.

L'exécution d'un programme s'effectue d'une façon séquentielle.

le compteur ordinal s'incrémente automatiquement pour passer de l'adresse d'une case mémoire à l'adresse suivante.

Le nombre de bits de ce registre est égal au nombre de broches du registre d'adresse.

IV.1.1.2 registre d'adresse (RA)

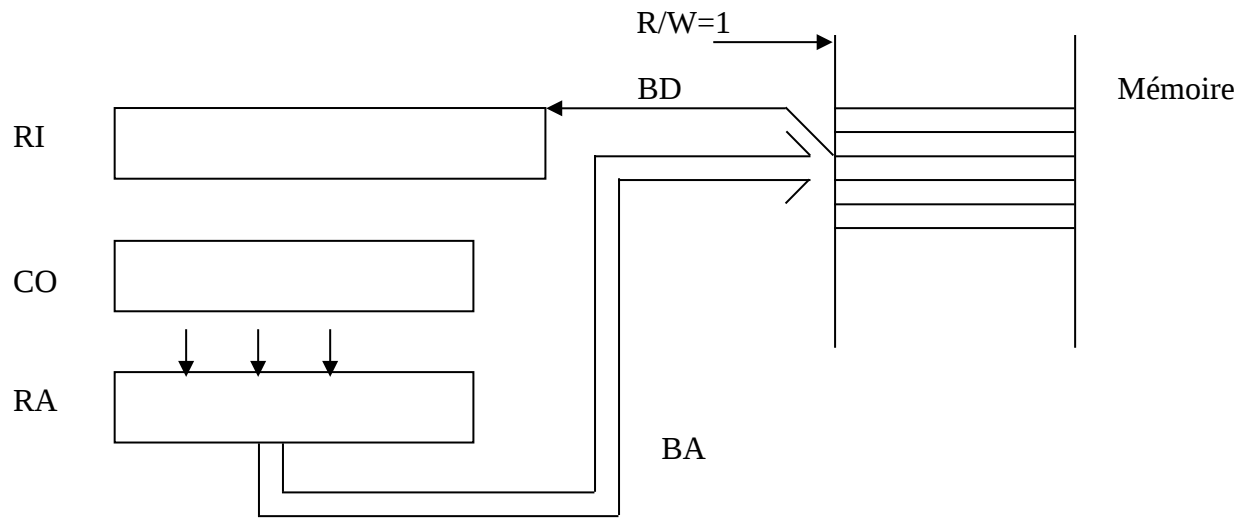
Le RA permet d'accéder à une donnée en mémoire.

Le RA a le même nombre de bits que le compteur ordinal.

IV.1.1.3 registre d'instruction (RI)

Le RI permet de stocker l'instruction à exécuter.

Le RI garde l'instruction pendant le temps nécessaire à son décodage et à son exécution.



Le compteur ordinal communique avec le RA pour lui donner l'adresse de la première instruction.

Le RA donne cette adresse au bus d'adresse qui va localiser cette adresse.

La mémoire reçoit un signal (la ligne R/W =1) de la part de la logique de contrôle pour l'informer qu'il s'agit d'une lecture.

Le bus de données va lire l'instruction et la mettre dans le registre d'instruction.

IV.1.1.4 registres temporaires

L'UAL possède deux entrées pour recevoir les données à traiter.

Au niveau de chaque entrée il y a un registre temporaire qui garde la donnée pendant le temps nécessaire à son traitement.

Exemple : traitement de deux données

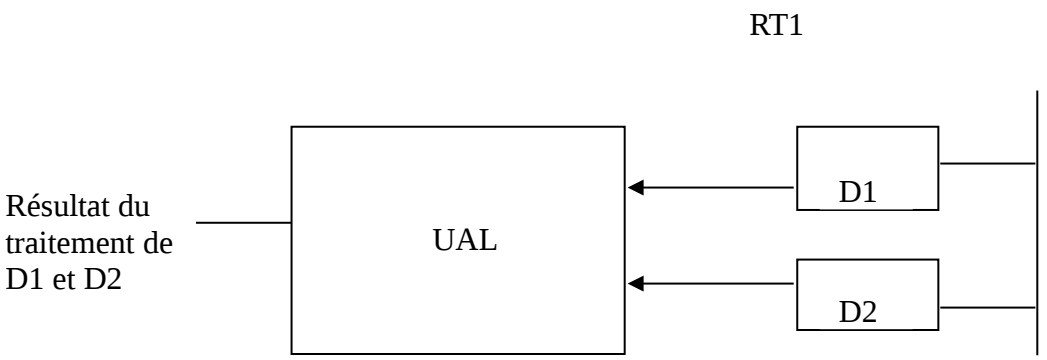


Figure IV.3 traitement de deux do par l'UAL

L'UAL est un circuit électronique qui ne peut pas stocker des données, elle se sert donc des registres temporaires RT1 et RT2.

IV.1.1.5 Registre d'Etat

Le registre d'Etat est directement lié à l'UAL.

Le registre d'Etat stocke certaines informations particulières concernant les opérations effectuées.

Le nombre de bits de ce registre varie d'un µp à l'autre.

Les bits les plus utilisés sont :

OF SF ZF CF

Bit de retenu ou indicateur de retenu (CF)

Si CF = 1 alors une opération d'addition ou soustraction a généré une retenue,

Si CF = 0 alors pas de retenue

Exemples

a.

10101111
00001010
10101001

CF=0 pas de retenue

b.
11111111 (255)
00000001 (1)
00000000

CF=1 il y a une retenue, ce 1 de retenue sera stocker dans le bit de retenue CF.

Bit de zéro ZF

Si ZF = 0 alors le résultat d'une opération est nul.
Si ZF = 1 alors le résultat d'une opération est non nul.

Bit de signe (SF)

Si SF = 1 alors le bit le plus significatif d'une donnée est égal à 1

Si SF = 0 alors le bit le plus significatif d'une donnée est égal à 0

Exemple : en complément à 1 ou à 2 la donnée 10000001 implique SF = 1.

Bit de débordement OF

Si OF = 1 alors il y a dépassement de la capacité maximale de codage.

Si OF = 0 alors il n'y a dépassement de la capacité maximale de codage.

IV.1.2 L'unité de traitement (UAL)

Le traitement effectué par l'unité arithmétique et logique sont +, -, *, /, le et logique, le ou logique, le ou exclusif.

Exemples :

a. le et logique (AND)

Soient N1 = 00101111 et N2 = 10100001
N3 = N1 AND N2 = 00100001

Table logique de AND :

A	B	A AND B
0	0	0
1	0	0
0	1	0
1	1	1

b. le ou logique (OR)

Table logique de OR :

A	B	A OR B
0	0	0
0	1	1

1 0 1
1 1 1

c. le ou exclusif (EOR)

table logique de EOR

A	B	A EOR B
0	0	0
0	1	1
1	0	1
1	1	0

IV.1.3 Unité de commande et de contrôle

L'unité de commande est le maître d'œuvre des différentes opérations de transfert de données.

L'unité de commande a pour fonction :

- Décodage des instructions stockées dans le registre d'instructions (nature de l'opération ou l'instruction à exécuter).
- Exécution des instructions par les organes exécutifs (UAL, Accumulateur, CO, …)

- Gestion du dialogue entre le µp, les mémoires et les interfaces d'E /S, par l'intermédiaire du bus de commande et des signaux de contrôle.

Les signaux de commandes permettent au processeur de communiquer avec les autres circuits de l'ordinateur.

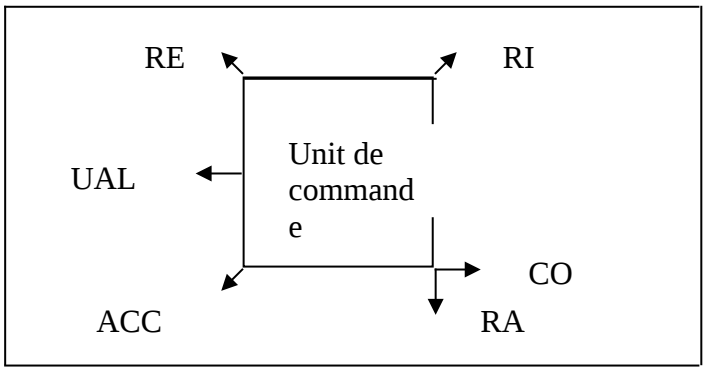
On trouve en particulier :

1. le signal R/W (Read/Write) : le µp indique à la mémoire principale s'il effectue un accès en lecture ou en écriture.

2. l'initialisation des registres internes du µp : la broche d'initialisation fait signal au µp de faire tous les registres à zéro.

3. l'interruption d'exécution du µp.

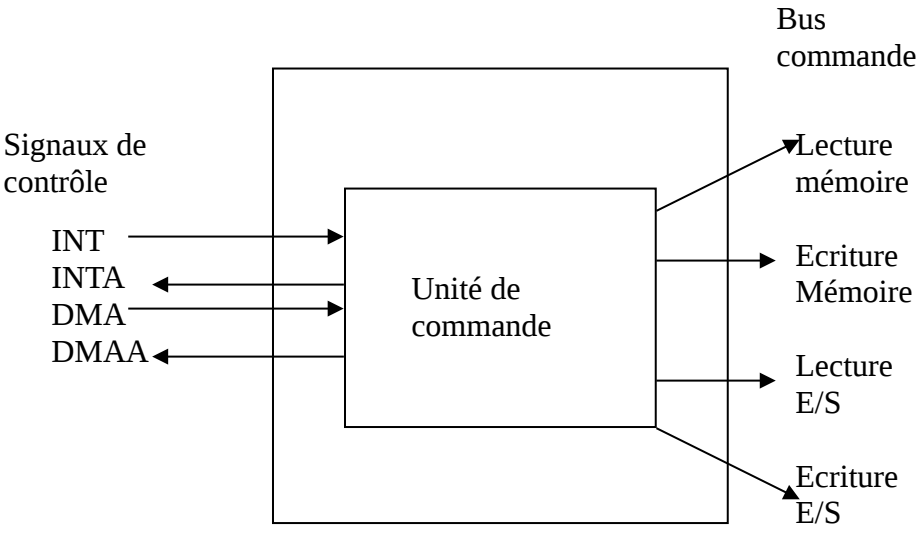
Le nombre de broches de ces signaux change d'un constructeur à un autre.



IV.1.3.1 Exemple de gestion interne

- Incrémentation du CO
- Mise à jour des bits du registre d'Etat en fonction des résultats fournis par l'UAL
- Nature de l'opération qui va être effectuée par l'UAL

IV.1.3.2 Exemple de gestion externe (dialogue)



- INT : interruption d'exécution d'un programme.

