RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ IBN-KHALDOUN DE TIARET

FACULTÉ DES SCIENCES APPLIQUEES
DÉPARTEMENT DE GENIE ELECTRIQUE

Polycopié de cours

Microprocesseurs et Microcontrôleurs

Dr KOUADRIA Mohamed

Cours destiné aux étudiants de Master Niveau : 1

Spécialité : Electrotechnique
Option : Commandes Electriques et Réseaux Electriques

Année: 2024

Avant-propos

Ce polycopié s'adresse aux étudiants de Master 1 en électrotechnique, option commandes électriques et réseaux électriques. Il vise à introduire les concepts fondamentaux des microprocesseurs et des microcontrôleurs, en se concentrant exclusivement sur les aspects théoriques nécessaires à leur compréhension et leur utilisation.

Les objectifs principaux de cet enseignement sont :

- `Comprendre` la structure d'un microprocesseur et son utilité.
- `Faire` la distinction entre microprocesseur, microcontrôleur et calculateur.
- `Découvrir` l'organisation d'une mémoire.
- `Maîtriser` les bases de la programmation en langage assembleur.
- `Comprendre` l'utilisation des interfaces d'entrée/sortie et des interruptions.
- `Assimiler` les principes liés à l'utilisation des microcontrôleurs dans le cadre de systèmes de commande.

Matières prérequises

Pour suivre cet enseignement dans de bonnes conditions, les étudiants doivent disposer des connaissances suivantes :

- Logiques combinatoire et séquentielle : indispensables pour comprendre les principes de base des systèmes numériques.
- Automatismes industriels : pour établir des liens entre les systèmes électroniques et leurs applications en commande de processus.

Ces bases permettront aux étudiants d'aborder efficacement ce cours, conçu de manière claire et progressive, pour comprendre les microprocesseurs et microcontrôleurs ainsi que leur rôle essentiel dans les systèmes modernes d'électrotechnique.

Sommaire

Chapitre 4: Architecture et fonctionnement d'un microcontrôleur

Chapitre 5: Applications des microprocesseurs et microcontrôleurs

Introduction générale

Dans un environnement de plus en plus numérisé et automatisé, les microprocesseurs et microcontrôleurs occupent une place clé dans la conception et la gestion des systèmes électroniques ainsi que des dispositifs intelligents. Ils sont indispensables au bon fonctionnement des technologies actuelles, qu'il s'agisse des systèmes de commande industrielle, des réseaux électriques ou des équipements périphériques.

Ce cours a pour objectif de vous familiariser avec les principes de base des microprocesseurs et des microcontrôleurs, en explorant à la fois leur architecture, leur fonctionnement et leurs applications concrètes, notamment pour la mise en œuvre de solutions automatisées, la commande de dispositifs électroniques et l'optimisation des systèmes électriques.

Dans le **premier chapitre**, vous étudierez l'architecture et le fonctionnement d'un microprocesseur. À travers l'exemple du microprocesseur Intel 8086, vous découvrirez les composants internes, leur rôle et leur interaction pour permettre le traitement des données.

Le **deuxième chapitre** est consacré à la programmation en assembleur. Vous apprendrez les bases de ce langage bas niveau, essentiel pour programmer directement un microprocesseur et optimiser les performances des systèmes.

Dans le **troisième chapitre**, nous aborderons les interruptions et les interfaces d'entrée/sortie. Ces concepts sont cruciaux pour la gestion des interactions entre un microprocesseur et les composants externes, comme les capteurs ou les actionneurs, dans un système de commande.

Le **quatrième chapitre** portera sur l'architecture et le fonctionnement d'un microcontrôleur, avec un focus particulier sur le microcontrôleur PIC. Vous étudierez sa structure, ses caractéristiques et son utilisation dans les systèmes de commande. En complément, une introduction au microcontrôleur de la carte Arduino sera incluse pour illustrer qu'il existe d'autres options couramment utilisées, notamment dans les applications modernes.

Enfin, dans le **cinquième chapitre**, vous explorerez diverses applications des microprocesseurs et des microcontrôleurs dans des domaines comme l'interface LCD, la commande de moteurs, la mesure de fréquence, le contrôle des appareils AC/DC, et la gestion des systèmes d'acquisition de données.

Ce cours est conçu pour vous fournir une compréhension de ces technologies, afin que vous puissiez les intégrer efficacement dans vos futurs projets professionnels, qu'ils concernent la conception de systèmes de contrôle, la gestion des réseaux électriques ou l'automatisation des processus industriels. Ce module s'appuie sur des connaissances préalables en logiques combinatoire et séquentielle, ainsi qu'en automatismes industriels, afin de vous offrir une base solide pour appréhender les sujets abordés.

Chapitre 1 : Architecture et fonctionnement d’un microprocesseur

Objectifs

Une compréhension approfondie du microprocesseur est indispensable pour les étudiants dans les domaines de l'électricité, de l'électronique, de l'instrumentation et du génie logiciel. A l’issue de ce chapitre, vous serez capable de :

Connaître structure d’un calculateur,

Comprendre Circulation de l’information dans un calculateur,

Décrire les différents éléments matériels constituant un microprocesseur,

Expliquer le principe de fonctionnement d’un microprocesseur,

Expliquer le principe de fonctionnement des mémoires.

1.1 Introduction

Ce chapitre vous introduira aux bases du microprocesseur, un composant essentiel dans les systèmes informatiques. Nous commencerons par comprendre comment un calculateur est structuré et comment l'information circule à l'intérieur. Ensuite, nous explorerons les principales parties d'un microprocesseur et son rôle dans l'exécution des programmes, en interaction avec les mémoires. L'exemple du microprocesseur Intel 8086 vous aidera à illustrer ces notions fondamentales.

1.2 Historique

- `1971` **:** Le premier a été inventé par Intel ; une entreprise américaine fondée le 18 juillet 1968 pour la fabrication des semi-conducteurs), d'ailleurs il était nommé Intel 4004 et fonctionne avec une fréquence d'horloge de `740KHz`**.**

- `1974` : Le premier microprocesseur employé de façon courante a été le Intel 8080, mais il était destiné pour les gros ordinateurs dans l'univers des entreprises. Sa fréquence d'horloge était de `2 MHz`.

- `De 1979 à 1989` développement des microprocesseurs de la famille 80_286, 80_386, 80_486 avec la fréquence d'horloge allant de `16MHz à 100MHz`.

- `Entre 1993 et 2004` c'est la configuration Pentium qui s'est généralisée avec une fréquence d'horloge pouvant atteindre `3.6 GHz`.

- `Depuis 2006 jusqu'à 2019`, il y a plusieurs configurations de microprocesseurs. Chez intel on parle de Core i3, i5, i7 et i9 avec une fréquence d'horloge pouvant dépasser les `4.50 GHz`**.**

`En 2022`, Intel appelle la version « KS » du haut de gamme i9 12900K « le processeur de bureau le plus rapide au monde » en se basant sur le fait qu'il prend en charge des vitesses turbo de `5,5 GHz`.

La Figure1.1 illustre une sélection de microprocesseurs classés en fonction de leurs fréquences et de la période durant laquelle ils ont été utilisés.

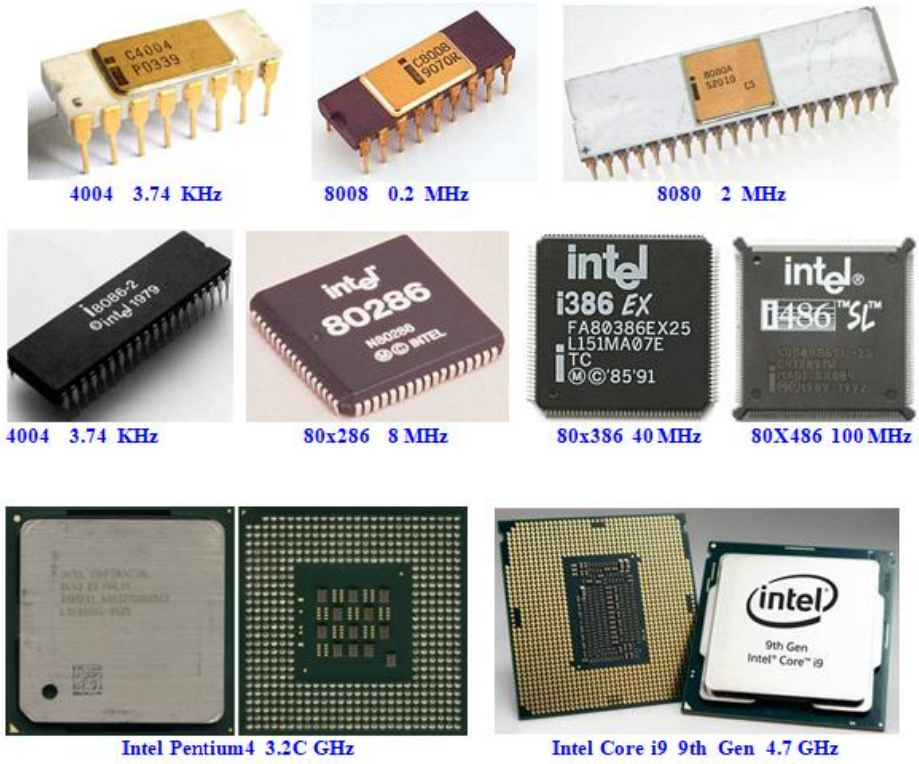


Figure 1.1 *Type de microprocesseurs et leurs fréquences*

Exemple 1.1

Quelle est la différence entre la fréquence d'horloge du microprocesseur conçu en 1971 et celle du microprocesseur de l'année 2019 ?

Solution

La première génération de microprocesseurs, comme l'Intel `4004` lancé en 1971, avait une fréquence d'horloge d'environ `740 kHz`. En revanche, les microprocesseurs modernes de 2019, comme l'`Intel Core i9-9900K`, ont des fréquences d'horloge qui peuvent atteindre environ `5 GHz (5000 MHz).`

Pour déterminer combien de fois le microprocesseur de 2019 est plus rapide que celui de 1971, on utilise :

$$Rapport\,de\,vitesse = \frac{Fréquence\,de\,2019}{Fréquence\,de\,1971} = \frac{5000\,MHz}{0.740\,MHz} \approx 6756,76$$

Ainsi, le microprocesseur de 2019 est environ `6757 fois` plus rapide que celui de 1971.

1.3 Structure et fonctionnement des calculateurs

1.3.1 Structure d'un calculateur

Cette section va permettre de saisir la nature d'un calculateur ainsi que les différences qui le distinguent d'un microprocesseur.

Le microprocesseur est un circuit intégré qui traite les informations à l'aide de composants internes comme l'unité arithmétique et logique et l'unité de contrôle. Cependant, il ne peut pas stocker d'informations par lui-même, donc la `mémoire` est indispensable pour accéder et enregistrer les données. De plus, sans les `circuits d'interface d'entrée et de sortie (E/S)`, le microprocesseur ne pourrait communiquer avec l'extérieur. Un signal d'horloge est aussi nécessaire pour rythmer son fonctionnement. L'ensemble de ces éléments forme un `calculateur` (figure 1.2).

- Si les informations sont assez importantes alors le microprocesseur devrait être assez puissant, et ce qui demande une mémoire assez importante pour stocker telles informations, on parle alors d'un `micro-ordinateur`.

- Si les informations sont moins importantes et complètement liées à une fonction d'usage donnée, par exemple l'électroménager, les jouets,…et qui sont des systèmes embarqués, on parle alors d'un calculateur spécialisé ou d'un `microcontrôleur.`

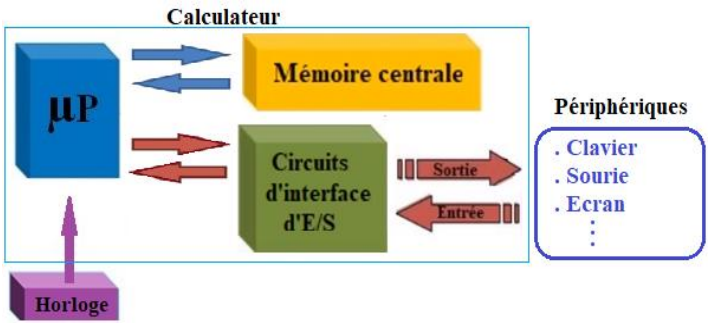


Figure 1.2 *Structure d'un calculateur*

Exemple 1.2

Quelle est la différence entre un microprocesseur et un microcontrôleur?

Solution

`Microprocesseur`	`Microcontrôleur`
• Les composants de mémoires et d'E/S sont connectés en externe	• Dispose d'un microprocesseur ainsi que les composants de mémoires et d'E/S
• Le coût de l'ensemble augmente	• Le coût est faible
• Circuit est volumineux	• Circuit est petit
• Est le cœur d'un système informatique	• Est le cœur d'un système embarqué

1.3.2 Circulation de l'information dans un calculateur

Un calculateur est capable de traiter divers types d'informations : `nombres, textes, images, sons, vidéos`. De plus, les `instructions d'un programme informatique` sont aussi des informations.

L'information circule dans un calculateur principalement à travers des *bus*, qui sont des ensembles de lignes de communication.

- Les `bus de données` transportent les données entre les composants.
- Les `bus d'adresse` spécifient où les données doivent être envoyées ou lues en mémoire.
- Les `bus de contrôle` acheminent les signaux de commande pour coordonner les opérations.

Ces bus connectent le microprocesseur, la mémoire et les périphériques d'entrée/sortie, permettant la transmission des informations et le bon fonctionnement du calculateur. Ce principe est illustré dans la figure 1.3

➢ **Remarque**

Ces types de bus fonctionnent toujours ensemble, et jamais un bus n'est utilisé seul. Chacun a un rôle spécifique mais complémentaire pour assurer le bon déroulement des opérations dans le système.

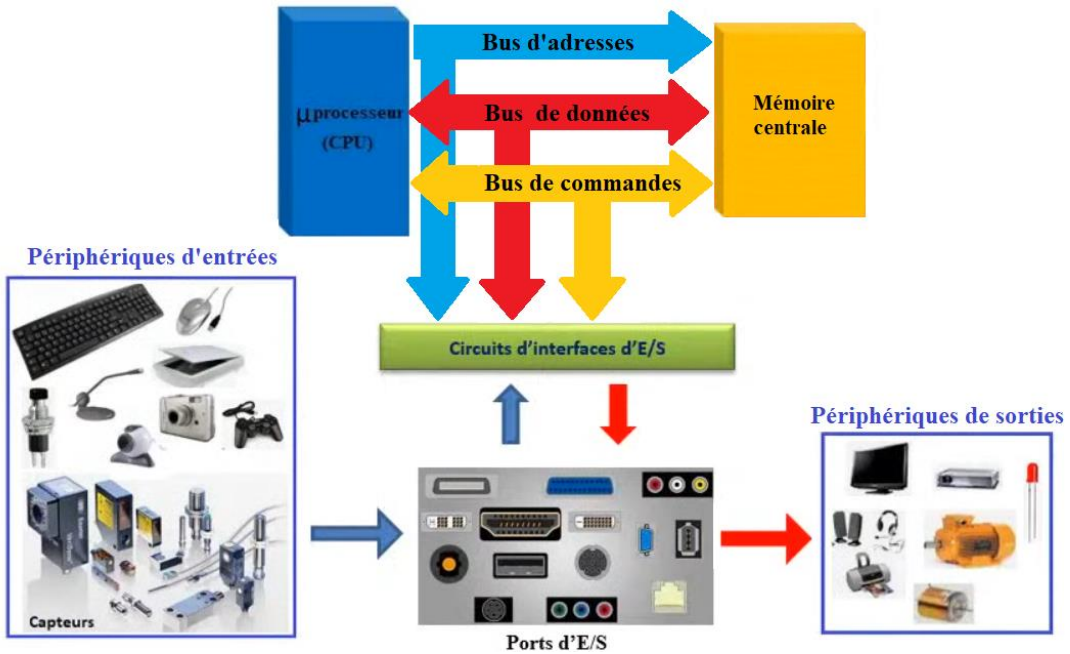


Figure 1.3 *Organisation des informations*

Exemple 1.3

Lorsqu'un `calculateur` effectue une addition (comme 5 + 3), le nombre 5 est récupéré depuis la mémoire via le `bus de données`, tandis que le `bus d'adresse` spécifie la localisation en mémoire de ce nombre. Le microprocesseur exécute ensuite l'opération en utilisant des instructions transmises par le `bus de contrôle`.

Exemple 1.4

Un calculateur doit additionner les nombres 8 et 5. Explique le rôle de chaque type de bus (bus de données, bus d'adresse, bus de contrôle) dans cette opération.

Solution

- `Bus d'adresse` **:**

Le bus d'adresse détermine l'emplacement des nombres 8 et 5 en mémoire. Il indique au calculateur où aller chercher ces valeurs.

- `Bus de données` **:**

Le bus de données transporte les nombres 8 et 5 de la mémoire vers le microprocesseur, où l'addition sera effectuée.

- `Bus de contrôle` **:**

Le bus de contrôle envoie les signaux pour coordonner l'opération. Il indique au microprocesseur quand il doit lire les nombres et réaliser l'addition.

1.4 Description matérielle d'un microprocesseur

Dans ce chapitre, nous aborderons en détail le microprocesseur 8086 d'Intel.

- Il a été introduit en 1978.
- C'est un microprocesseur avec une architecture `16 bits` et en technologie `HMOS` ***(High performance Metal-Oxide Semiconductor***).
- Il exécute les instructions à 2,5 MIPS (millions d'instructions par seconde).
- Le temps d'exécution d'une instruction est de 400 ns (= 1/MIPS = 1/ (2,5 x 10^6)).
- Il dispose d'un bus d'adresses de `20 bits`, il peut donc adresser 2^{20} `=1 MégaBytes` emplacements de mémoire.
- Ses fréquences d'horloge pour ses différentes versions sont : `5, 8 et 10 MHz`.
- Il a un jeu de `123 instructions`
- Le microprocesseur 8086 peut se présenter sous différentes variétés :

Types	Vitesses
8086	5 MHz
8086-1	10 MHz
8086-2	8 MHz

➢ **Fondamental**

Le choix est porté sur le 8086, parce que c'est un microprocesseur dont l'architecture est à la base de tous les processeurs de la série 80x86. Tout type de microprocesseur Intel du 80186 au Pentium III est rétro compatible (***signifie qu'un matériel ou logiciel prend en charge le même ensemble d'instructions qu'un système plus ancien***) avec le 8086 et peut fonctionner comme s'il s'agissait d'un 8086.

Exercice 1.1

Calculez combien de cycles d'horloge sont nécessaires pour exécuter une instruction, en supposant que le microprocesseur utilise une fréquence d'horloge de 10 MHz.

Solution

❖ `Étape 1 : Calcul de la période d'un cycle d'horloge`

La fréquence est l'inverse de la période. Pour une horloge de 10 MHz :

$$Période = \frac{1}{Fréquence} = \frac{1}{10\,x10^{6}} = 0.1\mu s = 100\,ns$$

❖ `Étape 2 : Calcul du nombre de cycles d'horloge nécessaires`

Une instruction prend 400 ns pour s'exécuter, et chaque cycle d'horloge dure 100 ns. Ainsi, le nombre de cycles d'horloge par instruction est :

$$Nombre\ de\ cycles = \frac{Temps\ d'exécution\ d'une\ instruction}{Période\ d'un\ cycle} = \frac{400\,ns}{100\,ns} = 4\,cycles$$

Exercice1.2

Quelle est l'impact de la fréquence d'horloge sur le nombre d'instructions exécutées par seconde ? Si le microprocesseur passe de 5 MHz à 10 MHz, comment cela affectera-t-il le temps d'exécution d'une instruction ?

Solution

Dans cet exercice, il faut penser à l'influence de la fréquence sur la rapidité des cycles d'horloge.

❖ `Impact de la fréquence d'horloge sur le nombre d'instructions exécutées par seconde`

Puisque la fréquence d'horloge affecte directement la vitesse des cycles, une fréquence d'horloge plus élevée réduit le temps d'un cycle, ce qui augmente le nombre d'instructions exécutées par seconde.

❖ Si la fréquence passe de 5 MHz à 10 MHz :

- A 5 MHz, la période d'un cycle est :

$$\frac{1}{5x10^6} = 200\,ns\ \ par\ cycle$$

- A 10 MHz, la période d'un cycle est de 100 ns (comme calculé précédemment).
- Puisque le temps d'exécution d'une instruction est de 400 ns, passer de 5 MHz à 10 MHz ne change pas le nombre de cycles par instruction (4 cycles), mais chaque cycle est plus rapide.

Cela signifie que le microprocesseur pourra exécuter des instructions plus rapidement à 10 MHz qu'à 5 MHz, permettant d'atteindre 2,5 MIPS avec une cadence plus rapide.

- Donc, en augmentant la fréquence d'horloge, le microprocesseur exécute davantage d'instructions par seconde. Passer de 5 MHz à 10 MHz divise le temps d'un cycle par deux, augmentant ainsi le rythme d'exécution des instructions.

1.4.1 Brochage du 8086

Le microprocesseur 8086 est construit sur une seule puce semi-conductrice de type **DIP à 40 broches** (***D**ua1 **I**n-Line **P**ackage*). La figure 1.4 montre ce type de circuit et ses différentes broches. Ainsi, sur 40 broches :

- `32 signaux` sont des signaux communs
- `8 signaux` sont utilisés séparément pour le mode minimum
- `8 signaux` sont utilisés séparément pour le mode maximum

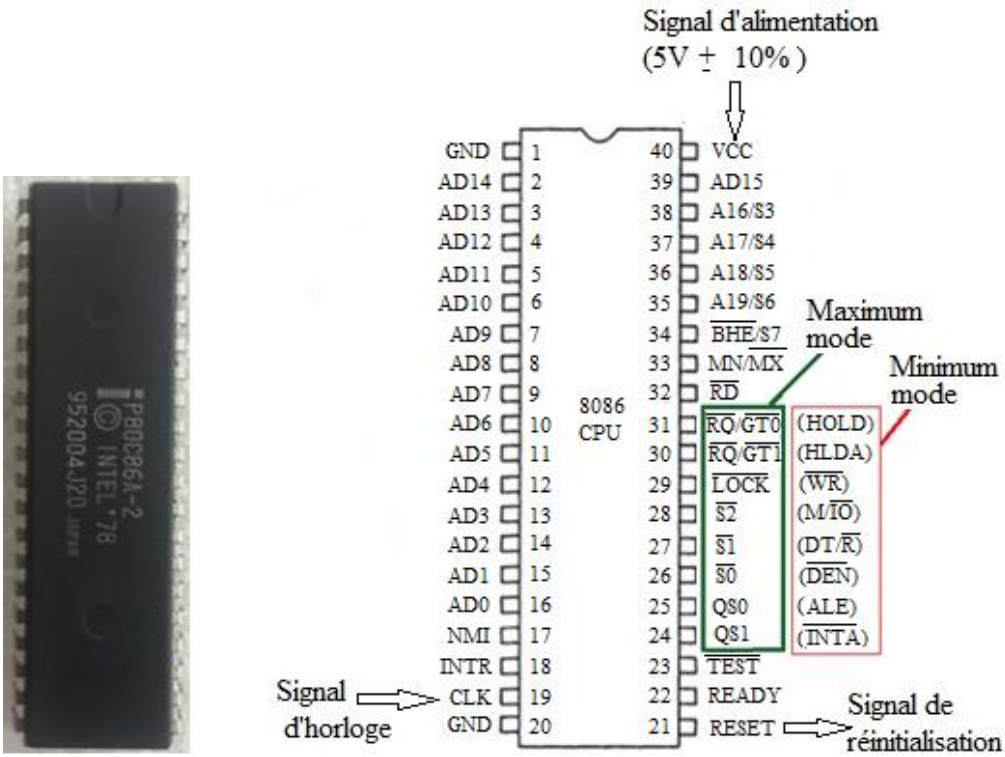


Figure 1.4 *Brochage du microprocesseur 8086*

Les broches et leurs fonctions

`AD0…AD15` **: C'est** un bus d'**adresses** ou de **données**. Ce bus contient les:

- `adresses` si le signal **ALE** est au niveau logique « **1** »
- `données` si le signal **ALE** est au niveau logique « **0** »

- **Principe**

La figure 1.5 représente un diagramme de temporisation et illustre les signaux de microprocesseur au cours de l'opération de séparation entre les adresses et les données.

Il est montré que le **cycle de bus** correspond à une séquence d'événements qui commence par la sortie d'une **adresse** sur le bus d'adresses système suivie d'un transfert de **données** d'écriture ou de lecture. De plus, au cours de cette opération, une série de signaux de commande (ALE, RD, WR) est également produite par le microprocesseur pour contrôler la direction et la synchronisation du bus.

Il y a au moins quatre périodes d'horloge dans un cycle de bus du microprocesseur 8086. Ces quatre périodes d'horloge sont appelées états **T1, T2, T3 et T4.**

Pendant **T1**, les seize lignes inférieures portent l'adresse (**A0 - A15**), tandis que pendant **T2, T3 et T4** elles transportent des données.

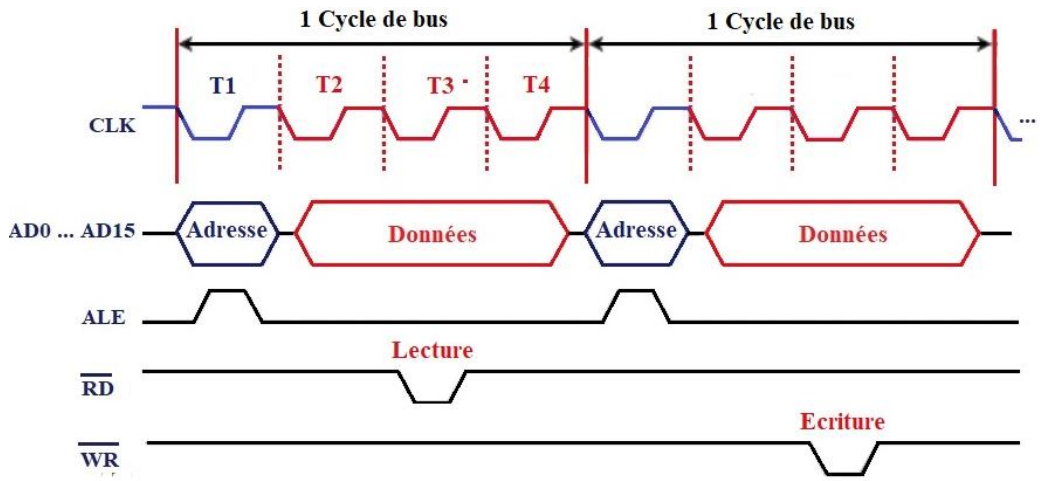


Figure 1.5 *Diagramme de temporisation.*
*Transfert des adresses et des données sur le bus **A/D***

ALE	**A**ddress **L**atch **E**nable. Signal de verrouillage d'adresse chaque fois que cette broche est activée (niveau logique « 1 »). Voir figure 3.2.
MN/MX	**M**inimum/ **M**aximum. Lorsque la broche **MN/MX** est **haute**, le 8086 fonctionne en **mode MIN** lorsque la broche **MN/MX est basse**, le 8086 fonctionne en **mode MAX**.
RD	**R**ead Control. Est le signal de lecture de données. Voir figure 3.2.
WR	**W**rite Control. Est le signal d'écriture de données. Voir figure 3.2.
READY	Wait state control. Est le signal d'entrée de synchronisation avec la mémoire.
NMI	**N**on-**M**askable **I**nterrup. C'est une interruption non masquable, ce qui signifie que nous ne pouvons pas désactiver ou ignorer cette interruption
INTR	**Int**errupt **R**equest.
INTA	**I**nterrupt **A**cknowledge. Signal indiquant que le microprocesseur accepte l'interruption
DEN	**D**ata **E**nable. Ce signal indique que des données sont en train de circuler sur le bus **A/D**
DT/R	**D**ata **T**ransmit/**R**eceive. Indique le sens de transfert de données Lorsque **DT/R =1**, le 8086 **émis les données** (en mode écriture) Lorsque **DT/R =0**, le 8086 **reçoit les données** (en mode lecture)

M/IO	**M**emory/**I**nput-**O**utput. Ce signal indique si le 8086 s'adresse à la mémoire ou aux Entrées/Sorties. Lorsque **M/IO=1**, le 8086 s'adresse à la mémoire Lorsque **M/IO=0**, le 8086 s'adresse aux Entrées/Sorties
BHE	**B**us **H**igh **E**nable Signal de lecture de l'octet de poids fort du bus de données
HOLD	l'entrée hold demande un accès direct à la mémoire
HLDA	**H**old **A**cknowledge indique que le microprocesseur 8086 est entré dans l'état d'attente.

Aussi, le microprocesseur 8086 peut fonctionner en deux modes :

Mode minimum	Mode maximum
Dans ce cas, il ne peut y avoir que le microprocesseur 8086 seul qui doit fournir tous les signaux de commande nécessaires aux opérations de mémoire et à l'interface des entrées/sorties. Ainsi, ce type de mode a un circuit moins complexe que le mode maximum.	Dans ce mode, le microprocesseur 8086 est connecté à plusieurs coprocesseurs pouvant effectuer leurs propres programmes (fonctions). **Exemple** : le 8086 est considéré comme le microprocesseur principal, il peut être associé au : ▪ **coprocesseur 8087**, qui ne fera que les opérations numériques. ▪ **coprocesseur 8089,** qui n'est dû qu'aux opérations d'entrées/sorties.

1.4.2 Structure interne du microprocesseur 8086

L'architecture du 8086 dispose de deux unités fonctionnelles distinctes :

- l'unité d'interface de bus (Bus Interface Unit - **BIU**)
- l'unité d'exécution (Execution Unit - **EU**).

Ces deux unités sont comme suit :

UIB	▪ Récupère l'instruction suivante ▪ Calcule les adresses d'adresse physique ▪ Gère la file d'attente (Queue de 6 octets). ▪ Etablit les communications avec les composants externes au microprocesseur (Mémoires, Interfaces d'E/S)
UE	▪ Décode et exécute les instructions

Figure 1.6 *Les unités fonctionnelles du 8086*

➢ **Fondamental**

La principale raison pour laquelle l'architecture du 8086 est divisée en ces 2 unités est à cause du concept connu sous le nom de **pipelining** ou le **traitement parallèle**. Donc, le traitement parallèle est essentiel. Quand le microprocesseur **exécute** des instructions dans **UE**, en même temps l'**UIB récupère (Fetch** en anglais**)** les instructions de la mémoire principale. Par conséquent, le 8086 prend en charge le pipeline en **2 étapes** :
Etape1 : **Fetch ,** Etape2 : **Execute**

L'architecture du 8086 est illustrée ci-dessous à la figure 1.7. Ce qui décrit l'organisation globale des deux unités à l'intérieur de la puce.

Nous remarquons que l'**UIB** a :

- des registres de segments,
- un pointeur d'instructions,
- un bloc logique de génération d'adresses et de contrôle de bus,
- une file d'attente d'instructions

tandis que l'**UE** a :

- des registres à usage général,
- ALU, une unité de contrôle,
- un registre d'indicateur (ou d'état).

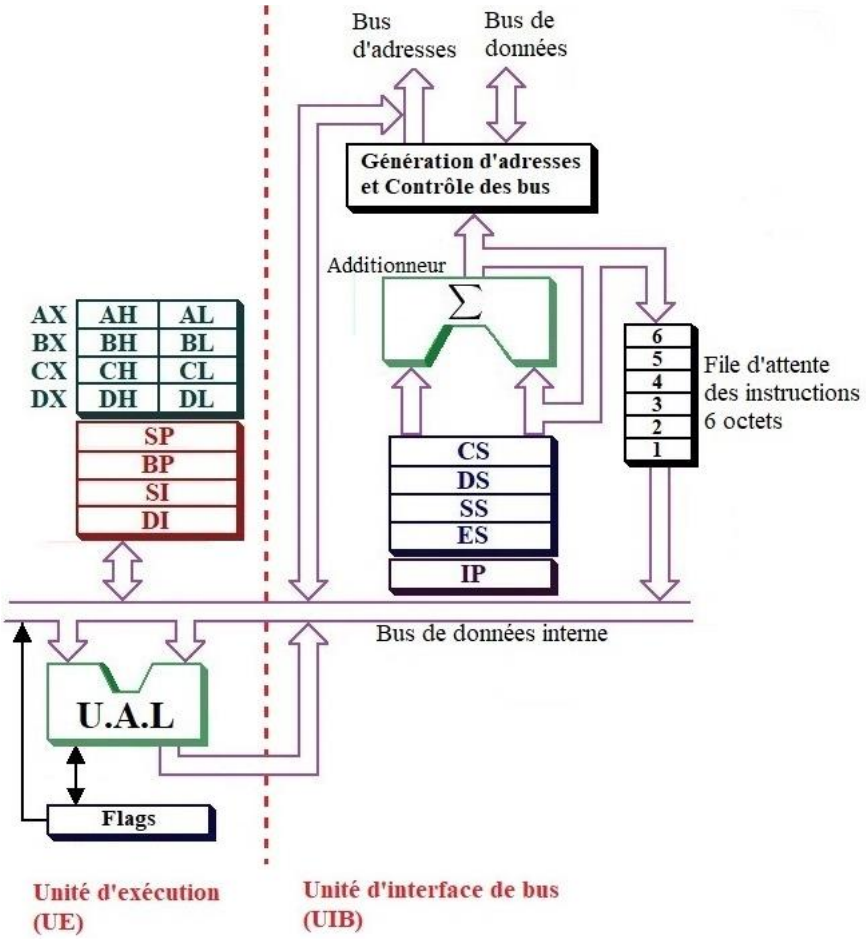


Figure 1.7 *Architecture du microprocesseur 8086*
*Séparation entre l'unité d'exécution (**UE**) et l'unité d'interface de bus (**UIB**).*

L'unité d'interface bus (**UIB**) du microprocesseur 8086 a pour principales tâches de :

1. Gérer les échanges de données entre le microprocesseur et la mémoire ou les périphériques externes.
2. Contrôler les cycles de lecture et d'écriture en gérant les signaux de commande.
3. Réaliser l'adressage de la mémoire et la génération des adresses physiques.
4. Permet de stocker temporairement les `instructions suivantes` en mémoire tampon appelée *file d'attente d'instructions* (ou *instruction queue*) pendant que l'unité d'exécution traite l'`instruction actuelle`. Ce qui assure un traitement plus rapide.
5. `Gérer l'accès direct à la mémoire (DMA).` Dans ce cas, l'UIB cède temporairement le contrôle du bus système pour permettre à un périphérique de lire ou d'écrire directement dans la mémoire sans l'intervention du processeur. Cela libère le processeur pour d'autres tâches.

Les principales tâches exercées par l'**UE** sont :

- Décodage/exécution d'instructions.
- Il accepte les instructions de la fin de sortie de la file d'attente d'instructions (résidant dans BIU) et les données des registres à usage général ou de la mémoire.
- Génération des adresses d'opérandes si nécessaire, les transmet à l'UIB en lui demandant d'effectuer un cycle de lecture ou d'écriture dans la mémoire ou les périphériques d'E/S.
- Test l'état des indicateurs dans le registre d'état et les met à jour lors de l'exécution des instructions.
- Attente des instructions de la file d'attente d'instructions lorsqu'elle est vide.

