Ministère de l'enseignement supérieur et de la recherche scientifique
Université MENTOURI Constantine
Département d'électrotechnique

Module : Informatique Industrielle 1 **Code : ME17**

Niveau : Master en électrotechnique **Méthode d'enseignement : Cours/ TD / TP**

Objectifs de la formation :

Contenu du module proposé par l'administration :

- Architecture, Microprocesseur à usage générale, processeurs de traitement numérique du signal DSP.
- Programmation, exemple de proc.

Les chapitres proposés par le responsable de module :

Chapitre 1 : Historique et évolution des ordinateurs et des microprocesseurs
Chapitre 2 : Le microprocesseur Intel 8086
Chapitre 3 : La programmation en assembleur du 8086
Chapitre 4 : Les interfaces d'entrées/sorties
Chapitre 5 : Les interruptions dans 8086
Chapitre 6 : Le Microcontrôleur PIC16F84

Chapitre 6 : Les processeurs de traitement numérique du signal DSP.

Annexe : Jeu d'instructions du 8086

Responsable de module : Mr MESSAOUDI Kamel Maître assistant A UMC

Ce document constitue le support du cours et ne prétend donc ni à l'originalité, ni à l'exhaustivité. Ces notes doivent beaucoup aux emprunts faits à de nombreux ouvrages et à différents travaux de collègues. Ce document s'adresse aux étudiants ***Master1 (Master en électrotechnique)****, département d'****électrotechnique****, université* ***Mentouri de Constantine****.*

Année Universitaire 2011-2012

Chapitre I : Historique et évolution des ordinateurs et des microprocesseurs

I. Préface :

Apparus dès la création des premiers circuits intégrés numériques, au début des années 1970, les microprocesseurs constituent le coeur de presque toutes les réalisations électroniques; on en trouve dans tous les domaines, notamment : l'informatique (de la calculatrice à l'ordinateur), l'automobile (ABS, injection, ...), l'automatique (automates programmables, contrôle de processus, ...), l'électronique domestique (thermomètre, télécommande, carte à puce, ...).

Les performances des microprocesseurs sont liées aux possibilités offertes par la technologie, en terme de capacité (nombre de portes logiques intégrées) et de vitesse, et au choix d'architectures adaptées (ou imposées pour cause de compatibilité ascendante); à l'heure actuelle, on trouve sur le marché des microprocesseurs intégrant des millions de transistors, fonctionnant à plus de 3000 MHz et disposés dans des boîtiers de plusieurs centaines de broches, ils sont issus de différentes approches architecturales : CISC, RISC, DSP et VLIW.

L'objectif de ce chapitre est de retracer l'évolution des microprocesseurs, en liaison avec celle de la technologie.

II. « Préhistoire » des ordinateurs :

Les premières machines à calculer étaient purement **mécaniques** : bouliers, abaques, ... (antiquité). La première vraie **machine à calculer** est apparue en 1942 : (Pascal, machine à additionner). Une autre machine à multiplier proposée par Leibniz en 1694, basée sur les travaux de John Neper (1617, logarithmes). Vers les 18eme siècle, une autre **machine programmable** à base des cartes perforées est apparue (métier à tisser, Jacquard).

II.1. Machines électromécaniques :

- Machine à calculer à cartes perforées : Hermann Hollerith, 1885, facilite le recensement américain.
- Machines industrielles pour la comptabilité et les statistiques. Ces machines sont à base de relais électromécaniques (Aiken et Stibitz, 1936-1939).

II.2. Machines électroniques :

- Première machine à calculer électronique : ENIAC, 1944, Eckert et Mauchly, 18000 tubes électroniques, machine à programme câblé.
- Machine à programme enregistré : John Von Neumann, 1946, les instructions sont enregistrées dans la mémoire du calculateur : ordinateur.
- Premier ordinateur commercialisé : SSEC d'IBM, 1948.
- Ordinateur à transistors : 1963, PDP5 de Digital Equipment Corporation (DEC), introduction des mémoires à ferrites : mini-ordinateurs.
- Micro-ordinateurs : 1969-70, utilisation des circuits intégrés LSI.
- Premier microprocesseur : Intel, 1971, microprocesseur 4004, puis 8008, premier microordinateur : le Micral, 1973, France, puis l'Altair, 1975, Etats-Unis.
- Autres microprocesseurs : 8080 et 8085 d'Intel, 6800 de Motorola, Z80 de Zilog : microprocesseurs 8 bits, début des années 1980.
- Microprocesseurs 16 bits : 8086/8088 d'Intel, 68000 de Motorola.
- Microprocesseurs 32 bits en 1986 : 80386 d'Intel et 68020 de Motorola.
- Fabrication en grandes séries des micro-ordinateurs : 1977, Apple, Commodore, Tandy.
- IBM PC + MS-DOS (Microsoft) en 1981.

II.3. Machines actuelles :

Les ordinateurs de nos jours sont de plus en plus puissants, basés sur des microprocesseurs performants : Pentium, Power PC, ... ; Avec des nouvelles architectures de microprocesseurs : RISC. Et avec des applications multimédia, réseaux, ...

De nos jours, est apparue aussi la notion des systèmes embarqués : microcontrôleurs, processeurs de traitement de signal (DSP), les microprocesseurs configurables ...

III. Architecture et fonctionnement d'un microprocesseur :

III.1 Structure d'un calculateur :

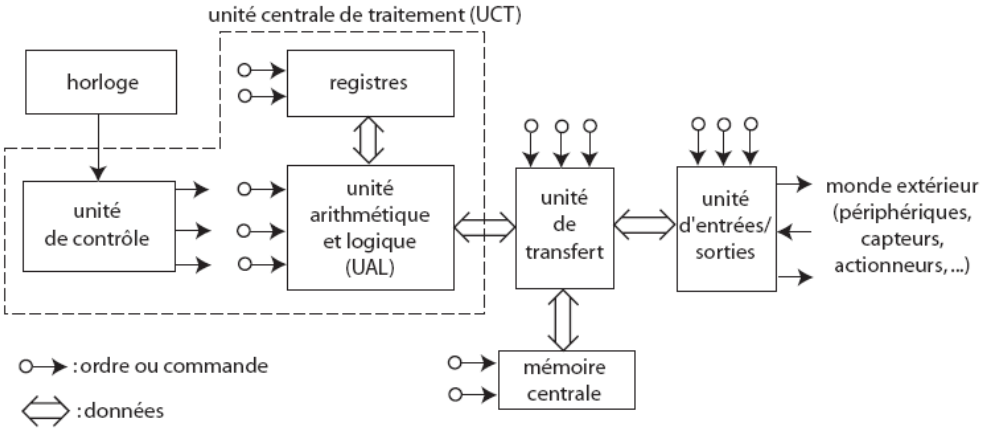


Figure n°1 : Structure d'un calculateur

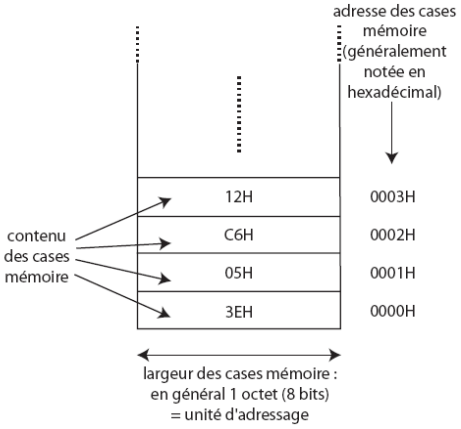
- L'élément de base d'un calculateur est constitué par **l'unité centrale de traitement** (UCT, CPU: Central Processing Unit). L'UCT est constituée :
  - D'une **unité arithmétique et logique** (UAL, ALU : Arithmetic and Logic Unit) : c'est l'organe de calcul du calculateur ;
  - De **registres** : zones de stockage des données de travail de l'UAL (opérandes, résultats intermédiaires) ;
  - D'une **unité de contrôle** (UC, CU : Control Unit) : elle envoie les ordres (ou commandes) à tous les autres éléments du calculateur afin d'exécuter un **programme**.
- **La mémoire centrale** contient :
  - Le programme à exécuter : suite d'instructions élémentaires ;
  - Les données à traiter.
- **L'unité d'entrées/sorties** (E/S) est un intermédiaire entre le calculateur et le monde extérieur.
- **L'unité de transfert** est le support matériel de la circulation des données.
- Les échanges d'ordres et de données dans le calculateur sont synchronisés par une **horloge** qui délivre des impulsions (signal d'horloge) à des intervalles de temps fixes.

Définition : Un **microprocesseur** consiste en une unité centrale de traitement (UAL + registres + unité de contrôle) entièrement contenue dans **un seul circuit intégré**. Un calculateur construit autour d'un microprocesseur est un **microcalculateur** ou un **microordinateur**. Un circuit intégré qui inclut une UCT, de la mémoire et des périphériques est un **microcontrôleur**.

III.2 Organisation de la mémoire centrale :

La mémoire peut être vue comme un ensemble de **cellules** ou **cases** contenant chacune une information : une instruction ou une donnée. Chaque case mémoire est repérée par un numéro d'ordre unique : son **adresse**. Une case mémoire peut être lue ou écrite par le microprocesseur (cas des **mémoires vives**) ou bien seulement lue (cas des **mémoires mortes**). La figure suivante montre le schéma synoptique d'une mémoire :

Figure n°2 : Organisation de la mémoire centrale



III.3 Circulation de l'information dans un calculateur

La réalisation matérielle des ordinateurs est généralement basée sur l'architecture de ***VonNeumann*** :

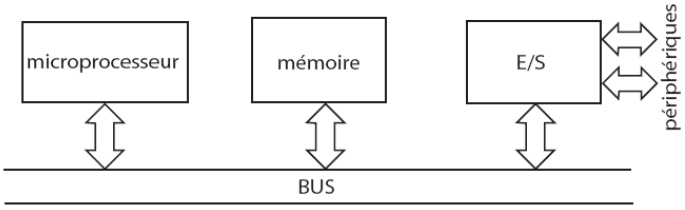


Figure n°3 : Architecture de base de VonNeumann de transfert des données

Le microprocesseur échange des informations avec la mémoire et l'unité d'E/S, sous forme de mots binaires, au moyen d'un ensemble de connexions appelé **bus**. Un bus permet de transférer des données sous forme **parallèle**, c'est-à-dire en faisant circuler *n* bits simultanément.

Les microprocesseurs peuvent être classés selon la longueur maximale des mots binaires qu'ils peuvent échanger avec la mémoire et les E/S : microprocesseurs 8 bits, 16 bits, 32 bits, ... Le bus peut être décomposé en trois bus distincts :

- Le **bus d'adresses** permet au microprocesseur de spécifier l'adresse de la case mémoire à lire ou à écrire ;
- Le **bus de données** permet les transferts entre le microprocesseur et la mémoire ou les E/S ;
- Le **bus de commande** transmet les ordres de lecture et d'écriture de la mémoire et des E/S.

