F.R OK

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

UNIVERSITE M'HAMED BOUGARA DE BOUMERDES

FACULTE DE TECHNOLOGIE
DEPARTEMENT D'Ingénierie des Systèmes Electrique

جامعة امحمد بوقرة-بومرداس
نائب العميد المكلف
بما بعد التدرج
البحث العلمي
والعلاقات الخارجية
كلية التكنولوجيا

Filière : Automatique

Spécialité : Automatique

Troisième Année de Licence LMD

Polycopié de cours : Microprocesseurs et Microcontrôleurs

Enseignant : Dr Triki Yacine
Maitre de Conférences Classe B

جامعة بومرداس
كلية التكنولوجيا
مكتبة
رقم: 114.101.01

جامعة امحمد بوقرة-بومرداس
مكتبة
الكلية
كلية التكنولوجيا

Année Universitaire : 2025/2026

Table des matières

Introduction générale

1 Objectifs du module 01
2 Compétences visées 02
3 Prérequis 02
4 Organisation du cours 02
5 Approche pédagogique 03

1 Rappel : Les systèmes de numération et codage

1 Le système décimal 04
2 Le système binaire 05
3 Le système hexadécimal 07
4 Un mot sur les unités de mesure 09
5 Circuits de commutation de l'information 10

2 Architecture de base d'un microprocesseur

1 Qu'est-ce qu'un microprocesseur ? 12
2 Architecture interne d'un microprocesseur. 12
3 Architecture externe d'un microprocesseur 14
4 Circulation de l'information dans un système à microprocesseur 15
5 Organisation de la mémoire dans un système à microprocesseur 16
6 Programmation d'un microprocesseur 17
7 Le cycle d'exécution des instructions par le microprocesseur 17

3 Le microprocesseur 8086

1 Description physique du 8086 19
2 Description et utilisation des signaux du 8086 19
3 Organisation interne du microprocesseur 8086. 21
4 Les registres du microprocesseur 8086 22
5 Segmentation de la mémoire dans le microprocesseur 8086 26

4 Introduction au jeu d'instructions et interruptions

1 Structure d'une instruction machine 28
2 Instructions de base en langage assembleur x86 28
3 Les instructions de branchement 33
4 Notion de pile 39
5 Les modes d'adressage 40
6 Les interruptions 44

5 Les mémoires et le décodage d'adresse dans les microprocesseurs

1 Qu'est-ce qu'une mémoire ? 50
2 Caractéristiques d'une mémoire 50
3 Classification des mémoires 51
4 Structure fondamentale des mémoires 53
5 Assemblage de mémoires 55
6 Le décodage d'adresse dans les microprocesseurs 59

6 Les interfaces d'entrées/sorties

1 Les circuits d'entrée/sortie 67
2 Adressage des ports d'entrée/sortie 68
3 Instructions de lecture et d'écriture d'un port d'entrée/sortie 70
4 L'interface parallèle 8255 72
5 L'interface série 8250 77

7 Le microcontrôleur

1 Du microprocesseur au microcontrôleur 82
2 Présentation d'un microcontrôleur PIC 82
3 Le microcontrôleur PIC 16F877 84

8 Programmation des microcontrôleurs PIC – Assembleur et C

1 Généralités et notions communes 94
2 Programmation en langage assembleur des microcontrôleurs PIC 97
3 Programmation en langage C des microcontrôleurs PIC 102

Références bibliographiques 117

République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche Scientifique

UNIVERSITE **M'**HAMED **B**OUGARA DE **B**OUMERDES

FACULTE DE **T**ECHNOLOGIE
DEPARTEMENT D'Ingénierie des Systèmes Electrique

Filière : Automatique

Spécialité : Automatique

Troisième Année de Licence LMD

Polycopié de cours : Microprocesseurs et Microcontrôleurs

Enseignant : Dr Triki Yacine
Maitre de Conférences Classe B

Année Universitaire : 2025/2026

Table des matières

Introduction générale

1 Objectifs du module 01
2 Compétences visées 02
3 Prérequis 02
4 Organisation du cours 02
5 Approche pédagogique 03

1 Rappel : Les systèmes de numération et codage

1 Le système décimal 04
2 Le système binaire 05
3 Le système hexadécimal 07
4 Un mot sur les unités de mesure 09
5 Circuits de commutation de l'information 10

2 Architecture de base d'un microprocesseur

1 Qu'est-ce qu'un microprocesseur ? 12
2 Architecture interne d'un microprocesseur. 12
3 Architecture externe d'un microprocesseur 14
4 Circulation de l'information dans un système à microprocesseur 15
5 Organisation de la mémoire dans un système à microprocesseur 16
6 Programmation d'un microprocesseur 17
7 Le cycle d'exécution des instructions par le microprocesseur 17

3 Le microprocesseur 8086

1 Description physique du 8086 19
2 Description et utilisation des signaux du 8086 19
3 Organisation interne du microprocesseur 8086. 21
4 Les registres du microprocesseur 8086 22
5 Segmentation de la mémoire dans le microprocesseur 8086 26

4 Introduction au jeu d'instructions et interruptions

1 Structure d'une instruction machine 28
2 Instructions de base en langage assembleur x86 28
3 Les instructions de branchement 33
4 Notion de pile 39
5 Les modes d'adressage 40
6 Les interruptions 44

5 Les mémoires et le décodage d'adresse dans les microprocesseurs

1 Qu'est-ce qu'une mémoire ? 50
2 Caractéristiques d'une mémoire 50
3 Classification des mémoires 51
4 Structure fondamentale des mémoires 53
5 Assemblage de mémoires 55
6 Le décodage d'adresse dans les microprocesseurs 59

6 Les interfaces d'entrées/sorties

1 Les circuits d'entrée/sortie 67
2 Adressage des ports d'entrée/sortie 68
3 Instructions de lecture et d'écriture d'un port d'entrée/sortie 70
4 L'interface parallèle 8255 72
5 L'interface série 8250 77

7 Le microcontrôleur

1 Du microprocesseur au microcontrôleur 82
2 Présentation d'un microcontrôleur PIC 82
3 Le microcontrôleur PIC 16F877 84

8 Programmation des microcontrôleurs PIC – Assembleur et C

1 Généralités et notions communes 94
2 Programmation en langage assembleur des microcontrôleurs PIC 97
3 Programmation en langage C des microcontrôleurs PIC 102

Références bibliographiques 117

Introduction générale

Le module *Microprocesseurs et Microcontrôleurs* s'adresse aux étudiants de troisième année de licence en Automatique. Il constitue une étape essentielle dans la formation en électronique et en systèmes embarqués, en permettant d'acquérir les bases nécessaires à la compréhension et à la conception de systèmes numériques programmables.

Les microprocesseurs et microcontrôleurs occupent aujourd'hui une place centrale dans les systèmes industriels, les systèmes embarqués et les applications de contrôle-commande. Leur maîtrise est donc indispensable pour concevoir, analyser et optimiser des systèmes automatisés modernes.

Ce cours a pour objectif principal de fournir aux étudiants une compréhension approfondie de l'architecture, du fonctionnement et de la programmation des microprocesseurs ainsi que des microcontrôleurs. Il vise également à développer des compétences pratiques permettant d'utiliser ces dispositifs dans des applications réelles.

1. Objectifs du module

À l'issue de ce cours, l'étudiant sera capable de :

- **Comprendre** l'architecture interne et externe des microprocesseurs ainsi que le fonctionnement des systèmes à base de microprocesseur.
- **Expliquer** les concepts liés au jeu d'instructions, aux interruptions et aux différents modes d'adressage.
- **Identifier** les différentes technologies de mémoire (RAM, ROM) et leurs caractéristiques.
- **Analyser** les mécanismes de fonctionnement des mémoires ainsi que les techniques de rafraîchissement et de décodage d'adresses.
- **Comprendre** le fonctionnement des interfaces d'entrées/sorties (série et parallèle).
- **Décrire** l'architecture et le fonctionnement du microprocesseur 8086.
- **Expliquer** le principe de fonctionnement des microcontrôleurs et de leurs périphériques.
- **Programmer** des systèmes simples en langage assembleur.
- **Appliquer** les connaissances théoriques à la conception et à la simulation de systèmes à base de microprocesseurs et microcontrôleurs.

2. Compétences visées

Ce module permet de développer les compétences suivantes :

- **Analyser** un système numérique basé sur un microprocesseur ou microcontrôleur.
- **Concevoir** des solutions matérielles et logicielles simples pour des applications embarquées.
- **Programmer** en langage assembleur pour le pilotage de systèmes électroniques.
- **Interfacer** un microprocesseur avec des périphériques d'entrées/sorties.
- **Diagnostiquer** et résoudre des problèmes liés au fonctionnement des systèmes numériques.

3. Prérequis

Pour suivre ce cours dans de bonnes conditions, les étudiants doivent maîtriser les notions suivantes :

- **Comprendre** les composants matériels d'un système numérique : processeur, mémoires (RAM, ROM), bus de données, d'adresse et de contrôle, ainsi que les périphériques d'entrée/sortie.
- **Maîtriser** les bases de la logique combinatoire : portes logiques (AND, OR, NOT), multiplexeurs, décodeurs et additionneurs.
- **Comprendre** les principes de la logique séquentielle : registres, compteurs et circuits de synchronisation.
- **Connaître** les bases de la programmation : algorithmes, structures de données (tableaux, piles, files), structures de contrôle (conditions, boucles).

4. Organisation du cours

Le cours est structuré en huit chapitres complémentaires. Chaque chapitre est conçu de manière progressive, en combinant des apports théoriques et des activités pratiques afin de faciliter l'assimilation des concepts.

Chapitre 1 : Rappel – Les systèmes de numération et codage

Ce chapitre présente les systèmes de numération (binaire, décimal, hexadécimal), les techniques de conversion entre bases ainsi que les principaux codes utilisés en informatique.

Chapitre 2 : Architecture de base d'un microprocesseur

Introduction aux systèmes à base de microprocesseur et étude de leur architecture interne et externe.

Chapitre 3 : Le microprocesseur 8086

Présentation détaillée de l'architecture, des registres et du principe de segmentation de la mémoire.

Chapitre 4 : Introduction au jeu d'instructions et interruptions

Ce chapitre présente les principales instructions de base du langage assembleur x86. Il aborde également les instructions de branchement, la notion de pile, les différents modes d'adressage et le principe des interruptions dans les microprocesseurs.

Chapitre 5 : Les mémoires

Étude des technologies RAM et ROM, de leurs caractéristiques, des techniques de rafraîchissement et du décodage d'adresses.

Chapitre 6 : Les interfaces d'entrées/sorties

Étude des interfaces série et parallèle, notamment le circuit 8255, ainsi que les instructions de configuration, de lecture et d'écriture.

Chapitre 7 : Le microcontrôleur

Ce chapitre introduit le concept de microcontrôleur en présentant l'évolution du microprocesseur vers le microcontrôleur. Il aborde également la structure générale d'un microcontrôleur PIC ainsi que les principales caractéristiques du microcontrôleur PIC16F877.

Chapitre 8 : Programmation des microcontrôleurs PIC – Assembleur et C

Ce chapitre présente les notions générales liées à la programmation des microcontrôleurs PIC. Il aborde la programmation en langage assembleur et en langage C, illustrée par des exemples d'applications et des programmes pratiques.

5. Approche pédagogique

Le cours s'appuie sur une approche combinant :

- des explications théoriques progressives,
- des exemples illustratifs,
- des exercices d'application,
- des activités pratiques de programmation et de simulation.

Cette approche permet à l'étudiant de passer de la compréhension des concepts à leur mise en œuvre concrète.

Rappel : Les systèmes de numération et codage

Introduction

Dans le domaine du numérique, différents systèmes de numération sont utilisés pour représenter les informations. Les systèmes les plus courants sont le décimal, le binaire et l'hexadécimal. Parmi eux, le système décimal est le plus familier puisqu'il est celui que nous utilisons quotidiennement pour compter et effectuer des calculs. Dans un système numérique, trois concepts clés sont à prendre en considération :

1. **La base (B) du système :** La base est un nombre entier arbitraire qui définit la façon dont les nombres sont représentés dans ce système.
2. **Les chiffres du système :** Ce sont des caractères distincts, chacun représentant un élément unique de la base. Il y a donc B chiffres différents au total.
3. **Le poids d'un chiffre en fonction de sa position :** Chaque chiffre a un poids qui dépend de sa position dans le nombre. Les chiffres les plus à gauche, appelés **MSB** (*Most Significant Bit*), ont un poids plus élevé, tandis que les chiffres les plus à droite, appelés **LSB** (*Least Significant Bit*), ont un poids plus faible.

Un nombre entier positif peut toujours s'écrire comme une somme de puissances de la base B, ce qui correspond à son écriture sous forme de polynôme arithmétique.

$$N = a_n \times B^n + \ldots + a_1 \times B^1 + a_0 \times B^0$$

Dans cette représentation, "N" représente le nombre entier, "a_n" jusqu'à "a_0" sont les chiffres du nombre, et "n" représente la position du chiffre dans la représentation.

1. Le système décimal

Le système décimal, appelé également système de numération en base 10, est basé sur l'utilisation de dix chiffres allant de 0 à 9 pour représenter les nombres. Il s'agit d'un système de numération positionnel, ce qui signifie que la valeur d'un chiffre dépend de sa position dans le nombre. Dans ce système, les chiffres situés à droite correspondent aux poids les plus faibles, tandis que la valeur associée aux positions augmente lorsqu'on se déplace vers la gauche. Chaque position correspond à une puissance de 10.

Prenons comme exemple le nombre 893. Le chiffre 8 correspond à 8 centaines, le chiffre 9 correspond à 9 dizaines et le chiffre 3 correspond à 3 unités. Ainsi, le chiffre le moins significatif (LSB) est 3, alors que le chiffre le plus significatif (MSB) est 8.

Il faut également noter que la numérotation des positions commence toujours à partir de la droite en commençant par la position 0. Par conséquent, le nombre 893 peut être exprimé sous la forme suivante : $893 = 8 \times 10^2 + 9 \times 10^1 + 3 \times 10^0$.

Exemple 2 :

Le nombre 79 431 : $79\,431 = 7 \times 10^4 + 9 \times 10^3 + 4 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$

➔ Le chiffre de poids le plus faible (LSB) est 1, tandis que le chiffre de poids le plus élevé (MSB) est 7.

2. Le système binaire

Lorsqu'il s'agit d'informatique, les ordinateurs sont fondamentalement des machines qui ne peuvent faire la distinction qu'entre deux états : la présence ou l'absence de tension électrique. Par conséquent, le système de numération décimal traditionnel, qui utilise 10 chiffres (0 à 9), n'est pas adapté à cette réalité binaire.

Pour représenter efficacement les deux niveaux logiques utilisés par les ordinateurs (niveau haut de tension et niveau bas), il est essentiel d'adopter un système de numération binaire. Ce système utilise seulement deux chiffres : 0 et 1. Chaque chiffre binaire est appelé un "*Bit*," une abréviation de "*BInary uniT*" en anglais. Un groupe de huit bits est communément appelé un "Octet" ou un "*Byte*" en anglais.

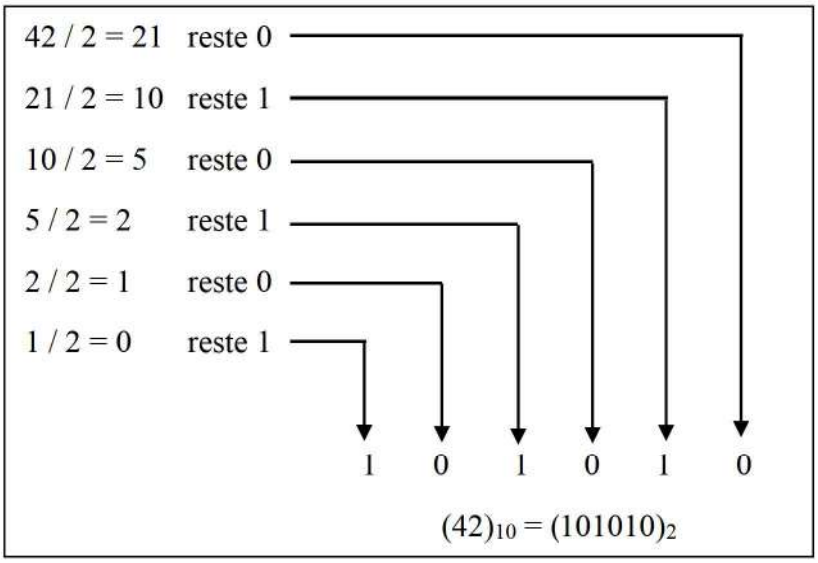
En utilisant le système binaire, n'importe quel nombre peut être représenté sous forme d'une séquence de 0 et de 1. Un nombre binaire est généralement écrit sous la forme B'xxxxxx', où chaque "x" peut prendre la valeur 0 ou 1. Cette représentation binaire permet aux ordinateurs de traiter des données et d'effectuer des calculs en utilisant des combinaisons de 0 et de 1 pour représenter des informations complexes.

2.1. Conversion décimale/binaire

Pour convertir un nombre décimal en nombre binaire, on utilise généralement la méthode des divisions successives par 2. Cette méthode consiste à diviser successivement le nombre décimal par 2 et à noter le reste obtenu à chaque division. L'opération est répétée jusqu'à ce que le quotient devienne nul.

Le nombre binaire est ensuite obtenu en lisant les restes dans l'ordre inverse : le premier reste correspond au bit de poids faible et le dernier reste correspond au bit de poids fort.

Exemple : Convertir le nombre $(42)_{10}$ en binaire



2.2. Conversion binaire/décimale

La conversion binaire-décimal consiste à multiplier chaque chiffre du nombre binaire par 2 élevé à la puissance de sa position (en partant de la droite) et à additionner les résultats. Cette

somme donne la valeur décimale équivalente. Le nombre binaire B'1011' serait interprété en décimal comme suit :

$$(1011)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = (11)_{10}$$

Le nombre binaire B'11010' exprimé en binaire serait interprété en décimal comme suit :

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16 + 8 + 0 + 2 + 0 = (26)_{10}$$

Un bit, en tant que plus petite unité, peut prendre l'une de deux valeurs, 0 ou 1. Ainsi, avec un seul bit, nous avons deux combinaisons possibles : 0 et 1. Cependant, lorsque nous augmentons le nombre de bits, nous multiplions le nombre de combinaisons possibles.

⇨ Sur 3 bits, on peut représenter 8 combinaisons : tous les entiers allant de 0 à 7.
⇨ Sur 4 bits, on peut représenter 16 combinaisons : tous les entiers allant de 0 à 15.
⇨ Sur 8 bits, on peut représenter 256 combinaisons : tous les entiers allant de 0 à 255.

Ainsi, en base 2, sur n bit on peut représenter 2^n combinaisons. En outre, tous les nombres N compris entre 0 et $(2^n - 1)$.

$$\mathbf{0 \leq N \leq (2^n - 1)}$$

2.3. Représentation binaire d'un nombre signé en utilisant la notation complément à 2

Pour trouver la représentation binaire d'un nombre signé en utilisant la notation complément à deux, voici les étapes que nous pouvons suivre :

1- Nous commençons par **trouver la représentation binaire** du nombre absolu (la valeur absolue) en utilisant la **notation binaire standard**.

2- Si le nombre est positif, nous avons terminé. La représentation binaire sera la même que celle trouvée à l'étape 1.

3- Si le nombre est négatif, nous trouvons le **complément à un (C_1)** de la représentation binaire trouvée à l'étape 1. Cela signifie que nous inverserons tous les bits (0 devient 1, et 1 devient 0).

4- Nous ajoutons 1 au résultat de l'étape 3 pour obtenir le **complément à deux (C_2)**.

5- Le résultat obtenu à l'étape 4 est la représentation binaire du nombre signé. Il est donné sous la forme suivante :

$$N = \underbrace{a_{n-1}}_{\text{MSB}} \overbrace{a_{n-2} \ \ldots \ a_1 \ a_0}^{\text{Valeur absolue}}$$

Signe

On constate que la représentation binaire signée est basée sur le bit le plus significatif (MSB), qui indique le signe du nombre.

- Si le MSB est 0, le nombre est positif,
- si le MSB est 1, le nombre est négatif.

Les bits restants (a_{n-2} à a_0) représentent la valeur absolue du nombre N en base 2.

Exemple : Représentation binaire des nombres signés – (48) et – (20) en utilisant la notation complément à deux sur 8 bits.

$$
\begin{aligned}
-(48)_{10} &= -(00110000)_2 \\
&= C_2(00110000)_2 \\
&= C_1(00110000)_2+1 \\
&= (11001111 + 1)_2 \\
&= (11010000)_2
\end{aligned}
\qquad
\begin{aligned}
-(20)_{10} &= -(00010100)_2 \\
&= C_2(00010100)_2 \\
&= C1(00010100)_2 + 1 \\
&= (11101011 + 1)_2 \\
&= (11101100)_2
\end{aligned}
$$

On notera également que pour les nombres signés, avec **une taille de *n* bits**, nous pouvons représenter **2^n combinaisons**. En outre, nous pouvons représenter tous les nombres N compris entre $-(2^{n-1})$ et $(2^{n-1}-1)$.

$$-(2^{n-1}) \leq N \leq (2^{n-1}-1).$$

2.4. Représentation décimale d'un nombre signé en binaire

Pour convertir un nombre signé en binaire en un nombre décimal, on peut suivre ces étapes :

- On **détermine si le nombre est positif ou négatif**. Cela est généralement indiqué par le MSB. Si ce bit est 0, le nombre est positif ; s'il est 1, le nombre est négatif.
- Si le nombre est négatif, on calcule d'abord **le complément à deux** pour obtenir sa valeur absolue en binaire.
- Ensuite, on convertit les bits (en binaire) en décimal en utilisant la méthode standard.
- Finalement, on ajoute un signe négatif (–) devant le résultat si le nombre était négatif à l'origine. **Si le nombre est positif,** on peut directement convertir la représentation binaire en décimal en utilisant la méthode standard.

Exemple : Représentation décimal du nombre signé $(11101100)_2$ sur 8 bits.

$$
\begin{aligned}
(11101100)_2 &= -C_2(11101100)_2 \\
&= -(C_1(11101100) + 1)_2 \\
&= -(00010100)_2 \\
&= -(1 \times 2^4 + 1 \times 2^2)_{10} \\
&= -(20)_{10}
\end{aligned}
$$

3. Le système hexadécimal

Écrire des nombres en binaire sur un grand nombre de bits peut être complexe à gérer et peut conduire à des erreurs d'écriture. Pour remédier à cette complexité, une méthode a été mise en place : **l'octet est divisé en deux groupes** de quatre bits, communément appelés **"quartets"**, et chaque quartet est représenté par un chiffre ou une lettre selon un tableau spécifique. Ainsi, **chaque quartet peut coder des nombres décimaux de 0 à 15**. Pour rendre cette représentation plus concise, Les valeurs comprises entre **10 et 15** sont **représentées** par les lettres **A à F**. Un nombre hexadécimal est généralement précédé du suffixe 0x.

Décimal	Binaire	Hexa		Décimal	Binaire	Hexa
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

Exemple 1 : Conversion d'un nombre binaire en hexadécimal

$$(1000\ \ 1100\ \ 0011\ \ 1111)_2 \rightarrow (\ 8\ \ C\ \ 3\ \ F\ )_{16}$$

Exemple 2 : Conversion d'un nombre hexadécimal en binaire

$$(\ A\ \ D\ \ 3\ \ F)_2 \rightarrow (1010\ \ 1101\ \ 0011\ \ 1111)_{16}$$

3.1. Conversion d'un nombre hexadécimal en décimal

La conversion hexadécimal-décimal consiste à multiplier chaque chiffre du nombre hexadécimal par 16 élevé à la puissance de sa position (en partant de la droite) et à additionner les résultats. Cette somme donne la valeur décimale équivalente. Par exemple, pour convertir le nombre « FFAB » en décimal, on applique la méthode suivante :

1- On décompose le nombre en ses chiffres individuels en utilisant la base 16.

$$FFAB = F \times 16^3 + F \times 16^2 + A \times 16^1 + B \times 16^0$$

2- On remplace les chiffres hexadécimaux (A, B, et F) par leurs équivalents décimaux.

$$F = 15 \quad ; \quad B = 11 \quad ; \quad A = 10$$

3- On effectuer le calcul.

$$\begin{aligned}(FFAB)_{16} &= F \times 16^3 + F \times 16^2 + A \times 16^1 + B \times 16^0 \\ &= 15 \times 16^3 + 15 \times 16^2 + 10 \times 16^1 + 11 \times 16^0 \\ &= 15 \times 4096 + 15 \times 256 + 10 \times 16 + 11 \times 1 \\ &- 61\,440 + 3\,840 + 160 + 11 \\ &= 65\,451\end{aligned}$$

Donc, "FFAB" en hexadécimal est équivalent à 65 451 en décimal.

3.2. L'addition en hexadécimal

```
   1 1
 1 A 5 E F
   9 0 B D
-----------
 1 3 6 A C
```

F + D = 15 + 13 = 28 ; 28 = 16 + 12 = 16 + C
E + B + 1 = 26 ; 26 = 16 + 10 = 16 + A
A + 9 = 19 ; 19 = 16 + 3 = 16 + 3

```
   1 1 1
 1 D D A 9
   3 A C E
-----------
 1 1 8 7 7
```

9 + E = 23 ; 23 = 16 + 7 = 16 + 7
A + C + 1 = 23 ; 23 = 16 + 7 = 16 + 7
D + A + 1 = 24 ; 24 = 16 + 8 = 16 + 8
D + 3 + 1 = 17 ; 17 = 16 + 1 = 16 + 1

3.3. La soustraction en hexadécimal

```
  A 3 B 7
- 9 0 1 6
---------
  1 3 A 1
```

```
    A 12 11 10
-   12 1B 1C F
--------------
    7  6  4  1
```

16 – 15 = 1
17 – 13 = 4
18 – 12 = 6
10 – 3 = 7

4. Un mot sur les unités de mesure

En informatique, **les unités de mesure** de l'information jouent un rôle fondamental pour évaluer **la taille des données ainsi que la capacité de stockage.**

- ✓ Cela commence par **le bit**, l'unité de base, qui peut être **0 ou 1**.
- ✓ **Huit bits forment un octet**, généralement utilisé pour représenter un caractère ou un petit nombre.

Lorsque on combine des octets, on obtient des unités plus grandes, telles que :

- 1 Ko (kilo-octet) = 1 024 octets (2^{10})
- 1 Mo (mégaoctet) = 1 024 Ko = 1 048 576 octets (2^{20})
- 1 Go (gigaoctet) – 1 024 Mo – 1 048 576 Ko – 1 073 741 824 octets (2^{30})

Ces unités sont couramment utilisées pour quantifier la taille des fichiers et de la mémoire. Ensuite, il y a le téraoctet (To), le pétaoctet (Po), l'exaoctet (Eo), et le zettaoctet (Zo), chaque unité étant 1 024 fois plus grande que la précédente. Ces unités sont utilisées pour quantifier la capacité de stockage de grands centres de données.

5. Circuits de commutation de l'information

5.1. Le multiplexeur

Un multiplexeur est un circuit logique qui permet de choisir l'une parmi plusieurs entrées et de la transmettre vers une seule sortie, en fonction d'un signal de commande. Les entrées du multiplexeur sont généralement notées D_0, D_1, D_2, ..., D_n, et une sortie, généralement notée Q. Le nombre d'entrées dépend du nombre de lignes de données que le multiplexeur peut choisir. Le signal de commande, généralement appelé "sélecteur" ou "sélecteur de données ou d'adresse", détermine quelle entrée est transmise à la sortie.

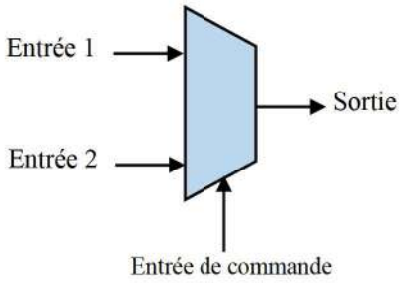
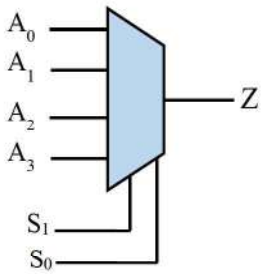


Fig. 1.1 : Schéma de principe d'un multiplexeur 2 vers 1.

Exemple d'un multiplexeur 4 vers 1 :



S_1	S_0	Z
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3

Fig. 1.2 : Schéma de principe et table de vérité du multiplexeur 4 vers 1.

Dans les système a microprocesseur, on retrouve généralement ce que l'on appelle un multiplexeur de bus. De la même manière qu'un multiplexeur permet de choisir entre plusieurs signaux, un multiplexeur de bus permettra de choisir entre plusieurs bus. Le bus est un ensemble de signaux ayant une fonction commune.

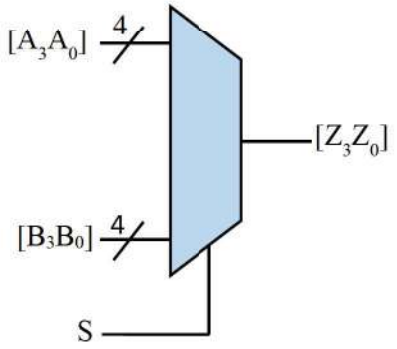
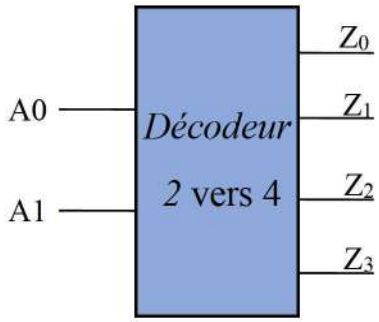


Fig. 1.3 : Schéma de principe d'un multiplexeur de bus.

5.2. Le décodeur

Un décodeur est un circuit logique qui établit une correspondance entre un code binaire d'entrée, composé de N bits et M lignes de sortie distinctes. Pour chaque combinaison possible des bits d'entrée, une seule sortie est activée.

Par exemple, un décodeur 2 vers 4 prend en entrée un nombre binaire sur 2 bits et active l'une des 4 sorties en fonction de la combinaison présente sur ses entrées. Chaque combinaison d'entrée correspond donc à une sortie unique validée.



A_1	A_0	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Fig. 1.4 : Décodeur 2 vers 4 (schéma de principe et table de vérité).

Dans les systèmes à microprocesseur, le décodeur est généralement utilisé dans les mémoires pour sélectionner une case mémoire à partir de son adresse, c'est ce qu'on appelle « décodeur d'adresse ».

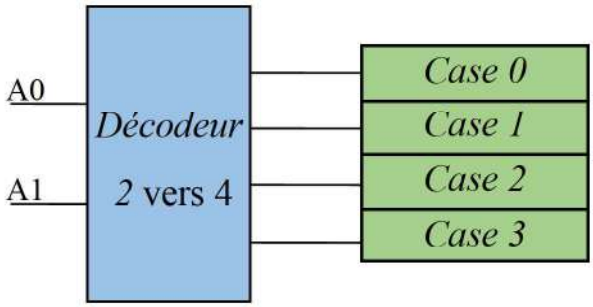


Fig. 1.5 : Schéma de principe d'un décodeur d'adresse.

Architecture de base d'un microprocesseur

Introduction

Le microprocesseur, souvent considéré comme le cerveau de tout ordinateur moderne, joue un rôle fondamental dans le fonctionnement des systèmes informatiques. Un système à microprocesseur est un ensemble électronique complexe qui intègre ce composant comme élément central. Le microprocesseur exécute des instructions programmées, ce qui lui permet de coordonner et de contrôler une multitude d'opérations au sein du système. Dans ce cours, nous allons explorer en détail l'architecture interne et externe du microprocesseur, ainsi que la manière dont l'information circule entre ses différentes parties via des bus de communication. Nous examinerons également le fonctionnement fondamental du microprocesseur, y compris sa programmation, ainsi que le cycle d'exécution des instructions, qui est essentiel pour comprendre comment un microprocesseur accomplit les tâches qui lui sont confiées.

1- Qu'est-ce qu'un microprocesseur ?

Un microprocesseur, également connu sous le nom de CPU (*Central Processor Unit*), est une puce électronique de petite taille, généralement en silicium, composée de nombreux transistors interconnectés. Sa principale fonction est d'exécuter les instructions et de traiter les données contenues dans un programme informatique. Il est généralement installé sur la carte mère de l'ordinateur, souvent dans un emplacement amovible appelé *socket*, et est équipé d'un dissipateur thermique et d'un ventilateur, car il est l'un des composants les plus énergivores de la carte mère. Il est à noter que le terme "microprocesseur" se réfère spécifiquement à ce type de processeur compact utilisé dans les ordinateurs personnels et les appareils électroniques grand public, tandis que "processeur" peut également désigner des unités centrales plus puissantes, comme celles dans les serveurs et supercalculateurs.

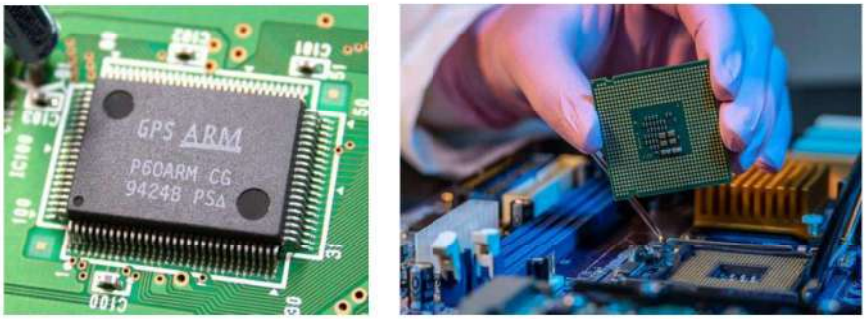


Fig. 2.1 : Le microprocesseur d'un ordinateur.

2- Architecture interne d'un microprocesseur

Un microprocesseur est essentiellement constitué de deux parties principales : **l'Unité de Calcule et l'Unité de Contrôle.** D'une part, l'unité de calcule est capable d'exécuter des instructions et de réaliser des opérations mathématiques et logiques. D'autre part, l'unité de contrôle supervise le déroulement du processus, décodant les instructions, coordonnant le

transfert de données et gérant les interruptions. Ensemble, ces deux composants jouent un rôle crucial dans le fonctionnement d'un microprocesseur, permettant l'exécution de programmes et le traitement de données avec efficacité.

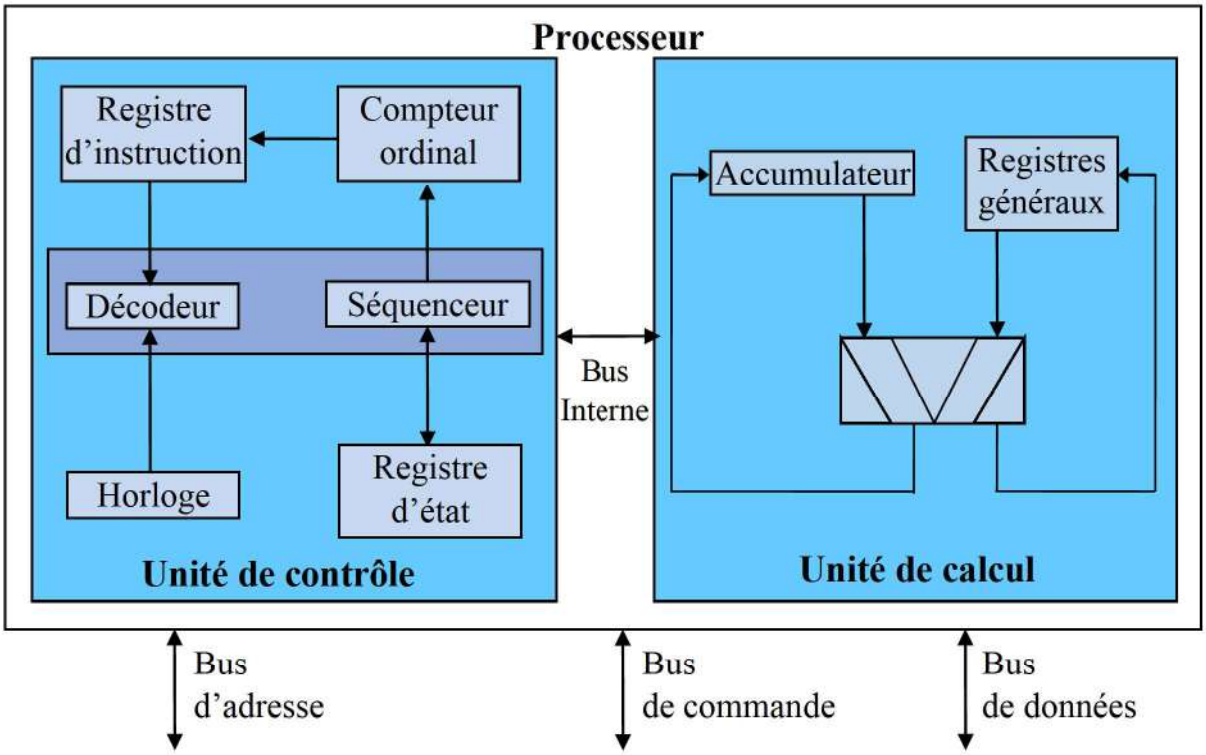


Fig. 2.2 : Structure interne d'un microprocesseur centrale.

L'unité de contrôle (*Control Unit* - CU) est la partie la plus complexe du processeur. Elle est responsable de la coordination de plusieurs composants essentiels :

- **Registre d'instruction :** il contient l'instruction que le processeur est en train d'exécuter. Les instructions, stockées dans la mémoire centrale, sont chargées une par une dans ce registre pour être traitées.
- **Décodeur :** il interprète l'instruction présente dans le registre d'instruction afin de déterminer les opérations que le processeur doit réaliser.
- **Séquenceur :** il organise et ordonne les différentes opérations élémentaires nécessaires à l'exécution de l'instruction, généralement à l'aide d'un microprogramme.
- **Compteur ordinal (Program Counter) :** ce registre particulier contient l'adresse de l'instruction suivante à exécuter, permettant au processeur de suivre le flux du programme.
- **Horloge :** elle détermine le rythme d'exécution des instructions en fournissant un signal périodique régulier au processeur.
- **Registre d'état :** similaire au compteur ordinal, ce registre reflète l'état actuel du processeur. Il signale, par exemple, des situations « exceptionnelles » comme un dépassement de capacité lors d'une addition (*overflow*) grâce à un bit spécifique.

** Le registre est un composant de mémoire de petite taille et de haute vitesse situé à l'intérieur du microprocesseur, utilisé pour stocker temporairement des données et des instructions lors de l'exécution d'un programme.*

L'unité de calcul est responsable de l'exécution de toutes les opérations arithmétiques et logiques. Grâce à cette unité, l'ordinateur peut traiter des structures de contrôle contenant des conditions et effectuer les calculs nécessaires à l'exécution des programmes. L'unité de calcul se compose de plusieurs éléments :

- **Accumulateur :** c'est le registre principal de calcul. Toutes les données entrant dans une opération ainsi que les résultats produits transitent par l'accumulateur.
- **Registres généraux :** ces registres viennent en appui de l'accumulateur pour stocker temporairement plusieurs termes ou états intermédiaires nécessaires au traitement des calculs. Leur nombre peut varier selon l'architecture du processeur.
- **Unité arithmétique et logique (*ALU – Arithmetic and Logic Unit*) :** elle réalise l'ensemble des opérations arithmétiques (addition, soustraction…) et logiques (ET, OU, NON, XOR…).