Exemple1.5

Pourquoi les 2 unités (UIB) et (UE) travaillent en parallèle ?

Solution : Le fonctionnement de ces 2 unités en parallèle permet d'accélérer la vitesse d'exécution. Ainsi, grâce à ce concept, l'exécution du programme devient plus rapide.

Exemple 1.6

Représenter le traitement d'un programme de 3 instructions en considérant les cas sans et avec le concept de pipeline.

Solution :

Sans pipeline	Fetch1	Execute1	Fetch2	Execute2	Fetch3	Execute3

Le programme est traité en 6 étapes

Avec pipeline	Fetch1	Fetch2	Fetch3	
		Execute1	Execute2	Execute3

Dans ce cas, le programme est traité en 4 étapes

Exercice1.3

Un microprocesseur 8086 fonctionne avec un pipeline simple en deux étapes : l'étape de "Fetch" (récupération) et l'étape "Execute" (exécution). Ce pipeline permet de récupérer une instruction pendant que l'instruction précédente est en cours d'exécution.

Supposons que chaque étape (fetch ou execute) prenne 4 cycles d'horloge.

- Si l'UIB récupère (fetch) une instruction et que l'UE exécute (execute) l'instruction précédente, combien de cycles d'horloge sont nécessaires pour compléter les deux étapes d'une instruction ?
- Supposons qu'il y ait trois instructions à exécuter (Instruction 1, Instruction 2, Instruction 3).

a. Combien de cycles d'horloge prendrait chaque instruction si le microprocesseur n'avait pas de pipeline (c'est-à-dire, si l'instruction suivante ne pouvait être récupérée qu'après l'exécution de la précédente) ?

b. En utilisant le pipelining du 8086, combien de cycles d'horloge sont nécessaires pour exécuter les trois instructions ?

c. Expliquez en une phrase en quoi le pipelining est avantageux en termes de performance du microprocesseur.

Solution

- Puisque chaque étape (**Fetch** ou **Execute**) prend 4 cycles d'horloge, compléter les deux étapes pour une instruction sans pipelining nécessiterait :

4 cycles (**fetch**) + 4 cycles (**execute**) = 8 cycles

Dans ce cas, sans pipelining, une instruction prend 8 cycles d'horloge.

`a. Nombre de cycles sans pipelining`

Si le microprocesseur n'avait pas de pipeline, chaque instruction serait entièrement terminée avant de passer à la suivante. Chaque instruction prendrait 8 cycles d'horloge (comme calculé précédemment).

Pour trois instructions : 8 cycles×3 instructions=24 cycles

Sans pipelining, il faudrait 24 cycles d'horloge pour exécuter les trois instructions.

`b. Nombre de cycles avec pipelining`

Avec le pipelining, le microprocesseur commence à récupérer (fetch) la deuxième instruction pendant qu'il exécute la première. Ce processus de chevauchement permet de réduire le nombre total de cycles.

- `Étape 1` **:** La première instruction est récupérée en 4 cycles.
- `Étape 2` **:** La première instruction est exécutée (4 cycles) pendant que la deuxième instruction est récupérée en parallèle (4 cycles).
- `Étape 3` **:** La deuxième instruction est exécutée (4 cycles) pendant que la troisième instruction est récupérée (4 cycles).
- `Étape 4` **:** La troisième instruction est exécutée en 4 cycles.

Total de cycles avec pipelining :
4(fetch Instruction 1) + 4(execute Instruction 1 et fetch Instruction 2) + 4(execute Instruction 2 et fetch Instruction 3) + 4(execute Instruction 3) =16cycles

Avec pipelining, il faut 16 cycles d'horloge pour exécuter les trois instructions.

c. Le pipelining permet de traiter plusieurs instructions en parallèle, ce qui réduit le temps total d'exécution et améliore l'efficacité du microprocesseur.

1.4.2.1 Les registres du microprocesseur 8086

Il y a en tout 14 registres de 16 bits, voir figure1.8. Les registres du microprocesseur 8086 sont regroupés en plusieurs catégories, chacun ayant des rôles spécifiques :

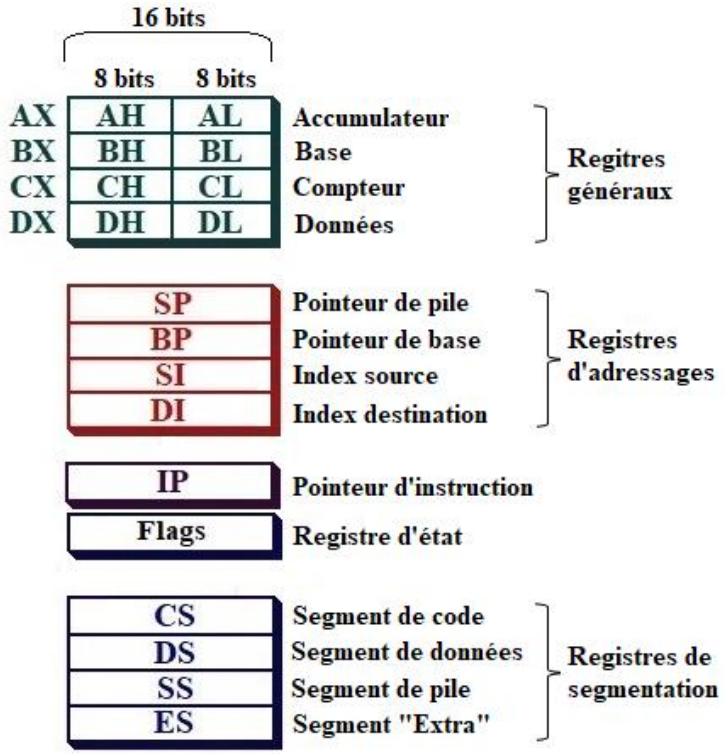


Figure 1.8 *Les registres du microprocesseur Intel 8086*

Note : Les alphabets **X**, **H** et **L** se réfèrent respectivement au **mot**, à l'**octet supérieur** ou à l'**octet inférieur** respectivement de n'importe quel registre.
Exemple : **AL** est le registre d'octet inférieur, **AH** est le registre d'octet supérieur

1.4.2.2 Description des registres du microprocesseur 8086

❖ **Registres généraux**

Le tableau 1.1 présente les registres généraux du microprocesseur 8086, ainsi que leur description. Ces registres sont utilisés pour stocker temporairement des données pendant l'exécution des programmes. Ils sont essentiels pour les calculs, le stockage d'adresses et la gestion du flux d'exécution dans le 8086. Chaque registre a un rôle spécifique, comme le stockage de données, d'adresses ou de résultats d'opérations arithmétiques.

Tableau 1.1 *Description des registres généraux du 8086*

Registre	Opération
AX	**Addition** de mots, **Soustraction** de mots, **Multiplication** de mots, **division** de mots, Opérations **logiques**, **transfert** de mots.
AL	**Addition** d'octets, **Soustraction** d'octets, **Multiplier** les octets, **diviser** les octets, **transfert** d'octets, Opérations **logiques**, **conversion** en BCD, **traitement** de chaines de caractères.
AH	**Addition** d'octets, **Soustraction** d'octets, **Multiplier** les octets, **diviser** les octets, Opérations **logiques**, **transfert** d'octets.
BX	**Addition** de mots, **Soustraction** de mots, Opérations **logiques**, **Calcul** des adresses.
CX	**Addition** de mots, **Soustraction** de mots, Opérations **logiques**, **Compteur** à 16 bits pour les boucles, les **décalages** et les **rotations.**
CL	**Addition** d'octets, **Soustraction** d'octets, Opérations **logiques**, **Compteur** à 8 bits pour les boucles, les **décalages** et les **rotations**
DX	**Addition** de mots, **Soustraction** de mots, Opérations **logiques**, **extension** pour les multiplications et divisions, **Contenir** les adresses des ports d'**E/S**.

Figure 3.6 *Description des registres généraux du 8086*

❖ **Registres d'adresses**

Le tableau 1.2 présente les registres d'adresse du 8086 et leur fonction. Ces registres sont utilisés pour gérer les adresses mémoire, permettant au processeur de localiser et d'accéder aux données et instructions en mémoire.

Tableau 1.2 *Description des registres d'adresses du 8086*

Registre	Opération
SP, BP, SI, DI	▪ Addition de mots, ▪ Soustraction de mots, ▪ Opérations logiques,

	▪ `Calcul` des adresses, ▪ Stockage des adresses (offset) des emplacements de mémoire par rapport aux registres de segment.
`SP, BP`	Pointent vers une adresse à l'intérieur de la pile.
`SI, DI`	Contiennent une adresse d'une position à l'intérieur de la mémoire de données.

❖ **Pointeur d'instruction**

IP (Instruction Pointer) : Pointeur d'instruction, qui indique l'adresse de la prochaine instruction à exécuter.

❖ **Registres de segment**

➢ **Fondamental**

Un **segment** est une partie de l'espace mémoire. Un **segment** constitue donc dans la **mémoire** principale une plage d'adresses.

Le microprocesseur 8086 adresse une mémoire segmentée (*mémoire divisée en plusieurs segments*). La segmentation de la mémoire a permis au microprocesseur 8086 de ne pouvoir accéder en tout qu'à **4** segments de **64 Ko** chacun dans sa plage de mémoire de **1 Mo**. Ainsi, ces 4 segments sont définis par les valeurs de données stockées dans 4 registres de segments indiqués comme suit :

- `CS (Code Segment)` : Contient l'adresse du segment du code.
- `DS (Data Segment)` : Contient l'adresse du segment de données.
- `SS (Stack Segment)` : Contient l'adresse du segment de la pile.
- `ES (Extra Segment)` : Utilisé pour des opérations de données supplémentaires.

1.4.2.3 L'unité arithmétique et logique

Comme son nom l'indique, cette unité peut exécuter deux types d'opérations.

- `Les opérations arithmétiques` incluent l'addition et la soustraction qui sont des opérations de base (une soustraction est une addition avec le complément à deux), la multiplication et la division.
- `Les opérations logiques` sont effectuées bit à bit sur les bits de même poids de deux mots, tel que ET, OU, NOT et le OU_EXCLUSIF, de même les opérations de rotation et de décalage (arithmétique et logique).

La figure 1.9 montre la représentation générale de l'U.A.L.
Où, « **n** » représente le nombre de bits. Pour le microprocesseur 8086, « ***n*** » peut prendre « **8** » ou « **16** ».

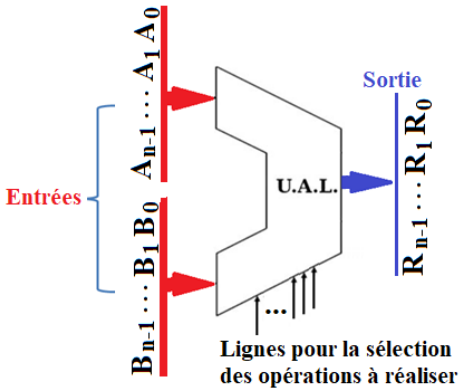


Figure 1.9 *Représentation de l'unité arithmétique et logique. Cas général*

Exemple 1.7

❖ Cas où « n=8 ». L'U.A.L. réalise l'addition des deux nombres sur 8 bits.

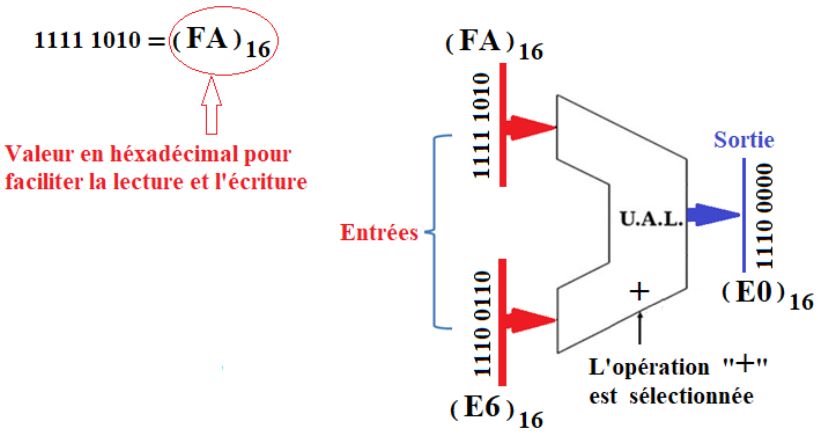


Figure 1.10 *Représentation de l'unité arithmétique et logique. Cas de 8bits*

1.4.2.4 Le registre d'état

Ce registre (registre drapeaux, **flag** en anglais) sert à contenir des bits qui sont des indicateurs dont l'état dépend du résultat de la dernière opération effectuée par le microprocesseur. La figure 1.11 illustre la position de ses bits dans le registre d'état.

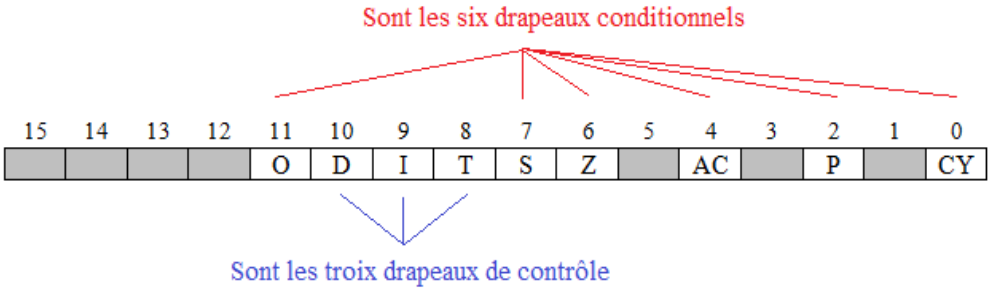


Figure 1.11 *Les bits du registre d'état du microprocesseur 8086*

En général, les **drapeaux conditionnels** ont principalement un lien avec les résultats et l'U.A.L. Ce type de drapeaux ainsi que leurs fonctions sont comme suit :

- Le bit de **retenue** représenté par « **CY** » (**C**arry).
  Alors, **CY=1**, si l'opération aboutit à la génération d'un report à partir du bit le plus significatif du résultat. Ce qui signifie qu'un débordement s'est produit. Donc, « **CY** » est utilisé pour détecter le débordement dans les nombres non signés.

- Le bit de la **parité** (paire ou impaire) représenté par « **P** » (**P**arity).
  Alors, **P =1**, si le nombre de « 1 » du résultat est pair.

- Le bit de la **retenue auxiliaire** représentée par « **AC** » (**A**uxillary **C**arry).
  Alors, **AC=1**, si l'opération aboutit à la génération d'un report après la position partir du « 4 » du résultat. Ce qui signifie que cet indicateur est utilisé avec des opérations 8 bits.

- Le bit de la **nullité** (nul ou différent de zéro) représentée par « **Z** » (**Z**ero).
  Alors**, Z=1**, si le résultat obtenu est égal à « **0** ».

- Le bit de **signe** (positif ou négatif) représenté par « **S** » (**S**ign).
  Ce drapeau est utilisé avec des nombres signés.
  Alors, si une opération arithmétique ou logique est effectuée sur des nombres signés, le bit de cet indicateur **S=1,** si le bit le plus significatif du résultat obtenu est égal à « **1** », ce qui signifie que la valeur de ce résultat est négative.

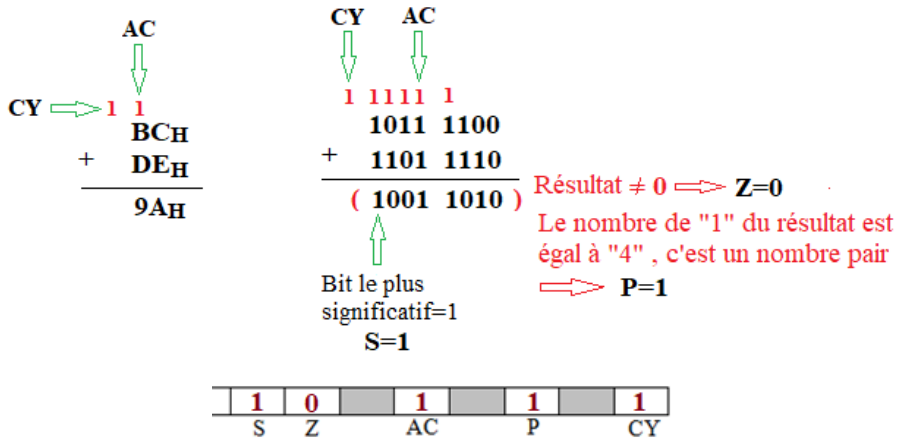
- Le bit de **débordement** représenté par « **O** » (**O**verflow).
  Ce drapeau est utilisé avec des nombres signés pour nous informer si un débordement s'est produit ou non.

Donc, tout cela était à propos sur les drapeaux conditionnels. Pour mieux comprendre le principe de ce type de drapeaux, prenons un exemple.

Exemple 1.8

Pour l'opération suivante : **BCH + DEH**. Quelles sont les valeurs des indicateurs stockées au niveau du registre d'état ?

Solution



Concernant les **drapeaux de contrôle**, ils sont utilisés pour contrôler divers modes du microprocesseur. Ainsi, nous avons :

- Le bit de **mode pas-à-pas** représenté par « **T**» (**T**rap)

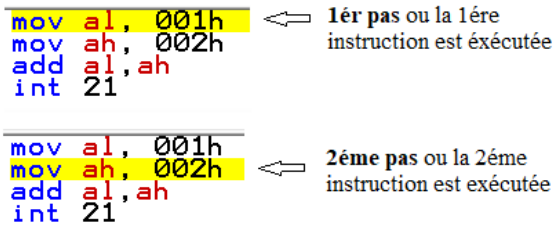
  Cet indicateur joue un rôle très utile dans le débogage des programmes (*débogage signifie corriger l'erreur, ce qui nous aide réellement à exécuter les programmes une instruction à la fois ou bien mode pas-à-pas*).

Exemple1.9

Comment est défini trace mode « **T** » pour exécuter un programme donné ?

Solution

A chaque pas (appui sur le bouton de mode pas-à-pas) « T=1 »



- L'indicateur d'**interruption** représenté par « **I** » (**I**nterrupt)

  Si le microprocesseur met l'indicateur d'interruption **I=0**, ce qui signifie que **l'interruption est désactivée** et qu'il n'est pas disposé à prendre des interruptions du

périphérique d'E/S et les programmes en cours d'exécution ne seront pas interrompus. Donc, ce bit de contrôle autorise ou interdit les interruptions.

- L'indicateur de **direction** représenté par « **D**» (**D**irection)

Ce bit de contrôle est utilisé pour les opérations de chaîne de caractères. Il indique que la manipulation d'une chaîne de caractères s'effectue dans le sens croissant ou décroissant.
Si **D=0**, la chaîne sera lue dans l'ordre croissant.
Si **D=1**, la chaîne sera lue dans l'ordre décroissant.

Exemple 1.10

Nous avons dans une mémoire aux adresses 0100H, 0101H et 0102H des bytes correspondant aux caractères « B », « C » et « D » respectivement. Définir le drapeau de direction selon l'ordre de lecture.

Solution

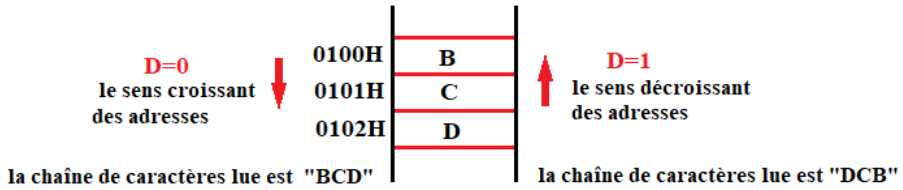


Figure 1.12 *Exemple de fonctionnement du bit de direction « D » du registre d'état*

1.5 Fonctionnement d'un microprocesseur, les mémoires

1.5.1 Fonctionnement d'un microprocesseur

Ci-dessous sont présentées les étapes simplifiées du fonctionnement du microprocesseur, basées sur sa structure interne (voir figure 1.7) :

❖ Récupération d'instructions (UIB) :

- L'**Unité d'Interface de Bus (UIB)** gère l'accès à la mémoire pour récupérer les instructions. Elle utilise le **registre IP (Instruction Pointer)** pour stocker l'adresse de la prochaine instruction et les **registres de segments** (comme le **CS - Code Segment**) pour construire une adresse complète de 20 bits en combinant le segment (CS) et l'offset (IP).

❖ Bloc de génération d'adresses et de contrôle des bus :

- Ce bloc de l'UIB est essentiel pour former l'adresse finale des instructions et des données. Il gère le **bus d'adresses** pour indiquer l'emplacement mémoire ciblé et le **bus de données** pour transférer les instructions et les données entre la mémoire et le microprocesseur. En parallèle, ce bloc utilise le **sommateur** pour incrémenter le registre IP et générer la prochaine adresse d'instruction.

❖ `Stockage des instructions dans la queue :`

- Une fois récupérées, les instructions sont placées dans une **queue** (file d'attente) dans l'UIB, prêtes pour l'exécution par l'UE. Cela permet de préparer les instructions en avance, optimisant le traitement grâce au **pipelining**.

❖ `Décodage et exécution des instructions (UE) :`

- `L'Unité d'Exécution (UE)` décode les instructions récupérées dans la queue et utilise ses différents registres pour exécuter les calculs :
  - `Registres généraux (AX, BX, CX, DX)` : Effectuent des opérations arithmétiques, logiques, et stockent des données temporaires.
  - `Registres pointeurs (SP - Stack Pointer, BP - Base Pointer)` : SP gère le sommet de la pile, et BP accède aux données dans la pile.
  - `Registres d'index (SI - Source Index, DI - Destination Index)` : Utilisés pour les opérations de manipulation de chaînes et pour l'adressage indexé.
  - `Registre de flags` : Contient des indicateurs comme le **flag de zéro** (ZF), le **flag de transport** (CF), et le **flag de signe** (SF), qui reflètent l'état des opérations et permettent de contrôler les branches conditionnelles.

❖ `Traitement parallèle (pipelining) :`

- Pendant que l'UE exécute une instruction, le bloc de génération d'adresses et de contrôle des bus de l'UIB prépare la prochaine instruction en accédant à la mémoire. Ce **pipeline en deux étapes** assure un flux d'instructions continu et efficace.

1.5.2 Les mémoires

1.5.2.1 Types de mémoires

Dans ce cours, nous parlerons de la `RAM` et de la `ROM`. Le 8086 utilise principalement la `RAM` pour les données et le programme en cours d'exécution. La `ROM`, quant à elle, contient

des données permanentes, comme le `BIOS`, utilisées au démarrage du système. Les deux types de mémoire et leur brochage de base sont présentés dans la figure 1.13.

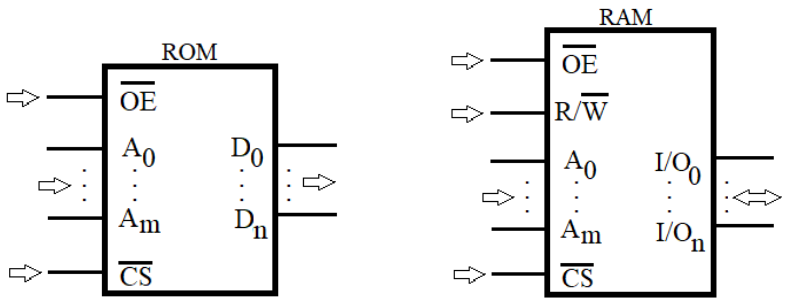


Figure 1.13 *Représentation logique des mémoires ROM et RAM*

$A_0 \ldots A_m$: les bits du bus d'adresse
$D_0 \ldots D_n$: les bits du bus des données
CS : Chip Select (sélection de puce)
OE : Output Enable (activation de sortie)
R/W : Read/Write (lecture/écriture)
I/O : Input/Output (entrée/sortie)

Critère	ROM (Read-Only Memory)	RAM (Random Access Memory)
Nature des données	Contient des données permanentes, non modifiables	Contient des données temporaires, modifiables
Volatilité	Non volatile (conserve les données sans alimentation)	Volatile (perd les données sans alimentation)
Usage principal	Stockage du firmware, BIOS, programmes de démarrage	Mémoire de travail pour les programmes et le système d'exploitation
Accessibilité	Lecture seulement (écriture difficile ou impossible)	Lecture et écriture
Vitesse d'accès	Généralement plus lente	Rapide
Capacité	limitée	plus élevée
Coût	Moins coûteuse	Plus coûteuse

Exemple1.11

Concevoir une mémoire de taille **16 × 8** à partir de modules de mémoire **16 × 4**

Solution

Dans la figure1.14, une mémoire de **16 × 8** est construite en utilisant deux modules de mémoire **16 × 4**. Chaque module peut stocker 16 mots de 4 bits. Pour avoir une mémoire de **16 × 8**, on combine les deux modules : l'un stocke les 4 premiers bits et l'autre stocke les 4 bits restants pour chaque mot. Cela permet d'obtenir un mot de 8 bits avec deux modules de 4 bits. C'est une méthode simple pour augmenter la capacité de stockage d'une mémoire.

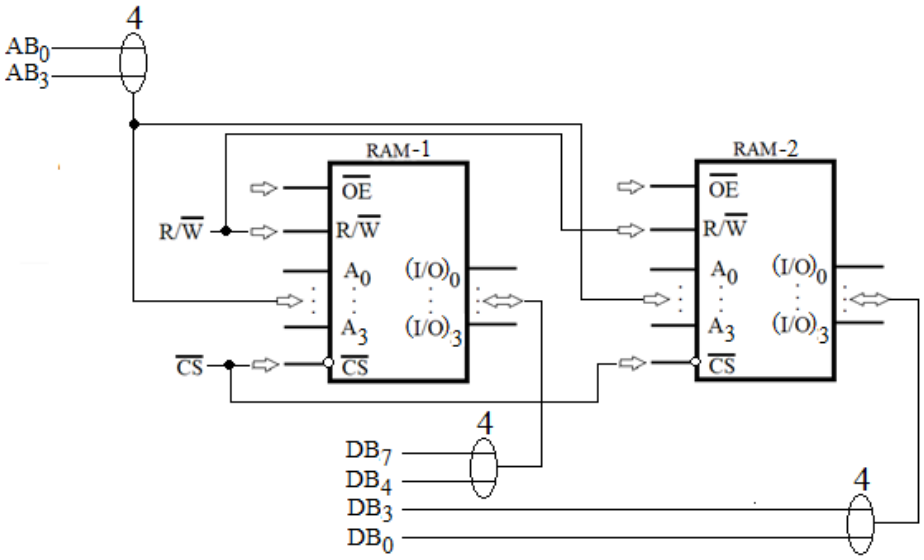


Figure 1.14 *Exemple de mémoire RAM de 16X8*

Exemple1.12

Développer un module de mémoire **32 × 4** en combinant deux puces de mémoire **16 × 4**.

Solution

Dans la figure1.15, une mémoire de **32 × 4** est réalisée en combinant deux puces de mémoire **16 × 4**. Chaque puce de **16 × 4** peut stocker 16 mots de 4 bits. Pour obtenir une mémoire de **32 × 4**, on utilise les deux puces en parallèle. Cela permet de doubler le nombre de mots tout en gardant la même taille de mot (4 bits). Chaque puce va gérer une moitié de l'adresse mémoire, et ensemble, elles permettent de stocker 32 mots de 4 bits. Cette méthode est utilisée pour augmenter la capacité de mémoire en reliant plusieurs puces de plus petite capacité.

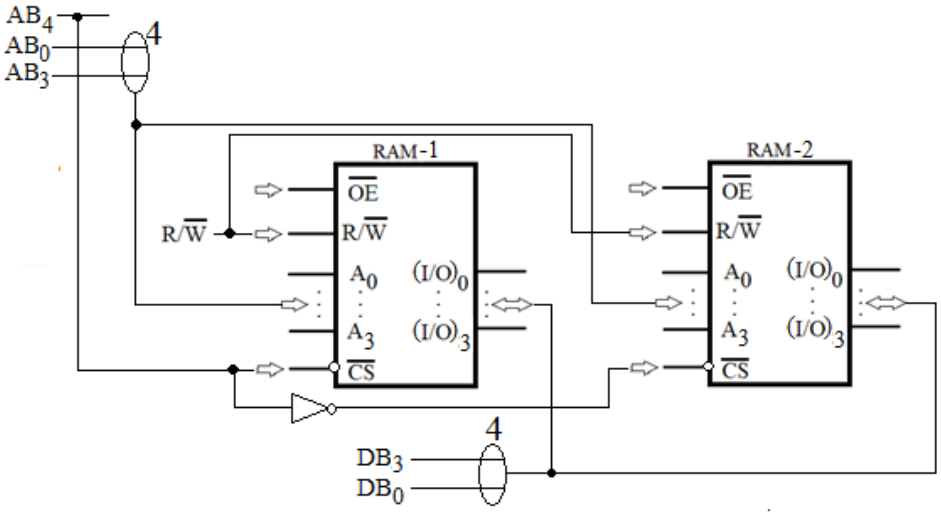


Figure 1.15 *Exemple de mémoire RAM 32X4*

1.5.2.2 Capacité d'une mémoire

La capacité de la mémoire dépend de deux éléments principaux :

1. **Le nombre de cellules de mémoire (adressables)**, déterminé par la largeur du bus d'adresse (A_0 à Am).
2. **La largeur de chaque cellule de mémoire**, déterminée par la largeur du bus de données (D_0 à Dn).

`1. Nombre de cellules adressables (en fonction du bus d'adresse)`

Le bus d'adresse détermine le nombre de `cellules de mémoire` (ou `emplacements mémoire`) qui peuvent être adressées. Si le bus d'adresse est constitué de `m` lignes (A_0 à Am), le nombre total d'adresses possibles est donné par :

$$\text{Nombre d'adresses} = 2^m$$

Chaque adresse correspond à une `cellule de mémoire`.

`2. Taille de chaque cellule de mémoire (en fonction du bus de données)`

Le bus de données détermine la `taille de chaque cellule mémoire`, c'est-à-dire le nombre de bits ou d'octets qui peuvent être stockés dans chaque emplacement mémoire. Si le bus de données a `n` lignes (D0 à Dn), la taille de chaque cellule est de `n bits` ou `n/8 octets` (si *n* est un nombre de bits multiples de 8).

3. Calcul de la capacité totale de la mémoire

La capacité totale de la mémoire est le produit du nombre d'adresses et de la taille de chaque cellule de mémoire.

Si la largeur du bus de données est en bits (n bits) :

$$\text{Capacité mémoire (en bits)} = 2^m \times n$$

Si la largeur du bus de données est en octets (n bits = 8 bits = 1 octet) :

$$\text{Capacité mémoire (en octets)} = 2^m \times n/8$$

Exemple1.13

Si nous avons un bus d'adresse de 16 bits (A_0 à A15) et un bus de données de 8 bits (D_0 à D_7) (soit une mémoire de 1 octet par adresse), la capacité totale sera :

- Nombre d'adresses = 2^{16}=65,536 adresses.
- Taille de chaque cellule = 1 octet (8 bits).
- Capacité mémoire = 65,536×1 octet = 65,536 octets = 64 Ko.

Exemple1.14

Si nous avons un bus d'adresse de **20 bits (A0 à A19)** et un bus de données de **16 bits (D0 à D15)** (soit 2 octets par adresse), la capacité totale sera :

- Nombre d'adresses = 2^{20}=1, 048,576 adresses.
- Taille de chaque cellule = 2 octets (16 bits).
- Capacité mémoire = 1, 048,576×2 octets = **2 Mo**.

Exemple 1.15

Un microprocesseur doit lire une valeur stockée dans la mémoire RAM.
Expliquez l'interaction entre le microprocesseur et la mémoire RAM lorsque le microprocesseur lit une donnée stockée en mémoire.

Solution

❖ **Interaction de la mémoire RAM avec le microprocesseur**

Dans cette partie, nous examinons l'interaction entre le microprocesseur et la mémoire en prenant comme exemple la RAM. Considérons les principales étapes de cette interaction en mode lecture (Voir figure1.16) :

1- Le microprocesseur place l'adresse de la donnée souhaitée sur le bus d'adresse.
2- Il active ensuite le signal de lecture ("MEMORY READ", par exemple) pour indiquer à la RAM qu'une donnée doit être lue.
3- La RAM reçoit l'adresse et le signal de lecture, puis localise la donnée à l'emplacement spécifié.
4- La RAM place la donnée sur le bus de données.
5- Le microprocesseur récupère la donnée sur le bus de données pour exécuter l'instruction ou poursuivre le traitement.

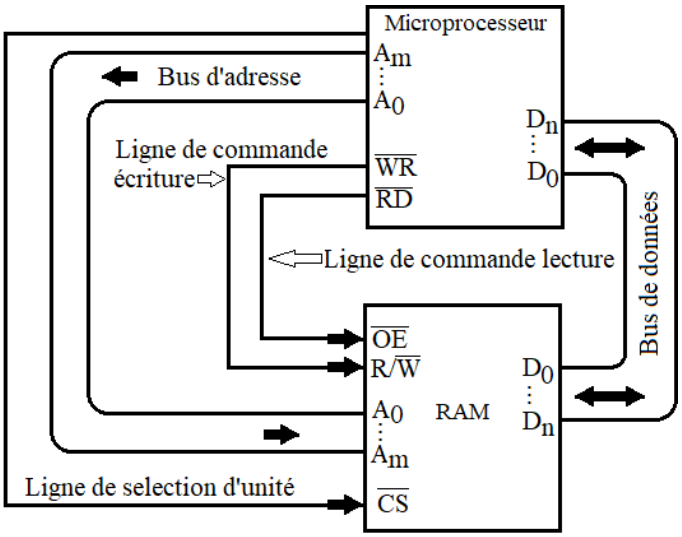


Figure 1.16 *Interaction de la mémoire RAM avec le microprocesseur*

1.5.2.3 Segmentation de la mémoire

La mémoire dans un système basé sur le microprocesseur 8086 a une capacité de 1 Méga-octet, organisée sous forme de mémoire segmentée (figure1.17). Cette capacité est déterminée par la taille du bus d'adresses, qui est constitué de 20 fils dans le cas du microprocesseur 8086. Ainsi, $\mathbf{2^{20}}$ octets = 1 Méga-octet (**1Mo**). Cette organisation permet de déterminer le nombre de segments de mémoire, car la taille d'un segment est déjà définie.

$$Nombre\ total\ de\ segments = \frac{Capacité\ de\ la\ mémoire}{Taille\ d'un\ segment} = \frac{2^{10}\ Kilo-Octets}{64\ Kilo-Octets} = 16$$

La figure suivante décrit un segment dans l'espace de la mémoire principale.