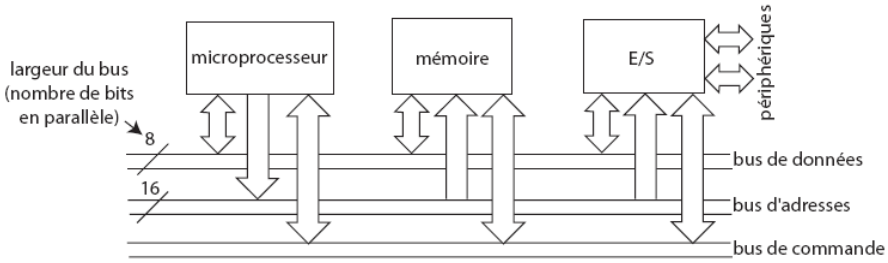


Figure n°4 : Architecture détallé de VonNeumann de transfert des données.

Remarque : Les bus de données et de commande sont **bidirectionnels**, le bus d'adresse est **unidirectionnel** : seul le microprocesseur peut délivrer des adresses (il existe une dérogation pour les circuits d'accès direct à la mémoire, DMA).

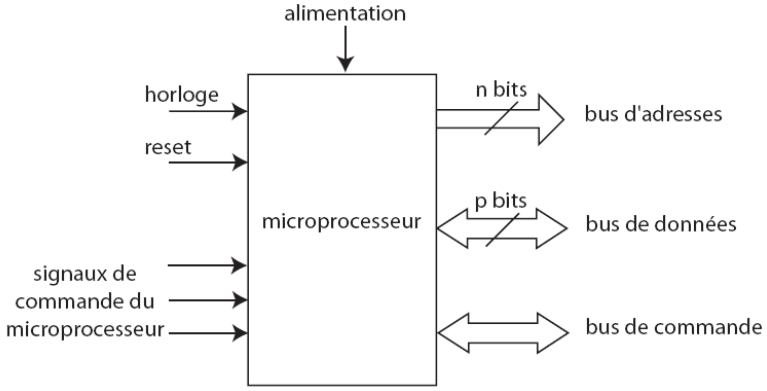
III.4. Description matérielle d'un microprocesseur :

Un microprocesseur se présente sous la forme d'un circuit intégré muni d'un nombre généralement important de broches. Exemples :

- Intel 8085, 8086, Zilog Z80 : 40 broches, DIP (Dual In-line Package) ;
- Motorola 68000 : 64 broches, DIP ;
- Intel 80386 : 196 broches, PGA (Pin Grid Array).

La technologie de fabrication : NMOS, PMOS, CMOS. On peut représenter donc un microprocesseur par son **schéma fonctionnel** :

Figure n°5 : Schéma fonctionnel d'un microprocesseur



III.5 Fonctionnement d'un microprocesseur :

Un microprocesseur exécute un **programme**. Le programme est une suite d'instructions stockées dans la mémoire. Une instruction peut être codée sur **un ou plusieurs octets**. Le format d'une instruction est données par :

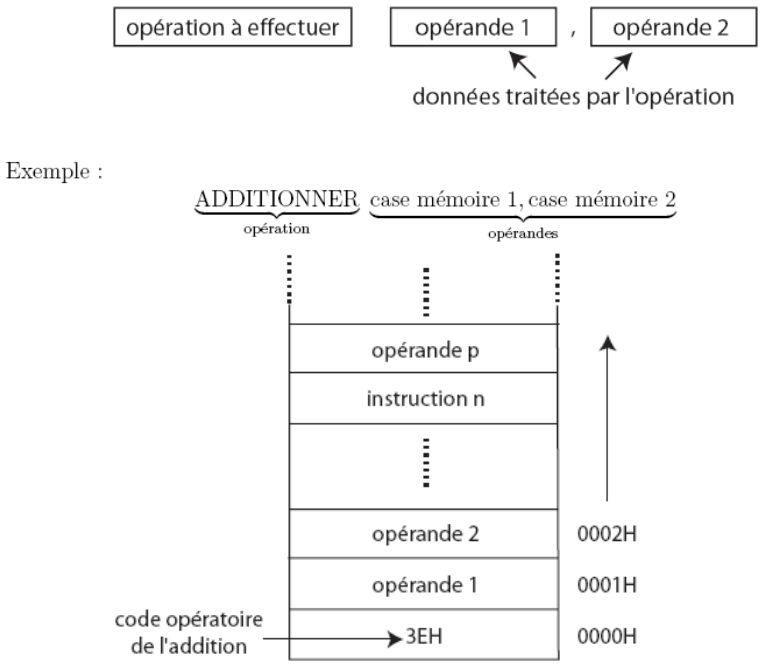


Figure n°6 : Format et arrangement d'une instruction exécutée par un microprocesseur.

Pour exécuter les instructions dans l'ordre établi par le programme, le microprocesseur doit savoir à chaque instant l'adresse de la prochaine instruction à exécuter. Le microprocesseur utilise un registre contenant cette information. Ce registre est appelé **pointeur d'instruction** (IP : Instruction Pointer) ou **compteur d'instructions** ou **compteur ordinal**. La figure suivante illustre un exemple d'IP :

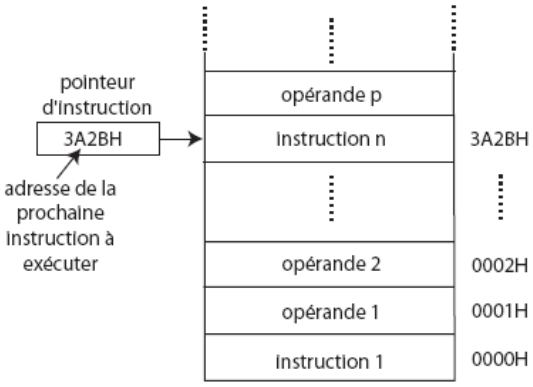


Figure n°7 : Exemple d'un pointeur d'instruction IP.

Remarque n°1 :

La valeur initiale du pointeur d'instruction est fixée par le constructeur du microprocesseur. Elle vaut une valeur bien définie à chaque mise sous tension du microprocesseur ou bien lors d'une remise à zéro (reset).

Pour savoir quel type d'opération doit être exécuté (addition, soustraction, ...), le microprocesseur lit le premier octet de l'instruction pointée par le pointeur d'instruction (code opératoire) et le range dans un registre appelé **registre d'instruction**. Le code opératoire est **décodé** par des circuits de décodage contenus dans le microprocesseur. Des signaux de commande pour l'UAL sont produits en fonction de l'opération demandée qui est alors exécutée.

Remarque n°2 : Pour exécuter une instruction, l'UAL utilise des **registres de travail**, exemple : l'**accumulateur**, registre temporaire recevant des données intermédiaires. Pendant que l'instruction est décodée, le pointeur d'instruction est incrémenté de façon à pointer vers l'instruction suivante :

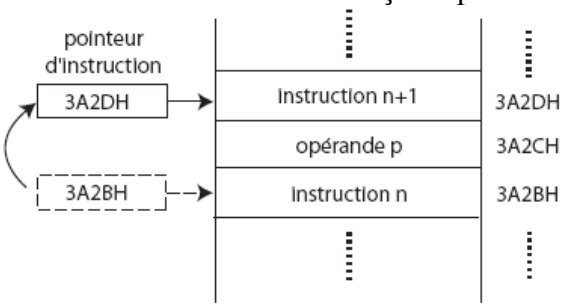


Figure n°8 : Incrémentation du pointeur d'instruction.

Ensuite le processus de lecture et de décodage des instructions recommence. A la suite de chaque instruction, un registre du microprocesseur est actualisé en fonction du dernier résultat : c'est le **registre d'état** du microprocesseur. Chacun des bits du registre d'état est un **indicateur d'état** ou **flag** (drapeau). Exemple : Registre d'état du microprocesseur Z80 :

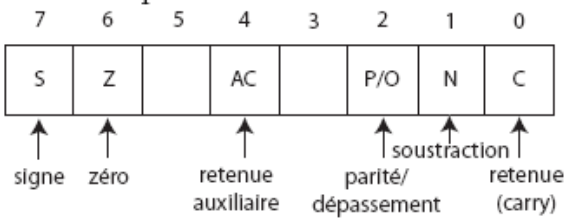


Figure n°9 : Registre d'état du microprocesseur Z80.

Les indicateurs d'état sont activés lorsqu'une certaine condition est remplie, exemple : le flag Z est mis à 1 lorsque la dernière opération a donné un résultat nul, le flag C est mis à un lorsque le résultat d'une addition possède une retenue, ...

Les indicateurs d'état sont utilisés par les instructions de **saut conditionnels** : en fonction de l'état d'un (ou plusieurs) flags, le programme se poursuit de manière différente.

Toutes ces étapes (lecture de l'instruction, décodage, exécution) sont synchronisées par un séquenceur qui assure le bon déroulement des opérations :

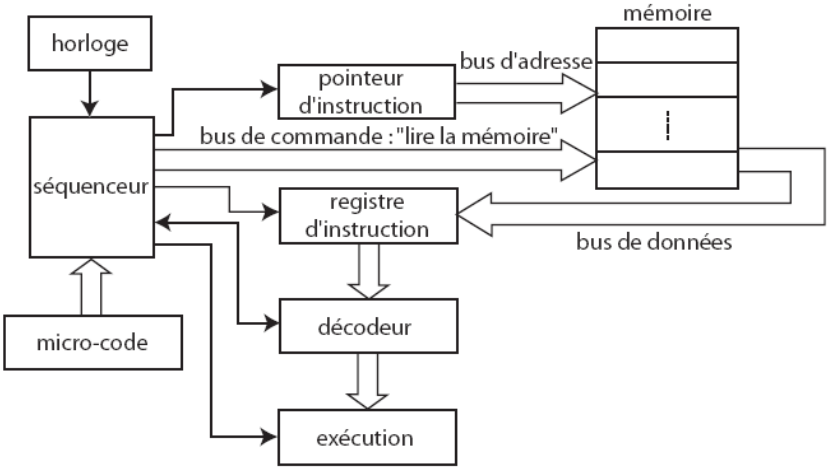


Figure n°10 : Synchronisation des étapes d'exécution d'une instruction.

Pour exécuter le programme contenu dans la mémoire centrale, le séquenceur du microprocesseur exécute lui-même un programme appelé **micro-code**, contenu dans une mémoire morte à l'intérieur du microprocesseur.