  INT est générée par une interface d'E/S

Si INT = 1 alors l'IE/S désire communiquer avec le µp.

Si INT = 0 pas de dialogue entre E/S et µp

- INTA : Réponse par le µp au signale INT

  Si INTA = 1 alors le µp a reçu le signal INT

  Si INTA = 0 alors le µp n'a pas reçu le signal INT

- DMA : Direct Memory Acces

  Si DMA = 1 alors une interface d'E/S demande les bus de données, d'adresses et de commandes pour être en lien avec la mémoire.

  Si DMA = 1 alors pas de demande des bus de la part les interfaces d'E/S.

- DMAA : réponse du µp au signal DMA

  Si DMAA = 1 alors le µp a reçu le signal DMA
  Si DMAA = 0 alors le µp n'a pas reçu le signal DMA

<u>Exemple d'utilisation</u>

- Chargement des programmes, fichiers de données, …, de l'unité disque dur ou disquette dans la mémoire vive (RAM).

- Stockage des programme, … de la RAM vers le disque dur ou disquette.

<u>Exemple de INT et INTA</u>

Si INT = 0, INTA = 0
Le µp exécute le programme principal (fonctionnement normal)

Si INT = 1 alors une interface d'E/S désire communiquer avec le µp pour un transfert de données :

1. Le µp va donc interrompre le programme principal

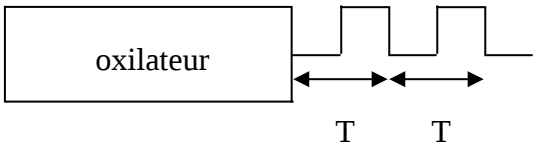
2. INTA = 1, le µp informe l'interface d'E/S qu'il a reçu le signal INT

3. Le µp exécute un sous programme pour le traitement de l'interruption (lecture des données de l'interface d'E/S vers le µp et du µp vers la mémoire).

4. A la fin de l'exécution du sous programme, le μp continue à exécuter le programme principal à partir de l'instruction suivante.

VI.1.4 Horloge

C'est un oscillateur qui génère un signal périodique.



à l'instant T le μp doit chercher l'instruction après une période il doit l'exécuter et ainsi de suite.

d'où la notion de temps pour μp.

IV.2 Fonctionnement d'un μp

L'exécution d'un programme se fait séquentiellement à partir du premier mot de la première instruction.

Pour chaque mot d'une instruction le µp effectue les phases suivantes :

- Phase de recherche
- Phase de décodage et d'exécution

IV.2.1 phase de recherche

le but de cette phase est de chercher le contenu d'un mot d'une instruction de la mémoire vers le registre d'instruction.

Elle s'effectue de la manière suivante :

1. Le contenu du compteur ordinal, qui contient l'adresse du premier mot de l'instruction est envoyé au registre d'adresse.
2. Un signal de lecture est envoyé par le bloc de commande et de contrôle vers la mémoire.
3. La mémoire une fois qu'elle reçoit l'adresse et le signal de lecture :

- effectue le décodage de l'adresse considérée (càd cherche la case mémoire dont l'adresse est dans le bus d'adresse).
- dépose sur le bus de données le contenu de la case mémoire spécifiée.

4. Le µp lit le bus de données et dépose le contenu dans le registre d'instruction.

5. Le contenu du compteur ordinal est incrémenté automatiquement par le bloc de contrôle et de commande.
   Les étapes de 1 à 5 vont se répéter jusqu'à la fin de la recherche de tous les mots de l'instruction.

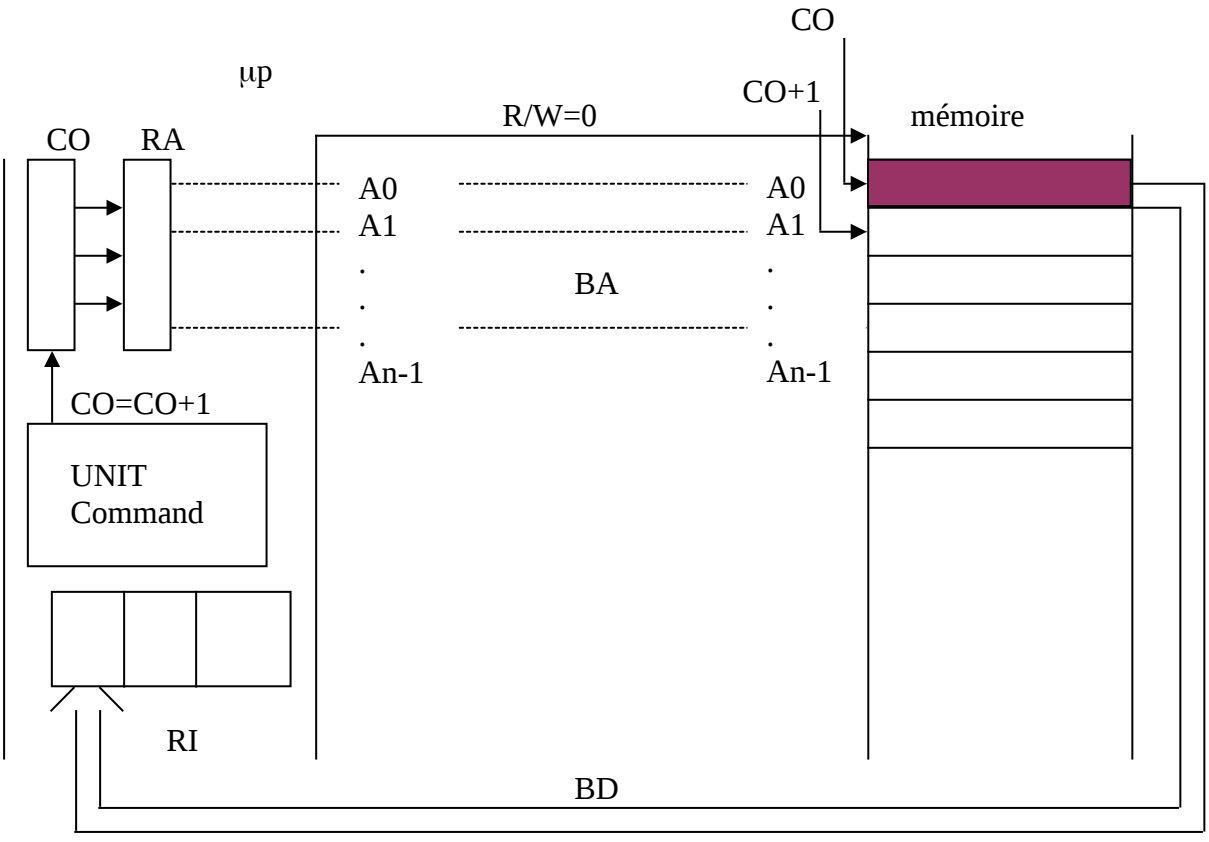


Figure IV.5 phase de recherche d’un cycle machine

IV.2.2 Phase de codage

L’instruction est stockée dans le registre d’instruction.

Le bloc de commande et de contrôle procède à son décodage et son exécution.
Le processus (recherche, décodage, et exécution) se répète avec l’instruction suivante dont l’adresse de son premier mot est déjà stockée dans le CO.

La synchronisation entre la phase de recherche, la phase de décodage et d’exécution se fait grâce à une base de temps fournis par un oscillateur.

Un oscillateur est un circuit électronique oscillant.

Le nombre de périodes par phase dépend du µp utilisé.

Le nombre de phases de recherche dépend de l’instruction à exécuter.

Le nombre de phases de décodage est 1.

Exemple

LOAD ACC $2000
Càd charger dans l'accumulateur le contenu de la case mémoire : $2000.