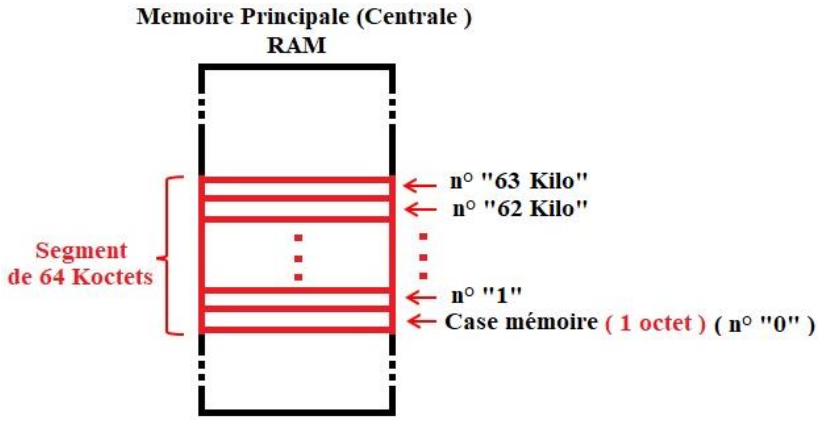


Figure 1.17 *Description d'un segment de mémoire*

Exemple1.16

Calculer en Kilo-Octets la taille d'un segment de mémoire du microprocesseur 8086. Les 4 registres segments permettent d'adresser combien d'octets ?

Solution

Il y a en tout 4 registres segments : `CS, DS, ES, SS`. Chacun de ces registres est de `16 bits.` Donc, en fonction de ces registres segments, on peut avoir 4 segments de mémoire constitué chacun de : 2^{16} octets = $2^{6}.2^{10}$ octets = 64 Kilo-Octets.

Les 4 registres permettent d'adresser : $\sum_{i=1}^{4} \boldsymbol{Tseg\_i}$, où *Tseg* représente la taille d'un segment. Donc,

(*Tseg_1*(CS) + *Tseg_2*(DS) + *Tseg_3*(ES) + *Tseg_4*(SS)) =

4***** 64 Kilo-Octets = 256 Kilo-Octets.

Dans ce cas, la mémoire physique peut être divisée en 16 segments logiques, chacun ayant une taille de 64 Ko. La figure suivante illustre la mémoire principale (physique) segmentée en 16 segments.

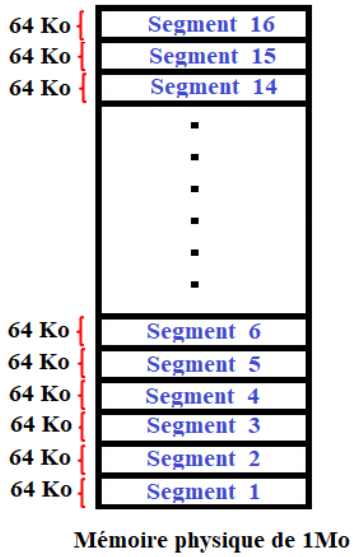


Figure 1.18 *Segmentation de la mémoire Physique*

Parmi les 16 segments de mémoire, seuls 4 peuvent être activés simultanément, car le microprocesseur accède à la mémoire à travers ses 4 registres : CS, DS, ES et SS, qui spécifient les adresses des segments correspondants. Par exemple,

Le `segment 3` peut correspondre à `SS`
Le `segment 4` peut correspondre à `ES`
Le `segment 5` peut correspondre à `DS`

Le `segment 6` peut correspondre à `CS`

❖ `Maintenant qu'en est-il des segments restants ?`

Les segments restants seront utilisés par un autre microprocesseur. C'est le concept de multi-traitement (fonctionnement en mode maximum) qui s'appliquera ici.

❖ `Calcul de l'adresse physique sur le 8086`

Le processeur 8086 utilise deux nombres de 16 bits pour calculer l'adresse réelle en mémoire, appelée `adresse physique`. Ces deux nombres sont :

- Un `segment` : il représente un bloc de mémoire de taille fixe.
- Un `offset` : il donne la position dans ce segment.

❖ `Comment fonctionne le calcul ?`

Pour trouver l'adresse physique, on utilise cette formule :

Adresse Physique= (Segment×16) + Offset

- `Segment` est multiplié par 16 (ou déplacé de 4 positions vers la gauche).
- Ensuite, on `ajoute` l'offset.

Cela nous donne une adresse de 20 bits, ce qui permet au 8086 d'adresser jusqu'à 1 Mo de mémoire.

Exemple1.17

Supposons qu'on ait : `Segment` = 1000 (en hexadécimal), `Offset` = 0010 (en hexadécimal)

Étapes

1. `Convertir le segment` : 1000×16 = 10000
2. `Ajouter l'offset` : 10000 + 0010 = 10010

Résultat : l'adresse physique est donc 10010 en hexadécimal.

Exercice 1.4

Le microprocesseur 8086 utilise une segmentation de mémoire pour accéder à des adresses physiques. Les adresses physiques sont calculées à l'aide des registres de segments et d'un décalage (offset) selon la formule suivante :

Adresse physique= (Registre segment×16) + Offset

1. Si le registre **CS** (Code Segment) contient la valeur **0x1234** et que l'offset est **0x002F**, calculez l'adresse physique de l'instruction.

2. Si le registre **DS** (Data Segment) contient **0x0A00** et l'offset est **0x0100**, calculez l'adresse physique de la donnée.

3. Expliquez la différence d'utilisation entre les registres **CS**, **DS**, **SS** dans le 8086.

Solution

1. Calcul de l'adresse physique avec CS et offset

Adresse physique= (Registre segment×16) + Offset

Données : CS=0X1234, Offset=0X002F

Calcul :
Adresse physique= (0X1234×16) + 0X002F, et 0X1234×16=0X12340, donc Adresse physique= 0X12340 + 0X002F = 0X1236F

Ainsi, l'adresse physique est **0x1236F**.

2. Calcul de l'adresse physique avec DS et offset

Données : DS=0x0A00, Offset=0x0100

Calcul : Adresse physique = (0x0A00×16) + 0x0100 = 0x0A100

3. Différence d'utilisation des registres segments

- `CS (Code Segment)` : Utilisé pour stocker le segment contenant les instructions à exécuter. L'adresse physique de l'instruction est déterminée par CS et le pointeur d'instruction IP.

- `DS (Data Segment)` : Contient les données utilisées par les programmes. La majorité des opérations de lecture/écriture de données utilisent DS avec un offset.

- `SS (Stack Segment)` : Définit le segment utilisé pour la pile. Les registres SS et SP (Stack Pointer) travaillent ensemble pour gérer les opérations de pile comme les appels de sous-programmes et les retours.

Exercice1.5

Un microprocesseur 8086 doit écrire une valeur dans la mémoire RAM.

1. Le registre **DS** contient la valeur 0x2000, et l'offset est 0x0010. Calculez l'adresse physique où la donnée sera écrite.
2. Décrivez les étapes nécessaires pour écrire la valeur 0x5A à cette adresse.

Solution

```
1.   Calcul de l'adresse
```

Le 8086 combine le registre **DS** (0x2000) et l'offset (0x0010) pour déterminer l'adresse physique 0x20010.

```
2.  les étapes nécessaires pour écrire la valeur 0x5A à cette
  adresse
```

- La valeur 0x5A est chargée dans un registre interne, comme **AL**.
- L'adresse physique 0x20010 est placée sur le bus d'adresse.
- La donnée 0x5A est placée sur le bus de données.
- Le microprocesseur active le signal **WRITE** pour déclencher l'écriture dans la RAM.
- La RAM reçoit et stocke 0x5A à l'adresse physique 0x20010.

Chapitre 2 : La programmation en assembleur

Objectifs

À l'issue de ce chapitre, vous serez capable de :

Comprendre les concepts de base de la programmation en assembleur.

Connaître le jeu d'instructions du processeur et leur utilisation.

Appliquer une méthode simple pour écrire des programmes en assembleur pour le microprocesseur 8086.

2.1 Introduction

Après avoir étudié l'architecture et le fonctionnement du microprocesseur 8086 dans le chapitre précédent, ce chapitre se concentre sur la programmation en assembleur. Pour les étudiants en master électrotechnique, il est essentiel de comprendre comment les systèmes de contrôle et les performances des dispositifs électroniques sont assurés par l'interaction avec des circuits programmables. Ce chapitre vous introduira d'abord aux bases de la programmation en assembleur, un langage de bas niveau permettant de manipuler directement les registres et les instructions du microprocesseur. Vous apprendrez à utiliser le jeu d'instructions du 8086 pour écrire des programmes simples, notamment pour des applications liées à la gestion des interfaces d'entrées et de sorties, dans le but de commander des périphériques.

2.2 Généralités

Comprendre le fonctionnement interne des microprocesseurs est une étape fondamentale pour la programmation en assembleur. En se focalisant sur le microprocesseur `8086`, ce chapitre explore les concepts essentiels du langage de bas niveau appelé `assembleur`, qui permet d'interagir directement avec le matériel en manipulant les registres.

Ci-dessous se trouve un classement des niveaux de langages de programmation :

Niveau	Langage (Exemple)	Description
Plus bas	**machine**	Constitué de code binaire (0 et 1) compris directement par le microprocesseur. Très rapide, mais difficile à lire et à écrire pour les utilisateurs.
Bas	**assembleur**	utilise des mots simples (mnémoniques, comme `MOV`) pour représenter les instructions en binaire. Il offre un contrôle précis du matériel mais reste complexe à utiliser.
Haut avec accès bas niveau	**C, C++**	souvent classés en haut niveau, car ils sont bien plus proches du langage humain, mais ils permettent aussi d'accéder directement à la mémoire et au matériel que l'assembleur.
Haut	**Python, Java**	conçus pour être plus faciles à lire et à écrire.
Très haut ou spécialisés	**SQL, HTML**	ne permettent pas de contrôler directement le matériel et sont utilisés dans des contextes spécifiques, comme gérer des bases de données (`SQL`) ou créer des pages web (`HTML`).

2.2.1 Le langage assembleur

Le langage assembleur (ou langage d'assemblage) est un type de langage de programmation qui permet de communiquer directement avec le matériel d'un ordinateur, comme le microprocesseur. Au lieu d'écrire des 0 et des 1, l'assembleur utilise des mots simples (appelés mnémoniques) pour représenter des instructions.

Exemple2.1

Une instruction comme `MOV` signifie déplacer des données d'un endroit à un autre.

Exemple2.2

Un autre exemple simple pour vous aider à comprendre le langage assembleur :
Imaginons que nous souhaitons déplacer une valeur (un nombre) d'un registre à un autre. En langage machine, cela se fait avec des 0 et des 1. En assembleur, ce même processus est exprimé avec des `mnémoniques` plus compréhensibles.

```
MOV AX, 5      ; Déplace la valeur 5 dans le registre AX
MOV BX, AX     ; Déplace la valeur de AX (qui est 5) dans le registre BX
```

Exemple2.3

Supposons que nous voulons additionner deux nombres

```
MOV AX, 3      ; Déplace 3 dans le registre AX
MOV BX, 4      ; Déplace 4 dans le registre BX
ADD AX, BX     ; Additionne la valeur de BX (4) à AX (3), donc AX devient 7
```

Ce code assembleur peut être exprimé en langage C :

```c
#include <stdio.h>

int main() {
   int AX = 3;  // Déclare une variable AX et lui attribue la valeur 3
   int BX = 4;  // Déclare une variable BX et lui attribue la valeur 4

   AX = AX + BX;  // Additionne la valeur de BX à AX, donc AX devient 7

   printf("La valeur de AX est : %d\n", AX);  // Affiche la valeur de AX

   return 0;
}
```

➢ **Explication**

- En assembleur, on utilise les registres comme `AX` et `BX`, mais en C, on déclare des variables normales.
- L'instruction `MOV` est utilisée pour assigner des valeurs aux registres en assembleur, ce qui correspond à l'initialisation des variables en C.
- L'instruction `ADD` est utilisée pour additionner les valeurs des registres en assembleur, ce qui est équivalent à l'opération `AX = AX + BX` en C.

Ces exemples montrent que, bien que le langage assembleur utilise des instructions simples comme `MOV` ou `ADD`, il est très proche du fonctionnement du microprocesseur. Les programmeurs utilisant l'assembleur peuvent contrôler précisément le microprocesseur, mais cela demande de bien connaître les registres et les instructions du microprocesseur.

2.2.2 Format d'une instruction du 8086

Le format d'une instruction en langage assembleur 8086 est généralement structuré comme suit :

```
[CodeOp] [Opérande(s)]   ; [Commentaire]
```

Le `CodeOp` est le code d'opération qui spécifie l'action à effectuer. Cela peut être des opérations comme `MOV (déplacer)`, `ADD (additionner)`, `SUB (soustraire)`, etc.

Les `Opérandes` sont les valeurs ou les registres sur lesquels l'instruction agit. Il peut s'agir de :

- Des `registres` (comme AX, BX, CX, etc.)
- Des `valeurs immédiates` (comme un nombre, par exemple 5)
- Des `adresses mémoire` (par exemple [BX] ou une valeur en mémoire)

Le `commentaire` est ajouté après un `point-virgule` **(;)** et permet d'expliquer ce que fait l'instruction. Le processeur ignore cette partie.

2.2.3 Directives utilisées dans le code assembleur du 8086

En assembleur pour le microprocesseur 8086, les **directives** aident à organiser le code, définir des constantes, et gérer la mémoire. Elles ne sont pas exécutées par le processeur, mais elles servent à indiquer à l'assembleur comment structurer le programme. Quelques-unes des directives les plus couramment utilisées sont :

```
EQU, ORG, DB, DW, DD, SEGMENT, ENDS, ASSUME, END, PROC, ENDP,
et LABEL.
```

Dans notre cas, nous considérons les directives suivantes : `EQU, ORG, DB et DW`

- **Directive EQU**

La directive `EQU` (qui signifie "`EQUate`", en anglais) en assembleur est utilisée pour définir une `constante`. Au lieu d'utiliser des valeurs fixes (nombres) dans tout le programme, on peut utiliser un nom associé à une valeur grâce à `EQU`**.**

Syntaxe : `[NOM] EQU [valeur]`

Exemple2.4

```
max_value  EQU  100   ; Définit MAX_VALUE comme 100
MOV  AX, max_value   ; Charge 100 dans le registre AX
```

- **Directive** `ORG`

La directive `ORG` (pour "`ORiGin`") est utilisée en assembleur pour définir l'adresse de départ d'un segment de code ou de données en mémoire. Cela indique à l'assembleur où placer le code ou les données en mémoire.

Syntaxe : `ORG adresse`

Exemple2.5

```
ORG 100h        ; Place le code à l'adresse 100h
MOV AX, 0       ; Instruction commençant à l'adresse 100h
```

- **Directive** `DB`

La directive `DB` (pour "`Define Byte`" ou "Définir un octet") en assembleur est utilisée pour `déclarer une variable` ou `allouer de la mémoire` pour stocker des données d'un octet (8 bits).

Syntaxe : `nom DB valeur`
`nom` : Le nom de la variable (facultatif).

Exemple2.6

```
Num  DB  10              ; Définit Num avec la valeur 10 (en décimal)
Text DB 'Assembleur'     ; Définit Text avec la chaîne "Assembleur" en ASCII
```

Dans cet exemple :

- `Num` est une variable initialisée avec la valeur **10**.
- `Text` est une variable contenant les caractères '**Assembleur**', stockés en ASCII (chaque caractère prend un octet).

- **Directive** `DW`

La directive `DW` (pour "`Define Word`" ou "Définir un mot") en assembleur est utilisée pour **déclarer une variable** ou **réserver de la mémoire** pour stocker des données de deux octets (16 bits), ce qu'on appelle un "mot" en assembleur.

Syntaxe : `nom DW valeur`
`nom` : Le nom de la variable (facultatif).

Exemple2.7

```
Count  DW 1000       ; Définit COUNT avec la valeur 1000 (en décimal)
List DW 1, 2, 3, 4   ; Définit une liste de mots : 1, 2, 3, et 4
```

2.3 Jeu d'instructions du microprocesseur 8086

Les instructions sont classées sur la base des fonctions qu'elles remplissent. Elles sont classées dans les principaux types suivants :

1. Instruction de transfert de données
2. Instructions arithmétiques
3. Instructions logiques et de manipulation de bits
4. Instruction de manipulation de la chaîne de caractères

5. Instructions de contrôle de programme
6. Instructions de contrôle du microprocesseur
7. Instructions d'itération et d'interruption

➢ **Remarque**

Dans la suite de ce cours, nous n'aborderons pas toutes les instructions du microprocesseur 8086. Nous nous concentrerons sur les instructions les plus couramment utilisées et celles qui vous seront utiles pour écrire des programmes simples et pertinents durant les séances de Travaux Pratiques. Cette approche vise à vous familiariser rapidement avec les bases de la programmation en assembleur tout en restant orientée vers des applications concrètes.

2.3.1 Instruction de transfert

Toutes les instructions qui effectuent le déplacement des données entrent dans cette catégorie.

- La source peut être un registre, un emplacement mémoire, un port, etc.
- la destination un registre, un emplacement mémoire ou un port.

Ces instructions se répartissent en quatre groupes

a- Instructions de transfert ou de chargement sur registres

MOV : transférer l'opérande source à l'opérande destination
XCHG : échanger l'opérande source avec l'opérande destination
PUSH : sauvegarder les donner dans la pile
POP : restituer les données de la pile
XLAT : traduire une table à travers les registres AL et BX

b- Instructions de transfert d'adresses

LEA : charger l'adresse effective
LDS : charger un registre de base ou d'index et le registre DS à partir de la mémoire
LES : charger un registre de base ou d'index et le registre ES à partir de la mémoire

c- Instructions de transfert et de chargement des indicateurs (Flags)

LAHF : placer l'octet de poids faible du registre d'état dans AH
SAHF : placer le contenu de AH dans l'octet de poids faible du registre d'état
PUSHF : placer dans la pile le registre d'état totalement
POPS : charger le contenu de la pile pointée par SP dans le registre d'état et incrémenter SP de deux unitées

d- Instructions d'entrées et de sorties

IN : lire un port d'entrées et de sorties
OUT : écrire dans un port d'entrées et de sorties

2.3.2 Instruction arithmétiques du microprocesseur 8086

a- Addition

ADD : Additionner
ADC : Additionner avec la retenue
INC : Incrémenter
AAA : Ajuster ASCII après l'addition
DAA : Ajuster décimal après l'addition

b- Soustraction

SUB : Soustraire
SBB : Soustraire avec la retenue
DEC : Décrémenter
NEG : Multiplier par (-1) ou le complément à 2
CMP : Comparer deux opérandes
AAS : Ajuster ASCII pour la soustraction
DAS : Ajuster décimal pour la soustraction

c- Multiplication

MUL : Multiplier
IMUL : Multiplier entièrement
AAM : Ajuster ASCII pour la multiplication

d- Division

DIV : Diviser
IDIV : Diviser entièrement
AAD : Ajuster ASCII pour la division
CBW : Convertir un octet en mot
CWD : Convertir un mot en double mot

2.3.3 Les instructions de saut de programme

a- Branchements inconditionnels

CALL : appeler un sous-programme
RET : retour d'un sous-programme
JMP : saut

b- Branchements conditionnels (arithmétiques non signé)

JA : saut si supérieur
JNBE : saut si non inférieur ou égal
JAE : saut si supérieur ou égal
JNB : saut si non inférieur
JB : saut si inférieur
JNAE : saut si non supérieur ni égal
JBE : saut si inférieur ou égal
JNA : saut si non supérieur

c- Branchements conditionnels (arithmétiques signé)

JG :	saut si plus grand
JNLE :	saut si pas inférieur ni égal
JGE :	saut si plus grand ou égal
JNL :	saut si pas inférieur
JL :	saut si moins que
JNGE :	saut si pas plus grand ni égal
JLE :	saut si moins que ou égal
JNG :	saut si pas plus grand

d- Branchements conditionnels (flags)

JC :	saut si retenue (carry=1)
JE :	saut si égal
JZ :	saut si zéro (zero=1)
JNC :	saut si pas de retenue (carry=0)
JNE :	saut si non égal
JNZ :	saut si non zéro (zero=0)
JNO :	saut si pas de débordement
JNP :	saut si pas de parité
JPO :	saut si parité impaire
JNS :	saut si pas de signe
JO :	saut si débordement
JP :	saut si parité
JPE :	saut si parité paire
JS :	saut si signe (négatif)

e- Boucles

LOOP :	boucle
LOOPE :	boucle si égal
LOOPZ :	boucle si zéro
LOOPNE :	boucle si différent
LOOPNZ :	boucle si différent de zéro
JCXZ :	Branchement si CX=0

f- Interruptions

INT :	interruption
INTO :	Interruption si débordement
IRET :	Retour d'interruption

2.3.4 Etude de quelques instructions

❖ Instructions de transfert

Les instructions étudiées dans ce contexte sont les suivantes :

`MOV, XCHG, PUSH, POP, LAHF, SAHF`

Format : `MOV Destination, Source`

- Cette instruction permet de copier l'opérande source dans l'opérande destination
- Cette instruction peut s'appliquer pour différentes sources et différentes destinations

MOV Registre1, Registre2 ; Copier le contenu d'un registre dans un registre
MOV Registre, Mémoire ; Copier le contenu d'une case mémoire dans un registre
MOV Mémoire, Registre ; Copier le contenu d'un registre dans la mémoire
MOV Registre, Immédiate ; Copier une constante dans un registre
MOV Mémoire, Immédiate ; Copier une constante dans la mémoire
MOV Mémoire, Mémoire ; `Cette instruction n'est pas opérationnelle`

- Cas de : `MOV Registre, Immédiate`

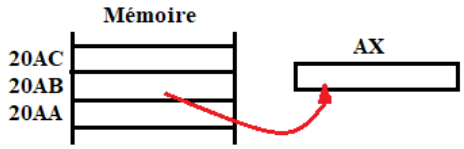
MOV AX, 10FB ; la valeur 10FBH est transférée au registre AX
Cette écriture peut être réécrite de la manière suivante : AX ⟵ 10FBH, ou bien : (AX) = 10FBH.
Note : ( ) : veut dire le contenu

- Cas de: `MOV Registre1, Registre2`

MOV AX, 10FB
MOV BX, AX ; BX ⟵ (AX) ; (BX) = 10FBH

- Cas de: `MOV Registre, Mémoire`

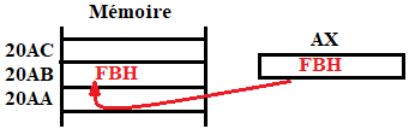
MOV AX, [20AB] ; le contenu de l'adresse mémoire est transférée au registre AX.
; L'adresse mémoire dans ce cas est : 20ABH.



Ou bien : AX ⟵ ([20ABH])
Note : [ ] : veut dire la case mémoire

- Cas de : `MOV Mémoire, Registre`

MOV AL, FB
MOV [20AB], AL



Format : `XCHG Destination, Source`

Cette instruction permet d'échanger l'opérande Source avec l'opérande Destination.

XCHG Registre1, Registre2 ; Copier le contenu du registre1 avec celui du registre2
XCHG Registre, [Adresse] ; Copier le contenu de l'adresse mémoire avec celui du registre
XCHG [Adresse], Registre ; Copier le contenu du registre avec celui de l'adresse mémoire
XCHG [Adresse1], [Adresse2] ; **Cette instruction n'est pas opérationnelle**

- Cas de: `XCHG Registre1, Registre2`

MOV AX, 12FB ; AX ⟵ 12FBh; (AX)= 12FBh
MOV BX, 42C2 ; BX ⟵ 42C2h ; (BX) = 42C2h
XCHG AX, BX ; (AX)= 42C2h et (BX) = 12FBh

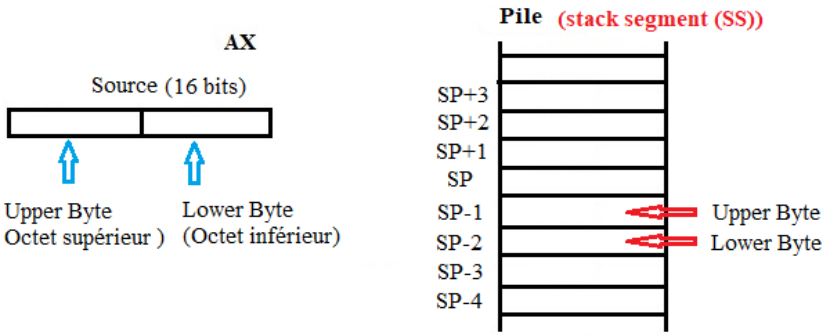
- Cas de: `XCHG [adresse], Registre`

MOV AX, 2EFB ; (AX)= 2EFBh
MOV [1007], C23B ; [1007] ⟵ 3Bh; [1008] ⟵ C2h; ([0007]) = 3Bh
XCHG [1007], AX ; ([1007]) = FBh, ([1008]) = 2Eh et (AX) =C23Bh

Format : `PUSH Source`

Cette instruction fonctionne seulement avec la pile, et une pile n'est qu'un bloc de mémoire qui peut être utilisé pour stocker temporairement les données.
Ainsi, pour s'adresser à la pile, le seul registre qui devrait être utilisé est le stack pointeur SP.

SS : SP ; ce qui signifie que : les adresses du registre SP permettent de se déplacer dans la pile (stack segment). Voir la figure donnée ci-dessous.



`PUSH Source`

- `SP-1` ; on sauvegarde le `Upper Byte` de la source dans la pile
- `SP-2` ; on sauvegarde le `Lower Byte` de la source dans la pile

Ainsi, la nouvelle valeur du registre SP est donc SP-2.

Exemple2.8

MOV AX, 1234
MOV BX, 5678
MOV CX, 9ABC
PUSH AX
PUSH BX
PUSH CX

Dans ce cas, il faut se baser sur le principe suivant :

- à chaque instruction PUSH correspond : SP-1 et SP-2

Supposons que (SS)=1111H et (SP) =2222H

`PUSH AX`

- `SP-1:` 2222-1=2221h; on sauvegarde à cette adresse le (AH)=12h
- `SP-2` : 2222-2=2220h; on sauvegarde à cette adresse le (AL)=34h

Dans ce cas, la nouvelle valeur du registre SP est 2220H

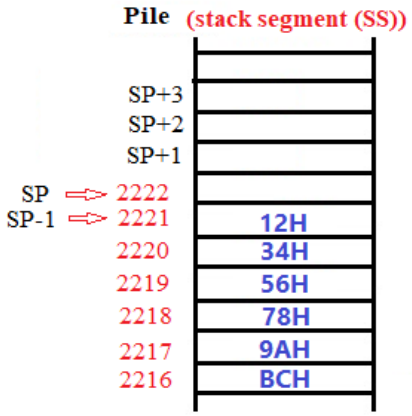
`PUSH BX`

- `SP-1:` 2220-1=2219h; on sauvegarde à cette adresse le (BH)=56h
- `SP-2` : 2220-2=2218h; on sauvegarde à cette adresse le (BL)=78h

Pour la suite, le nouveau (SP) = 2218H

`PUSH CX`

- `SP-1:` 2218-1=2217h; on sauvegarde à cette adresse le (CH)=9Ah
- `SP-2` : 2218-2=2216h; on sauvegarde à cette adresse le (CL)=BCh



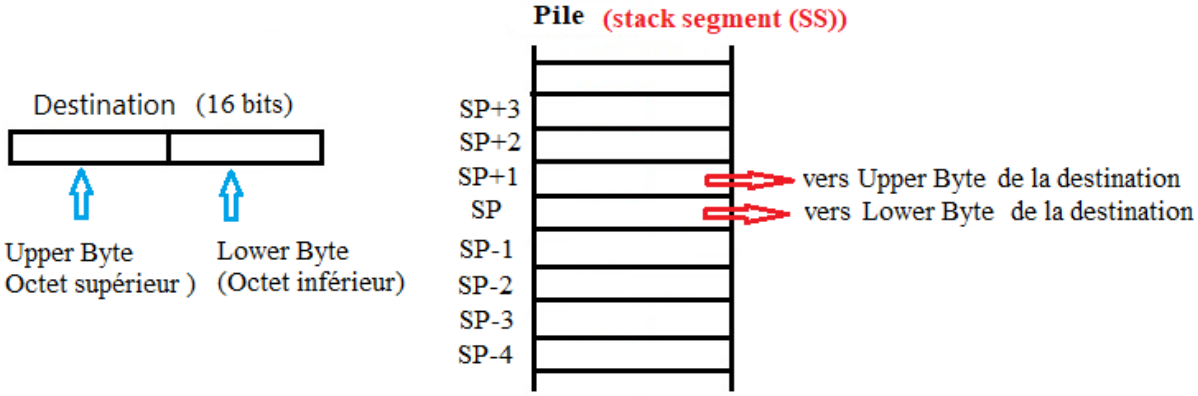
Format `POP Destination`

Cette instruction permet de transférer un mot (16 bits) depuis la pile, adressé par le registre SP, vers la destination spécifiée. Ainsi, l'instruction `POP` fonctionne exclusivement avec la pile en utilisant le registre `SP`.

`POP Destination`

- `SP` ; on transfert l'octet contenu dans la pile vers `Lower Byte` de la destination
- `SP-1` ; on transfert l'octet contenu dans la pile vers `Upper Byte` de la destination

et le nouveau (SP) = SP-2



Exemple2.9

MOV AX, FA51
MOV BX, E940
PUSH AX
PUSH BX
POP CX
POP DX

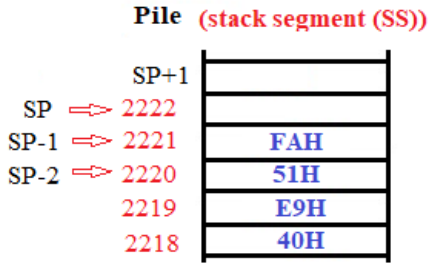
Dans ce cas, la récupération des octets depuis la pile avec l'instruction `POP` repose sur : `SP et SP+1`. Supposons que le registre `SP` ait la valeur 2222h.

`PUSH AX`

- `SP-1`: 2222-1=2221h; on sauvegarde à cette adresse le (AH)=FAh
- `SP-2` : 2222-2=2220h; on sauvegarde à cette adresse le (AL)=51h

`PUSH BX`

- `SP-1`: 2220-1=2219h; on sauvegarde à cette adresse le (BH)=E9h
- `SP-2` : 2220-2=2218h; on sauvegarde à cette adresse le (BL)=40h



A ce niveau du traitement le registre SP contient l'adresse 2218H

`POP CX`

- `SP`: 2218; on transfère le contenu de cette adresse au Lower Byte du registre CX
- `SP+1` : 2218+1=2219h; on transfère le contenu de cette adresse au Upper Byte du registre CX

Le nouveau (SP)=2219H+1 =2220H

`POP DX`

- `SP`: 2220; on transfère le contenu de cette adresse au Lower Byte du registre DX
- `SP+1` : 2220+1=2221h; on transfère le contenu de cette adresse au Upper Byte du registre DX

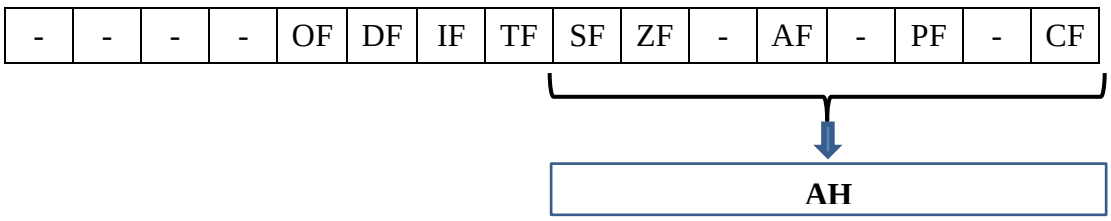
Le nouveau (SP)=2221H+1 =2222H

Format `LAHF`

(Load 8086 Flags into AH register)

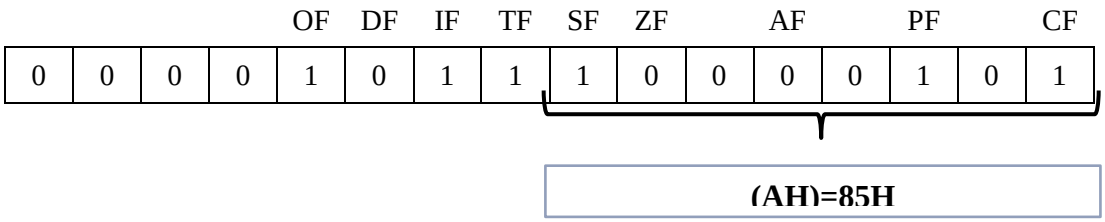
Cette instruction permet de transférer les 8 bits les plus à droite du registre des drapeaux dans le registre AH

Registre d'état (registre drapeaux)



Exemple2.10

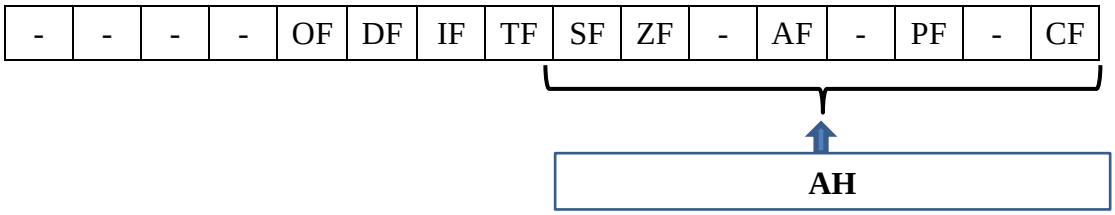
On suppose que le contenu du registre drapeaux est comme suit :



MOV AX, FA12 ; (AX) =FA12h ; (AL) =12h et (AH) =FAh
LAHF ; (AH) =85H
INT 21

Format `SAHF`

Cette instruction permet de transférer le contenu registre AH dans l'octet de poids faible du registre des drapeaux.



Exemple2.11

```
MOV AX, E901 ; (AX) =E901H ;  (AL) =01H  et  (AH) =E9H
SAHF ;   (AH) =E9H ;    SF=1 ; ZF=1 ; AF=0 ;  PF=0 ; CF=1
INT 21
```

❖ Instructions arithmétiques

Les instructions concernées sont les suivantes :

`ADD, ADC, INC, SUB, SBB, DEC, CMP, MUL, DIV`

Ces instructions sont utilisées pour les calculs arithmétiques et peuvent être appliquées avec des nombres signés ou non signés.

- Un `nombre signé` est un nombre qui peut être `positif ou négatif`. Il utilise le premier bit pour indiquer le signe : 0 pour positif, 1 pour négatif. Par exemple, un nombre signé de 8 bits peut représenter des valeurs de **-128 à +127**.

- Un `nombre non signé` est un nombre qui est `toujours positif (ou nul)`. Il utilise tous les bits pour représenter la valeur, sans indication de signe. Un nombre non signé de 8 bits peut représenter des valeurs de **0 à 255**.

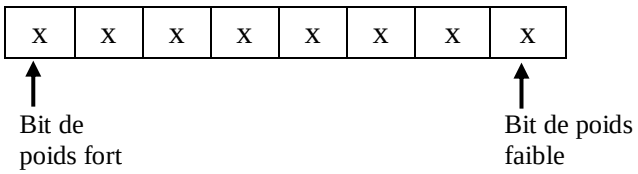
Pour un nombre de 16 bits, les valeurs maximales et minimales sont les suivantes :

- `Nombre signé de 16 bits` : peut représenter des valeurs de **-32 768 à +32 767**.
- `Nombre non signé de 16 bits` : peut représenter des valeurs de **0 à 65 535**.