3- Architecture externe d'un microprocesseur

Outre les éléments internes qui composent le cœur du microprocesseur et lui permettent d'exécuter des instructions, un système à microprocesseur repose sur une interaction harmonieuse entre le microprocesseur et une gamme d'éléments externes. Ces éléments externes jouent un rôle tout aussi essentiel dans le fonctionnement global du système. Ils facilitent l'acquisition de données, la communication avec le monde extérieur, le stockage à long terme de programmes et de données, ainsi que la gestion de l'ensemble du système. Voici une liste des principaux éléments externes clés qui interagissent de manière étroite et coordonnée avec le microprocesseur.

- **Mémoire centrale (RAM) :** Stockage temporaire d'instructions et de données pour l'exécution par le microprocesseur.
- **Mémoire ROM :** Stocke des données en lecture seule, telles que les instructions de démarrage et des données essentielles au fonctionnement de l'ordinateur.
- **Mémoire de masse (Disque Dur, SSD) :** Stockage à long terme de données utilisateur, des applications, des systèmes d'exploitation, etc.
- **Périphériques d'entrée/sortie (Clavier, Souris, Écran, Imprimante, etc.) :** Communication avec le monde extérieur pour l'entrée et la sortie de données.
- **Bus de Données, Bus d'Adresse et Bus de Contrôle :** Permettent au microprocesseur de communiquer avec la mémoire centrale, les périphériques et d'autres composants.
- **Horloge système :** Fournit un signal d'horloge régulier pour synchroniser les opérations du microprocesseur.
- **Unité de gestion de la puissance :** Gère la consommation d'énergie et la gestion de la puissance du système.

- **Carte mère :** Connecte tous les composants du système, y compris le microprocesseur, la mémoire centrale, les périphériques, etc.

4- Circulation de l'information dans un système à microprocesseur

Le microprocesseur communique avec la mémoire et les périphériques d'E/S en échangeant des **mots binaires** via un ensemble de **connexions appelé bus**. Un bus permet le transfert de données de manière parallèle, c'est-à-dire que plusieurs bits (*n* bits) circulent simultanément. La capacité d'un microprocesseur à transférer des données avec la mémoire et les périphériques dépend de la longueur maximale des mots binaires qu'il peut traiter : on distingue ainsi les microprocesseurs 8 bits, 16 bits, 32 bits, etc. On distingue généralement trois types de bus :

a) Bus de Données (*Data Bus*)

- Le bus de données est le canal par lequel les informations circulent entre le microprocesseur, la mémoire, les registres et d'autres composants du système.
- Il est bidirectionnel, ce qui signifie qu'il permet à la fois la lecture et l'écriture des données. Celles-ci peuvent être des instructions du programme, des valeurs de données ou des résultats de calcul.
- La largeur du bus de données détermine le nombre de bits pouvant être transférés à chaque cycle d'horloge. Par exemple, un bus de 32 bits peut transférer 32 bits simultanément à chaque cycle.

b) Bus d'Adresse (*Address Bus*)

- Le bus d'adresse est utilisé pour spécifier l'emplacement mémoire ou l'adresse d'un périphérique vers lequel les données doivent être lues ou écrites.
- Il est unidirectionnel, car il est principalement utilisé par le microprocesseur pour adresser des emplacements mémoire ou des périphériques.
- La largeur du bus d'adresse détermine la plage d'adresses que le microprocesseur peut atteindre. Par exemple, un bus d'adresse de 16 bits peut adresser 2^{16} (65536) emplacements mémoire différents.

c) Bus de Contrôle (*Control Bus*)

- Le bus de contrôle est utilisé pour transmettre des signaux de contrôle entre le microprocesseur et d'autres composants du système tels que la mémoire et les périphériques d'E/S.
- Il transporte des signaux tels que les signaux de lecture/écriture, les signaux de validation (pour indiquer que les données sont valides), les signaux d'interruption, les signaux d'horloge, etc.
- Le bus de contrôle coordonne le flux des données et des instructions à travers le système et assure la synchronisation des opérations.

4-1- Architecture Von Neumann et Harvard

Il est à préciser qu'en fonction de la configuration du bus de transfert (instructions et données), deux architectures fondamentales peuvent être définies dans le domaine des microprocesseurs : l'architecture Von Neumann lorsque seulement un seul bus est partagé pour le transfert des instructions et des données, et l'architecture Harvard lorsqu'il y a deux bus distincts, l'un réservé aux instructions et l'autre aux données.

- Dans **l'architecture de Von Neumann**, le microprocesseur utilise un bus unique pour accéder à la mémoire centrale, ce qui signifie qu'il lit à la fois les instructions et les données sur le même canal. Ainsi, instructions et données partagent la même mémoire. Cette architecture est la plus répandue dans les ordinateurs personnels et les systèmes informatiques généraux en raison de sa simplicité de conception.
- En revanche, **l'architecture Harvard** sépare complètement la mémoire des instructions de celle des données en utilisant deux bus distincts. Cette organisation permet au microprocesseur d'accéder simultanément aux instructions et aux données, offrant ainsi de meilleures performances, notamment dans les systèmes embarqués et les microcontrôleurs.

Toutefois, l'architecture la plus couramment utilisée dans les microprocesseurs modernes est l'architecture de type Von Neumann, dans laquelle un seul bus est partagé pour le transfert des instructions et des données. La plupart des ordinateurs personnels et des systèmes informatiques actuels sont basés sur cette architecture.

5- Organisation de la mémoire dans un système à microprocesseur

La mémoire est constituée de cellules individuelles, appelées cases mémoire, qui stockent des informations, qu'il s'agisse d'instructions ou de données. Chaque case mémoire est identifiée par un numéro unique appelé adresse. Une case mémoire peut être lue et écrite par le microprocesseur (comme c'est le cas des mémoires vives RAM) ou seulement lue (comme c'est le cas des mémoires mortes ROM).

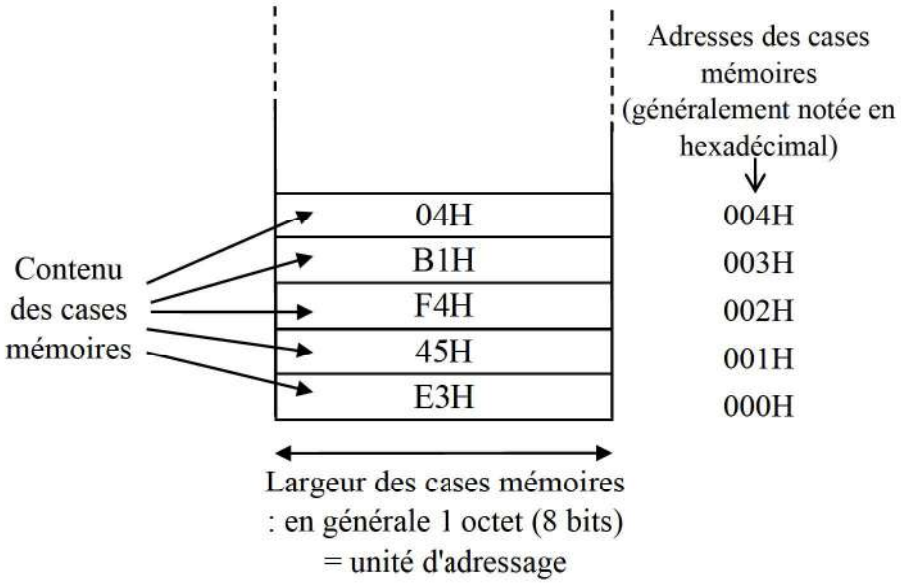


Fig. 2.3 : Représentation d'une mémoire centrale.

6- Programmation d'un microprocesseur

Le rôle d'un microprocesseur est d'exécuter des programmes. Un programme est constitué d'une suite d'instructions, généralement rédigées ligne par ligne. La complexité de ces instructions dépend du langage de programmation utilisé. Par exemple, dans un langage de bas niveau comme l'assembleur, une instruction réalise une tâche élémentaire, telle qu'une opération d'addition. En revanche, dans un langage de haut niveau comme le C++, une seule instruction peut correspondre à plusieurs actions qui nécessiteraient plusieurs instructions en assembleur.

Lors de la rédaction d'un programme, on peut choisir parmi différents langages de programmation tels que l'Assembleur, le *Basic/QBasic,* le *Pascal,* le *C/C++,* le *Visual Basic,* le *Visual C++,* le *Delphi,* le *Java*, et bien d'autres. Les lignes de code écrites constituent le programme source, généralement enregistré dans un fichier texte avec une extension propre au langage utilisé (par exemple, *.pas* pour le Pascal, *.cpp* pour le C++, etc.).

Ces programmes sources sont lisibles par les programmeurs mais pas directement compréhensibles par le microprocesseur. Pour que le processeur puisse exécuter ces instructions, il faut les traduire en langage machine, c'est-à-dire en une suite de codes binaires. Cette traduction est effectuée par un compilateur, spécifique à chaque langage, qui transforme le programme source en programme exécutable que le processeur peut interpréter. Il est important de noter que, quel que soit le langage utilisé, tous les programmes exécutables présentent la même structure pour le processeur.

Le principal avantage du langage assembleur réside dans le fait que chaque instruction correspond directement à une instruction élémentaire du processeur. Programmer en assembleur revient donc à travailler presque directement en langage machine, offrant un contrôle total sur le fonctionnement du processeur.

7- Le cycle d'exécution des instructions par le microprocesseur

Les instructions sont exécutées par l'unité de contrôle suivant un ensemble d'étapes précises, notamment :

1. Chargement de l'instruction à exécuter depuis la mémoire dans **le registre d'instruction**.
2. Interprétation de cette instruction par **le décodeur** pour comprendre la tâche à accomplir.
3. Ordonnancement des opérations nécessaires pour exécuter l'instruction grâce à un **séquenceur**.
4. Exécution effective de l'instruction, y compris les opérations arithmétiques ou logiques, par **l'unité arithmétique et logique (ALU).**
5. Sauvegarde de l'adresse de l'instruction suivante dans **le compteur ordinal.**

La coordination de toutes les étapes mentionnées dans le cycle d'exécution des instructions est synchronisée par un signal d'horloge. L'horloge est un oscillateur électronique intégré dans le processeur qui émet des signaux électriques périodiques à une fréquence donnée, mesurée en hertz (Hz). Chaque signal d'horloge marque le début d'un nouveau cycle d'horloge, ce qui permet au processeur de passer à l'étape suivante de l'exécution de l'instruction.

La durée d'exécution d'une instruction, exprimée en cycles d'horloge, dépend de la complexité de l'instruction et de l'architecture du processeur. Chaque instruction requiert un certain nombre de cycles d'horloge (ou microcycles) pour être exécutée. Par exemple, une instruction simple comme une addition peut nécessiter un ou deux cycles, tandis qu'une instruction plus complexe, telle qu'une multiplication ou une division, peut en demander plusieurs.

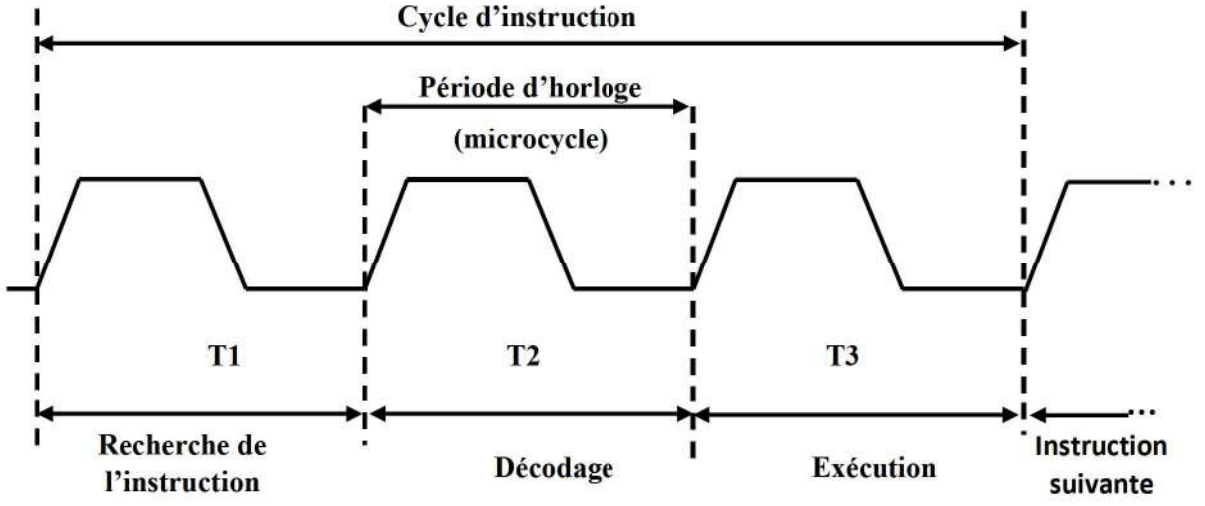


Fig. 2.4 : Représentation du cycle d'instruction d'un microprocesseur.

Pour calculer la durée d'exécution d'une instruction en fonction du nombre de cycles d'horloge, on utilise la formule suivante :

Durée d'exécution (en secondes) = Nombre de cycles d'horloge / Fréquence de l'horloge

Exemple : Si la fréquence de l'horloge (f_clk) est de 5 MHz, la période d'une horloge individuelle (T) est alors de $1/f$ = 0.2 µs. Si une instruction s'exécute en 3 microcycles, la durée totale d'exécution de cette instruction est :

$$3 \times 0{,}2\ \mu s = 0.6\ \mu s$$

Cette formule prend en compte le nombre de cycles d'horloge nécessaires pour l'exécution de l'instruction et la fréquence à laquelle l'horloge émet des signaux (mesurée en Hz). Le résultat nous donne la durée en secondes. En général, plus la fréquence de l'horloge est élevée, plus les instructions sont exécutées rapidement, mais cela peut également augmenter la consommation d'énergie et la chaleur générée par le processeur.

Le microprocesseur 8086

Introduction

Le microprocesseur 8086, également connu sous la désignation x86, représente un jalon essentiel dans l'histoire de la technologie des microprocesseurs. Introduit par Intel en 1978, le 8086 a été l'un des premiers microprocesseurs à architecture x86, qui a ensuite évolué pour devenir la base des processeurs utilisés dans la plupart des ordinateurs personnels. Son architecture a marqué le début de l'ère des processeurs x86, qui est toujours prédominante aujourd'hui. Le 8086 a apporté des avancées significatives en termes de puissance de calcul, de mémoire et de capacités d'adressage, contribuant ainsi à façonner l'informatique moderne.

1. Description physique du 8086

Le microprocesseur Intel 8086 est un circuit intégré (ou puce) de forme rectangulaire et plate, généralement logé dans un boîtier en céramique ou en plastique. Introduit en 1978, il constitue le premier processeur de la famille Intel 80x86, suivi par d'autres modèles comme le 80186, 80286, 80386, 80486 et les Pentium. Ce microprocesseur est présenté dans un boîtier à double rangée de broches (DIP, *Dual In-line Package*) comportant 40 broches.

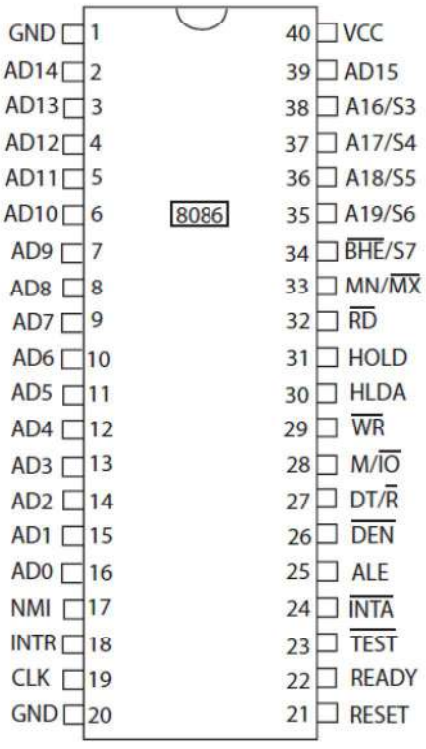


Fig. 3.1 : Architecture externe et brochage du microprocesseur 8086.

2. Description et utilisation des signaux du 8086

Le microprocesseur 8086 est équipé de 40 broches, chacune ayant un rôle spécifique dans son fonctionnement. Dans ce qui suit, nous explorerons en détail l'utilisation de ces signaux

pour mieux comprendre comment le 8086 communique avec d'autres composants et exécute les instructions.

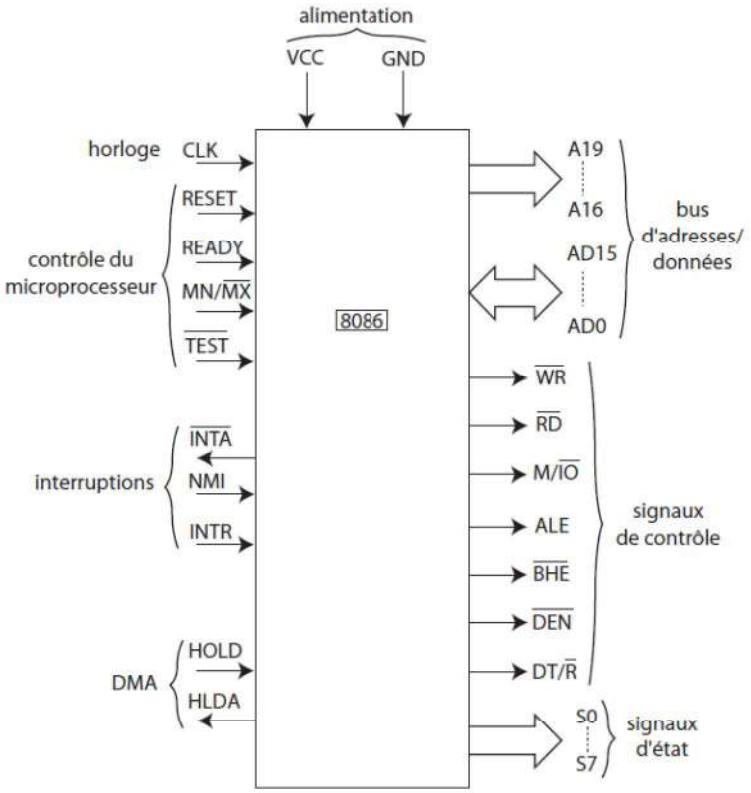
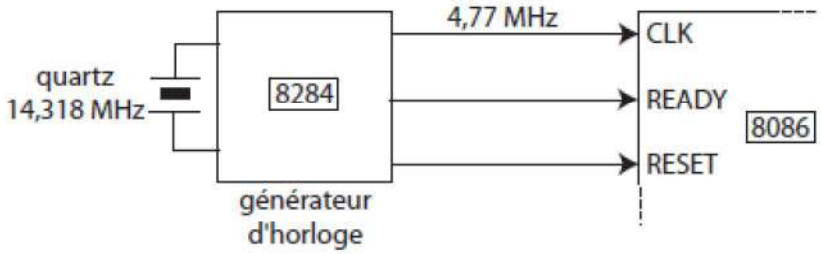


Fig. 3.2 : Description et utilisation des signaux du microprocesseur 8086.

- **CLK** : entrée du signal d'horloge qui rythme le fonctionnement du microprocesseur. Ce signal est fourni par le générateur d'horloge 8284.



- **RESET** : entrée de remise à zéro du microprocesseur. Lorsque cette entrée est activée (mise à l'état haut) pendant au moins 4 périodes d'horloge, le microprocesseur est réinitialisé. Il commence alors l'exécution à l'adresse FFFF0H, appelée adresse de *bootstrap*, correspondant au point de départ du processus de démarrage du 8086. Le signal RESET est également fourni par le générateur d'horloge.
- **READY :** entrée de synchronisation avec la mémoire. Elle permet au microprocesseur de s'assurer que les données provenant de la mémoire sont prêtes à être traitées. Ce signal est fourni par le générateur d'horloge.
- **TEST :** entrée permettant de tester ou de mettre en attente le microprocesseur en fonction d'un événement externe.

- **MN/MX :** entrée de choix du mode de fonctionnement du microprocesseur :
  - Mode minimum (MN/MX = 1) : le 8086 fonctionne de manière autonome, il génère lui-même le bus de commande (*RD*, *WR*, etc.) ;
  - Mode maximum (MN/MX = 0) : le 8086 fonctionne en association avec d'autre contrôleur. Ce mode permet de réaliser des systèmes multiprocesseurs.
- **NMI et INTR :** ce sont des entrées de demande d'interruption. Le signal **INTR** correspond à une interruption ordinaire, tandis que **NMI** correspond à une interruption non masquable et prioritaire. **Une interruption** est un événement externe ou interne qui provoque l'arrêt temporaire de l'exécution normale d'un programme par le microprocesseur afin de traiter cet événement. Une fois le traitement terminé, le microprocesseur reprend l'exécution du programme initial.
- $\overline{\text{INTA}}$ **:** signal de sortie indiquant que le microprocesseur a accepté la demande d'interruption et qu'il s'apprête à la traiter.
- **HOLD et HLDA :** signaux de demande d'accord d'accès direct à la mémoire du microprocesseur par les périphériques externes.
- **S0 à S7 :** ce sont des signaux d'état qui indiquent le type d'opération en cours sur le bus d'adresses et le bus de données lorsque le microprocesseur communique avec la mémoire ou les périphériques d'entrée/sortie.
- **Les lignes A_{19}-A_0 et D_{15}-D_0 :** elles constituent respectivement le bus d'adresses et le bus de données. Elles sont multiplexées, ce qui signifie qu'elles partagent les mêmes broches. Les lignes AD_{15} à AD_0 servent à la fois pour les données et les adresses, tandis que les lignes A_{19} à A_{16} sont réservées exclusivement aux adresses :
  - Bus de données sur 16 bits (microprocesseur 16 bits) ;
  - Bus d'adresses sur 20 bits, ce qui permet d'adresser 2^{20} = 1 Mo de mémoire avec le microprocesseur 8086.

3. Organisation interne du microprocesseur 8086

Le microprocesseur 8086 est l'un des microprocesseurs les plus célèbres et largement utilisés de l'histoire de l'informatique. Il se caractérise par une architecture sophistiquée composée de deux unités collaboratives :

- **L'unité d'interface de bus (BIU)** assume la responsabilité de la recherche et du chargement des instructions depuis la mémoire. Elle les stocke soigneusement dans une file d'attente, prêtes à être traitées.
- Pendant ce temps, **l'unité d'exécution (EU)** se consacre à l'exécution des instructions contenues dans cette file d'attente.

Ces deux unités fonctionnent simultanément, créant ainsi un processus fluide et continu.

De plus, le microprocesseur 8086 tire sa puissance de ses registres internes, des emplacements de stockage de données cruciaux qui jouent un rôle central dans l'exécution des opérations et le traitement des informations.

4. Les registres du microprocesseur 8086

Les registres dans les microprocesseurs sont de petits emplacements de stockage de données à haute vitesse intégrés directement dans l'unité centrale de traitement (CPU) du microprocesseur. Ils sont utilisés pour stocker temporairement des informations importantes, telles que des nombres, des adresses mémoire ou des indicateurs de statut, pendant l'exécution des opérations de calcul et de traitement de données. Les registres permettent au processeur de travailler rapidement avec des données en les conservant à portée de main, ce qui accélère le fonctionnement de l'ordinateur.

Le microprocesseur 8086 dispose de **14 registres** soigneusement répartis en quatre groupes distincts.

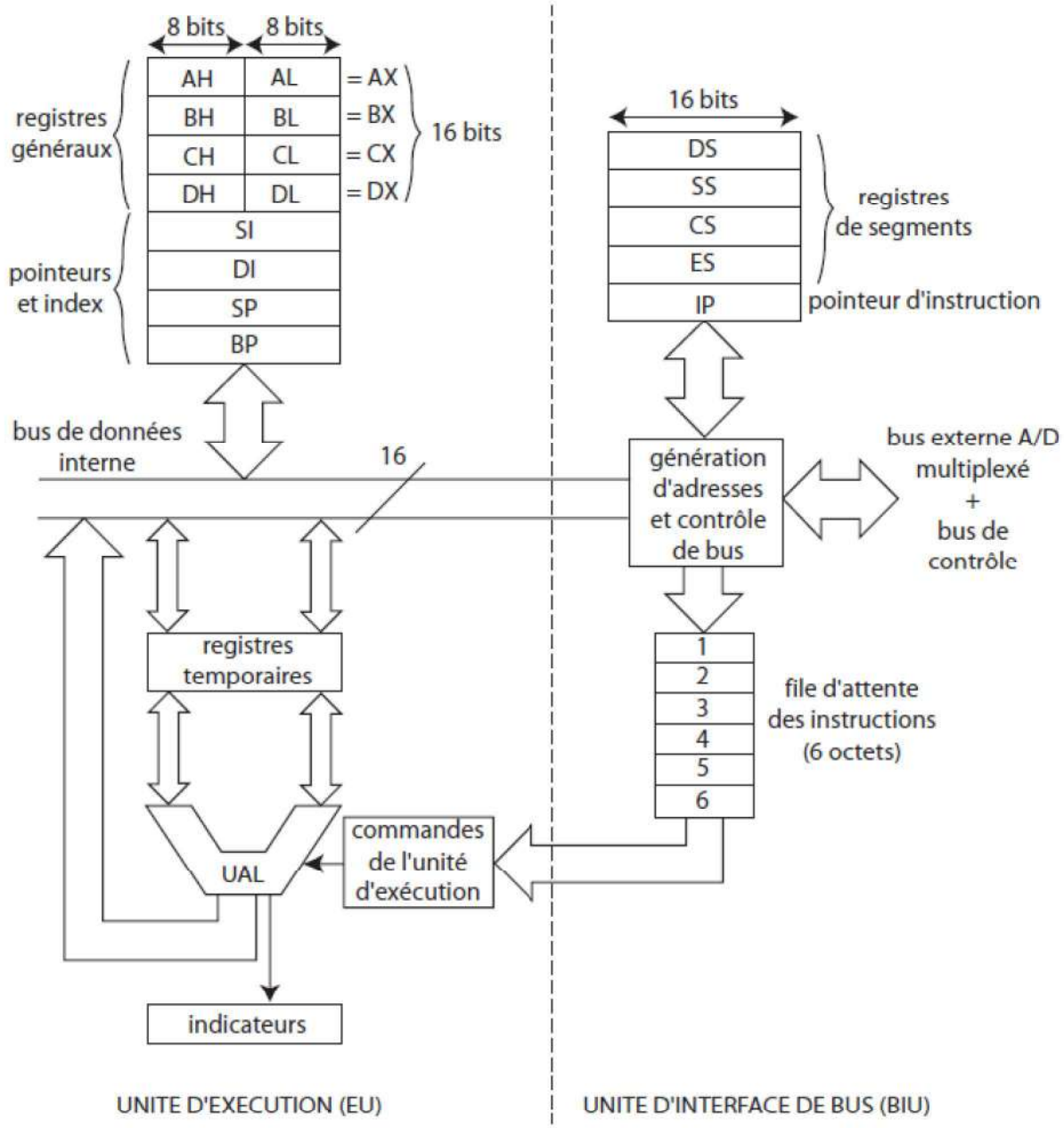


Fig. 3.3 : Organisation interne du microprocesseur 8086.

4.1. Registres généraux : *4 registres sur 16 bits*

Les registres généraux, appelés également registres de données, sont principalement utilisés pour effectuer des opérations arithmétiques et logiques. Ils servent à stocker temporairement des valeurs pendant l'exécution des instructions. Dans le cas du microprocesseur 8086, ces registres sont AX, BX, CX et DX, chacun ayant une taille de 16 bits. Comme indiqué dans le tableau suivant, chaque registre peut être divisé en deux parties de 8 bits, permettant ainsi de manipuler séparément des octets individuels.

	15 8	7 0
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
	Bits de poids élevé	*Bits de poids faible*

Ces subdivisions permettent de stocker et de manipuler des octets individuels au sein de ces registres, ce qui offre une grande flexibilité pour le traitement des données dans le microprocesseur 8086.

4.2. Registres de pointeurs et d'index : *4 registres sur 16 bits*

Les registres de pointeurs et d'index sont essentiels en programmation assembleur pour gérer la mémoire et faciliter les opérations sur les données. Ils contiennent des adresses mémoire importantes.

- **Le Pointeur de Pile (*Stack Pointer, SP*) :** Il s'agit d'un pointeur de la pile qui indique toujours le sommet de la pile où les données sont empilées ou retirées. **La pile** est une zone de mémoire utilisée pour stocker temporairement des données pendant l'exécution d'un programme. *SP* pointe toujours vers le sommet de la pile, indiquant où la prochaine donnée sera empilée ou déployée. Lorsqu'une valeur est poussée (PUSH) sur la pile, SP est décrémenté, et lorsqu'une valeur est retirée (POP) de la pile, SP est incrémenté.
- **Pointeur de Base (*Base Pointer, BP*) :** Le registre BP est principalement utilisé pour accéder à des données stockées sur la pile.
- **Index Source et Destination (*Source Index, SI et Destination Index, DI*) :** Les registres SI et DI sont utilisés pour effectuer des opérations sur des chaînes de caractères ou des tableaux. SI est généralement utilisé comme index source lors de la copie de données depuis une source vers une destination, tandis que DI est utilisé comme index de destination. Ces registres sont essentiels pour les boucles de transfert de données, tels que les opérations de copie de mémoire, et simplifient le traitement de données en série.

4.3. Pointeur d'instruction et indicateurs (flags) : *2 registres sur 16 bits*

- **Le pointeur d'instruction IP :** également appelé compteur ordinal, c'est un registre qui s'incrémente automatiquement après chaque instruction. Il contient en permanence l'adresse de la prochaine instruction à exécuter. Effectuer un saut dans un programme consiste à charger dans le registre IP l'adresse de la nouvelle instruction à exécuter. L'incrémentation du pointeur d'instruction se fait ensuite à partir de cette nouvelle adresse afin de poursuivre l'exécution du programme.
- **Le registre d'état FLAG :** ce registre contient l'état du processeur après l'exécution des opérations. Il regroupe plusieurs indicateurs (flags) qui signalent certaines conditions particulières résultant d'une opération arithmétique ou logique, comme un résultat nul, un dépassement de capacité ou une retenue.

Registre d'état FLAG



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

15															0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

1- CF (*Carry Flag*) :

Retenue : cet indicateur et mis à 1 lorsqu'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes.

Exemple : opération d'addition sur 8 bits

CF=0

```
  1 0 0 1 0 1 1 0
+ 0 1 0 1 0 1 0 0
-----------------
  1 1 1 0 1 0 1 0
```

CF=1

```
  1 1 0 1 1 0 0 1
+ 0 1 0 1 0 0 1 0
-----------------
  0 0 1 0 1 0 1 1
```

2- PF (*Parity Flag*) :

Parité : cet indicateur permet de vérifier la parité du résultat d'une opération. Il est mis à 1 lorsque le résultat contient un nombre pair de bits à 1. Dans le cas contraire, il est mis à 0.

3- AF (*Auxiliary Carry*) :

Demie retenue : Ce bit est égal à 1 si on a une retenue du quartet de poids faible dans le quartet de poids plus fort.

4- ZF (*Zero Flag*) :

Zéro : cet indicateur est mis à 1 lorsque le résultat d'une opération est nul. Après une opération de soustraction ou une instruction de comparaison, si ZF = 1, cela signifie que les deux opérandes sont égaux. Sinon, il est mis à 0.

5- SF (*Sign Flag*) :

Signe : cet indicateur prend la valeur 1 lorsque le bit de poids fort du résultat est égal à 1, ce qui indique un résultat négatif dans le cas des nombres signés. Dans le cas contraire, il est mis à 0.

6- OF (*Overflow Flag*) :

Débordement : si on a un débordement arithmétique ce bit est mis à 1. Cela se produit lorsque le résultat d'une opération dépasse la capacité d'un registre ou d'une case mémoire. Par exemple, dans un registre de 8 bits, si vous additionnez 120 et 140, le résultat est supérieur à ce que peut stocker un registre de 8 bits, donc OF serait mis à 1 pour indiquer ce débordement.

7- DF (*Direction Flag*) :

Auto Incrémentation/Décrémentation : utilisée pendant les instructions de chaîne de caractères pour auto incrémenté ou auto décrémenter les opérations de traitement de données. Lorsque DF est à 0, les opérations de chaînes s'effectuent de manière ascendante (du bas vers le haut). Lorsque DF est à 1, elles s'effectuent de manière descendante (du haut vers le bas). Cela permet un traitement plus efficace des tableaux et des chaînes de caractères.

8- IF (*Interrupt Flag*) :

Masque d'interruption : pour masquer les interruptions venant de l'extérieur ce bit est mis à 0, dans le cas contraire le microprocesseur reconnaît l'interruption de l'extérieur.

9- TF (*Trap Flag*) :

Piège : pour que le microprocesseur exécute le programme pas à pas. En effet, lorsque TF est à 1, le microprocesseur exécute chaque instruction individuellement, ce qui permet aux programmeurs de suivre l'exécution du programme à des fins de débogage. Cette fonction est précieuse pour identifier les erreurs dans le code.

4.4. Registres de segments : *4 registres sur 16 bits*

Ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le début de chacun d'entre eux. Chaque segment de mémoire représente une plage d'emplacements mémoire et est utilisé pour stocker différents types de données et de programmes.

- **CS (*Code Segment* - Segment de Code) :** Il pointe sur le segment qui contient les codes des instructions du programme en cours. En outre, il définit la zone mémoire contenant les instructions à exécuter.
- **DS (*Data Segment* - Segment de Données) :** Il pointe vers le segment de données, qui est utilisé pour stocker des données telles que des variables, des tableaux et d'autres informations de programme.
- **ES (*Extra Segment* - Segment Supplémentaire) :** ES est un registre de segment supplémentaire qui peut être utilisé pour diverses opérations, notamment le stockage de données supplémentaires.
- **SS (*Stack Segment* - Segment de Pile) :** ce registre contient l'adresse de base de la pile. La pile est une zone mémoire utilisée pour stocker temporairement des informations telles que des registres, des adresses de retour ou des données. Ces informations peuvent ensuite être récupérées après l'exécution d'un sous-programme ou lors du traitement d'une interruption.

5. Segmentation de la mémoire dans le microprocesseur 8086

Le microprocesseur 8086 est équipé d'un bus d'adresses de 20 bits, ce qui lui confère la capacité d'adresser un total de 2^{20} ou 1 048 576 emplacements mémoire différents, soit l'équivalent de **1 Mo** (mégaoctet).

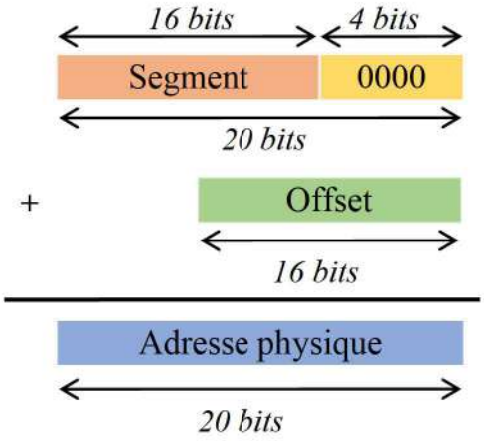
Cependant, une limitation majeure réside dans le fait que les registres du 8086 sont des registres de seulement 16 bits, ce qui signifie qu'ils ne peuvent adresser directement qu'une portion de la mémoire, limitée à 64 Ko. Pour surmonter cette limitation, Intel a introduit le concept de segmentation. La mémoire totale de 1 Mo a été subdivisée en segments de 64 Ko chacun. Chaque segment est identifié par un registre de segment, notamment les registres : CS, DS, ES et SS.

Pour accéder à une case mémoire précise, le microprocesseur utilise deux registres. Le premier registre sert à indiquer le segment mémoire, il est appelé registre de segment. Le second registre permet de localiser la position à l'intérieur de ce segment, il est appelé registre de déplacement ou offset. L'association de ces deux valeurs constitue ce que l'on appelle une adresse logique, généralement exprimée sous la forme d'un couple :

Segment : offset

L'adresse réellement transmise sur le bus d'adresses A_0 - A_{19} est ce que l'on appelle l'adresse physique, laquelle est constituée de 20 bits.

Correspondance entre l'adresse logique et l'adresse physique :



Ainsi, l'adresse physique est obtenue à partir de la relation suivante :

Adresse physique = (segment × 16) + offset

Il est à noter que par défaut, chaque registre segment est associé à un ou plusieurs registres d'offset afin de former une adresse physique réelle, comme indiqué dans le tableau de correspondance suivant :

Registre de segment	Registres d'offset associés
CS	**IP**
SS	**SP et/ou BP**
DS	**SI et/ou DI**
ES	**Selon les besoins du programmeur**

Ces associations simplifient la gestion des adresses mémoire dans les programmes assembleur x86, mais il est important de noter que d'autres combinaisons sont possibles en fonction des besoins spécifiques d'un programme.

Exemples :

1- Pour CS = 8000H et IP = 4004H, l'adresse physique CS:IP est calculer comme suit :

Adresse physique = 84004H

$$\begin{array}{r} 80000 \\ +\ 4004 \\ \hline 84004 \end{array}$$

2- Pour DS = FAB3H et DI = 2334H, l'adresse physique DS:DI est calculer comme suit :

Adresse physique = FCE64H

$$\begin{array}{r} FAB30 \\ +\ 2334 \\ \hline FCE64 \end{array}$$

Introduction au jeu d'instructions et interruptions

Introduction

Le jeu d'instructions d'un microprocesseur est un ensemble spécifique d'instructions que la CPU peut exécuter. Une instruction est caractérisée par un code opératoire, représenté sous forme binaire, qui est difficilement exploitable directement par l'être humain. Pour rendre ces instructions plus compréhensibles pour les programmeurs, on utilise des mnémoniques, une notation symbolique, pour faciliter l'écriture de programmes en langage assembleur.

En fonction du nombre d'instructions reconnues par le microprocesseur, on peut les classer en deux grandes catégories :

- **Les microprocesseurs CISC (*Complex Instruction Set Computer*) :** entre 75 et 150 instructions.
- **Les microprocesseurs RISC (*Reduced Instruction Set Computer*) :** entre 10 et 30 instructions.

1. Structure d'une instruction machine

La structure d'une instruction machine x86 (8086) peut être décomposée comme suit :

[Étiquette :] Mnémonique [Opérandes] [; Commentaire]

- **L'étiquette** est optionnelle et précède généralement l'instruction, suivie d'un deux-points (:). Elle sert à marquer une adresse mémoire ou un point de saut (une référence) dans le programme. Exemple : *"start:"*.
- **Le mnémonique** (appelé aussi *Opcode*) est l'opération à effectuer. C'est généralement le nom de l'instruction, comme "MOV" pour déplacer des données ou "ADD" pour additionner.
- **Les opérandes** sont facultatifs et spécifient les données sur lesquelles l'instruction doit agir. Ils peuvent être des registres, des valeurs immédiates, des adresses mémoire, etc. Le nombre d'opérandes peut varier en fonction de l'instruction. Certaines n'en nécessitent pas (par exemple, les instructions NOP), tandis que d'autres en utilisent un ou plusieurs.
- **Le commentaire** est facultatif et permet d'expliquer brièvement l'objectif ou le comportement de l'instruction. Il est précédé par un point-virgule (;).