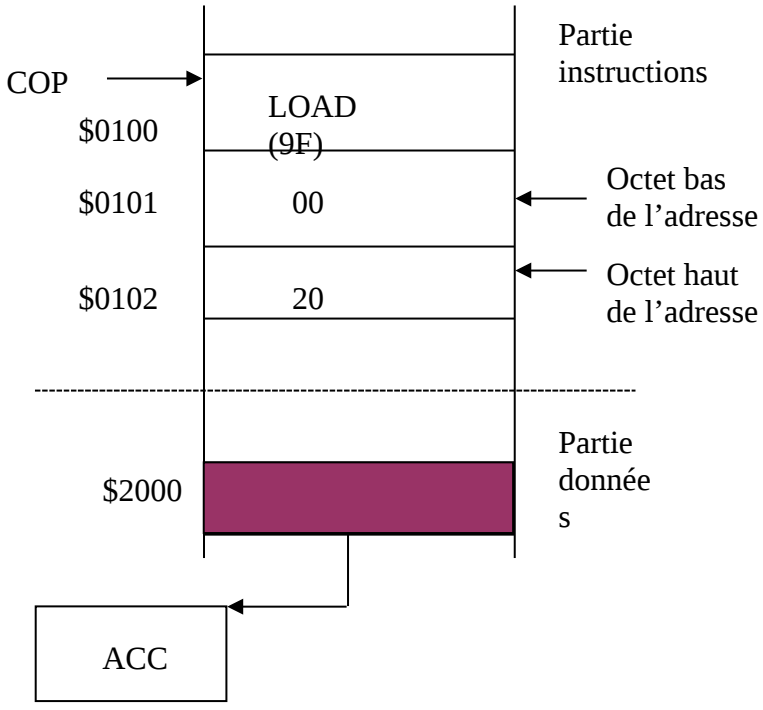


Figure IV.6 phase de codage d'un cycle machine

Phase de recherche : 3 périodes

Phase d'exécution : 2 périodes

Un cycle machine : 5 période

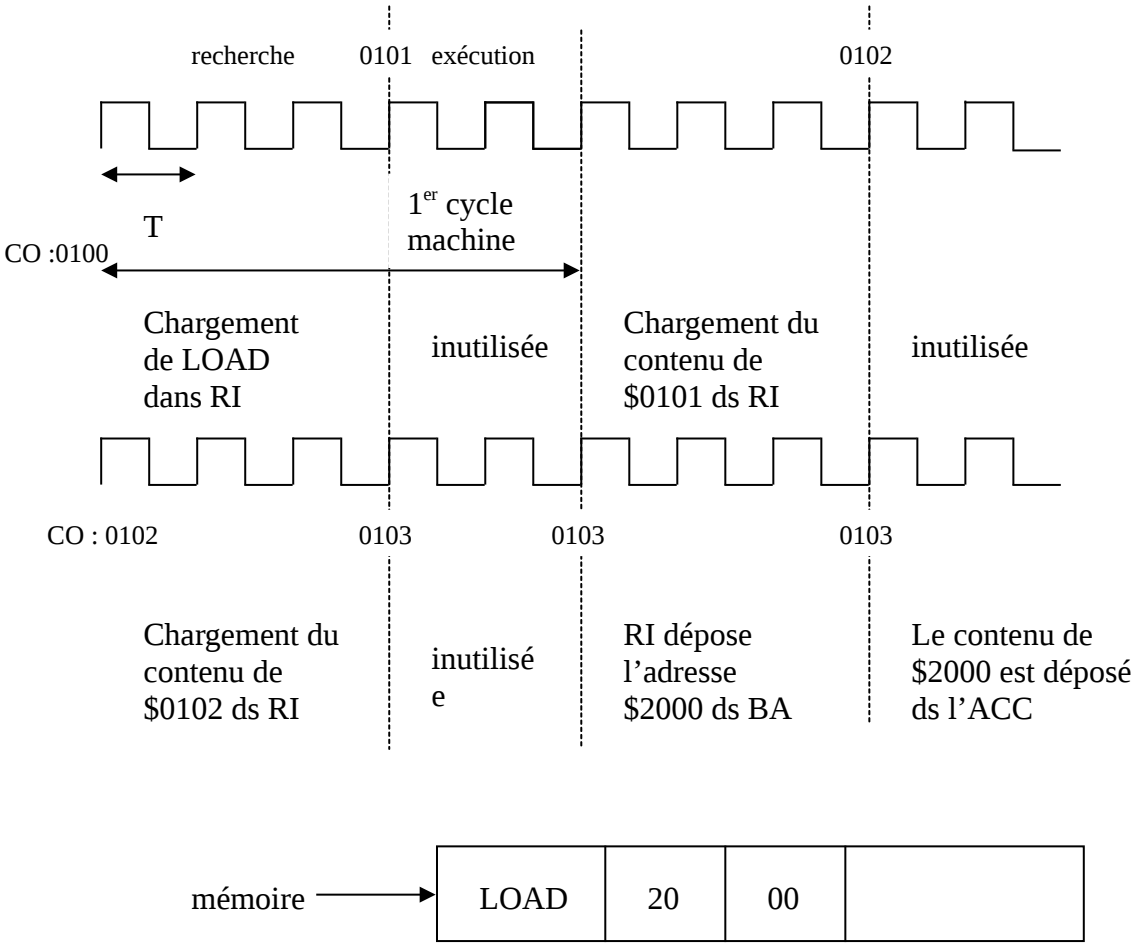


Figure IV.7 phases de recherche et d'exécution de l'instruction LOAD ACC 2000

L’adresse du CO s’incrémente à la fin de chaque phase de recherche d’adresse d’instruction et pas d’adresse de données.

CHAPITRE V: Introduction au langage machine

V.1 Caractéristiques du processeur étudié

La gamme de microprocesseurs 80x86 équipe les micro-ordinateurs de type PC et compatibles.

Les premiers modèles de PC, commercialisés au début des années 1980, utilisaient le 8086, un microprocesseur 16 bits.

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

Chacun de ces processeurs est plus puissant que les précédents : augmentation de la fréquence d'horloge, de la largeur des bus, ajout d'instructions et ajout de registres.

Chacun d'entre eux est *compatible* avec les modèles précédents.

un programme écrit dans le langage machine du 286 peut s'exécuter sans modification sur un 486. L'inverse n'est pas vrai : *compatibilité ascendante*.

Voici les caractéristiques du processeur simplifié que nous étudierons :

CPU
16 bits à accumulateur :

Bus de données 16 bits;

Bus d'adresse 32 bits;

Registres :

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

Les registres de données de 16 bits peuvent parfois être utilisés comme deux registres indépendants de 8 bits (AX devient la paire (AH,AL)).

V.2 Jeu d'instruction

Le jeu d’instruction d’un microprocesseur est appelé langage Assembleur.

Le jeu d’instruction représente l’aspect programmable du µp.

un microprocesseur fonctionne selon la nature de l’application à élaborer.

La programmation du langage assembleur exige la connaissance des registres internes du microprocesseur.

V.2.1 Types d'instructions

V.2.1.1 Instructions d'affectation

Déclenchent un transfert de données entre l'un des registres du processeur et la mémoire principale.

- transfert CPU ← Mémoire Principale (MP) (= lecture en MP);
- transfert CPU → Mémoire Principale (MP) (= écriture en MP);

V.2.1.2 Instructions arithmétiques et logiques

Opérations entre une donnée et l'accumulateur AX. Le résultat est placé dans l'accumulateur.

La donnée peut être une constante ou une valeur contenue dans un emplacement mémoire.

Exemples :

- addition : AX ← AX + donnée;
- soustraction : AX ← AX - donnée;
- incrémentation de AX : AX ← AX + 1;
- décrémentation : AX ← AX - 1;
- décalages à gauche et à droite;

V.2.1.3 Instructions de comparaison

Comparaison du registre AX à une donnée et positionnement des indicateurs.

V.2.1.4 Instructions de branchement

La prochaine instruction à exécuter est repérée en mémoire par le registre IP.

Les instructions de branchement permettent de modifier la valeur de IP pour exécuter une autre instruction (boucles, tests, etc.).

On distingue deux types de branchements :

- *branchements inconditionnels* : IP ← adresse d'une instruction;
- *branchements conditionnels* : Si une condition est satisfaite, alors branchement, sinon passage simple à l'instruction suivante.

V.2.2 Codage des instructions et mode d'adressage

Les instructions et leurs opérandes (paramètres) sont stockées en mémoire principale.

La taille totale d'une instruction dépend du type de l'instruction et du type d'opérande.

Chaque instruction est toujours codée sur un nombre entier d'octets, afin de faciliter son décodage par le processeur.

Une instruction est composée de deux champs :

- Le code opération;
- Le code opérande qui contient la donnée, ou l'adresse de la donnée en mémoire.

Champ1	Champ2
code opération	code opérande

Selon le mode d'adressage de la donnée, une instruction sera codée par 1, 2, 3 ou 4 octets.

On distingue ici quatre modes d'adressage : **implicite**, **immédiat**, **direct** et **relatif**.

V.2.2.1 Adressage implicite

L'instruction contient seulement le code opération, sur 1 ou 2 octets.

code opération
(1 ou 2 octets)

L'instruction porte sur des registres ou spécifie une opération sans opérande (exemple : ``incrémenter AX'').

V.2.2.2 Adressage immédiat

Le champ opérande contient la donnée (une valeur constante sur 1 ou 2 octets).

Code opération	valeur
(1 ou 2 octets)	(1 ou 2 octets

Exemple : ``Ajouter la valeur 5 à AX''. Ici l'opérande 5 est codée sur 2 octets puisque l'opération porte sur un registre 16 bits (AX).

V.2.2.3 Adressage direct

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

Code opération	Adresse de la donnée
(1 ou 2 octets)	(2 octets)

Dans le 80x86, les adresses sont toujours manipulées sur 16 bits, quelle que soit la taille réelle du bus.

Exemple : ``Placer dans AX la valeur contenue dans l'adresse 130H''.

V.2.2.4 Adressage relatif

Ce mode d'adressage est utilisé pour certaines instructions de branchement.

Le champ opérande contient un entier relatif codé sur 1 octet, nommé *déplacement*, qui sera ajouté à la valeur courante de IP.

Code opération	déplacement
(1 octet)	(1octet)

V.2.3 Temps d'exécution

Chaque instruction nécessite un certain nombre de cycles d'horloges pour s'effectuer.

Le nombre de cycles dépend de la complexité de l'instruction et du mode d'adressage : il est plus long d'accéder à la mémoire principale qu'à un registre du processeur.

La durée d'un cycle dépend bien sûr de la fréquence d'horloge de l'ordinateur.

Plus l'horloge bat rapidement, plus un cycle est court et plus on exécute un grand nombre d'instructions par seconde.

V.2.4 Ecriture des instructions en langage symbolique

Voici un programme en langage machine 80486, implanté à l'adresse 0100H :

A1 01 10 03 06 01 12 A3 01 14

Ce programme additionne le contenu de deux cases mémoire et range le résultat dans une troisième.

Nous avons simplement transcrit en hexadécimal le code du programme.

Il est clair que ce type d'écriture n'est pas très utilisable par un être humain.

A chaque instruction que peut exécuter le processeur correspond une représentation binaire sur un ou plusieurs octets, comme on l'a vu plus haut.

C'est le travail du processeur de décoder cette représentation pour effectuer les opérations correspondantes.

Afin de pouvoir écrire (et relire) des programmes en langage machine, on utilise une notation symbolique, appelée *langage assembleur*.

Ainsi, la première instruction du programme ci-dessus (code `A1 01 10`) sera notée :

MOV AX, [0110]

elle indique que la cellule mémoire d'adresse 0110H est chargée dans le registre AX du processeur.

On utilise des programmes spéciaux, appelés *assembleurs*, pour traduire automatiquement le langage symbolique en code machine.

Voici une transcription langage symbolique du programme complet.

L'adresse de début de chaque instruction est indiquée à gauche (en hexadécimal).

Adresse	Contenu MP	Langage Symbolique	Explication en français
0100	*A1 01 10*	*MOV AX, [0110]*	*AX ← contenu de $0110*
0103	*03 06 01 12*	*ADD AX, [0112]*	*AX = AX + contenu de $0112*
0107	*A3 01 14*	*MOV [0114], AX*	*Ranger AX en $0114.*

V.2.4.1 Sens des mouvements de données

La plupart des instructions spécifient des mouvements de données entre la mémoire principale et le microprocesseur.

En langage symbolique, on indique toujours la destination, puis la source.

L'instruction *MOV AX, [0110]* transfère le contenu de l'emplacement mémoire 0110H dans l'accumulateur,

L'instruction `MOV [0112], AX` transfère le contenu de l'accumulateur dans l'emplacement mémoire 0112.

L'instruction `MOV` (déplacer) s'écrit donc toujours :

MOV destination, source

V.2.4.2 Modes d'adressage

En adressage immédiat, on indique simplement la valeur de l'opérande en hexadécimal.

Exemple :

MOV AX, 12

En adressage direct, on indique l'adresse d'un emplacement en mémoire principale en hexadécimal entre crochets :

MOV AX, [A340]

En adressage relatif, on indique simplement l'adresse (hexa). L'assembleur traduit automatiquement cette adresse en un déplacement (relatif sur un octet).

Exemple :

JNE 0108

(nous étudierons l'instruction JNE par la suite).