Le bit de poids le plus fort (le plus à gauche) est le bit de signe. Si ce bit est mis à « 0 » le nombre est positif, s'il est mis à « 1 », le nombre est négatif.

Exemple2.12

Cas d'une longueur de 8 bits



Addition

Format `ADD Destination, Source`

- Cette instruction est utilisée pour effectuer une addition
- Il est obligatoire d'avoir la même taille de source et de destination
- Après addition, le résultat sera stocké uniquement dans la destination
- La destination doit être uniquement un registre (8 bits ou 16 bits)
- La source peut être un registre, des données ou un emplacement mémoire

Destination ⟵ (Destination) + donnée
Destination ⟵ (Destination) + (registre)
Destination ⟵ (Destination) + (emplacement mémoire)

Exemple2.13

```
ADD  AL,  AE      ; AL ⟵ (AL) + AEH
ADD  BX, 10FB     ; BX ⟵ (BX) + 10FBh
ADD  CX, BX       ; CX ⟵ (CX) + (BX)
ADD AH, AL        ; AH ⟵ (AH) + (AL)
ADD  AX, [1FFF]   ; AX ⟵ (AX) + ([1FFFh])
ADD  CL, [2FEE]   ; CL ⟵ (CL) + ([2FEEh])
```

Format `ADC Destination, Source`

Destination ⟵ (Destination) + donnée + retenue
Destination ⟵ (Destination) + (registre) + retenue
Destination ⟵ (Destination) + (emplacement mémoire) + retenue

Exemple2.14

Dans cet exemple, on suppose que le bit d'état CF contient « 1 »

```
MOV AL, FF     ; (AL)=FFh
ADC AL, 01     ; (AL)=(AL) + 01h + CF = FFh + 01h + 1 = 1
INT 21
```

Format `INC Destination`

- Cette instruction ajoute 1 à la destination.

- La destination peut être un registre ou un emplacement mémoire.
- Cette instruction n'affecte pas l'indicateur de retenus CF.

(Registre) ⟵ (Registre) + 1
(Emplacement mémoire) ⟵ (Emplacement mémoire) + 1

Exemple2.15

```
MOV  AX, 111E      ; (AX)=111Eh
INC  AX            ; (AX)=(AX) + 1 = 111Eh + 1 = 111Fh
INC  [0003]        ; ([0003h])= ([0003h]) +1 ;  le contenu de la case mémoire d'adresse 0003h est
                                                incrémenté de 1
```

Soustraction

Format `SUB  Destination, Source`
(Substrate)

- Cette instruction est utilisée pour effectuer une soustraction
- Instruction similaire à ADD

(Destination) ⟵ (Destination) - donnée
(Destination) ⟵ (Destination) - (registre)
(Destination) ⟵ (Destination) - (emplacement mémoire)

Exemple2.16

```
MOV  AL,  AE       ; (AL) ⟵  AEH
SUB  AL, B2        ; (AL) ⟵  (AL) – B2H
```

Format `SBB  Destination, Source`
(subtract with borrow)

Cette instruction permet de soustraire à la fois un opérande source et la valeur de l'indicateur Carry CF d'un opérande de destination.

(Destination) ⟵ (Destination) - donnée - CF
(Destination) ⟵ (Destination) - (registre) - CF
(Destination) ⟵ (Destination) - (emplacement mémoire) - CF

Exemple2.17

```
MOV  AL,  F0       ; (AL) ⟵  F0H
SBB  AL, 04        ; (AL) ⟵  (AL) – 04H – CF
Si CF =0,  alors (AL) = F0H – 04H – 0 = ECH
Si CF =1,  alors (AL) = F0H – 04H – 1 = EBH
```

Format `DEC Destination`

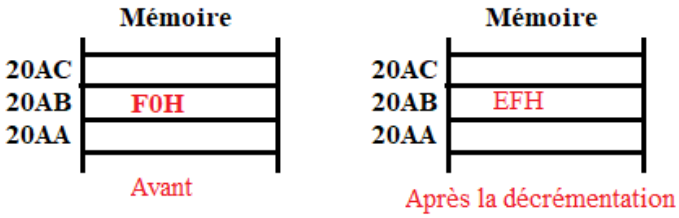
(decrement)

Cette instruction permet de décrémenter le contenu de la destination. Cette destination peut être un registre ou un emplacement mémoire.

Exemple2.18

```
MOV AL, F0      ; (AL) ←  F0H
DEC AL          ; (AL) ←  (AL) – 1 =F0H – 1=EFH

DEC [20AB]      ; [20AB] ←  ([20AB]) – 1= F0H – 1=EFH
```



Format `NEG Destination`

(negate)

Cette instruction permet de trouver le complément à 2

(Destination) ← Complément à 1 de (Destination) + 1

Exemple2.19

```
MOV AX, ABCD ;  (AX)= ABCDH
NEG  AX ;  not (ABCDH) +1 = 5432H+1= 5433H
INT 21
```

Format `CMP Destination, Source`

Cette instruction permet de comparer des octets, mots et doubles mots.
`Source` : peut-être un registre, un emplacement mémoire ou une constante
`Destination` : peut-être seulement un registre, un emplacement mémoire

La comparaison permet d'effectuer : **(**`destination`**) - (**`source`**)**

Si **(**`destination`**) = (**`source`**)** alors ZF= **1**
Si **(**`destination`**) < (**`source`**)** alors **ZF= 0** et **CF=1**
Si **(**`destination`**) > (**`source`**)** alors **ZF= 0** et **CF=0**

Note : voir le cours déjà fait sur le registre d'état

Exemple2.20

MOV AX, 0012 ; (AX)= 0012H
CMP AX, 0300 ; 0012H – 0300H ; 0012H < 0300H ; **ZF= 0** et **CF=1**
INT 21

Multiplication

Multiplication non signée dans les processeurs 8086, une des données doit être stockée dans l'accumulateur et une autre donnée peut être stockée dans le registre/mémoire. Après multiplication, le produit sera dans les registres AX et DX ou AL et AH.

Format `MUL source`

Signification

		Extention de la multiplication sur :
(AX)*Registre (16 bits)	→ (AX)	DX
(AX)*Memory	→ (AX)	DX
(AL)*Registre (8 bits)	→ (AL)	AH
(AL)*Memory	→ (AL)	AH

Multiplicande AX
X Multiplicateur (reg.16/8 bits)/mémoire

DX	AX

Multiplicande AL
X Multiplicateur (reg. 8 bits)/mémoire

AH	AL

Exemple2.21

Écrire un programme en langage assembleur pour multiplier deux nombres de 16 bits

DE29h*CF61h

Dans ce cas, une des données doit être stockée dans le registre AX et une autre donnée peut être stockée dans le registre/mémoire.

➢ Considérons le cas de la multiplication entre deux registres

				D	E	2	9	AX
			*	C	F	6	1	BX
				D	E	2	9	
		5	3	4	F	6		
	D	0	4	6	7			
A	6	9	E	C				
B	**3**	**F**	**7**	**5**	**4**	**8**	**9**	
	DX				AX			

MOV AX, DE29
MOV BX, CF61
MUL BX

- Considérons le cas de la multiplication entre AX et un nombre qui est le contenu de la mémoire

Mémoire

20FE	
20FF	
2100	**29**
2101	**DE**
2102	**61**
2103	**CF**
2104	**89**
2105	**54**
2106	**F7**
2107	**B3**
2108	

MOV SI, 2100	Charger l'adresse des données dans le registre SI
MOV AX, [SI]	Transférer le premier nombre dans le registre AX
MOV BX, [SI+2]	Transférer le deuxième nombre dans le registre BX
MUL BX	Multiplier les contenus de AX et de BX. le produit sera AX et DX
MOV [SI+4], AX	Enregistrer le produit (AX et DX) en mémoire
MOV [SI+6], DX	

Division

Format

```
DIV    source
```

Dividende (AX)	Diviseur (Source)
Reste(R)	Quotient(Q) (Résultat)

Signification

Si la source = 8 bits la division **AX/Source** est réalisée alors R est chargé dans AH et Q est chargé dans AL.

Si la source = 16 bits la division **(DX, AX) /Source** est réalisée alors R est chargé dans DX et Q est chargé dans AX.

La source peut être registre/mémoire

Exemple2.22

Écrire un programme en langage assembleur qui permet d'effectuer la division entre 2 nombres.

a) 0101h/2h b) 2001h/1000h

Source (8bits) Source (16 bits)

(AX) = 0101h	(BL) = 2h
(AH)=1h	(AL)=80h

(DX, AX) = 2001h	(BX) = 1000h
(DX)=1h	(AX)=2h

```
MOV  AX, 0101
MOV BL, 02
DIV  BL
```

```
MOV  DX ,0000
MOV  AX, 2001
MOV BX, 1000
DIV  BX
```

❖ Instructions de sauts de programme

Elles permettent de faire des sauts dans l'exécution d'un programme (rupture de séquence)

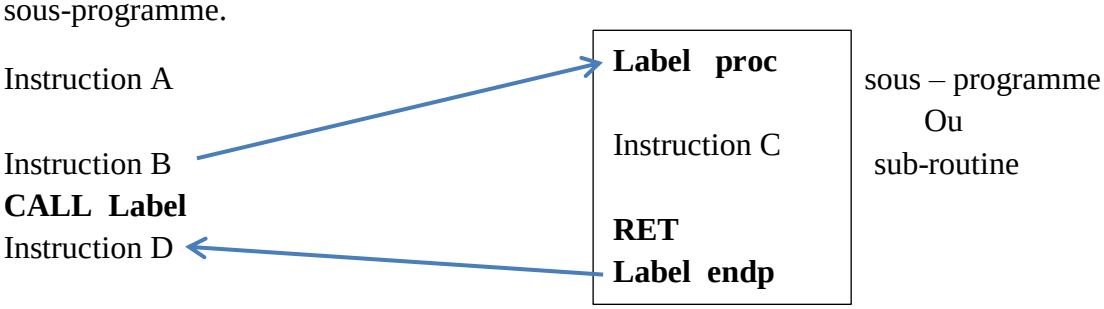
Remarque :

Ces instructions n'affectent pas les Flags. Dans cette catégorie on trouve toutes les instructions de branchement, de boucle et d'interruption après un branchement, le tableau suivant donne ces instructions :

BRANCHEMENTS INCONDITIONNELS

Format `CALL Label`

L'instruction CALL est utilisée chaque fois que nous devons appeler une procédure ou un sous-programme.



Label : est le nom du sous-programme. Ce nom peut représenter tout identifiant significatif, donc n'importe quel nom.

RET : retour (return) au programme principal
Proc : procedure (sous-ptogramme)
Endp : end of procedure (la fin de la procedure)

Exemple2.23

```
MOV  AX, 1
CALL  Exemple
MOV  AX, 3
```

```
INT 21

Exemple proc

MOV AX, 2

RET
Exemple endp
```

Format `JMP Label`

(Jump to the instruction identified by Label)

Cette instruction permet d'exécuter un saut inconditionnel parce qu'aucune condition n'est vérifiée. Donc, le saut est simplement effectué à ce Label.

```
    Instruction A
    Instruction B
        .
Label : Instruction
        .
        .
    JMP Label
    Instruction N
```

Exemple 2.24

```
    MOV AX, 1
    MOV BX, 2
    JMP Adresse
    MOV CX, 3   ; cette instruction n'est pas exécutée
Adresse :
    MOV DX, 4
    INT 21
```

BRANCHEMENTS CONDITIONNELS (FLAGS)

Format `Jcondition Label`

Condition= C, NC, E, NE, Z, NZ, P, NP, S, NS, O, NO,….sont les conditions correspondantes aux indicateurs du registre d'état (drapeaux).

➢ **Remarque**

Ces instructions ne fonctionnent pas seulement en fonction du contenu du registre drapeaux, mais d'autres conditions peuvent être utilisées ou l'arithmétique signée est considérée :

Condition= G, NG, L, NL,….

Instruction A
Instruction B
.
Label : Instruction
.
.
Jcondition Label

Il y aura un saut à l'adresse spécifiée par Label que si la condition est vérifiée

Instruction N ; exécution de cette instruction que si la condition n'est pas satisfaite

BOUCLES

Ces instructions sont utilisées pour effectuer l'itération de certains processus pour exécuter le mécanisme de bouclage en langage d'assemblage 8086.

Format `LOOP Label`

(Boucle)

Les instructions de boucle utilisent le registre CX pour indiquer le nombre de boucles.

L'instruction `LOOP` décrémente CX sans changer aucun drapeau, si CX n'est pas zéro après la décrémentation, le saut continu.

CX ← (CX) - 1

Format `LOOPE Label`

(Boucle si égal)

Décrémenter le registre CX :

- Si le registre CX n'est pas zéro et le drapeau zéro CZ est 1, faire une boucle.
- Si le registre CX n'est pas zéro et le drapeau zéro CZ est 0, sortir de la boucle.

Exemple2.25

```
    MOV AX, 1
    MOV CX, 3
Label:
    SUB AX, 1
    LOOPE   Label

INT 21
```

(AX)=1 ;
(CX)=3

(AX)=1-1=0, donc une boucle parce que (ZF=1) ; (CX)=2

(AX)=0-1=FFFF, donc sortie de la boucle parce que (ZF=0) ; (CX)=2

2.3 Méthode de programmation

Dans ce contexte, nous adopterons une démarche qui vous aidera à structurer l'analyse d'un problème avant de le programmer en assembleur. Ainsi, la résolution se fera en décomposant le travail en étapes claires et organisées. Les étapes sont les suivantes :

1. Comprendre pleinement le problème et le résoudre d'abord de manière classique, ce qui vous permettra de mieux le cerner.
2. Identifier les registres appropriés à utiliser pour la manipulation des données.
3. Élaborer un organigramme ou une liste d'étapes pour structurer votre programme.
4. Tester l'organigramme étape par étape en utilisant des données (traitement), que ce soit pour un transfert ou un calcul, afin de vérifier si les résultats attendus sont obtenus ou non.
5. Si les résultats attendus sont confirmés, identifier les instructions assembleur adaptées pour chaque étape.
6. Rédiger le programme étape par étape, assembler-le, puis simuler-le à l'aide d'un assembleur.

Cette méthode progressive vous permettra de développer des compétences solides tout en se donnant confiance dans la programmation en assembleur.

Exercice 2.1

Ecrire un programme en assembleur qui permet de calculer Y.

Y= $\sum_{i=1}^{FH} i$

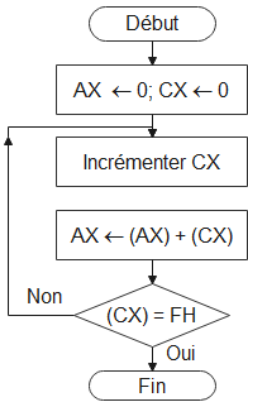
Solution

a) La solution classique

$\sum_{i=1}^{FH} i$ = 1+ 2+ 3+ 4+ 5+ . . . + EH+ FH = 78H

Pour le calcul, on peut se baser sur le registre AX et CX

b) L'organigramme



c) Traitement

(AX)=0 ; (CX)=0

(CX)=1 ; (CX)=2 ; … ; (CX)=FH

(AX)=0+1 ; (AX)=0+1+2 ; … ; (AX)=0+1+2+…+FH

(CX)=1 ; (CX)=2 ; … ; (CX)=FH

d) Programme en assembleur

```
MOV   AX, 0
```

```
        MOV   CX, 0
Label : INC  CX
        ADD AX, CX
        CMP CX, 0F
        JNZ   label ; on peut utiliser  JNE,  JL
        INT 21
```

Exercice 2.2

Reprendre l'exercice1 pour le calcul de : Y= $\prod_{i=1}^{7} i$

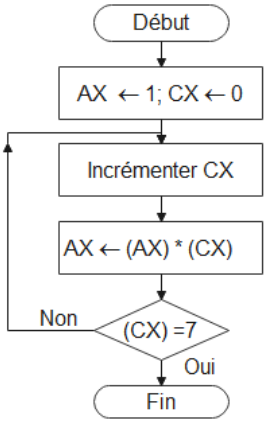
Solution

a) La solution classique

$\prod_{i=1}^{7} i$ = 1* 2* 3* . . . *7 = 13B0H

Pour le calcul, on peut se baser sur le registre AX et CX

b) L'organigramme



c) Traitement

(AX)=1 ; (CX)=0

(CX)=1 ; (CX)=2 ; … ; (CX)=7

(AX)=1*1 ; (AX)=1*1*2 ; … ; (AX)=1*1*2*…*F

(CX)=1 ; (CX)=2 ; … ; (CX)=7

c) Programme en assembleur

```
        MOV   AX, 1
        MOV   CX, 0
Label : INC  CX
        MUL  CX
        CMP CX, 7
        JNZ   label ;  on peut utiliser  JNE,  JL
        INT 21
```

Exercice 2.3

Ecrire un programme en assembleur pour calculer Y= $\sum_{i=1}^{5} i^2$

a) Sans sous-programme

b) Avec sous-programme

Solution

a) Sans sous-programme

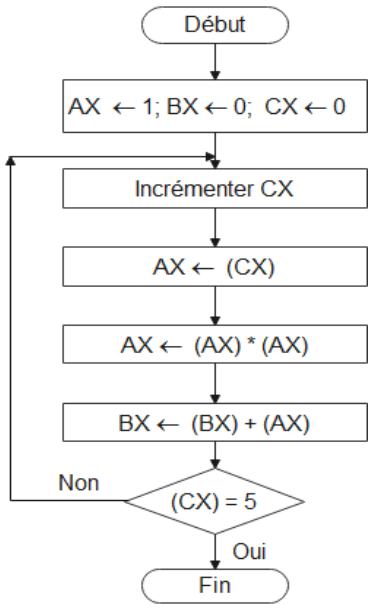
- Solution classique

$$\sum_{i=1}^{5} i^2 = 1^2 + 2^2 + ... + 5^2 = 1*1 + 2*2 + ... + 5*5 = 37H = 55_{10}$$

Dans ce cas, les registres à utiliser sont :

CX : pour l'incrémentation
AX : pour le produit
BX : pour la somme

- L'organigramme



- Traitement

(AX)=1 ; (BX)=0 ; (CX)=0

(CX)=1 ; (CX)=2 ; ……. ; (CX)=5

(AX)=1 ; (AX)=2 ; ……..… ; (AX)=5

(AX)=1*1 ; (AX)=2*2 ; …… ; (AX)=5*5

(BX)=0+1*1 ; (BX)=0+1*1+2*2 ;.. ; (BX)=0+1*1+2*2+…+ 5*5

(CX)=1 ; (CX)=2 ; ……. ; (CX)=5

- Programme en assembleur

```
        MOV AX, 1    ; comme on peut mettre  MOV AX, 0
        MOV CX, 0
        MOV BX, 0
Label : INC CX
        MOV AX, CX
        MUL AX
        ADD  BX, AX
        CMP CX, 5
        JNZ   label
        INT 21
```

b) Avec sous-programme

- Programme en assembleur

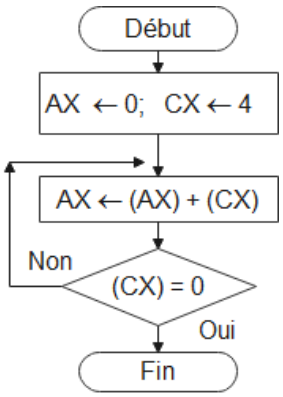
```
        MOV AX, 1    ;  comme on peut mettre  MOV AX, 0
        MOV CX, 0
        MOV BX, 0
Label : INC CX
        CALL Calcul
        CMP CX, 5
        JNZ  label
        INT 21

Calcul  proc
    MOV AX, CX
    MUL AX
   ADD BX, AX
    ret
 Calcul  endp
```

Exercice2.4

Ce programme additionne le contenu du registre **CX** au registre **AX** jusqu'à ce que **CX** atteigne la valeur 0. (Cas de boucles)

Solution



(AX)=1 ; (CX)=4

(AX)=0+4 ; (AX)=0+4+3 ; (AX)=0+4+3+2 ; (AX)=0+4+3+2 +1

(CX)=3 ; (CX)=2 ; (CX)=1 ;(CX)=0

```
        MOV AX, 0
        MOV CX, 4
Label:  ADD AX, CX
        LOOP   label
        HLT      ;  l'instruction HALT  fait passer le microprocesseur dans un état d'arrêt
```

Exercice 2.5

Ecrire un programme en assembleur pour le calcul de : Y= $\prod_{i=1}^{3} i^6$; Y1= $\sum_{i=1}^{3} i^6$; Y-Y1

À la suite de ces exercices, vous serez en mesure de résoudre cet exercice par vous-même.

Chapitre 3 : Les interruptions et les interfaces d'entrées/sorties (E/S)

Objectifs

A l'issue de ce chapitre, vous serez capable de :

Comprendre le rôle des interruptions dans un système à microprocesseur.

Savoir comment le 8086 gère les interruptions et comment la table des vecteurs d'interruption est utilisée.

Maîtriser les principes de base de l'adressage des ports d'E/S et la communication avec les périphériques.

Savoir configurer et utiliser l'interface 8255 pour gérer les ports d'E/S en mode parallèle.

3.1 Introduction

Ce chapitre aborde les concepts fondamentaux relatifs aux `interruptions` et aux interfaces d'entrées/sorties (E/S), qui jouent un rôle `essentiel` dans la gestion des communications entre un microprocesseur et les périphériques externes. En se concentrant sur le `microprocesseur 8086` et l'`interface parallèle 8255`, il offre une compréhension approfondie de leur fonctionnement et de leur interaction.

3.2 Définition d'une interruption

Une interruption est un signal envoyé au microprocesseur pour interrompre l'exécution normale du programme et demander un traitement spécifique.

Exemple3.1

- **Interruption d'horloge** : Le microprocesseur reçoit un signal régulier qui lui rappelle de faire certaines actions importantes, comme sauvegarder un fichier ou mettre à jour l'écran.
- **Interruption de clavier** : Quand on appuie sur une touche, un signal est envoyé au microprocesseur pour que le système réagisse immédiatement.
- **Interruption de souris** : Quand on bouge la souris ou clique, un signal est envoyé pour que le curseur se mette à jour à l'écran.
- **Interruption de disque dur** : Quand le disque dur a fini de lire ou écrire des données, il envoie un signal pour dire que l'opération est terminée, permettant ainsi au processeur de passer à une autre tâche.
- **Interruption de division par zéro** : Si un programme essaie de diviser par zéro, une erreur se produit et le processeur arrête l'exécution.

```
Ces exemples montrent comment les interruptions permettent de
réagir en temps réel aux événements, assurant une gestion
efficace des ressources.
```

Types d'interruptions	
Interruptions matérielles *déclenchées par des périphériques externes (clavier, souris, etc.).*	**Interruptions logicielles** *générées par le programme lui-même pour effectuer certaines actions spécifiques.*

Figure3.1 *Types d'interruptions*

❖ La Table des Vecteurs d'Interruption

Le `microprocesseur 8086` utilise une `table des vecteurs d'interruption` pour savoir où se trouve le code pour gérer chaque type d'interruption.

La table est située dans la `mémoire RAM` au début de l'adresse mémoire, c'est-à-dire à `l'adresse 0x0000`. Dans les systèmes basés sur le microprocesseur `8086`, cette zone de mémoire est réservée et dédiée uniquement aux vecteurs d'interruption.

Comme nous remarquons dans la table (la RAM), chaque entrée de la table des vecteurs d'interruption occupe `4 octets` (16 bits pour le segment (CS) et 16 bits pour l'offset(IP)). Avec 256 vecteurs d'interruption possibles, cela prend un total de `1024 octets` (1 Ko), soit les adresses `0x0000 à 0x03FF` en RAM.

Adresse mémoire	Entrée de la table	Définition du vecteur
3FE	CS 255	Vecteur 255
3FC	IP 255	
3FA	CS 254	Vecteur 254
3F8	IP 254	
	⋮	
82	CS 32	Vecteur 32
80	IP 32	
7E	CS 31	Vecteur 31
7C	IP 31	
	⋮	
12	CS 4	Vecteur 4 (Overflow)
10	IP 4	
0E	CS 3	Vecteur 3 (Logiciel)
0C	IP 3	
0A	CS 2	Vecteur 2 (NMI)
08	IP 2	
06	CS 1	Vecteur 1(Pas-à-Pas)
04	IP 1	
02	CS 0	Vecteur 0 (Erreur DIV)
00	IP 0	
	◄---- 2 octets ----►	

Figure 3.2 *Organisation de la table d'interruption*

Donc, chaque vecteur d'interruption dans la table est associé à une tâche ou un événement spécifique.

- Les `vecteurs 0 à 31` sont réservés pour des interruptions internes du microprocesseur, comme des erreurs de division par zéro ou des erreurs système.

- Les `vecteurs au-delà de 31` sont souvent utilisés pour des interruptions provenant de périphériques externes (clavier, souris, etc.) ou des interruptions logicielles personnalisées (créées par les utilisateurs ou pour des périphériques supplémentaires).

Dans la `table des vecteurs d'interruption` du 8086, chaque vecteur d'interruption pointe vers une adresse spécifique en mémoire grâce à deux valeurs :

1. `CS (Code Segment)` : Un segment de 16 bits qui indique dans quel segment de la mémoire se trouve le code de l'interruption.
2. `IP (Instruction Pointer)` : Un offset de 16 bits qui indique la position exacte dans ce segment où commence le sous-programme d'interruption.

3.3 Prise en charge d'une interruption par le microprocesseur

Dans cette partie sera expliqué `le processus global` par lequel le microprocesseur répond à une interruption. Il couvre les étapes générales, de la réception de l'interruption jusqu'au retour au programme principal. On y voit comment le 8086 gère une interruption, notamment :

- La `réception du signal d'interruption.`
- La `sauvegarde de l'état` actuel
- La `recherche du sous-programme` à exécuter.
- L'`exécution` du code du sous-programme
- Le `retour au programme principal`

Pour rendre cette partie plus accessible, voyons comment le microprocesseur 8086 gère une erreur de `division par zéro` de manière simple.

Exemple3.2

Supposons qu'un programme essaye de diviser un nombre par zéro, ce qui déclenche une erreur. Voyons les étapes de la prise en charge de cette interruption.

1. **Détection de l'Erreur (Interruption) :** Lorsque la division par zéro est tentée, le microprocesseur détecte une **interruption** due à cette erreur.

2. Il met en pause le programme en cours.
3. **Sauvegarde de l'État Actuel :** Avant de s'occuper de l'erreur, le microprocesseur enregistre l'emplacement exact où il s'est arrêté. Cela lui permet de revenir à ce point précis après avoir traité l'erreur.

4. **Recherche du Code d'Erreur :** Pour savoir comment gérer la division par zéro, le microprocesseur consulte la `table des vecteurs d'interruption`.

5. **Exécution du Code d'Erreur :** Le microprocesseur exécute le code pour gérer la division par zéro. Par exemple, ce code pourrait afficher un message du type « Erreur : Division par Zéro » à l'écran, ou encore arrêter le programme pour éviter d'autres problèmes.

5. Retour au Programme Principal : Après avoir traité l'erreur, le microprocesseur retourne à l'endroit exact où il s'était arrêté dans le programme principal.

3.4 Adressage des sous-programmes d'interruptions

Cette partie se concentre sur **la manière dont le microprocesseur trouve le code spécifique (sous-programmes)** à exécuter pour chaque type d'interruption.

L'adressage des sous-programmes d'interruption repose sur une structure organisée en vecteurs dans une table d'interruptions (voir figure3.2). Chaque entrée de cette table contient l'adresse segmentée d'un sous-programme d'interruption, avec 16 bits pour le segment (CS) et 16 bits pour l'offset (IP).

Ainsi, la table des vecteurs d'interruption fournit les adresses des sous-programmes d'interruption, permettant au 8086 de localiser le code nécessaire pour gérer chaque interruption.

Ensemble, le `CS` et l'`IP` forment l'adresse complète d'un sous-programme d'interruption dans la mémoire.

Exemple3.3

Si le vecteur d'interruption contient un `CS de 0x1234` et un `IP de 0x5678`, alors le `sous-programme d'interruption` se trouvera à l'adresse `0x1234:0x5678` (segment 0x1234 avec offset 0x5678). Le microprocesseur ira directement à cet emplacement pour exécuter le code associé à cette interruption.

Exemple3.4

Pour bien comprendre l'adressage des sous-programmes d'interruption sur le **8086**, prenons un exemple clair étape par étape. `Cas d'une Interruption de Clavier`

1- on appuie sur une touche du clavier.

2- Le clavier envoie une interruption pour informer le microprocesseur 8086.

3- Le 8086 consulte alors la table des vecteurs d'interruption pour savoir où trouver le code qui gère cet événement.

4- Supposons que l'interruption du clavier est associée au **vecteur numéro 9**. Cela signifie que l'entrée numéro 9 dans la table des vecteurs d'interruption contient l'adresse du sous-programme qui gère l'interruption du clavier.

5- Cette adresse est segmentée, c'est à dire :

- Chaque entrée de la table est composée de **2 parties** : un **segment (CS)** de 16 bits et un **offset (IP)** de 16 bits. Ensemble, ces deux parties forment l'adresse segmentée du sous-programme.
- Par exemple, imaginons que l'entrée pour le clavier (vecteur 9) contient **CS = 0x1000** et **IP = 0x0020**.

6- Calcul de l'adresse réelle :
- L'adresse complète où se trouve le sous-programme est donc CS:IP, soit 0x1000:0x0020. Cette notation signifie que le sous-programme d'interruption du clavier est situé dans le segment mémoire 0x1000, avec un décalage (offset) de 0x0020 à partir du début de ce segment.
- Le microprocesseur se rend à cette adresse pour exécuter le code (appelé **sous-programme d'interruption**) qui traitera l'interruption.

7- Le microprocesseur exécute le sous-programme d'interruption.

8- Après avoir traité l'interruption, le microprocesseur retourne au programme principal, là où il s'était arrêté, pour continuer sa tâche.

3.5 Adressages des ports d'E/S

3.5.1 Principe

- Les `ports d'Entrée et de sortie (E/S)` sont des `portes` par lesquelles le microprocesseur échange des informations avec les périphériques. Chaque port a une adresse unique, ce qui permet au microprocesseur d'envoyer ou de recevoir des données vers le bon périphérique.

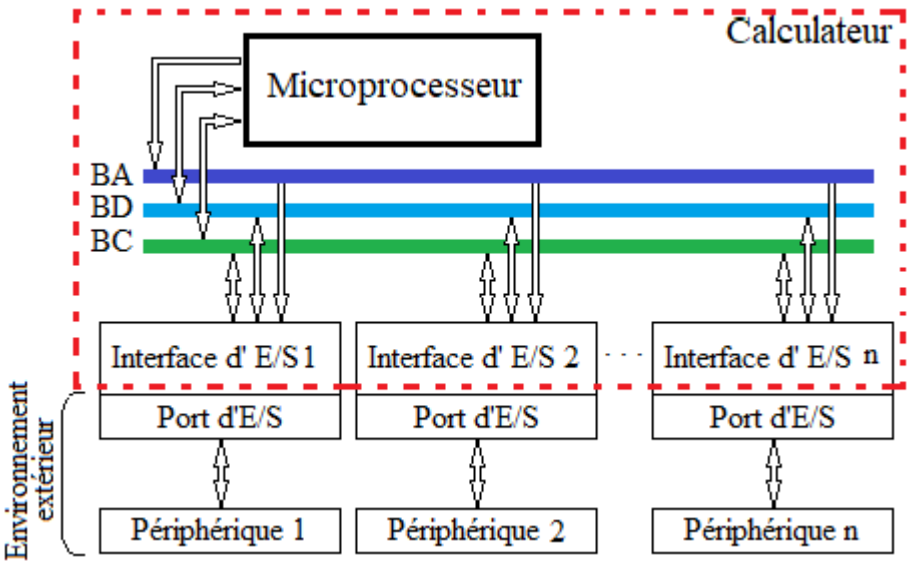


Figure3.3 *Ports d'E/S*

Chaque périphérique est relié à un port d'E/S, permettant au microprocesseur de lire ou d'écrire des données vers ce périphérique.

Exemple3.5

- Si un microprocesseur doit allumer une LED connectée, il envoie un signal de sortie au port d'E/S associé à cette LED.
- De la même manière, pour lire l'état d'un capteur de température, le microprocesseur interroge le port d'E/S associé à ce capteur pour obtenir sa valeur.

- L'`adressage des ports d'E/S` est la `méthode` permettant au processeur d'accéder aux périphériques comme les claviers, souris ou capteurs. Chaque périphérique est connecté à un port d'E/S avec une adresse unique, ce qui permet au processeur de communiquer avec lui.
  Pour accéder à ces périphériques, on utilise les instructions `IN` et `OUT`.

3.5.2 Instructions IN et OUT

Les instructions `IN et OUT` permettent au microprocesseur d'accéder aux ports d'E/S.

- `IN (Input)` : Cette instruction permet de lire des données depuis un périphérique (entrée) et de les stocker dans un registre du microprocesseur.

- `OUT (Output)` : Cette instruction permet d'envoyer des données depuis un registre du microprocesseur vers un périphérique (sortie).

3.5.3 Format de base des instructions

- `IN Port, Registre` : Lire une donnée à partir du port spécifié et la stocker dans le registre.
- `OUT Registre, Port` : Écrire une donnée depuis le registre vers le port spécifié.

3.5.4 Registres utilisés pour l'adressage des Ports d'E/S

Les registres qui sont utilisés pour manipuler les données avec `IN et OUT` sont les suivants :

- **Registres de 8 bits :**
  - `AL (Accumulateur de 8 bits)` : Utilisé pour les opérations sur des données de `8 bits`. L'instruction `IN` ou `OUT` manipule souvent `AL` lorsque l'on travaille avec des données de `8 bits`.

- **Registres de 16 bits :**
  - `AX (Accumulateur de 16 bits), BX, CX, DX` : Ces registres sont utilisés pour des opérations sur des données de `16 bits.`

3.5.5 Adressage des Ports d'E/S

Dans ce cas, l'adresse du port peut être spécifiée directement dans l'instruction ou stockée dans le registre `DX.`

- **Adressage avec des données de 8 bits :**

Exemple3.6

Instruction `IN` pour lire `8 bits` :
IN DX, AL ; Lire 1 octet du périphérique connecté au port 0x60 et le stocker dans AL

Exemple3.7

Instruction `OUT` pour écrire `8 bits` :
OUT AL, DX ; Écrire 1 octet du registre AL vers le périphérique connecté au port 0x60

- **Adressage avec des données de 16 bits**

Exemple3.8

Instruction `IN` pour lire `16 bits` :
IN DX, AX **;** Lire 2 octets du périphérique connecté au port 0x60 et les stocker dans AX

Exemple3.9

Instruction `OUT` pour écrire `16 bits` :
OUT AX, DX ; Écrire 2 octets du registre AX vers le périphérique connecté au port 0x60

- **Adresse du Port dans l'Instruction**

`Adresse immédiate` : L'adresse du port peut être donnée directement dans l'instruction sous la forme d'une valeur immédiate (comme 0x60).