Le séquenceur est dirigé par une horloge qui délivre un signal de fréquence donnée permettant d'enchaîner les différentes étapes de l'exécution d'une instruction :

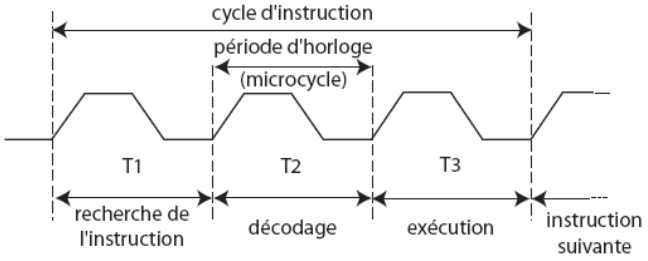


Figure n°11 : Le signal Horloge et le cycle d'instruction.

Chaque instruction est caractérisée par le nombre de périodes d'horloge (ou microcycles) qu'elle utilise (donnée fournie par le fabricant du microprocesseur). Exemple : horloge à 5 MHz, période $T = 1/f = 0,2\ \mu s$. Si l'instruction s'exécute en 3 microcycles, la durée d'exécution de l'instruction est : $3 \times 0,2 = 0,6\ \mu s$. L'horloge est constituée par un oscillateur à quartz dont les circuits peuvent être internes ou externes au microprocesseur.

La figure suivante montre la structure complète d'un microprocesseur simple : pour fonctionner, un microprocesseur nécessite donc au minimum les éléments suivants :

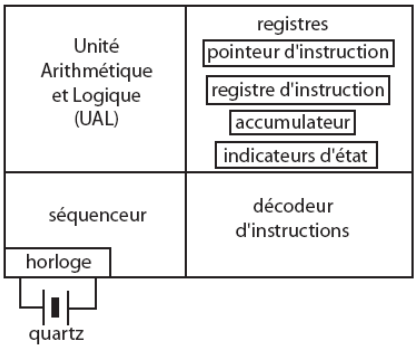


Figure n°12 : Structure complète d'un microprocesseur.

IV. Les mémoires et les microprocesseurs :

IV.1. Mémoires ROM et RAM :

On distingue deux types de mémoires :

- Les **mémoires vives** (RAM : Random Access Memory) ou mémoires volatiles. Elles perdent leur contenu en cas de coupure d'alimentation. Elles sont utilisées pour stocker temporairement des données et des programmes. Elles peuvent être lues et écrites par le microprocesseur ;
- Les **mémoires mortes** (ROM : Read Only Memory) ou mémoires non volatiles. Elles conservent leur contenu en cas de coupure d'alimentation. Elles ne peuvent être que lues par le microprocesseur (pas de possibilité d'écriture). On les utilise pour stocker des données et des programmes de manière définitive.

Les mémoires sont caractérisées par leur **capacité** : nombre total de cases mémoire contenues dans un même boîtier.

IV.2. Schéma fonctionnel d'une mémoire :

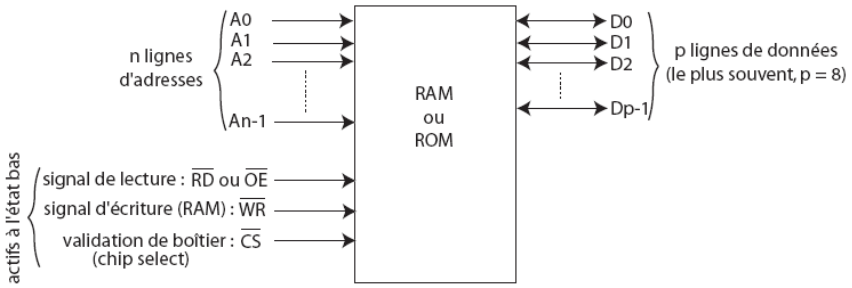


Figure n°13 : Schéma fonctionnel d'une mémoire.

Le nombre de lignes d'adresses dépend de la capacité de la mémoire : *n* lignes d'adresses permettent d'adresser 2*n* cases mémoire : 8 bits d'adresses permettent d'adresser 256 octets, 16 bits d'adresses permettent d'adresser 65536 octets (= 64 Ko), ...etc. Exemple : mémoire RAM 6264, capacité = 8K×8 bits : 13 broches d'adresses A0 `a A12, 213 = 8192 = 8 Ko.

IV.3. Interfaçage microprocesseur/mémoire :

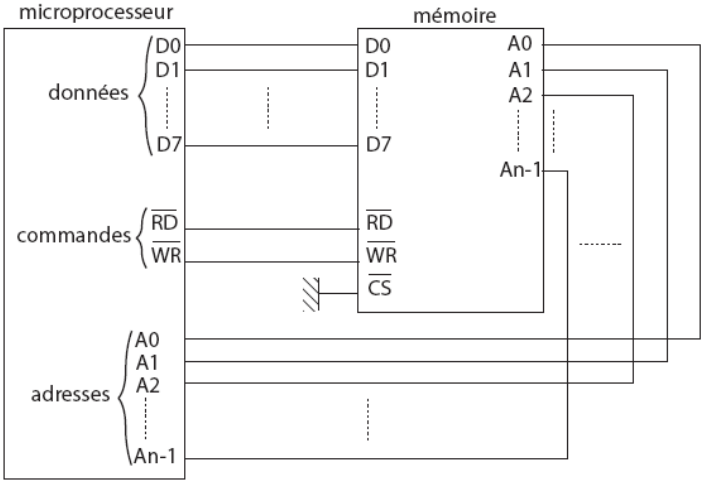


Figure n°14 : Interfaçage microprocesseur/mémoire.

Représentation condensée (plus pratique) :

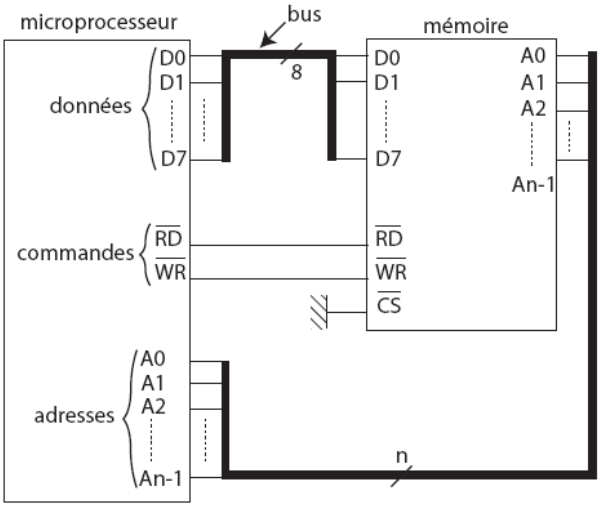


Figure n°15 : Représentation condensée de l'interfaçage microprocesseur/mémoire.

IV.4. Chronogrammes de lecture/écriture en mémoire :

Une caractéristique importante des mémoires est leur **temps d'accès** : c'est le temps qui s'écoule entre l'instant où l'adresse de la case mémoire est présentée sur le bus d'adresses et celui où la mémoire place la donnée demandée sur le bus de données. Ce temps varie entre 50ns (mémoires rapides) et 300ns (mémoires lentes).

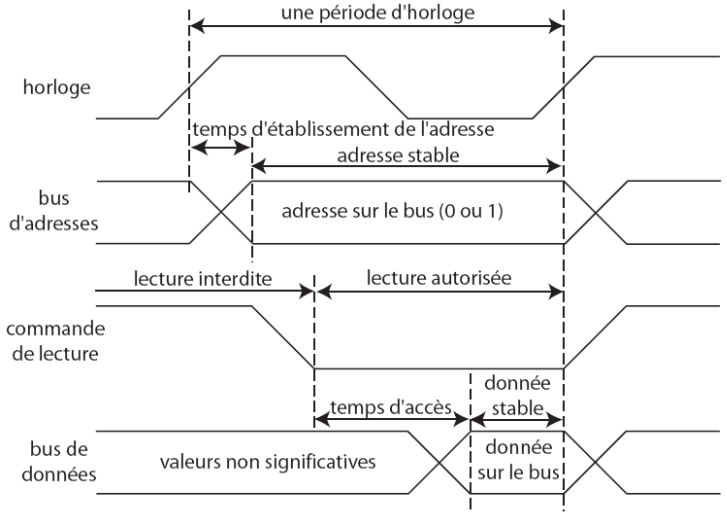
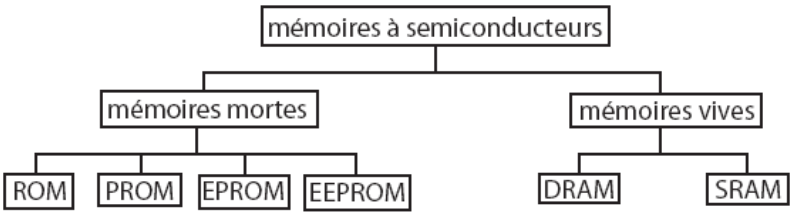


Figure n°16 : Chronogrammes de lecture/écriture en mémoire.

IV.5. Classification des mémoires

Jusqu'à la fin des années 1970, on utilisait des mémoires à tores magnétiques, lentes et de faibles capacités. Actuellement, on n'utilise plus que des mémoires à semi-conducteurs.



Mémoires mortes :

- **ROM** : Read Only Memory. Mémoire à lecture seule, sans écriture. Son contenu est programmé une fois pour toutes par le constructeur. Avantage : faible coût. Inconvénient : nécessite une production en très grande quantité.
- **PROM**: Programmable Read Only Memory. ROM programmable une seule fois par l'utilisateur (ROM OTP : One Time Programming) en faisant sauter des fusibles. Nécessite un programmateur spécialisé : application d'une tension de programmation (21 ou 25 V) pendant 20 ms.
- **EPROM** : Erasable PROM, appelée aussi UVPROM. ROM programmable électriquement avec un programmateur et effaçable par exposition à un rayonnement ultraviolet pendant 30 minutes. Famille 27nnn, exemple : 2764 (8 Ko), 27256 (32 Ko). Avantage : reprogrammable par l'utilisateur.
- **EEPROM** : Electrically Erasable PROM. ROM programmable et effaçable électriquement. Lecture à vitesse normale (≤ 100 ns). Ecriture (= effacement) très lente (≈ 10 ms). Application : les EEPROM contiennent des données qui peuvent être modifiées de temps en temps, exemple : paramètres de configuration des ordinateurs. Avantage : programmation sans extraction de la carte et sans programmateur. Inconvénient : coût élevé.