2. Instructions de base en langage assembleur x86

Les instructions de base en langage assembleur x86 sont des commandes qui permettent de réaliser des opérations fondamentales telles que le transfert de données, les opérations arithmétiques et les opérations logiques. Voici quelques-unes des instructions les plus couramment utilisées :

2.1. Instruction de transfert de données

Les instructions de transfert de données en langage assembleur x86 sont utilisées pour déplacer des données entre registres, la mémoire et d'autres emplacements de données. Voici quelques-unes des instructions de transfert de données couramment utilisées avec des exemples :

1- **MOV (*Move*) :** L'instruction MOV est utilisée pour déplacer des données d'une source vers une destination.

- Déplacer une valeur immédiate (constante) dans un registre :

```
MOV AX, 42          ; Charge la valeur 42 dans le registre AX
```

- Déplacer des données d'un registre à un autre :

```
MOV BX, AX          ; Copie la valeur du registre AX dans BX
```

- Déplacer des données de la mémoire vers un registre :

```
MOV CX, [1000H]  ;  Copie la valeur stockée à l'adresse mémoire 1000H dans CX
```

2- **XCHG (*Exchange*) :** L'instruction XCHG est utilisée pour échanger les valeurs de deux opérandes.

```
XCHG AX, BX         ; Échange les valeurs des registres AX et BX
```

3- **LEA (*Load Effective Address*) :** L'instruction LEA est utilisée pour charger l'adresse effective (plutôt que la valeur) d'une source dans une destination.

```
LEA SI, [BX+DI+2]       ; Charge l'adresse calculée (BX+DI+2) dans le registre SI
```

4- **PUSH et POP :** Ces instructions sont utilisées pour empiler (PUSH) et dépiler (POP) des données à partir de la pile.

```
PUSH AX         ; Empile la valeur du registre AX sur la pile
POP BX          ; Dépile la valeur du dessus de la pile dans le registre BX
```

2.2. Instructions arithmétiques

Ces exemples montrent comment effectuer diverses opérations arithmétiques en langage assembler x86. Chacune de ces instructions a des utilisations spécifiques pour effectuer des calculs arithmétiques sur les données stockées dans les registres et/ou en mémoire.

1- **ADD (Addition) :** L'instruction ADD est utilisée pour effectuer une addition entre deux opérandes.

- Addition de deux registres

```
ADD AX, BX          ; Ajoute la valeur de BX à la valeur de AX
```

- Addition d'une valeur immédiate à un registre

```
ADD CX, 10          ; Ajoute 10 à la valeur du registre CX
```

2- **SUB (Soustraction) :** L'instruction SUB est utilisée pour effectuer une soustraction entre deux opérandes.

- Soustraction de deux registres

SUB AX, BX ***; Soustrait la valeur de BX de la valeur de AX***

- Soustraction d'une valeur immédiate à un registre

SUB CX, 5 ***; Soustrait 5 de la valeur du registre CX***

3- **MUL (Multiplication) :** L'instruction MUL est utilisée pour effectuer une multiplication non signée.

MUL BX ***; Multiplie la valeur de AX par la valeur de BX (non signée)***

En règle générale, l'instruction MUL est utilisée pour effectuer des multiplications avec un seul opérande, par exemple, *« MUL BX »* pour multiplier la valeur dans l'accumulateur (AX) par la valeur contenue dans BX.

4- **IMUL (Multiplication avec signe) :** L'instruction IMUL est utilisée pour effectuer une multiplication signée.

IMUL CX, –2 ***; Multiplie la valeur du registre CX par –2 (signé)***

5- **DIV (Division non signée) :** L'instruction DIV est utilisée pour effectuer une division non signée.

DIV DX ***; Diviser la valeur contenue dans le registre DX par AX***

6- **IDIV (Division avec signe) :** L'instruction IDIV est utilisée pour effectuer une division signée.

IDIV BX ***; Divise la paire de registres DX:AX par la valeur de BX (signé)***

Il convient de noter que dans le cas de la division signée, DX:AX est utilisé pour stocker un nombre signé 32 bits, où DX contient les bits de poids élevé (y compris le bit de signe) et AX contient les bits de poids faible du nombre.

7- **INC (Incrémentation) et DEC (Décrémentation) :** Ces instructions sont utilisées pour augmenter ou diminuer la valeur d'un opérande de 1.

INC SI ***; Incrémente la valeur du registre SI de 1***

DEC CX ***; Décrémente la valeur du registre CX de 1***

8- **NEG (Négation) :** L'instruction NEG est utilisée pour inverser le signe d'un opérande.

NEG AX ***; Inverse le signe de la valeur dans le registre AX***

2.3. Instructions logiques

Voici quelques instructions logiques couramment utilisées en langage assembleur x86, accompagnées d'exemples :

1- AND (ET logique) : L'instruction AND effectue un "ET logique" bit à bit entre deux opérandes.

AND AX, BX ***; Effectue un "ET logique" entre les valeurs de AX et BX***

```
MOV AL,10010110B          AL= 1 0 0 1 0 1 1 0
MOV BL,11001101B    ➔     BL= 1 1 0 0 1 1 0 1
AND AL, BL                ------------------
                          AL= 1 0 0 0 0 1 0 0
```

Application : L'instruction AND est couramment utilisée pour forcer certains bits à zéro dans un mot, ce qu'on appelle le masquage de bits. Par exemple, dans le mot 1101 1101B, on peut vouloir mettre à zéro les bits 0, 2, 4 et 7 tout en conservant les autres inchangés.

```
7 6 5 4 3 2 1 0
1 1 0 1 1 1 0 1
0 1 1 0 1 0 1 0   ← Masque
---------------
0 1 0 0 1 0 0 0
```

2- OR (OU logique) : L'instruction OR effectue un "OU logique" bit à bit entre deux opérandes.

OR CX, DX ***; Effectue un "OU logique" entre les valeurs de CX et DX***

Application : mettre à 1 un ou plusieurs bits dans un mot. Par exemple, dans le mot 1000 1101B, on souhaite positionner à 1 les bits 1 et 5 tout en laissant les autres inchangés.

```
7 6 5 4 3 2 1 0
1 0 0 0 1 1 0 1
0 0 1 0 0 0 1 0   ← Masque
---------------
1 0 1 0 1 1 1 1
```

3- XOR (OU exclusif logique) : L'instruction XOR effectue un "OU exclusif logique" bit à bit entre deux opérandes.

XOR AL, FFH ***; Effectue un "OU exclusif logique" entre AL et FFH***

```
MOV AL, 25
XOR AL, AL    ➔    AL= 0
```

4- NOT (Négation logique, complément à 1) : L'instruction NOT effectue une négation logique (inversion des bits) sur un opérande.

NOT BX ***; Effectue une négation logique des bits dans BX***

```
MOV AL,10010110B
NOT AL                  ➔        AL= 0 1 1 0  1 0 0 1B
```

5- NEG (Complément à 2) : L'instruction NEG effectue la négation arithmétique de l'opérande en le traitant comme un nombre signé en complément à deux.

```
NEG BL          ; Effectue la négation de la valeur stocké dans BL

MOV AL, 30
MOV BL, 11      ➔        AL = 30 + (–11) = 19
NEG BL
ADD AL, BL
```

6- SHL/SHR (Décalage logique à gauche/droite) : Les instructions SHL (*Shift Left*) et SHR (*Shift Right*) effectuent des décalages logiques à gauche ou à droite des bits dans un registre.

```
SHL BX, 1       ; Effectue un décalage logique à gauche de 1 bit dans BX
SHR CX, 2       ; Effectue un décalage logique à droite de 2 bits dans CX

MOV BL,  11001101B
SHL BL, 1       ➔        BL = 10011010B
SHR BL, 2       ➔        BL = 00100110B
```

7- ROL/ROR (Rotation à gauche/droite) : Les instructions ROL (*Rotate Left*) et ROR (*Rotate Right*) effectuent des rotations circulaires des bits dans un registre.

- **ROL :** effectue une rotation circulaire de l'opérande vers la gauche de *n* positions. Les bits sortant par la gauche sont réinjectés par la droite.
  ROL AL, 1 ; Effectue une rotation circulaire à gauche d'1 bit dans AL
- **ROR :** effectue une rotation circulaire de l'opérande vers la droite de n positions. Les bits sortant par la droite sont réinjectés par la gauche.
  ROR DL, 3 ; Effectue une rotation circulaire à droite de 3 bits dans DL

```
MOV AL,  11001011B
MOV BL,  11001011B
MOV CL,  11001011B
MOV DL,  11001011B
ROL AL, 1       ➔        AL = 10010111B
ROR BL, 1       ➔        BL = 11100101B
ROR CL, 3       ➔        CL = 01111001B
ROL DL, 4       ➔        DL = 10111100B
```

Exemple d'application : Écrivez un programme en assembleur capable de réaliser deux opérations :

1- Additionnez deux nombres de 16 bits stockés aux adresses mémoire 2000H et 2002H, puis enregistrer le résultat à l'adresse mémoire 2004H.

2- Effectuez ensuite une opération logique ET (AND) entre le résultat précédent et une valeur de masque. Ce masque a pour fonction de mettre à zéro les quatre premiers chiffres du résultat. Sauvegardez enfin le résultat de cette opération à l'adresse mémoire 2024.

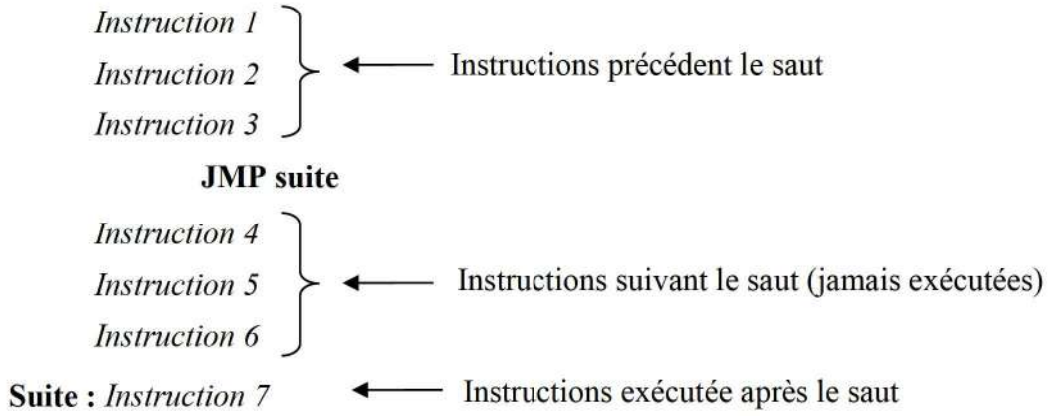
```
MOV AX, [2000H]   ; Charge le premier nombre depuis l'adresse mémoire 2000H dans le
                    registre AX
ADD AX, [2002H]   ; Ajoute le deuxième nombre situé à l'adresse mémoire 2002H au
                    contenu de AX
MOV [2004H], AX   ; Enregistre le résultat de l'addition à l'adresse mémoire 2004H
MOV BX, [2004H]   ; Charge le résultat depuis l'adresse mémoire 2004H dans le registre
                    BX
AND BX, FFF0H     ; Applique un masque pour mettre à zéro les quatre chiffres LSB du
                    résultat
MOV [2024H], BX   ; Enregistre le résultat modifié à l'adresse mémoire 2023h
```

3. Les instructions de branchement

Les instructions de branchement, également appelées instructions de saut, permettent de modifier l'ordre d'exécution des instructions d'un programme en fonction de certaines conditions. On distingue trois types principaux d'instructions de branchement :

3.1. Instruction de saut inconditionnel *JMP label*

Cette instruction permet de réaliser un saut (*jump*) vers un label spécifié. Un label (ou étiquette) est une référence symbolique représentant l'emplacement d'une instruction en mémoire. L'instruction JMP est utilisée pour effectuer des sauts inconditionnels dans un programme en langage assembleur. Elle permet ainsi, de transférer le contrôle du programme d'une partie du code à une autre vers une autre adresse (ou instruction) dans le programme.



Exemple : Le code assembleur suivant représente une boucle infinie qui incrémente le registre AX de 1 et décrémente le registre BX de 1 à chaque itération, puis ajoute la nouvelle valeur d'AX à BX. En l'absence d'une condition de sortie, cette boucle continuera indéfiniment à exécuter ces opérations, créant ainsi une boucle infinie.

Boucle : INC AX
DEC BX ➔ boucle infinie
ADD BX,AX
JMP **boucle**

3.2. Instructions de sauts conditionnels *Jcondition label*

Ce sont des instructions qui ne s'exécutent que si une condition particulière est remplie. Dans le cas contraire, l'exécution du programme continue normalement avec l'instruction suivante. Les instructions de sauts conditionnels permettent de prendre des décisions dans un programme en fonction de l'état des indicateurs d'état. Si une condition spécifique est vérifiée, le saut est effectué vers l'étiquette (label) spécifiée, sinon le programme continue normalement. Cela permet d'implémenter des structures de contrôle telles que les boucles et les structures de décision dans un programme en langage assembleur.

La condition de saut est basée sur l'état des indicateurs d'état (flags), qui sont des drapeaux ou des indicateurs internes dans le microprocesseur qui reflètent le résultat de l'exécution des instructions précédentes. Ces indicateurs d'état incluent généralement des informations telles que les drapeaux de zéro (ZF) pour indiquer si le résultat était zéro, les drapeaux de signe (SF) pour indiquer si le résultat était négatif, etc.

Instruction	Nom	Condition
JZ label	Jump if Zero	Saut si la valeur est zéro.
JNZ label	Jump if Not Zero	Saut si la valeur n'est pas zéro.
JE label	Jump if Equal	Saut si les valeurs sont égales.
JNE label	Jump if Not Equal	Saut si les valeurs ne sont pas égales.
JC label	Jump if Carry	Saut si une retenue est générée.
JNC label	Jump if Not Carry	Saut si aucune retenue n'est générée.
JS label	Jump if Sign	Saut si la valeur est négative.
JNS label	Jump if Not Sign	Saut si la valeur est positive.
JO label	Jump if Overflow	Saut si un dépassement de capacité se produit.
JNO label	Jump if Not Overflow	Saut si aucun dépassement de capacité ne se produit.
JP label	Jump if Parity	Saut si la parité est paire.
JNP label	Jump if Not Parity	Saut si la parité est impaire.

Il existe également un autre type de saut appelé **saut conditionnel arithmétique**. Ces sauts sont généralement utilisés après une instruction de comparaison, telle que :

CMP opérande1, opérande2

Condition	Nombres signés	Nombres non signés
=	JEQ label	JEQ label
>	JG label	JA label
≥	JGE label	JAE label
<	JL label	JB label
≤	JL label	JBE label
≠	JNE label	JNE label

Structure :

```
            CMP AX,BX
            JG superieur
            JL inferieur
superieur : ……..
            ...……...
            ……..
inferieur : ……..
            ...……...
```

Exemple n° 1 : On désire additionner deux nombres signés N1 et N2, situés aux adresses mémoires 1100H et 1101H respectivement. Le résultat sera stocké à l'adresse 1102H si le résultat est positif, à l'adresse 1103H s'il est négatif, et à l'adresse 1104H si le résultat est nul.

```
; Charger N1 et N2 depuis la mémoire
MOV   AL, [1100H]          ; Charger N1 depuis l'offset 1100H
MOV   BL, [1101H]          ; Charger N2 depuis l'offset 1101H

; Additionner les nombres
ADD   AL, BL

; Vérifier si le résultat est nul ou négatif
JZ nul                     ; Sauter à nul si le résultat est nul
JS negatif                 ; Sauter à negatif si le résultat est négatif

; Instructions a exécutées dans le cas où les conditions ne sont pas vérifié
MOV [1102H], AL            ; Stocker le résultat à l'offset 1102H
JMP fin                    ; Terminer le programme

; Instructions a exécutées dans le cas où le résultat négatif
negatif:  MOV [1103H], AL  ; Stocker le résultat à l'offset 1103H
          JMP fin          ; Terminer le programme

; Instructions a exécutées dans le cas où le résultat est nul
nul:      MOV [1104H], AL  ; Stocker le résultat à l'offset 1104H
          JMP fin          ; Terminer le programme
fin:
```

Exemple n° 2 : Écrivez un programme en assembleur x86 qui compare deux nombres (nombre1 et nombre2) stockés en mémoire et effectue une opération différente en fonction du résultat de la comparaison. Si nombre1 est supérieur ou égal à nombre2, ajoutez les deux nombres et stockez le résultat dans la variable *resultat*. Sinon, soustrayez nombrer2 de nombre1 et stockez le résultat dans *resultat*.

```
MOV AL, [nombre1]                    ; Charger le premier nombre dans AL
MOV BL, [nombre2]                    ; Charger le deuxième nombre dans BL

CMP AL, BL                           ; Comparer les deux nombres

    ; Effectuer un saut conditionnel en fonction du résultat de la comparaison
JGE superieur_ou_egal                ; Saut si le premier nombre est supérieur ou égal

    ; Si la condition n'est pas vérifiée alors l'opération de soustraction est effectuée
SUB AL, BL                           ; Effectuer l'opération de soustraction
JMP fin                              ; Saut vers la sortie

superieur_ou_egal:                   ; Dans le cas où le premier nombre est supérieur ou égal
ADD AL, BL                           ; Effectuer l'opération d'addition

fin:
; Stocker le résultat dans la variable 'resultat'
MOV [resultat], AL
```

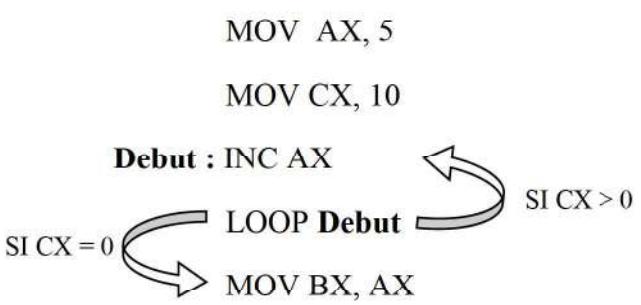
3.3. La boucle "*LOOP*"

En assembleur x86, la boucle "LOOP" est une instruction de boucle conditionnelle. Elle est utilisée pour exécuter une série d'instructions un nombre spécifique de fois. La boucle "LOOP" utilise le registre CX comme compteur de boucle. En effet, cette instruction décrémente automatiquement le registre de compteur de boucle (CX) et effectue le saut, vers l'étiquette spécifiée, tant que le compteur n'est pas égal à zéro.

Syntaxe de la boucle "LOOP" :

LOOP label

Exemple d'utilisation de "LOOP" :

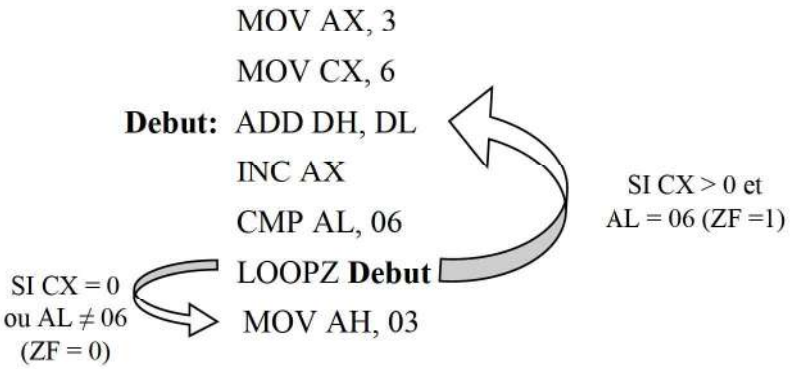


Ce programme débute en initialisant les registres AX et CX avec les valeurs 5 et 10 respectivement. En utilisant une boucle LOOP, il incrémente la valeur du registre AX jusqu'à ce que le registre CX atteigne zéro. Une fois la boucle terminée, la valeur finale d'AX est stockée dans le registre BX.

3.3.1. Les boucle "LOOPZ" et "LOOPNZ"

Les boucles "LOOPZ" et "LOOPNZ" sont des instructions d'itération en assembleur x86. LOOPZ effectue un saut vers l'étiquette spécifiée si le registre CX n'est pas nul et que le bit ZF (Zero Flag) est à 1 (le résultat de l'opération est égal à zéro). En revanche, LOOPNZ effectue un saut si le registre CX n'est pas nul et que le bit ZF est remis à zéro.

Exemple :



Il est important de noter que les boucles LOOP en assembleur sont souvent utilisées pour :

- ✓ **Parcourir des tableaux ou des chaînes** et appliquer des opérations sur chaque élément.
- ✓ **Répéter un bloc d'instructions** un nombre spécifique de fois, contrôlé par un compteur.

3.4. Appel de sous-routine

Les sous-routines, également appelés sous-programme, procédures ou fonction, sont des éléments essentiels en programmation pour éviter la répétition de séquences d'instructions identiques à plusieurs endroits dans un programme. Au lieu de rédiger la même séquence maintes fois, on la définit une seule fois en lui attribuant un nom au choix. Lorsque cette séquence est nécessaire, elle peut être appelée depuis d'autres parties du programme. Le programme principal qui effectue ces appels est le point de départ de l'exécution.

Un sous-programme est une séquence de code identifiée par un nom, souvent précédée de la déclaration "PROC" (abréviation de procédure) et suivie de "ENDP" pour marquer la fin du sous-programme. À l'intérieur du sous-programme, on retrouve les instructions spécifiques à accomplir, puis une instruction de "retour" (habituellement "RET") pour indiquer la fin du sous-programme et le retour à l'emplacement d'appel dans le programme principal.

```
nom_sp      PROC
            Instruction 1
            Instruction 2      ←— Instructions du sous-programme
            Instruction 3
              RET              ←— Instructions de retour au programme principal
nom_sp      ENDP
```

L'appel d'un sous-programme par le programme principal s'effectue en utilisant l'instruction "CALL nom_sp"

```
Inst. 1
Inst. 2      ←— Instructions précédent l'appel au sous-programme
Inst. 3
CALL nom_sp  ←— Appel au sous-programme
Inst. 1
Inst. 2      ←— Instructions exécutées après le retour au programme principal
Inst. 3
```

Lorsqu'une instruction CALL est exécutée, l'adresse de la première instruction du sous-programme est chargée dans le registre IP afin de transférer l'exécution vers ce sous-programme. À la fin du sous-programme, le microprocesseur doit reprendre l'exécution du programme principal à l'instruction qui suit le CALL. Pour cela, l'adresse de retour est restaurée dans le registre IP pour que l'exécution puisse continuer normalement.

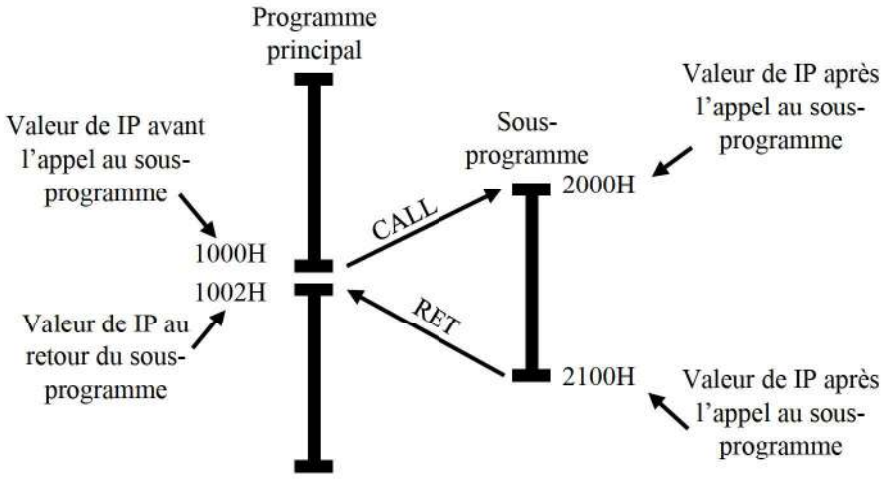


Fig. 4.1 : Exécution d'un sous-programme.

En résumé, les sous-programmes sont utilisés pour réduire la duplication de code en le regroupant sous un nom identifiable, ce qui simplifie la maintenance du code et améliore la lisibilité. Ils sont appelés depuis le programme principal chaque fois que leur fonctionnalité est requise.

4. Notion de pile

Lorsqu'un sous-programme est appelé, l'adresse de l'instruction suivante du programme principal est automatiquement sauvegardée dans la pile. Cela permet au sous-programme de s'exécuter, puis de revenir ensuite au programme principal en récupérant cette adresse stockée dans la pile afin de poursuivre l'exécution normale du programme.

La pile est une zone mémoire utilisée pour stocker temporairement des informations telles que des données ou des adresses mémoire pendant l'exécution d'un programme. Elle fonctionne selon le principe *Last-In-First-Out* (*LIFO*), ce qui signifie que la dernière information déposée dans la pile est la première à être récupérée. Ce fonctionnement est particulièrement important lors des appels de sous-programmes, car il permet de conserver les adresses de retour et certaines données temporaires sans les perdre. Deux opérations principales peuvent être réalisées sur la pile :

- **Empiler une donnée :** consiste à placer une donnée au sommet de la pile. Cette opération se fait à l'aide de l'instruction PUSH (empiler).

```
PUSH AX          ; Empile la valeur du registre AX sur la pile
```

- **Dépiler une donnée :** consiste à récupérer la donnée située au sommet de la pile. Cette opération se fait à l'aide de l'instruction POP (dépiler)

```
POP BX           ; Dépile la valeur du dessus de la pile dans le registre BX
```

Le pointeur de pile (le registre SP) est utilisé pour suivre la position actuelle de la pile en mémoire. À chaque opération de push ou de pop, le SP est automatiquement mis à jour pour pointer vers la nouvelle position de la pile. La pile est essentielle pour garantir que les données et l'état du processeur sont correctement gérés lors de l'exécution de programmes complexes avec de nombreuses sous-routines. Elle permet également de conserver des variables locales et temporaires, ce qui est particulièrement utile dans la programmation.

Exemple : Supposons que vous ayez une sous-routine appelée "*calculer()*" dans votre programme principal. Avant d'appeler "*calculer()*", vous poussez les valeurs de certains registres sur la pile pour les sauvegarder, comme ceci :

```
push BX          ; Enregistre le contenu du registre BX dans la pile
push AX          ; Enregistre le contenu du registre AX dans la pile
```

Ensuite, lorsque la sous-routine "*calculer()*" se termine, vous pouvez restaurer les valeurs des registres en les retirant de la pile, comme ceci :

```
pop BX           ; Restaure la valeur du registre BX
pop AX           ; Restaure la valeur du registre AX
```

Cela garantit que les valeurs des registres sont les mêmes avant et après l'appel à "*calculer()*".

5. Les modes d'adressage

En langage assembleur, les modes d'adressage définissent la manière dont les opérandes, qu'ils soient des valeurs de données ou des adresses mémoire, sont spécifiés et accédés dans le cadre de l'exécution d'un programme. Ils jouent un rôle essentiel dans la programmation assembleur en déterminant comment le microprocesseur interagit avec la mémoire et les registres pour exécuter les instructions.

Pour le microprocesseur 8086, les modes d'adressage peuvent être regroupés en deux catégories principales : les modes d'adressage simples et les modes d'adressage évolués.

5.1. Modes d'adressage simples

a) Mode d'adressage immédiat

Dans ce cas, la donnée est fournie directement dans l'instruction. Le CPU transfère une valeur numérique (1 ou 2 octets) dans le registre de destination.

MOV AL, 42H ***; Charge le registre AL avec la valeur immédiate 42H***

Il a précisé que dans le cas de l'adressage immédiat de la mémoire, il est préférable de spécifier le format de la donnée, que ce soit en octet (8 bits) ou en mot (16 bits), car le microprocesseur 8086 peut traiter des données dans ces deux formats. Pour ce faire, l'utilisation d'un spécificateur de format est requise :

MOV byte ptr [1100H],65H ***;écrit la valeur 65H (1 octet) dans la case mémoire d'offset 1100H***

MOV word ptr [1100H],65H ***;écrit la valeur 0065H (2 octets) dans les cases mémoire d'offset 1100H et 1101H***

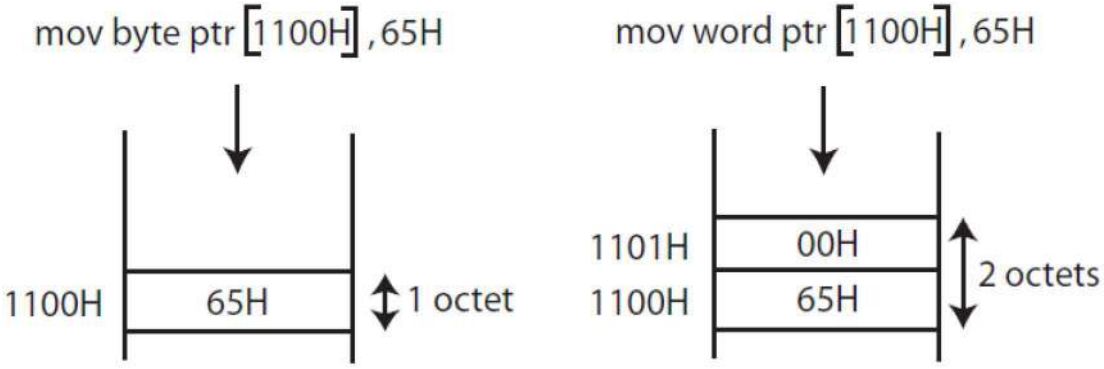


Fig. 4.2 : L'utilisation d'un spécificateur de format lors de l'adressage immédiat.

b) Mode d'adressage registre

Dans ce cas, le transfert s'effectue entre deux registres. Le processeur copie une donnée, qu'il s'agisse d'un octet (8 bits) ou d'un mot (16 bits), d'un registre source vers un registre destination, à condition que les deux registres soient de même taille.

MOV AX, BX ***; copie le contenu du registre BX dans le registre AX***

c) Mode d'adressage direct

Dans ce cas, l'instruction contient directement l'adresse de la case mémoire où se trouve la donnée (un octet ou un mot de 2 octets) qui doit être chargée dans un registre déterminé.

MOV AL, [1400H] ***;charge dans AL la valeur (8 bits) contenue à l'adresse mémoire d'offset 1400H***

MOV AX, [1400] ***;charge dans AX la valeur (16 bits) contenue à l'adresse mémoire d'offset 1400H***

Dans le cas de l'adressage direct, l'adresse effective correspond à l'offset de la case mémoire par rapport au segment de données DS utilisé par défaut. Autrement dit, l'instruction considère que l'adresse mémoire indiquée appartient au segment de données, dont l'adresse de base est contenue dans le registre DS.

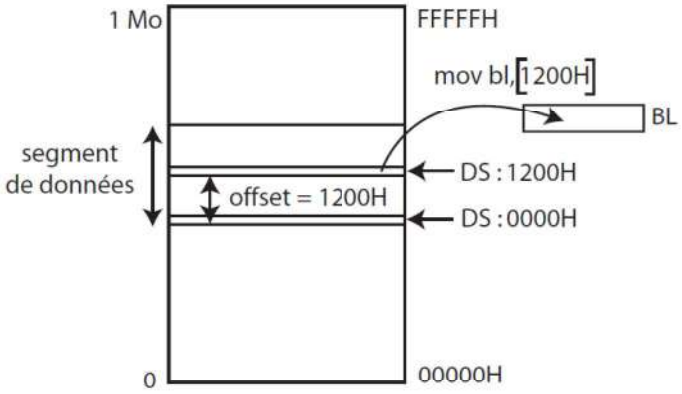


Fig. 4.3 : Exemple d'adressage directe par défaut.

Cependant, il est possible de modifier le segment utilisé lors d'un adressage direct en ajoutant un **préfixe de segment** à l'instruction. On parle alors de **forçage de segment**, car on indique explicitement au processeur quel segment mémoire doit être utilisé, au lieu d'utiliser le segment par défaut qui est généralement le segment de données DS.

Exemple : MOV BL,ES :[1200H]

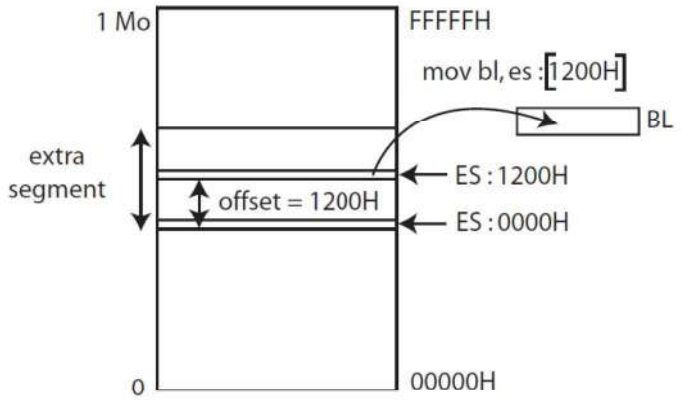


Fig. 4.4 : Exemple d'adressage directe avec forçage de segment.

5.2. Modes d'adressage évolués

a) *Mode d'adressage* basé

Dans ce mode d'adressage, au lieu d'indiquer directement une adresse mémoire, on utilise un registre de base qui contient l'offset de la donnée en mémoire. Les registres de base utilisés sont généralement BX et BP.

Exemples :

```
MOV AL,[BX]     ; Transfère dans AL la donnée située à l'adresse dont l'offset
                est  ; contenu dans BX. Le segment associé par défaut à BX est
                le   ; segment de données DS.

MOV AL,[BP]     ; Le segment par défaut associé au registre BP est le segment
                de   ; pile SS. Dans ce cas, l'adressage est basé sur SS.
```

b) *Mode d'adressage* indexé

Ce mode d'adressage est similaire à l'adressage basé, mais l'offset est contenu dans un registre d'index tel que SI ou DI. Ces registres sont associés par défaut au segment de données DS.

Exemples :

```
MOV AL,[SI]     ; charge le registre AL avec la donnée située à l'adresse dont l'offset
                ;est contenu dans SI.

MOV [DI],BX     ; Stocke le contenu du registre BX dans les cases mémoire
                d'offset  ; DI et DI+1.
```

Remarque : Il est possible d'ajouter une valeur constante au registre de base ou d'index afin de calculer l'offset.

Exemple : MOV [SI+100H],AX

Cette instruction peut également s'écrire :

MOV [SI][100H],AX

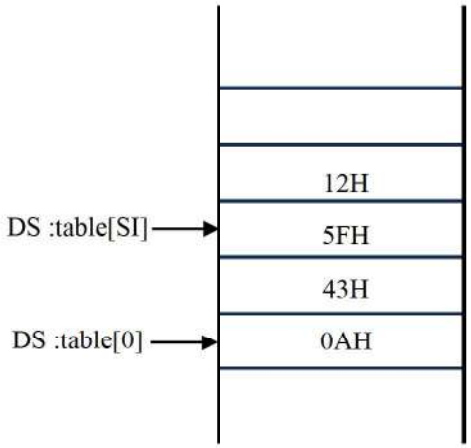
ou MOV 100H[SI],AX

Les modes d'adressage basé et indexé sont très utilisés pour la manipulation de tableaux stockés en mémoire.

Exemple :

```
MOV SI,0
MOV word ptr table[SI],1234H
MOV SI,2
MOV word ptr table[SI],5678H
```

Dans cet exemple, table représente l'adresse (offset) du premier élément du tableau, tandis que le registre SI est utilisé comme indice du tableau pour accéder aux différents éléments.



Dans cet exemple "*table*" est un symbole utilisé comme un label pour représenter un emplacement mémoire. Il n'est pas directement reconnu par le langage assembleur comme une constante ou une variable. Au lieu de cela, il est utilisé pour marquer un endroit spécifique en mémoire où les valeurs sont stockées.

c) Adressage basé indexé

Il s'agit d'une combinaison des modes d'adressage basé et indexé, à laquelle on ajoute un déplacement (valeur constante), ce qui offre une grande souplesse pour accéder aux données en mémoire. Dans ce mode, l'offset est calculé comme la somme d'un registre de base, d'un registre d'index et d'une valeur constante.

Exemple :

MOV AH,[BX+SI+100H]

Ce mode d'adressage est particulièrement utile pour manipuler des structures de données complexes telles que les matrices ou les enregistrements.

Exemple :

```
MOV BX,10
MOV SI,15
MOV byte ptr matrice[BX][SI],12H
```

Dans cet exemple, les registres BX et SI sont utilisés respectivement comme indice de ligne et indice de colonne dans la matrice.

6. Les interruptions

Un système à microprocesseur peut être connecté à différents périphériques tels qu'un écran, une souris, une imprimante, un disque dur, un convertisseur numérique-analogique (CNA), un convertisseur analogique-numérique (CAN), etc. Pour communiquer avec ces périphériques, le microprocesseur utilise principalement deux méthodes :

- **Scrutation périodique (ou *polling*) :** le programme principal vérifie régulièrement l'état des ports d'E/S en exécutant des instructions dédiées. Cette méthode est simple à programmer mais présente certains inconvénients :
  - Perte de temps lorsque plusieurs périphériques doivent être interrogés.
  - Les nouvelles données ne sont pas garanties à chaque cycle.
  - Risque de perte de données si les informations changent rapidement.
- **Interruption :** lorsqu'un périphérique a de nouvelles données disponibles, il en informe le microprocesseur via une demande d'interruption (*IRQ : Interrupt Request*). Cette méthode est plus efficace car le microprocesseur ne lit les ports d'E/S que lorsque des données sont prêtes, optimisant ainsi l'utilisation du temps et réduisant le risque de perte d'informations.

Des exemples typiques d'utilisation des interruptions incluent :

- Le clavier, qui déclenche une interruption lorsqu'une touche est pressée.
- Le port série, qui génère une interruption à l'arrivée d'un caractère.

Une interruption est donc un signal ou un événement inattendu qui interrompt temporairement le déroulement normal d'un programme pour permettre au microprocesseur de gérer rapidement un événement particulier. Une fois la routine d'interruption exécutée, le microprocesseur reprend l'exécution normale du programme, assurant ainsi une réponse rapide aux périphériques externes sans perturber le fonctionnement global du système.

6.1. Prise en charge d'une interruption par le microprocesseur

Lorsqu'un périphérique demande une interruption :

1) Le microprocesseur suspend l'exécution de l'instruction en cours.
2) Il sauvegarde le contenu des principaux registres sur la pile, comme le pointeur d'instruction et les indicateurs (flags).
3) Il envoie un accusé de réception au périphérique pour signaler que la demande d'interruption est acceptée.

Remarque : Le microprocesseur peut ignorer certaines interruptions en les masquant via un indicateur dans le registre d'état. Cependant, certaines interruptions, dites non masquables, sont toujours prises en compte et traitées immédiatement.

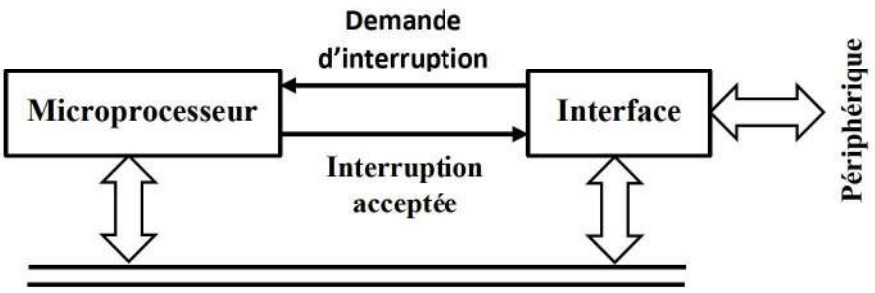


Fig. 4.5 : Prise en charge d'une interruption par le microprocesseur

4) Le microprocesseur interrompt l'exécution du programme en cours et se dirige vers l'exécution d'un sous-programme de service d'interruption *(ISR : Interrupt Service Routine*).
5) Une fois l'ISR terminée, les registres sont restaurés à partir de la pile, et le microprocesseur reprend l'exécution du programme initial.