Exemple3.10

IN 0x60, AL ; Lire 1 octet du port 0x60 et le stocker dans AL

3.6 Gestion des ports d'E/S - Interface 8255

Dans cette partie, nous allons étudier la `gestion des ports d'E/S` en utilisant un composant spécifique : le `8255 Programmable Peripheral Interface (PPI).` Ce composant est couramment utilisé pour permettre au microprocesseur de se connecter facilement à différents périphériques externes (clavier, capteurs, afficheurs, etc.).

Le 8255 est un circuit intégré qui simplifie la gestion des ports d'E/S en organisant les échanges entre le microprocesseur et les périphériques. Il possède plusieurs ports programmables, configurables selon les besoins de chaque périphérique.

3.6.1 Présentation du 8255 PPI

La structure interne de ce type de circuit comprend principalement des `ports d'E/S` et un `registre de contrôle` (figure 3.4)

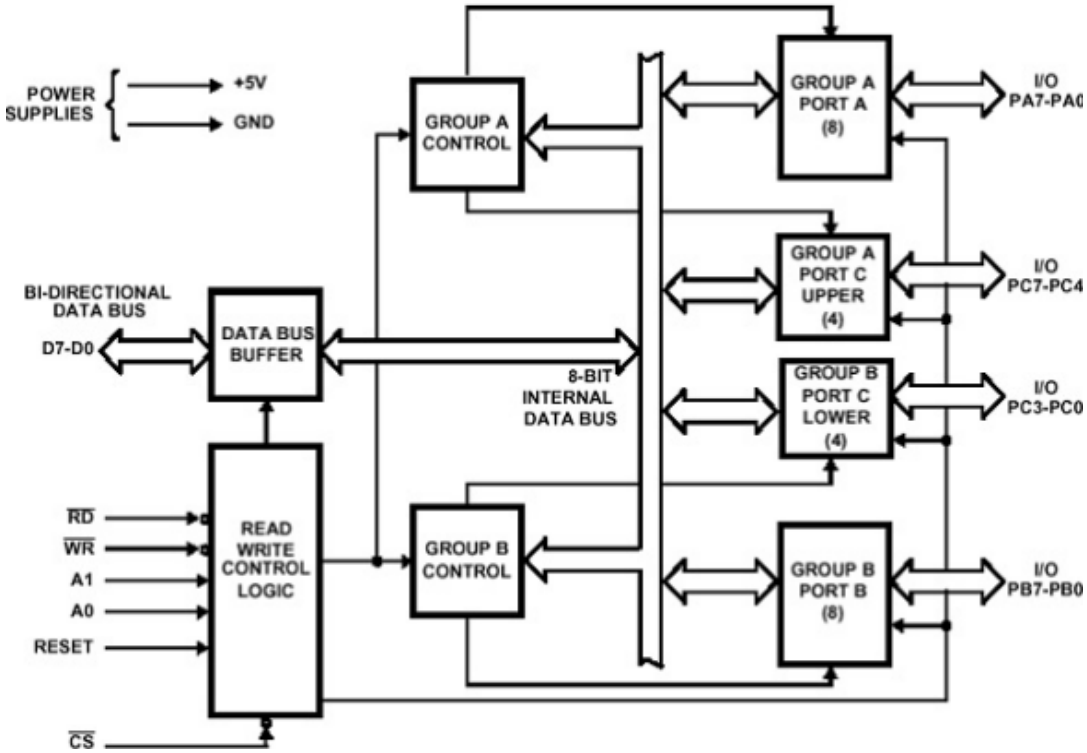


Figure 3.4 *Architecture interne du 8255 IPP*

- `Data Bus Buffer` (Tampon du bus de données): Il est utilisé pour connecter le bus interne du 8255 avec le bus système (Microprocesseur 8086). Le `tampon du bus de données` permet d'effectuer l'opération de :
  - lecture depuis les ports vers le microprocesseur.
  - écriture depuis le microprocesseur vers les ports.

- `Read/Write control Logic` (Logique de contrôle de lecture/écriture) : Cette unité permet de contrôler le transfert de données entre le 8255 et le 8086.

- `Group A and Group B control` (Contrôle du groupe A et du groupe B) : ces deux groupes permettent d'envoyer des mots de contrôle au contrôle des ports du groupe A et du groupe B.

Comme nous remarquons que :

- le groupe A a accès au port A et aux bits d'ordre supérieur du port C.
- le groupe B contrôle le port B avec les bits d'ordre inférieur du port C.

- **Reset** (Réinitialiser) : C'est un signal qui permet la réinitialisation du PPI (efface les registres de contrôle)

- **RD** (READ) - C'est le signal utilisé pour l'opération de lecture sur les ports du 8255.

- **WR** (WRITE) - permet au microprocesseur d'effectuer une opération d'écriture sur les ports.
- **CS** (Chip select) permet de sélectionner la puce (le 8255).
- **A0** et **A1** : sont utilisés pour sélectionner le port souhaité parmi tous les ports du 8255.

Ainsi, le schéma logique fonctionnel est donné par la figure3.5

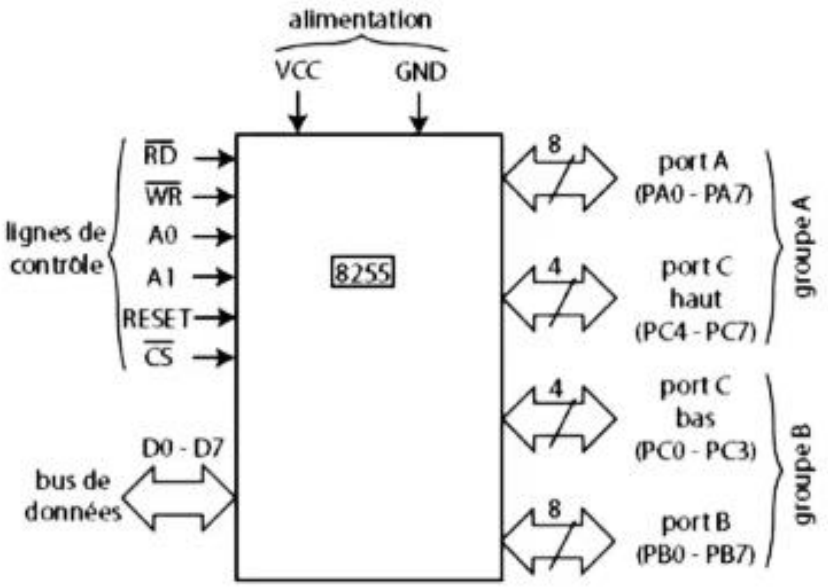


Figure3.5 *Schéma fonctionnel*

Donc, le 8255 dispose de 4 registres dont l'accès se fait grâce aux lignes A0 et A1 :

A_1	A_0	Port
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Registre de contrôle

Chaque port du 8255 est associé à une `adresse unique`. Le microprocesseur utilise ces adresses pour lire ou écrire des données sur les ports.

3.6.2 Modes de fonctionnement du 8255

Le 8255 peut être configuré dans différents modes de fonctionnement, selon les besoins de l'application :

Mode 0 (`Mode de base`)**: L**es ports sont configurés en entrée et en sortie simple.

Mode 1 : le port A et le port B peuvent être utilisés comme ports d'entrée ou de sortie, le port C est utilisé pour l'établissement de liaison.

Mode 2 : seul le port A peut fonctionner, et le port B peut être en mode 0 ou en mode 1, et le port C est utilisé pour l'établissement de liaison.

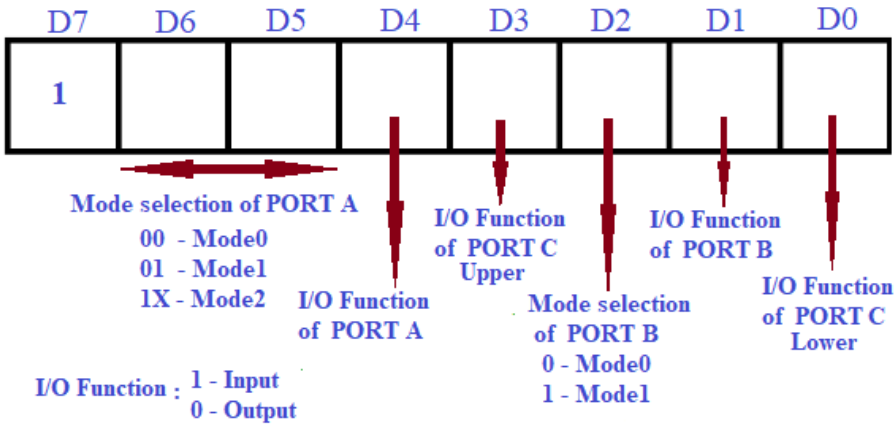


Figure 3.6 *Registre de contrôle*

3.6.3 Communication du 8255 avec les périphériques

- Pour assurer une communication en entrée (lecture des données des périphériques), l'instruction IN devrait être utilisée.

IN AL, PORT_adresse ou bien IN AL, DX ; données 8 bits
IN AX, PORT_adresse ou bien IN AX, DX ; données 16 bits

- Pour assurer une communication en sortie (écriture des données à la sortie des ports vers les périphériques), l'instruction OUT devrait être utilisée.

OUT PORT_adresse , AL ou bien OUT DX, AL ; données 8 bits
OUT PORT_adresse, AX ou bien OUT DX, AX; données 16 bits

❖ Configuration du 8255

La configuration du 8255 est effectuée à l'aide d'un `registre de contrôle`. Le processeur écrit dans ce registre pour choisir le mode de fonctionnement et déterminer si chaque port doit fonctionner en `entrée ou en sortie`.

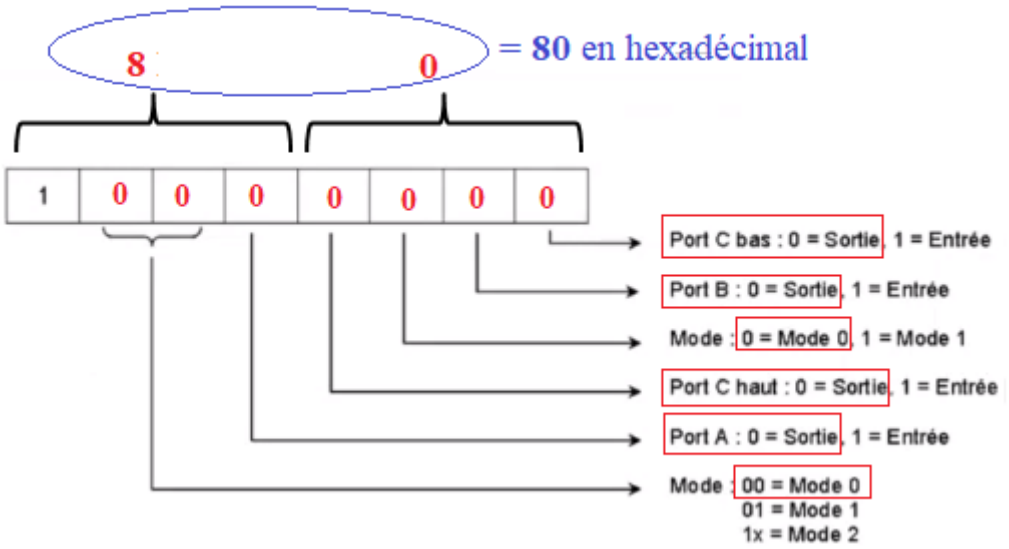
Exemple3.11

Trouver le mot de contrôle pour chaque configuration en mode 0.

a) Les ports A, B et C sont considérés comme des ports de sortie
b) Port A= entrée ; Port B= sortie ; Port C_Upper= sortie ; Port C_Lower= sortie
c) Port A= sortie ; Port B= entrée ; Port C_Upper= sortie ; Port C_Lower= entrée

Solution

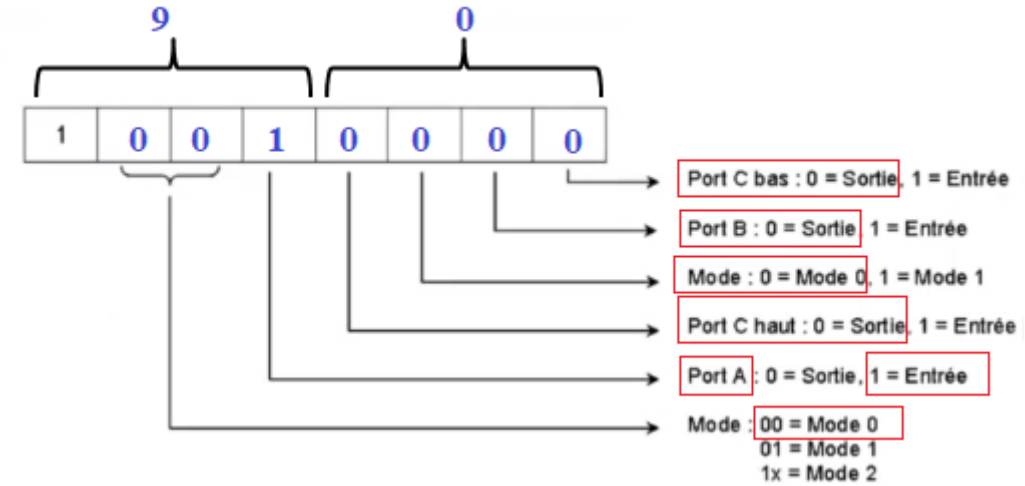
a) Tous les ports sont configurés en sortie




```
Le mot de contrôle pour ce 1ér cas est 80H
```

Figure 3.7 *Registre de contrôle. Mode* 0. a) *de l'exercice*

b) PA= entrée ; PB= sortie ; PC_Upper= sortie ; PC_Lower= sortie




```
Le mot de contrôle pour ce 2éme cas est 90H
```

Figure 3.7 *Registre de contrôle. Mode* 0. b) *de l'exercice*

c) PA= sortie ; PB= entrée ; PC_Upper= sortie ; PC_Lower= entrée

```
Le mot de contrôle pour ce 3éme cas est 83H
```

Exemple3.12

Ecrire un programme en assembleur qui permet d'allumer les **Leds** à la sortie du port B de l'interface 8255 comme indiqué sur la figure suivante :

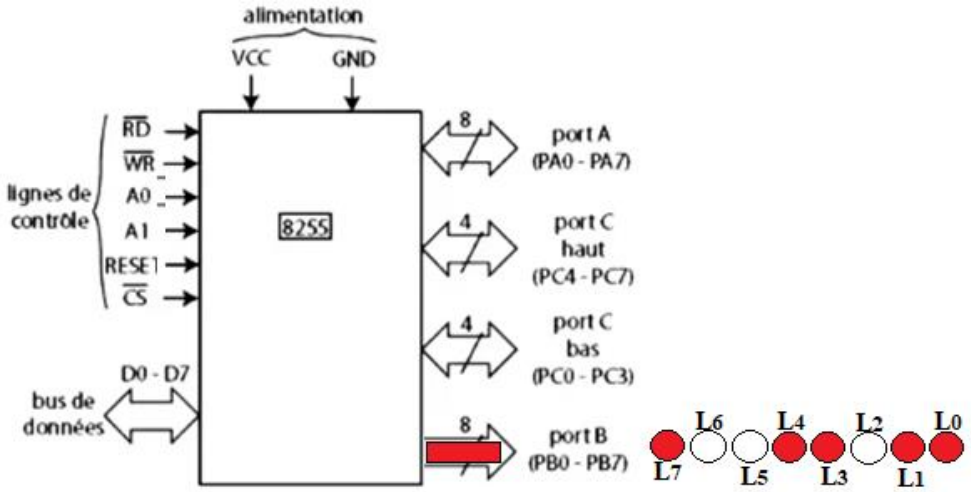
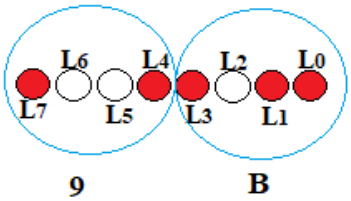


Figure 3.8 *Sorties via le port B du 8255*

Solution

Deux étapes sont à considérer :

1) Configuration du mot de contrôle
2) Sortie de la donnée 9Bh selon les Leds allumées de la figure



`1) Configuration du mot de contrôle`

Ici, le cas qui nous intéresse est le Port B et qui devrait être utilisé en sortie. Les autres ports leur configuration en entrée ou en sortie ne cause aucun problème, donc on peut utiliser les mots de contrôle trouvés en a) et en b) de l'exemple précédent, à savoir **80h** ou **90h**

`2) Programmation`

Dans ce cas les adresses du registre de contrôle et des ports sont connues. Nous supposons les adresses sont comme suit :

Registre de contrôle (mot de contrôle) : 3FD6h
Port B : 3FD2h

```
< Configuration des ports du 8255>
```

MOV AL, 90H ou 80H ; transférer le mot de contrôle dans l'accumulateur
MOV DX, 3FD6H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le registre de contrôle

OUT DX, AL ; sortie de données 90H vers le port de contrôle du 8255

```
< Sortie de la donnée  9BH vers les Leds>
```

MOV AL, 9BH ; transférer la donnée 9BH dans l'accumulateur
MOV DX, 3FD2H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le Port B.
OUT DX, AL ; sortie de la donnée 9BH vers le port B de sortie (ce qui permet d'allumer les Leds)

Exercice3.1

Ecrire un programme en assembleur qui permet d'allumer la Led suite à l'action sur l'interrupteur I.

Port A= sortie (sortie sur la broche PA_0); Port B= entrée (entrée sur la broche PB_1)

Solution

Deux étapes sont à considérer :

1) Configuration du mot de contrôle
2) Sortie de la donnée 01H sur PA selon l'entrée 02 sur PB

```
Configuration du mot de contrôle
```

PA=01H ; PB =02H ; les autres ports peuvent être configurés soit en entrée, soit en sortie.

Donc, D7= toujours 1 ; Mode0 :**00** ; PA (Input : **1**) ; PC Upper (Output :**0**) ; mode (**0**) ; PB(Output :**0**) ; PC Lower (Output :**0**)

Dans ce cas, le mot de contrôle est 90H

```
Programmation
```

Les adresses du registre de contrôle et des ports sont connues :

Port A : 3FD0H
Port B : 3FD2H
Port C : 3FD4H
Registre de contrôle (mot de contrôle) : 3FD6H

```
< Configuration des ports du 8255>
```

MOV AL, 90H ; transférer le mot de contrôle dans l'accumulateur
MOV DX, 3FD6H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le registre de contrôle
OUT DX, AL ; sortie de données 90H vers le port de contrôle du 8255

< Entrée de la donnée 01H (Puisque PA_0 =1 ; PA_1= PA_2=…= PA_7 =0)>

Retour : MOV DX, 3FD0H ; transférer l'adresse du PA dans le registre DX (ce qui permet d'activer le PA.
IN AL, DX; entrée de la donnée 01H vers le port A (Puisque PA0 est concernée)
CMP AL, 02 ; si c'était L0 qui s'allume, l'instruction CMP n'est pas obligatoire, mais l'entrée est 01H et la sortie est 02H (L1)
JNZ **Retour**

< Sortie de la donnée 9BH vers les Leds>

MOV DX, 3FD2H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le Port B.
OUT DX, AL ; sortie de la donnée 02H vers le port B de sortie (ce qui permet d'allumer la Led L1)
JMP **Retour**

Exercice3.2

Ecrire un programme en assembleur qui permet d'allumer la Leds (L_0, …, L_7) suite à l'action sur les interrupteurs (I_0,…, I_7).

Port A= sortie; Port C= entrée

Solution

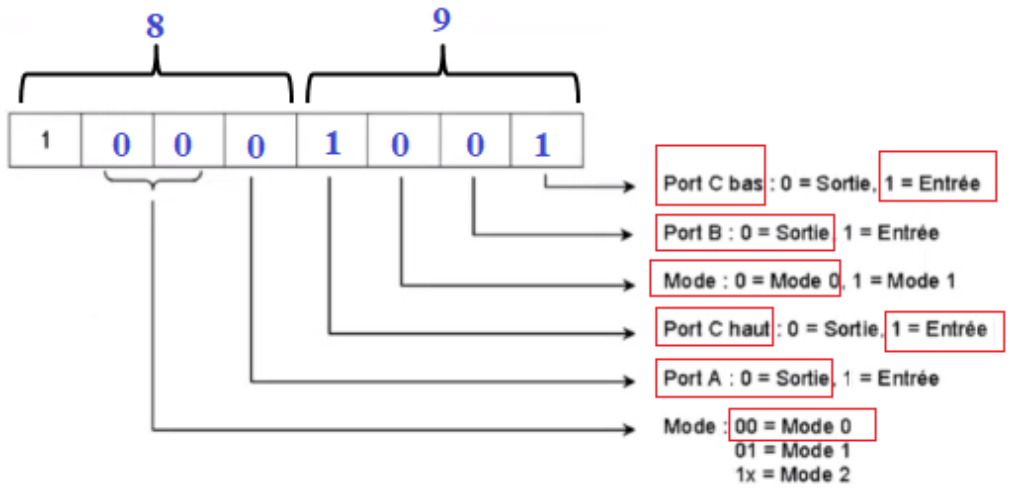


Figure 3.10 *L'interface 8255 utilisé en entrée et en sortie*

```
< Configuration des ports du 8255>
```

MOV AL, 89H ; transférer le mot de contrôle dans l'accumulateur

MOV DX, 3FD6H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le registre de contrôle
OUT DX, AL ; sortie de données 89H vers le port de contrôle du 8255

< Entrée des interrupteurs 0 à l'entrée de PC >

Retour : MOV DX, 3FD4H ; transférer l'adresse du PC dans le registre DX (ce qui permet d'activer le PC.
IN AL, DX; entrée des données vers le port C

< Sortie des données vers la sortie de PA>

MOV DX, 3FD0H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le Port A.
OUT DX, AL ; sortie d'une donnée vers le port A de sortie (ce qui permet d'allumer la Led ou les Leds concernées)
JMP **Retour**

Exercice3.3

Ecrire un programme en assembleur qui permet de faire clignoter Led L_0 à la sortie de PB.

Solution

```
Mot de contrôle : 80H
```

❖ **Programmation**

```
< Configuration des ports du 8255>
```

MOV AL, 80H; transférer le mot de contrôle dans l'accumulateur
MOV DX, 3FD6H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le registre de contrôle
OUT DX, AL ; sortie de données 80H vers le port de contrôle du 8255

```
< Sortie de la donnée  9BH vers les Leds>
```

Clignoter : MOV AL, 01H ; transférer la donnée 01H dans l'accumulateur
MOV DX, 3FD2H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le Port B.
OUT DX, AL ; sortie de la donnée 01H vers le port B de sortie (ce qui permet d'allumer la Led L_0)

MOV CX, 1000H ; charger la valeur dans le registre CX pour la temporisation
LOOP $; effectuer la boucle jusqu'à ce que (CX)=0

MOV AL, 00H ; transférer la donnée 00H dans l'accumulateur
MOV DX, 3FD2H ; transférer l'adresse du registre de contrôle dans le registre DX (ce qui permet d'activer le Port B.
OUT DX, AL ; sortie de la donnée 00H vers le port B de sortie (ce qui permet d'éteindre la Led L_0)

MOV CX, 1000H ;
LOOP $
JMP **Clignoter** ; retour pour allumer L_0 (L0 : tantôt elle s'allume et tantôt elle s'éteint, et cela continu puisqu'il y a une boucle qui ne s'arrêtera jamais)

Exercice3.4

Utiliser l'instruction CALL pour éviter une partie d'instructions qui se répète dans le programme de l'`exercice3`.

Chapitre 4 : Architecture et fonctionnement d’un microcontrôleur

Objectifs

A l’issue de ce chapitre, vous serez capable de :

Identifier les composants matériels d'un microcontrôleur,

Expliquer le rôle des éléments de base d’un microcontrôleur

Maîtriser les principes de base de la programmation d'un microcontrôleur, en utilisant les langages assembleur et C.

4.1 Introduction

Dans ce chapitre, nous explorerons les bases de l'architecture et du fonctionnement des microcontrôleurs, éléments essentiels pour comprendre leur rôle dans les systèmes électroniques. Nous aborderons la description matérielle d'un microcontrôleur, en détaillant ses composants internes et leur interaction, ainsi que son mode de fonctionnement. Ensuite, nous introduirons la programmation des microcontrôleurs en illustrant les concepts avec deux exemples : le microcontrôleur PIC et celui de la carte Arduino. Ce chapitre vise à fournir aux étudiants les connaissances nécessaires pour comprendre la programmation et l'utilisation des microcontrôleurs dans diverses applications.

Un microcontrôleur est un circuit intégré qui combine un processeur, de la mémoire et des interfaces d'entrées/sorties (E/S) pour gérer des tâches spécifiques dans des applications embarquées.

4.2 Description matérielle d'un µ-contrôleur et son fonctionnement

4.2.1 Structure interne d'un microcontrôleur

Le microcontrôleur utilise ces composants pour exécuter des tâches complexes. La figure 4.1, illustre les blocs de base sur lesquels repose le fonctionnement d'un microcontrôleur. Chacun de ces éléments a une fonction spécifique, et ensemble, ils permettent au microcontrôleur :

- d'exécuter des tâches variées,
- de traiter des données,
- de communiquer avec d'autres systèmes (`capteurs, actionneurs, autres microcontrôleurs, modules de communication sans fil`,..)
- de contrôler des appareils externes.

Dans la suite, une explication simple de l'architecture interne d'un microcontrôleur, avec des descriptions des différents composants illustrés dans la figure :

1. `Oscillateurs`

- Ce sont des composants essentiels qui fournissent l'horloge nécessaire au fonctionnement du système.

Exemple4.1

Un microcontrôleur peut fonctionner à des fréquences allant jusqu'à 200 MHz.

2. `Mémoire`

La mémoire stocke les informations et les programmes que le microcontrôleur utilise.

- `RAM (Random Access Memory)` : Mémoire temporaire utilisée pour stocker temporairement les données et les variables pendant l'exécution des programmes.

Exemple4.2

Elle stocke les variables pendant que le programme fonctionne, comme la valeur d'une température lue.

- `ROM (Read-Only Memory)` : Mémoire permanente utilisée pour stocker le programme principal du microcontrôleur, ainsi que les instructions nécessaires à son fonctionnement. Cela inclut le `firmware` et les `routines de démarrage` (éléments essentiels dans le fonctionnement des microcontrôleurs).

Exemple4.3

Le programme de base qui demande au microcontrôleur de lire un capteur de température au démarrage.

3. `Ports (I/O)`

- Ce sont des points de connexion pour que le microcontrôleur puisse communiquer avec l'extérieur.

Exemple4.4

Un port `entrée` peut recevoir des signaux d'un capteur, et un port `sortie` peut envoyer un signal pour allumer une LED.

4. `PWM (Pulse Width Modulation)`

- C'est une technique utilisée pour contrôler la vitesse des moteurs ou la luminosité des voyants en modulant la durée d'un signal électrique.

Exemple4.5

Pour un moteur, plus le signal PWM est long, plus le moteur tourne vite.

5. `ADC/DAC (Convertisseur Analogique/Numérique)`

- `ADC` : Convertit un signal analogique (comme la température) en un signal numérique que le microcontrôleur peut comprendre.

Exemple4.6

Un capteur de température donne un signal analogique, que l'ADC transforme en un nombre numérique.

- `DAC` : Convertit un signal numérique en un signal analogique.

Exemple4.7

Pour produire un son, un DAC convertit un signal numérique en un signal analogique qui peut être envoyé à un haut-parleur.

6. `Timers(Temporisateurs)/Counters(Compteurs)`

- Les `temporisateurs` sont utilisés pour créer des délais dans les programmes ou pour générer des signaux périodiques.
- Les `compteurs` sont utilisés pour compter des événements.

Exemple4.8

Un timer peut être utilisé pour faire une action toutes les secondes, comme clignoter une LED toutes les 1 seconde.

Exemple4.9

Un compteur peut être utilisé pour compter le nombre de tours d'un moteur.

7. `Microprocesseur (CPU)`

- Le microprocesseur est le "cerveau" du microcontrôleur. Il exécute le programme, effectue des calculs et prend des décisions.

Exemple4.10

Si un programme ordonne au microcontrôleur de "lire la température et afficher la valeur", c'est le microprocesseur qui le fait.

8. `UART/USART/SPI/I2C/USB`

Ce sont des protocoles de communication qui permettent au microcontrôleur d'interagir avec d'autres appareils.

- `UART/USART` : Permet d'envoyer et recevoir des données série.
- `SPI` et `I2C` : Permettent de connecter plusieurs appareils ensemble.
- `USB` : Utilisé pour connecter des appareils comme des claviers ou des souris.

Exemple4.11

Un microcontrôleur peut utiliser `UART` pour envoyer des données à un autre appareil, comme un micro-ordinateur.

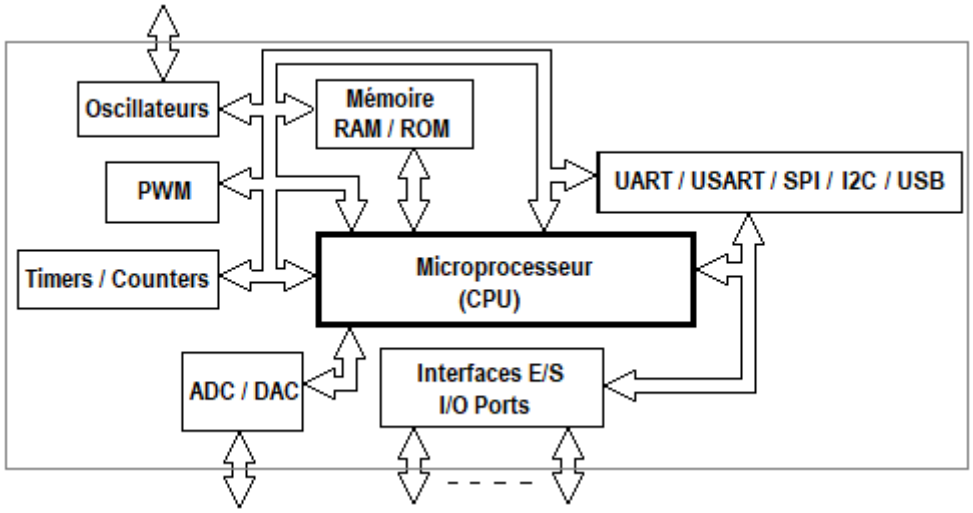


Figure4.1 *les éléments principaux d'un microcontrôleur*

➢ **Fondamental**

Dans le cas du microcontrôleur, l'ensemble du système à microprocesseur (voir figure 4.1) peut être intégré dans un unique circuit intégré. Nous aurons donc un seul composant intégré renfermant l'ensemble de ce système informatique.

4.2.2 Types de boîtiers de microcontrôleurs

Les microcontrôleurs existent dans plusieurs types de boîtiers, chacun ayant des caractéristiques qui les rendent plus ou moins adaptés à certaines applications.

- **DIP (Dual Inline Package):**
  Broches de chaque côté.
  Facile à manipuler et à souder.
- **SOIC (Small Outline Integrated Circuit):**
  Plus petit, broches sur les côtés.
  Plus difficile à souder que le DIP.
- **QFP (Quad Flat Package) / TQFP (Thin QFP):**
  Broches sur les 4 côtés.
  Plus compact, mais un peu plus compliqué à souder.
- **MLP (Micro Lead Frame Package)** :
  Composant plat avec des broches sur les côtés.
  Bon pour économiser de l'espace.
- **BGA (Ball Grid Array)** :
  Pas de broches visibles, seulement des billes de soudure sous le composant.
  Très petit et puissant, mais difficile à souder.

- **PGA (Pin Grid Array)** :
  Broches sous forme de grille.
  Facile à souder, mais un peu plus grand.

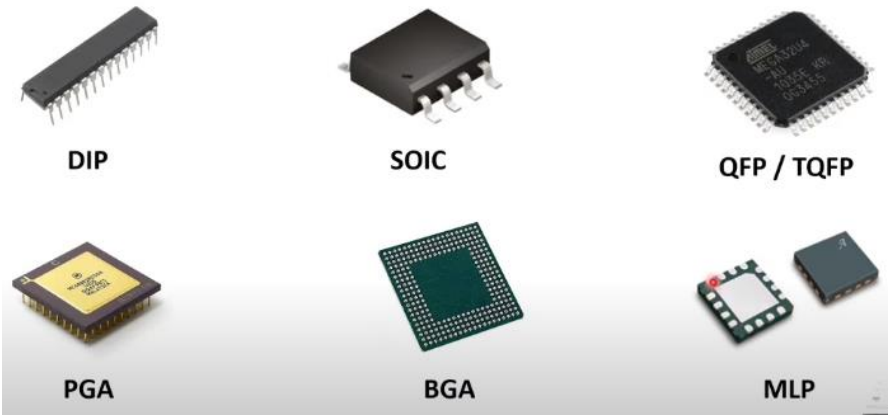


Figure4.2 : Différents types de boîtiers de microcontrôleurs

4.2.3 Principaux domaines d'application des microcontrôleurs

1. Systèmes embarqués

Les microcontrôleurs sont largement utilisés dans les systèmes embarqués, qui sont des systèmes électroniques intégrés dans des appareils pour exécuter des tâches spécifiques. Ces systèmes incluent :

- `Appareils électroménagers` : Contrôle des fonctions dans les réfrigérateurs, lave-linge, et autres appareils.
- `Automobile` : Gestion des moteurs, systèmes de sécurité, et contrôle des dispositifs d'info divertissement.
- `Électronique grand public` : Télécommandes, jouets électroniques, et dispositifs portables.

2. Automatisation industrielle

Dans le secteur industriel, les microcontrôleurs sont utilisés pour :

- `Contrôle de processus` : Surveillance et régulation des machines et des systèmes de production.
- `Systèmes de contrôle de qualité` : Analyse des données en temps réel pour assurer la conformité des produits.

3. Communication

Les microcontrôleurs jouent un rôle clé dans les systèmes de communication, notamment :

- `Protocoles de communication` : Gestion des communications série (UART, I2C, SPI) et des réseaux sans fil (Bluetooth, Wi-Fi).
- `Dispositifs de réseau` : Routeurs, modems, et autres équipements de communication.

4. Médecine

Dans le domaine médical, les microcontrôleurs sont utilisés dans :

- `Appareils de diagnostic` : Équipements pour surveiller les signes vitaux et analyser les échantillons.
- `Dispositifs implantables` : Pacemakers et autres dispositifs médicaux nécessitant un contrôle précis.

5. Internet des objets (IoT)

Les microcontrôleurs sont au cœur des dispositifs IoT, permettant :

- `Connectivité` : Intégration de capteurs et d'actionneurs pour collecter et transmettre des données.
- `Automatisation domestique` : Systèmes de gestion de la maison intelligente, comme l'éclairage et la sécurité.

6. Éducation et recherche

Les microcontrôleurs sont également utilisés dans les environnements éducatifs pour :

- `Projets d'apprentissage` : Enseignement de la programmation et de l'électronique à travers des kits de développement.
- `Prototypes de recherche` : Développement de nouveaux dispositifs et technologies.