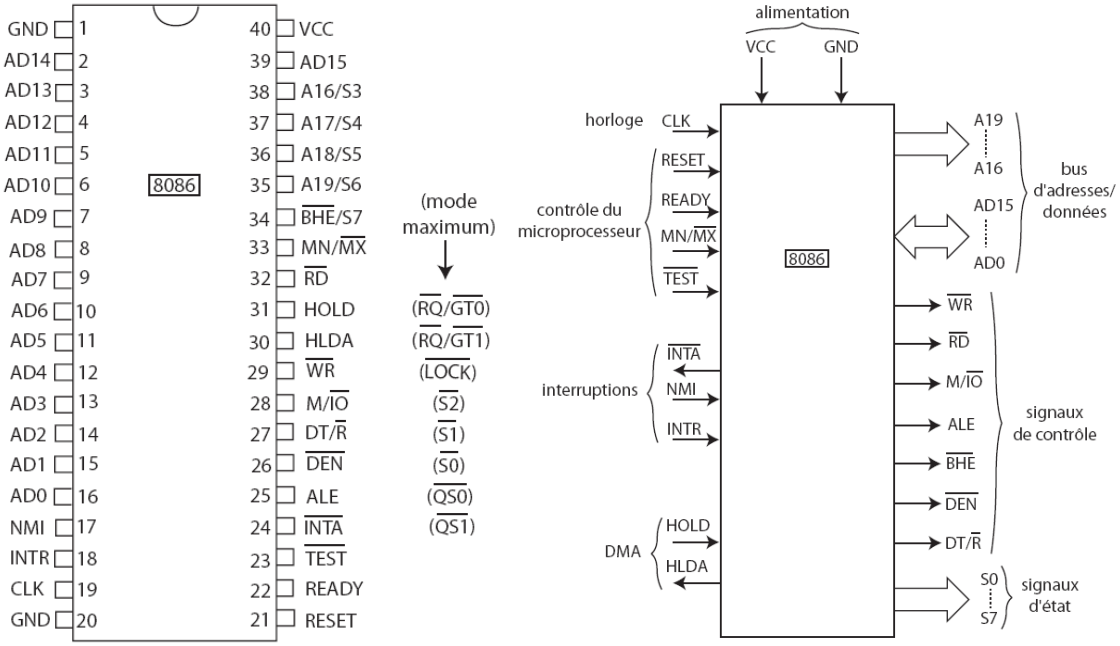
Mémoires vives :

- **SRAM** : Static Random Access Memory. Mémoire statique à accès aléatoire, à base de bascules à semi-conducteurs à deux états (bascules RS). Famille 62nnn, exemple : 62128 (16 Ko). Avantage : très rapide, simple d'utilisation. Inconvénient : compliqué à réaliser.
- **DRAM** : Dynamic RAM. Basée sur la charge de condensateurs : condensateur chargé = 1, condensateur déchargé = 0. Avantage : intégration élevée, faible coût. Inconvénient : nécessite un rafraîchissement périodique à cause du courant de fuite des condensateurs. Application : réalisation de la mémoire vive des ordinateurs (barrettes mémoire SIMM : Single In-line Memory module).

Chapitre II : Le microprocesseur Intel 8086

I. Description physique du 8086 :

Le microprocesseur Intel 8086 est un microprocesseur 16 bits, apparu en 1978. C'est le premier microprocesseur de la famille Intel 80x86 (8086, 80186, 80286, 80386, 80486, Pentium I, PII, ...). Il se présente sous la forme d'un boîtier DIP (Dual In-line Package) à 40 broches :



Boîtier DIP du microprocesseur 8086 *Schéma fonctionnel du 8086*

Figure n°1 : Boîtier DIP et schéma fonctionnel du microprocesseur 8086.

II. Description et utilisation des signaux du 8086 :

CLK : entrée du signal d'horloge qui cadence le fonctionnement du microprocesseur. Ce signal provient d'un **générateur d'Horloge** : le 8284.

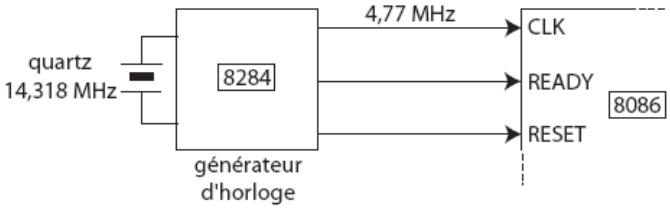


Figure n°2 : Générateur d'Horloge (8284) du microprocesseur 8086.

RESET : entrée de remise à zéro du microprocesseur. Lorsque cette entrée est mise à l'état haut pendant au moins 4 périodes d'horloge, le microprocesseur est réinitialisé : il va exécuter l'instruction se trouvant à l'adresse FFFF0H (adresse de bootstrap). Le signal de RESET est fourni par le générateur d'horloge.

READY : entrée de synchronisation avec la mémoire. Ce signal provient également du générateur d'horloge.

TEST : entrée de mise en attente du microprocesseur d'un événement extérieur.

MN/MX : entrée de choix du mode de fonctionnement du microprocesseur :

- Mode minimum (MN/MX = 1) : le 8086 fonctionne de manière autonome, il génère lui-même le bus de commande (RD, WR, ...) ;
- Mode maximum (MN/MX=0) : Ces signaux de commande sont produits par un **contrôleur de bus**, le 8288. Ce mode permet de réaliser des systèmes multiprocesseurs.

NMI et **INTR** : entrées de demande d'interruption. INTR : interruption normale, NMI (Non Maskable Interrupt) : interruption prioritaire.
INTA : Interrupt Acknowledge, indique que le microprocesseur accepte l'interruption.
HOLD et **HLDA** : signaux de demande d'accord d'accès direct à la mémoire (DMA).
S0 à **S7** : signaux d'état indiquant le type d'opération en cours sur le bus.
A16/S3 à **A19/S6** : 4 bits de poids fort du bus d'adresses, **multiplexés** avec 4 bits d'état.
AD0 à **AD15** : 16 bits de poids faible du bus d'adresses, **multiplexés** avec 16 bits de données.
Le bus A/D est multiplexé (multiplexage temporel) d'où la nécessité d'un **démultiplexage** pour obtenir séparément les bus d'adresses et de données :

- 16 bits de données (microprocesseur 16 bits) ;
- 20 bits d'adresses, d'où 2^{20} = 1 Méga adresses dans l'espace mémoire adressable par le 8086.

Chronogramme du bus A/D :

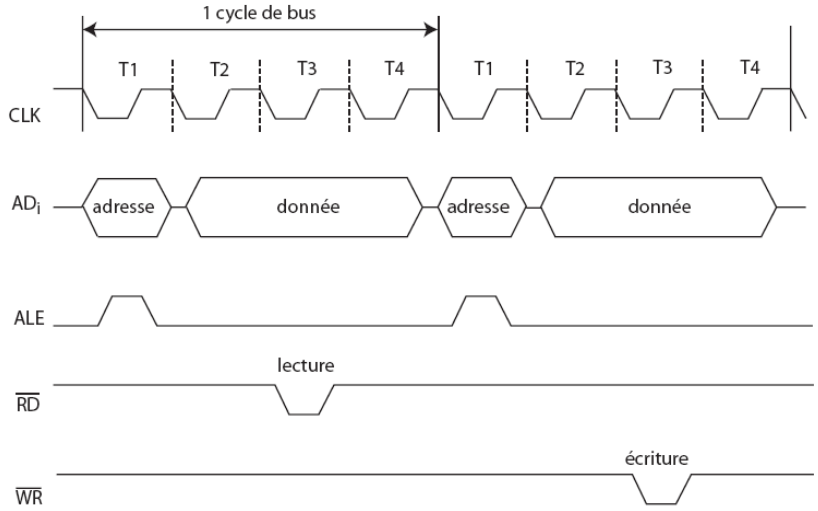


Figure n°3 : Chronogramme du bus A/D du microprocesseur 8086.

Le démultiplexage des signaux AD0 à AD15 (ou A16/S3 à A19/S6) se fait en mémorisant l'adresse lorsque celle-ci est présente sur le bus A/D, à l'aide d'un **verrou** (latch), ensemble de bascules D. La commande de mémorisation de l'adresse est générée par le microprocesseur : c'est le signal **ALE**, Address Latch Enable.

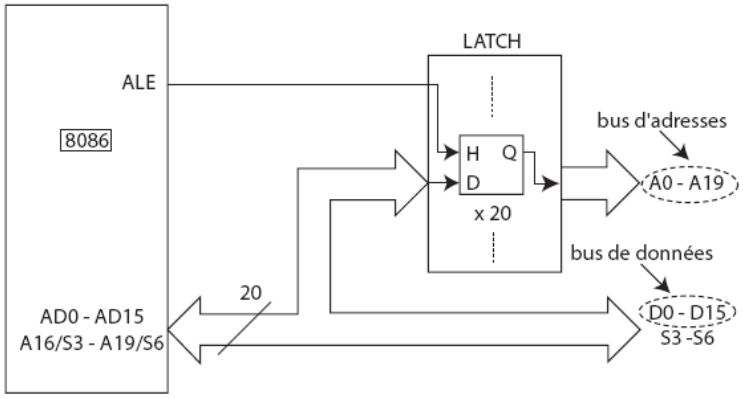


Figure n°4 : Circuit de démultiplexage A/D du microprocesseur 8086.

Fonctionnement :

- si ALE = 1, le verrou est transparent (Q = D) ;
- si ALE = 0, mémorisation de la dernière valeur de D sur les sorties Q;
- les signaux de lecture (RD) ou d'écriture (WR) ne sont générés par le microprocesseur que lorsque les données sont présentes sur le bus A/D.

Exemples de bascules D : circuits 8282, 74373, 74573.

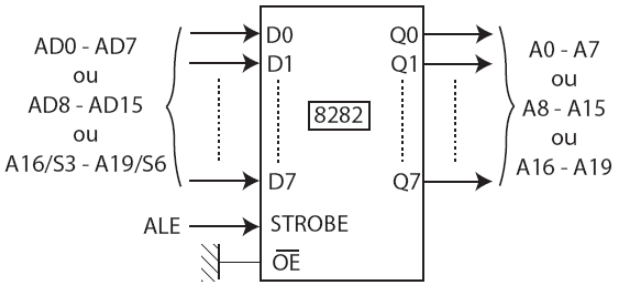


Figure n°5 : Exemples de bascules D : circuits 8282.

RD : Read, signal de lecture d'une donnée.
WR : Write, signal d'écriture d'une donnée.
M/IO : Memory/Input-Output, indique si le 8086 adresse la mémoire (M/IO = 1) ou les entrées/sorties (M/IO = 0).
DEN : Data Enable, indique que des données sont en train de circuler sur le bus A/D (équivalent de ALE pour les données).
DT/R : Data Transmit/Receive, indique le sens de transfert des données :

- DT/R = 1 : données émises par le microprocesseur (écriture) ;
- DT/R = 0 : données reçues par le microprocesseur (lecture).