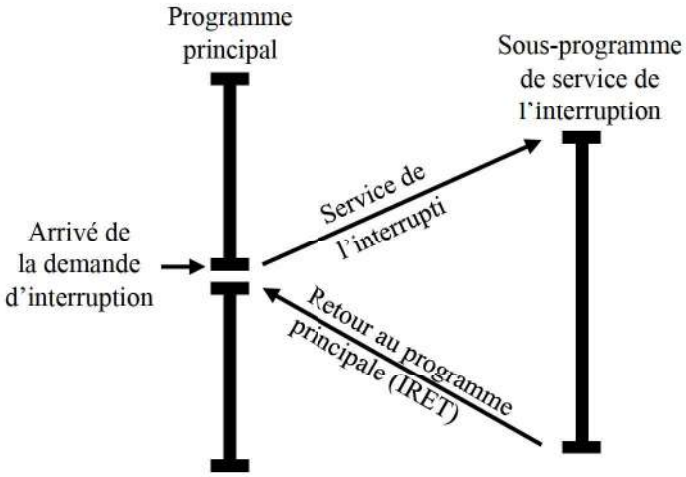


Fig. 4.6 : Excution d'une inturuption par le microcontroleur.

Remarque : La dernière instruction d'un ISR doit être IRET (*Interrupt Return*), qui permet le retour correct du flux d'exécution. Si plusieurs interruptions se produisent simultanément, elles doivent être hiérarchisées afin que le microprocesseur sache dans quel ordre les traiter.

6.2. Adresse des sous-programmes d'interruptions

Lorsqu'une interruption survient, le périphérique envoie un code numérique au microprocesseur. Ce code est utilisé pour consulter une table des vecteurs d'interruption stockée en mémoire centrale. Chaque vecteur contient l'adresse d'un sous-programme spécifique, dédié à gérer le type d'événement correspondant. Ainsi, le microprocesseur sait Exactement quel ISR exécuter pour chaque interruption ;

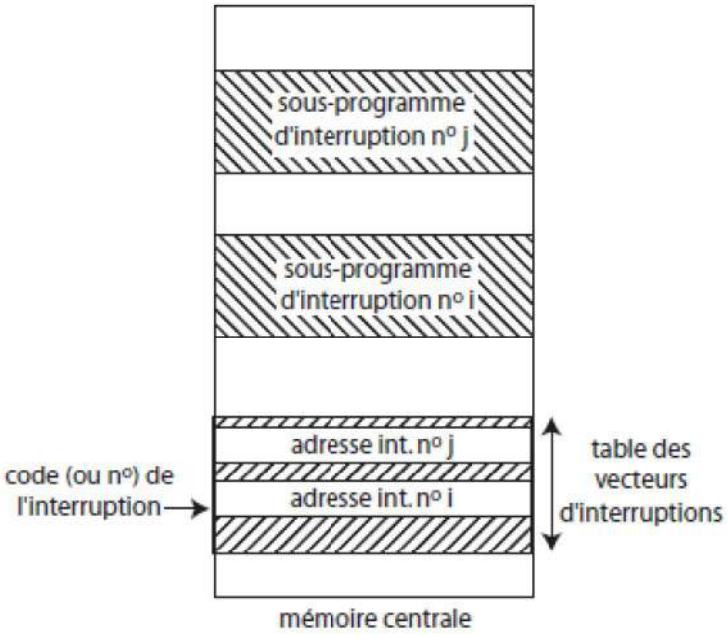


Fig. 4.7 : Adressage des sous-programmes d'interruptions en mémoire.

Lorsque les adresses des sous-programmes d'interruption sont organisées de cette façon, on parle d'interruptions vectorisées. Cette méthode présente l'avantage de pouvoir placer une ISR (*Interrupt Service Routine*) n'importe où en mémoire. Il suffit simplement d'indiquer le vecteur d'interruption correspondant pour accéder à l'adresse exacte du sous-programme à exécuter.

6.3. Les interruptions du 8086

Le microprocesseur 8086 peut gérer jusqu'à 256 interruptions, chaque interruption étant identifiée par un numéro de type allant de 0 à 255. Ces interruptions sont classées en deux catégories principales :

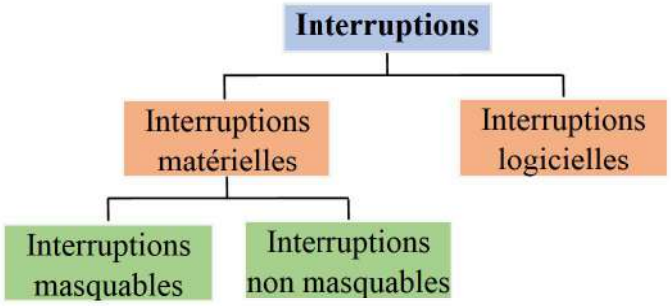


Fig. 4.8 : Type d'interruptions.

6.3.1. Les interruptions matérielles

Ces interruptions sont déclenchées par l'activation des lignes INTR (*Interrupt Request*) et NMI (*Non-Maskable Interrupt*) du microprocesseur, indiquant des événements spécifiques nécessitant une réponse immédiate. Les interruptions matérielles peuvent être classées en deux catégories : masquables et non masquables.

1. **INTR (*Interrupt Request*) :** La ligne INTR est associée aux interruptions masquables, générées par des périphériques externes. Lorsqu'un périphérique requiert l'attention du microprocesseur, il active la ligne INTR, signalant ainsi une demande d'interruption. Les interruptions masquables peuvent être temporairement négligées en fonction de conditions préalablement définies, offrant au microprocesseur une certaine flexibilité dans la gestion des événements externes.
2. **NMI (*Non-Maskable Interrupt*) :** La ligne NMI est liée aux interruptions non masquables, déclenchées par des événements critiques. Contrairement à INTR, NMI a une priorité plus élevée et est réservée pour des situations urgentes, telles que des erreurs matérielles graves. Les interruptions non masquables exigent une réponse immédiate et ne peuvent pas être temporairement ignorées, garantissant ainsi une réactivité du microprocesseur même dans des circonstances cruciales.

Remarque : La ligne RESET sur un microprocesseur, bien que n'étant pas formellement classée comme une ligne d'interruption, peut être considérée comme ayant des similitudes avec le mécanisme d'interruption. Son rôle essentiel est d'effectuer une réinitialisation complète du microprocesseur, rétablissant ainsi tous ses composants à un état initial prédéfini.

6.3.2. Les interruptions logicielles

Ces interruptions sont générées par l'instruction INT n, où *n* représente le type spécifique d'interruption. En outre, chaque numéro d'interruption est lié à un sous-programme spécifique, préalablement écrites et stockées en mémoire. Lorsqu'une interruption survient, le microprocesseur identifie la routine correspondante grâce au numéro associé. La routine d'interruption s'exécute pour traiter l'événement, garantissant un traitement approprié de la situation. À la fin de chaque routine, l'instruction IRET est utilisée pour assurer un retour correct au programme qui a été interrompu, permettant ainsi une reprise fluide de l'exécution normale.

Les interruptions du 8086 utilisent un système vectorisé. La table des vecteurs d'interruptions est obligatoirement placée au début de la mémoire centrale, à l'adresse physique 00000H. Chaque vecteur occupe 4 octets et contient l'adresse logique d'un sous-programme, exprimée sous la forme CS:IP (Code Segment : Instruction Pointer).

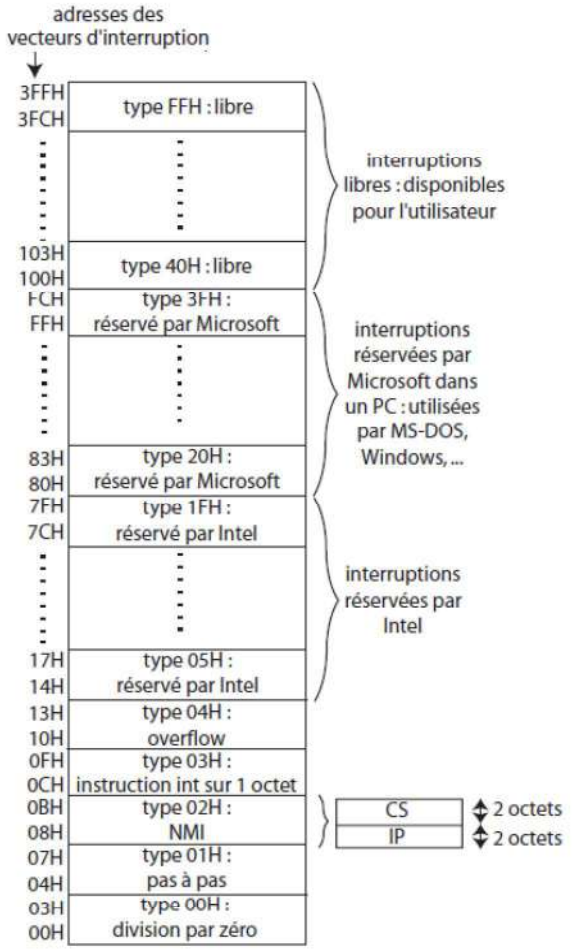


Fig. 4.9 : Table des vecteurs d'interruptions du 8086.

6.4. Le contrôleur programmable d'interruptions 8259

Le microprocesseur 8086 est équipé de seulement deux lignes de demande d'interruptions matérielles, à savoir NMI et INTR. Pour permettre la connexion de plusieurs périphériques utilisant des interruptions et simplifier la gestion de ces demandes, on peut intégrer un contrôleur programmable d'interruptions 8259.

6.4.1. Le rôle du 8259 est multiple

1- **Réception des demandes d'interruptions :** Le 8259 est conçu pour recevoir les demandes d'interruptions provenant des divers périphériques connectés au système.

2- **Résolution des priorités des interruptions :** Il résout les priorités entre les différentes demandes d'interruptions, garantissant que les événements les plus critiques soient traités en premier.

3- **Génération du signal INTR pour le 8086 :** Le 8259 génère le signal INTR (*Interrupt Request)* qui est envoyé au microprocesseur 8086 pour l'informer de la présence d'une demande d'interruption.

4- **Émission du numéro d'interruption sur le bus de données :** Lorsqu'une demande d'interruption est acceptée, le 8259 émet le numéro de l'interruption correspondante sur le bus de données, fournissant ainsi au microprocesseur l'information nécessaire pour identifier la source de l'interruption.

Le 8259 est capable de gérer jusqu'à 8 demandes d'interruptions matérielles, ce qui permet de connecter efficacement plusieurs périphériques au système sans surcharger les deux lignes d'interruption du microprocesseur 8086.

6.4.2. Schéma fonctionnel du 8259

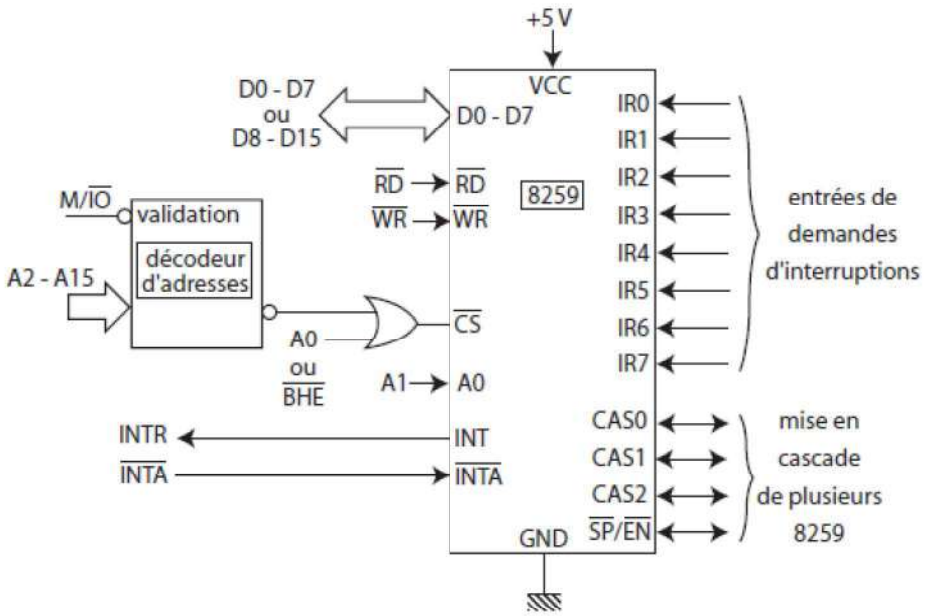


Fig. 4.10 : Schéma fonctionnelle du contrôleur d'interruptions 8259.

Si le nombre de demandes d'interruptions excède la capacité maximale qu'un seul contrôleur d'interruptions 8259 peut gérer, une solution consiste à les placer en cascade. Ainsi, le microprocesseur 8086 a la possibilité d'utiliser jusqu'à 8 contrôleurs d'interruptions 8259 en cascade. Chacun de ces 8259 peut gérer jusqu'à 8 interruptions, permettant ainsi d'élargir la capacité totale à 64 interruptions distinctes.

La gestion des interruptions en cascade implique plusieurs étapes :

1. Les périphériques générant des interruptions envoient leurs demandes au 8259 maître.
2. Le 8259 maître attribue des priorités aux demandes, traitant en premier lieu celles jugées les plus urgentes.
3. En cas de demande d'interruption émanant d'un esclave 8259, le 8259 maître coordonne cette demande avec le reste du système.
4. Le 8259 maître génère le signal INTR pour le microprocesseur, signalant la présence d'une demande d'interruption.
5. En cas d'acceptation de l'interruption, le 8259 maître émet le numéro correspondant sur le bus de données, permettant au microprocesseur d'identifier la source de l'interruption.

Les mémoires et le décodage d'adresse dans les microprocesseurs

1. Qu'est-ce qu'une mémoire ?

Une mémoire est un composant électronique ou un dispositif qui permet de stocker, d'organiser et de récupérer des données ou des informations. Elle peut prendre différentes formes, y compris des composants électroniques intégrés, des puces mémoires, des disques durs, des cartes mémoire, etc. Les mémoires sont utilisées pour conserver de manière temporaire ou permanente des données numériques, telles que des instructions de programme, des données utilisateur, des fichiers multimédias, etc.

Dans le contexte des microprocesseurs et des systèmes informatiques, la mémoire se réfère spécifiquement aux composants utilisés pour stocker et gérer les données et les instructions nécessaires au fonctionnement du microprocesseur et à l'exécution des programmes. La mémoire dans un système informatique peut être classée en deux catégories : mémoire interne et mémoire externe :

- **La mémoire centrale,** également connue sous le nom de mémoire interne, est utilisée pour stocker temporairement des données pendant l'exécution des programmes. Elle se trouve à l'intérieur du processeur, sous forme de registres, et offre une vitesse d'accès ultra-rapide pour les opérations en temps réel.
- **La mémoire externe,** également appelée mémoire physique ou mémoire secondaire, est située en dehors du processeur. Elle sert au stockage de données à long terme, même lorsque l'ordinateur est éteint. Bien qu'elle puisse stocker une grande quantité de données et d'instructions, son accès est légèrement plus lent en raison de sa localisation externe.

2. Caractéristiques d'une mémoire

Les principales caractéristiques d'une mémoire sont :

- **Adresse :** valeur numérique qui permet d'identifier un emplacement précis dans la mémoire.
- **Capacité ou taille :** quantité totale de données (en octets) que la mémoire peut contenir.
- **Temps d'accès :** durée nécessaire entre la demande d'une opération (lecture ou écriture) et sa réalisation effective.
- **Temps de cycle :** intervalle minimal entre deux accès consécutifs à la mémoire, généralement supérieur au temps d'accès.
- **Débit :** quantité d'informations pouvant être lue ou écrite par seconde.
- **Volatilité :** caractéristique indiquant si les données sont conservées en cas de coupure dc courant. Unc mémoirc volatilc pcrd son contcnu lorsquc l'alimcntation cst interrompue.
- **Durée de vie :** elle est mesurée en cycles d'écriture/effacement pour les mémoires réinscriptibles. Une longue durée de vie est essentielle pour la fiabilité à long terme.

3. Classification des mémoires

Les mémoires informatiques se divisent en deux catégories principales : les mémoires mortes et les mémoires vives.

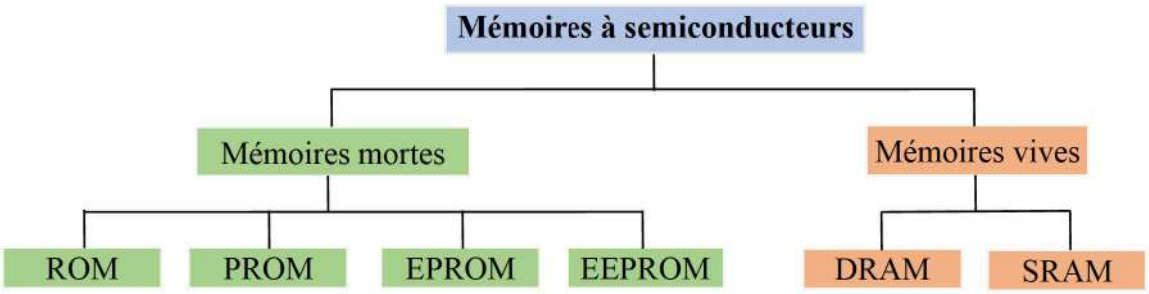


Fig. 5.1 : Classification des mémoires.

3.1. Les mémoires mortes

Les mémoires mortes sont des composants de stockage non volatiles qui renferment des informations permanentes, préprogrammées en usine ou par l'utilisateur. Il existe principalement quatre variantes de mémoires mortes : la ROM, la PROM, l'EPROM, l'EEPROM et la Flash EPROM, qui diffèrent par leur programmabilité et leur méthode d'effacement.

a) ROM (*Read Only Memory*)

La ROM est une mémoire permanente à lecture seule, préprogrammée en usine pour contenir des données immuables, comme le BIOS, et stockée de manière inaltérable grâce à des composants électroniques. Elle est largement utilisée dans les ordinateurs, les consoles de jeu, les smartphones et d'autres appareils électroniques pour conserver des données essentielles, ne devant pas être modifiées par l'utilisateur.

b) PROM (*Programmable ROM*)

La PROM est une ROM qui peut être programmée à l'aide d'un dispositif de graveur par l'utilisateur. Contrairement à la ROM préprogrammée en usine, la PROM peut être personnalisée une seule fois par l'utilisateur, après quoi son contenu reste fixe et inaltérable.

➢ **Principe de fonctionnement d'une ROM et d'une PROM**

Le fonctionnement d'une mémoire ROM est assez simple. Elle contient une matrice de diodes. Lorsqu'une adresse spécifique est fournie pour accéder à un mot, cette adresse active un décodeur correspondant. La sortie de la ROM fournit alors un mot binaire, constitué de bits à 1 ou à 0.

- Les bits à 1 sont générés grâce à des résistances électriques reliées à la tension d'alimentation du circuit.
- Les bits à 0 sont obtenus là où des diodes forcent la ligne correspondante à une tension représentant le niveau logique 0.

En somme, le rôle des diodes dans la ROM est de déterminer la sortie binaire en fonction de l'adresse d'accès, en établissant des niveaux logiques 1 et 0 par des liaisons électriques.

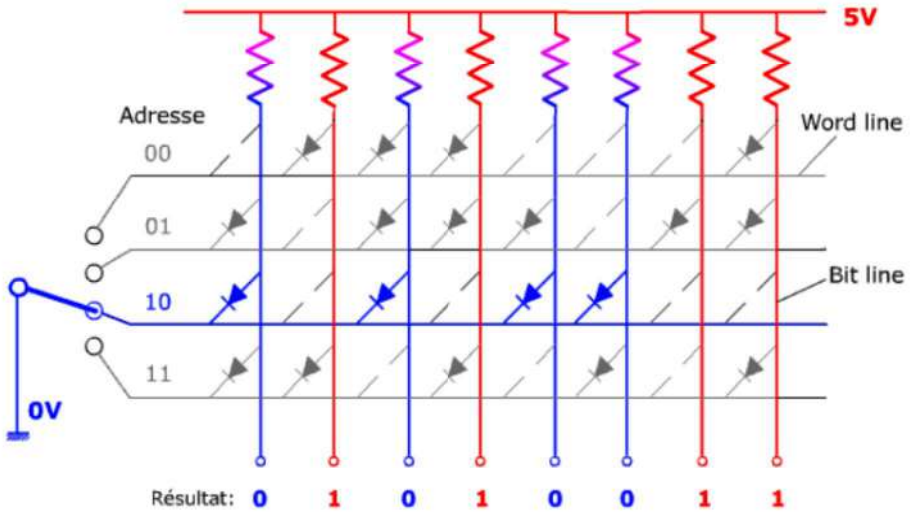


Fig. 5.2 : Architecture interne d'une ROM de 4 octets.

Le fonctionnement d'une PROM est très proche de celui d'une ROM classique. Dans une PROM neuve, les diodes sont associées à de petits fusibles. La programmation consiste à « griller » les fusibles correspondant aux bits que l'on souhaite définir à 1. Cette opération est définitive et irréversible.

c) EPROM (*Erasable PROM*)

L'EPROM est une forme de mémoire ROM qui peut être réécrite après un processus d'effacement par exposition aux rayons ultraviolets, nécessitant habituellement une exposition de 10 à 20 minutes. Cette opération se déroule à travers une petite fenêtre transparente du composant, permettant aux UV d'effacer les données existantes, préparant ainsi la mémoire pour une nouvelle programmation.

d) EEPROM (*Electricaly Erasable PROM*)

L'EEPROM est une version d'EPROM qui peut être effacée et reprogrammée par des impulsions électriques, sans avoir besoin de retirer la puce de son support.

e) Flash EPROM

La Flash EPROM est une mémoire ROM effaçable électriquement, qui permet des opérations d'effacement et de réécriture beaucoup plus rapides que celles des EEPROM classiques. Elle est largement utilisée dans les appareils électroniques, notamment pour stocker des micrologiciels et des données qui nécessitent des mises à jour régulières.

Il est important de souligner qu'après l'effacement, les EPROM, EEPROM et Flash PROM peuvent être réécrites au moyen de dispositifs de programmation spécifiques qui appliquent des charges électriques pour inscrire de nouvelles données. Cette opération de réécriture consiste en la modification de l'état électrique des cellules mémoires dans ces dispositifs, où des charges électriques spécifiques sont utilisées pour représenter des données binaires (0 ou 1). Ces modifications électriques permettent d'actualiser ou de réinscrire les informations stockées, autorisant ainsi la réutilisation de la mémoire.

3.2. Les mémoires vives

Les mémoires vives, représentées par la RAM (*Random Access Memory*), sont des mémoires volatiles utilisées pour stocker temporairement les données et les programmes en cours d'utilisation par un ordinateur. Elles offrent un accès rapide aux informations et jouent un rôle central dans le fonctionnement des systèmes informatiques en permettant la manipulation des données en temps réel. Il existe principalement deux types de RAM :

a) La SRAM (*Static RAM*)

La SRAM (RAM Statique) e st une mémoire plus ancienne qui stocke les données à l'aide de bascules électroniques, chaque bit nécessitant six transistors. Les informations restent disponibles tant que la mémoire est alimentée. Avantage : très rapide et facile à utiliser. Inconvénient : réalisation complexe et coûteuse.

b) La DRAM (*Dynamic RAM*)

La DRAM est plus simple à fabriquer que la SRAM, ce qui permet d'obtenir des puces plus denses et moins coûteuses. Chaque bit y est stocké sous forme de charge électrique dans un petit condensateur. Cette conception compacte présente un inconvénient : la charge se perd rapidement, généralement en quelques millisecondes. Pour conserver l'information, un circuit de rafraîchissement lit et réécrit régulièrement les données. C'est pour cette raison que l'on parle de RAM dynamique.

4. Structure fondamentale des mémoires

Une mémoire peut être vue comme un ensemble de registres regroupés en une seule unité. Chaque registre peut contenir des informations sous forme de mots binaires, chaque mot étant constitué d'un nombre déterminé de bits. Dans une mémoire, chaque mot est accessible via une adresse unique.

Par exemple, la figure 3 illustre une mémoire contenant 8 mots de 16 bits :

- La case mémoire à l'adresse 0 contient la donnée 462F H.
- La case mémoire à l'adresse 4 contient la donnée C821 H.

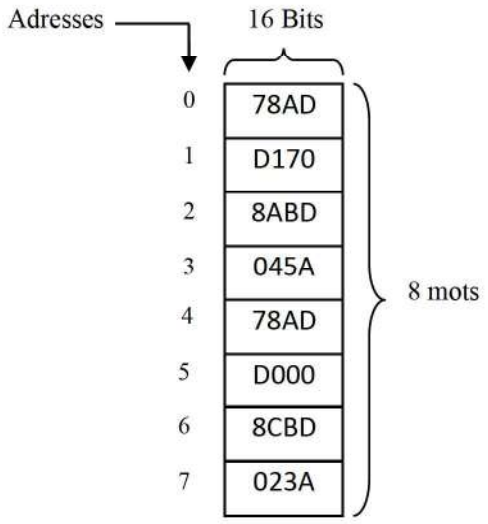


Fig. 5.3 : Mémoire contenant 8 mots de 16 bits.

L'organisation interne d'une mémoire dépend de deux caractéristiques principales : le nombre de mots qu'elle contient et le nombre de bits par mot.

Pour la suite du cours, nous utiliserons les deux définitions suivantes :

- **Profondeur d'une mémoire :** correspond au nombre de mots qu'elle contient.
- **Largeur d'une mémoire :** correspond au nombre de bits par mot.

On peut également exprimer la capacité totale d'une mémoire en bits ou en octets :

- **Capacité en bits** = Profondeur × Largeur
- **Capacité en octets** = Profondeur × Largeur / 8

Pour notre mémoire de 8 mots de 16 bits (voir Fig. 3) :

- **Capacité en bits :** 8 × 16 = 128 bits
- **Capacité en octets :** (8 × 16) ÷ 8 = 16 octets

Les mémoires présentent des caractéristiques clés, notamment des entrées d'adressage, des entrées/sorties de données, et des entrées de contrôle.

1. **Entrées d'adressage :** Ces entrées permettent au microprocesseur de spécifier l'emplacement mémoire exact auquel il souhaite accéder.
2. **Entrées/Sorties de Données :** Ce sont les canaux par lesquels les données sont transférées entre la mémoire et le microprocesseur.
3. **Entrées de Contrôle (Commandes) :** Les entrées de contrôle permettent au microprocesseur de communiquer avec la mémoire pour définir les opérations à effectuer, telles que la lecture, l'écriture, l'effacement, etc.

La figure 4 illustre les signaux d'entrée/sortie d'une mémoire qui a une structure de *p* bits par mot et de n lignes.

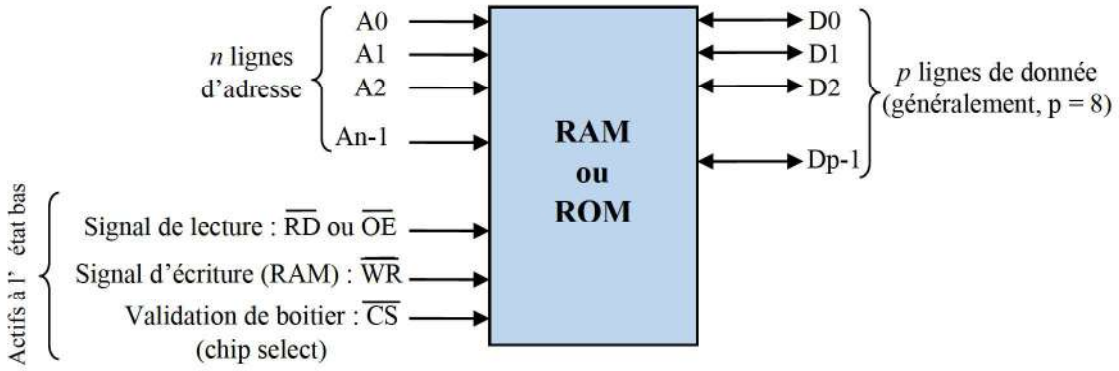


Fig. 5.4 : Les signaux d'entrée/sortie d'une mémoire.

Pour la suite du cours, nous simplifierons le bus de contrôle au strict nécessaire.

- **L'entrée d'activation *CS* (*Chip Select*) :** Cette entrée, telle que nommée, active ou désactive une mémoire, et est essentielle pour tous les types de mémoires. Si une mémoire est désactivée, elle ne réagit ni aux signaux d'entrée ni ne produit de signaux de sortie, se plaçant dans un état de haute impédance, comparable à être déconnectée.
- **L'entrée de sélection entre la lecture et l'écriture *WE* (*Write Enable*) :** Cette entrée, présente sur les mémoires de type RAM, permet de choisir entre une opération de lecture ou d'écriture. Elle peut toutefois porter un nom différent d'une mémoire à une autre (par exemple, R/W pour *Read / Write*).

5. Assemblage de mémoires

L'assemblage de boîtiers mémoire est une pratique courante pour augmenter la capacité de stockage et la largeur des données dans un système informatique. Cela permet d'accroître la mémoire disponible pour les opérations et le stockage de données. Ainsi, pour créer une mémoire de capacité C à partir d'un ensemble de boîtiers mémoire de capacité inférieure, nous pouvons procéder comme suit :

1. **Définir la mémoire cible :** Supposons que nous souhaitons construire une mémoire M de capacité C, où m est le nombre de mots et n est la taille d'un mot.
2. **Utiliser des boîtiers mémoire existants :** Nous disposons de boîtiers mémoire M' de capacité C', avec m' mots et une taille de mot n' (avec $m >= m'$ et $n >= n'$) .
3. **Calcul du nombre de boîtiers nécessaires :** Pour déterminer combien de boîtiers M' sont nécessaires pour construire la mémoire M, nous calculons deux facteurs :
   - $\boldsymbol{P = m / m'}$ (nombre de mots dans la mémoire cible par rapport à un boîtier M')
   - $\boldsymbol{Q = n / n'}$ (taille d'un mot dans la mémoire cible par rapport à un boîtier M')

   La multiplication de ces deux facteurs, c'est-à-dire (P x Q), nous donne le nombre total de boîtiers nécessaires pour construire la mémoire M.

Exemple 1 : Pour réaliser une mémoire de 1 Ko avec une taille de mot de 8 bits en utilisant des boîtiers de 256 mots de 8 bits, nous pouvons suivre la solution suivante :

1) **Identifier la taille de la mémoire ciblée :** Mémoire de 1 Ko avec une taille de mot de 8 bits : $\boldsymbol{m = 1024}$ et $\boldsymbol{n = 8}$ ➔avec un bus d'adresses de 10 bits (A9...A0) et un bus de données de 8 bits (D7...D0).
2) **Identifier la taille des boîtiers mémoire disponibles :** Boîtiers mémoires de 256 mots de 8 bits : $\boldsymbol{m' = 256}$ et $\boldsymbol{n' = 8}$ ➔avec un bus d'adresses de 8 bits (A'7...A'0) et un bus de données de 8 bits (D'7 à D'0).
3) **Calculer les facteurs d'extension des lignes et des colonnes**

$$P = m/m' = 1024/256 = 4 \text{ (extension des lignes).}$$

$$Q = n/n' = 8/8 = 1 \text{ (extension des colonnes).}$$

En multipliant P par Q, soit $P.Q = 4 \times 1 = 4$, nous obtenons le nombre total de boîtiers nécessaires pour construire la mémoire cible. Dans cet exemple, 4 boîtiers de 256 mots de 8 bits sont nécessaires pour réaliser une mémoire de 1 Ko avec une taille de mot de 8 bits.

Exemple 2 : On souhaite réaliser une mémoire de 2 Ko avec une taille de mot de 16 bits en utilisant des boîtiers de taille 1Ko mots de 4 bits :

- **Mémoire ciblé :** m = 2048 (bus d'adresse de 11 bits (A10 à A0)) et n = 8 (bus de données de 8 bits (D7... D0)).
- **Mémoires disponibles :** $\boldsymbol{m' = 1024}$ (bus d'adresse de 10 bits (A'9 à A'0)) et $\boldsymbol{n' = 4}$ (bus de données de 4 bits (D'3 à D'0)).

  P = m/m' = 2048/1024 = 2
  Q = n/n' = 16/4 = 4
  ➔ Le nombre total de boitier nécessaire est de 8.

L'assemblage de mémoires vise à augmenter la largeur et/ou la capacité de stockage de la mémoire. Pour ce faire, trois types d'assemblage sont principalement utilisés, notamment l'assemblage parallèle, l'assemblage série et l'assemblage série-parallèle.

5.1. L'assemblage en parallèle

L'assemblage en parallèle est une technique utilisée pour **augmenter la largeur d'une mémoire**, c'est-à-dire le nombre de bits contenus dans chaque mot. Cette configuration entraîne également une augmentation du nombre de lignes du bus de données.

Considérons deux mémoires RAM, chacune disposant d'un bus d'adresses de 10 bits et d'un bus de données de 4 bits. Chaque mémoire possède donc une profondeur de 1024 mots (2^{10}) et une largeur de 4 bits par mot.

Il est possible de regrouper ces deux mémoires pour former une seule mémoire ayant une largeur de 8 bits par mot. Pour cela, les deux circuits doivent être connectés de manière appropriée afin de fonctionner ensemble comme une seule unité mémoire.

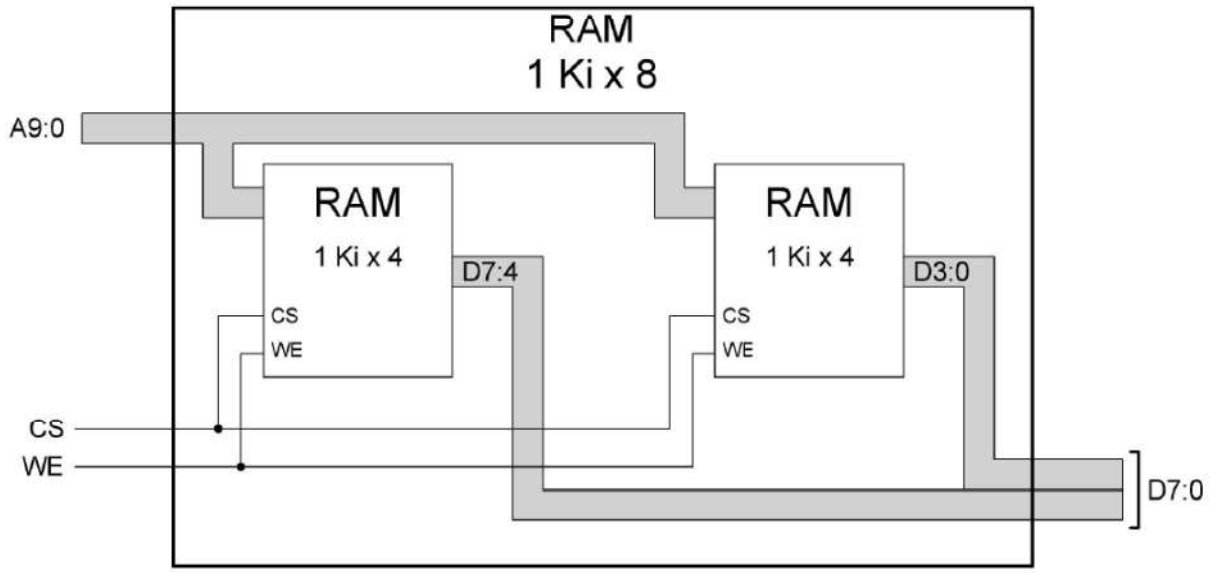


Fig. 5.5 : Assemblage en parallèle de deux RAM.

On observe que :

- Les bus d'adresses des mémoires internes sont interconnectés et constituent le bus d'adresses de la mémoire globale ;
- Les bus de contrôle des différentes mémoires sont également reliés entre eux pour former le bus de contrôle de la mémoire externe ;
- Les bus de données ne sont pas reliés directement, mais placés côte à côte. Cette juxtaposition permet de former un bus de données plus large ;
- Les deux mémoires doivent être activées simultanément afin de fournir une donnée complète sur le bus de données global.

5.2. L'assemblage en série

L'assemblage en série est une technique permettant d'**augmenter la profondeur d'une mémoire**, c'est-à-dire le nombre total de mots (ou d'adresses) qu'elle peut contenir. Cette configuration entraîne une augmentation du nombre de lignes du bus d'adresses.

Considérons deux mémoires RAM disposant chacune d'un bus d'adresses de 7 bits et d'un bus de données de 8 bits. Chaque mémoire possède donc une profondeur de 128 mots (2^7) et une largeur de 8 bits par mot.

En combinant ces deux mémoires, il est possible d'obtenir une mémoire unique ayant une profondeur de 256 mots. Pour cela, les deux circuits doivent être interconnectés de manière appropriée afin d'assurer leur fonctionnement coordonné.

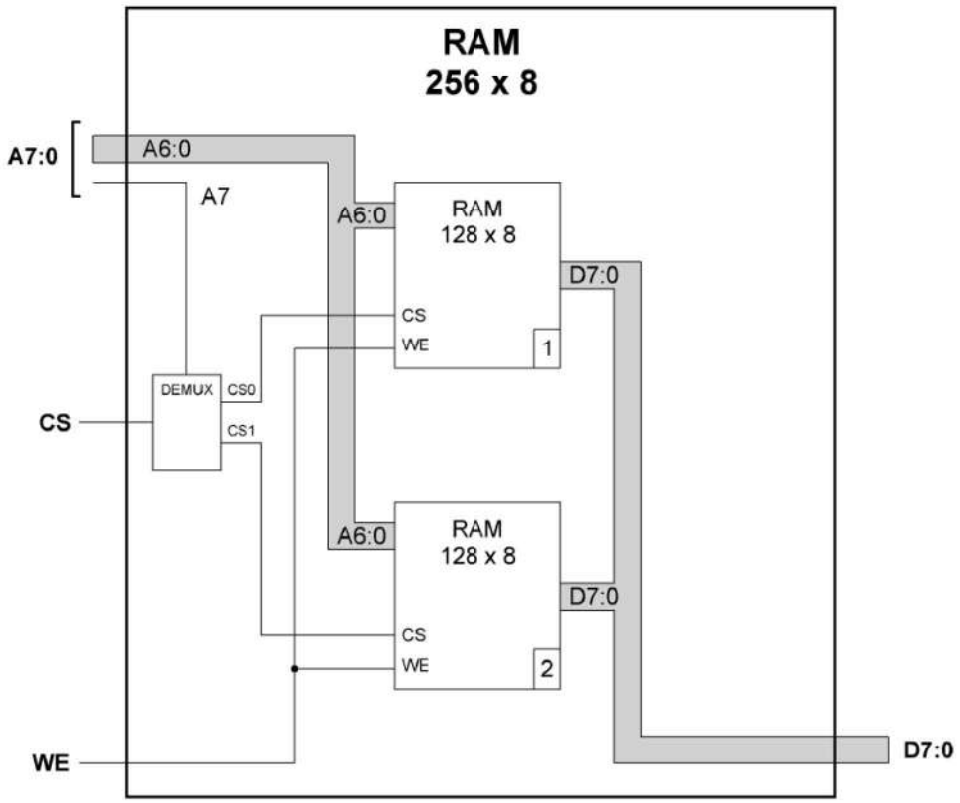


Fig. 5.6 : Assemblage **en série** de deux RAM.