4.2.4 Familles de microcontrôleurs

Les microcontrôleurs se classifient en plusieurs familles, chacune ayant des caractéristiques spécifiques adaptées à différentes applications : nous présentons un aperçu des principales familles de microcontrôleurs :

1. Famille PIC

`Baseline` : Utilise un mot d'instruction de 12 bits, avec une architecture simple, idéale pour les applications de base. Exemples de numéros de référence : `10Fxxx, 12Fxxx, 16Fxxx`.

`Mid-Range` : Offre des fonctionnalités avancées avec un mot d'instruction de 14 bits. Exemples de numéros de référence : `10Fxxx, 12Fxxx, 16Fxxx`.

`Enhanced Mid-Range` : Améliore les performances tout en restant compatible avec la famille Mid-Range. Exemples de numéros de référence : `12F1xxx, 16F1xxx`.

`High-End` : Utilise un mot d'instruction de 16 bits, avec des fonctionnalités avancées comme CAN, USB, et Ethernet. Exemples de numéros de référence : `18Fxxxx, 18FxxJxx, 18FxxKxx`.

2. Famille AVR

Conçue par Atmel, cette famille est populaire pour sa simplicité et sa flexibilité. Les `microcontrôleurs AVR` sont souvent utilisés dans des applications d'éducation et de prototypage, notamment avec des plateformes comme `Arduino`.

3. Famille ARM

Les `microcontrôleurs ARM` sont basés sur l'architecture ARM et sont largement utilisés dans des applications nécessitant des performances élevées et une faible

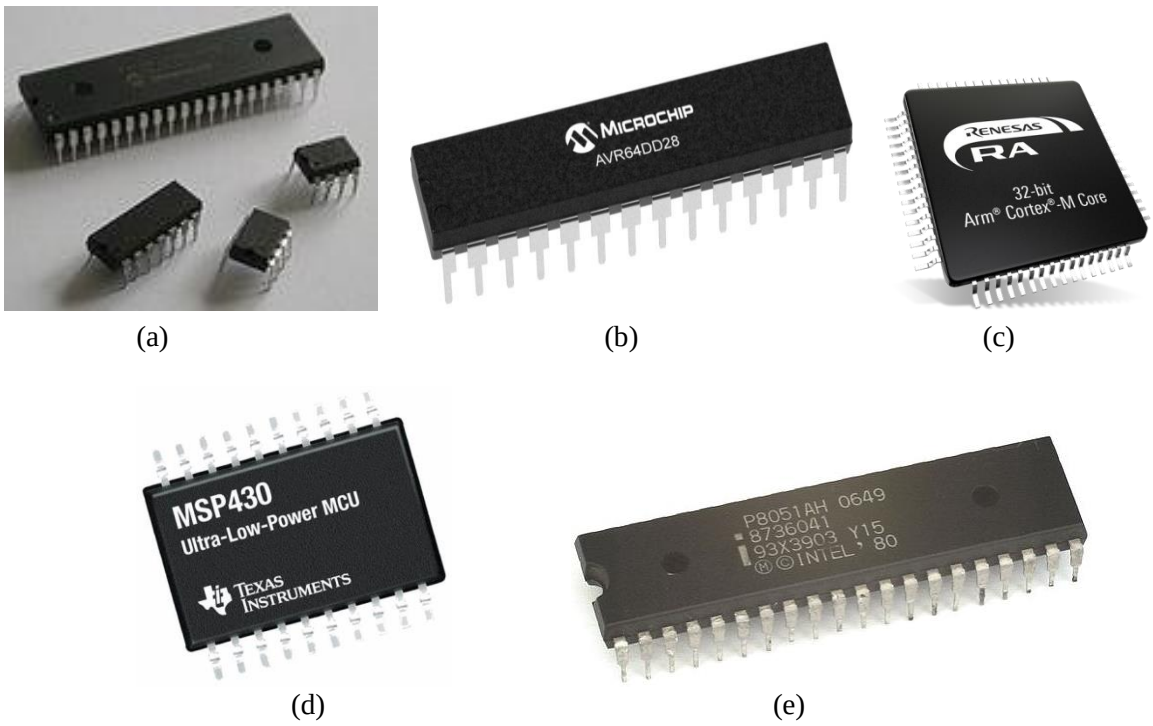
consommation d'énergie. Ils sont présents dans de nombreux dispositifs mobiles et embarqués.

4. Famille MSP430

Développée par Texas Instruments, cette famille est connue pour sa faible consommation d'énergie, ce qui la rend idéale pour les applications portables et les capteurs.

5. Famille 8051

Basée sur l'architecture 8051, cette famille est largement utilisée dans les systèmes embarqués. Elle est appréciée pour sa simplicité et sa large disponibilité sur le marché.



(a) (b) (c)

(d) (e)

Figure4.3 *la famille de microcontrôleurs*
(a) *PIC* (*microcontrôleurs de la marque Microship)*
(b) *PIC de type AVR* (c) *microcontrôleurs ARM*
(d) *MSP430 Microcontrollers* (e) *Famille 8051*

```
Chaque famille de microcontrôleurs a ses propres avantages et
inconvénients, et le choix d'une famille dépend souvent des
exigences spécifiques de l'application, telles que la
puissance de traitement, la consommation d'énergie, et les
interfaces disponibles.
```

Exemples de sociétés fabriquant des microcontrôleurs :

1. Microchip Technology Inc.
- Microchip est un fabricant de semi-conducteurs, notamment de microcontrôleurs et de circuits intégrés.

2. Atmel (acquise par Microchip)
- Atmel était un fabricant majeur de semi-conducteurs, particulièrement reconnu pour ses microcontrôleurs AVR (Alf and Vegard's RISC processor).
- Atmel a été acquis par Microchip en 2016.

3. NXP Semiconductors
- NXP est une entreprise leader dans le domaine des semi-conducteurs, offrant une large gamme de produits, y compris des microcontrôleurs et des solutions pour l'industrie automobile, l'électronique grand public, et d'autres secteurs.

4. Texas Instruments
- Texas Instruments (TI) est un fabricant mondial de semi-conducteurs et de solutions intégrées.
- Ils produisent une variété de produits, y compris des microcontrôleurs, des processeurs et des dispositifs analogiques.

5. Cypress (acquise par Infineon Technologies)
- Cypress était spécialisé dans les semi-conducteurs, notamment les microcontrôleurs PSoC (Programmable System-on-Chip).
- Cypress a été acquise par Infineon Technologies en 2020.

6. STMicroelectronics
- STMicroelectronics est un leader mondial dans la fabrication de semi-conducteurs.
- Ils produisent une vaste gamme de produits, y compris des microcontrôleurs, des capteurs et des dispositifs de puissance.

`Chacune de ces entreprises joue un rôle significatif dans l'industrie des semi-conducteurs, fournissant des solutions électroniques pour diverses applications et secteurs.`

4.3 Programmation du microcontrôleur

Avant de passer au sous-chapitre de la programmation du microcontrôleur, il est important de rappeler que dans la première section de ce chapitre, nous avons étudié l'architecture des microcontrôleurs. Pour illustrer ces concepts, nous avons pris comme exemples le `PIC 16F877A` et `l'Arduino Uno`. Ces deux plateformes nous permettront de mieux comprendre les principes de programmation qui seront abordés dans la suite de ce chapitre. En effet, en comparant ces deux microcontrôleurs, nous pourrons apprécier leurs différences

et similitudes, ce qui facilitera notre apprentissage des techniques de programmation spécifiques à chacun d'eux.

4.3.1 Éléments essentiels pour la programmation

- **Environnements de développement**

Chaque famille de microcontrôleurs dispose de son propre environnement de développement intégré (IDE) et de ses outils de programmation :

- `PIC` : `Utilise MPLAB X`, qui est un IDE basé sur NetBeans, offrant des fonctionnalités avancées pour le développement et le débogage. Il est aussi possible de programmer le `PIC` avec `mikroC for PIC`.

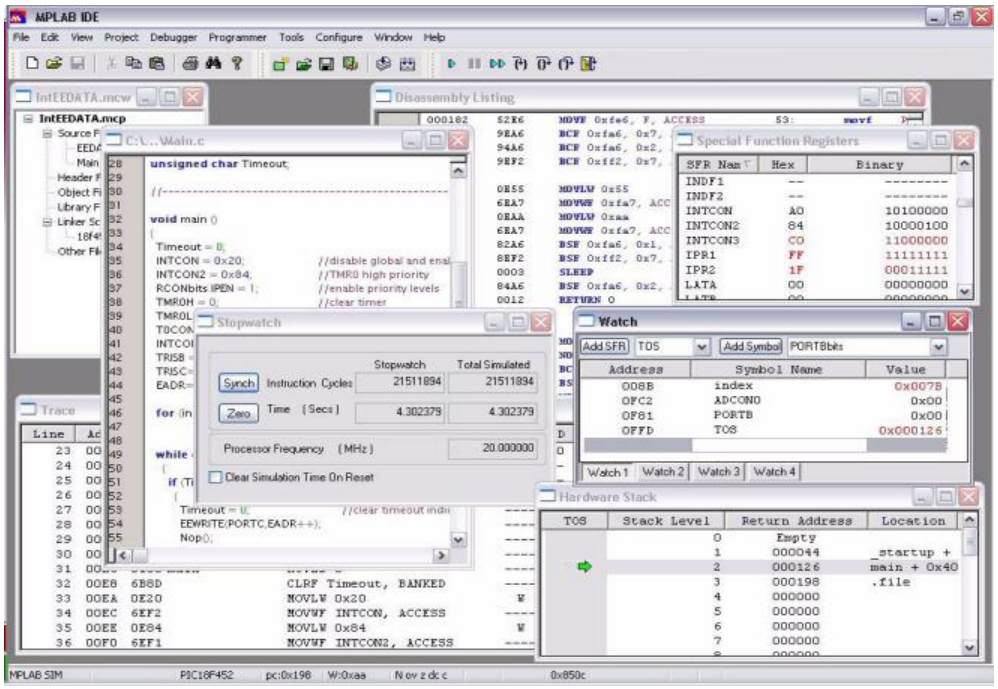
- `Arduino` : Utilise `Arduino IDE`, un environnement de développement simple et facile à utiliser, qui prend en charge le langage **C/C++**. Le code peut être compilé et téléversé via un convertisseur USB-to-serial intégré ou un programmateur externe comme `AVRISP`.

- `AVR` : `Atmel Studio` (maintenant Microchip Studio) est l'IDE recommandé, `avec un support pour le langage C et l'assembleur`. Les utilisateurs peuvent également utiliser l'outil avr-gcc pour compiler le code C.

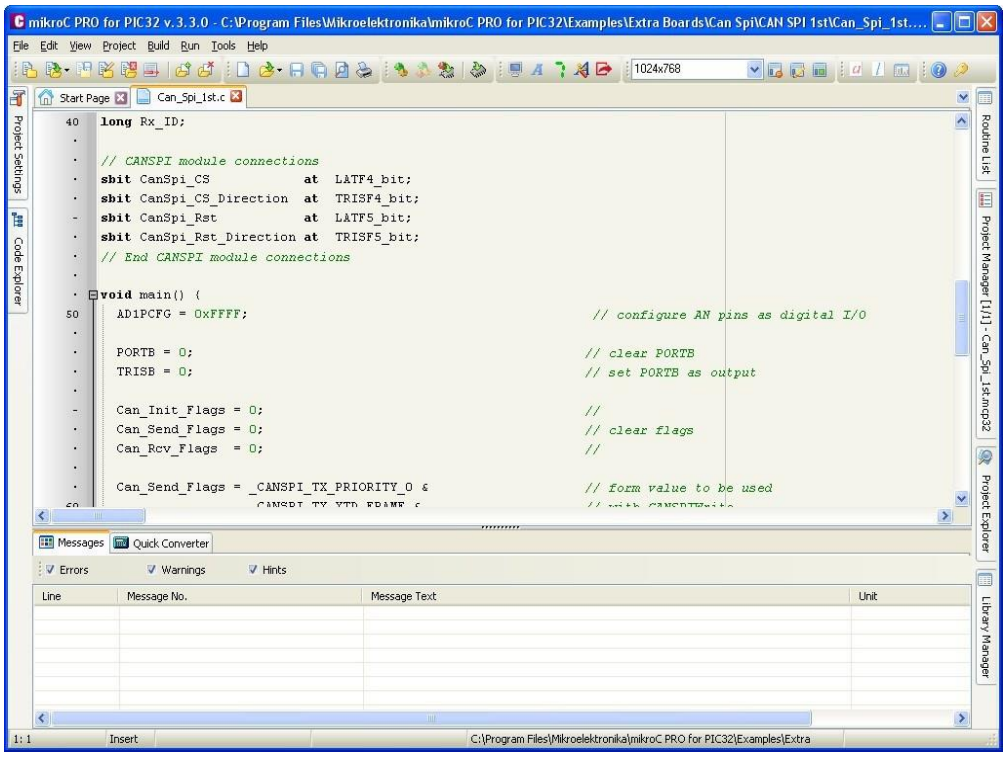
- `ARM` : Les microcontrôleurs ARM peuvent être programmés avec des `IDE comme Keil, IAR Embedded Workbench`, ou des outils open-source comme `GCC` pour ARM.

- `MSP430`: Texas Instruments propose `Code Composer Studio (CCS)` pour le développement sur MSP430, qui inclut des outils de débogage et de simulation.

- `8051` : Divers `IDE` sont disponibles, y compris `Keil uVision`, qui est populaire pour le développement sur cette architecture.



(a)



(b)

Figure4.4 *Interface de l'environnement de développement pour les PIC*
(a) *MPLAB (assembleur)* (b) *mikroC for PIC (langage C)*

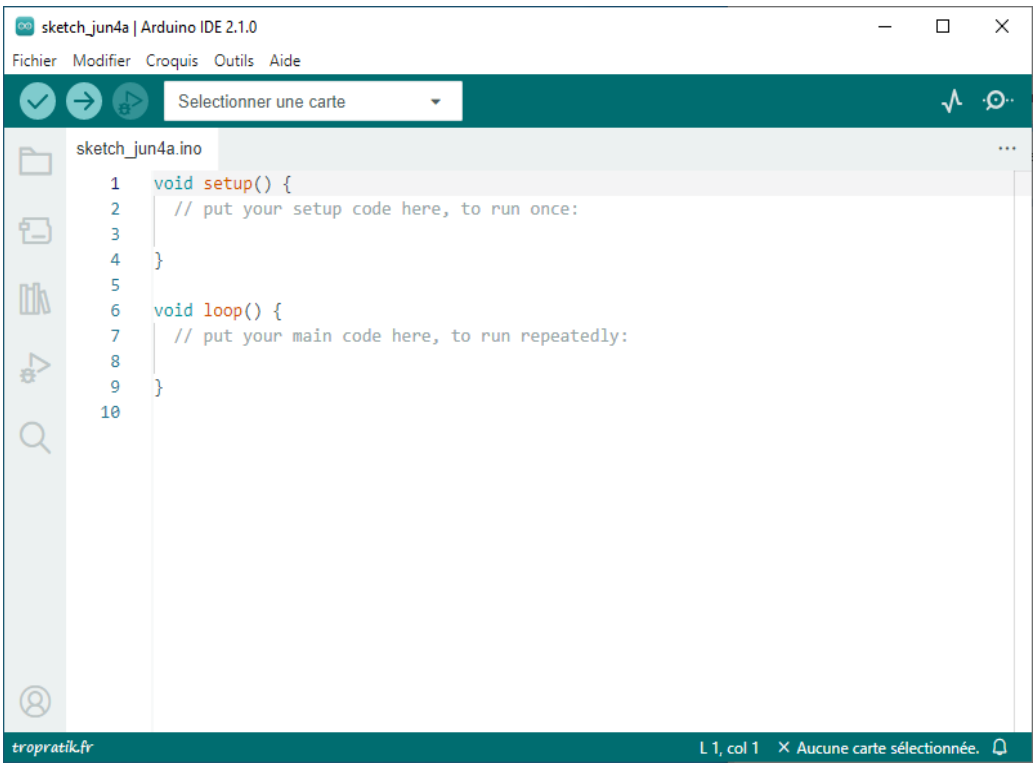


Figure4.5 *Interface de l'environnement de développement pour Arduino(langage C)*

- **Langages de programmation**

Les `microcontrôleurs` peuvent être `programmés` dans plusieurs langages, les plus courants étant :

- `C : Le langage C` est largement utilisé en raison de sa portabilité et de sa capacité à interagir directement avec le matériel.

- `Assembleur` : Bien que plus complexe, la programmation en assembleur permet un contrôle précis des ressources matérielles et est souvent utilisée pour des applications critiques en termes de performance.

- `C++` : Utilisé dans certains environnements, notamment pour les projets plus complexes qui nécessitent une programmation orientée objet.

- **Outils de programmation**

Pour programmer un microcontrôleur, il est nécessaire d'utiliser des outils spécifiques :

`Programmers` : Des dispositifs comme le :

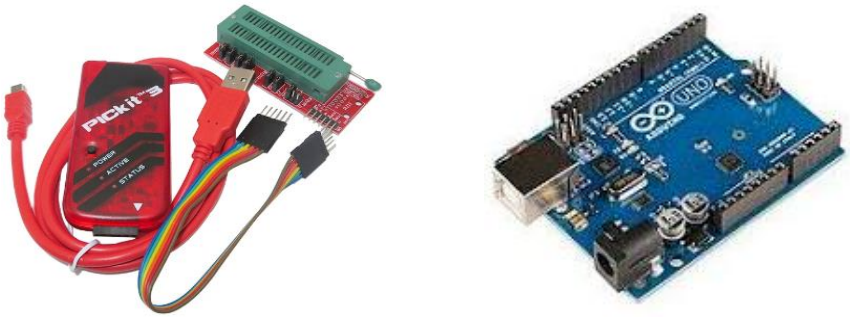
- `PICkit pour les PIC` (figure 4.6.a).

- AVRISP pour les AVR (L'**Arduino Uno** utilise un microcontrôleur ATmega328P, qui peut être programmé via un programmateur USB-to-serial intégré sur la carte (figure4.6.b).

- JTAG pour les ARM

Ces dispositifs sont utilisés pour transférer le code sur le microcontrôleur.

Débogueurs : Les débogueurs permettent de tester et de corriger le code en temps réel, facilitant le développement.



(a) (b)

Figure4.6 *Exemple de programmateur*

(a) *PICKIT-3 de microcontrôleurs* PIC (b) *Carte arduino UNO*

La programmation des microcontrôleurs, tels que les familles PIC, AVR, ARM, MSP430 et 8051, implique plusieurs étapes et considérations spécifiques à chaque architecture. Nous présentons un aperçu des principales méthodes et outils utilisés pour programmer un microcontrôleur.

N°	Etape	description
1	Créer un algorithme ou un organigramme	Avant de coder, il est important de définir les étapes du programme sous forme d'algorithme ou d'organigramme. Cela aidera à visualiser le déroulement du programme.
2	Choisir un microcontrôleur	Dans notre cas, nous allons utiliser le PIC 16F877A ou l'Arduino Uno.
3	Rassembler le matériel nécessaire	microcontrôleur, un programmateur (si nécessaire), des composants électroniques (comme des LEDs, résistances, etc.), et des câbles de connexion.

4	Installer l'environnement de développement	Installation d' un IDE (Environnement de Développement Intégré) approprié, comme MPLAB X pour les microcontrôleurs PIC ou Arduino IDE pour les cartes Arduino.
5	Écrire le code (programme)	Rédigez le code source dans le langage de programmation approprié (C pour PIC, C/C++ pour Arduino) en suivant l'algorithme ou l'organigramme que vous avez créé.
6	Compiler le code	Utilisation de l'IDE pour compiler le code et vérifier qu'il n'y a pas d'erreurs
7	Connecter le microcontrôleur	Connection à l'ordinateur via un programmateur ou un câble USB
8	Téléverser le code sur le microcontrôleur	Vérifiez que le code compilé sur le microcontrôleur est assuré
9	Tester le programme	Vérifiez que le microcontrôleur fonctionne comme prévu. Observez le comportement des composants connectés
10	Expérimenter et itérer	Modifiez le code pour ajouter de nouvelles fonctionnalités ou améliorer le programme
11	Documenter le travail	Prise des notes sur le code, les connexions et les résultats pour référence future

Exemple de circuit réalisé après la programmation du `microcontrôleur PIC 16F877A` et sa connexion aux composants appropriés du système.

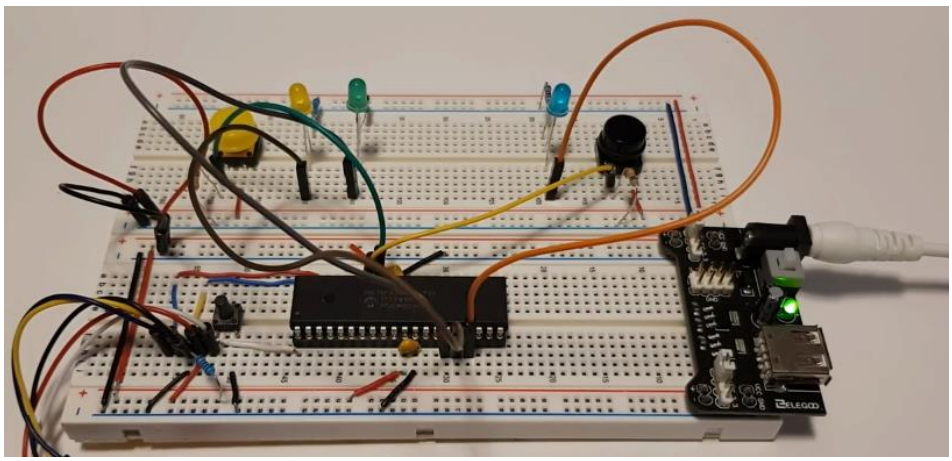


Figure4.7 *Exemple de circuit à base microcontrôleur*

Dans ce qui suit, pour appliquer les étapes mentionnées ci-dessus concernant les éléments de base de la programmation, nous utiliserons le microcontrôleur PIC16F877A ainsi que le microcontrôleur de la carte Arduino Uno. Les deux approches de programmation seront explorées : le langage assembleur pour le PIC et le langage C pour l'Arduino.

4.3.2 Algorithme pour un programme de microcontrôleur

Dans cette partie, nous allons apprendre à établir une structure sous forme d'étapes claires et généralisées. Cela permet de la traduire ou de l'adapter facilement dans un programme, quel que soit le langage ou le microcontrôleur utilisé. Le principe d'un algorithme général pour structurer un programme destiné à un microcontrôleur se présente comme suit :

Etape_1 : Initialisation

- Définir les constantes et les variables nécessaires.
- Configurer les ports d'entrée/sortie (GPIO) selon les besoins du projet.
- Initialiser les périphériques (par exemple, timers, communications série).

Etape_2 : Boucle Principale

- Lire les entrées : Vérifier l'état des capteurs ou des boutons.
- Traiter les données : Exécuter les algorithmes nécessaires en fonction des entrées lues.
- Contrôler les sorties : Mettre à jour l'état des sorties (LED, moteurs, etc.) en fonction des résultats du traitement.
- Attendre : Inclure un délai si nécessaire pour éviter une exécution trop rapide (par exemple, utiliser un timer).

Etape_3 : Gestion des Interruptions (si applicable)

- Configurer les interruptions pour gérer des événements asynchrones (par exemple, un bouton pressé).
- Écrire des routines de service d'interruption pour traiter ces événements.

Etape_4 : Fin du Programme

Exemple 4.12

Nous allons voir un algorithme simple pour allumer une LED en appuyant sur un bouton

Étape 1 : Initialisation

1. Définir une variable pour l'état du bouton.
2. Configurer l'une des broches d'un port du microcontrôleur comme **sortie** (pour la LED).

3. Configurer une autre broche comme **entrée** (pour le bouton).

Étape 2 : Boucle Principale

1. Lire l'état du bouton (broche configurée en entrée).
2. Si le bouton est pressé, allumer la LED (activer la broche configurée en sortie).
3. Sinon, éteindre la LED (désactiver la broche configurée en sortie).

Ainsi, à travers les étapes de cet exemple, il est possible d'écrire un programme adapté à un langage spécifique pour un microcontrôleur donné.

4.3.3 Application des microcontrôleurs

❖ Application du microcontrôleur PIC 16F877A

1) Prise de connaissance de l'environnement de développement et du langage de programmation du 16f877A.

Nous allons développer une simple application allumage d'une LED à la sortie RB0 du microcontrôleur PIC16F877A, en utilisant les concepts suivants :

1. Son environnement de développement (IDE)
2. Ses langages de programmation (assembleur avec MPLAB X et C avec mikroC for PIC)

D'abord, Commençons par examiner en détail l'Environnement de Développement pour le PIC16F877A.

Le PIC16F877A est un microcontrôleur populaire de la série PIC de Microchip. Pour développer des applications avec ce microcontrôleur, nous aurons besoin de MPLAB X IDE et mikroC for PIC :

- **MPLAB X IDE**

  - **Description :**

    MPLAB est un IDE (Environnement de Développement Intégré) basé sur NetBeans, spécialement conçu pour le développement des microcontrôleurs PIC. Il permet de programmer, déboguer et simuler les applications. Il supporte plusieurs langages de programmation, y compris l'assembleur, le langage C, et C++.

  - **Installation de MPLAB X :**

1. Nous avons besoin de télécharger `MPLAB X IDE` sur le site officiel de Microchip.
2. Puis, l'installer en suivant les instructions du programme d'installation.
3. Comme on peut télécharger également le `MPLAB X IDE` + `MPLAB XC8 compiler` pour utiliser le langage C.

- **mikroC for PIC**

`mikroC for PIC` est un compilateur spécifique pour les microcontrôleurs `PIC`. Ce langage **C** est simple à utiliser, avec de nombreuses bibliothèques et un code facile à comprendre.

2) Prise de connaissance du microcontrôleur 16f877A.

`Avant de passer à la programmation d'un microcontrôleur, il est essentiel de connaître trois éléments, sans lesquels la programmation serait impossible :`

- brochage du microcontrôleur (figure4.8)
- les différentes bank pour connaître ou sont situés les différents registres à programmer (figure4.12)
- les différentes fonctions des registres (figure4.9)

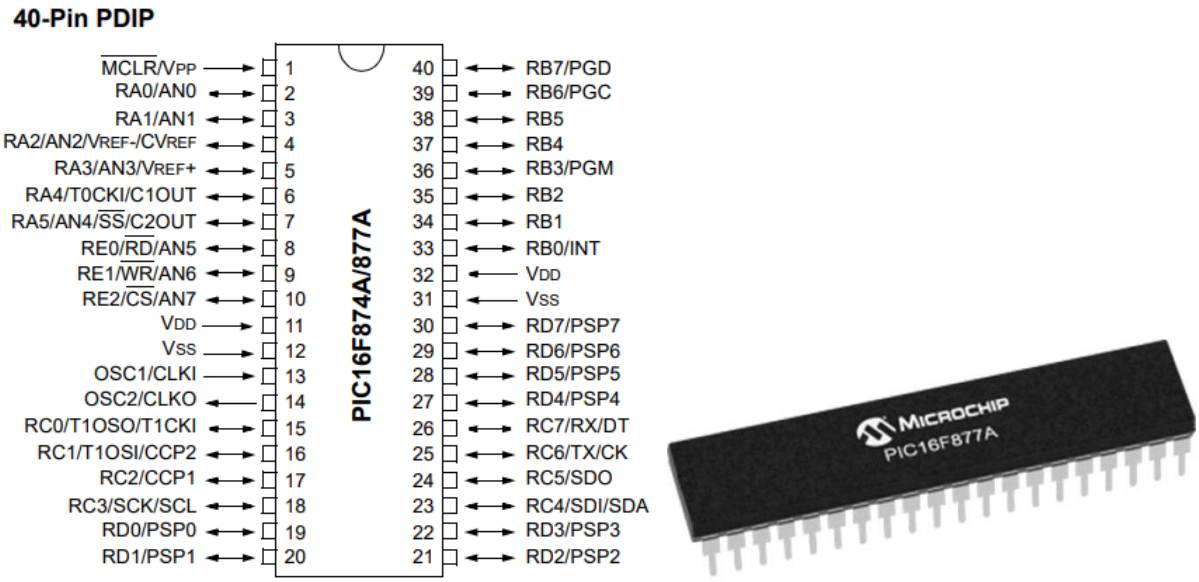


Figure4.8 *Exemple de brochage d'un microcontrôleur (16F877)*

Pour une meilleure appréhension du principe, il est instructif de se référer à l'ensemble des registres qui composent le microcontrôleur 16F877.

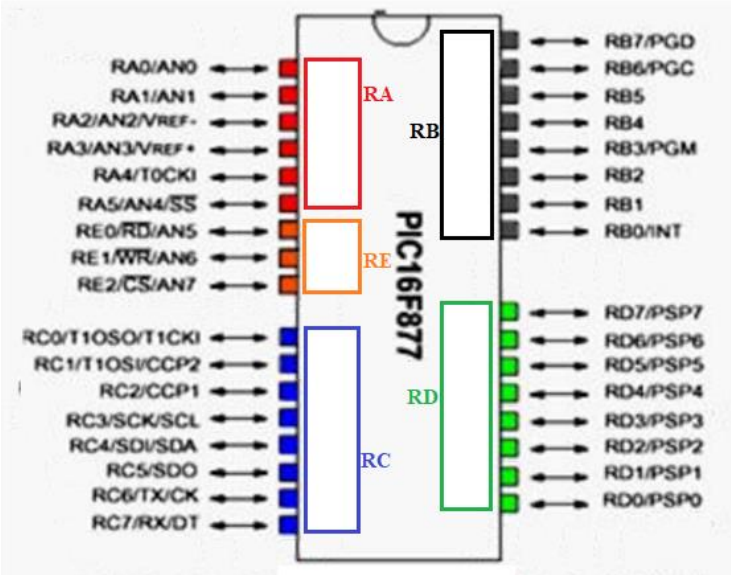


Figure 4.9 Les registres *du microcontrôleur 16f877*

Nous avons cinq registres : RA, RB, RC, RD et RE. Ce qui permet de les utiliser à la fois en mode entrée et en mode sortie.

Exemple 4.13

L'algorithme de l'exemple 4.12 est traduit en code en utilisant mikroC for PIC pour le PIC16F877A.

```
void main() {
   // Étape 1 : Initialisation
   TRISB = 0x02;          // Configure RB0 comme sortie (LED), RB1 comme entrée (bouton)
   PORTB = 0x00;          // Initialise le port B (éteint la LED)

   // Étape 2 : Boucle Principale
   while(1) {
      if (PORTBbits.RB1 == 1) {     // Si le bouton est pressé (RB1 = 1)
         PORTBbits.RB0 = 1;         // Allumer la LED (RB0 = 1)
      } else {                      // Sinon, si le bouton n'est pas pressé
         PORTBbits.RB0 = 0;         // Éteindre la LED (RB0 = 0)
      }
   }
}
```

Exemple 4.14

Nous allons examiner un autre cas où seul le registre RB du PIC16F877A, et plus précisément la broche RB0, est utilisé pour allumer la LED.

Ainsi, l'objectif de cet exemple est d'allumer une LED connectée à la broche RB0. Pour cela, les éléments requis sont les suivants :

- 1 LED-rouge
- Résistante de résistance R à calculer
- Microcontrôleur 16F877A
- Quartz de 8 Mhz , et 2 condensateurs de *22pF*

`Calcul du circuit:`

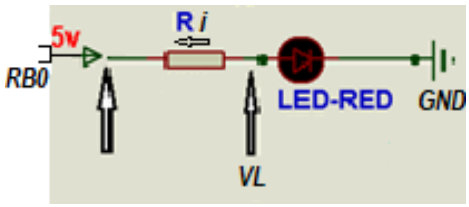


Figure4.10 *Circuit connecté à la broche RB0 du microcontrôleur*

La LED rouge fonctionne sous une tension de 1,76 V et un courant de 15 mA. La tension délivrée par une broche du microcontrôleur est de 5 V. Dans ce cas, calculons la valeur de la résistance *R*.

$$R = \frac{5V - VL}{15\,mA} = 213.33 \cong 220\,\Omega$$

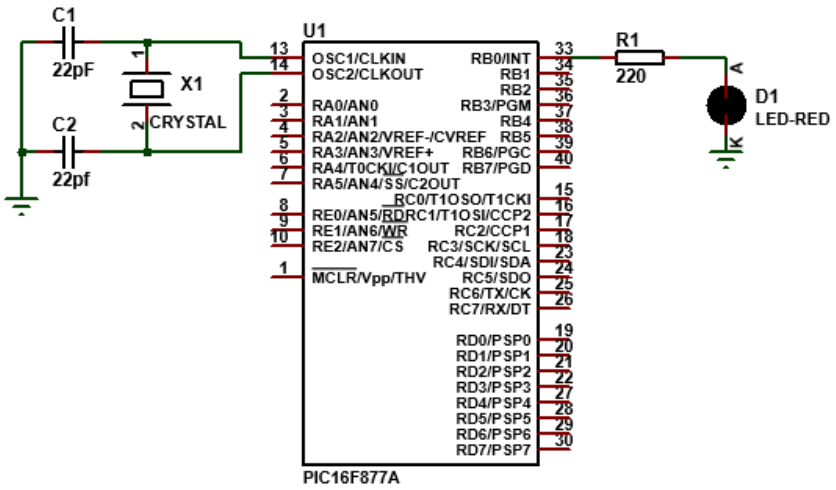


Figure 4.11 *Connexion d'une LED au microcontrôleur PIC16F877A*

URE 2-3: PIC16F876A/877A REGISTER FILE MAP

File Address	Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3
00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h	Indirect addr.(*)
01h	TMR0	81h	OPTION_REG	101h	TMR0	181h	OPTION_REG
02h	PCL	82h	PCL	102h	PCL	182h	PCL
03h	STATUS	83h	STATUS	103h	STATUS	183h	STATUS
04h	FSR	84h	FSR	104h	FSR	184h	FSR
05h	PORTA	85h	TRISA	105h		185h	
06h	PORTB	86h	TRISB	106h	PORTB	186h	TRISB
07h	PORTC	87h	TRISC	107h		187h	
08h	PORTD(1)	88h	TRISD(1)	108h		188h	
09h	PORTE(1)	89h	TRISE(1)	109h		189h	
0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah	PCLATH
0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh	INTCON
0Ch	PIR1	8Ch	PIE1	10Ch	EEDATA	18Ch	EECON1
0Dh	PIR2	8Dh	PIE2	10Dh	EEADR	18Dh	EECON2
0Eh	TMR1L	8Eh	PCON	10Eh	EEDATH	18Eh	Reserved(2)
0Fh	TMR1H	8Fh		10Fh	EEADRH	18Fh	Reserved(2)
10h	T1CON	90h		110h	General Purpose Register 16 Bytes	190h	General Purpose Register 16 Bytes
11h	TMR2	91h	SSPCON2	111h		191h	
12h	T2CON	92h	PR2	112h		192h	
13h	SSPBUF	93h	SSPADD	113h		193h	
14h	SSPCON	94h	SSPSTAT	114h		194h	
15h	CCPR1L	95h		115h		195h	
16h	CCPR1H	96h		116h		196h	
17h	CCP1CON	97h		117h		197h	
18h	RCSTA	98h	TXSTA	118h		198h	
19h	TXREG	99h	SPBRG	119h		199h	
1Ah	RCREG	9Ah		11Ah		19Ah	
1Bh	CCPR2L	9Bh		11Bh		19Bh	
1Ch	CCPR2H	9Ch	CMCON	11Ch		19Ch	
1Dh	CCP2CON	9Dh	CVRCON	11Dh		19Dh	
1Eh	ADRESH	9Eh	ADRESL	11Eh		19Eh	
1Fh	ADCON0	9Fh	ADCON1	11Fh		19Fh	
20h	General Purpose Register 96 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h	General Purpose Register 80 Bytes
7Fh		EFh	accesses 70h-7Fh	16Fh	accesses 70h-7Fh	1EFh	accesses 70h - 7Fh
		F0h		170h		1F0h	
		FFh		17Fh		1FFh	

Unimplemented data memory locations, read as '0'.
* Not a physical register.