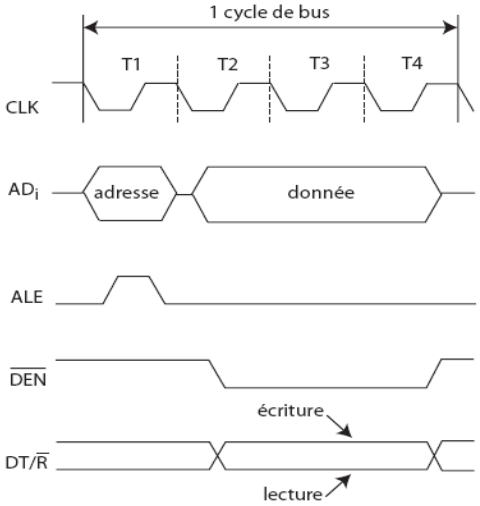


Figure n°6 : Chronogramme de bascules D.

Les signaux DEN et DT/R sont utilisés pour la commande de **tampons de bus** (Buffers) permettant d'amplifier le courant fourni par le microprocesseur sur le bus de données.

Exemples de tampons de bus : circuits transmetteurs bidirectionnels 8286 ou 74245.

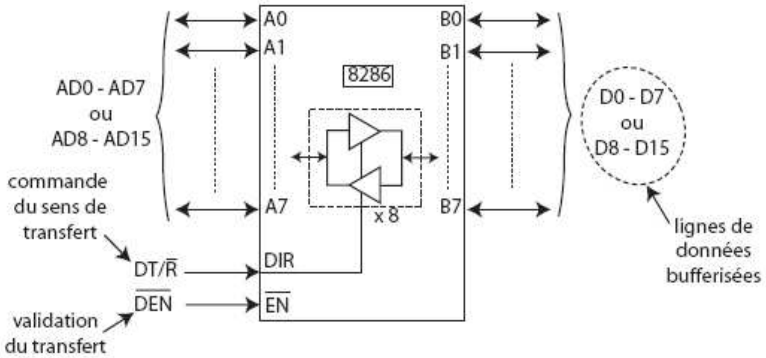


Figure n°7 : Circuits transmetteurs bidirectionnels 8286 ou 74245.

BHE : Bus High Enable, signal de lecture de l'octet de poids fort du bus de données. Le 8086 possède un bus d'adresses sur 20 bits, d'où la capacité d'adressage de 1 Mo ou 512 Kmots de 16 bits (bus de données sur 16 bits).
Le méga-octet adressable est divisé en deux **banques** de 512 Ko chacune : la banque **inférieure** (ou **paire**) et la banque **supérieure** (ou **impaire**). Ces deux banques sont sélectionnées par :

- A_0 pour la banque paire qui contient les octets de poids faible ;
- BHE pour la banque impaire qui contient les octets de poids fort.

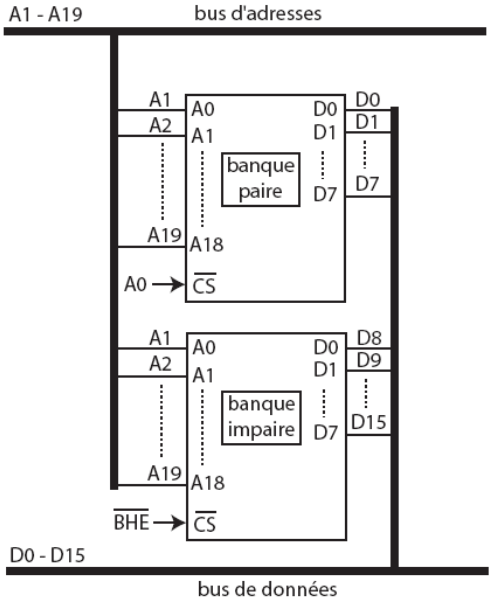


Figure n°8 : Banque inférieure (paire) et la banque supérieure (impaire) .

Seuls les bits A1 à A19 servent à désigner une case mémoire dans chaque banque de 512 Ko. Le microprocesseur peut ainsi lire et écrire des données sur 8 bits ou sur 16 bits :

BHE	A_0	Octets transférés
0	0	les deux octets (mot complet)
0	1	octet fort (adresse impaire)
1	0	octet faible (adresse paire)
1	1	aucun octet

Remarque : Le 8086 ne peut lire une donnée sur 16 bits en une seule fois, uniquement si l'octet de poids fort de cette donnée est rangé à une adresse impaire et l'octet de poids faible à une adresse paire (alignement sur les adresses paires), sinon la lecture de cette donnée doit se faire en deux opérations successives, d'où une augmentation du temps d'exécution du transfert dû à un mauvais alignement des données.

Réalisation des deux banques avec plusieurs boîtiers mémoire :

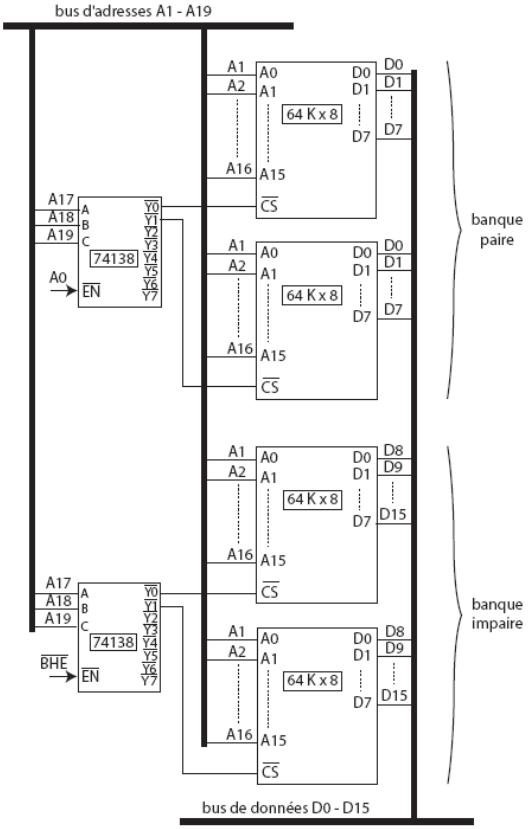


Figure n°9 : Réalisation des deux banques avec plusieurs boîtiers mémoire.

Création du bus système du 8086 :

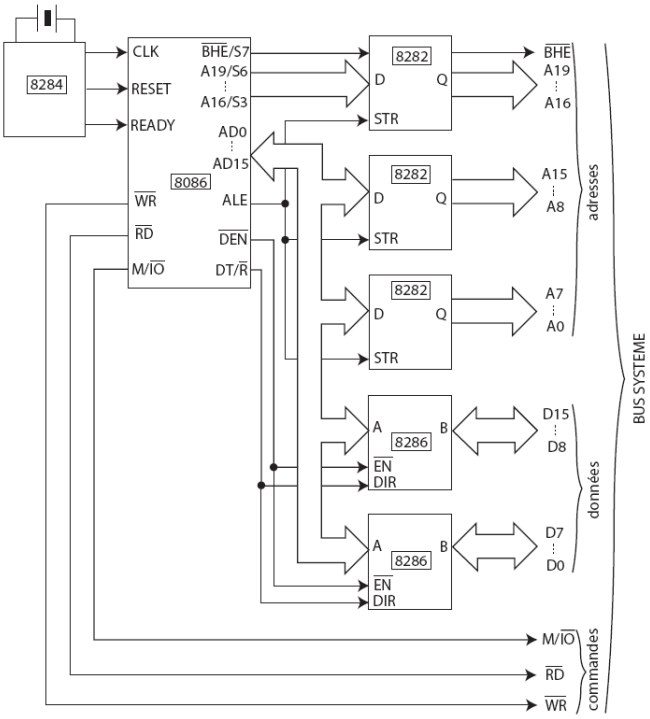


Figure n°10 : Création du bus système du 8086.

III. Organisation interne du 8086 :

Le 8086 est constitué de deux unités fonctionnant en parallèle :

- L'**unité d'exécution** (EU : Exécution Unit) ;
- L'**unité d'interface de bus** (BIU : Bus Interface Unit).

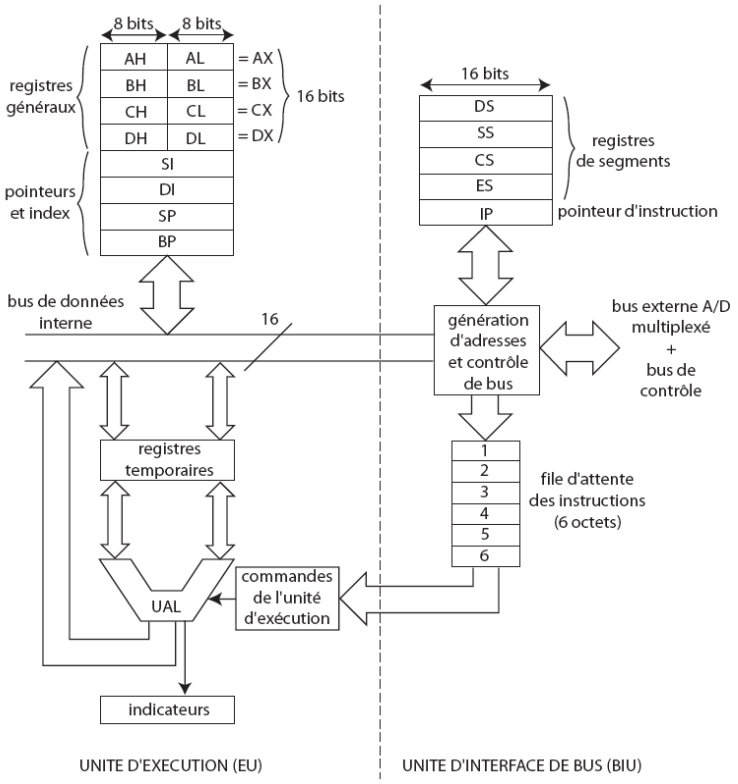


Figure n°11 : L'unité d'exécution et l'unité d'interface du 8086.

Rôle des deux unités :

- L'unité d'interface de bus (BIU) recherche les instructions en mémoire et les range dans une **file d'attente** ;
- L'unité d'exécution (EU) exécute les instructions contenues dans la file d'attente.

Les deux unités fonctionnent simultanément, d'où une accélération du processus d'exécution d'un programme (fonctionnement selon le principe du **pipe-line**).

Le microprocesseur 8086 contient 14 registres répartis en 4 groupes :

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

  AX = (**AH**,**AL**) ;
  BX = (**BH**,**BL**) ;
  CX = (**CH**,**CL**) ;
  DX = (**DH**,**DL**).

Ils peuvent être également considérés comme 8 registres sur 8 bits. Ils servent à contenir temporairement des données. Ce sont des registres généraux mais ils peuvent être utilisés pour des opérations particulières. Exemple : AX = accumulateur, CX = compteur.

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

  Pointeurs :

  SP : Stack Pointer, pointeur de pile (la pile est une zone de sauvegarde de données en cours d'exécution d'un programme) ;
  BP : Base Pointer, pointeur de base, utilisé pour adresser des données sur la pile.

  Index :

  SI : Source Index ;
  DI : Destination Index.