On observe que :

- Les bus de données des mémoires internes sont reliés entre eux et constituent le bus de données de la mémoire globale ;
- Le même signal de lecture/écriture est appliqué aux deux mémoires, tandis que leurs entrées de sélection (CS) sont différentes : CS0 = 1 (0) et CS1 = 0 (1) ;
- Les bus d'adresses des mémoires internes sont interconnectés et représentent les bits de poids faible du bus d'adresses de la mémoire externe ;
- Un bit d'adresse supplémentaire est nécessaire pour atteindre 256 adresses ($2^8 = 256$). Ce bit sert à sélectionner l'une des deux mémoires internes à l'aide d'un démultiplexeur ;

- Les mémoires internes ne doivent jamais être actives simultanément, afin d'éviter tout conflit sur le bus de données. Ainsi, le signal CS de la mémoire externe active une seule mémoire à la fois, tandis que l'autre reste désactivée.

Dans le cas où le CS de la mémoire externe est à 1, on distingue deux situations :

- **Si A7 = 0**, alors CS0 = 1 et CS1 = 0 → la mémoire 1 est activée ;
- **Si A7 = 1**, alors CS0 = 0 et CS1 = 1 → la mémoire 2 est activée.

Si le CS de la mémoire externe est à 0, alors toutes les mémoires internes sont désactivées.

5.3. L'assemblage en parallèle et en série

Il est possible de combiner les deux techniques d'assemblage afin d'**augmenter à la fois la largeur et la profondeur** de la mémoire. Prenons l'exemple de quatre mémoires RAM disposant chacune d'un bus d'adresses de 15 bits et d'un bus de données de 8 bits. Chaque mémoire possède donc une profondeur de 32 768 mots (2^{15}) et une largeur de 8 bits par mot.

En combinant ces quatre mémoires, on obtient une mémoire globale d'une profondeur de 65 536 mots (2 × la profondeur de la RAM) et d'une largeur de 16 bits par mot (2 × la largeur de la RAM). P × Q= 2 × 2 = 4 mémoires en totale (extension)

Les deux mémoires du haut doivent être activées simultanément pour fournir un mot complet de 16 bits. Il en est de même pour les deux mémoires du bas. Cependant, lorsque les mémoires du haut sont actives, celles du bas doivent être désactivées, et inversement. Sinon, plusieurs données seraient placées en même temps sur le bus de données, ce qui entraînerait un conflit d'accès.

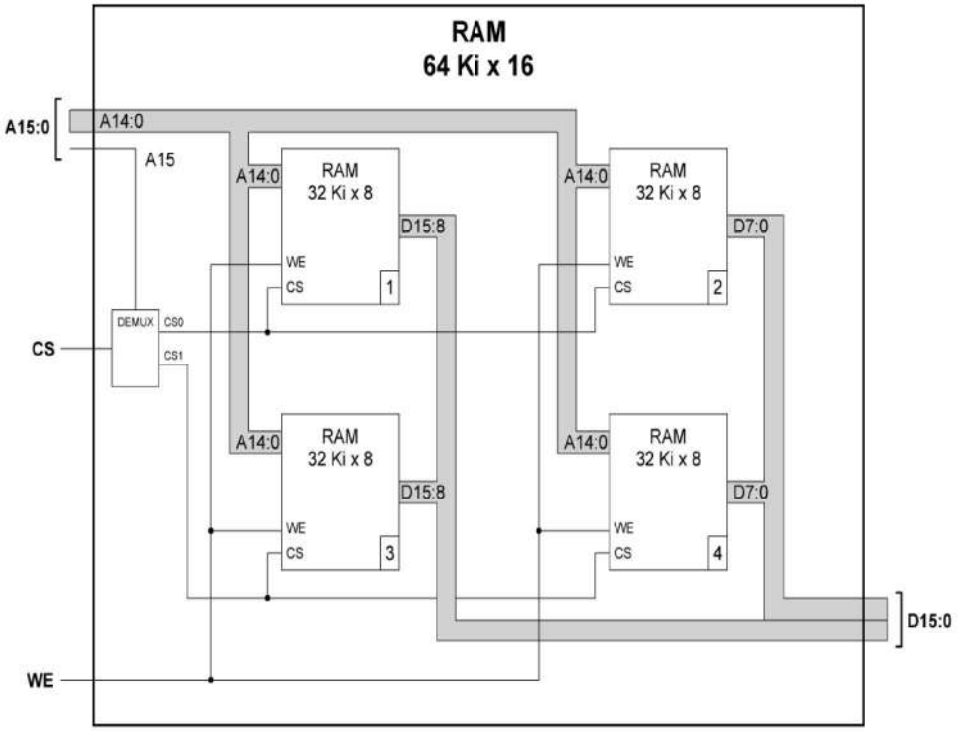


Fig. 5.7 : Assemblage en parallèle et en série des mémoires.

On distingue facilement les deux types d'assemblage :

- Les mémoires 1 et 2 sont montées en parallèle : elles sont activées simultanément et partagent le même signal CS ;
- Les mémoires 3 et 4 sont également en parallèle : elles fonctionnent en même temps avec un même signal CS ;
- Les mémoires 1 et 3 sont en série : elles ne doivent jamais être actives simultanément ;
- Les mémoires 2 et 4 sont en série : elles ne doivent pas être actives en même temps.

Lorsque le CS de la mémoire externe est à 1, deux situations sont possibles :

- Si A15 = 0, alors CS0 = 1 et CS1 = 0 → les mémoires 1 et 2 sont actives ;
- Si A15 = 1, alors CS0 = 0 et CS1 = 1 → les mémoires 3 et 4 sont actives.

Lorsque le CS de la mémoire externe est à 0, toutes les mémoires internes sont désactivées.

6. Le décodage d'adresse dans les microprocesseurs

Le décodage d'adresse a pour objectif de permettre au microprocesseur de communiquer avec différents composants tels que les mémoires, les cartes d'extension ou encore les périphériques.

Pour illustrer les différentes méthodes de décodage d'adresse étudiées dans ce cours, considérons un exemple simple : on souhaite connecter un microprocesseur à quatre éléments distincts : une mémoire ROM, une mémoire RAM, ainsi que deux périphériques notés P1 et P2.

Composants	Bus d'adresse	Bus de donnée	Capacité d'adressage
Microprocesseur	20 fils	8 fils	1 Mo
ROM	16 fils	8 fils	64 Ko
RAM	15 fils	8 fils	32 Ko
P1	10 fils	8 fils	1 Ko
P2	4 fils	8 fils	16 octets

La communication entre le microprocesseur et les différents composants s'effectue à travers les bus de données, d'adresses et de contrôle. La sélection d'un composant particulier nécessite la mise en place d'une logique de décodage d'adresse.

Tous les composants susceptibles d'échanger des informations avec le microprocesseur doivent disposer d'une entrée d'activation appelée CS (Chip Select). En effet, un seul composant doit être actif à un instant donné pour accéder aux bus. Si plusieurs composants sont activés simultanément, cela entraîne un conflit d'accès sur les bus.

Le microprocesseur choisit le composant avec lequel il souhaite communiquer en fonction de l'adresse présente sur le bus d'adresses. Cette adresse est interprétée par la logique de décodage afin d'activer le composant correspondant.

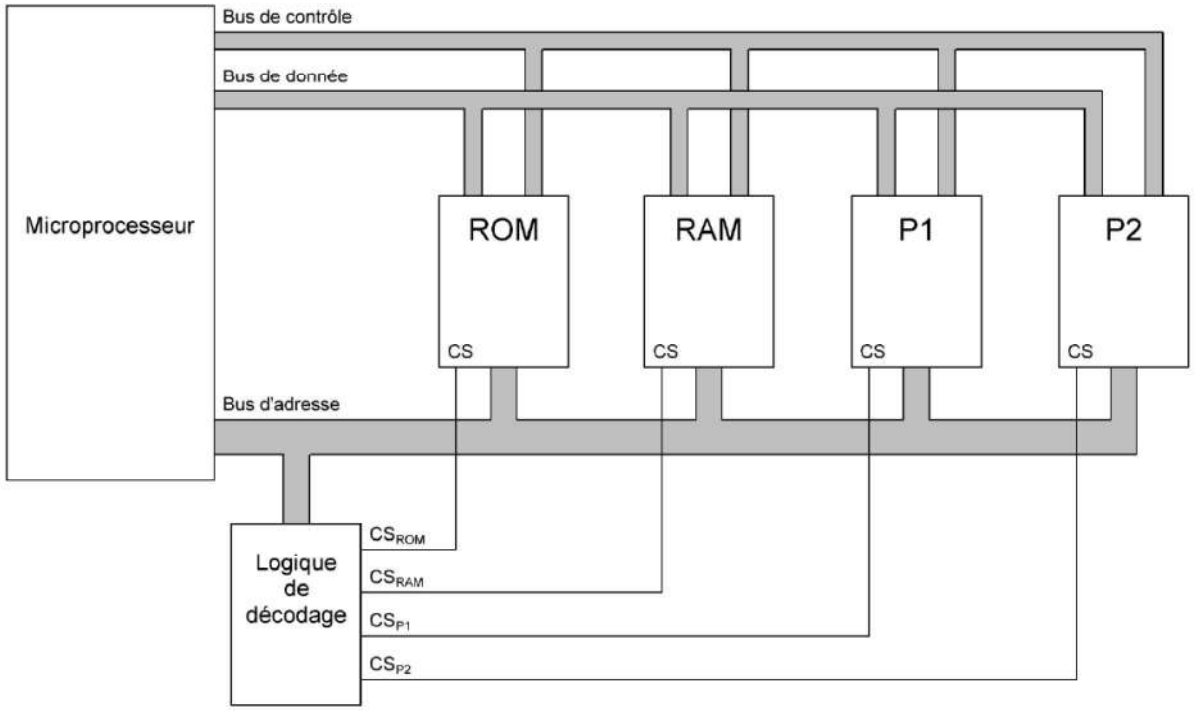


Fig. 5.8 : Schéma de câblage du système à microprocesseur.

Le décodage d'adresse consiste à analyser l'adresse présente sur le bus d'adresses du microprocesseur afin d'activer un seul composant en fonction de cette valeur.

Pour cela, **une logique de décodage** doit être réalisée à l'aide d'un circuit logique. Ce circuit génère les différents signaux CS en fonction de l'adresse fournie par le microprocesseur, permettant ainsi de sélectionner le composant concerné.

Il existe plusieurs méthodes de décodage d'adresse. Dans ce cours, nous nous limiterons à deux approches principales : **le décodage linéaire** et **le décodage par zone**. La compréhension de ces deux techniques est suffisante pour maîtriser les notions fondamentales du décodage d'adresse. Avant de les étudier en détail, il est important de noter qu'elles reposent sur un principe commun :

Le bus d'adresses du microprocesseur est divisé en deux parties :

- **les bits de poids fort**, utilisés pour **la sélection des composants** ;
- **les bits de poids faible**, utilisés pour **adresser les emplacements internes des composants.**

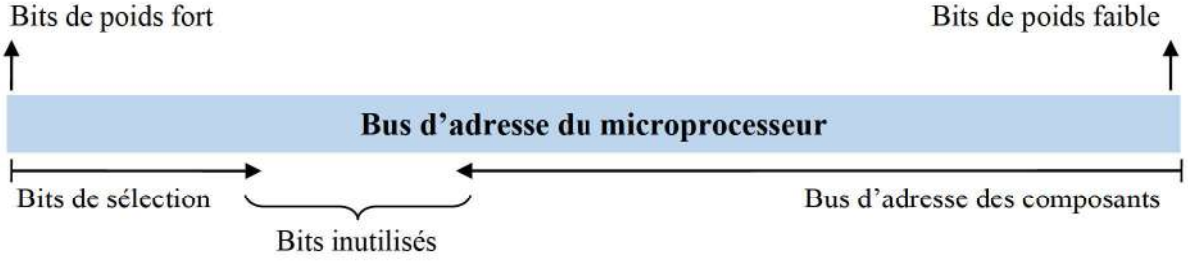


Fig. 5.9 : Représentation du bus d'adresse du microprocesseur.

Selon les configurations (nombre de composants, taille du bus d'adresses des composants, etc.), certains bits du bus d'adresses du microprocesseur peuvent ne pas être utilisés. Dans ce cas, ces bits sont généralement fixés à 0.

La figure 10 présente le schéma de câblage détaillé, en mettant en évidence la répartition entre les bits de poids fort et les bits de poids faible du bus d'adresses du microprocesseur.

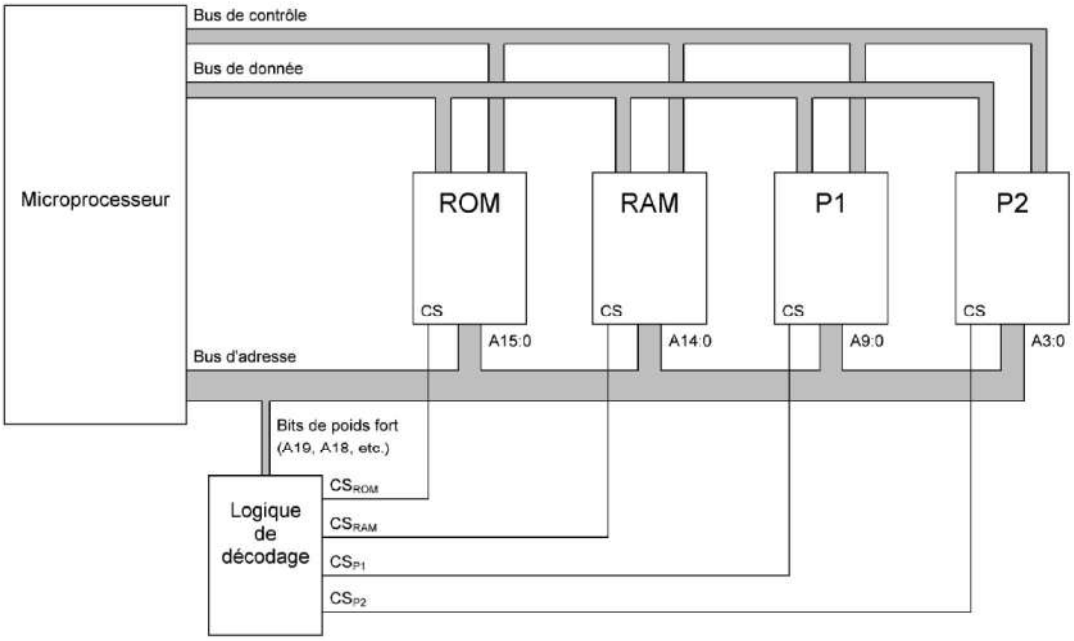


Fig. 5.10 : Schéma de câblage du système à microprocesseur.

- Les bits de poids faible du bus d'adresse du microprocesseur sont connectés aux bus d'adresses des différents composants ;
- Les bits de poids fort du bus d'adresse constituent les entrées du circuit logique de décodage. Ils correspondent aux bits de sélection. Leur nombre dépend de la technique de décodage utilisée.

6.1. Le décodage linéaire

Le décodage linéaire consiste à attribuer un bit de sélection par composant, en commençant par les bits de poids fort du bus d'adresse.

Dans notre exemple, la présence de quatre composants nécessite l'utilisation de quatre bits de sélection : A19, A18, A17 et A16. Chacun de ces bits est associé à un composant donné.

Dans la pratique, le choix d'associer un composant à un bit particulier reste libre et dépend des contraintes de conception. Le concepteur effectue cette répartition en fonction des besoins du système.

Bit de sélection	Composant associé	
A16	ROM	Si **A16 = 1**, alors la ROM est activée
A17	RAM	Si **A17 = 1**, alors la RAM est activée
A18	P1	Si **A18 = 1**, alors P1 est activée
A19	P2	Si **A19 = 1**, alors P2 est activée

6.1.1. Logique de décodage

La logique du décodage linéaire est relativement simple. L'entrée de sélection (CS) d'un composant est activée lorsque le bit de sélection correspondant est à 1. Ainsi, les entrées CS des différents composants sont directement reliées aux bits de sélection du bus d'adresses :

$$CS_{ROM} = A16, \qquad CS_{RAM} = A17, \qquad CS_{P1} = A18, \qquad CS_{P2} = A19.$$

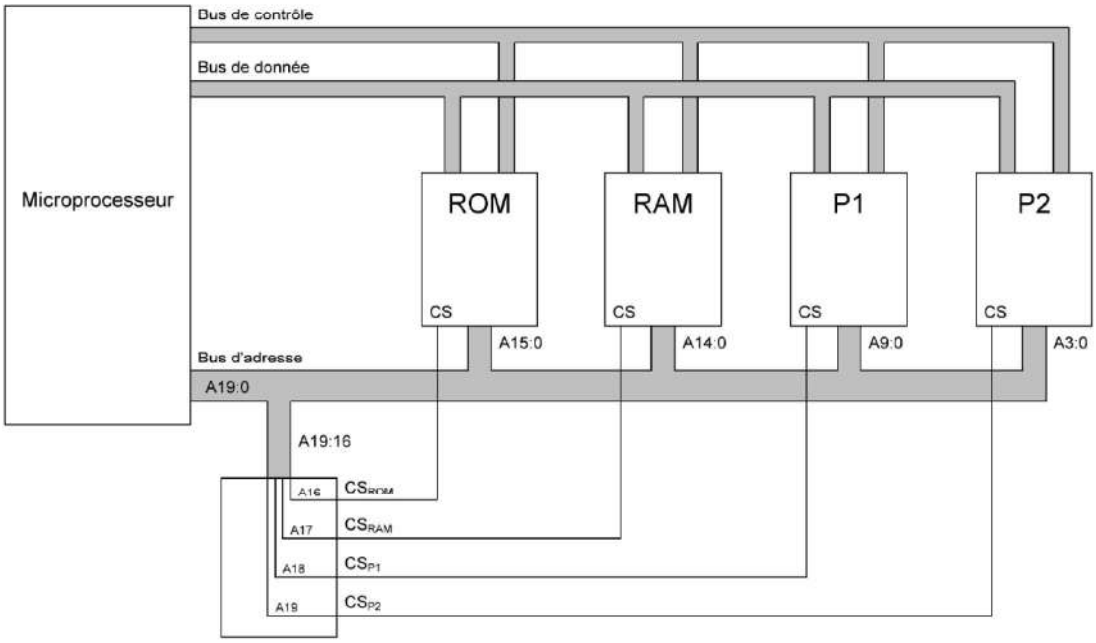


Fig. 5.11 : Schéma de câblage du système à microprocesseur.

6.1.2. Représentation de l'espace mémoire

La représentation de l'espace mémoire d'un microprocesseur permet de préciser, pour chaque composant, les plages d'adresses auxquelles il est accessible. Il s'agit donc de déterminer, pour chaque composant, l'adresse la plus basse et l'adresse la plus haute dans l'espace mémoire.

Pour cela, il est nécessaire d'identifier les bits de poids fort (utilisés pour la sélection des composants) et les bits de poids faible (utilisés pour l'adressage interne du composant). Les bits qui ne sont utilisés ni pour la sélection ni pour l'adressage sont généralement fixés à 0.

Prenons l'exemple de la ROM :

- A16 = 1 afin de sélectionner la ROM ;
- A17, A18 et A19 = 0 afin de désactiver les autres composants ;
- Pour l'adresse basse, les 16 bits d'adressage sont mis à 0 ;
- Pour l'adresse haute, ces mêmes bits sont mis à 1.

Ce qui nous donne pour tous les composants

ROM basse : **0001 0000 0000 0000 0000**$_2$ = 10000_{16}

ROM haute : **0001 1111 1111 1111 1111**$_2$ = $1FFFF_{16}$

RAM basse : **0010** 0000 0000 0000 0000$_2$ = 20000_{16}

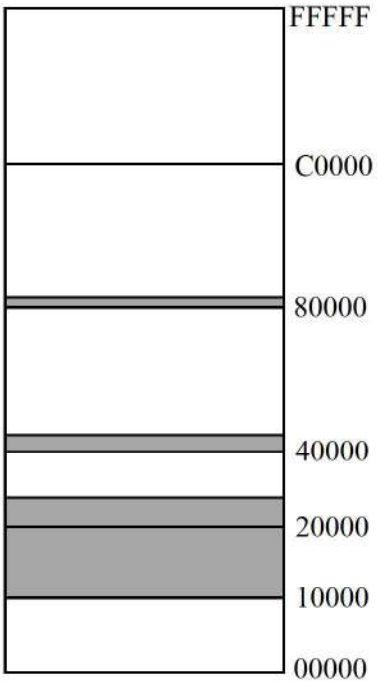
RAM haute : **0010** 0111 1111 1111 1111$_2$ = $27FFF_{16}$

P1 basse : **0100** 0000 0000 0000 0000$_2$ = 40000_{16}

P1 haute : **0100** 0000 0011 1111 1111$_2$ = $403FF_{16}$

P2 basse : **1000** 0000 0000 0000 0000$_2$ = 80000_{16}

P2 haute : **1000** 0000 0000 0000 1111$_2$ = $8000F_{16}$



Par exemple, si le microprocesseur place l'adresse 207F2H sur son bus d'adresses, alors l'adresse 7F2H de la RAM est sélectionnée :

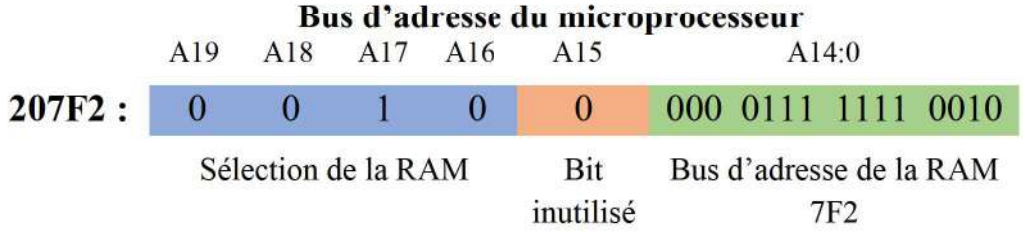


Fig. 5.12 : Exemple d'adressage de l'adresse 207F2H par le microprocesseur.

Quelques caractéristiques du décodage d'adresse linéaire :

- Mise en œuvre simple, car la logique de décodage repose essentiellement sur des connexions directes ;
- Nombre de composants limité, car l'association d'un fil par composant devient rapidement peu efficace ;
- Absence de protection contre les conflits d'accès.

Le principal inconvénient du décodage linéaire est l'absence de gestion des conflits d'accès. En effet, si l'adresse appliquée sur le bus contient plusieurs bits de sélection à 1, plusieurs composants peuvent être activés simultanément, ce qui entraîne un conflit sur le bus.

Il est néanmoins possible d'améliorer ce principe en modifiant la logique de décodage afin de sécuriser la sélection des composants. Cela consiste à n'activer le signal CS d'un composant que si tous les autres sont inactifs, c'est-à-dire si leurs bits de sélection respectifs sont à 0.

$$CS_{ROM} \equiv A19.A18.A17.A16 \equiv 0.0.0.1$$
$$CS_{RAM} \equiv A19.A18.A17.A16 \equiv 0.0.1.0$$
$$CS_{P1} \equiv A19.A18.A17.A16 \equiv 0.1.0.0$$
$$CS_{P2} \equiv A19.A18.A17.A16 \equiv 1.0.0.0$$

6.2. Le décodage par zone

Le principe du décodage par zone consiste à découper l'espace mémoire en plusieurs zones de taille identique. Chaque zone correspond à une combinaison spécifique des bits de sélection. Ces bits de sélection sont toujours choisis parmi les bits de poids fort du bus d'adresses. Ainsi :

- 1 bit de sélection permet de définir 2 zones ;
- 2 bits de sélection permettent de définir 4 zones ;
- *n* bits de sélection permettent de définir 2^n zones.

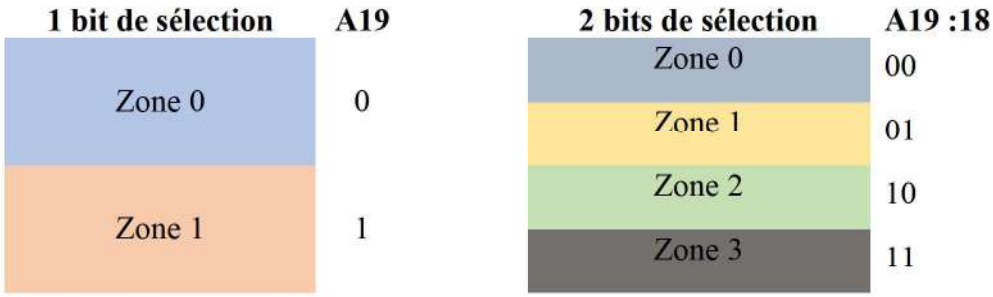


Fig. 5.13 : Principe du découpage par zone en fonction du nombre de bits de sélection.

Dans cet exemple, le bus d'adresses le plus large est celui de la ROM, qui comporte 16 lignes. Il reste donc 4 lignes supplémentaires pouvant être utilisées pour la sélection des composants.

Le décodage par zone consiste à attribuer un composant à chaque zone mémoire. Étant donné que nous disposons de quatre composants, il faut prévoir au minimum quatre zones, ce qui correspond à deux bits de sélection. Il est toutefois possible de définir un nombre de zones supérieur au nombre de composants ; les zones non utilisées restent alors disponibles.

Il est ensuite nécessaire d'associer chaque composant à une zone. Ce choix reste flexible et dépend des contraintes et des besoins du concepteur. Prévoir des zones libres permet notamment d'ajouter ultérieurement de nouveaux composants et d'assurer une meilleure évolutivité du système.

Dans la suite de cet exemple, l'espace mémoire sera divisé en huit zones, et les composants seront affectés aux quatre premières zones.

Zone	A19	A18	A17	Composant
0	0	0	0	ROM
1	0	0	1	RAM
2	0	1	0	P1
3	0	1	1	P2
4	1	0	0	zone inutilisée
5	1	0	1	zone inutilisée
6	1	1	0	zone inutilisée
7	1	1	1	zone inutilisée

6.2.1. Logique de décodage

Le signal CS de chaque composant est activé lorsque la combinaison de bits correspondant à ce composant apparaît sur les lignes de sélection. On obtient ainsi la relation suivante :

$$CS_{ROM} \equiv A19.A18\&A17 \equiv 0.0.0$$
$$CS_{RAM} \equiv A19.A18\&A17 \equiv 0.0.1$$
$$CS_{P1} \equiv A19.A18\&A17 \equiv 0.1.0$$
$$CS_{P2} \equiv A19.A18\&A17 \equiv 0.1.1$$

Cette logique de décodage correspond à celle d'un décodeur 1 parmi 8, comportant 3 entrées et 8 sorties, où une seule sortie est activée à la fois en fonction de la combinaison binaire appliquée en entrée.

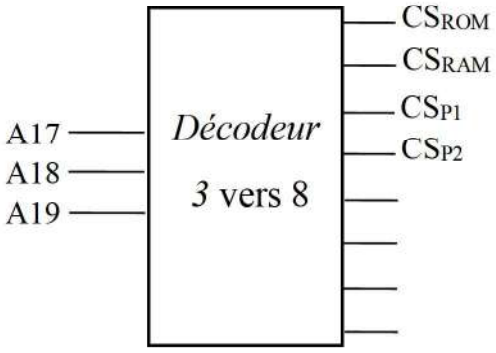


Fig. 5.14 : Décodeur 3 vers 8.

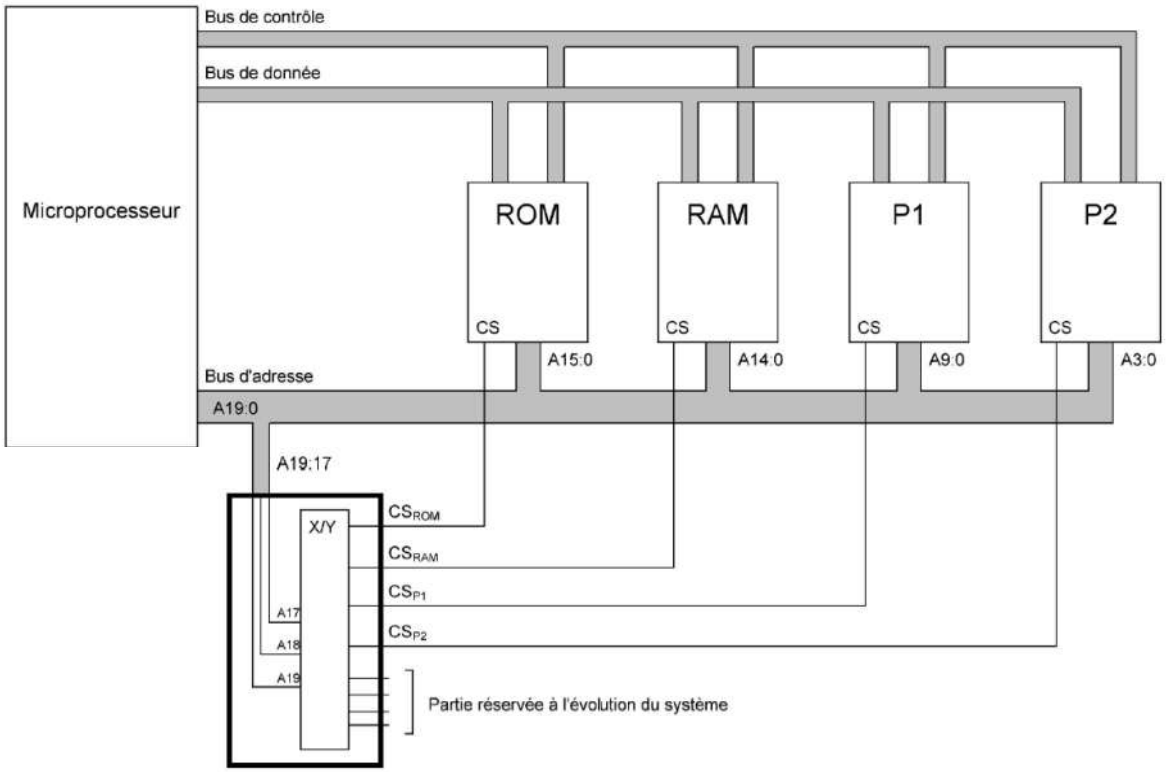


Fig. 5.15 : Schéma de câblage du système à microprocesseur.

6.2.2. Représentation de l'espace mémoire

Le principe reste similaire à celui du décodage linéaire. Il consiste à déterminer, pour chaque composant, l'adresse minimale et l'adresse maximale dans l'espace mémoire.

Pour cela, il est nécessaire d'identifier les bits de poids fort, utilisés pour la sélection du composant et les bits de poids faible, utilisés pour l'adressage interne du composant. Les bits non utilisés, c'est-à-dire ceux qui ne participent ni à la sélection ni à l'adressage, sont généralement fixés à 0. Prenons l'exemple de la ROM :

- A16 = 0, car ce bit n'est pas utilisé ;
- A17, A18 et A19 = 0, afin de sélectionner la ROM ;
- Pour l'adresse basse, les 16 bits d'adressage sont mis à 0 ;
- Pour l'adresse haute, ces 16 bits sont mis à 1.

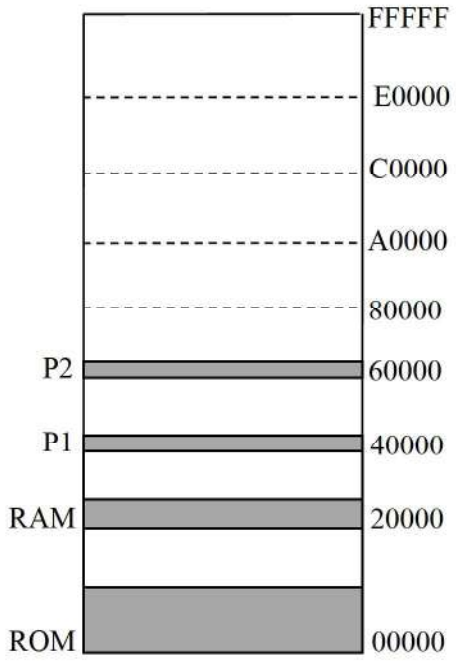
On obtient ainsi les plages d'adresses correspondantes pour l'ensemble des composants :

ROM basse : **000**0 **0000 0000 0000 0000**$_2$ = 00000$_{16}$
ROM haute : **000**0 **1111 1111 1111 1111**$_2$ = 0FFFF$_{16}$

RAM basse : **001**0 **0000 0000 0000 0000**$_2$ = 20000$_{16}$
RAM haute : **001**0 **0111 1111 1111 1111**$_2$ = 27FFF$_{16}$

P1 basse : **010**0 0000 0**000 0000 0000**$_2$ = 40000$_{16}$
P1 haute : **010**0 0000 0**011 1111 1111**$_2$ = 403FF$_{16}$

P2 basse : **011**0 0000 0000 0000 **0000**$_2$ = 60000$_{16}$
P2 haute : **011**0 0000 0000 0000 **1111**$_2$ = 6000F$_{16}$



Par exemple, si le microprocesseur place l'adresse 403A6$_{16}$ sur son bus d'adresses, cela correspond à la sélection de l'adresse 3A6$_{16}$ du composant P1.

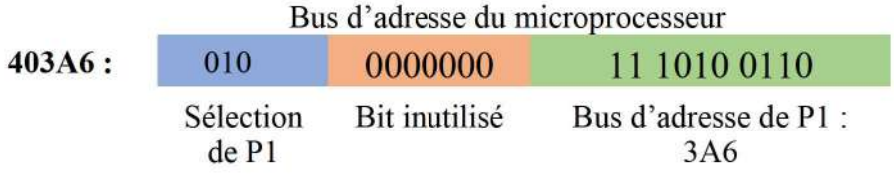


Fig. 5.16 : Exemple d'adressage de l'adresse 403A6$_{16}$ par le microprocesseur.

Quelques caractéristiques du décodage d'adresse par zone :

- Mise en œuvre simple, basée généralement sur l'utilisation d'un décodeur ;
- Permet de connecter un nombre de composants plus important comparé au décodage linéaire ;
- Lorsque le nombre de zones augmente, la taille des composants associés diminue (et inversement) ;
- Évite les conflits d'accès, car un seul composant est sélectionné à la fois.

Les interfaces d'entrées/sorties

Introduction

Les interfaces d'entrées/sorties, également désignées par l'acronyme I/O (*Input/Output*) ou E/S (Entrées/Sorties), sont des composants vitaux pour l'interaction entre le processeur et les périphériques externes. Elles facilitent les échanges de données entre le microprocesseur et divers équipements tels que les systèmes de stockage (disques durs, SSD), les capteurs, les écrans, les claviers, les imprimantes et d'autres appareils. Les interfaces E/S sont connectées au microprocesseur via plusieurs types de bus : des bus d'adresse, des bus de données et des bus de commandes comme représenté à la figure 1.

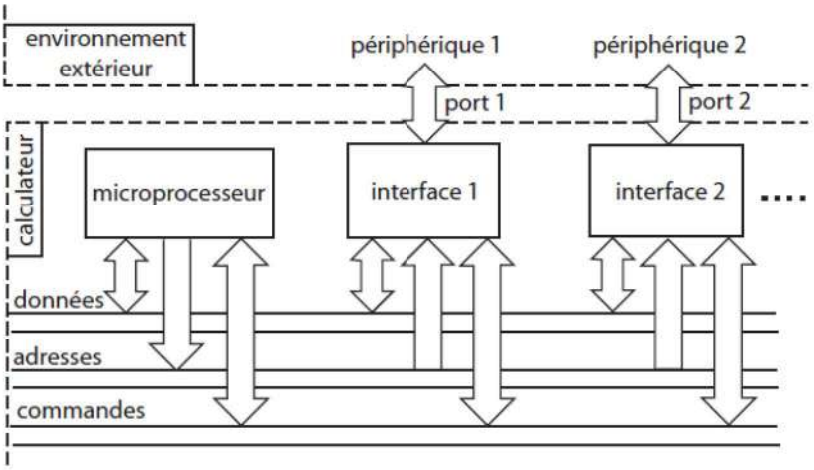


Fig. 6.1 : Microprocesseur et circuits d'interfaces.

Les connexions entre le microprocesseur et ces interfaces s'effectuent via des ports spécifiques. Les ports permettent au processeur de recevoir des données en provenance des périphériques ou de transmettre des données vers ces derniers. Ce processus permet la communication bidirectionnelle entre le microprocesseur et les dispositifs externes. Dans les systèmes à microprocesseur, il est important de distinguer entre les circuits d'E/S et les interfaces d'E/S :

- **Un circuit d'entrée/sortie** est une composante matérielle dédiée à la transmission et à la réception des signaux entre un microprocesseur et des périphériques.
- **Une interface d'entrée/sortie**, quant à elle, intègre non seulement le circuit d'E/S, mais également les protocoles de communication et les mécanismes nécessaires au décodage des adresses et à la gestion des interruptions.

1. Les circuits d'entrée/sortie

Les circuits d'E/S, sont généralement des circuits intégrés qui agissent comme interfaces entre le microprocesseur et les périphériques externes. La figure 2 illustre le schéma synoptique d'un circuit d'E/S. Ce schéma met en évidence les bus de données, d'adresse et de contrôle, utilisés pour le transfert de données, l'adressage des informations et la gestion des opérations d'entrée/sortie.

Les lignes d'E/S assurent, la transmission bidirectionnelle des données. Elles permettent l'envoi de données du périphérique vers le microprocesseur (entrée) et la réception de données du microprocesseur vers le périphérique (sortie).

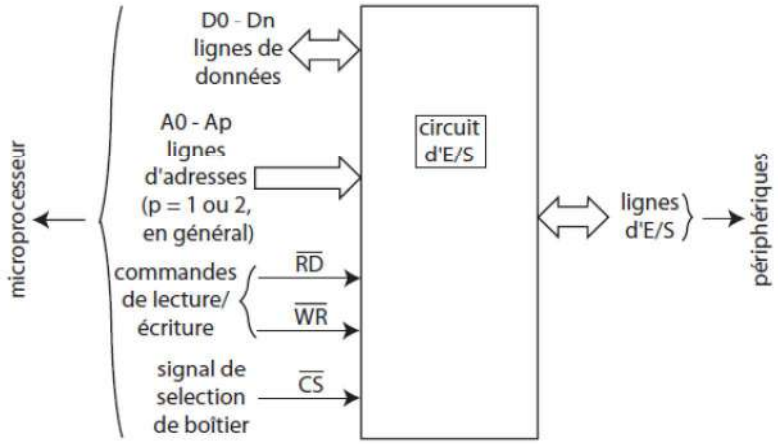


Fig. 6.2 : Schéma synoptique d'un circuit d'E/S.

2. Adressage des ports d'entrée/sortie

Un circuit d'E/S possède des registres pour gérer les échanges avec les périphériques. Ces registres sont désignés sous forme de ports d'E/S. Chaque port d'E/S correspond à un ensemble de registres, incluant :

- Registres de configuration ;
- Registres de données.

Le microprocesseur accède à un port d'E/S en spécifiant l'adresse de l'un de ses registres. Ces adresses peuvent être organisés et gérés de deux façons par le microprocesseur :

2.1. Adressage cartographique

Dans le cas de l'adressage cartographique, les adresses des ports d'entrées/sorties appartiennent au même espace d'adressage que celui de la mémoire. On dit alors que les E/S sont **mappées en mémoire**.

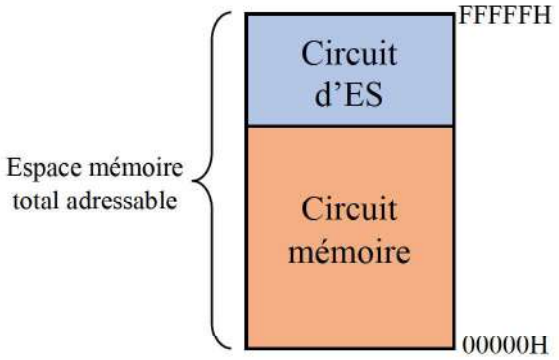


Fig. 6.3 : Principe de l'adressage cartographique.