Figure4.12 *Tableau des banques de mémoire du 16f877A*

TABLE 2-1: SPECIAL FUNCTION REGISTER SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 0											
00h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
01h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
02h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
03h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
04h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
05h	PORTA	—	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	43, 150
06h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
07h	PORTC	PORTC Data Latch when written: PORTC pins when read								xxxx xxxx	47, 150
08h(4)	PORTD	PORTD Data Latch when written: PORTD pins when read								xxxx xxxx	48, 150
09h(4)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	49, 150
0Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
0Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
0Ch	PIR1	PSPIF(3)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	26, 150
0Dh	PIR2	—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF	-0-0 0--0	28, 150
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{T1SYNC}$	TMR1CS	TMR1ON	--00 0000	57, 150
11h	TMR2	Timer2 Module Register								0000 0000	62, 150
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	61, 150
13h	SSPBUF	Synchronous Serial Port Receive Buffer/Transmit Register								xxxx xxxx	79, 150
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000	82, 82, 150
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	63, 150
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	63, 150
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	64, 150
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	112, 150
19h	TXREG	USART Transmit Data Register								0000 0000	118, 150
1Ah	RCREG	USART Receive Data Register								0000 0000	118, 150
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	63, 150
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	63, 150
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	64, 150
1Eh	ADRESH	A/D Result Register High Byte								xxxx xxxx	133, 150
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	$\overline{GO/DONE}$	—	ADON	0000 00-0	127, 150

Figure4.13 *Fonctions des registres du pic16f877A*

3. Programmation en Assembleur avec MPLAB X

a) Objectif : Nous allons programmer le PIC16F877A en **assembleur** pour allumer la LED connectée à **RB0**.

b) Étapes à suivre :

1. **Création d'un projet MPLAB X :**
   - Ouverture de `MPLAB X IDE`.
   - Création d'un projet `Assembler` pour le `PIC16F877A`.

Code en assembleur(MPLAB) pour allumer la LED

```
ORG 0x00       ; Spécifie l'adresse de départ du programme (ici à 0x00, l'adresse par défaut pour
                ;les microcontrôleurs PIC
GOTO START     ; Sauter à l'étiquette START pour commencer l'exécution du programme

START:
BSF STATUS, RP0    ; Sélectionne la banque 1 pour accéder aux registres de configuration du
                    microcontrôleur)
MOVLW 0x00         ; Charger la valeur 0x00 dans le registre W
MOVWF TRISB        ; Configurer le port B en mode sortie (tous les bits à 0 pour RB0 à RB7)
                ; Le registre TRISB détermine la direction des pins du port B (0 = sortie, 1 = entrée).
BCF STATUS, RP0    ; Reviens à la banque 0 pour pouvoir manipuler les registres généraux
BSF PORTB, 0       ; Met le bit 0 du registre PORTB à 1, ce qui allume la LED connectée à RB0

GOTO  START        ; Crée une boucle infinie pour empêcher que le programme
                ne   continue après avoir allumé la LED
END
```

4. Programmation en C avec mikroC for PIC

a) Objectif : Nous allons programmer le PIC16F877A en **C** pour allumer la LED sur la sortie **RB0**.

b) Étapes à suivre :

2. Ouverture **d'un projet mikroC for PIC :**
   - Ouverture du **mikroC for PIC**.
   - Création d'un projet pour le **PIC16F877A**.

Code en C (mikroC for PIC) pour allumer la LED

```
void main() {
  TRISB = 0x00;            // Configure tous les bits du port B comme sorties
  PORTBbits.RB0 = 1;    // Allume la LED sur RB0
  while(1);                // Boucle infinie pour garder la LED allumée
}
```

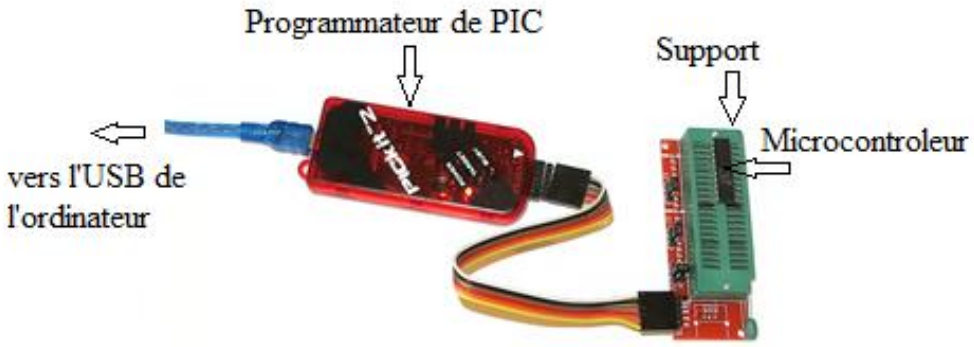


Figure4.14*Connexion pour le transfert du programme vers le microcontrôleur*

❖ Application du microcontrôleur de l'Arduino

Dans ce cas, le microcontrôleur est remplacé par une carte Arduino. La LED est connectée à l'une des broches numériques (2-13), configurée en mode sortie, afin de permettre son allumage. Ainsi, nous Connectons l'anode de la LED à une broche numérique (par exemple, broche 3, indiquée par la flèche jaune sur la figure4.15) via la résistance.

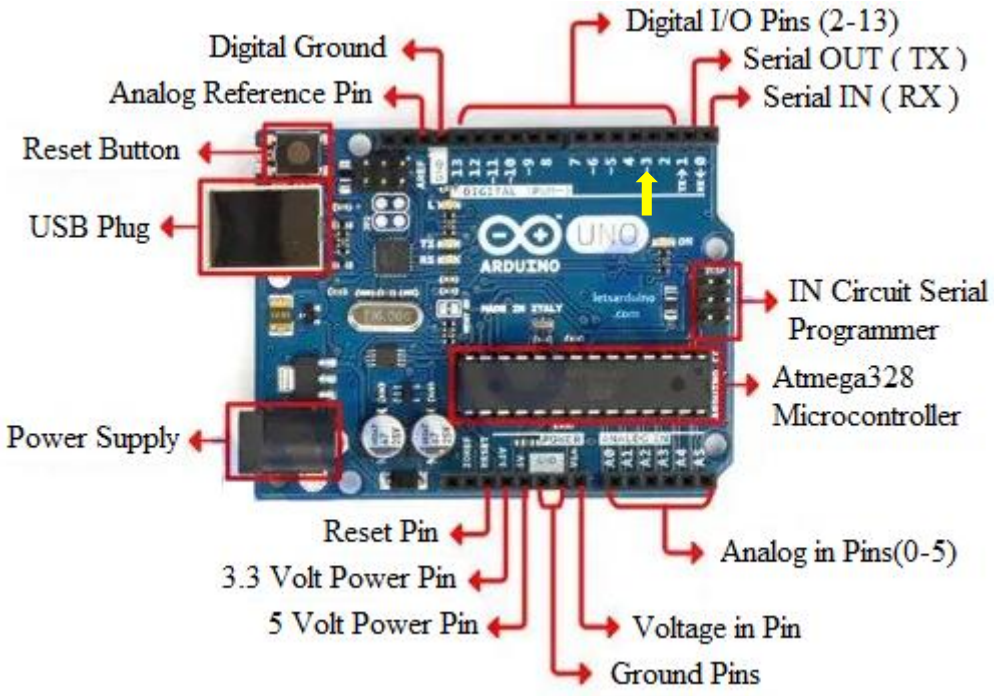


Figure4.15 *Brochage d'une Carte Arduino UN0*

Code en C (ARDUINO) pour allumer la LED

```
void setup () {
```

```
    pinMode (3, OUTPUT);  // Configure la broche 3 comme sortie
    digitalWrite (3, HIGH) ;   // Allume la LED
}
void loop() {
    // Rien à faire ici
}
```

Exercice4.1

Elaborer un programme qui déclenche le clignotement de la LED connectée à la broche RB0 du microcontrôleur 16F877.

Solution

Au lieu de rester allumée en continu, la LED clignote en s'allumant puis en s'éteignant alternativement. Le principe de fonctionnement est illustré dans l'organigramme ci-dessous.

Nous allons revisiter le schéma d'allumage d'une LED illustré dans la figure 4.11. Cependant, cette fois-ci, au lieu d'un allumage direct, la LED clignote alternativement, s'allumant puis s'éteignant. Le principe est décrit dans l'organigramme ci-dessous :

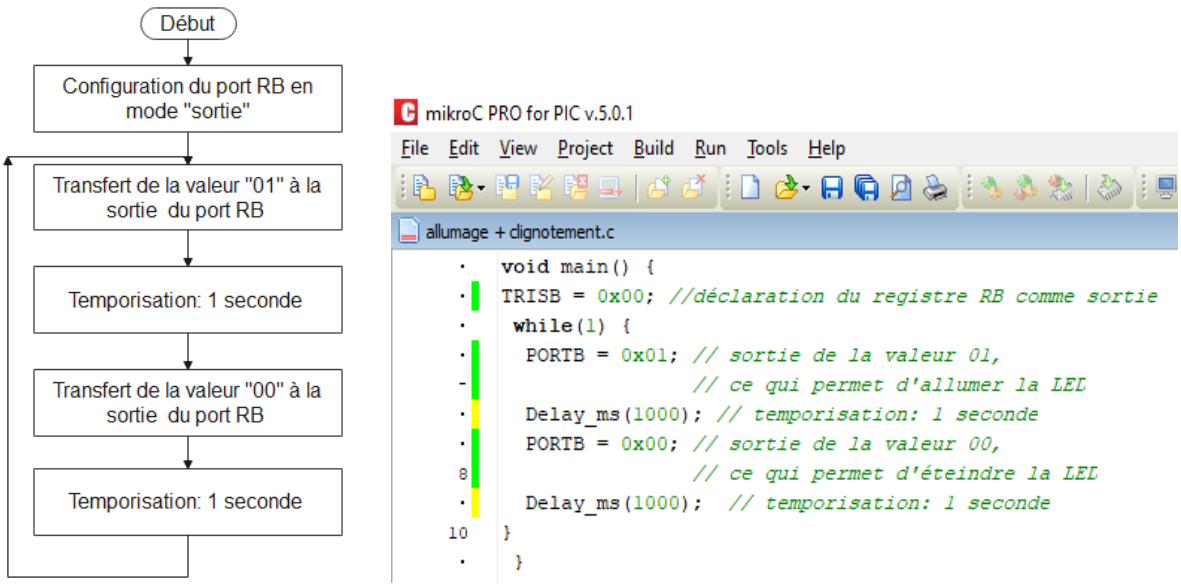


Figure 4.13 Environnement mikroC PRO - *Programmation de clignotement d'une LED*

Exercice4.2

Élaborer un programme qui allume une LED en réponse à l'action sur un interrupteur INT.

- La LED est connectée à la broche RB5.
- L'interrupteur est connecté à la broche RB0, par exemple.

Solution

Code en C (mikroC for PIC)

```
void main() {
   // Configuration des ports
   TRISB = 0b00000001; // RB0 en entrée, RB1 à RB7 en sortie
   PORTB = 0;         // Initialiser PORTB à 0 (toutes les sorties éteintes)

   while (1) {   // Boucle infinie
      if (PORTB.B0 == 1) {      // Si l'interrupteur est actionné (RB0 est à 1)
         PORTB.B5 = 1; // Allumer la LED (RB5 à 1)
      } else {
         PORTB.B5 = 0; // Éteindre la LED (RB5 à 0)
      }
   }
}
```

Code en C (ARDUINO) pour allumer la LED

```
// Définition des broches
const  int  ledPin = 5;          // LED connectée à la broche 5
const  int  switchPin = 2;      // Interrupteur connecté à la broche 2

void setup() {
   // Initialisation des broches
   pinMode(ledPin, OUTPUT);      // Configure la broche de la LED comme sortie
   pinMode(switchPin, INPUT);    // Configure la broche de l'interrupteur comme entrée
}

void loop() {
   // Lire l'état de l'interrupteur
   int switchState = digitalRead(switchPin);

   // Vérifier si l'interrupteur est actionné
   if (switchState == HIGH) {
      digitalWrite(ledPin, HIGH);     // Allumer la LED
   } else {
```

```
      digitalWrite(ledPin, LOW);     // Éteindre la LED
   }
}
```

➢ **Signification des Termes**

const :
Indique que la valeur de la variable ne changera pas. Par exemple, « const int ledPin = 5 »; signifie que ledPin est toujours 5.

int : C'est un type de donnée pour les nombres entiers. Ici, il est utilisé pour les numéros de broches.

ledPin et switchPin : Ce sont des noms de variables :
ledPin : La broche où la LED est connectée (ici, broche 5).
switchPin : La broche où l'interrupteur est connecté (ici, broche 2).

switchState : C'est une variable qui garde en mémoire si l'interrupteur est ON ou OFF.
Par exemple, si l'interrupteur est pressé, switchState peut être 1 (ON) ; sinon, c'est 0 (OFF).

digitalRead(pin) : C'est une fonction qui vérifie si une broche est allumée ou éteinte.
On utilise digitalRead pour savoir si l'interrupteur est pressé (ON) ou non (OFF).

digitalWrite(pin, value) : C'est une fonction qui allume ou éteint une broche.
Par exemple, digitalWrite(ledPin, HIGH); allume la LED, et digitalWrite(ledPin, LOW); l'éteint.

Exercice4.3

Traduire l'algorithme de l'exemple 4.12 en code assembleur en utilisant MPLAB pour le PIC16F877A.

```
; Programme en assembleur pour le PIC16F877A
; Allumer la LED sur RB0 lorsque le bouton sur RB1 est pressé

   ORG 0x00        ; Début du programme (adresse de départ)

   ; Étape 1 : Initialisation

   ; Configurer RB0 comme sortie et RB1 comme entrée
   BSF    STATUS, RP0    ; Sélectionner le Bank 1 (registres de configuration)
   MOVLW  0x02           ; 0x02 = 00000010 : RB0 en sortie, RB1 en entrée
   TRISB  MOVWF  TRISB   ; Mettre la valeur dans le registre TRISB

   ; Initialiser PORTB à 0 (éteindre la LED)
   CLRF   PORTB          ; Mettre tous les bits de PORTB à 0
```

```
   ; Étape 2 : Boucle Principale
main_loop:
   ; Lire l'état du bouton (RB1)
   BTFSS   PORTB, 1      ; Tester si RB1 est à 1 (bouton pressé)
   GOTO    button_non_pressé     ; Si RB1 = 0, bouton non pressé

   ; Si le bouton est pressé, allumer la LED (mettre RB0 à 1)
   BSF     PORTB, 0      ; Mettre RB0 à 1 (allumer la LED)
   GOTO    main_loop     ; Revenir au début de la boucle

button_non_pressé:
   ; Sinon, éteindre la LED (mettre RB0 à 0)
   CLRF    PORTB, 0      ; Mettre RB0 à 0 (éteindre la LED)
   GOTO    main_loop     ; Revenir au début de la boucle

   END
```

Exercice4.4 (à résoudre)

Élaborer un programme qui allume deux LEDs à l'aide de deux interrupteurs distincts.

- La LED_1 s'allume en réponse à l'action sur INT_1, tandis que la LED_2 reste éteinte.
- La LED_2 s'allume en réponse à l'action sur INT_2, tandis que la LED_1 reste éteinte.

Exercice4.5 (à résoudre)

Élaborer un programme permettant :

- Le clignotement de la LED en réponse à l'action sur le bouton poussoir_1.
- L'extinction de la même LED en réponse à l'action sur bouton poussoir_2.

Chapitre 5 : Applications des microprocesseurs et microcontrôleurs

Objectifs

A l'issue de ce chapitre, vous serez capable de :

Développer des applications basées sur différents microprocesseurs et microcontrôleurs.

Programmer des périphériques (LCD, claviers, moteurs) dans des systèmes simulés.

Analyser les particularités des codes et des environnements utilisés pour le 8086, Arduino et PIC.

5.1 Introduction

Dans ce chapitre, nous allons explorer des applications mettant en œuvre des microprocesseurs et des microcontrôleurs pour répondre à divers besoins techniques. Ces applications seront réalisées par `simulation`, ce qui permettra de comprendre et d'expérimenter les concepts sans nécessiter de matériel physique.

Nous utiliserons trois types de dispositifs pour illustrer nos projets :

1. `Le microprocesseur 8086` : pour montrer comment une architecture classique est programmée pour piloter des systèmes.
2. `La carte Arduino` : pour sa simplicité d'utilisation et sa grande popularité dans les projets pratiques.
3. `Le microcontrôleur PIC` : pour ses performances spécifiques et son usage répandu dans l'industrie.

Applications abordées dans ce chapitre :

- Interface avec des périphériques comme des écrans LCD et des claviers.
- Génération de signaux pour contrôler des convertisseurs et des moteurs.
- Contrôle des appareils électriques DC/AC.
- Conception d'un système d'acquisition de données.

Chaque application sera développée par `simulation dans des environnements logiciels`, tels que Proteus ou mikrroC for PIC pour les microcontrôleurs, et avec des outils spécifiques pour le 8086. Cette approche vous permettra :

- D'observer directement le comportement des systèmes.
- De comparer les différences dans la programmation et le fonctionnement des dispositifs.
- De mieux comprendre les avantages et les limites de chaque architecture.

Ainsi, ce chapitre constitue une passerelle entre la théorie et la pratique, vous préparant à concevoir des systèmes tout en acquérant une compréhension approfondie des outils et techniques utilisés.

5.2 Applications du microprocesseur 8086

5.2.1 Allumage d'une LED

Objectif : Ce système, basé sur le microprocesseur 8086 et le circuit d'interface 8255, permet de contrôler l'allumage d'une LED. En programmant en assembleur, les étudiants apprennent à configurer le 8255 pour gérer les entrées/sorties, à utiliser les instructions d'accès aux

périphériques et à comprendre le fonctionnement pratique de la communication entre un microprocesseur et ses périphériques.

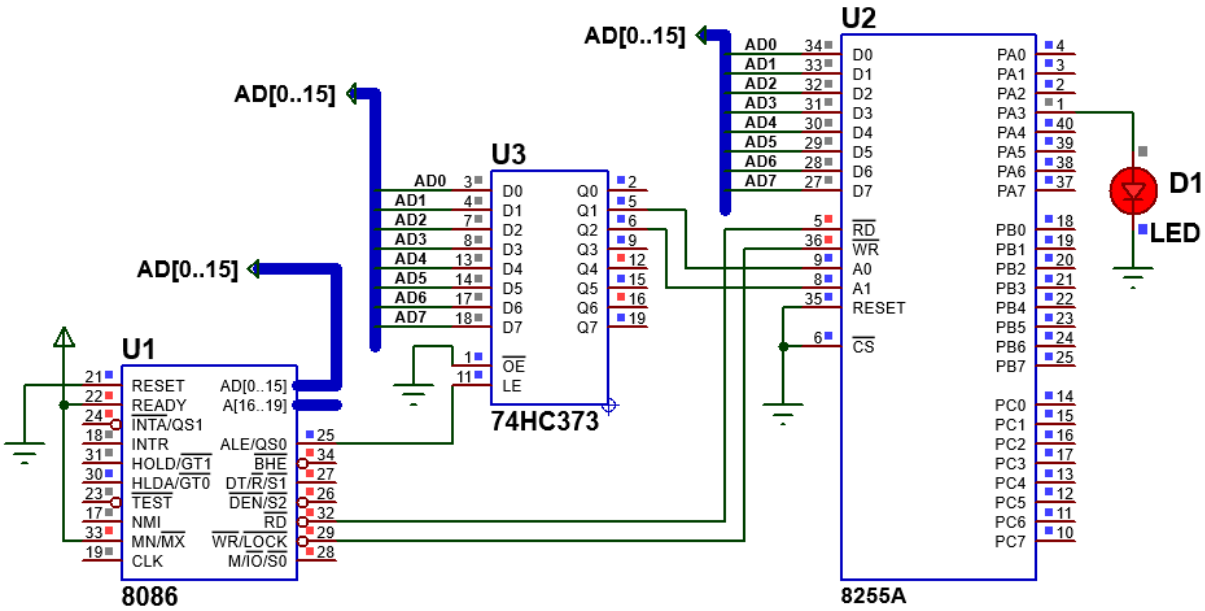


Figure 5.1 *Microprocesseur 8086. Allumage d'une led*

```
DATA SEGMENT          ; Début du segment de données
   PORTA EQU 00H      ; Définition de l'adresse de base pour PORTA
   PORTB EQU 02H      ; Définition de l'adresse de base pour PORTB
   PORTC EQU 04H      ; Définition de l'adresse de base pour PORTC
   PORT_CON EQU 06H   ; Définition de l'adresse de base pour PORT_CON
DATA ENDS             ; Fin du segment de données

CODE SEGMENT          ; Début du segment de code
   MOV AX, DATA       ; Charger l'adresse de base du segment de données dans AX
   MOV DS, AX         ; Initialiser le registre de segment de données DS avec l'adresse de base

ORG 0000H             ; Point d'origine du programme, l'adresse 0000H
START:                ; Marqueur de début du code exécutable
   MOV DX, PORT_CON   ; Charger l'adresse de PORT_CON dans DX
   MOV AL, 10000000B  ; Configuration du port C (sortie) en mettant le bit 7 à 1, mode « 0 »
   OUT DX, AL         ; Envoyer la configuration au port défini par DX

   MOV DX, PORTA      ; Charger l'adresse de PORTA dans DX
   MOV AL, 00001000B  ; Mettre 1 dans le bit 3 pour allumer la LED connectée à PA0
   OUT DX, AL         ; Envoyer la valeur au PORTA
```

```
   MOV AH, 4CH              ; Charger la fonction de terminaison du programme dans AH

   INT 21H                  ; Appeler l'interruption 21H pour terminer le programme
CODE ENDS                   ; Fin du segment de code
END   START                      ;  Fin  du  programme,  avec  START  comme  point  d'entrée
```

5.2.2 Clignotement d'une led

5.2.2.1 Programme1 : sans l'instruction call

Objectif : Ce système, basé sur le microprocesseur 8086 et le circuit 8255, permet de réaliser le clignotement d'une LED. Contrairement au système précédent, le programme est élaboré sans l'instruction "CALL", offrant aux étudiants une compréhension approfondie de l'exécution séquentielle du code en assembleur. Ce projet permet également de consolider leurs compétences en gestion des délais et en programmation des périphériques via le 8255, l'étudiant doit apprendre à le configurer, tout en approfondissant sa compréhension des aspects liés à l'écriture d'un code en assembleur.

```
DATA SEGMENT             ; Début du segment de données
   PORTA EQU 00H         ; Définition de l'adresse de base pour PORTA
   PORTB EQU 02H         ; Définition de l'adresse de base pour PORTB
   PORTC EQU 04H         ; Définition de l'adresse de base pour PORTC
   PORT_CON EQU 06H      ; Définition de l'adresse de base pour PORT_CON
DATA ENDS                ; Fin du segment de données

CODE SEGMENT             ; Début du segment de code
   MOV AX,DATA           ; Charger l'adresse de base du segment de données dans AX
   MOV DS,AX             ; Initialiser le registre de segment de données DS avec l'adresse de
base

ORG 0000H                ; Point d'origine du programme, l'adresse 0000H
START:                   ; Marqueur de début du code exécutable
   MOV DX, PORT_CON      ; Charger l'adresse de PORT_CON dans DX
   MOV AL, 10000000B     ; Configuration du port C (sortie) en mettant le bit 7 à 1
   OUT DX, AL            ; Envoyer la configuration au port défini par DX

   MOV DX, PORTA         ; Charger l'adresse de PORTA dans DX
   MOV CX, 1000          ; Initialiser le compteur pour le nombre de clignotements

LOOP_START:              ; Marqueur de début de la boucle
   MOV AL, 00001000B     ; Mettre 1 dans le bit 3 pour allumer la LED connectée à PA0
   OUT DX, AL            ; Envoyer la valeur à PORTA

   ; Délai d'attente
   MOV BX, 1000          ; Initialiser un compteur pour le délai
DELAY_LOOP:              ; Marqueur de début de la boucle de délai
```

```
    DEC BX              ; Décrémenter le compteur
    JNZ DELAY_LOOP      ; Répéter la boucle tant que BX n'est pas égal à 0

    MOV AL, 00000000B   ; Mettre 0 pour éteindre la LED
    OUT DX, AL          ; Envoyer la valeur à PORTA

    ; Délai d'attente
    MOV BX, 1000        ; Initialiser un compteur pour le délai
DELAY_LOOP_OFF:         ; Marqueur de début de la boucle de délai
    DEC BX              ; Décrémenter le compteur
    JNZ DELAY_LOOP_OFF  ; Répéter la boucle tant que BX n'est pas égal à 0

    LOOP LOOP_START     ; Répéter la boucle jusqu'à ce que CX soit égal à 0

    MOV AH, 4CH         ; Charger la fonction de terminaison du programme dans AH
    INT 21H             ; Appeler l'interruption 21H pour terminer le programme

CODE ENDS               ; Fin du segment de code
END START               ; Fin du programme, avec START comme point d'entrée
```

5.2.2.2 Programme2 : avec l'intruction call

Objectif : Ce système, basé sur le microprocesseur 8086 et le circuit 8255, permet de faire clignoter une LED en utilisant l'instruction "CALL". Cela permet de diviser le programme en petites parties réutilisables, ce qui rend le code plus facile à comprendre et à modifier. Les étudiants apprendront à organiser leur programme de manière plus claire et à utiliser les sous-routines pour simplifier leur travail et gagner du temps.

```
DATA SEGMENT            ; Début du segment de données
    PORTA EQU 00H       ; Définition de l'adresse de base pour PORTA
    PORTB EQU 02H       ; Définition de l'adresse de base pour PORTB
    PORTC EQU 04H       ; Définition de l'adresse de base pour PORTC
    PORT_CON EQU 06H    ; Définition de l'adresse de base pour PORT_CON
DATA ENDS               ; Fin du segment de données

CODE SEGMENT            ; Début du segment de code
    MOV AX, DATA        ; Charger l'adresse de base du segment de données dans AX
    MOV DS, AX          ; Initialiser le registre de segment de données DS avec l'adresse de
                        ;base

ORG 0000H               ; Point d'origine du programme, l'adresse 0000H
START:                  ; Marqueur de début du code exécutable
    MOV DX, PORT_CON    ; Charger l'adresse de PORT_CON dans DX
    MOV AL, 10000000B   ; Configuration du port C (sortie) en mettant le bit 7 à 1
    OUT DX, AL          ; Envoyer la configuration au port défini par DX
```

```
    MOV DX, PORTA       ; Charger l'adresse de PORTA dans DX
    MOV CX, 1000        ; Initialiser le compteur pour le nombre de clignotements

LOOP_START:             ; Marqueur de début de la boucle
    MOV AL, 00001000B   ; Mettre 1 dans le bit 3 pour allumer la LED connectée à PA0
    OUT DX, AL          ; Envoyer la valeur à PORTA
    CALL DELAY          ; Appeler la procédure de délai
    MOV AL, 00000000B   ; Mettre 0 pour éteindre la LED
    OUT DX, AL          ; Envoyer la valeur à PORTA
    CALL DELAY          ; Appeler la procédure de délai
    LOOP LOOP_START     ; Répéter la boucle jusqu'à ce que CX soit égal à 0

    MOV AH, 4CH         ; Charger la fonction de terminaison du programme dans AH
    INT 21H             ; Appeler l'interruption 21H pour terminer le programme

DELAY PROC              ; Procédure de délai
    MOV CX, 1000        ; Initialiser un compteur pour le délai
DELAY_LOOP:             ; Marqueur de début de la boucle de délai
    LOOP DELAY_LOOP     ; Décrémenter le compteur et répéter la boucle tant qu'il n'est
                        ; pas égal à 0
    RET                 ; Retour de la procédure de délai
DELAY ENDP              ; Fin de la procédure de délai

CODE ENDS               ; Fin du segment de code
END START               ; Fin du programme, avec START comme point d'entrée
```

5.2.3 Allumer et éteindre une LED via un bouton poussoir

Objectif : Ce système, basé sur le microprocesseur 8086 et le circuit 8255, permet d'allumer et d'éteindre une LED à l'aide d'un bouton poussoir. L'objectif est d'apprendre à programmer non seulement les sorties pour contrôler la LED, mais aussi les entrées pour détecter l'état du bouton poussoir, par exemple. Les étudiants apprendront à configurer et à gérer les ports d'entrée et de sortie du 8255, ainsi qu'à utiliser les instructions du microprocesseur 8086 pour interagir avec les périphériques, renforçant ainsi leur compréhension des principes de gestion des entrées/sorties.

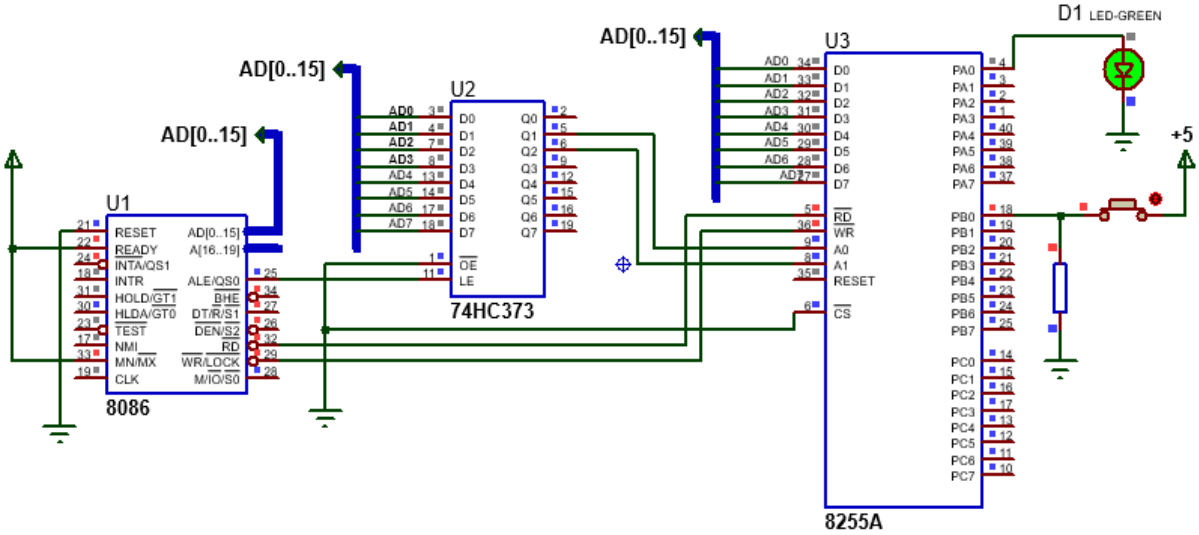


Figure 5.2 *Microprocesseur 8086. Allumage et extinction d'une LED à l'aide d'un bouton poussoir*

```
DATA SEGMENT
 PORTA EQU 00H
 PORTB EQU 02H
 PORTC EQU 04H
 PORT_CON EQU 06H
DATA ENDS
CODE SEGMENT
 MOV AX,DATA
 MOV DS, AX

ORG 0000H
START:
; setup 8255 control word register

 MOV DX, PORT_CON
 MOV AL, 10000000B; port C (output), port A (output) in mode 0 and PORT B (INPUT) in mode 0
 OUT DX, AL

 ; input from tact switch

XX:
 MOV DX, PORTB
 IN AL, DX
 ;not AL
 ; output to LED

 MOV DX, PORTA
 OUT DX, AL
```

```
 JMP XX

CODE ENDS
END   START
```

5.2.4 Contrôle d'un moteur

Objectif : Ce système, basé sur le microprocesseur 8086, le circuit 8255 et le L298, permet de contrôler le sens de rotation d'un moteur. À travers cet exemple, les étudiants apprendront à programmer le contrôle des moteurs en utilisant les ports d'entrée/sortie du 8255 et à interagir avec le L298, un pilote de moteur, via des instructions en assembleur. L'intérêt de cet exercice réside dans l'application de la programmation des périphériques pour contrôler un moteur par exemple, un élément central dans de nombreuses applications en automatisation et robotique. Ce cas diffère des précédents par l'ajout d'un circuit de commande moteur (L298) et l'introduction d'une logique de contrôle plus complexe pour la gestion du sens de rotation du moteur, offrant ainsi aux étudiants une compréhension plus poussée du contrôle de dispositifs en utilisant un microprocesseur.