Ils sont utilisés pour les transferts de chaînes d'octets entre deux zones mémoire. Les pointeurs et les index contiennent des adresses de cases mémoire.

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

  Le pointeur d'instruction : IP, contient l'adresse de la prochaine instruction à exécuter.
  Le registre des indicateurs (flags)

Flags :

				O	D	I	T	S	Z		A		P		C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

C : indicateur de retenue (carry) ;
P : indicateur de parité ;
A : indicateur de retenue auxiliaire ;
Z : indicateur de zéro ;
S : indicateur de signe ;
T : indicateur d'exécution pas à pas (trap) ;
I : indicateur d'autorisation d'interruption ;
D : indicateur de décrémentation ;
O : indicateur de dépassement (overflow).

Figure n°12 : Le registre falgs du 8086.

- **Registres de segments :** 4 registres sur 16 bits.
  CS : Code Segment, registre de segment de code ;
  DS : Data Segment, registre de segment de données ;
  SS : Stack Segment, registre de segment de pile ;
  ES : Extra Segment, registre de segment supplémentaire pour les données ;

Les registres de segments, associés aux pointeurs et aux index, permettent au microprocesseur 8086 d'adresser l'ensemble de la mémoire.

IV. Gestion de la mémoire par le 8086 :

L'espace mémoire adressable par le 8086 est de 2^{20} = 1 048 576 octets = 1 Mo (20 bits d'adresses). Cet espace est divisé en **segments**. Un segment est une zone mémoire de 64 Ko (65 536 octets) définie par son adresse de départ qui doit être un multiple de 16.

Dans une telle adresse, les 4 bits de poids faible sont à zéro. On peut donc représenter l'adresse d'un segment avec seulement ses 16 bits de poids fort, les 4 bits de poids faible étant implicitement à 0.

Pour désigner une case mémoire parmi les 2^{16} = 65 536 contenues dans un segment, il suffit d'une valeur sur 16 bits.

Ainsi, une case mémoire est repérée par le 8086 au moyen de deux quantités sur 16 bits :

- L'adresse d'un segment ;
- Un déplacement ou **offset** (appelé aussi **adresse effective**) dans ce segment.

Cette méthode de gestion de la mémoire est appelée **segmentation de la mémoire**.

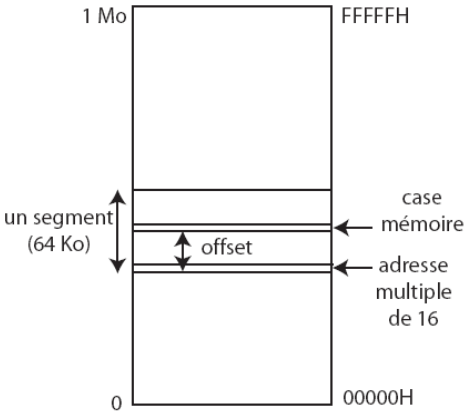


Figure n°13 : Segmentation de mémoire du 8086.

La donnée d'un couple (segment,offset) définit une **adresse logique**, notée sous la forme **segment : offset**.

L'adresse d'une case mémoire donnée sous la forme d'une quantité sur 20 bits (5 digits hexa) est appelée **adresse physique** car elle correspond à la valeur envoyée réellement sur le bus d'adresses A0 - A19.

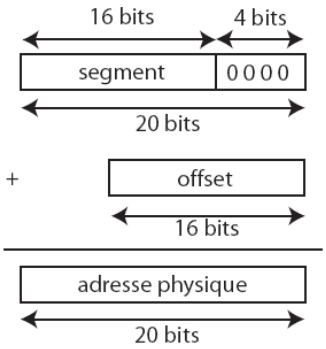


Figure n°14 : Correspondance entre adresse logique et adresse physique.

Ainsi, l'adresse physique se calcule par l'expression :

Adresse physique = 16 × segment + offset

Car le fait d'injecter 4 zéros en poids faible du segment revient à effectuer un décalage de 4 positions vers la gauche, c'est à dire une multiplication par $2^4 = 16$.

A un instant donné, le 8086 a accès à 4 segments dont les adresses se trouvent dans les registres de segment CS, DS, SS et ES. Le segment de code contient les instructions du programme, le segment de données contient les données manipulées par le programme, le segment de pile contient la pile de sauvegarde et le segment supplémentaire peut aussi contenir des données.

Le registre CS est associé au pointeur d'instruction IP, ainsi la prochaine instruction à exécuter se trouve à l'adresse logique CS : IP.

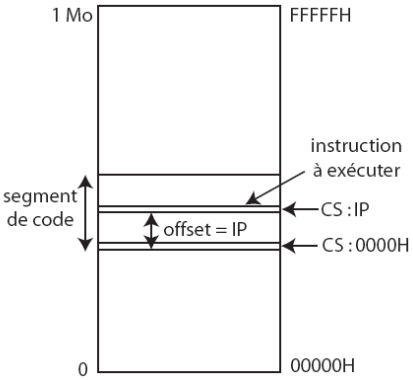


Figure n°15 :.

De même, les registres de segments DS et ES peuvent être associés à un registre d'index. Exemple : DS : SI, ES : DI. Le registre de segment de pile peut être associé aux registres de pointeurs : SS : SP ou SS : BP. Mémoire accessible par le 8086 à un instant donné :

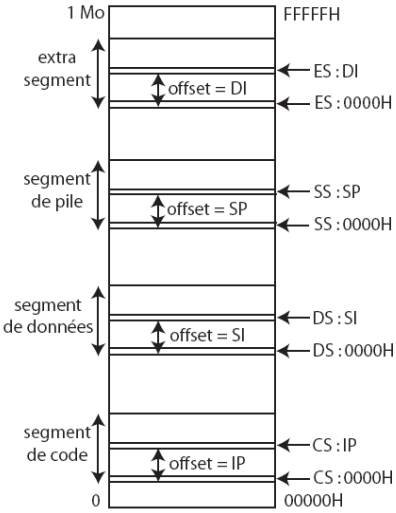


Figure n°16 : .

Remarque : les segments ne sont pas nécessairement distincts les uns des autres, ils peuvent se chevaucher ou se recouvrir complètement.

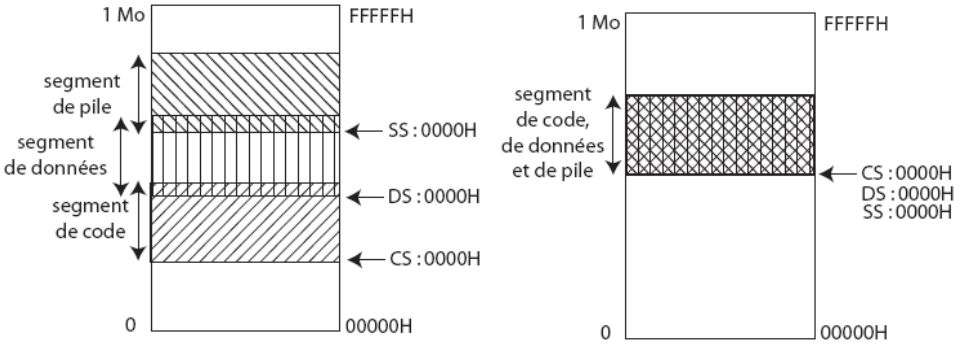


Figure n°17 : .

Le nombre de segments utilisé définit le **modèle mémoire** du programme. Contenu des registres après un RESET du microprocesseur :

IP = 0000H
CS = FFFFH
DS = 0000H
ES = 0000H
SS = 0000H

Puisque CS contient la valeur FFFFH et IP la valeur 0000H, la première instruction exécutée par le 8086 se trouve donc à l'adresse logique FFFFH : 0000H, correspondant à l'adresse physique FFFF0H (bootstrap). Cette instruction est généralement un saut vers le programme principal qui initialise ensuite les autres registres de segment.

Chapitre III : La programmation en assembleur du microprocesseur 8086

I. Généralités :

Chaque microprocesseur reconnaît un ensemble d'instructions appelé **jeu d'instructions** (Instruction Set) fixé par le constructeur. Pour les microprocesseurs classiques, le nombre d'instructions reconnues varie entre 75 et 150 (microprocesseurs **CISC** : Complex Instruction Set Computer). Il existe aussi des microprocesseurs dont le nombre d'instructions est très réduit (microprocesseurs **RISC** : Reduced Instruction Set Computer) : entre 10 et 30 instructions, permettant d'améliorer le temps d'exécution des programmes.

Une instruction est définie par son code opératoire, valeur numérique binaire difficile à manipuler par l'être humain. On utilise donc une **notation symbolique** pour représenter les instructions : les **mnémoniques**. Un programme constitué de mnémoniques est appelé **programme en assembleur**.

Les instructions peuvent être classées en groupes :

- Instructions de transfert de données ;
- Instructions arithmétiques ;
- Instructions logiques ;
- Instructions de branchement ...

II. Les instructions de transfert :

Elles permettent de déplacer des données d'une **source** vers une **destination** :

- Registre vers mémoire ;
- Registre vers registre ;
- Mémoire vers registre.

Syntaxe : MOV destination, source

Remarques :

- Le microprocesseur 8086 n'autorise pas les transferts de mémoire vers mémoire (pour ce faire, il faut passer par un registre intermédiaire).
- MOV est l'abréviation du verbe « to move » : déplacer.

II.1. Mode d'adressage :

Il existe différentes façons de spécifier l'adresse d'une case mémoire dans une instruction (ce sont les **modes d'adressage**).

Exemples de modes d'adressage simples :

- mov ax,bx : charge le contenu du registre BX dans le registre AX. Dans ce cas, le transfert se fait de registre à registre : **adressage par registre** ;
- mov al,12H : chargé le registre AL avec la valeur 12H. La donnée est fournie immédiatement avec l'instruction : **adressage immédiat**.
- mov bl,[1200H] : transfère le contenu de la case mémoire d'adresse effective (offset) 1200H vers le registre BL. L'instruction comporte l'adresse de la case mémoire où se trouve la donnée : **adressage direct**. L'adresse effective représente l'offset de la case mémoire dans le segment de données (segment dont l'adresse est contenue dans le registre DS) : segment par défaut.

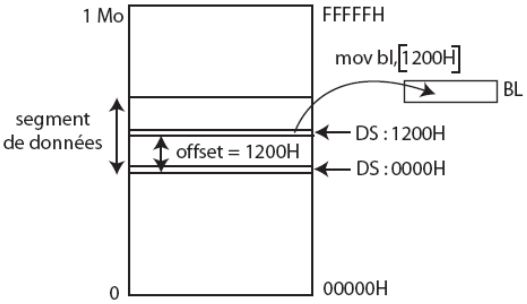


Figure n°1 : adressage direct.

On peut changer le segment lors d'un adressage direct en ajoutant un **préfixe de segment**, exemple : mov bl,es :[1200H]. On parle alors de **forçage de segment**.