Caractéristiques de l'adressage cartographique

- L'espace mémoire disponible est réduit, puisqu'il est partagé avec les ports d'E/S ;
- Les ports d'E/S sont adressés avec des adresses de même taille que celles des cases mémoire ;
- Les mêmes instructions peuvent être utilisées pour accéder à la mémoire et aux ports d'E/S, puisque les modes d'adressage sont identiques dans les deux cas.

2.2. Adressage indépendant

Dans le cas de l'adressage indépendant, le microprocesseur distingue deux espaces séparés :

- l'espace mémoire ;
- l'espace réservé aux ports d'E/S.

Le microprocesseur 8086 adopte ce principe, ce qui lui permet de gérer séparément les accès mémoire et les accès aux ports d'entrée/sortie.

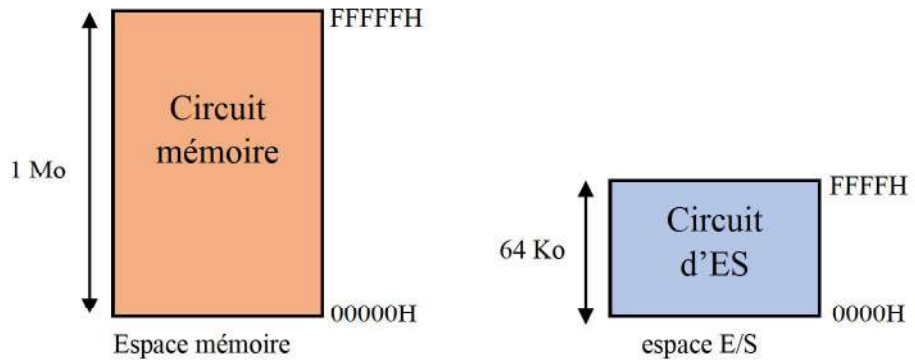


Fig. 6.4 : Principe de l'adressage indépendant.

Caractéristiques de l'adressage indépendant

- Contrairement à l'adressage cartographique, l'espace mémoire n'est pas réduit ;
- Les ports d'E/S peuvent être adressés avec un nombre de bits inférieur à celui utilisé pour la mémoire ;
- Les instructions utilisées pour la mémoire ne sont pas directement applicables aux E/S, qui nécessitent des instructions spécifiques ;
- Une même valeur d'adresse peut correspondre soit à une case mémoire, soit à un port d'E/S, d'où la nécessité d'un signal de distinction.

2.3. Cas du microprocesseur 8086

Le microprocesseur 8086 dispose d'un espace mémoire de 1 Mo (adressage sur 20 bits) et d'un espace d'entrées/sorties de 64 Ko (adressage sur 16 bits).

La distinction entre accès mémoire et accès E/S est assurée par le signal M/IO :

- M/IO = 1 : accès à la mémoire ;
- M/IO = 0 : accès aux ports d'E/S.

3. Instructions de lecture et d'écriture d'un port d'entrée/sortie

Les instructions IN et OUT en assembleur x86 sont utilisées, respectivement, pour lire et écrire des données sur les ports d'entrée/sortie (E/S) de l'ordinateur. À noter que seules les registres AL (pour un octet) et AX (pour un mot) peuvent être utilisés avec ces instructions sur l'architecture x86 ; l'utilisation d'autres registres n'est pas permise.

Dans l'architecture 8086, pour lire ou écrire sur un port d'entrées/sorties (E/S), les instructions et le format dépendent de la largeur de l'adresse du port :

3.1. Lecture d'un port E/S

- Si l'**adresse** du port E/S est codée **sur un octet** :

  IN AL, adresse *; lecture d'un port sur 8 bits ;*

  IN AX, adresse *; lecture d'un port sur 16 bits.*

- Si l'**adresse** du port E/S est codée sur **deux octets** :

  IN AL, BX *; lecture d'un port sur 8 bits ;*

  IN AX, BX *; lecture d'un port sur 16 bits.*

  où le registre BX contient l'adresse du port d'E/S à lire.

Exemple : Lecture d'un port d'E/S sur 8 bits à l'adresse 400H :

```
MOV DX, 400H
IN AL, DX
```

3.2. Écriture sur un port E/S

- Si l'**adresse** du port E/S est codée sur **un octet** :

  OUT adresse, AL *; écriture d'un port sur 8 bits ;*

  OUT adresse, AX *; écriture d'un port sur 16 bits.*

- Si l'**adresse** du port d'E/S est sur **deux octets** :

  OUT DX, AL *; écriture d'un port sur 8 bits ;*

  OUT DX, AX *; écriture d'un port sur 16 bits.*

  où le registre DX contient l'adresse du port d'E/S à écrire.

Exemple : Ecriture de la valeur 12ABH dans le port d'E/S sur 16 bits à l'adresse 93H :

```
MOV AX, 12ABH
OUT 93H, AX
```

3.3. Instruction de temporisation

Les instructions de temporisation, également connues sous le nom de "NOP" (*no operation*), sont souvent utilisées dans le contexte des interfaces d'E/S pour introduire un délai délibéré dans l'exécution d'un programme. Dans le cas du 8086, ces instructions sont généralement insérées pour fournir un court laps de temps entre deux opérations d'entrée/sortie.

En utilisant des instructions "*NOP*", un programmeur peut créer une pause entre les opérations d'E/S, permettant ainsi à un périphérique de traiter les données. Ces instructions ne réalisent aucune opération significative, mais introduisent un léger retard dans l'exécution du programme. Cela peut être utilisé pour synchroniser les vitesses de fonctionnement de différents composants ou pour respecter les temps nécessaires à certains périphériques pour traiter les données.

Syntaxe :

NOP ; Cette instruction ne fait rien, elle occupe simplement un cycle d'horloge

Exemple d'application :

Ecrire un programme qui permet de lire 200 caractères d'un octet (tapé au clavier) au port d'adresse 260H et de les écrire (affichage sur l'écran) au port d'adresse 280H. La lecture et l'écriture de chaque caractère nécessite une temporisation de 2 cycles d'horloge.

```
MOV CX, 200                          ; Nombre de caractères à lire

Lire_Ecrire_Boucle:
    MOV DX, 260H                     ; Port d'adresse de lecture
    IN AL, DX                        ; Lire le caractère tapé au clavier (port 260H)

  ; Temporisation pour la lecture (2 cycles d'horloge)
  NOP                                ; Temporisation d'un cycle d'horloge
  NOP                                ; Temporisation d'un cycle d'horloge

  ; Écriture du caractère au port 28H
  MOV DX, 280H                       ; Port d'adresse d'affichage
  OUT DX, AL                         ; Afficher le caractère lu

  ; Temporisation pour l'écriture (2 cycles d'horloge)
  NOP                                ; Premier cycle
  NOP                                ; Deuxième cycle

  LOOP Lire_Ecrire_Boucle            ; Répéter jusqu'à ce que CX atteigne 0
```

Les microprocesseurs interagissent avec les périphériques externes via deux types d'interfaces : série et parallèle. Les interfaces série transmettent les données bit par bit, tandis que les interfaces parallèles transfèrent plusieurs bits simultanément.

4. L'interface parallèle 8255

Une interface parallèle est un circuit d'E/S qui permet la transmission de donnée sur plusieurs bits d'une manière simultanée. Contrairement à une transmission série qui envoie un bit à la fois sur un seul fil, une interface parallèle utilise plusieurs fils pour transférer un groupe de bits à chaque impulsion d'horloge. Cela permet des transferts de données plus rapides, mais cela nécessite un plus grand nombre de fils pour véhiculer l'information. Les interfaces parallèles sont souvent utilisées pour connecter des périphériques rapides tels que les imprimantes, les disques durs, ou les mémoires à un ordinateur.

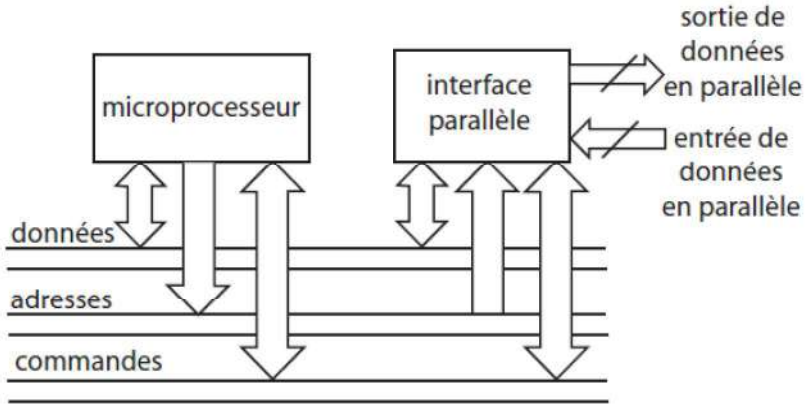


Fig. 6.5 : Microprocesseur et circuit d'interface parrallèle.

Le 8255 est un circuit intégré d'interface programmable d'E/S utilisé pour permettre au microprocesseur de communiquer avec des périphériques externes. Il est conçu pour contrôler divers types de périphériques en fournissant des ports d'entrée et de sortie programmables.

4.1. Schéma fonctionnel du 8255

La figure 6 montre le schéma fonctionnel du 8255.

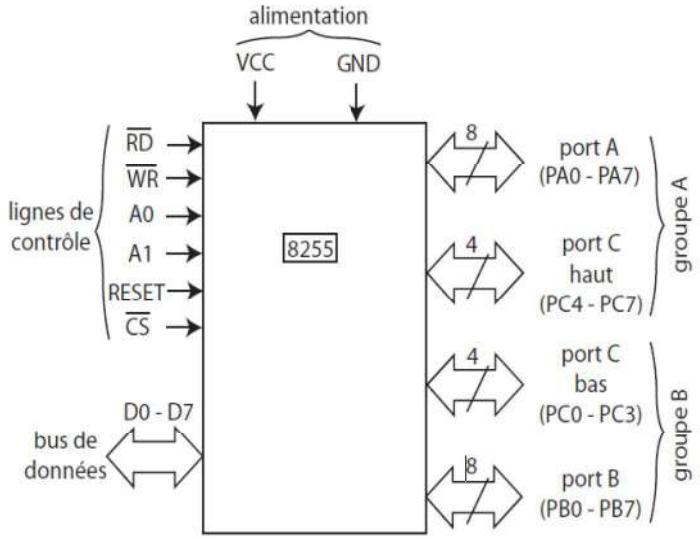


Fig. 6.6 : Schéma fonctionnel du 8255.

Le circuit 8255 dispose de 24 lignes d'entrées/sorties, organisées en trois ports de 8 bits : A, B et C. Ces ports sont eux-mêmes regroupés en deux ensembles :

- **Groupe A :** composé du port A et de la partie haute du port C ;
- **Groupe B :** composé du port B et de la partie basse du port C.

Le 8255 comprend également quatre registres internes :

- **Trois registres de données**, associés aux ports A, B et C et servant au transfert des informations ;
- **Un registre de commande**, utilisé pour configurer les ports en entrée et/ou en sortie.

4.2. Accès aux registres du 8255

L'adressage interne du 8255 repose principalement sur deux lignes d'adresse (A0 et A1) permettant de sélectionner les différents ports ainsi que le registre de commande, comme indiqué dans le tableau suivant :

A1	A0	Registre
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Registre de commande

Plus en détail l'adressage des différents ports en entrée sortie (opération d'écriture ou lecture) se fait selon la table de vérité suivante :

A1	A0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	*Opération de lecture (Read)*
0	0	0	**1**	0	Lecture du port A
0	1	0	**1**	0	Lecture du port B
1	0	0	**1**	0	Lecture du port C
					Opération d'écriture (Write)
0	0	**1**	0	0	Écriture du port A
0	1	**1**	0	0	Écriture du port B
1	0	**1**	0	0	Écriture du port C
1	1	**1**	0	0	Écriture du registre de commande*
					Le 8255 est hors fonctionnement
X	X	X	X	1	Pas de transaction
1	1	0	1	0	Illégal
X	X	1	1	0	Pas de transaction

* *Il est à préciser que le registre de commande est uniquement accessible en écriture, et sa lecture n'est pas permise.*

4.3. Configuration du 8255

La configuration du 8255 permet plusieurs modes de fonctionnement pour ses ports. D'une part, les ports peuvent être **définis en entrée ou en sortie** selon le contenu du registre de commande. D'autre part, le circuit propose **trois modes de fonctionnement** :

- **Le mode 0** est le plus simple ; il permet une configuration basique des ports en entrée ou en sortie ;
- **Les modes 1 et 2** sont plus avancés et sont utilisés pour la communication avec des périphériques nécessitant un contrôle plus élaboré.

La configuration du format (entrée ou sortie) et le choix des modes se font à partir du registre de commande. Ce registre est un mot de contrôle défini de la manière suivante :

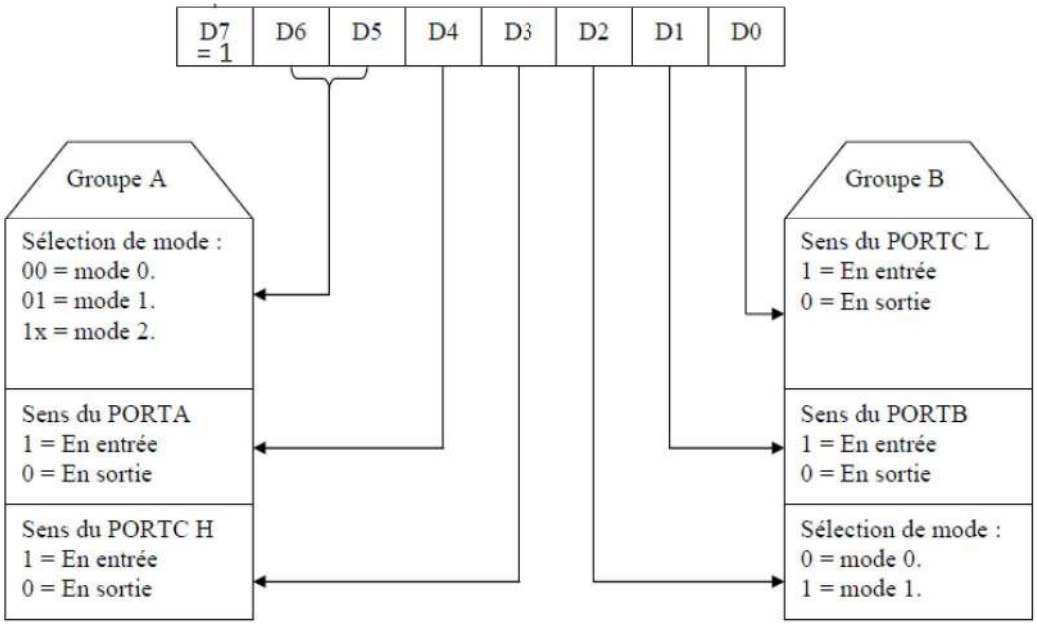


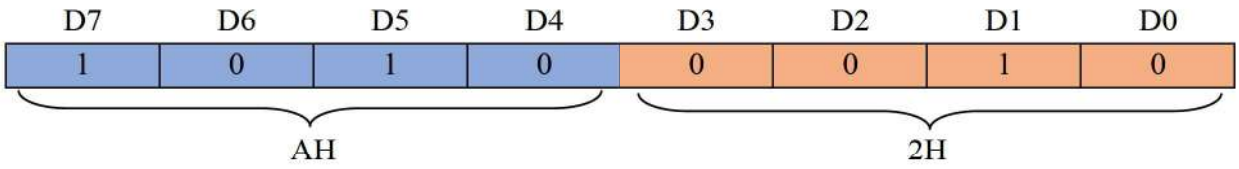
Fig. 6.7 : Registre de commande du 8255.

Exemple : Faire la configuration suivante dans le cas d'un mode de fonctionement registre :

- PORT A en mode 1 en sortie
- PORTB en mode 0 en entrée.
- PORTC en sortie.

Solution :

- PORT A en mode 1 en sortie ➔ **mode 1 :** D6 = 0, D5 = 1; **en sortie :** D4 = 0.
- PORTB en mode 0 en entrée ➔ **mode 0** : D2 = 0, **en entrée :** D1 = 1.
- PORTC en sortie ➔ D3 = 0 et D0 = 0.




```
MOV AL, A2H          ; charger dans AL la valeur de CR qui assure la configuration.
MOV DX, ADR_CR       ; l'adresse du registre de commande (CR) dans DX.
OUT DX, AL           ; écrire dans le registre de commande.
```

4.4. Connexion du 8255 sur les bus du 8086

Le bus de données du 8255 est sur 8 bits, tandis que celui du microprocesseur 8086 est sur 16 bits. Il est donc possible de connecter le bus de données du 8255 soit sur les lignes de poids faible (D0 à D7), soit sur les lignes de poids fort (D8 à D15) du 8086. Une donnée est alors envoyée ou lue par le microprocesseur 8086 :

- sur la partie faible du bus de données lorsque l'adresse concernée est paire, c'est-à-dire si le bit d'adresse A0 (ou $\overline{BHE}$) est à 0.
- sur la partie haute lorsque l'adresse est impaire (A0 ou $\overline{BHE} = 1$).

BHE *(Bus High Enable),* c'est un signal utilisé pour activer la partie de poids fort du bus, permettant le transfert sur les lignes de données de poids fort lorsque l'adresse est impaire.

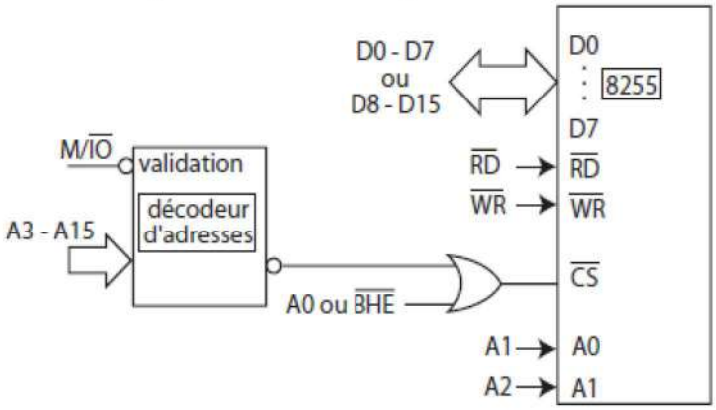


Fig. 6.8 : Connexion du 8255 sur les bus du 8086.

A partir du schéma de la figure 8, on constate que l'adresse du 8255 est une donnée divisée en deux parties.

- Tout d'abord, l'**adresse de base** du 8255 est déterminée par un décodeur d'adresses. Cette adresse de base identifie le point de départ du 8255.
- Ensuite, les lignes A1 et A2 permettent de sélectionner **les registres internes** spécifiques de ce circuit d'entrée-sortie.

4.4.1. Détermination de l'adresse de base et de l'adresse des registres du 8255

a) Détermination de l'adresse de base

L'adresse de base correspond à l'adresse initial du 8255 dans l'espace d'adressage du microcontrôleur. Elle est généralement choisie de manière arbitraire (par les bits A15 – A3). Pour illustrer cela, considérons la connexion du 8255 sur la partie basse du bus de données du 8086 (D0 – D7), avec un décodage d'adresses partiel (certaines lignes d'adresses entre A3 et A15 ne sont pas toutes exploitées).

Dans cette exemple les bits d'adresse A10 = 0, A9 = A8 = 1, tandis que les bits A15 – A11 et A7 – A3 sont non décodés. Ainsi, l'adresse de base peut être calculée comme suit :

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	**A2**	**A1**	A0
X	X	X	X	X	0	1	1	X	X	X	X	X	A1	A0	0
Adresse de base = 300H													Sélection de registre		$\overline{CS}$ = 0

Dans cet exemple, l'adresse de base 300H représente la première adresse accessible du composant, lorsque les bits d'adresses non décodés sont tous positionnés à zéro.

b) Détermination de l'adresse des registres internes du 8255

L'adresse des registres internes du 8255 est déterminée après avoir fixé l'adresse de base (bits A15-A3). Les lignes A1 et A0 sont ensuite utilisées pour identifier l'adresse des différents registres internes (tableau 1, page 7), comme suit :

- **A2 = 0 et A1 = 0 :** adresse du port A = adresse de base + 0 = 300H.
- **A2 = 0 et A1 = 1 :** adresse du port B = adresse de base + 2 = 302H.
- **A2 = 1 et A1 = 0 :** adresse du port C = adresse de base + 4 = 304H.
- **A2 = 1 et A1 = 1 :** adresse du registre de commande = adresse de base + 6 = 306H.

Exemple de programmation : soit le montage suivant :

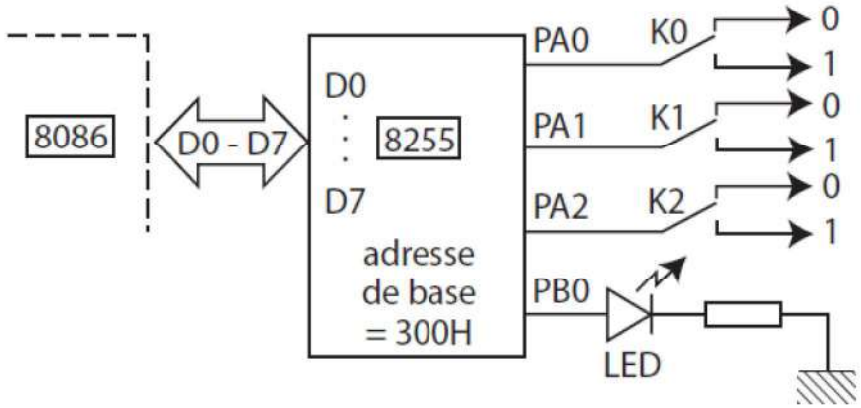


Fig. 6.9 : Exemple de montage à base de l'interface 8255.

On souhaite que la Led s'allume lorsqu'on a la combinaison : K0 = 1 et K1 = 0 et K2 = 1.

Programme :

; configuration du registre de commande (RC) – Initialisation du port A en entrée et du port B en sortie (mode 0) : donnée a placée dans le RC = 10010000B = 90H (voir Fig. 7).

```
        MOV   DX, 306H          ; charger l'adresse du RC dans le registre DX
        MOV   AL, 90H           ; charger la valeur a placer dans le RC dans AL
        OUT   DX, AL            ; écriture du RC

boucle: MOV   DX, 300H          ; charger l'adresse du port A dans DX
        IN    AL, DX            ; lecture du port A
        AND   AL, 00000111B     ; mise à 0 des états PA7 à PA3
        CMP   AL, 00000101B     ; tester si PA0 = 1, PA1 = 0 et PA2 = 1
        JNE   faux              ; si ils ne sont pas égaux ➔ aller à "faux" ...
        MOV   AL, 00000001B     ; si égaux ➔ mettre PB0 à 1 (allumer la Led)
        JMP   suite             ; et continuer au label « suite »

faux:   MOV   AL, 00000000B     ; ... mettre PB0 à 0 (éteindre la Led)
        JMP   suite             ; et continuer au label "suite"
```

```
suite:   MOV   DX, 302H      ; charger l'adresse du port B dans DX
         OUT   DX, AL        ; écriture du port B
         JMP   boucle        ; retourner pour lire le port A
```

5. L'interface série 8250

Une interface série permet d'échanger des données entre le microprocesseur et un périphérique bit par bit. Contrairement à une transmission parallèle, les interfaces série nécessitent moins de fils pour la communication, ce qui les rend souvent utilisés pour les communications à longue distance.

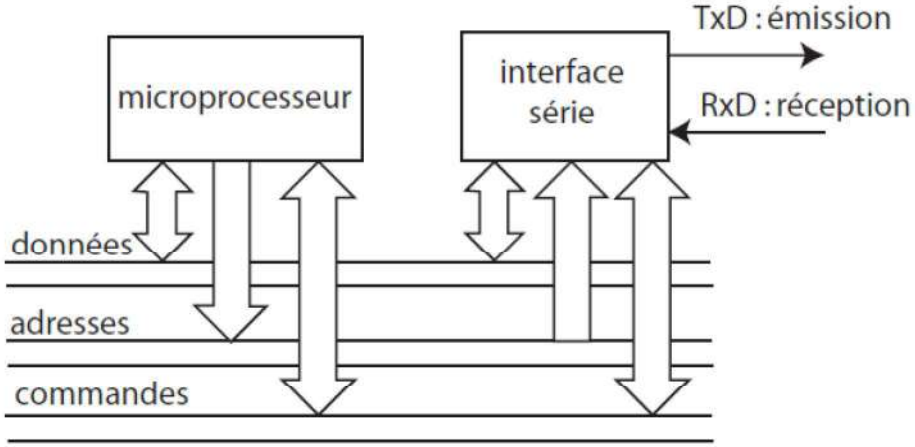


Fig. 6.10 : Microprocesseur et circuit d'interface série.

- **Avantage :** réduction du nombre de connexions nécessaires (un fil pour l'émission et un fil pour la réception) ;
- **Inconvénient :** débit de transmission généralement plus faible que celui d'une interface parallèle.

Il existe deux types de transmissions series :

- **asynchrone :** Aucun intervalle de temps fixe n'est défini entre chaque octet, permettant à chaque octet d'être émis ou reçu de manière indépendante avec des délais variables entre eux.
- **synchrone :** Les octets sont organisés en blocs et transmis à des intervalles réguliers, ce qui permet de synchroniser les données entre les deux appareils.

La **transmission asynchrone** prédominante, définie par **la norme RS232**, est largement utilisée dans les communications série. La norme RS232 établit les protocoles et signaux électriques pour les liaisons série entre équipements électroniques, jouant un rôle clé dans les connexions entre ordinateurs et périphériques.

Un circuit intégré assurant l'interface de communication série asynchrone est appelé UART (*Universal Asynchronous Receiver Transmitter*). Lorsqu'il prend en charge à la fois les modes synchrone et asynchrone, il est désigné sous le nom USART.

Exemple : transmission du caractère "E" (code ASCII 45H = 01000101B) en liaison série selon la norme RS232 :

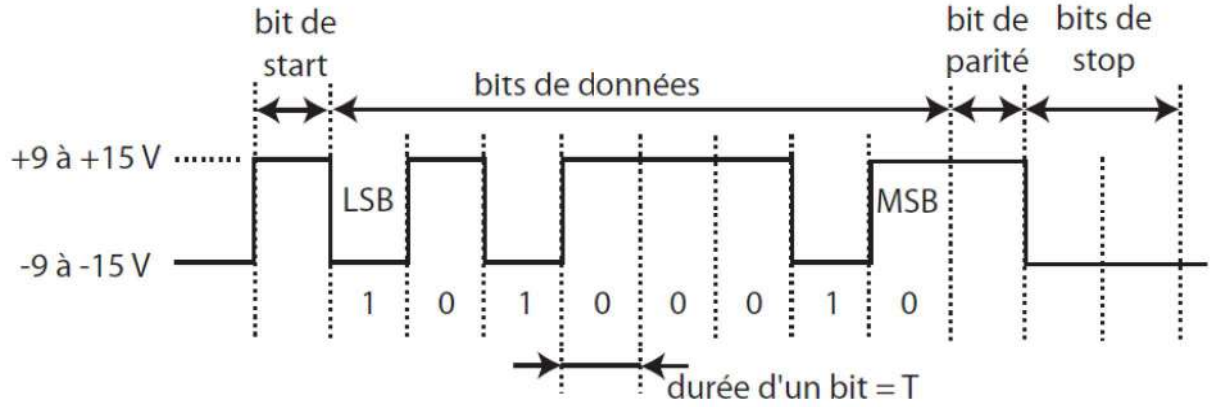


Fig. 6.11 : Exemple de transmission série asynchrone du caractère "E".

- L'**état logique 1** correspond à une tension négative comprise entre −9 V et −15 V, tandis que l'**état logique 0** correspond à une tension positive comprise entre +9 V et +15 V. Au repos, la ligne est maintenue à l'état 1 (niveau négatif) ;
- Le bit de start indique le début de la transmission d'un caractère ;
- **Les bits de données** sont transmis successivement, en commençant par le bit de poids faible. Leur nombre peut varier entre 5, 6, 7 ou 8 bits. Chaque bit est transmis pendant une durée fixe T. L'inverse de cette durée correspond à la vitesse de transmission (baud rate), exprimée en bits par seconde. Les vitesses normalisées incluent : 50, 75, 110, 134.5, 150, 300, 600, 1200, 2400, 4800 et 9600 bits/s ;
- Le **bit de parité** (optionnel) est utilisé pour détecter d'éventuelles erreurs de transmission. Sa valeur dépend du nombre de bits à 1 dans les données ;
- Les **bits de stop** (1, 1.5 ou 2 bits) marquent la fin de la transmission du caractère.

5.1. Connexion de deux équipements par une liaison série RS232

La liaison série RS232 est un standard de communication permettant l'échange de données entre deux équipements électroniques. Les dispositifs connectés sont généralement classés en deux catégories :

- **Les équipements terminaux de données (DTE)**, qui génèrent ou exploitent les données. Il s'agit par exemple des ordinateurs, terminaux, imprimantes ou scanners ;
- **Les équipements de communication de données (DCE)**, qui assurent la transmission des données sur le support de communication, tels que les modems, routeurs ou concentrateurs.

Les connecteurs RS232 transportent plusieurs signaux nécessaires à la communication entre DTE et DCE. Pour établir la liaison, on utilise généralement des connecteurs normalisés de type DB9 ou DB25.

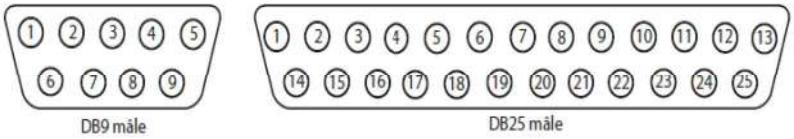


Fig. 6.12 : Connecteurs DB9 et DB25 mâle.

Différents signaux sont transportés par ces connecteurs :

Signal	N° de broche DB9	N° de broche DB25	Description	Sens	
				DTE	DCE
TxD	3	2	Transmit Data	Sortie	Entrée
RxD	2	3	Receive Data	Entrée	Sortie
RTS	7	4	Request To Send	Sortie	Entrée
CTS	8	5	Clear To Send	Entrée	Sortie
DTR	4	20	Data Terminal Ready	Sortie	Entrée
DSR	6	6	Data Set Ready	Entrée	Sortie
DCD	1	8	Data Carrier Detect	Entrée	Sortie
RI	9	22	Ring Indicator	Entrée	Sortie
GND	5	7	Ground	/	/

- **TxD :** signal de données transmises par le DTE vers le DCE.
- **RxD :** signal de données reçues par le DTE depuis l'équipement DCE.
- **RTS :** signal envoyé par le DTE pour demander l'autorisation de transmettre des données.
- **CTS :** signal envoyé par le DCE pour autoriser la transmission de données.
- **DTR :** signal envoyé par le DTE pour indiquer qu'il est prêt à transmettre des données.
- **DSR :** signal envoyé par le DCE pour indiquer qu'il est prêt à recevoir des données.
- **DCD :** signal envoyé par le DCE pour indiquer que la connexion est établie et que la communication peut commencer.
- **RI :** signal envoyé par le DCE pour indiquer une sonnerie entrante.

À noter que seuls les signaux **TxD** et **RxD** sont utilisés pour la transmission des données. Les autres lignes servent principalement au contrôle et à la gestion du transfert d'informations.

Exemple d'application des liaisons série :

Lors d'une communication via une ligne téléphonique, le modem assure l'adaptation entre les signaux numériques et le canal analogique. Il convertit ainsi les données numériques issues de l'interface série en signaux analogiques (modulation) compatibles avec la ligne téléphonique. Inversement, il reconvertit les signaux analogiques reçus en signaux numériques (démodulation) afin qu'ils puissent être traités par l'interface série.

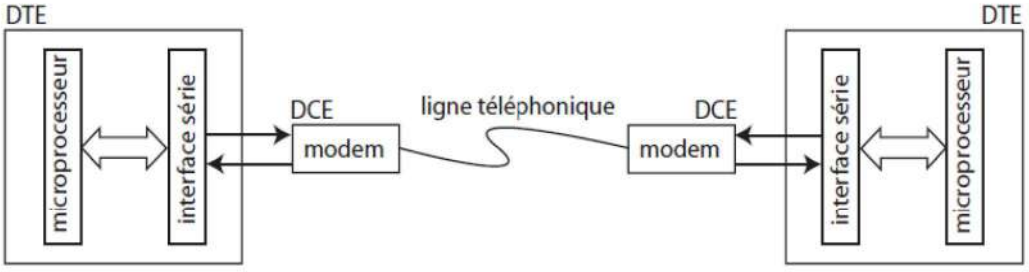


Fig. 6.13 : Exemple de transmission série.

5.2. Mise en œuvre d'une interface série, l'UART 8250

L'UART 8250, présente un ensemble de 11 registres essentiels à sa fonctionnalité. Ces registres servent à stocker et à contrôler divers aspects de la communication série. Parmi ces registres on retrouve :

- **Le registre LCR (*Line Control Register*) :** Il est utilisé pour configurer les paramètres de la communication série, tels que le débit de transmission, le nombre de bits de données, le nombre de bits de stop, le mode de parité, etc.
- **Le registre LSR (*Line Status Register*) :** Il est utilisé pour indiquer l'état de la communication série, tels que la présence de données à lire ou à écrire, les erreurs de communication, etc.
- **Registres du diviseur d'horloge (*DLM, DLL*) :** Ils sont utilisés pour le calcul de de la vitesse de transmission de la communication série. En effet, la vitesse de transmission est fixée par la valeur du diviseur d'horloge. LM (*Divisor Latch MSB*) est utilisé pour stocker les bits de poids forts du diviseur de fréquence, tandis que DLL (*Divisor Latch LSB*) pour stocker les bits de poids faibles

$$\text{vitesse (bit/ s)} = \frac{\text{fréquence d'horloge}}{16 \text{ x (DLM, DLL)}}$$

Exemple de calcul : vitesse de transmission désirée = 1200 bit/s, fréquence d'horloge = 1.8432 MHz, détermination de la valeur du diviseur d'horloge :

$$\text{diviseur} = \frac{\text{fréquence d'horloge}}{16 \text{ x vitesse}} = \frac{1.8432 \text{ x } 10^6}{16 \text{ x } 1200} = 96 \text{ DLM} = 0 \text{ et DLL} = 96.$$

Le rôle principal du diviseur d'horloge est de réduire la fréquence d'un signal d'horloge pour l'adapter aux besoins de différents circuits ou composants électroniques qui fonctionnent à des vitesses différentes.

5.3. Port USB (*Universal Serial Bus*)

L'USB est une norme d'interface permettant la connexion de divers périphériques à des ordinateurs ou à d'autres appareils compatibles. Cette norme a été conçue pour simplifier et standardiser les connexions entre les appareils et les ordinateurs. L'USB est devenu un élément essentiel de l'informatique en raison de sa facilité d'utilisation et de ses améliorations constantes de vitesse et de capacité, passant des premières versions USB 1.0 à l'USB 3.2 actuel.

Utilisé pour connecter une variété de périphériques tels que des souris, des claviers, des disques durs externes, des smartphones, et bien d'autres, l'USB offre une interface standard, simplifiant la connexion des appareils pour les utilisateurs.

Comparé à l'interface série RS232, l'USB offre une connectivité plus rapide, polyvalente et universelle, remplaçant progressivement les anciennes interfaces en raison de ses

performances accrues, de sa facilité d'utilisation et de ses connecteurs compacts et réversibles.

Le port USB est constitué de 4 broches : une paire d'alimentation (VCC et GND) et une paire de données inversées (-DATA et +DATA) pour les transferts de données en mode différentiel. Cette configuration permet des transferts de données via deux fils, où les variations de tension entre ces fils transportent l'information.

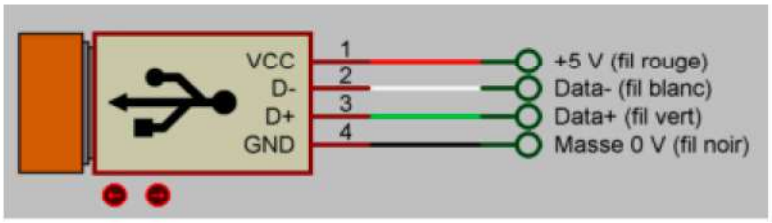


Fig. 6.14 : Broches d'un Connecteur USB

5.4. Le *Serial Peripheral Interface* (SPI)

Le *Serial Peripheral Interface* (SPI) est une norme de communication série utilisée pour connecter des microcontrôleurs à des périphériques. Contrairement à l'USB, il est plus limité en termes de périphériques connectables, mais il offre une connexion série rapide et simple pour des applications spécifiques. Le SPI est couramment utilisé pour la communication entre microcontrôleurs et divers capteurs, écrans, mémoires et autres périphériques.

Cette interface est principalement utilisée dans des applications nécessitant des transferts de données à grande vitesse, car le SPI permet des échanges à des vitesses élevées. Il fonctionne en mode série synchrone, permettant la transmission simultanée de données dans les deux sens.

Comparé à l'USB, le SPI offre une vitesse de transfert plus élevée dans des applications spécifiques, mais il est plus limité en termes de types de périphériques connectés. Son architecture et son mode de communication sont adaptés aux applications nécessitant des vitesses de transmission élevées.

Le microcontrôleur PIC16F877

1. Du microprocesseur au microcontrôleur

Un microcontrôleur est un composant électronique intégré qui regroupe, au sein d'un même boîtier, un microprocesseur, de la mémoire, des interfaces d'entrée/sortie, des *timers* (compteurs), des convertisseurs analogique-numérique, ainsi que d'autres fonctionnalités spécifiques à la tâche qu'il doit accomplir.

En résumé, un microcontrôleur peut être considéré comme un "ordinateur miniature" spécialement conçu pour gérer et contrôler des systèmes électroniques dédiés. Il est élaboré pour superviser et diriger diverses applications, allant du contrôle embarqué dans les voitures à la gestion des capteurs et dispositifs de sécurité, en passant par la coordination des fonctions dans des dispositifs médicaux avancés, et bien plus encore.

Les microcontrôleurs sont spécifiquement conçus pour des applications qui n'exigent pas une puissance de calcul complexe étendue, mais qui requièrent une gestion intensive des entrées/sorties. Ces composants sont optimisés pour des tâches où la manipulation efficace des signaux provenant de capteurs, l'interaction avec des dispositifs externes, ou la régulation de systèmes en temps réel sont primordiales.

Il existe plusieurs familles de microcontrôleurs dont les plus connues sont :

- **Atmel :** AT; familles AT89Sxxxx, AT90xxxx, …
- **Motorolla :** famille 68HCxxx, …
- **Microship :** PIC ; familles 12Cxxx, 16Cxxx, 16Fxxx, 18Fxxx, …
- **Intel :** famille 80C186XX
- **STMicroelectronics :** famille STX

Nous allons nous intéresser dans le cadre de ce cours à la famille Microchip PIC (***P**rogrammable **I**ntegrated **C**ircuit*) de moyenne gamme (*MIDRANGE*).

2. Présentation d'un microcontrôleur PIC

Les microcontrôleurs PIC sont des composants de type RISC (*Reduced Instruction Set Computing*), ou encore composants à jeu d'instructions réduit. Chaque instruction complexe peut être décomposée en plusieurs instructions simples. La réduction du nombre d'instructions facilite et accélère le processus de décodage, ce qui permet au composant de fonctionner plus rapidement.