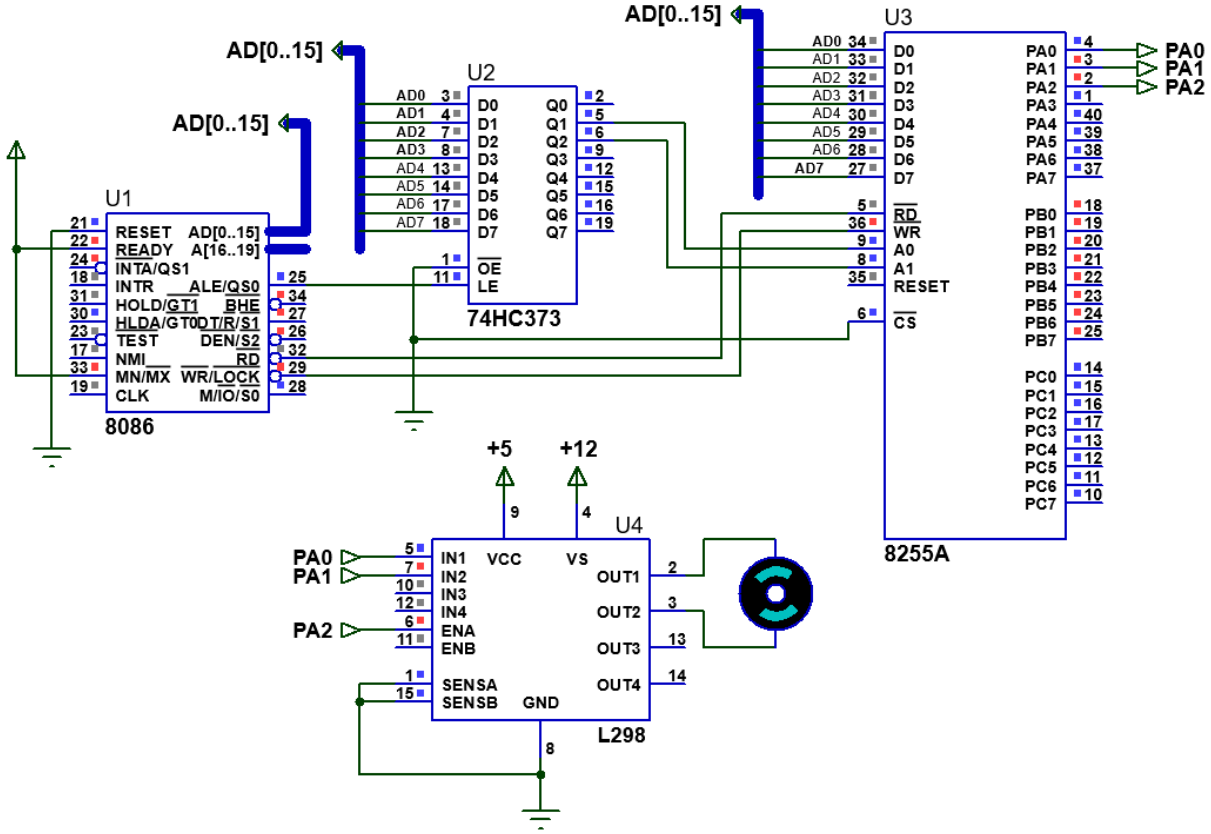


Figure 5.3 *Microprocesseur 8086. Contrôle du sens de rotation d'un moteur*

```
PORTA EQU 00H   ; Définir le PORTA comme une constante égale à l'adresse mémoire 00H
PORTB EQU 02H
PORTC EQU 04H
PORT_CON EQU 06H

CODE SEGMENT    ; Début d'un segment de code
ORG 100H      ; Code assembleur qui suit sera placé à l'adresse mémoire 100H

MOV DX, PORT_CON
MOV AL, 10000010B ; port C (output), port A (output) in mode 0 and PORT B (INPUT) in mode 0
OUT DX, AL

START:      ; Début de la boucle

MOV AL, 00000101B    ; Charge AL avec la valeur binaire 00000101,
MOV DX, PORTA   ; Charge DX avec l'adresse du port
OUT DX, AL      ; Envoie la valeur de AL au port de données du 8255A

MOV BX, 0FF00H ; Charge le registre BX avec la valeur hexadécimale 0FF00H.
CALL DELAY   ; Appelle la fonction DELAY.

MOV AL, 00000000B
MOV DX, PORTA
OUT DX, AL

MOV BX, 0FF00H
CALL DELAY

MOV AL, 00000110B
MOV DX, PORTA
OUT DX, AL

MOV BX, 0FF00H
CALL DELAY

MOV AL, 00000000B
MOV DX, PORTA
OUT DX, AL

MOV BX, 0FF00H
CALL DELAY

JMP START

DELAY:   ; Fonction DELAY qui effectue une attente en décrémentant le registre BX.
   DEC BX   ; Décrémente la valeur du registre BX.
   JNZ DELAY ; Répéter la boucle tant que BX n'est pas égal à 0
RET   ; Retourne à l'instruction suivante à l'emplacement où la fonction a été appelée.

CODE ENDS    ;Fin de la section de code segment
END      ; Fin du programme assembleur
```

5.3 Application de la carte ARDUINO

5.3.1 Microcontrôleur (Carte ARDUINO) pour la lecture des touches du clavier

Objectif : Ce système, utilisant un microcontrôleur d'une carte Arduino, permet de lire les touches d'un clavier et d'afficher les caractères correspondants, par exemple. Les étudiants apprendront à programmer un microcontrôleur pour interfacer un périphérique en mode entrée (le clavier) et à gérer les entrées numériques via des ports spécifiques. L'intérêt de cet exercice est de familiariser les étudiants avec l'utilisation d'un microcontrôleur en utilisant Arduino pour interagir avec des périphériques externes, ce qui leur permet de comprendre la gestion des entrées et sorties dans des applications réelles. Ce cas diffère des précédents en remplaçant le microprocesseur par un microcontrôleur, et en offrant une introduction à la gestion des périphériques avec un microcontrôleur.

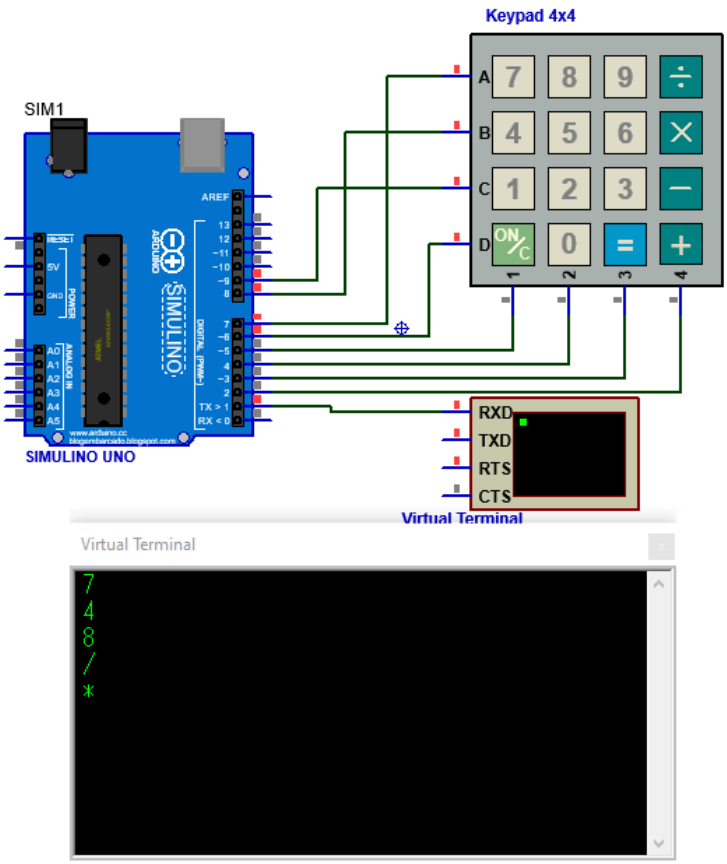


Figure 5.4 *Microcontrôleur (carte ARDUINO). Lecture des touches d'un clavier*

Ce code permettra de lire les touches du clavier matriciel et de les afficher via le port série de l'ARDUINO.

```
#include <Keypad.h>
```

```
const byte ROWS = 4; // Nombre de lignes du clavier
const byte COLS = 4; // Nombre de colonnes du clavier

char keys[ROWS][COLS] = {
  {'1','2','3','/'},
  {'4','5','6','*'},
  {'7','8','9','-'},
  {'#','0','=','+'}
};

byte rowPins[ROWS] = {9, 8, 7, 6}; // Broches pour les lignes du clavier
byte colPins[COLS] = {5, 4, 3, 2}; // Broches pour les colonnes du clavier

Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

void setup(){
  Serial.begin(9600);
}
void loop(){
  char key = keypad.getKey();

  if (key != NO_KEY) {
    Serial.println(key);
  }
    delay(100); // Délai pour éviter la lecture multiple des touches
}
```

5.3.2 Carte ARDUINO pour le Contrôle de la direction d'un moteur à courant continu

Objectif : Ce système, utilisant un microcontrôleur d'une carte Arduino, permet de contrôler la direction d'un moteur à courant continu, pris pour exemple dans ce cas. Les étudiants apprendront à programmer en langage "C" pour contrôler un moteur à l'aide d'un microcontrôleur. L'intérêt de cet exercice est de familiariser les étudiants avec un autre langage de programmation, le "C", largement utilisé dans la programmation des microcontrôleurs. Apprendre à utiliser ce langage est essentiel car il est non seulement utilisé dans de nombreux projets, mais aussi parce qu'il permet de comprendre les bases de la programmation des microcontrôleurs et des périphériques. Ce projet leur offrira une expérience qui les préparera à d'autres applications liées au domaine de commandes des actionneurs.

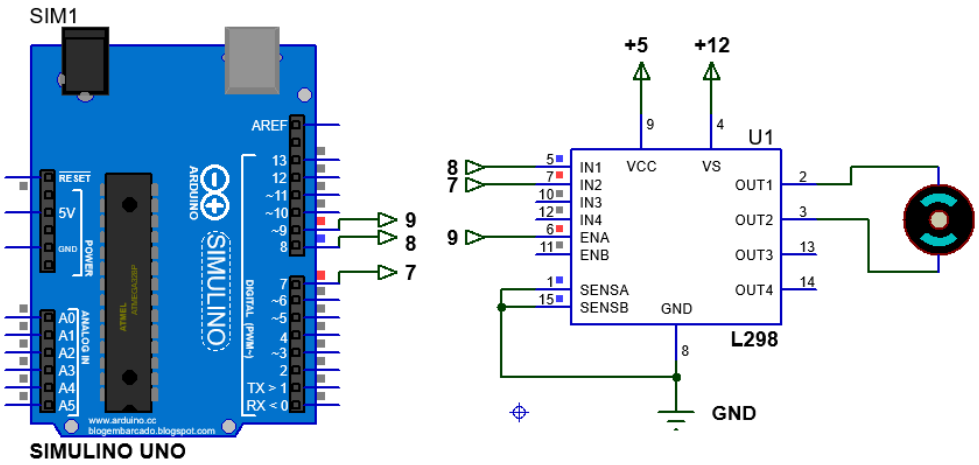


Figure 5.5 *Carte ARDUINO. Contrôle de la direction d'un moteur*

```
// Définir les broches
const int brocheEnable = 9;      // Broche pour activer/désactiver le moteur
const int brocheIn1 = 8;         // Entrée 1 du pont en H
const int brocheIn2 = 7;         // Entrée 2 du pont en H

void setup() {
  // Initialiser les broches
  pinMode(brocheEnable, OUTPUT);
  pinMode(brocheIn1, OUTPUT);
  pinMode(brocheIn2, OUTPUT);
}

void loop() {
  // Faire tourner le moteur en avant à pleine vitesse
  digitalWrite(brocheIn1, HIGH);
  digitalWrite(brocheIn2, LOW);
  analogWrite(brocheEnable, 255);    // 255 est la vitesse maximale

  delay(2000);       // Le moteur tourne en avant pendant 2 secondes

  // Arrêter le moteur
  digitalWrite(brocheIn1, LOW);
  digitalWrite(brocheIn2, LOW);
  analogWrite(brocheEnable, 0);        // Désactiver le moteur

  delay(1000);       // Attendre 1 seconde

  // Faire tourner le moteur en arrière à pleine vitesse
  digitalWrite(brocheIn1, LOW);
  digitalWrite(brocheIn2, HIGH);
  analogWrite(brocheEnable, 255);     // 255 est la vitesse maximale
```

```
  delay(2000);       // Le moteur tourne en arrière pendant 2 secondes

  // Arrêter le moteur
  digitalWrite(brocheIn1, LOW);
  digitalWrite(brocheIn2, LOW);
  analogWrite(brocheEnable, 0);  // Désactiver le moteur

  delay(1000);  // Attendre 1 seconde
}
```

5.3.3 Carte ARDUINO pour l'acquisition de données

Objectif : Ce système basé sur une carte Arduino et un capteur de température permettant de surveiller la température ambiante ou d'un appareil. Le système affiche la température sur un écran LCD ou un terminal, ce qui permet de pouvoir déclencher d'éventuelles actions spécifiques lorsque la température dépasse un seuil prédéfini.

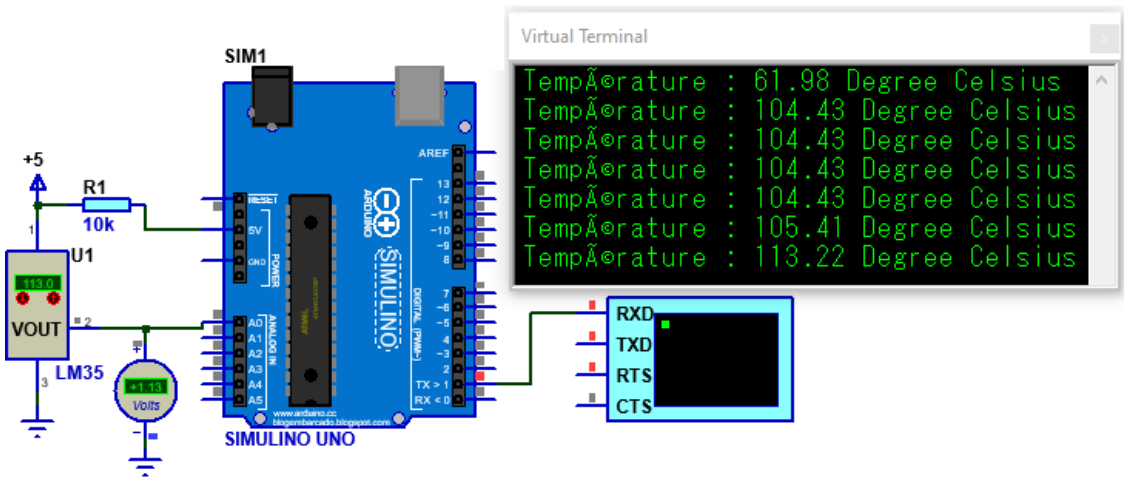


Figure 5.6 *Carte ARDUINO. Acquisition de données*

5.3.4 Carte ARDUINO pour la génération d'un signal carré

Objectif : Ce projet a pour objectif de générer un signal carré sur l'une des broches d'une carte ARDUINO. Le signal alterne entre un état haut et un état bas, avec une période de 1 seconde (500 ms haut et 500 ms bas).

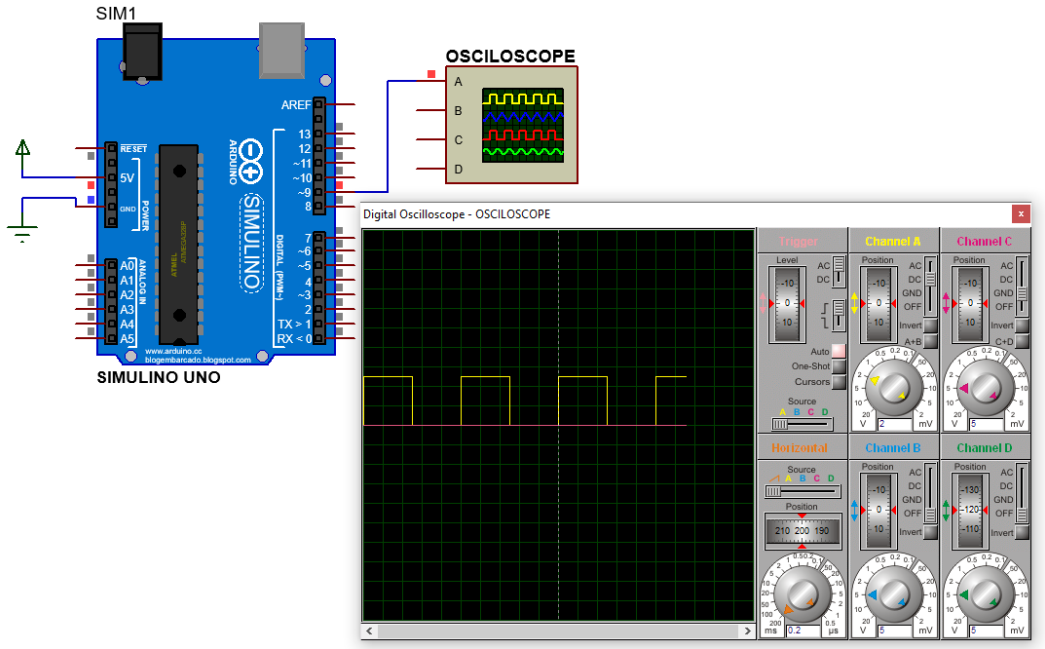


Figure 5.7 *Carte ARDUINO. Génération d'un signal carré*

```
void setup() {
  pinMode(9, OUTPUT);    // Configure la broche 9 comme sortie
}

void loop() {
  digitalWrite(9, HIGH);    // Met la broche 9 à un niveau haut (5V)
  delay(500);                // Attend pendant 500 millisecondes
  digitalWrite(9, LOW);    // Met la broche 9 à un niveau bas (0V)
  delay(500);                // Attend pendant 500 millisecondes
}
```

5.4 Application du microcontrôleur PIC

5.4.1 Microcontrôleur PIC pour la génération d'un signal carré

Objectif : Ce projet a pour objectif de générer un signal carré sur la broche RB0 du microcontrôleur PIC16F877A. Le signal alterne entre un état haut et un état bas, avec une période de 1 seconde (500 ms haut et 500 ms bas).

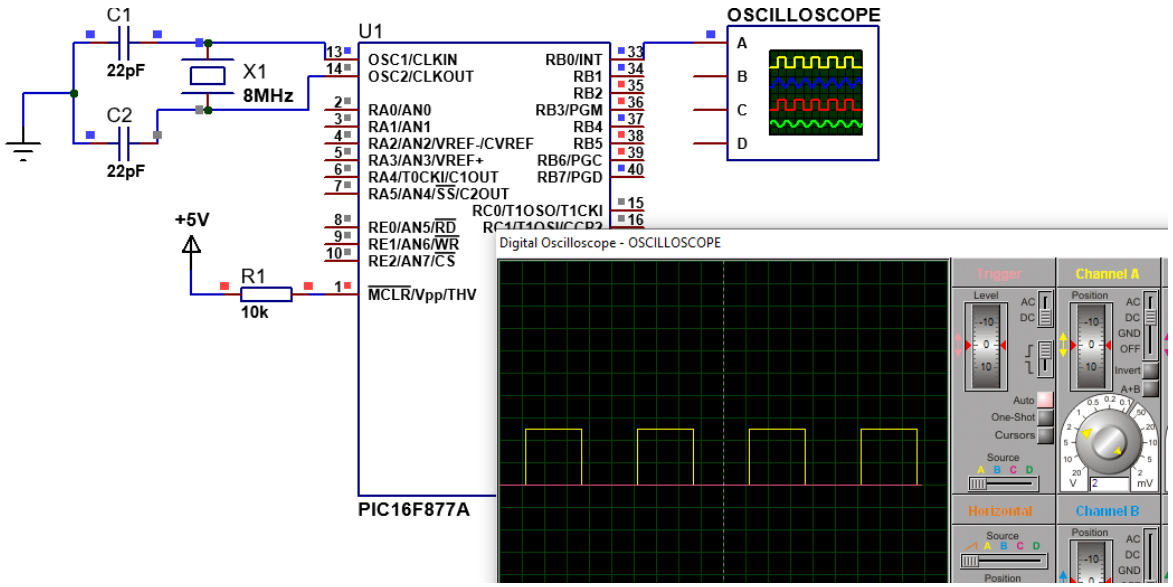


Figure 5.8 *Microcontrôleur PIC. Génération d'un signal carré*

```
void main() {
   TRISB = 0x00;            // Configure le port B comme sortie (0 = sortie, 1 = entrée)
   while(1) {               // Boucle infinie
      PORTB.RB0 = 1;        // Met la broche RB0 à l'état haut (1)
      Delay_ms(500);        // Attendre 500ms
      PORTB.RB0 = 0;        // Met la broche RB0 à l'état bas (0)
      Delay_ms(500);        // Attendre 500ms
   }
}
```

5.4.2 Microcontrôleur PIC pour la commande d'un moteur 220Vvia un bouton poussoir

Objectif : Ce projet a pour objectif de commander le fonctionnement d'un moteur de 220V via un circuit adaptateur et un bouton poussoir connectés aux broches d'un microcontrôleur PIC. Lorsque le bouton est pressé, le moteur commence à tourner ; sinon, elle reste en arrêt.

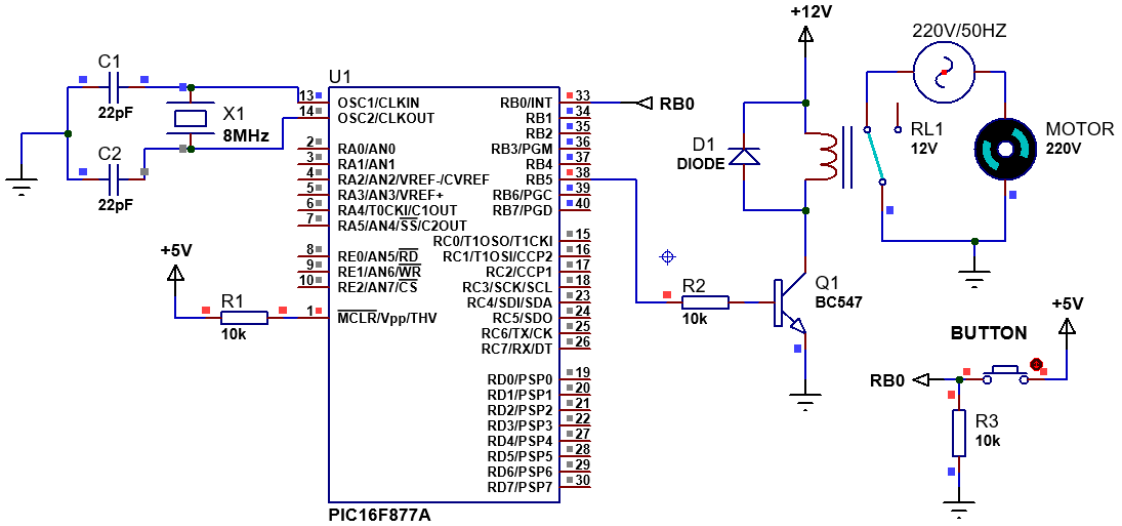


Figure 5.9 *Microcontrôleur PIC. Commande de fonctionnement d'un moteur 220V*

```
void main() {
   TRISB = 0x01;       // Configure RB0 comme entrée, le reste (RB1 à RB7) comme sorties
   PORTB = 0x00;          // Initialise toutes les broches de PORTB à 0

   while (1) {                    // Boucle infinie
      if (PORTB & 0x01) {      // Si RB0 (bit 0) est à 1 (bouton pressé)
         PORTB = 0x21;         // Allume RB5 (bit 5) et maintient RB0 actif
      } else {                  // Sinon
         PORTB = 0x00;           // Éteint toutes les broches
      }
   }
}
```

5.4.3 Microcontrôleur PIC pour le contrôle de la vitesse du moteur à courant continu avec PWM

Objectif : Le projet consiste à contrôler la vitesse d'un moteur à courant continu en ajustant le rapport cyclique du signal PWM à l'aide de deux boutons. Un bouton permet d'augmenter la vitesse du moteur, tandis que l'autre permet de la réduire. Le moteur tourne dans une direction déterminée par les broches de commande. Le rapport cyclique est ajusté de manière incrémentale ou décrémentale, avec une limitation pour éviter les valeurs excessives.

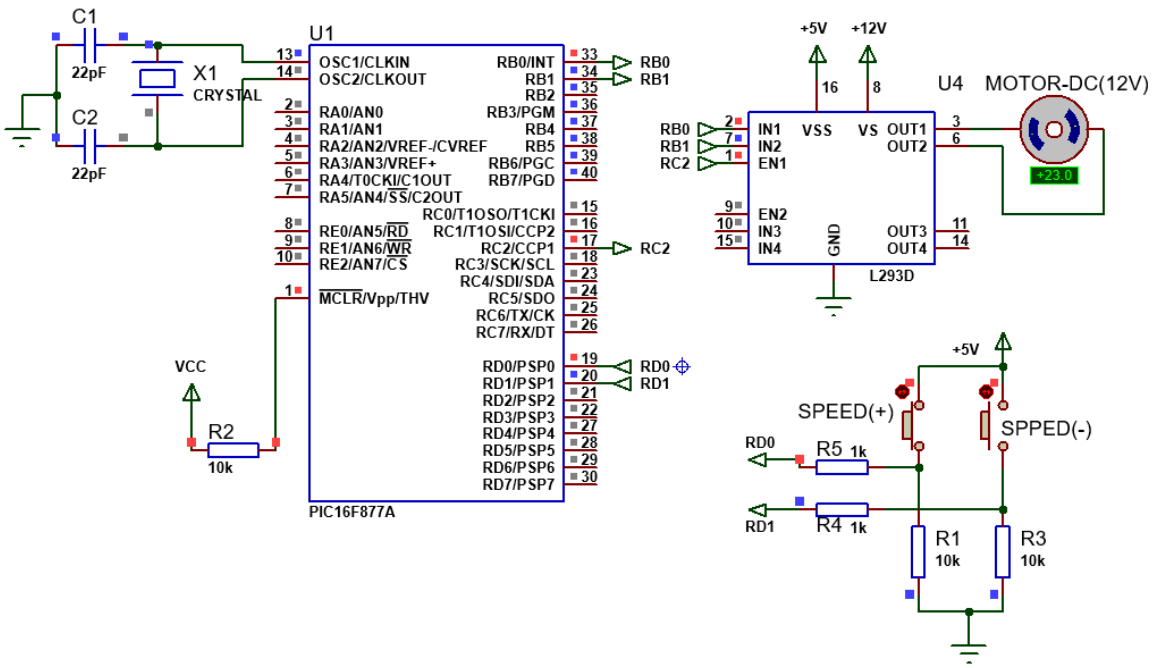


Figure 5.10 *Microcontrôleur PIC. Contrôle de la vitesse d'un moteur par PWM*

```
void main() {
   short duty = 0;          // duty (rapport cyclique en français)
                        //Valeur initiale du rapport cyclique pour le moteur
   TRISB = 0x00;           // Configurer PORTB comme sortie (IN1 et IN2 du moteur)
   TRISD = 0xFF;           // Configurer PORTD comme entrée (pour les boutons)
   PORTB = 0x00;           // Initialiser PORTB à 0 (moteur éteint)

   PWM1_Init(2000);        // Initialiser le PWM à 2000 Hz
   PWM1_Start();           // Démarrer le PWM
   PWM1_Set_Duty(duty);    // Appliquer le rapport cyclique initial à 0%

   PORTB.F0 = 1;           // Active IN1 (pour faire tourner
                           //le moteur dans une direction)
   PORTB.F1 = 0;           // Désactive IN2

   while (1) {             // Boucle infinie
      // Si le bouton (PORTD.F0) est pressé et que le rapport cyclique < 250
      if (PORTD.F0 == 1 && duty < 250) {
         Delay_ms(200);    // Anti-rebond
         if (PORTD.F0 == 1) {    // Vérification de l'état du bouton après anti-rebond
            duty += 10;          // Augmenter le rapport cyclique
            if (duty > 250) duty = 250;      // Limiter la valeur à 250
```

```
            PWM1_Set_Duty(duty);        // Appliquer le nouveau rapport cyclique
        }
    }

      // Si le bouton (PORTD.F1) est pressé et que le rapport cyclique > 0
    if (PORTD.F1 == 1 && duty > 0) {
        Delay_ms(200);    // Anti-rebond
        if (PORTD.F1 == 1) {      // Vérification de l'état du bouton après anti-rebond
            duty -= 10;      // Réduire le rapport cyclique
            if (duty < 0) duty = 0;      // Limiter la valeur à 0
            PWM1_Set_Duty(duty);      // Appliquer le nouveau rapport cyclique
        }
    }
  }
}
```

5.4.4 Microcontrôleur pour le Contrôle d'un moteur 220V à base d'un capteur

Objectif : Le programme lit la température d'un moteur (à courant continu de 220V) à partir d'un capteur analogique LM35 et affiche la température sur un écran LCD. Si la température dépasse 40°C, il éteint le moteur, sinon il le met en marche.

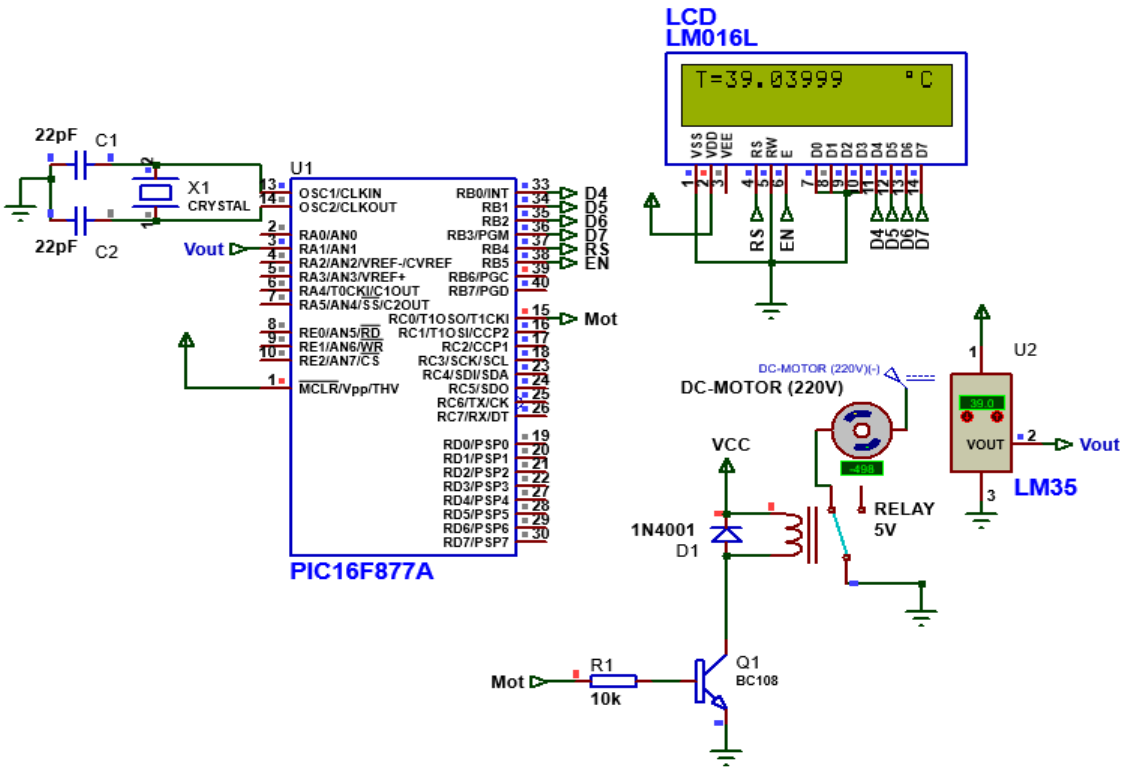


Figure 5.11 *Microcontrôleur PIC. Contrôle d'un moteur par un capteur*

```
// Déclaration des broches utilisées pour l'affichage LCD
sbit LCD_RS at RB4_bit;    // Broche RS (Register Select) pour le LCD
sbit LCD_EN at RB5_bit;    // Broche EN (Enable) pour le LCD
sbit LCD_D4 at RB0_bit;    // Broche D4 (Data) pour le LCD
sbit LCD_D5 at RB1_bit;    // Broche D5 (Data) pour le LCD
sbit lcd_D6 at RB2_bit;    // Broche D6 (Data) pour le LCD
sbit lcd_D7 at RB3_bit;    // Broche D7 (Data) pour le LCD

// Déclaration des directions des broches utilisées pour l'affichage LCD
sbit lcd_RS_Direction  at TRISB4_bit;   // Direction de la broche RS
sbit lcd_EN_Direction  at TRISB5_bit;   // Direction de la broche EN
sbit lcd_D4_Direction  at TRISB0_bit;   // Direction de la broche D4
sbit lcd_D5_Direction  at TRISB1_bit;   // Direction de la broche D5
sbit lcd_D6_Direction  at TRISB2_bit;   // Direction de la broche D6
sbit lcd_D7_Direction  at TRISB3_bit;   // Direction de la broche D7

char affichage[16]="";   // Déclaration d'une chaîne de caractères pour afficher la température

void main() {

 unsigned int result;          // Variable pour stocker le résultat de la lecture ADC
 float volt, temp;           // Variables pour stocker la tension et la température calculées

 // Configuration des directions des ports
 trisb = 0;               // PORTB configuré en sortie (pour contrôler le LCD)
 trisc = 0;             // PORTC configuré en sortie (pour la LED)
 trisa = 0xff;        // PORTA configuré en entrée (pour lire la température via ADC)

 // Initialisation du LCD
 lcd_init();         // Initialisation du LCD
 lcd_cmd(_lcd_clear);             // Effacer l'affichage du LCD
 lcd_cmd(_lcd_cursor_off);             // Désactiver le curseur du LCD

 // Initialisation du module ADC
 adc_init();         // Initialisation de l'ADC (convertisseur analogique-numérique)
 adcon1 = 0x80;       // Configurer les paramètres de l'ADC

 // Boucle infinie pour afficher la température en temps réel
 while(1)
 {
    result = adc_read(1);       // Lire la valeur ADC sur le canal 1 (température)
    volt = result * 4.88;       // Convertir la valeur ADC en tension (4.88 mV par unité)
    temp = volt / 10;          // Calculer la température en degrés Celsius (1 unité = 0.1°C)

    // Affichage de la température sur le LCD
    lcd_out(1,1,"T=");         // Afficher "T=" à la position (1,1) du LCD
```

```
    floattostr(temp, affichage);     // Convertir la valeur flottante de la température en chaîne
de caractères
    lcd_out_cp(affichage);     // Afficher la température convertie à la position actuelle du
                               //curseur
    lcd_chr(1,15,223);          // Afficher le caractère de degré (°) à la position (1,15)
    lcd_out(1,16,"C");         // Afficher "C" pour indiquer que la température est en °C

    // Si la température est supérieure à 40°C
    if (temp > 40)
    {
       portc = 0;             // Éteindre la LED en mettant le PORTC à 0
    }
    else
    {
       portc = 1;              // Allumer la LED en mettant le PORTC à 1
    }
 }
}
```

Conclusion générale

En conclusion, ce cours a permis d'explorer les principes fondamentaux des microprocesseurs et des microcontrôleurs, en mettant l'accent sur leur architecture, leur fonctionnement et leurs applications pratiques. Chaque chapitre a été soigneusement développé avec des exemples concrets et des exercices résolus, permettant de relier la théorie à des situations réelles.

Grâce à cette approche, vous avez pu acquérir une compréhension approfondie du microcontrôleur PIC et découvrir d'autres options, comme celui de la carte Arduino, pour élargir votre vision des technologies actuelles. Ces connaissances et compétences vous offrent une base solide pour aborder des projets plus complexes et appréhender efficacement les systèmes embarqués.

Ce que vous avez appris trouvera également des applications directes dans votre spécialité, notamment en commandes électriques et réseaux électriques. L'électrotechnique ne se limite pas aux machines : elle inclut aussi des dimensions technologiques et numériques, indispensables pour répondre aux besoins des systèmes modernes. Vous êtes ainsi mieux outillés pour innover et relever les défis techniques de votre domaine.

Références bibliographiques

1. Livre :

- **Bennasar, H.** (1993). *Cours de microprocesseurs 16 bits : 8086/68000*. Office des publications universitaires, Ben-Aknoun, Alger.
- **Mazidi, M. A., McKinlay, R. D., & Causey, D.** (2008). *PIC Microcontroller and Embedded Systems: Using Assembly and C for PIC18*. Pearson Education.
- **Ayala, K. J.** (1995). *The 8086 Microprocessor: Programming and Interfacing the PC*. Delmar Publishers, New York.
- **Hugues, J.-M.** (2018). *Arduino, le guide complet*. Éditions Dunod.

2. Sites web :

- Microchip Technology Inc. (2024). *PIC16F877A Data Sheet*. https://www.microchip.com
- Arduino. (2024). *Arduino Programming Basics*. https://www.arduino.cc