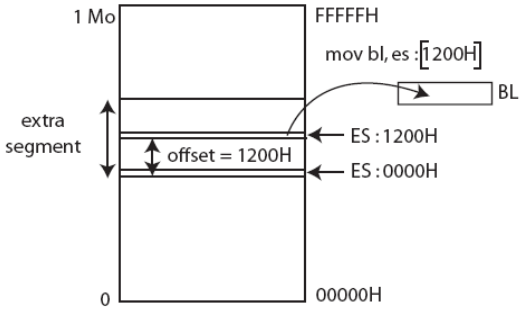


Figure n°2 : Forçage de segment.

Remarque : Dans le cas de l'adressage immédiat de la mémoire, il faut indiquer le **format** de la donnée : octet ou mot (2 octets) car le microprocesseur 8086 peut manipuler des données sur 8 bits ou 16 bits. Pour cela, on doit utiliser un **spécificateur de format** :

- mov byte ptr [1100H],65H : transfère la valeur 65H (sur 1 octet) dans la case mémoire d'offset 1100H ;
- mov word ptr [1100H],65H : transfère la valeur 0065H (sur 2 octets) dans les cases mémoire d'offset 1100H et 1101H.

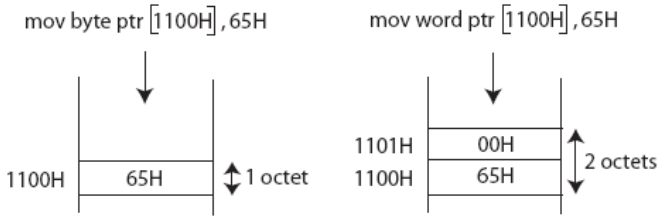


Figure n°3 : Spécification de format lors de l'adressage direct.

Remarque : Les microprocesseurs Intel rangent l'octet de poids faible d'une donnée sur plusieurs octets à l'adresse la plus basse (format **Little Endian**).

II.2. Modes d'adressage évolués :

- **Adressage basé** : l'offset est contenu dans un **registre de base** BX ou BP.

  Exemples :
  - mov al,[bx] : transfère la donnée dont l'offset est contenu dans le registre de base BX vers le registre AL. Le segment associé par défaut au registre BX est le segment de données : on dit que l'adressage est **basé sur DS**;
  - mov al,[bp] : le segment par défaut associé au registre de base BP est le segment de pile. Dans ce cas, l'adressage est **basé sur SS**.

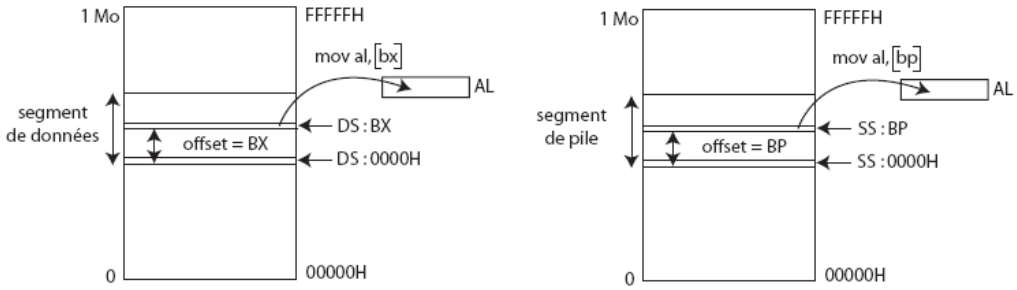


Figure n°4 : Adressage basé.

- **adressage indexé** : semblable à l'adressage basé, sauf que l'offset est contenu dans un registre d'index SI ou DI, associés par défaut au segment de données.

  Exemples :
  - mov al,[si] : charge le registre AL avec le contenu de la case mémoire dont l'offset est contenu dans SI ;

- mov [di],bx : charge les cases mémoire d'offset DI et DI + 1 avec le contenu du registre BX.

Remarque : Une valeur constante peut éventuellement être ajoutée aux registres de base ou d'index pour obtenir l'offset. Exemple :

mov [si+100H],ax
qui peut aussi s'écrire
mov [si][100H],ax
ou encore
mov 100H[si],ax

Les modes d'adressage basés ou indexés permettent la manipulation de tableaux rangés en mémoire. Exemple :

mov si,0
mov word ptr table[si],1234H
mov si,2
mov word ptr table[si],5678H

Dans cet exemple, table représente l'offset du premier élément du tableau et le registre SI joue le rôle d'indice de tableau :

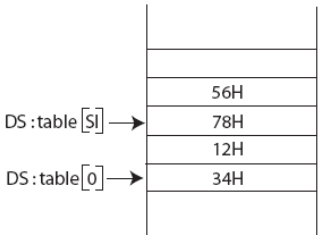


Figure n°5 : Adressage indexé.

- **Adressage basé et indexé** : l'offset est obtenu en faisant la somme d'un registre de base, d'un registre d'index et d'une valeur constante. Exemple :

  mov ah,[bx+si+100H]

  Ce mode d'adressage permet l'adressage de structures de données complexes : matrices, enregistrements, ... Exemple :

  mov bx,10
  mov si,15
  mov byte ptr matrice[bx][si],12H

  Dans cet exemple, BX et SI jouent respectivement le rôle d'indices de ligne et de colonne dans la matrice.

III. Les instructions arithmétiques :

Les instructions arithmétiques de base sont l'**addition**, la **soustraction**, la **multiplication** et la **division** qui incluent diverses variantes. Plusieurs modes d'adressage sont possibles.

III.1. L'addition :

ADD opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 + opérande2.

Exemples :

- add ah,[1100H] : ajoute le contenu de la case mémoire d'offset 1100H à l'accumulateur AH (adressage direct) ;
- add ah,[bx] : ajoute le contenu de la case mémoire pointée par BX à l'accumulateur AH (adressage basé) ;
- add byte ptr [1200H],05H : ajoute la valeur 05H au contenu de la case mémoire d'offset 1200H (adressage immédiat).

III.2. La soustraction :

SUB opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 − opérande2.

III.3. La multiplication :

MUL opérande, où opérande est un registre ou une case mémoire.

Cette instruction effectue la multiplication du contenu de AL par un opérande sur 1 octet ou du contenu de AX par un opérande sur 2 octets. Le résultat est placé dans AX si les données à multiplier sont sur 1 octet (résultat sur 16 bits), dans (DX,AX) si elles sont sur 2 octets (résultat sur 32 bits).

Exemples :

- mov al,51
  mov bl,32
  mul bl
  → AX = 51 × 32
- mov ax,4253
  mov bx,1689
  mul bx
  → (DX, AX) = 4253 × 1689
- mov al,43
  mov byte ptr [1200H],28
  mul byte ptr [1200H]
  → AX = 43 × 28
- mov ax,1234
  mov word ptr [1200H],5678
  mul word ptr [1200H]
  → (DX, AX) = 1234 × 5678

III.4. La division :

DIV opérande, où opérande est un registre ou une case mémoire.

Cette instruction effectue la division du contenu de AX par un opérande sur 1 octet ou le contenu de (DX,AX) par un opérande sur 2 octets. Résultat : si l'opérande est sur 1 octet, alors AL = quotient et AH = reste ; si l'opérande est sur 2 octets, alors AX = quotient et DX = reste.

Exemples :

- mov ax,35
  mov bl,10
  div bl
  → AL = 3 (quotient) et AH = 5 (reste)
- mov dx,0
  mov ax,1234
  mov bx,10
  div bx
  → AX = 123 (quotient) et DX = 4 (reste)

III.5. Autres instructions arithmétiques :

- ADC : addition avec retenue ;
- SBB : soustraction avec retenue ;
- INC : incrémentation d'une unité ;
- DEC : décrémentation d'une unité ;
- IMUL : multiplication signée ;
- IDIV : division signée.

IV. Les instructions logiques :

Ce sont des instructions qui permettent de manipuler des données au niveau des bits. Les opérations logiques de base sont :

- ET;
- OU;

- OU exclusif ;
- Complément à 1;
- Complément à 2;
- Décalages et rotations.

Les différents modes d'adressage sont disponibles.

IV.1. ET logique :

AND opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 ET opérande2.

Application : **masquage** de bits pour mettre à zéro certains bits dans un mot.

Exemple 01 :

mov al,10010110B
mov bl,11001101B
and al, bl
→ AL= 1 0 0 1 0 1 1 0
BL= 1 1 0 0 1 1 0 1
AL= 1 0 0 0 0 1 0 0

Exemple 02 : masquage des bits 0, 1, 6 et 7 dans un octet :

7 6 5 4 3 2 1 0
0 1 0 1 0 1 **1 1**
0 0 1 1 1 1 0 0 ← masque
0 0 0 1 0 1 **0 0**

IV.2. OU logique :

OR opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 OU opérande2.

Application : mise à 1 d'un ou de plusieurs bits dans un mot.

Exemple : dans le mot 10110001B on veut mettre à 1 les bits 1 et 3 sans modifier les autres bits.

7 6 5 4 3 2 1 0
1 0 1 1 **0** 0 **0** 1
0 0 0 0 1 0 1 0 ← masque
1 0 1 1 **1** 0 **1** 1

Les instructions correspondantes peuvent s'écrire :

mov ah,10110001B
or ah,00001010B

IV.3. Complément à 1 :

NOT opérande

L'opération effectuée est : opérande ← *NOT(*opérande).

Exemple :

mov al,10010001B
not al
→ AL = 10010001B
AL = 01101110B

IV.4. Complément à 2 :

NEG opérande

L'opération effectuée est : opérande ← *NOT(*opérande) + 1.

Exemple :

mov al,25
mov bl,12
neg bl
add al,bl
→ AL = 25 + (−12) = 13

IV.5. OU exclusif :

XOR opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 ⊕ opérande2.

Exemple : mise à zéro d'un registre :

```
mov al,25
xor al,al
```

→ AL = 0

IV.6. Instructions de décalages et de rotations :

Ces instructions déplacent d'un certain nombre de positions les bits d'un mot vers la gauche ou vers la droite.

Dans les décalages, les bits qui sont déplacés sont remplacés par des zéros. Il y a les décalages logiques (opérations non signées) et les décalages arithmétiques (opérations signées). Dans les rotations, les bits déplacés dans un sens sont réinjectés de l'autre côté du mot.

IV.6.1. Décalage logique vers la droite (Shift Right) :

SHR opérande,n

Cette instruction décale l'opérande de n positions vers la droite.

Exemple 01 :

```
mov al,11001011B
shr al,1
```

→ entrée d'un 0 à la place du bit de poids fort ; le bit sortant passe à travers l'indicateur de retenue CF.

Remarque : si le nombre de bits à décaler est supérieur à 1, ce nombre doit être placé dans le registre CL ou CX.