Les microcontrôleurs de la famille PIC adoptent une architecture interne de type Harvard, comme illustré dans la Figure 1. Dans ce schéma, les instructions et les données sont distinctement séparées et circulent sur des bus distincts. Cette conception permet une exécution efficiente des instructions, nécessitant un seul cycle machine, car elle autorise la recherche simultanée du code instruction et/ou des données qu'elle manipule grâce aux deux bus.

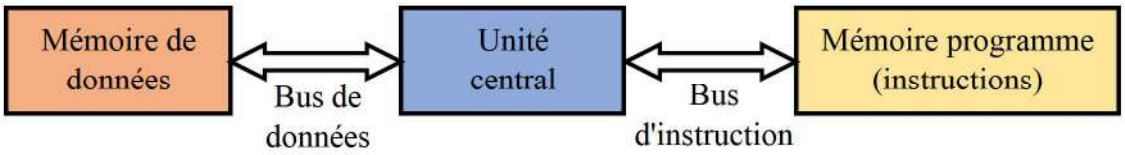


Fig. 7.1 : Principe de l'architecture, de type Harvard, utilisée par les microcontrôleurs PIC.

Selon la **taille du bus de données**, les microcontrôleurs PIC sont disponibles en trois catégories : 8 bits, 16 bits et 32 bits. Cependant, la catégorie des PIC à processeur 8 bits est la plus répandue. Par ailleurs, cette famille de microcontrôleurs à processeur 8 bits est subdivisée en trois grandes catégories distinctes.

- ✓ **Base-Line :** ils utilisent des mots d'instruction de 12 bits.
- ✓ **Mid-Range :** ils utilisent des mots d'instruction de 14 bits.
- ✓ **High-End :** ils utilisent des mots d'instruction de 16 bits.

Les PICs de la gamme *Mid-Range* sont caractérisés par un ensemble de 35 instructions. Ces microcontrôleurs peuvent exécuter chaque instruction (à l'exception des sauts) en un seul cycle machine, ce qui entraîne des performances élevées. Cette conception permet d'atteindre des vitesses considérables et facilite une assimilation rapide des instructions.

La fréquence de l'horloge fournie au PIC est divisée par 4 pour déterminer la base de temps d'un cycle. En utilisant, par exemple, un quartz de 4 MHz, cela se traduit par 1 000 000 de cycles par seconde. Étant donné que le PIC exécute pratiquement une instruction par cycle, à l'exception des sauts, cela se traduit par une puissance d'environ 1 MIPS. Un microcontrôleur PIC est identifiable par un numéro de format suivant : **xx(L)XXyy –zz**.

- **xx :** Famille du composant (12, 14, 16, 17, 18)
- **L :** indique que le microcontrôleur est une version basse tension (Low Voltage), capable de fonctionner avec une plage d'alimentation étendue.
- **XX :** Type de mémoire de programme
  - **C** - EPROM ou EEPROM
  - **CR** - PROM
  - **F** - FLASH
- **yy :** Identification
- **zz :** Vitesse maximum du quartz

La référence **PIC16F84-10**, pour illustrer, identifie un microcontrôleur de la catégorie *Mid-Range*. Sa mémoire programme est de type FLASH, référencée par le numéro 84, et il est apte à fonctionner avec une fréquence d'horloge de 10 MHz.

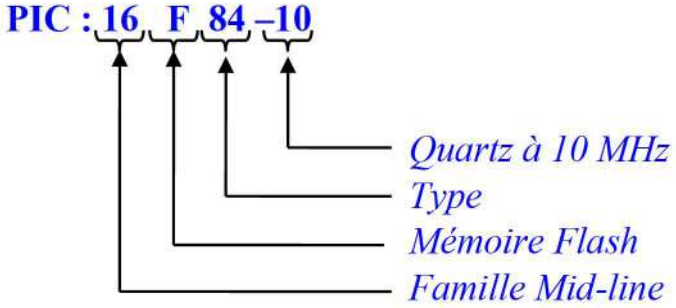


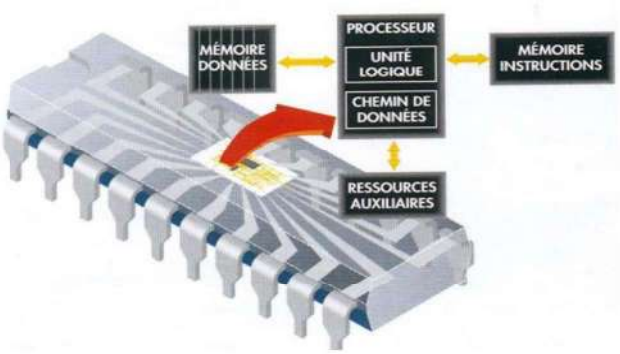
Fig. 7.2 : Exemple d'identification d'un microcontrôleur PIC.

Dans la continuité de ce chapitre, nous avons choisi d'aborder l'apprentissage des microcontrôleurs de la famille PIC16F87x (x = 3, 4, 6, 7). Ces microcontrôleurs sont emblématiques de la catégorie dite "moyenne puissance" de Microchip. L'étude se focalisera sur le modèle PIC16F877, dont nous détaillerons la configuration structurelle ainsi que les composants essentiels.

3. Le microcontrôleur PIC16F877

3.1. Structure générale du microcontrôleur PIC16F877

La structure générale du PIC16F877 est organisée autour de quatre blocs principaux, comme illustré à la figure 3 :



- **Mémoire de programme**
- **Mémoire de données**
- **Processeur**
- **Ressources auxiliaires (périphériques)**

❖ **La mémoire de programme**

La mémoire de programme contient l'ensemble des instructions qui pilotent l'application à laquelle le microcontrôleur est destiné. Il s'agit d'une mémoire non volatile de type FLASH, ce qui signifie qu'elle peut être programmée, effacée et reprogrammée par l'utilisateur à l'aide d'un programmateur et d'un ordinateur. Dans le cas du PIC16F877, la mémoire programme permet de stocker des applications relativement complexes grâce à sa capacité étendue par rapport aux microcontrôleurs d'entrée de gamme.

❖ **La mémoire de données**

La mémoire de données est utilisée pour le stockage temporaire :

- des variables,
- des données intermédiaires,
- des résultats de calcul,

lors de l'exécution du programme.

❖ **Le processeur**

Le processeur du PIC16F877 est constitué de deux éléments principaux :

- **Une unité arithmétique et logique (UAL)**, chargée d'exécuter les opérations arithmétiques et logiques.
- **Un registre de travail**, noté **W**, sur lequel l'UAL effectue l'ensemble des opérations.

❖ **Ressources auxiliaires (périphériques intégrés du PIC16F877)**

Le PIC16F877 intègre un ensemble riche de périphériques internes, facilitant son utilisation dans les applications électrotechniques et industrielles :

- **Temporisateurs (*Timers*) :** il s'agit des Timer0, Timer1 et Timer2. Ils permettent de mesurer le temps ou de générer des délais de manière automatique. Ce sont des compteurs qui s'incrémentent à partir d'une horloge interne ou externe, sans intervention du programme principal. Ils sont utilisés pour réaliser des temporisations précises, générer des signaux périodiques ou déclencher des interruptions à des instants précis.
- **Ports d'Entrée/Sortie (E/S)** : Le microcontrôleur comporte **cinq ports d'E/S** (PORTA à PORTE), permettant la connexion avec des capteurs, des actionneurs et des circuits externes. Ces ports sont **bidirectionnels** et configurables par logiciel.
- **Modules Capture, Comparaison et PWM (CCP)** : ils permettent de travailler avec des signaux liés au temps (mesure, génération ou commande).
  - **Mode Capture (16 bits) :** il permet d'enregistrer l'instant précis où un événement se produit (par exemple l'arrivée d'un signal). Il est utilisé pour mesurer des durées, des périodes ou des fréquences.
  - **Mode Comparaison :** ce mode compare une valeur interne avec celle d'un timer. Lorsque les deux valeurs sont égales, le microcontrôleur peut déclencher une action (changer l'état d'une sortie, générer une interruption, etc.).
  - **Mode PWM (*Pulse Width Modulation*) :** ce mode génère un signal dont le rapport cyclique est variable. Il est très utilisé pour commander la vitesse d'un moteur, la luminosité d'une LED ou des systèmes de puissance.
- **Convertisseur Analogique–Numérique (CAN / ADC)** : Le PIC16F877 est équipé d'un **convertisseur analogique–numérique 10 bits**, avec plusieurs entrées analogiques, permettant la conversion de signaux analogiques en données numériques.
- **Port Série Synchrone et Asynchrone (SSP / USART)** : Ces interfaces permettent au microcontrôleur de communiquer avec d'autres dispositifs électroniques, tels que des capteurs, des mémoires externes ou d'autres microcontrôleurs
  - Interface **SSP** supportant les protocoles **SPI** et **I²C**
  - Interface **USART** pour la communication série asynchrone ou synchrone

3.2. Caractéristiques du CPU du microcontrôleur PIC16F877

Le CPU du microcontrôleur PIC16F877 présente les caractéristiques suivantes :

- **Architecture RISC (8 bits) :** Le PIC16F877 repose sur une architecture **RISC 8 bits**, caractérisée par un jeu d'instructions réduit et optimisé, permettant une exécution rapide (la majorité des instructions s'exécutent en un seul cycle d'horloge).

- **Mémoire programme :** La mémoire programme a une capacité de 8 kilo-mots, chaque mot étant codé sur 14 bits. Cette mémoire est de type FLASH, ce qui permet la programmation, l'effacement et la reprogrammation par l'utilisateur.
- **Mémoire de données (RAM)** : Le microcontrôleur dispose de **368 octets de mémoire RAM**, utilisés pour le stockage temporaire des variables, des registres et des résultats intermédiaires lors de l'exécution du programme.
- **EEPROM de données** : Une mémoire **EEPROM de 256 octets** est intégrée, offrant un espace de stockage **non volatile** pour conserver des données même en cas de coupure d'alimentation.
- **Générateur d'horloge** : Le générateur d'horloge peut fonctionner avec :
  - un oscillateur RC,
  - un quartz ou résonateur externe, permettant un fonctionnement jusqu'à une fréquence maximale de **20 MHz**, offrant ainsi une grande flexibilité selon les besoins de l'application.
- **Mode Sleep (veille)** : Le PIC16F877 dispose d'un **mode Sleep** permettant de réduire significativement la consommation d'énergie lorsque le microcontrôleur n'est pas en activité. Ce mode est particulièrement utile dans les applications à faible consommation.
- **Programmation ICSP** : La programmation peut être réalisée en **ICSP (*In-Circuit Serial Programming*)**, ce qui permet de programmer ou reprogrammer le microcontrôleur directement sur la carte électronique, sans le retirer de son support.

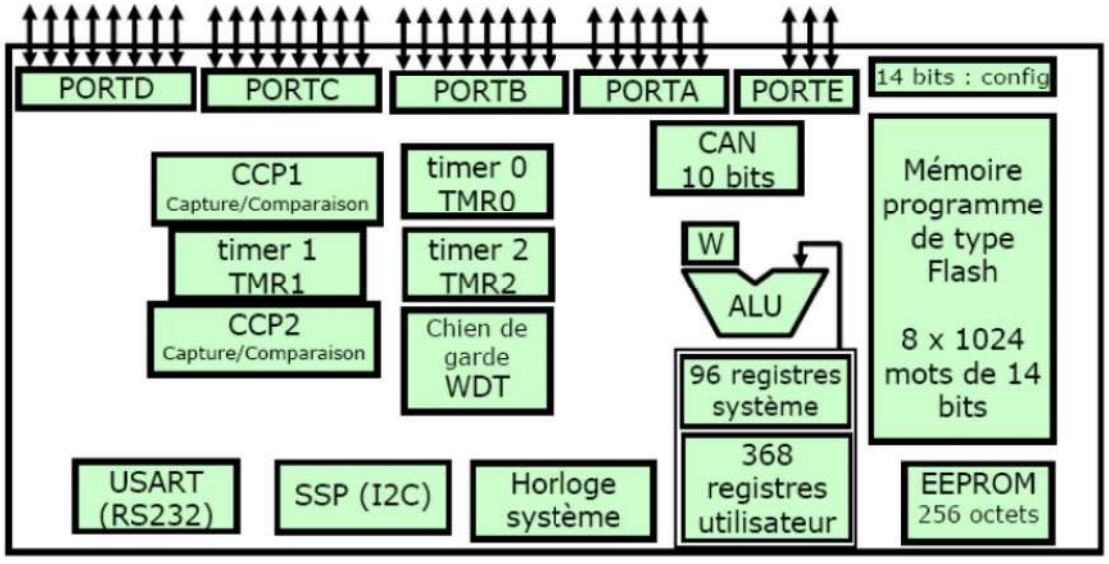


Fig. 7.3 : Architecture interne du PIC16F877.

Les microcontrôleurs PIC sont encapsulés dans un boîtier *Dual In Line* (*DIL*). Une marque, sous la forme d'un point ou d'une encoche, sert de repère pour la broche 1, à partir de laquelle on se déplace vers la droite pour accéder aux autres broches. Le parcours autour du circuit s'effectue dans le sens trigonométrique.

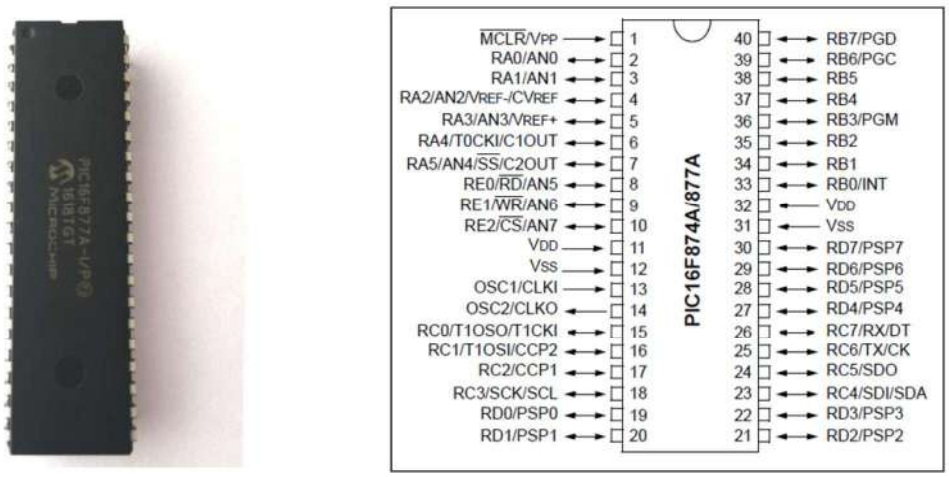


Fig. 7.4 : Brochage du PIC16F877.

3.3. Éléments de base du microcontrôleur PIC16F877

3.3.1. L'alimentation

L'alimentation du PIC16F877 est assurée par les broches **VDD** et **VSS** :

- Les broches **VDD** sont reliées à la borne positive de l'alimentation,
- Les broches **VSS** sont connectées à la masse (0 V).

Le microcontrôleur fonctionne avec une tension d'alimentation comprise entre 4 et 5.5 V. La présence de deux broches VDD et deux broches VSS permet une meilleure stabilité de l'alimentation et une réduction des parasites, particulièrement important en électrotechniques.

3.3.2. L'horloge

Le fonctionnement du PIC16F877 repose sur une **horloge système** qui cadence l'exécution des instructions du programme. Plusieurs modes d'oscillation sont disponibles :

- **Oscillateur à quartz ou résonateur céramique** : Ce mode est le plus utilisé en pratique. Il permet d'obtenir des fréquences stables allant généralement de **4 MHz à 20 MHz**, en associant le quartz à deux condensateurs de stabilisation.
- **Oscillateur RC** : La fréquence d'oscillation dépend de la tension d'alimentation **VDD** et des composants externes ***Rext*** et ***Cext***. Ce mode est économique mais moins précis.
- **Horloge externe** : Le microcontrôleur peut être cadencé par un signal d'horloge externe appliqué directement sur l'entrée prévue à cet effet.

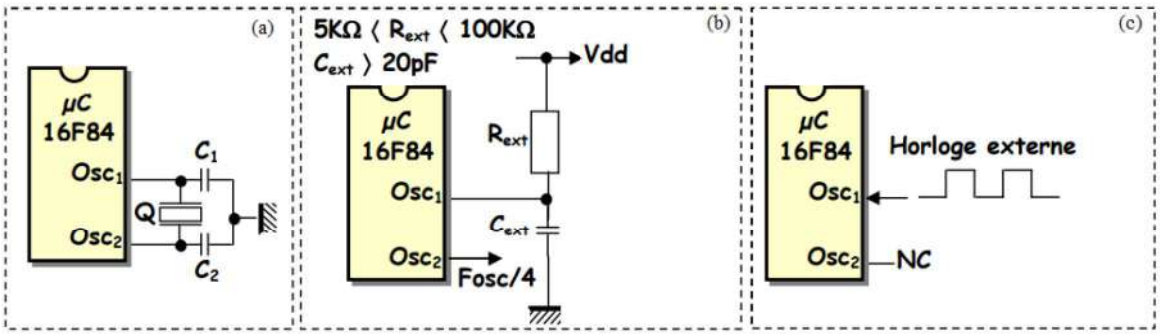


Fig. 7.5 : Les trois modes d'horloge du PIC16F877.

En fonctionnement normal, la **fréquence d'instruction** du PIC16F877 est égale à la fréquence d'oscillation divisée par 4. Par exemple, un quartz de **4 MHz** génère une fréquence d'instruction de **1 MHz**, ce qui correspond à un temps d'exécution d'une instruction égal à **1 µs**.

3.3.3. Circuit de réinitialisation (MCLR)

La broche **MCLR (*Master Clear Reset*)** permet la réinitialisation du microcontrôleur. Lorsqu'un niveau logique bas est appliqué sur cette broche, le PIC16F877 est immédiatement réinitialisé. La réinitialisation peut être réalisée :

- automatiquement à la mise sous tension,
- manuellement à l'aide d'un bouton poussoir,
- ou par un circuit externe de supervision.

Lors d'un **RESET**, l'ensemble des registres est initialisé et le **compteur programme** est positionné sur l'adresse du **vecteur de reset**, à partir de laquelle le programme commence son exécution.

3.3.4. Ports d'entrées/sorties

Le PIC16F877 dispose de **cinq ports d'entrées/sorties** permettant une interface directe avec l'environnement extérieur :

- **PORTA :** 6 broches (RA0 à RA5).
- **PORTB :** 8 broches (RB0 à RB7).
- **PORTC :** 8 broches (RC0 à RC7).
- **PORTD :** 8 broches (RD0 à RD7).
- **PORTE :** 3 broches (RE0 à RE2).

Le courant maximal en sortie est typiquement de **25 mA par broche**, avec une limite globale par port à respecter pour garantir la fiabilité du circuit.

Chacun des cinq ports du **PIC16F877** est associé à deux registres fondamentaux : le registre **PORTx**, utilisé pour la lecture et l'écriture des données, et le registre **TRISx**, destiné à la configuration directionnelle des broches (entrée ou sortie).

a- Registres TRISx

Les registres **TRISx** (*TRISA, TRISB, TRISC, TRISD et TRISE*), un par port, sont des **registres de direction** permettant de configurer chaque broche du microcontrôleur soit en **entrée**, soit en **sortie**. La configuration se fait **bit par bit** selon la règle suivante :

- **Bit à 1** : la broche correspondante est configurée en **entrée**.
- **Bit à 0** : la broche correspondante est configurée en **sortie**.

Exemple :

TRISB = 0b00001111;

Interprétation :

- ✓ RB0 à RB3 → entrée
- ✓ RB4 à RB7 → sortie

b- Registres PORTx

Les registres **PORTx** (*PORTA, PORTB, PORTC, PORTD et PORTE*) sont des registres de données permettant de **lire l'état logique des broches** correspondantes ou de **modifier leur état en écriture**. Ils assurent ainsi l'interface entre le microcontrôleur et les périphériques externes connectés aux ports d'entrée/sortie.

- Écrire dedans met la valeur en sortie (0 ou 1 sur la broche configurée en sortie).
- Lire dedans récupère l'état des broches en entrée.

Exemple :

PORTB = 0b10100000;

Interprétation :

- RB7 = 1, RB5 = 1 → niveau haut
- RB6, RB4 = 0 → niveau bas

(Seules les broches configurées en sortie sont affectées.)

3.3.5. Chien de garde (*Watchdog Timer*)

Le **chien de garde (*Watchdog Timer*)** est un compteur interne destiné à améliorer la fiabilité du système. Il doit être périodiquement réinitialisé par le programme en cours d'exécution. Si, pour une raison quelconque, le programme ne réinitialise pas le chien de garde dans le temps imparti (blocage logiciel, perturbation externe), celui-ci provoque automatiquement une **réinitialisation du microcontrôleur**.

Ce mécanisme est particulièrement important dans les applications industrielles et électrotechniques où la continuité de fonctionnement est essentielle.

3.4. Organisation de la mémoire du PIC16F877

L'architecture mémoire du PIC16F877 est conçue pour stocker à la fois le **programme exécutable** et les **données nécessaires au fonctionnement** du microcontrôleur. Comme l'ensemble des microcontrôleurs PIC de type *mid-range*, le PIC16F877 adopte une **architecture Harvard**, dans laquelle la mémoire programme et la mémoire de données sont physiquement séparées.

Cette organisation permet un **accès simultané** aux instructions et aux données au cours d'un même cycle machine, améliorant ainsi les performances du système. Le PIC16F877 dispose de **trois types principaux de mémoire** :

3.4.1. Mémoire programme (Flash)

La mémoire programme, également appelée **mémoire Flash**, est utilisée pour stocker l'ensemble des instructions du programme utilisateur. Dans le cas du PIC16F877, cette mémoire est composée de **8192 mots de 14 bits** (8K mots). Chaque mot correspond à une instruction machine et est identifié par une adresse comprise entre **0000h et 1FFFh**.

Cette mémoire est **non volatile**, ce qui signifie que son contenu est conservé même en cas de coupure d'alimentation. Elle peut être **programmée, effacée et reprogrammée** à volonté à l'aide d'un programmateur, ce qui facilite le développement et la mise à jour des applications embarquées.

3.4.2. Mémoire de données non volatile (EEPROM)

Le PIC16F877 intègre une mémoire **EEPROM de 256 octets**, accessible par logiciel, avec des adresses allant de **00h à FFh**. Chaque octet peut être lu ou écrit individuellement.
Cette mémoire est principalement utilisée pour stocker des **données permanentes ou semi-permanentes**, telles que les paramètres de configuration ou les identifiants et codes d'accès,

3.4.3. Mémoire de données volatile (RAM)

La mémoire RAM du PIC16F877 est utilisée pour le stockage temporaire des variables, des registres et des résultats intermédiaires durant l'exécution du programme. Elle est organisée en **plusieurs banques de mémoire**, ce qui permet une gestion efficace des ressources internes. La RAM du PIC16F877 est divisée en deux grandes catégories de registres :

a) Registres SFR (*Special Function Registers*)

Les **registres de fonction spéciale (SFR)** occupent les adresses basses de chaque banque mémoire. Ils sont utilisés pour :

- configurer les ports d'entrées/sorties,
- contrôler les timers,
- gérer les interruptions,
- piloter les modules internes (CCP, ADC, communications, etc.).

b) Registres GPR (*General Purpose Registers*)

Les **registres à usage général (GPR)** occupent le reste de l'espace RAM. Ils sont mis à la disposition du programmeur pour :

- stocker des variables,
- mémoriser des résultats intermédiaires,
- gérer des tableaux ou compteurs.

Le PIC16F877 dispose de **368 octets de RAM**, répartis sur plusieurs banques, ce qui permet la réalisation d'applications plus complexes que celles possibles avec les microcontrôleurs d'entrée de gamme.

Qu'est-ce qu'une banque mémoire (Bank) ?

Dans les microcontrôleurs **PIC mid-range** (comme le **PIC16F877**), la mémoire RAM n'est pas vue comme un seul bloc continu. Elle est **découpée en plusieurs banques mémoire (pages)**, appelées **Bank0, Bank1, Bank2 et Bank3**.

Pourquoi ce découpage ?

Parce que le PIC utilise un **bus d'adressage limité**, ce qui impose de sélectionner explicitement la banque mémoire active avant d'accéder à une donnée ou à un registre.
Chaque banque contient :

- des **registres SFR** (registres de contrôle),
- des **registres GPR** (registres à usage général).

La sélection de la banque active se fait à l'aide de **bits spécifiques du registre STATUS**.

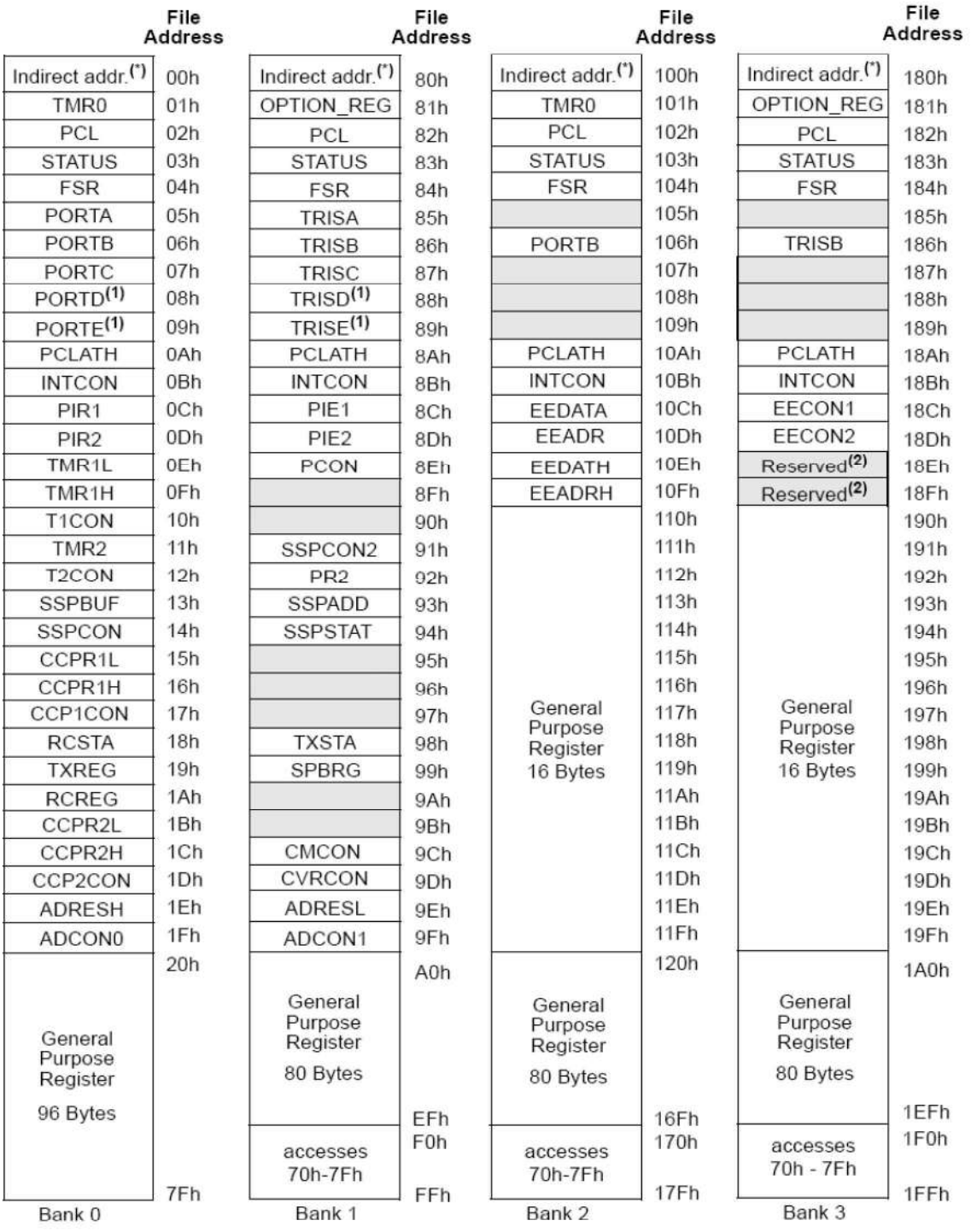


Fig. 7.6 : Organisation de la mémoire de données de type RAM dans le cas du PIC16F877.

3.5. Accès à la mémoire RAM du PIC16F877

Dans le microcontrôleur PIC16F877, la mémoire RAM totale est de **512 octets**. Cependant, le jeu d'instructions de ce microcontrôleur ne permet de coder directement que des **adresses sur 7 bits**, ce qui limite l'accès à **128 registres** à la fois.

Cette limitation pose donc une **problématique d'adressage** : comment accéder à l'ensemble des 512 octets de RAM avec des instructions capables d'adresser seulement 128 emplacements mémoire ?

- **Solution adoptée : organisation en banques mémoire**

Pour répondre à cette contrainte, le PIC16F877 adopte une **organisation de la RAM en banques mémoire**. La mémoire est ainsi découpée en **4 banques**, chacune contenant **128 octets** (Bank0 à Bank3).

- **Principe de sélection de la banque**

Pour accéder à l'ensemble des 512 octets de RAM, **9 bits sont nécessaires** ($2^9 = 512$). Pour résoudre ce problème, le PIC16F877 utilise un mécanisme de **sélection de banque**, utilisant des bits spécifiques du registre **STATUS**. Ainsi, les **7 bits d'adresse** sont complétés par **2 bits supplémentaires**, ce qui permet de former une **adresse réelle sur 9 bits** et d'accéder à l'ensemble de la RAM.

- Les **7 bits contenus dans l'instruction** adressent le registre à l'intérieur d'une banque.
- Les **2 bits manquants**, permettant de sélectionner la banque mémoire, sont fournis par le **registre STATUS**.

Bits RP1 et RP0 (registre STATUS) + 7 bits De l'instruction

Les bits **RP1** et **RP0**, correspondant respectivement aux **bits 6 et 5 du registre STATUS**, sont utilisés pour sélectionner la **banque mémoire active**. La correspondance entre les bits RP1, RP0 et les banques RAM est la suivante :

RP1	RP0	Banque RAM sélectionnée
0	0	Bank 0
0	1	Bank 1
1	0	Bank 2
1	1	Bank 3

Exemple : calcul de l'adresse réelle en RAM

Pour accéder à une case mémoire située dans la **Bank2**, à l'adresse **0x15** à l'intérieur de la banque, l'instruction ne code que cette adresse sur **7 bits** (0010101_2). La sélection de la banque est assurée par les bits **RP1 = 1** et **RP0 = 0** du registre **STATUS**.

L'adresse réelle utilisée par le PIC16F877 est obtenue en concaténant **RP1/RP0** avec les **7 bits de l'instruction**, ce qui donne :

RP1	RP0	7 bits instruction
1	0	0010101

Ainsi, les 7 bits de l'instruction sont concaténés (placés côte à côte) avec RP1/RP0 pour former l'adresse complète de 9 bits (512 octets RAM totale).

En plus de l'adressage direct présenté ci-dessus, le PIC16F877 permet également un adressage indirect via les registres FSR/INDF, qui offre plus de flexibilité pour accéder à la RAM. Dans ce cours, nous nous limiterons à l'adressage direct, et l'adressage indirect ne sera pas abordé.

3.5.1. Le rôle du registre STATUS

Le registre **STATUS** est un registre de fonction spéciale (SFR) qui joue un rôle essentiel en indiquant l'état actuel du microcontrôleur et contient notamment les **bits de sélection de banque**. Il se compose de 8 bits, chacun ayant des définitions spécifiques et des utilisations diverses dans différents contextes :

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	**RP1**	**RP0**	**TO**	**PD**	**Z**	**DC**	**C**

- **C (*Carry*) :** Bit indiquant la retenu.
- **DC (*Digit Carry*) :** Indique si une retenue a été générée lors d'une opération arithmétique.
- **Z (*Zero*) :** Indique si le résultat d'une opération est zéro.
- **PD (*Power Down*) :** Signale que le microcontrôleur est en mode de mise en veille.
- **TO (*Time Out*) :** Indique un dépassement de temps dans des conditions spécifiques (*Watchdog Timer*).
- **RP0 (*Register Page Select*) :** Sélection de banque RAM (adressage direct).
- **RP1 (*Register Page Select*) :** Sélection de banque RAM (adressage direct).
- **IRP (Indirect Register Pointer) :** Sélection de banque pour l'adressage indirect.

Programmation du PIC 16F877 – Assembleur et C

Introduction

Le PIC16F877 est un microcontrôleur 8 bits de la famille *mid-range* de *Microchip*, largement utilisé dans les applications électrotechniques pour le pilotage de moteurs, capteurs, lampes et autres systèmes embarqués. Ce chapitre a pour objectif de permettre la compréhension du jeu d'instructions du PIC16F877, d'apprendre à développer des programmes en assembleur (langage proche du matériel) et en C, et d'appliquer ces connaissances à des applications électrotechniques simples.

Le jeu d'instructions d'un microcontrôleur est un ensemble spécifique d'instructions que le microcontrôleur peut exécuter. Une instruction est définie par son code opératoire, une valeur numérique binaire difficile à manipuler directement par l'être humain. Pour rendre ces instructions plus compréhensibles pour les programmeurs, on utilise des mnémoniques, c'est-à-dire une notation symbolique, afin de faciliter l'écriture des programmes en langage assembleur.

En fonction du nombre d'instructions reconnues par le microcontrôleur, on peut les classer en deux grandes catégories :

- **Les architectures CISC (*Complex Instruction Set Computer*) :** elles comportent un grand nombre d'instructions complexes, généralement entre 75 et 150 instructions.
- **Les architectures RISC (*Reduced Instruction Set Computer*) :** elles utilisent un jeu d'instructions réduit, simple et optimisé, généralement entre 30 et 35 instructions.

1. Généralités et notions communes

1.1. Étapes de développement d'un programme pour le PIC16F877

Le développement d'un programme pour le PIC16F877 suit une démarche structurée, commune à la programmation en assembleur et en langage C, permettant de passer d'un besoin électrotechnique à une application fonctionnelle.

La première étape consiste à **analyser l'application** électrotechnique. Il s'agit d'identifier les entrées et les sorties du système, les capteurs et actionneurs à piloter, ainsi que les contraintes de fonctionnement telles que les seuils, les temporisations et les exigences de sécurité. Cette analyse permet de choisir les ports et les périphériques internes à utiliser.

La **conception du programme** se fait ensuite à l'aide d'un algorithme ou d'un organigramme décrivant la logique de fonctionnement, les conditions de décision et les boucles. Cette étape facilite la compréhension globale du programme avant le codage.

Avant l'exécution, une phase d'**initialisation** est nécessaire. Elle comprend la configuration des ports d'entrées/sorties via les registres TRISx, l'initialisation des sorties avec les registres PORTx, ainsi que la configuration des périphériques internes tels que les timers, le CAN ou les interruptions.

Le programme principal s'**exécute** dans une boucle répétitive où le microcontrôleur lit les entrées, traite les données et commande les sorties. Enfin, des **tests et opérations de débogage** sont réalisés afin de vérifier le bon fonctionnement du programme, corriger les erreurs éventuelles et valider l'application sur le matériel réel.

1.2. Ressources informatiques et logicielles pour la programmation du PIC

Le travail d'un microcontrôleur est d'exécuter des programmes. Un programme est une suite d'instructions, généralement écrites ligne par ligne. La complexité de ces instructions varie en fonction du langage de programmation utilisé. Par exemple, dans un langage de bas niveau comme l'assembleur, une instruction accomplit une tâche élémentaire, comme une opération d'addition. En revanche, dans un langage de haut niveau tel que le C++, une instruction peut regrouper plusieurs tâches qui nécessiteraient plusieurs instructions en assembleur.

Lors de la rédaction d'un programme, on a le choix parmi une variété de langages de programmation tels que l'*Assembleur*, le *Basic/QBasic*, *le Pascal*, le *C/C++*, le *Visual Basic*, le *Visual C++*, le *Delphi*, le *Java*, et bien d'autres. Les lignes de code que nous écrivons constituent un **"programme source"** qui est généralement stocké dans un fichier texte avec une extension dépendant du langage choisi (*.pas* pour le Pascal, *.cpp* pour le *C++, etc*).

Ces programmes sources sont lisibles par les programmeurs, mais pas par le processeur lui-même. Pour que le processeur comprenne ces instructions, il est nécessaire de les traduire en langage machine, une séquence de codes binaires. Chaque langage de programmation possède son propre **compilateur** qui effectue cette traduction, transformant ainsi le programme source en un programme exécutable compréhensible par le processeur. Il est important de noter que tous les programmes exécutables ont une apparence similaire pour le processeur, quel que soit le langage de programmation utilisé.

Lorsque l'on travaille avec un microcontrôleur programmable, il est essentiel de disposer des outils informatiques appropriés, ainsi que potentiellement d'un programmateur dédié. Chaque type de microcontrôleur a son propre ensemble d'outils de développement associés, généralement regroupés dans ce que l'on appelle un environnement de développement.

Un environnement de développement intégré (IDE, pour *Integrated Development Environment*) est un logiciel qui propose un ensemble intégré d'outils et de fonctionnalités pour simplifier le processus de développement logiciel. Ce cadre complet permet la création, la compilation, le débogage et le téléchargement de programmes sur le microcontrôleur cible. Ainsi, un IDE constitue une plateforme unifiée où les développeurs peuvent concevoir, tester et optimiser leurs applications pour le microcontrôleur spécifique qu'ils utilisent.



Fig. 8.1 : Outils de développement pour un PIC.

Un IDE typique pour la programmation de microcontrôleurs inclut généralement les éléments suivants :

- ✓ **Editeur de Code Source :** Permet la saisie, l'édition et la mise en forme du code source.
- ✓ **Compilateur :** Transforme le code source écrit par le programmeur en langage machine compréhensible par le microcontrôleur.
- ✓ **Débogueur :** Facilite la détection, la localisation et la correction des erreurs dans le code source.
- ✓ **Outil de *Téléversement* :** Permet de transférer (*téléverser*) le programme compilé vers le microcontrôleur, facilitant ainsi le processus de chargement du logiciel sur le matériel cible.

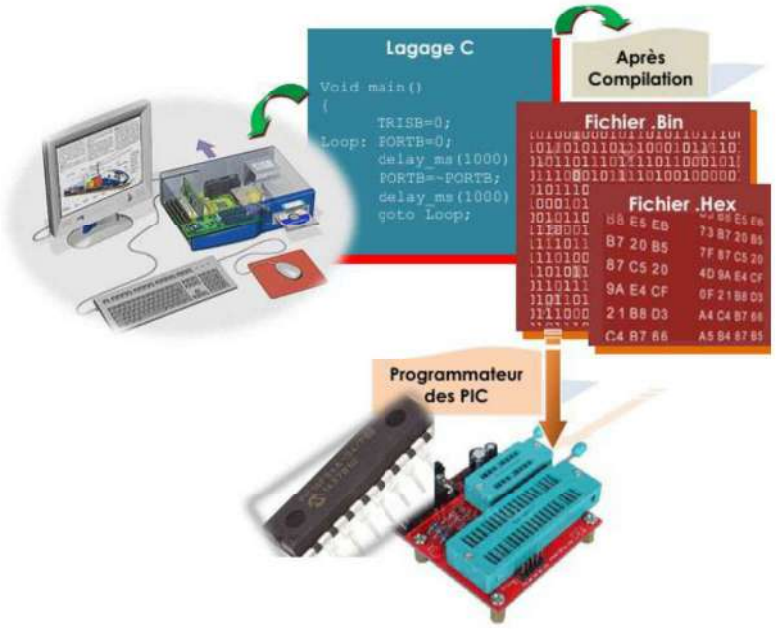


Fig. 8.2 : Etape nécessaire à la programmation d'un microcontrôleur PIC.