V.2.4.3 Tableau des instructions

Symbole	Code Op.	Octets	
MOV AX, *valeur*	**B8**	**3 AX ← *valeur***	
MOV AX, [*adr*]	**A1**	**3**	**AX ← contenu de l'adresse *adr*.**
MOV [*adr*], AX	**A3**	**3**	**range le contenu deAX à l'adresse *adr*.**
ADD AX, *valeur*	**05**	**3**	**AX ← AX + *valeur***
ADD AX, [*adr*]	**03 06**	**4**	**AX ← AX + contenu de *adr*.**
SUB AX, *valeur*	**2D 3**	**AX ← AX - *valeur***	
SUB AX, [*adr*]	**2B 06**	**4**	**AX ← AX - contenu de *adr*.**
SHR AX, 1	**D1 E8**	**2**	**décale AX à droite.**
SHL AX, 1	**D1 E0**	**2**	**décale AX à gauche.**
INC AX	**40**	**1**	**AX ← AX + 1**
DEC AX	**48**	**1**	**AX ← AX - 1**
CMP AX, *valeur*	**3D**	**3 compare AX et *valeur*.**	
CMP AX, [*adr*]	**3B 06**	**4**	**compare AX et contenu de *adr*.**
JMP *adr*	**EB**	**2**	**saut inconditionnel (adr. relatif).**

JE *adr*	**74**	**2**	**saut si =**
JNE *adr*	**75**	**2**	**saut si ≠**
JG *adr*	**7F**	**2**	**saut si >**
JLE *adr*	**7E**	**2**	**saut si ≤**
JA *adr*		**saut si CF = 0**	
JB *adr*			**saut si CF = 1**

Table V.1: Quelques instructions du 80x86.

Le code de l'instruction est donné en hexadécimal dans la deuxième colonne.

La colonne suivante précise le nombre d'octets nécessaires pour coder l'instruction complète (opérande inclus).

V.3 Branchements

Le processeur exécute les instruction d’une manière séquentielle.

Il arrive fréquemment que l'on veuille faire répéter au processeur une certaine suite d'instructions, comme dans le programme :

Répéter 3 fois:

ajouter 5 au registre BX

En d'autres occasions, il est utile de déclencher une action qui dépend du résultat d'un test :

Si x < 0:

y = - x

sinon

y = x

Dans ces situations, on doit utiliser une instruction de *branchement*, ou *saut*, qui indique au processeur l'adresse de la prochaine instruction à exécuter.

Rappelons que le registre IP du processeur conserve l'adresse de la prochaine instruction à exécuter.

Lors d'un déroulement normal, le processeur effectue les actions suivantes pour chaque instruction :

1. lire et décoder l'instruction à l'adresse IP;
2. IP ← IP + *taille de l'instruction*;
3. exécuter l'instruction.

Pour modifier le déroulement normal d'un programme, il suffit que l'exécution de l'instruction modifie la valeur de IP : C'est ce que font les instructions de branchement.

On distingue deux catégories de branchements :

- Sauts *inconditionnels* (le saut est toujours effectué) ;
- Sauts *conditionnels* (le saut est effectué seulement si une condition est vérifiée).

V.3.1 Saut inconditionnel

La principale instruction de saut inconditionnel est *JMP*.

En adressage relatif, l'opérande de JMP est un *déplacement*, c'est à dire une valeur qui va être ajoutée à IP.

L'action effectuée par JMP est :

IP = IP + *déplacement*

Le déplacement est un entier relatif codé sur 8 bits.

La valeur du déplacement à utiliser pour atteindre une certaine instruction est :

***déplacement* = adr. instruction visée - adr. Instruction suivante**

Exemple : le programme suivant écrit indéfiniment la valeur 0 à l'adresse 0140H. La première instruction est implantée à l'adresse 100H.

Adresse	Contenu MP	Langage Symbolique	Explication en français
0100	B8 00 00	MOV AX, 0	met AX à zéro
0103	A3 01 40	MOV [140], AX	écrit à l'adresse 140
0106	EB FC	JMP 0103	branche en 103
0107	xxx	->	instruction jamais exécutée

Le déplacement est ici égal à FCH, c'est à dire -4 (=103H-107H).

V.3.2 indicateurs

Les instructions de branchement conditionnels utilisent les *indicateurs* (des bits spéciaux positionnés par l'UAL après certaines opérations).

Les indicateurs sont regroupés dans le *registre d'état* du processeur.

Ce registre n'est pas accessible globalement par des instructions;

chaque indicateur est manipulé individuellement par des instructions spécifiques.

V.3.2.1 Instruction CMP

Il est souvent utile de tester la valeur du registre AX sans modifier celui-ci.

L'instruction CMP effectue exactement les même opération que SUB, mais ne stocke pas le résultat de la soustraction.

Le seul effet de CMP est donc de positionner les indicateurs.

Exemple : après l'instruction

CMP AX, 5

on aura ZF = 1 si AX contient la valeur 5, et ZF = 0 si AX est différent de 5.

V.3.2.2 Instructions STC et CLC

Ces deux instructions permettent de modifier la valeur de l'indicateur CF.

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

V.3.3 Sauts conditionnels

Les instructions de branchements conditionnels effectuent un saut (comme JMP) si une certaine condition est vérifiée.

Si ce n'est pas le cas, le processeur passe à l'instruction suivante.

Les conditions s'expriment en fonction des valeurs des indicateurs.

Les instructions de branchement conditionnel s'utilisent en général immédiatement après une instruction de comparaison CMP.

Exemples :

JE
Jump if Equal
saut si ZF = 1

JNE
Jump if Not Equal
saut si ZF = 0;

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

V.4 Instructions Arithmétiques et logiques

Les instructions arithmétiques et logiques sont effectuées par l'UAL .

Nous abordons ici les instructions qui travaillent sur la représentation binaire des données : décalages de bits, opérations logiques bit à bit.

Notons que toutes ces opérations modifient l'état des indicateurs.

V.4.1 Instructions de décalage et de rotation

Ces opérations décalent vers la gauche ou vers la droite les bits de l'accumulateur.

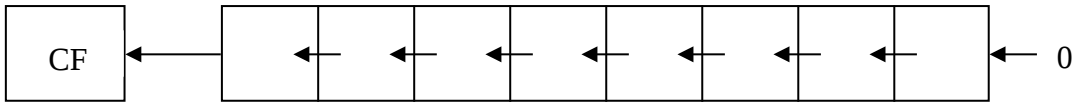
Elles sont utilisées pour décoder bit à bit des données, ou simplement pour diviser ou multiplier rapidement par une puissance de 2.

En effet, décaler AX de n bits vers la gauche revient à le multiplier par 2^n (sous réserve qu'il représente un nombre naturel et qu'il n'y ait pas de dépassement de capacité).

Un décalage vers la droite revient à diviser par 2^n.

Voici les variantes les plus utiles de ces instructions. Elles peuvent opérer sur les registres AX ou BX (16 bits) ou sur les registres de 8 bits AH, AL, BH et BL :

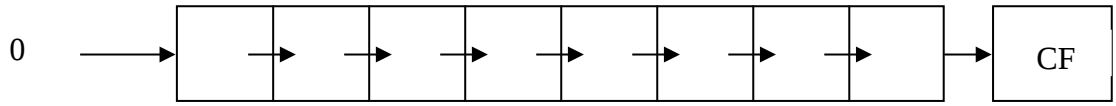
SHL *registre*, 1
(Shift Left)



Décale les bits du registre d'une position vers la gauche. Le bit de gauche est transféré dans l'indicateur CF.

Les bits introduits à droite sont à zéro.

***SHR registre*, 1**
(Shift Right)



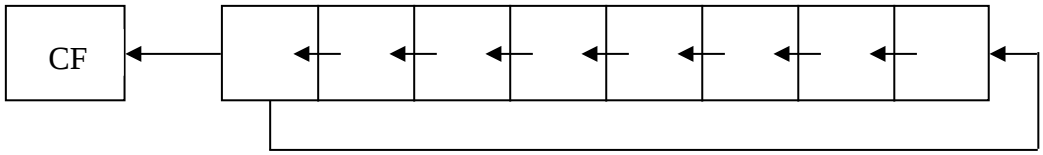
Comme SHL mais vers la droite.

Le bit de droite est transféré dans CF.

SHL et SHR peuvent être utilisé pour multiplier/diviser des entiers *naturels* (et non des relatifs car le bit de signe est perdu).

ROL *registre*, 1

(Rotate left)

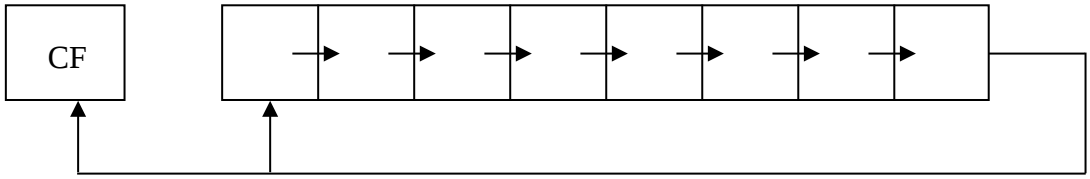


Rotation vers la gauche : le bit de poids fort passe à droite, et est aussi copié dans CF. Les autres bits sont décalés d'une position.

ROR *registre*, 1

(Rotate Right)

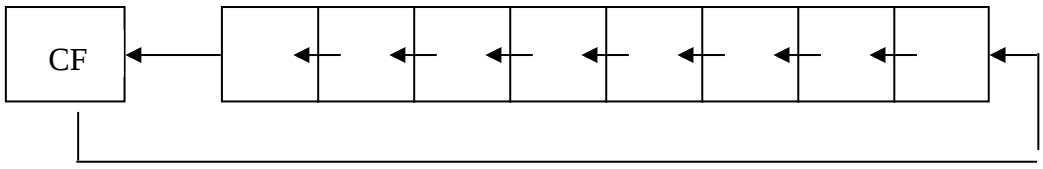
Comme ROL, mais à droite.



***RCL registre*, 1**

(Rotate Carry Left)

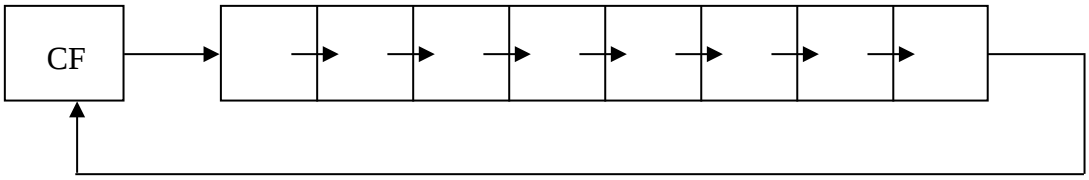
Rotation vers la gauche en passant par l'indicateur CF. CF prend la place du bit de poids faible; le bit de poids fort part dans CF.



RCR *registre*, 1

(Rotate Carry Right)

Comme RCL, mais vers la droite.



RCL et RCR sont utiles pour lire bit à bit le contenu d'un registre.

V.4.2 Instructions logiques

Les instructions logiques effectuent des opérations logiques bit à bit.

On dispose de trois opérateurs logiques : ET, OU et OU exclusif.

Il n'y a jamais propagation de retenue lors de ces opérations (chaque bit du résultat est calculé indépendamment des autres).

```
      0011          0011              0011
OU    0101    ET    0101      OU EX   0101
-------------------------------------------

      0111          0001              0110
```

Les trois instructions OR, AND et XOR sont de la forme :

OR *destination, source*.

destination désigne le registre ou l'emplacement mémoire (adresse) où doit être placé le résultat.

source désigne une constante (adressage immédiat), un registre (adressage implicite), ou une adresse (adressage direct).

Exemples :

```
OR   AX, FF00   ; AX <-  AX ou FFOO

 OR   AX, BX     ; AX <-  AX ou BX

OR   AX, [1492] ; AX <-  AX ou [1492]
```

OR *destination, source*

(OU)

OU logique : Chaque bit du résultat est égal à 1 si au moins l'un des deux bits opérande est 1.

OR est souvent utilisé pour forcer certains bits à 1.

Exemple : après `OR AX, FF00`, l'octet de poids fort de AX vaut FF, tandis que l'octet de poids faible est inchangé.

AND *destination, source*

(ET)

ET logique : Chaque bit du résultat est égal à 1 si les deux bits opérandes sont à 1.

AND est souvent utilisé pour forcer certains bits à 0.

Après `AND AX, FF00`, l'octet de poids faible de AX vaut 00, tandis que l'octet de poids fort est inchangé.

XOR *destination, source*

OU exclusif : Chaque bit du résultat est égal à 1 si l'un ou l'autre des bits opérandes (mais *pas les deux*) vaut 1.

XOR est souvent utilisé pour inverser certains bits. Après `XOR AX, FFFF`, tous les bits de AX sont inversés.

CHAPITRE VI. L’assembleur 80x86

VI.1 L’assembleur

VI.1.1 Pourquoi l'assembleur ?

Lorsque l'on doit lire ou écrire un programme en langage machine, il est difficile d'utiliser la notation hexadécimale.

On écrit les programmes à l'aide de symboles comme MOV, ADD, etc.

Les concepteurs du processeur fournissent toujours une documentation avec les codes des instructions de leur processeur, et les symboles correspondant.

le programmeur doit spécifier lui même les adresses des données et des instructions.

Soit par exemple le programme suivant, qui multiplie une donnée en mémoire par 8 :

```
0100    MOV BX, [0112]  ; charge la donnée
0103    MOV AX, 3
0106    SHL BX          ; décale à gauche
0108    DEC AX
0109    JNE  0106       ; recommence 3 fois
010B    MOV [0111], BX  ; range le résultat
010E    MOV AH, 4C
0110    INT 21H
0112    ; on range ici la donnée
```

Nous avons spécifié que la donnée était rangée à l'adresse 0111H, et que l'instruction de branchement JNE allait en 0106H.

Si on désire modifier légèrement ce programme, par exemple ajouter une instruction avant `MOV [0111] BX,` il va falloir modifier ces deux adresses.

On conçoit aisément que ce travail devient très difficile si le programme manipule beaucoup de variables.

L'utilisation d'un *assembleur* résout ces problèmes.

L'assembleur permet en particulier de nommer les variables et de repérer par des *étiquettes* certaines instructions sur lesquelles on va effectuer des branchements.

VI.1.2 De l'écriture du programme à son exécution

L'assembleur est un utilitaire qui n'est pas interactif.

Le programme qu'on désire traduire en langage machine (on dit *assembler*) doit être placé dans un fichier texte (avec l'extension **`.ASM`** sous DOS).

La saisie du programme source au clavier nécessite un programme appelé *éditeur de texte*.

L'opération d'assemblage traduit chaque instruction du programme source en une instruction machine.

Le résultat de l'assemblage est enregistré dans un fichier avec l'extension **`.OBJ`** (*fichier objet*).

Le fichier `.OBJ` n'est pas directement exécutable :

En effet, il arrive fréquemment qu'on construise un programme exécutable à partir de plusieurs fichiers sources.

Il faut ``relier'' les fichiers objets à l'aide d'un utilitaire nommé **éditeur de lien** (même si l'on en a qu'un seul).

L'éditeur de liens fabrique un fichier exécutable , avec l'extension **`.EXE`**.

Le fichier `.EXE` est directement exécutable.

Un utilitaire spécial du système d'exploitation, le **chargeur** est responsable de la lecture du fichier

exécutable, de son implantation en mémoire principale, puis du lancement du programme.

VI.1.3 Structure du programme source

La structure générale d'un programme assembleur est représentée comme suit.

```
Data SEGMENT ; data est le nom du segment de données

Directives de déclaration de données

Data ENDS ; fin du segment de données

Assume DS:data, CS:code

Code SEGMENT ; code est le nom du segment d'instructions

Debut : ; 1ere instruction, avec l'étiquette debut ;

Suite d'instructions

code ENDS

END debut ; fin du programme, avec l'étiquette ; de la première instruction.
```

Figure VI.1: Structure d'un programme en assembleur

Comme tout programme, un programme écrit en assembleur comprend des définitions de données et des instructions, qui s'écrivent chacune sur une ligne de texte.

Les données sont déclarées par des **directives**, **mots clef** spéciaux que comprend l'assembleur.

Les directives qui déclarent des données sont regroupées dans le **segment de données**, qui est délimité par les directives **`SEGMENT`** et **`ENDS`**.

Les instructions sont placées dans un autre segment, le **segment de code**.

La directive **`ASSUME`** est toujours présente et sera expliquée plus loin (section).

La première instruction du programme (dans le segment d'instruction) doit toujours être repérée par **une étiquette**.

Le fichier doit se terminer par la directive **`END`** avec le nom de l'étiquette de la première instruction.

ceci permet d'indiquer à l'éditeur de liens quelle est la première instruction à exécuter lorsque l'on lance le programme.

Les points-virgules indiquent **des commentaires**.

VI.1.4 Déclaration de variables

On déclare les variables à l'aide de **directives**.

L'assembleur attribue à chaque variable une adresse.

Dans le programme, on repère les variables grâce à leur nom.

Les noms des variables (comme les étiquettes) sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre.

Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les caractères @, ? et _.

Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

VI.1.4.1 Variables de 8 ou 16 bits

Les directives **DB** (*Define Byte*) et **DW** (*Define Word*) permettent de déclarer des variables de respectivement 1 ou 2 octets.

Exemple d'utilisation :

```
data    SEGMENT
entrée  DW  15     ; 2 octets initialisés à 15
sortie  DW  ?      ; 2 octets non initialisés
clé     DB  ?     ; 1 octet non initialisé
nega    DB  -1     ; 1 octet initialisé à -1
data    ENDS
```

Les valeurs initiales peuvent être données en hexadécimal (constante terminée par **H**) ou en binaire (terminée par **b**) :

```
data    SEGMENT
truc    DW  0F0AH     ; en hexa
masque  DB  01110000b ; en binaire
data    ENDS
```

Les variables s'utilisent dans le programme en les désignant par leur nom.

Après la déclaration précédente, on peut écrire par exemple :

```
MOV  AX, truc
AND  AL, masque
MOV  truc, AX
```

L'assembleur se charge de remplacer les noms de variable par les adresses correspondantes.

VI.1.4.2 Tableaux

Il est aussi possible de déclarer des tableaux, c'est à dire des suites d'octets ou de mots consécutifs.

Pour cela, utiliser plusieurs valeurs initiales :

```
data    SEGMENT
machin  DB  10, 0FH  ; 2 fois 1 octet
chose   DB  -2, 'ALORS'
data    ENDS
```

Remarquez la déclaration de la variable `chose` : un octet à -2 (=FEH), suivi d'une suite de caractères.

L'assembleur n'impose aucune convention pour la représentation des chaînes de caractères : c'est à l'utilisateur d'ajouter si nécessaire un octet nul pour marquer la fin de la chaîne.

Après chargement de ce programme, la mémoire aura le contenu suivant :

Début du segment **`data`** →	0AH	← machin
	OFH	← machin + 1
	FEH	← chose
	41H	← chose + 1
	4CH	← chose + 2
	4FH	← chose + 3
	52H	← chose + 4
	53H ← chose + 5	

Si on veut écrire un caractère `X` à la place du `O` de `ALORS`, on pourra écrire :

```
 MOV  AL, 'X'
MOV  chose+1, AL
```

Notons que `chose+1` est une constante (valeur connue au moment de l'assemblage) : l'instruction générée par l'assembleur pour

```
   MOV  chose +1,  AL
```

est `MOV [adr], AL` .

VI.1.4.3 Directive dup

Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive **dup** :

```
 tab   DB  100 dup (15) ; 100 octets valant 15
 zzz    DW   10   dup (?)    ; 10 mots de 16 bits non initialisés
```

VI.2 Segmentation de la mémoire

Nous abordons ici le problème de la segmentation de la mémoire.

Les données sont normalement regroupées dans une zone mémoire nommée **segment de données**.

Les instructions étaient placées dans un **segment**

d'instructions.

Ce partage se fonde sur la notion plus générale de **segment de mémoire**, qui est à la base du mécanisme de gestion des adresses par les processeurs 80x86.

Nous avons vu plus haut que les instructions utilisaient normalement des adresses codées sur 16 bits.

Nous savons aussi que le registre IP, qui stocke l'adresse d'une instruction, fait lui aussi 16 bits.

Or, avec 16 bits il n'est possible d'adresser que 2^{16} = 64 Kilo octets.

Le bus d'adresse du 80486 possède 32 bits.

Cette adresse de 32 bits est formée par la juxtaposition d'un registre segment (16 bits de poids fort) et d'un déplacement (***offset*** , 16 bits de poids faible).

Les adresses que nous avons manipulé jusqu'ici sont des déplacements.

Le schéma suivant illustre la formation d'une adresse 32 bits à partir du segment et du déplacement sur 16 bits :

On appellera **segment de mémoire** une zone mémoire adressable avec une valeur fixée du segment (les 16 bits de poids fort).

Un segment a donc une taille maximale de 64 Ko.

VI.2 .1 Segment de code et de données

La valeur du segment est stockée dans des registres spéciaux de 16 bits.

Le registre DS (*Data Segment*) est utilisé pour le segment de données, et le registre CS (*Code Segment*) pour le segment d'instructions.