1.2.1. l'IDE MPLAB pour la programmation des PIC

MPLAB est un IDE développé par *Microchip Technology*, conçu spécifiquement pour la programmation et le développement de logiciels destinés aux microcontrôleurs PIC. Doté d'un éditeur de texte avec coloration syntaxique, d'un compilateur, d'un débogueur avancé, et d'outils de téléversement, il offre une solution complète pour la création, le débogage et la gestion de projets PIC. Avec un simulateur pour tester le code sans matériel physique, une intégration matérielle possible, des outils d'analyse de performances, et une interface utilisateur conviviale, MPLAB facilite le développement en fournissant des fonctionnalités variées telles que des bibliothèques prêtes à l'emploi et des exemples de code pour accélérer le processus de programmation.

MPLAB supporte plusieurs langages de programmation, notamment l'assembleur et le langage C, offrant ainsi une flexibilité aux développeurs. Il prend également en charge une large gamme de microcontrôleurs PIC, permettant aux utilisateurs de travailler avec différents modèles de la famille PIC.

1.3. Ressources matérielles nécessaires à la programmation du PIC

La mise en œuvre d'un programme sur le PIC16F877 nécessite des éléments matériels et logiciels de base. Le microcontrôleur constitue le cœur du système et assure le traitement des données, la gestion des entrées/sorties et le pilotage des périphériques internes.

Un ordinateur équipé d'un environnement de développement comme MPLAB X est indispensable pour écrire, compiler et charger le programme à l'aide d'un programmateur (PICkit, ICD). Le microcontrôleur doit également être alimenté correctement (environ 5 V) et disposer d'un système d'horloge, généralement assuré par un quartz externe, afin de garantir un fonctionnement stable et fiable.

2. Programmation en langage assembleur des microcontrôleurs PIC (PIC16F877)

La programmation en assembleur du PIC16F877 permet de contrôler directement le microcontrôleur via son jeu d'instructions RISC 8 bits. Chaque instruction correspond à une opération machine codée sur 14 bits, exécutée rapidement (1 cycle d'instruction = 4 cycles d'horloge). Cette approche offre un contrôle précis du matériel et des temporisations, mais demande une bonne connaissance de l'architecture interne.

La structure d'une instruction en langage assembleur peut être décomposée comme suit :

[Étiquette :] Mnémonique [Opérandes] [; Commentaire]

- **L'étiquette** est optionnelle et précède généralement l'instruction, suivie d'un deux-points (:). Elle sert à marquer une adresse mémoire ou un point de saut (une référence) dans le programme. Exemple : *"start:"*.
- **Le mnémonique** (appelé aussi *Opcode*) est l'opération à effectuer. C'est généralement le nom de l'instruction, comme "MOV" pour déplacer des données ou "ADD" pour additionner.
- **Les opérandes** sont facultatifs et spécifient les données sur lesquelles l'instruction doit agir. Ils peuvent être des registres, des valeurs immédiates, des adresses mémoire, etc. Le nombre d'opérandes peut varier en fonction de l'instruction. Certaines n'en nécessitent pas (par exemple, les instructions NOP), tandis que d'autres en utilisent un ou plusieurs.
- **Le commentaire** est facultatif et permet d'expliquer brièvement l'objectif ou le comportement de l'instruction. Il est précédé par un point-virgule (;).

2.1. Le jeu d'instructions du PIC 16F877

Le jeu d'instructions en assembleur du PIC16F877 se compose de 35 instructions qui permettent d'accomplir diverses tâches. Afin de faciliter la compréhension des fonctions de chaque instruction, la notation suivante est adoptée pour les données et les adresses manipulées par ces instructions :

- *W* représente l'accumulateur (Working register), utilisé dans la majorité des opérations arithmétiques et logiques.

- ***f*** représente un registre (*file Register*)
- ***b*** représente un numéro de bit, où 0 correspond toujours au bit de poids faible (le plus à droite dans le registre)
- ***k*** représente une donnée, également appelée valeur immédiate ou littéral (*Literal value*).

Les principales instructions du PIC sont présentées ci-après à travers leurs mnémoniques, leurs opérandes, leur syntaxe et une description claire de l'opération réalisée, accompagnée d'exemples d'utilisation.

2.1.1. Instructions de transfert de données

Elles permettent de déplacer des données entre registres, mémoire et accumulateur **W**.

1- MOVF (*Move F*) :

Syntaxe : ***MOVF f,d***

Opération : met la variable contenue dans le registre *f* dans *W* si d=0 ou dans *f* si d=1.

2- MOVLW (*Move Litteral to W*) :

Syntaxe : ***MOVLW k***

Opération : met la variable *k* dans *W*.

3- MOVWF (*Move W to f*) :

Syntaxe : ***MOVWF f***

Opération : met la variable contenue dans *W* à l'adresse *f*.

2.1.2. Instructions arithmétiques

Pour faire des calculs simples.

4- ADDLW (*add litteral to W*) :

Syntaxe : ***ADDLW k***

Opération : ajoute la valeur d'une constante *k* au registre *W* et on place le résultat dans W.

5- ADDWF (*add W to f*) :

Syntaxe : ***ADDWF f,d***

Opération : additionne le contenu du registre *f* avec *W*. Résultat dans f si $d = 1$ ou dans *W* si $d = 0$.

6- SUBLW (*substract litteral from W*) :

Syntaxe : ***SUBLW k***

Opération : soustrait la valeur d'une constante *k* du registre *W* et le résultat est placé dans *W*.

7- SUBWF (*substract W from f*) :

Syntaxe : ***SUBWF f,d***

Opération : soustrait le contenu du registre *W* du contenue du registre *f*. Le résultat est dans *f* si $d - 1$ ou dans *W* si $d - 0$.

8- INCF (*Increment f*) :

Syntaxe : ***INCF f,d***

Opération : incrémente de +1 le contenu du registre *f*. Le résultat est dans *f* si *d*=1 ou dans *W* si *d*=0.

9- DECF (*Decrement f*) :

Syntaxe : ***DECF f,d***

Opération : décrémente de – 1 le contenu du registre *f*. Le résultat est dans *f* si d=1 ou dans *W* si *d*=0.

10- DECFSZ (*Decrement f, skip if zero*) :

Syntaxe : ***DECFSZ f,d***

Opération : décrémente de – 1 le contenu du registre *f*. Le résultat est dans *f* si *d*=1 ou dans *W* si *d*=0. Si le résultat de l'opération vaut 0, alors l'instruction suivante est ignorée (instruction NOP exécuté à la place).

2.1.3. Instructions logiques

Pour manipuler les bits.

11- ANDLW (*and litteral with W*) :

Syntaxe : ***ANDLW k***

Opération : effectue un *ET* logique entre le contenu de *W* et une constante *k*. Le résultat est stocké dans *W*.

12- ANDWF (and *W* with *f*) :

Syntaxe : ***ANDWF f,d***

Opération : effectue un *ET* logique entre une variable à l'adresse *f* et le contenu de *W*. Le résultat est dans *f* si *d* = 1 ou dans *W* si d = 0.

13- COMF (*Complement f*) :

Syntaxe : ***COMF f,d***

Opération : complément à 1 de *f*. Résultat est dans *f* si *d*=1 ou dans *W* si *d*=0.

14- IORWF (Inclusive OR W with f) :

Syntaxe : ***IORWF f, d***

Opération : effectue un OU logique entre le contenu du registre *W* et celui du registre *f*. Le résultat est placé dans f si d = 1, ou dans W si d = 0.

15- XORWF (Exclusive OR W with f) :

Syntaxe : ***XORWF f, d***

Opération : effectue un OU exclusif entre le contenu du registre *W* et celui du registre *f*. Le résultat est placé dans f si d = 1, ou dans W si d = 0.

☞ ***OU exclusif (XOR) :*** le bit résultat vaut 1 si les deux bits sont différents ;

2.1.4. Instructions sur les bits

Indispensables pour les ports d'E/S.

16- BCF (*Bit clear f*) :

Syntaxe : ***BCF f,b***

Opération : met à zéro le bit *b* du registre *f*.

17- BSF (*Bit set f*) :

Syntaxe : ***BCF f,b***

Opération : met à un le bit *b* du registre *f*.

18- BTFS (*Bit test, Skip if clear*) :

Syntaxe : ***BTFSC f,b***

Opération : test le bit *b* de *f*. Si ce bit vaut 0 alors l'instruction suivante est ignorée (instruction NOP exécuté à la place).

19- BTFSS (*Bit test, Skip if Set*) :

Syntaxe : ***BTFSS f,b***

Opération : test le bit *b* de *f*. Si ce bit vaut 1 alors l'instruction suivante est ignorée (instruction NOP exécuté à la place).

2.1.5. Autres instructions (Importantes)

20- CLRF (*Clear f*) :

Syntaxe : ***CLRF f***

Opération : met à 0 le contenu du registre *f*.

21- CLRW (Clear W) :

Syntaxe : ***CLRW***

Opération : met à 0 le contenu du registre de travail *W*.

22- GOTO (*go to, branchment incoditionelle*) :

Syntaxe : ***GOTO Label***

Opération : effectue un saut dans le programme vers l'adresse pointée par le Label.

23- NOP (*No operation*) :

Syntaxe : ***NOP***

Opération : aucune opération effectué (1 cycle machine exécuté).

Exemple de programme en assembleur du PIC16F877 :

Cprogramme décrit l'implémentation d'un système avec un interrupteur connecté à RB0 en tant qu'entrée et une LED à RB1 en tant que sortie sur le port B d'un PIC16F877. Lorsque l'interrupteur est activé, la LED s'allume, et lorsqu'il est désactivé, la LED s'éteint.

```
BSF     STATUS, RP0          ; Sélectionne la banque 1 (mise à 1 de RP0)
MOVLW   B'00000001'          ; Charge 00000001 dans le registre W
        ; Chaque bit correspond à une broche du PORTB (bit0 → RB0, bit1 → RB1, ...)

MOVWF   TRISB                ; Copie la valeur de W dans le registre TRISB pour
                             ; configurer la direction des broches (RB0 entrée et RB1 sortie
BCF     STATUS, RP0          ; Revient à la banque 0 de la RAM

TEST BTFSC  PORTB, 0         ; Teste si RB0 est à l'état 0 (interrupteur désactivé)
 GOTO   OFF                  ; Si RB0 est à l'état 0, sauter à l'étiquette OFF et
                               ignore les deux instructions suivantes
 BSF    PORTB, 1             ; Allume la LED (met le bit 1 de PORTB à 1)
 GOTO   TEST                 ; Saute à l'étiquette TEST

OFF BCF    PORTB, 1          ; Éteint la LED (met le bit 1 de PORTB à 0)
 GOTO   TEST                 ; Saute à l'étiquette TEST
 END                         ; Fin du programme
```

Il est important de mentionner que le PIC16F877 présente des limitations dans ses capacités d'instructions en assembleur, notamment en l'absence d'instructions matérielles dédiées pour des opérations telles que la multiplication, la division, les calculs sur les nombres à virgule flottante, et les manipulations complexes de chaînes de caractères. Par conséquent, l'exécution de ces opérations nécessite souvent la mise en œuvre de séquences d'instructions spécifiques, souvent impliquant des opérations d'addition et des boucles pour effectuer des calculs plus avancés.

3. Programmation en langage C des microcontrôleurs PIC (PIC 16F877)

La programmation en C des microcontrôleurs PIC consiste à utiliser le langage C pour développer des applications embarquées telles que la lecture de capteurs, le pilotage de moteurs, la commande de LEDs, la communication série ou la gestion du temps. Le langage C est un langage de haut niveau qui permet d'écrire des programmes plus clairs, plus structurés et plus faciles à comprendre que le langage assembleur.

Dans un PIC, le programme écrit en C n'est pas exécuté directement : il est d'abord traduit en langage machine par un compilateur (comme CCS C ou XC8), puis chargé dans la mémoire du microcontrôleur. Cela permet au programmeur de se concentrer sur la logique du programme sans manipuler directement les codes binaires ou les registres internes, sauf lorsque cela est nécessaire.

Par rapport à l'assembleur, le langage C facilite le développement des applications complexes grâce à l'utilisation de variables, de conditions, de boucles et de fonctions. Il réduit le temps de programmation, améliore la lisibilité du code et permet une meilleure maintenance. L'assembleur reste toutefois utile lorsque l'on souhaite un contrôle très précis du matériel ou une optimisation maximale des ressources.

En pratique, la programmation en C représente un bon compromis entre simplicité, performance et efficacité, ce qui en fait le langage le plus utilisé pour la programmation des microcontrôleurs PIC dans l'enseignement et les applications industrielles.

3.1. Structure générale d'un programme en C pour PIC

Un programme en C pour microcontrôleur PIC suit une structure bien définie qui permet d'organiser le code, configurer le matériel, déclarer les variables et écrire la logique de l'application. Voici les parties principales :

3.1.1. Directives de précompilation

Les directives de précompilation sont des instructions spéciales pour le compilateur, elles ne font pas partie du code exécutable. Elles permettent de :

- inclure des fichiers d'en-tête,
- définir la configuration matérielle,
- déclarer des constantes ou paramètres globaux.

Syntaxe et exemples :

```
#include <16F877.h>          // Fichier d'en-tête spécifique au PIC16F877
```

#include <16F877.h> sert à inclure les fichiers d'en-tête, qui contiennent les définitions des registres, des bits, des constantes et des fonctions spécifiques au microcontrôleur choisi. Ces fichiers permettent au compilateur de connaître les adresses des ports, des registres spéciaux et des périphériques, afin de pouvoir les utiliser directement dans le programme.

```
#use delay(clock=4000000)      // Définition de la fréquence du quartz (4 MHz)
```

#use delay informe le compilateur de la fréquence du quartz pour les temporisations (*delay ms(), delay us()*).

Remarque : Ces lignes doivent toujours être placées avant la fonction *main()*.

3.1.2. Bits de configuration du microcontrôleur

Les bits de configuration (fuses) définissent certains paramètres matériels essentiels du PIC :

- Type d'oscillateur (quartz, RC, interne),
- Activation ou désactivation du *watchdog timer*,
- Activation du Power-up Timer, qui est temporisateur interne qui maintient le microcontrôleur en état de reset pendant un court délai après la mise sous tension.
- Mode de programmation (Low Voltage (5 V) ou High Voltage (~12 V)).

Syntaxe et exemple :

#fuses XT,NOWDT,NOPUT,NOLVP

- **XT :** quartz externe
- **NOWDT :** watchdog timer désactivé
- **NOPUT :** Power-up timer désactivé
- **NOLVP :** programmation Low Voltage désactivée

Ces fuses doivent également être placés avant la fonction *main()*.

3.1.3. Déclaration des variables et constantes

Dans un programme en langage C pour PIC, les variables sont utilisées pour stocker temporairement des données, tandis que les constantes permettent de définir des valeurs fixes qui ne changent pas pendant l'exécution du programme. Une bonne gestion des variables et constantes est essentielle pour écrire un code clair, efficace et sans erreur.

a) Types de variables

Le langage C met à disposition plusieurs **types de variables**, permettant de stocker différents types d'informations selon les besoins de l'application (états logiques, comptage, mesures analogiques, etc.).

Type char (8 bits) : Le type « char » est un type de données sur 8 bits qui peut représenter soit un seul caractère ASCII, soit une petite valeur entière. Il est souvent utilisé pour stocker des états, des options ou des données simples.

```
char etatLED = 1;          // 1 = LED allumée, 0 = LED éteinte
char lettre = 'A';         // caractère ASCII
```

Type int (16 bits) : Le type « int » permet de stocker des nombres entiers plus grands. Il est souvent utilisé pour les compteurs, les temporisations ou les valeurs issues d'un capteur.

```
int compteur = 250;        // Compteur ou valeur mesurée
```

Type long (32 bits) : Le type « long » est réservé aux très grandes valeurs numériques, par exemple pour des calculs sur de longues durées ou des comptages importants.

```
long temps_total = 100000;        // Temps cumulé ou grand compteur
```

Type float (32 bits) : Le type « float » permet de représenter des valeurs décimales. Il est très utilisé pour les grandeurs physiques comme la tension, le courant ou la température.

```
float tension = 3.25 ;            // Tension mesurée par l'ADC
```

b) Déclaration et initialisation des variables

La syntaxe générale pour déclarer une variable est :

```
type nom_variable;
```

Exemple :

```
int compteur;          // Variable entière pour compter les tours
char etat_LED;         // Variable pour stocker l'état d'une LED
float tension;         // Variable pour stocker une mesure analogique
```

On peut aussi initialiser la variable dès sa déclaration :

```
int compteur = 0;      // Initialise compteur à 0
char etat_LED = 1;     // LED allumée
float tension = 3.3;   // Tension initiale
```

c) Déclaration des constantes

Les constantes sont définies avec le mot-clé « **const** » ou avec « **#define** » pour des valeurs fixes.

Avec const :

```
const int DELAI_MS = 100;          // Durée fixe d'un délai en millisecondes
const char PIN_LED = PIN_D0;       // Broche fixe pour la LED
```

Avec #define :

```
#define V_MAX 5.0                  // Définition d'une tension maximale
#define N_LED 8                    // Nombre de LED
```

Remarque : « #define » crée une constante avant la compilation (préprocesseur), tandis que « const » crée une variable en lecture seule gérée par le compilateur.

3.1.4. Spécificités importantes pour les PIC

1. Les variables de type *int* et char sont les plus utilisées pour limiter l'usage de la RAM (le PIC 16F877 n'a que 368 octets de RAM pour les variables).

2. Les *floats* consomment plus de mémoire et de cycles CPU pour les calculs, donc à utiliser uniquement si nécessaire (mesures analogiques, calculs de tension ou courant).
3. Les constantes permettent d'éviter des erreurs et de rendre le code plus lisible, par exemple pour les broches ou les seuils.
4. Les variables peuvent être **globales** (déclarées avant *main()*) ou **locales** (déclarées à l'intérieur de *main()* ou d'une fonction). Les locales sont effacées à la sortie de la fonction, tandis que les globales restent en mémoire tout le programme.

3.1.5. Fonctions principales en langage C pour PIC

En langage C pour PIC (avec le compilateur CCS), de nombreuses fonctions facilitent la programmation et le contrôle des périphériques et du microcontrôleur. Elles permettent de gérer les entrées/sorties, les temporisations, la lecture analogique, la communication série, etc.

1. Fonctions de gestion des entrées/sorties numériques

Le compilateur C pour PIC met à disposition plusieurs fonctions simples pour commander et lire l'état des broches du microcontrôleur.

La fonction *output_high(pin)* permet de placer une broche donnée à l'état logique haut (1). Elle est généralement utilisée pour allumer un composant externe comme une LED, un relais ou un moteur.

Exemple :

```
output_high(PIN_D0);          // Met la broche D0 à 1
```

La fonction *output_low(pin)* met la broche spécifiée à l'état logique bas (0), ce qui permet d'éteindre un dispositif connecté.

Exemple :

```
output_low(PIN_D0);           // Met la broche D0 à 0
```

La fonction *output_toggle(pin)* inverse automatiquement l'état actuel de la broche. Si elle était à 1, elle passe à 0, et inversement. Cette fonction est très pratique pour faire clignoter une LED.

Exemple :

```
output_toggle(PIN_D0);        // Inverse l'état de la broche D0
```

La fonction *input(pin)* permet de lire l'état logique d'une broche configurée en entrée. Elle retourne 1 si la broche est à l'état haut et 0 si elle est à l'état bas.

Exemple :

```
int etat = input(PIN_B0);     // Lecture de l'état d'un bouton placé sur B0
```

2. Fonctions de temporisation

Les temporisations logicielles sont essentielles pour ralentir l'exécution du programme ou créer des délais visibles.

La fonction *delay_ms(ms)* provoque une pause de la durée spécifiée en millisecondes. Elle est souvent utilisée pour faire ou temporiser une action (faire clignoter une LED par exemple).

Exemple :

```
delay_ms(500);          // Pause de 500 ms (0,5 seconde)
```

La fonction *delay_us(us)* introduit une temporisation en microsecondes. Elle est utilisée dans les applications nécessitant des délais très courts et précis.

Exemple :

```
delay_us(10);           // Pause de 10 microsecondes
```

Ces fonctions sont largement utilisées pour allumer ou éteindre des sorties, générer des signaux temporisés ou synchroniser des actions.

3. Fonctions liées aux *timers*

Les timers internes du PIC permettent de mesurer le temps ou de générer des interruptions périodiques.

La fonction *set_timer0(val)* initialise la valeur de départ du Timer0.

Exemple :

```
set_timer0(6);          // Initialise le compteur du Timer0 à 6 pour ajuster
                        //  le délai avant le prochain débordement
```

La fonction *set_timer1(val)* permet d'initialiser le Timer1 avec une valeur donnée.

Exemple :

```
set_timer1(3036);       // Initialise le compteur du Timer1 à 3036 pour obtenir
                        //  une période précise (ex: 50ms selon la fréquence)
```

La fonction *setup_timer_0(mode)* sert à configurer le mode de fonctionnement du Timer0, notamment la source d'horloge et le prescaler.

Exemple :

```
setup_timer_0(T0_INTERNAL | T0_DIV_256);  // Configure le Timer0 : utilise l'horloge
                                          // l'horloge interne du PIC et divise la fréquence par 256 (Prescaler)
```

De la même manière, ***setup_timer_1(mode)*** configure le Timer1.

Exemple :

```
setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);   // Configure le Timer1 : utilise
                                            // l'horloge interne et divise la fréquence par 8 pour un comptage plus rapide
```

4. Fonctions analogiques (ADC)

Le PIC16F877 dispose d'un convertisseur analogique–numérique (ADC) permettant de lire des tensions analogiques.

La fonction *setup_adc(mode)* configure le module ADC, notamment la fréquence de conversion.

Exemple :

```
tup_adc(ADC_CLOCK_DIV_32);      // on choisit une horloge de conversion égale à la
                                // fréquence du microcontrôleur divisée par 32.
```

La fonction *set_adc_channel(ch)* permet de sélectionner le canal analogique à lire (RA0 à RA5).

Exemple :

```
set_adc_channel(0);             // Sélection de l'entrée RA0
```

La fonction *read_adc()* lance la conversion et retourne la valeur numérique correspondant à la tension analogique appliquée.

Exemple :

```
int valeur = read_adc();    // Lance la conversion analogique-numérique et stocke la
                            // valeur numérique obtenue dans la variable "valeur"
```

Le PIC16F877 peut ainsi mesurer une tension comprise entre 0 et 5 V et la convertir en une valeur numérique sur 10 bits.

5. Fonctions de communication série

La communication série permet au microcontrôleur d'échanger des données avec un ordinateur ou un autre système.

La directive #use rs232(...) configure la liaison série en définissant la vitesse de communication et les broches utilisées. La vitesse de communication correspond au nombre de bits transmis par seconde au cours de la communication série.

Exemple :

```
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)    // Configure la communication
// Série à 9600 bauds, avec transmission sur la broche C6 et réception sur la broche C7
```

La fonction *printf()* permet d'envoyer des messages ou des valeurs formatées vers le terminal série. Elle est très utilisée pour l'affichage et le débogage.

Exemple :

```
printf("Valeur ADC = %d\n\r", valeur);
```

La fonction *scanf()* permet de recevoir des données depuis le port série.

Exemple :

```
scanf("%d", &valeur);
```

Les symboles de format utilisés dans printf() et scanf() permettent de préciser le type de donnée à afficher :

Format	Signification
%d	entier
%lu	entier long
%f	float
%c	caractère
%s	chaîne de caractères
%x	%x

Exemple sur les fonctions printf() et scanf() : Ce programme permet de réaliser une communication série entre le microcontrôleur et un ordinateur. Le microcontrôleur envoie d'abord un message demandant à l'utilisateur d'entrer un nombre, puis il lit ce nombre via la liaison série grâce à la fonction *scanf()*. Enfin, il renvoie le nombre reçu en utilisant la fonction *printf()* pour l'afficher sur le terminal série.

```
void main()
{
  int nombre;

  printf("Entrer un nombre : \r\n");
  // \r\n permet de faire un retour à la ligne sur le terminal série

  scanf("%d", &nombre);
  // %d indique que scanf doit lire un nombre entier

  printf("Vous avez tape : %d\r\n", nombre);
  // %d permet d'afficher la valeur de la variable nombre
  // \r\n ajoute un retour à la ligne après l'affichage

  while(1);
}
```

3.1.6. Fonction principale main()

La fonction main() est le cœur du programme. Elle contient :

1. Initialisation des ports et périphériques,
2. Boucle principale pour exécuter la logique en continu,
3. Gestion des entrées/sorties et temporisations.

Syntaxe :

```
void main() {
                    // 1. Initialisation des ports et périphériques
  // set_tris_x() pour configurer les ports
  // initialisation des périphériques (ADC, PWM, timers, UART, etc.)

                    // 2. Boucle principale
  while(TRUE) {

                    // 3. Gestion des entrées/sorties
    // Lecture des entrées : input(PIN_x)
    // Commande des sorties : output_high(PIN_x) / output_low(PIN_x)

    // Temporisations : delay_ms(), delay_us()

    // Autres traitements ou fonctions
  }
}
```

3.2. Gestion des ports d'entrées/sorties du PIC16F877

Le microcontrôleur PIC16F877 dispose de plusieurs ports d'entrées/sorties numériques permettant d'interfacer des composants externes tels que des boutons poussoirs, des LEDs, des relais ou des capteurs logiques. La gestion de ces ports repose essentiellement sur deux types de registres : les registres TRISx et PORTx.

3.2.1. Registres TRISx et PORTx

Chaque port du PIC16F877 est associé à deux registres principaux :

- **TRISx :** registre de direction du port
- **PORTx :** registre de données du port

Où *x* peut être A, B, C, D ou E.

Le registre TRISx permet de définir le rôle de chaque broche :

- **bit à 1** → broche configurée en entrée
- **bit à 0** → broche configurée en sortie

Le registre PORTx permet :

- de lire l'état des broches configurées en entrée
- de commander l'état logique des broches configurées en sortie

3.2.2. Configuration des broches en entrée et en sortie

Avant toute utilisation d'un port, il est indispensable de configurer sa direction à l'aide du registre TRISx.

Exemple :

❖ Configurer toutes les broches du PORTB en sortie :

```
set_tris_b(0x00);          // 0000 0000 → toutes les broches en sortie
```

❖ Configurer toutes les broches du PORTA en entrée :

```
set_tris_a(0xFF);          // 1111 1111 → toutes les broches en entrée
```

3.2.3. Lecture des entrées numériques

La lecture d'une entrée numérique consiste à tester l'état logique (0 ou 1) d'une broche configurée en entrée.

Exemple : lecture de l'état d'un bouton connecté à RB0 :

```
if (input(PIN_B0)) {
  // le bouton est appuyé
}
```

- ***input(PIN_B0)*** retourne 1 si la broche est à l'état haut
- ***input(PIN_B0)*** retourne 0 si la broche est à l'état bas

3.2.4. Commande des sorties numériques

La commande d'une sortie numérique permet de placer une broche à l'état haut (1) ou bas (0).

Exemple :

❖ **allumer une LED connectée à RD0 :**

```
output_high(PIN_D0);          // RD0 = 1
```

❖ **éteindre la LED connectée à RD0 ::**

```
output_low(PIN_D0);           // RD0 = 0
```

Exemples d'applications : Commande d'une LED par un bouton

Dans cette exemple le microcontrôleur PIC16F877 lit en permanence l'état d'un bouton connecté à la broche RB0. Selon l'état du bouton, il allume ou éteint une LED connectée à la broche RD0. Le programme utilise une boucle infinie pour que la surveillance et la commande se fassent en continu.

```
#include <16F877.h>          // Inclusion du fichier d'en-tête spécifique au PIC16F877
#fuses XT,NOWDT,NOPUT,NOLVP          // Configuration des fusibles
#use delay(clock=4000000)          // Définition de la fréquence du quartz : 4 MHz

void main() {
                                    // Configuration des ports
   set_tris_b(0xFF);                // PORTB en entrée (1111 1111)
   set_tris_d(0x00);                // PORTD en sortie (0000 0000)

                                    // Boucle principale infinie
   while(TRUE) {
      if (input(PIN_B0)) {          // Lecture de la broche RB0
         output_high(PIN_D0);       // Si le bouton est appuyé, allumer la LED RD0
      } else {
         output_low(PIN_D0);        // Sinon, éteindre la LED RD0
      }
   }
}
```

La boucle while(TRUE) crée une exécution continue, ce qui permet au microcontrôleur de surveiller en permanence les entrées et de commander les sorties. Elle est utilisée dans les programmes PIC pour que la logique principale fonctionne sans interruption.

3.3. Les structures de contrôle en C

Dans la programmation en C pour microcontrôleurs PIC, la compréhension des structures de contrôle est essentielle pour écrire des programmes clairs, efficaces et fiables. Ces instructions permettent au programme de prendre des décisions selon des conditions.

3.3.1. if / else

```
if(condition) {
   // instructions exécutées si condition vraie
} else {
   // instructions exécutées si condition fausse
}
```

Exemple : Ce programme lit la valeur d'un capteur analogique grâce à la fonction *read_adc()* et la stocke dans la variable capteur. Il compare ensuite cette valeur à un seuil défini (SEUIL = 512). Si la valeur dépasse le seuil, le moteur connecté à la broche D0 est allumé *(output_high)*, sinon il est éteint *(output_low)*.

```
int capteur = read_adc();          // Lecture d'un capteur analogique
#define SEUIL 512                  // Définir une valeurs seuil à 512

if(capteur > SEUIL) {
   output_high(PIN_D0);            // Allume le moteur
} else {
   output_low(PIN_D0);             // Éteint le moteur
}
```

3.3.2. switch / case

Permet de gérer plusieurs conditions sur une seule variable.

```
switch(variable) {
   case valeur1:
      // instructions
      break;
   case valeur2:
      // instructions
      break;
   default:
      // instructions si aucune valeur ne correspond
}
```

Exemple : Ce programme utilise une structure switch pour exécuter une action en fonction de la valeur de la variable mode. Si mode = 1, la broche D0 est mise à l'état haut, si mode = 2, la broche D0 est mise à l'état bas, sinon la broche D1 est mise à l'état bas.

```
int mode = 2;
switch(mode) {
   case 1:
      output_high(PIN_D0);
      break;
   case 2:
      output_low(PIN_D0);
      break;
```

```
    default:
      output_low(PIN_D1);
  }
```

3.4. Les boucles

Les boucles permettent de répéter des instructions plusieurs fois.

3.4.1. La boucle *while*

Une boucle while permet de répéter des instructions tant qu'une condition est vraie. La condition est testée avant chaque répétition, et la boucle s'arrête dès qu'elle devient fausse.

```
while(condition) {
  // instructions répétées tant que condition vraie
}
```

Exemple : Cet exemple crée une boucle infinie qui fait clignoter une LED : la LED est allumée pendant 500 ms, puis éteinte pendant 500 ms, et ce cycle se répète sans arrêt.

```
while(TRUE) {
  output_high(PIN_D0);          // Allume la LED
  delay_ms(500);                // Temporisation
  output_low(PIN_D0);           // Éteint la LED
  delay_ms(500);
}
```

3.4.2. La boucle for

La boucle for est une structure de répétition utilisée lorsqu'on connaît à l'avance le nombre de fois qu'une action doit être exécutée. Elle regroupe en une seule ligne l'initialisation, la condition d'arrêt et l'incrémentation du compteur.

```
for(initialisation; condition; incrément) {
  // instructions répétées
}
```

L'incrémentation du compteur correspond à la façon dont la variable de boucle évolue à chaque tour. Elle peut prendre plusieurs formes selon le besoin :

- **Incrémentation simple :** i++ → le compteur augmente de 1
- **Incrémentation par pas :** i += 2 → le compteur augmente de 2
- **Décrémentation :** i-- ou i -= 1 → le compteur diminue
- **Pas plus grand :** i += 5 → utile pour sauter des valeurs

Exemple : Cet exemple utilise une boucle for pour répéter une action 10 fois : à chaque itération, l'état de la LED est inversé (allumée ↔ éteinte), puis le programme attend 200 ms. Au final, la LED clignote 10 fois, puis la boucle se termine et le programme passe à la suite.

```
for(int i=0; i<10; i++) {
  output_toggle(PIN_D0);          // Inverse l'état de la LED
  delay_ms(200);
}
```

3.4.3. La boucle *do…while*

La boucle *do...while* permet d'exécuter un bloc d'instructions au moins une fois, puis de le répéter tant que la condition reste vraie. Contrairement à la boucle while, la condition est testée à la fin de la boucle.

```
do {
  // instructions
} while(condition);
```

Exemple : Dans cet exemple, la LED connectée à la broche PIN_D0 s'allume et s'éteint avec une temporisation de 300 ms, et cette séquence est répétée 5 fois grâce au compteur x qui est incrémenté à chaque itération.

```
int x = 0;
do {
  output_high(PIN_D0);
  delay_ms(300);
  output_low(PIN_D0);
  delay_ms(300);
  x++;
} while(x < 5);                   // Répète 5 fois
```

3.5. Définition et appel de fonctions

Une fonction est un bloc de code indépendant qui réalise une tâche spécifique. Elle peut être définie avant la fonction *main()* ou dans un fichier séparé et appelée à tout moment depuis le programme principal ou d'autres fonctions. Son rôle principal est de rendre le code plus modulable, réutilisable et facile à maintenir. Elle permet également de simplifier le programme en évitant les répétitions inutiles.

Syntaxe générale :

```
type_de_retour nom_fonction(paramètres) {
  // Instructions de la fonction
  return valeur;                  // si type_de_retour != void
}
```

- **type_de_retour :** indique le type de la valeur renvoyée par la fonction (int, float, char, void si pas de valeur).
- **nom_fonction :** identifiant de la fonction.
- **paramètres :** valeurs à fournir à la fonction pour qu'elle fonctionne (optionnel).

Exemple : Ici, on présente un exemple de fonction allumer_LED() qui prend comme paramètres le numéro de la broche et la durée d'allumage en millisecondes. Cette approche améliore la lisibilité du programme en encapsulant une action répétitive, ce qui permet d'appeler facilement cette fonction à différents endroits du code sans dupliquer les instructions.

```
                    // Fonction pour allumer une LED pendant une durée définie
void allumer_LED(char pin, int duree_ms) {
   output_high(pin);                    // Allume la LED
   delay_ms(duree_ms);                  // Temporisation
   output_low(pin);                     // Éteint la LED
}
                              // Appel de la fonction dans le main
void main() {
   allumer_LED(PIN_D0, 500);            // Allume la LED sur D0 pendant 500 ms
}
```

3.6. Organisation modulaire du code

La modularité consiste à organiser un programme en plusieurs fonctions et fichiers distincts, chacun étant responsable d'une tâche précise (gestion des LEDs, des capteurs, des moteurs, etc.).

La programmation des PIC en langage C facilite fortement cette approche grâce à l'utilisation des fonctions, des fichiers d'en-tête (.h) et des fichiers source (.c). Par exemple, un projet peut être structuré de la manière suivante :

- **main.c :** programme principal et logique générale
- **led.c / led.h :** fonctions liées au pilotage des LEDs
- **moteur.c / moteur.h :** fonctions de commande du moteur
- **capteur.c / capteur.h :** fonctions de lecture des capteurs

Les fichiers d'en-tête sont inclus dans le programme principal afin de rendre les fonctions disponibles :

```
#include "led.h"
#include "moteur.h"
#include "capteur.h"
```

Le programme principal devient alors plus clair et plus lisible :

```
void main() {
  int cap = lire_capteur();          // Fonction définie dans capteur.c
  allumer_LED(PIN_D0, 200);          // Fonction définie dans led.c
  controle_moteur(cap, 512);         // Fonction définie dans moteur.c
}
```

Grâce à cette organisation modulaire, chaque partie du programme est indépendante, ce qui présente plusieurs avantages :

- une meilleure lisibilité du code,
- une réutilisation facile des fonctions dans d'autres projets,
- une maintenance simplifiée, car modifier un module n'impacte pas le reste du programme.

La modularité est donc un atout majeur de la programmation en C pour les PIC, en particulier pour les applications électrotechniques de taille moyenne à grande.

3.7. Bonnes pratiques de programmation et lisibilité du code

Pour développer des programmes fiables, clairs et faciles à maintenir en C pour les PIC, il est essentiel d'adopter de bonnes pratiques de programmation. Tout d'abord, l'utilisation de **commentaires** est fortement recommandée afin d'expliquer les étapes importantes du programme, en particulier lors de la configuration du matériel ou de la prise de décision.

Par exemple : *// Allume le moteur si la température dépasse 50 °C*

Il est important de choisir des **noms de variables explicites**, qui reflètent clairement leur rôle dans le programme. Cela améliore la compréhension du code et facilite sa modification :

```
int temperatureCapteur;          // plus clair que : int t;
```

Les **actions répétitives** doivent être regroupées dans des **fonctions**, ce qui permet d'améliorer la lisibilité et de réutiliser le code sans duplication inutile :

Une **indentation cohérente** doit être respectée dans tout le programme afin de visualiser facilement les blocs conditionnels, les boucles et les fonctions. Par indentation, on entend le fait de décaler les lignes de code vers la droite à l'aide d'espaces ou de tabulations pour montrer la structure logique du programme. De plus, le programme doit être **structuré de manière logique**, en séparant clairement les différentes parties : déclaration des constantes et variables au début, phase d'initialisation, boucle principale, puis fonctions auxiliaires.

En appliquant ces règles, le programme devient plus lisible, plus structuré et plus facile à déboguer ou à faire évoluer.

Références bibliographiques

1. Bouchet, D., *Architecture des ordinateurs : Systèmes à microprocesseur*, support de cours MLI336, Université Paris Descartes (Paris 5), Faculté des Sciences, année universitaire 2013–2014.
2. Farouki, A., Laroussi, T., Benhabiles, T., *Microprocesseurs 8086 : Support de cours*, Université de Constantine, Algérie.
3. Fontaine, A.-B., *Le microprocesseur 16 bits 8086/8088*, 2e édition, Paris : Masson, 1997.
4. Aumiaux, M., *Microprocesseurs 16 bits*, Paris : Éditions Eyrolles (ou préciser si autre éditeur), 1997.
5. Lilen, H., *Cours fondamental des microprocesseurs*, Paris : Dunod, 1993.
6. Haggège, J.-Y., *Microprocesseurs*. Support de cours, Institut National Supérieur de l'Enseignement Technologique (INSET), Tunisie, 2003.
7. Brey, B. B., *The Intel Microprocessors: Architecture, Programming, and Interfacing*, 8th ed., Pearson, 2015.
8. Hall, D. V., *Microprocessors and Interfacing: Programming and Hardware*, 2nd ed., McGraw-Hill, 2006.
9. Crisp, J., *Introduction to Microprocessors and Microcontrollers*, 2nd ed., Elsevier, 2004.
10. Ayala, K. J., *The 8051 Microcontroller*, 3rd ed., Cengage Learning, 2004.
11. Tavernier, C., *Microcontrôleurs PIC 10, 12, 16 : Description et mise en œuvre*, Paris : Dunod, 2007.
12. Mayeux, P., *Apprendre la programmation des PIC Mid-Range par l'expérimentation et la simulation*, Paris : Dunod, 2010.
13. Singh, R. K., *Microcontroller Programming and Interfacing: 8051, PIC, AVR, ARM*, 2nd ed., McGraw-Hill, 2017.
14. Mazidi, M. A., Naimi, S., McKinlay, R., *PIC Microcontroller and Embedded Systems: Using Assembly and C for PIC18*, 2nd ed., Pearson, 2008.