VI.2 .1 .1Registre CS

Lorsque le processeur lit le code d'une instruction, l'adresse 32 bits est formée à l'aide du registre segment CS et du registre déplacement IP.

La paire de ces deux registres est notée **CS:IP.**

VI.2 .1.2 Registre DS

Le registre DS est utilisé pour accéder aux données manipulées par le programme. Ainsi, l'instruction

```
MOV AX, [0145]
```

donnera lieu à la lecture du mot mémoire d'adresse **DS:0145H.**

VI.2 .2 Initialisation des registres segment

Dans ce cours, nous n'écrirons pas de programmes utilisant plus de 64 Ko de code et 64 Ko de données.

ceci nous permettra de n'utiliser qu'un seul segment de chaque type.

Par conséquent, la valeur des registres CS et de DS sera fixée une fois pour toute au début du programme.

Le programmeur en assembleur doit se charger de l'initialisation de DS, c'est à dire de lui affecter l'adresse du segment de données à utiliser.

Par contre, le registre CS sera automatiquement initialisé sur le segment contenant la première instruction au moment du chargement en mémoire du programme (par le chargeur du système d'exploitation).

VI.2 .3 Déclaration d'un segment en assembleur

Comme nous l'avons vu, les directives **SEGMENT** et **ENDS** permettent de définir les segments de code et de données.

La directive **ASSUME** permet d'indiquer à l'assembleur quel est le segment de données et celui de code, afin qu'il génère des adresses correctes.

Enfin, le programme doit commencer, avant toute référence au segment de données, par initialiser le registre segment DS, de la façon suivante :

```
MOV AX, nom_segment_de_données
MOV DS, AX
```

(Il serait plus simple de faire `MOV DS, nom_segment_de_données` mais il se trouve que cette instruction n'existe pas.)

La figure VI.2 donne un exemple complet de programme assembleur.

Programme calculant la somme de deux entiers de 16 bits

```
data SEGMENT
 A DW 10 ; A = 10
B DW 1789 ; B = 1789
Result DW ? ; resultat
data ENDS
code SEGMENT
ASSUME DS:data, CS:code
Debut : MOV AX, data ;  étiquette car 1ere instruction
MOV DS, AX ; initialise DS
; Le programme:
MOV AX, A
ADD AX, B
 MOV result, AX ; range le résultat
```

```
; Retour au DOS:
MOV AH, 4CH
INT 21H
code ENDS
END Debut ; étiquette de la 1ere instruction.
```

Figure VI.2 Exemple de programme en assembleur

VI. 3 Adressage indirect

Nous introduisons ici un nouveau mode d'adressage, l'adressage indirect : très utile par exemple pour traiter des tableaux.

L'adressage indirect utilise le registre BX pour stocker l'adresse d'une donnée.

En adressage direct, on note l'adresse de la donnée entre crochets :

```
MOV  AX, [130]   ; adressage direct
```

De façon similaire, on notera en adressage indirect :

```
MOV  AX, [BX]   ; adressage indirect
```

Ici, BX contient l'adressage de la donnée.

L'avantage de cette technique est que l'on peut modifier l'adresse en BX, par exemple pour accéder à la case suivante d'un tableau.

Avant d'utiliser un adressage indirect, il faut charger BX avec l' *adresse* d'une donnée.

Pour cela, on utilise une nouvelle directive de l'assembleur, **`offset`**.

```
data        SEGMENT
truc        DW   1996
data        ENDS
        ...
MOV  BX, offset truc
                        ...
```

Si l'on avait employé la forme

```
            MOV  BX, truc
```

on aurait chargé dans BX la *valeur* stockée en `truc` (ici 1996), et non son adresse.

VI.3.1 Exemple : parcours d'un tableau

Voici un exemple plus complet utilisant l'adressage indirect.

Ce programme passe une chaîne de caractères en majuscules.

La fin de la chaîne est repérée par un caractère $.

On utilise un ET logique pour masquer le bit 5 du caractère et le passer en majuscule (voir le code ASCII).

```
data    SEGMENT
tab     DB 'Uu bbbbb Bbbbbbbbbbb', '$'
data    ENDS

code    SEGMENT
        ASSUME DS:data, CS:code

debut:  MOV  AX, data
        MOV  DS, AX

        MOV  BX, offset tab ; adresse debut tableau

repet:  MOV  AL, [BX]      ; lis 1 caractère
        AND  AL, 11011111b ;  force bit 5 a zero
        MOV  [BX], AL      ; range le caractère
        INC  BX            ; passe au suivant
        CMP  AL, '$'       ; arrive au $ final ?
        JNE  repet         ; sinon recommencer
```

```
        MOV  AH, 4CH
        INT  21H          ; Retour au DOS
code      ENDS
        END debut
```

VI.3.2 Spécification de la taille des données

Dans certains cas, l'adressage indirect est ambigu.

Par exemple, si l'on écrit

```
MOV [BX], 0 ; range 0 à l'adresse spécifiée par BX
```

l'assembleur ne sait pas si l'instruction concerne 1, 2 ou 4 octets consécutifs.

Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

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

VI.4 La pile

VI.4.1 Notion de pile

Les *piles* offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs.

Une pile est une zone de mémoire et un pointeur qui conserve l'adresse du *sommet* de la pile.

VI.4.2 Instructions PUSH et POP

Deux nouvelles instructions, PUSH et POP , permettent de manipuler la pile.

PUSH *registre*

empile le contenu du registre sur la pile.

POP *registre*

retire la valeur en haut de la pile et la place dans le registre spécifié.

Exemple : transfert de AX vers BX en passant par la pile.

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

(Note : cet exemple n'est pas très utile, il vaut mieux employer `MOV BX, AX`.)

La pile est souvent utilisée pour sauvegarder temporairement le contenu des registres :

AX et BX contiennent des données à conserver

```
 PUSH AX
 PUSH BX

MOV  BX, truc  ; on utilise AX
ADD  AX, BX    ;   et BX
MOV  truc, AX
```

POP BX ; récupère l'ancien BX
POP AX ; et l'ancien AX

On voit que la pile peut conserver plusieurs valeurs.

La valeur dépilée par POP est la *dernière* valeur empilée;

c'est pourquoi on parle ici de pile LIFO (*Last In First Out*, Premier Entré Dernier Sorti).

VI.4.3 Registres SS et SP

La pile est stockée dans un segment séparé de la mémoire principale.

Le processeur possède deux registres dédiés à la gestion de la pile, SS et SP.

Le registre SS (*Stack Segment*) est un registre segment qui contient l'adresse du segment de pile courant (16 bits de poids fort de l'adresse).

Il est normalement initialisé au début du programme et reste fixé par la suite.

Le registre SP (*Stack Pointer*) contient le déplacement du sommet de la pile (16 bits de poids faible de l' adresse).

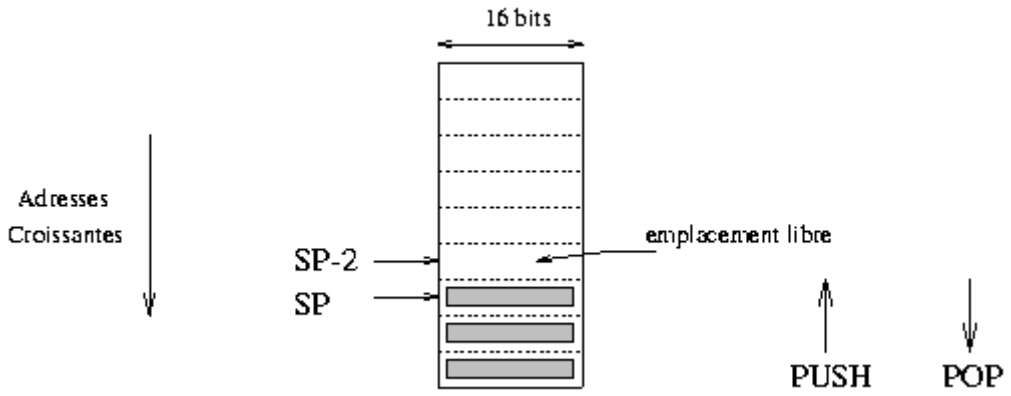


Figure VI.3: La pile.

SP pointe sur le sommet (dernier emplacement occupé).

La figure VI.3 donne une représentation schématique de la pile.

L'instruction PUSH effectue les opérations suivantes :

- SP ← SP - 2
- [SP] ← valeur du registre 16 bits.

Notons qu'au début (pile vide), SP pointe ``sous'' la pile.

L'instruction POP effectue le travail inverse :

- registre destination ← [SP]
- SP ← SP + 2

Si la pile est vide, POP va lire une valeur en dehors de l'espace pile, donc imprévisible.

VI.4.4 Déclaration d'une pile

Pour utiliser une pile en assembleur, il faut déclarer un segment de pile, et y réserver un espace suffisant.

Ensuite, il est nécessaire d'initialiser les registres SS et SP pour pointer sous le sommet de la pile.

Voici la déclaration d'une pile de 200 octets :

```
seg_pile    SEGMENT  stack   ; mot clef stack car pile
            DW  100 dup (?)  ; reserve espace
base_pile   EQU this word    ; etiquette base de la pile
seg_pile    ENDS
```

Noter le mot clef ``stack '' après la directive SEGMENT, qui indique à l'assembleur qu'il s'agit d'un segment de pile.

Afin d'initialiser SP, il faut repérer l'adresse du bas de la pile; c'est le rôle de la ligne
base_pile EQU this word

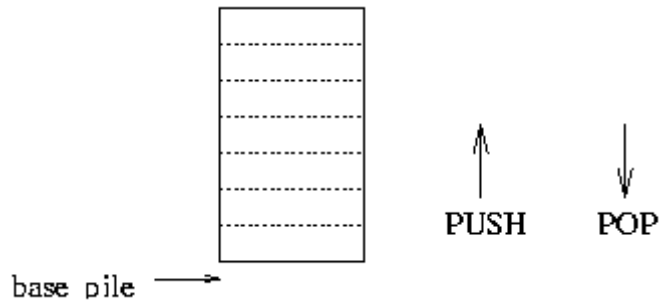


Figure VI.4: Une pile vide.

L'étiquette base-pile repère la base de la pile, valeur initiale de SP.

Après les déclarations ci-dessus, on utilisera la séquence d'initialisation :

```
ASSUME SS:seg_pile

MOV  AX, seg_pile
MOV  SS, AX       ; init Stack Segment

MOV  SP, base_pile ; pile vide
```

Noter que le registre SS s'initialise de façon similaire au registre DS;

par contre, on peut accéder directement au registre SP.

VI.5 Procédures

VI.5.1 Notion de procédure

La notion de procédure en assembleur correspond à celle de fonction en langage C, ou de sous-programme dans d'autres langages.

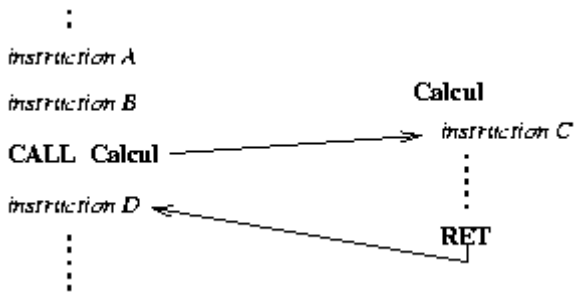


Figure VI.5: Appel d'une procédure.

La procédure est nommée **calcul**.

Après l'instruction B, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D.

Une procédure est une suite d'instructions effectuant une action précise.

Ces instructions sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme.

Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

L'exécution d'une procédure est déclenchée par un programme *appelant*.

Une procédure peut elle-même appeler une autre procédure, et ainsi de suite.

VI.5.2 Instructions CALL et RET

L'appel d'une procédure est effectué par l'instruction CALL.

***CALL* adresse_debut_procedure**

L'adresse est sur 16 bits, la procédure est donc dans le même segment d'instructions.

CALL est une nouvelle instruction de branchement inconditionnel.

La fin d'une procédure est marquée par l'instruction **RET** :

RET ne prend pas d'argument; le processeur passe à l'instruction placée immédiatement après le CALL.

RET est aussi une instruction de branchement : le registre IP est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL.

Comment le processeur retrouve-t-il cette valeur ?

Le problème est compliqué par le fait que l'on peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure.

L'*a*dresse de retour, utilisée par RET, est en fait sauvegardée sur la pile par l'instruction CALL.

Lorsque le processeur exécute l'instruction RET, il dépile l'adresse sur la pile (comme POP), et la range dans IP.

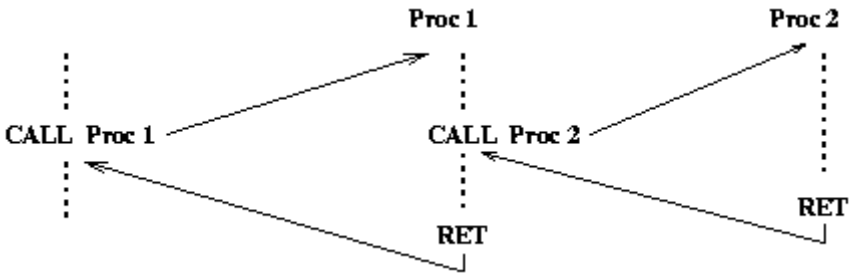


Figure VI.6: Plusieurs appels de procédures imbriqués.

L'instruction CALL effectue donc les opérations :

- Empiler la valeur de IP. A ce moment, IP pointe sur l'instruction qui suit le CALL.

- Placer dans IP l'adresse de la première instruction de la procédure (donnée en argument).

Et l'instruction RET :

- Dépiler une valeur et la ranger dans IP.

VI.5.3 Déclaration d'une procédure

L'assembleur possède quelques directives facilitant la déclaration de procédures.

On déclare une procédure dans le segment d'instruction comme suit :

```
Calcul    PROC near      ; procédure nommée Calcul

          ...            ; instructions

          RET            ; dernière instruction
Calcul    ENDP           ; fin de la procédure
```

Le mot clef **PROC** commence la définition d'une procédure.

near indiquant qu'il s'agit d'une procédure située dans le même segment d'instructions que le programme appelant.

L'appel s'écrit simplement :

```
CALL  Calcul
```

VI.5.4 Passage de paramètres

En général, une procédure effectue un traitement sur des données (*paramètres*).

Ces paramètres sont fournies par le programme appelant.

Une procédure produit un résultat qui est transmis à ce programme.

Plusieurs stratégies peuvent être employées :

1. **Passage par registre** : les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).
2. **Passage par la pile** : les valeurs des paramètres sont empilées. La procédure lit la pile.

VI.5.4 .1 Exemple avec passage par registre

On va écrire une procédure ``SOMME'' qui calcule la somme de 2 nombres naturels de 16 bits.

Convenons que les entiers sont passés par les registres AX et BX, et que le résultat sera placé dans le registre AX.

La procédure s'écrit alors très simplement :

```
SOMME   PROC near       ; AX <- AX + BX
        ADD AX, BX
        RET
SOMME   ENDP
```

et son appel, par exemple pour ajouter 6 à la variable Truc :

```
MOV  AX, 6
MOV  BX, Truc
CALL SOMME
MOV  Truc, AX
```

VI.5.4.2 Exemple avec passage par la pile

Cette technique met en oeuvre un nouveau registre, BP (*Base Pointer*), qui permet de lire des valeurs sur la pile sans les dépiler ni modifier SP.

Le registre BP permet un mode d'adressage indirect spécial, de la forme :

```
MOV  AX, [BP+6]
```

cette instruction charge le contenu du mot mémoire d'adresse BP+6 dans AX.

Ainsi, on lira le sommet de la pile avec :

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

et le mot suivant avec :

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

L'appel de la procédure ``SOMME2" avec passage par la pile est :

```
PUSH  6
PUSH  Truc
CALL  SOMME2
```

La procédure SOMME2 va lire la pile pour obtenir la valeur des paramètres.

Pour cela, il faut bien comprendre quel est le contenu de la pile après le CALL :

(premier paramètre) (adresse de retour)		
SP+4 → Truc SP+2 → IP SP →	6	(deuxième paramètre)

Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP empilée par CALL).

Chaque élément de la pile occupe deux octets.

La procédure SOMME2 s'écrit donc :

```
SOMME2      PROC near       ; AX <- arg1 + arg2
            MOV  BP, SP     ; adresse sommet pile
            MOV  AX, [BP+2] ; charge argument 1
            ADD  AX, [BP+4] ; ajoute argument 2
            RET
 SOMME2     ENDP
```

La valeur de retour est laissée dans AX.

La solution avec passage par la pile parait plus lourde sur cet exemple simple.

Cependant, elle est beaucoup plus souple dans le cas général que le passage par registre.

Il est très facile par exemple d'ajouter deux paramètres supplémentaires sur la pile.

Une procédure bien écrite modifie le moins de registres possible.

En général, l'accumulateur est utilisé pour transmettre le résultat et est donc modifié.

Les autres registres utilisés par la procédure seront normalement sauvegardés sur la pile.

Voici une autre version de SOMME2 qui ne modifie pas la valeur contenue par BP avant l'appel :

```
SOMME2      PROC near       ; AX <- arg1 + arg2
            PUSH BP         ; sauvegarde BP
            MOV  BP, SP     ; adresse sommet pile
            MOV  AX, [BP+4] ; charge argument 1
            ADD  AX, [BP+6] ; ajoute argument 2
            POP  BP         ; restaure ancien BP
            RET
```

SOMME2 ENDP

Noter que les index des arguments (BP+4 et BP+6) sont modifiés car on a ajouté une valeur au sommet de la pile.

CHAPITRE VII : Les interruptions

VII.1 Introduction

Le microprocesseur ne peut exécuter qu'une seule instruction à la fois.

Pour connaître son adresse, il utilise le couple de registres CS:IP dont la valeur est incrémentée automatiquement.

Par conséquent, le code du programme courant est exécuté *de manière linéaire*.

Imaginons cependant qu'un événement extérieur demande l'attention de l'ordinateur, par exemple la pression d'une touche du clavier.

La machine doit pouvoir réagir *immédiatement*, sans attendre que le programme en cours d'exécution se termine.

Pour cela, elle interrompt ce dernier pendant un bref instant, le temps de traiter l'événement survenu puis rend le contrôle au programme interrompu.

Une interruption n'est rien d'autre que l'appel d'une routine spéciale présente en mémoire appelée ISR (« Interrupt Service Routine »).

Comment sont-elles déclenchées ?

Une interruption peut être déclenchée par votre matériel. C'est ce qui arrive lorsque vous appuyez sur une touche du clavier.

Aucun logiciel n'intervient et le contrôle est passé directement à la routine qui gère le clavier : ce sont les *interruptions matérielles*.

Les *interruptions logicielles* sont quant à elles appelées par des *instructions* en langage machine au sein d'un programme.

Leur importance est capitale. Rappelez-vous que contrairement au PASCAL ou au C, l'assembleur ne dispose pas de fonction préprogrammée.

Chaque instruction doit être directement traduisible en langage machine.

Mais alors, comment fait-on pour écrire une chaîne de caractères à l'écran ? Ou bien pour lire un caractère entré au clavier ?

On déclenche les interruptions appropriées à l'aide de l'instruction « INT » du langage machine.

C'est donc une routine du DOS (ou parfois du BIOS) qui fera tout le travail. Les paramètres (ou leurs adresses) sont passés dans les registres.

Voici un petit exemple en assembleur qui écrit la lettre 'A' à l'écran :

```
;(...)

mov dl, 'A'
mov ah, 02
int 21h

;(...)
```

Examinons-le ligne par ligne :

- l'instruction "MOV DL, '*A*'" demande au processeur de mettre dans le registre DL le code ASCII de la lettre 'A', c'est-à-dire 65, ou 41h.
- "MOV AH, *02*" : mettre le nombre 2 dans AH.
- Enfin, la dernière instruction appelle l'interruption numéro *21h*. Il existe 256 interruptions. *Toutes sont notées en base hexadécimale.*

Explications :

l'interruption 21h est l'interruption du DOS par excellence.

Elle permet d'appeler de nombreuses fonctions très diverses. Pour cela, il suffit de mentionner leur numéro dans le registre AH.

Il est très difficile de mémoriser le rôle de chaque interruption, et a fortiori de chaque fonction ou sous-fonction, d'autant plus qu'elles sont désignées par des numéros hexadécimaux et qu'elles attendent des paramètres dans des registres précis.

C’est pourquoi tout programmeur doit avoir à sa disposition une liste des interruptions pour travailler.

Revenons à notre exemple. La fonction numéro 2 de l’interruption 21h sert à écrire un caractère à l’écran.

Il faut pour cela écrire le code ASCII du caractère dans le registre DL et bien sûr placer le nombre 2 dans AH.

Une fois que AH et DL ont été ajustés, l’interruption 21h peut être appelée à l’aide de l’instruction INT.

D’autres interruptions ne remplissent qu’une seule tâche. Vous n’avez donc pas besoin de mettre un numéro de fonction dans AH. L’appel de l’interruption suffit.

VII.2 La table des vecteurs d’interruptions

A chaque appel d’interruption, l’ordinateur doit pouvoir trouver l’adresse de l’ISR correspondante.

Pour cela, il dispose de *la table des vecteurs d’interruptions* (TVI, ou IVT : « *Interrupt Vector Table* »). Cette table est implantée à l’adresse 0000:0000 c’est-à-dire au début de la RAM.

VII.3 Sauvegarde de l’état des registres lors de l’appel

Un appel d’interruption obéit à certaines règles, car il est indispensable, une fois l’ISR exécutée, que le programme interrompu retrouve les registres dans le même état qu’ils étaient auparavant.

Tout doit se passer comme si l’interruption n’avait jamais eu lieu.

C’est pourquoi avant de faire un saut à l’ISR pointée par l’entrée correspondante dans la TVI, l’ordinateur sauvegarde tous les registres sur la pile courante.

Il restaurera leur contenu avant de rendre le contrôle. Cette procédure est automatique.

Nous étudions dans ce chapitre les interruptions *matérielles* (ou externes), c'est à dire déclenchées par le matériel (hardware) extérieur au processeur.

Nous nous appuyons ici aussi sur l'exemple du PC.

VII.4 Présentation

Les interruptions permettent au matériel de communiquer avec le processeur.

Les échanges entre le processeur et l'extérieur que nous avons étudiés jusqu'ici se faisaient toujours à l'initiative du processeur : par exemple, le processeur demande à lire ou à écrire une case mémoire.

Dans certains cas, on désire que le processeur réagisse rapidement à un évènement extérieur :

- arrivé d'un paquet de données sur une connexion réseau,
- frappe d'un caractère au clavier,

Les interruptions sont surtout utilisées pour la gestion des périphériques de l'ordinateurs.

Une interruption est signalée au processeur par un signal électrique sur une borne spéciale.

Lors de la réception de ce signal, le processeur ``traite'' l'interruption dès la fin de l'instruction qu'il était en train d'exécuter.

Le traitement de l'interruption consiste soit :

- à l'ignorer et passer normalement à l'instruction suivante : c'est possible uniquement pour certaines interruptions, nommées *interruptions masquables*.

  Il est en effet parfois nécessaire de pouvoir ignorer les interruptions pendant un certain temps, pour effectuer des traitements très urgents par exemple.

  Lorsque le traitement est terminé, le processeur *démasque* les interruptions et les prend alors en compte.

- à exécuter un *traitant d'interruption* (interrupt handler).

  Un traitant d'interruption est un programme qui est appelé automatiquement lorsqu'une interruption survient.

  Lorsque le traitant à effectuer son travail, il exécute l'instruction spéciale `IRET` qui permet de reprendre l'exécution à l'endroit où elle avait été interrompue.

VII.5 interruption matérielle sur PC

VII.5.1 Signaux d'interruption

Les processeurs de la famille 80x86 possèdent trois bornes pour gérer les interruptions : NMI , INTR , et [$\overline{\text{INTA}}$)] (voir figure).

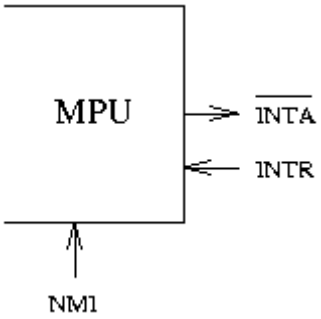


Figure VII.5: Bornes d'interruptions.

NMI

est utilisée pour envoyer au processeur une interruption non masquable (NMI, Non Maskable Interrupt).

Le processeur ne peut pas ignorer ce signal, et va exécuter le traitant donné par le vecteur 02H.

Ce signal est normalement utilisé pour détecter des erreurs matérielles (mémoire principale défaillante par exemple).

INTR

(Interrupt Request), demande d'interruption masquable.

Utilisée pour indiquer au MPU l'arrivée d'une interruption.

INTA

(Interrupt Acknowledge) Cette borne est mise à 0 lorsque le processeur traite effectivement l'interruption signalée par INTR (c'est à dire qu'elle n'est plus masquée).

VII.5.2 Indicateur IF

A un instant donné, les interruptions sont soit masquées soit autorisées, suivant l'état d'un indicateur spécial du registre d'état, IF (Interrupt Flag).

- si IF = 1, le processeur accepte les demandes d'interruptions masquables, c'est à dire qu'il les traite immédiatement;
- si IF = 0, le processeur ignore ces interruptions.

L'état de l'indicateur IF peut être modifié à l'aide de deux instructions, CLI (*CLear IF*, mettre IF à 0), et STI (*SeT IF*, mettre IF à 1).

VII.5.3 Contrôleur d'interruptions

L'ordinateur est relié a plusieurs périphériques, mais nous venons de voir qu'il n'y avait qu'un seul signal de demande d'interruption, INTR.

Le *contrôleur d'interruptions* est un circuit spécial, extérieur au processeur, dont le rôle est de distribuer et de mettre en attente les demandes d'interruptions provenant des différents périphériques.

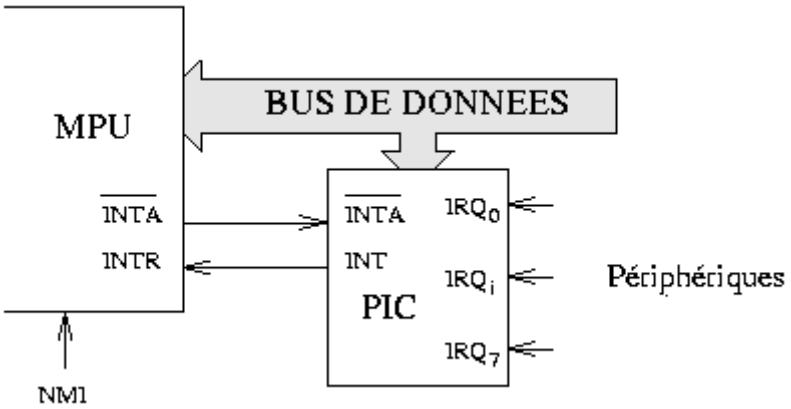


Figure VII.5: Le contrôleur d'interruptions (PIC, pour *Programmable Interruption Controler*).

La figure VII.5 indique les connexions entre le MPU et le contrôleur d'interruptions.

Le contrôleur est relié aux interfaces gérant les périphériques par les bornes IRQ (*InteRrupt reQuest*).

Il gère les demandes d'interruption envoyées par les périphériques, de façon à les envoyer une par une au processeur (via INTR).

Avant d'envoyer l'interruption suivante, le contrôleur attend d'avoir reçu le signal [$\overline{\text{INTA}}$)], indiquant que le processeur a bien traité l'interruption en cours.

VII.5.4 Déroulement d'une interruption externe masquable

Reprenons les différents évènements liés à la réception d'une interruption masquable :

1. Un signal INT est émis par un périphérique (ou plutôt par l'interface gérant celui-ci).

2. Le contrôleur d'interruptions reçoit ce signal sur une de ses bornes IRQ_i.

   Dès que cela est possible (suivant les autres interruptions en attente de traitement), le contrôleur envoie un signal sur sa borne INT.

3. Le MPU prend en compte le signal sur sa borne INTR après avoir achevé l'exécution de l'instruction en cours

Si l'indicateur IF=0, le signal est ignoré, sinon, la demande d'interruption est acceptée.

4. Si la demande est acceptée, le MPU met sa sortie [$\overline{\text{INTA}}$)] au niveau 0 pendant 2 cycles d'horloge, pour indiquer au contrôleur qu'il prend en compte sa demande.

5. En réponse, le contrôleur d'interruption place le numéro de l'interruption associé à la borne IRQ_i sur le bus de données.

6. Le processeur lit le numéro de l'interruption sur le bus de données et l'utilise pour trouver le vecteur d'interruption.

   Ensuite, tout se passe comme pour un appel système (interruption logicielle), c'est à dire que le processeur :

   a. sauvegarde les indicateurs du registre d'état sur la pile;
   b. met l'indicateur IF à 0 (masque les interruptions suivantes);
   c. sauvegarde CS et IP sur la pile;
   d. cherche dans la table des vecteurs d'interruptions l'adresse du traitant d'interruption, qu'il charge dans `CS:IP`.

7. La procédure traitant l'interruption se déroule. Pendant ce temps, les interruptions sont masquées (IF=0).

   Si le traitement est long, on peut dans certains cas ré-autoriser les interruptions avec l'instruction `STI`.

8. La procédure se termine par l'instruction `IRET`, qui restaure CS, IP et les indicateurs à partir de la pile, ce qui permet de reprendre le programme qui avait été interrompu.

VII.6 Entrées/Sorties par interruption

En général, les périphériques qui *reçoivent* des données de l'extérieur mettent en oeuvre un mécanisme d'interruption : clavier, contrôleurs de disques durs et CD-ROMS, etc.

Un exemple

Etudions ici très schématiquement le cas d'une lecture sur disque dur.

Soit un programme lisant des données sur un disque dur, les traitant et les affichant sur l'écran.

Voici l'algorithme général sans utiliser d'interruption :

- Répéter :
    1. envoyer au contrôleur de disque une demande de lecture d'un bloc de données.
    2. attendre tant que le disque ne répond pas (*scrutation*);
    3. traiter les données;

4. afficher les résultats.

Cette méthode simple est appelée entrée/sortie par *scrutation*.

L'étape 2 est une boucle de scrutation, de la forme :

- Répéter:
  - regarder si le transfert du disque est terminé;
- Tant qu'il n'est pas terminé.

La scrutation est simple mais inefficace : l'ordinateur passe la majorité de son temps à attendre que les données soit transférées depuis le disque dur.

Pendant ce temps, il répète la boucle de scrutation.

Ce temps pourrait être mis à profit pour réaliser une autre tâche.

Très grossièrement, les entrées/sorties par interruption fonctionnent sur le modèle suivant :

1. Installer un traitant d'interruption disque qui traite les données reçues et les affiche;
2. envoyer au contrôleur de disque une demande de lecture des données;

3. faire autre chose (un autre calcul ou affichage par exemple).

Dans ce cas, dès que des données arrivent, le contrôleur de disque envoie une interruption (via le contrôleur d'interrruptions) au processeur, qui arrête temporairement le traitement 3 pour s'occuper des données qui arrivent.

Lorsque les données sont traitées, le traitement 3 reprend (IRET). Pendant l'opération de lecture du disque dur, le processeur peut faire autre chose .

Dans la pratique, les choses sont un peu plus compliquées : il faut avoir plusieurs tâches à faire en même temps pour que l'utilisation des interruptions permettent un gain intéressant.

Ce principe est surtout mis à profit dans les systèmes multi-tâches comme UNIX ou Windows NT, que nous étudierons en deuxième année.