Exemple 02 : décalage de AL de trois positions vers la droite :

```
mov cl,3
shr al,cl
```

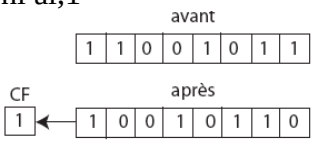
IV.6.2. Décalage logique vers la gauche (Shift Left) :

SHL opérande,n

Cette instruction décale l'opérande de n positions vers la droite.

Exemple :

```
mov al,11001011B
shl al,1
```



→ entrée d'un 0 à la place du bit de poids faible ; le bit sortant passe à travers l'indicateur de retenue CF.

Même remarque que précédemment si le nombre de positions à décaler est supérieur à 1.

IV.6.3. Décalage arithmétique vers la droite :

SAR opérande,n

Ce décalage conserve le bit de signe bien que celui-ci soit décalé.

Exemple :

```
mov al,11001011B
sar al,1
```

→ le bit de signe est **réinjecté**.

IV.6.4. Décalage arithmétique vers la gauche :

SAR opérande,n

Identique au décalage logique vers la gauche.

IV.6.5. Applications des instructions de décalage :

- **Cadrage à droite d'un groupe de bits.**

  Exemple : on veut avoir la valeur du quartet de poids fort du registre AL :

  mov al,11001011B
  mov cl,4
  shr al,cl

  → AL = 0000**1100**B

- **Test de l'état d'un bit dans un mot.**

  Exemple : on veut déterminer l'état du bit 5 de AL :

  mov cl,6
  shr al,cl
  ou
  mov cl,3
  shl al,cl

  → avec un décalage de 6 positions vers la droite ou 4 positions vers la gauche, le bit 5 de AL est transféré dans l'indicateur de retenue CF. Il suffit donc de tester cet indicateur.

- **Multiplication ou division par une puissance de 2 :** Un décalage à droite revient à faire une division par 2 et un décalage à gauche, une multiplication par 2.

  Exemple :

  mov al,48
  mov cl,3
  shr al,cl

  → AL = 48/23 = 6

IV.6.6. Rotation à droite (Rotate Right) :

ROR opérande,n

Cette instruction décale l'opérande de n positions vers la droite et réinjecte par la gauche les bits sortant.

Exemple :

mov al,11001011B
ror al,1

→ réinjection du bit sortant qui est copié dans l'indicateur de retenue CF.

IV.6.7. Rotation à gauche (Rotate Left) :

ROL opérande,n

Cette instruction décale l'opérande de n positions vers la gauche et réinjecte par la droite les bits sortant.

Exemple :

mov al,11001011B
rol al,1

→ réinjection du bit sortant qui est copié dans l'indicateur de retenue CF.

IV.6.8. Rotation à droite avec passage par l'indicateur de retenue (Rotate Right through Carry)

RCR opérande,n

Cette instruction décale l'opérande de n positions vers la droite en passant par l'indicateur de retenue CF.

Exemple :

mov al,11001011B
rcr al,1

→ le bit sortant par la droite est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la gauche.

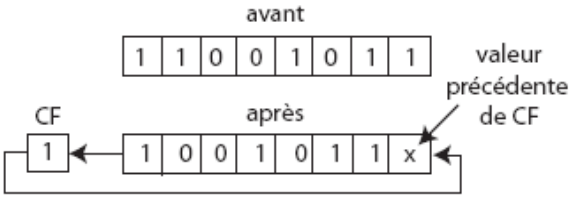
IV.6.9. Rotation à gauche avec passage par l'indicateur de retenue (Rotate Left through Carry) :

RCL opérande,n

Cette instruction décale l'opérande de n positions vers la gauche en passant par l'indicateur de retenue CF.

Exemple :

mov al,11001011B
rcl al,1



→ le bit sortant par la gauche est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la droite.

V. Les instructions de branchement :

Les instructions de branchement (ou **saut**) permettent de modifier l'ordre d'exécution des instructions du programme en fonction de certaines conditions. Il existe 3 types de saut :

- Saut inconditionnel ;
- Sauts conditionnels ;
- Sppel de sous-programmes.

V.1. Instruction de saut inconditionnel : JMP label

Cette instruction effectue un saut (**jump**) vers le label spécifié. Un **label** (ou **étiquette**) est une représentation symbolique d'une instruction en mémoire :

Exemple :

```
                ⋮ } ← instructions précédant le saut
          jmp suite
                ⋮ } ← instructions suivant le saut (jamais exécutées)
suite :   ...       ← instruction exécutée après le saut

boucle :  inc ax
          dec bx          →      boucle infinie
          jmp boucle
```

Remarque : l'instruction JMP ajoute au registre IP (pointeur d'instruction) le nombre d'octets (distance) qui sépare l'instruction de saut de sa destination. Pour un saut en arrière, la distance est négative (codée en complément à 2).

V.2. Instructions de sauts conditionnels : *condition* label

Un saut conditionnel n'est exécuté que si une certaine condition est satisfaite, sinon l'exécution se poursuit séquentiellement à l'instruction suivante.

La condition du saut porte sur l'état de l'un ou des indicateurs d'état (flags) du microprocesseur :

instruction	nom	condition
JZ label	Jump if Zero	saut si ZF = 1
JNZ label	Jump if Not Zero	saut si ZF = 0
JE label	Jump if Equal	saut si ZF = 1
JNE label	Jump if Not Equal	saut si ZF = 0
JC label	Jump if Carry	saut si CF = 1
JNC label	Jump if Not Carry	saut si CF = 0
JS label	Jump if Sign	saut si SF = 1
JNS label	Jump if Not Sign	saut si SF = 0
JO label	Jump if Overflow	saut si OF = 1
JNO label	Jump if Not Overflow	saut si OF = 0
JP label	Jump if Parity	saut si PF = 1
JNP label	Jump if Not Parity	saut si PF = 0

Remarque1 : les indicateurs sont positionnés en fonction du résultat de la dernière opération.

Exemple1 :

```
                ⋮ } ← instructions précédant le saut conditionnel
          jnz suite
                ⋮ } ← instructions exécutées si la condition ZF = 0 est vérifiée
suite :   ...       ← instruction exécutée à la suite du saut
```

Remarque2 : il existe un autre type de saut conditionnel, les **sauts arithmétiques**. Ils suivent en général l'instruction de comparaison : CMP opérande1,opérande2

condition	nombres signés	nombres non signés
=	JEQ label	JEQ label
>	JG label	JA label
<	JL label	JB label
≠	JNE label	JNE label

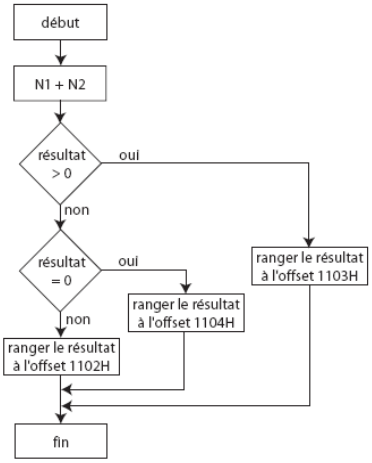
Exemple2 :

```
              cmp ax,bx
              jg superieur
              jl inferieur
superieur :   ...
                  ⋮
inferieur :   ...
```

V.3. Exemple d'application des instructions de sauts conditionnels :

On veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul :

Organigramme :



Programme :

```
            mov    al,[1100H]
            add    al,[1101H]
            js     negatif
            jz     nul
            mov    [1102H],al
            jmp    fin
negatif :   mov    [1103H],al
            jmp    fin
nul :       mov    [1104H],al
fin :       hlt
```

VI. Appel de sous-programmes :

Pour éviter la répétition d'une même séquence d'instructions plusieurs fois dans un programme, on rédige la séquence une seule fois en lui attribuant un nom (au choix) et on l'appelle lorsqu'on en a besoin. Le programme appelant est le **programme principal**. La séquence appelée est un **sous-programme** ou **procédure**.

VI.1. Ecriture d'un sous-programme :

```
nom_sp    PROC
            ⋮ } ← instructions du sous-programme
          ret   ← instruction de retour au programme principal
nom_sp    ENDP
```

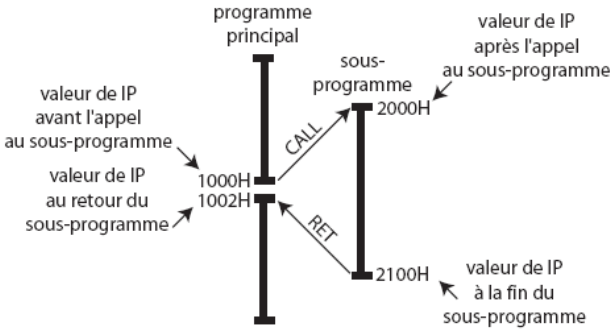
Remarque : une procédure peut être de type NEAR si elle se trouve dans le même segment ou de type FAR si elle se trouve dans un autre segment. Exemple :

```
ss_prog1    PROC   NEAR
ss_prog2    PROC   FAR
```

VI.2. Appel d'un sous-programme par le programme principal : CALL procédure

```
        ⋮ }        ← instructions précédant l'appel au sous-programme
 call nom_sp       ← appel au sous-programme
        ⋮ }        ← instructions exécutées après le retour au programme principal
```

Lors de l'exécution de l'instruction CALL, le pointeur d'instruction IP est chargé avec l'adresse de la première instruction du sous-programme. Lors du retour au programme appelant, l'instruction suivant le CALL doit être exécutée, c'est-à-dire que IP doit être rechargé avec l'adresse de cette instruction.



Avant de charger IP avec l'adresse du sous-programme, l'adresse de retour au programme principal, c'est-à-dire le contenu de IP, est sauvegardée dans une zone mémoire particulière appelée **pile**. Lors de l'exécution de l'instruction RET, cette adresse est récupérée à partir de la pile et rechargée dans IP, ainsi le programme appelant peut se poursuivre.

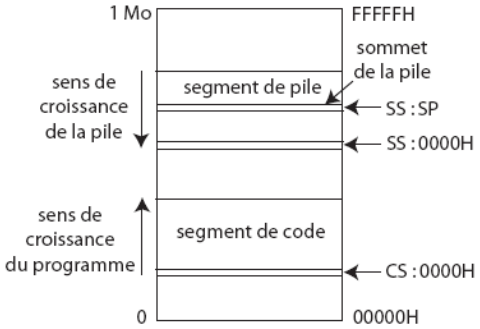
VI.3. Fonctionnement de la pile :

La pile est une zone mémoire fonctionnant en mode **LIFO** (Last In First Out : dernier entrée, premier sortie). Deux opérations sont possibles sur la pile :

- **Empiler** une donnée : placer la donnée au sommet de la pile ;
- **Dépiler** une donnée : lire la donnée se trouvant au sommet de la pile.

Le sommet de la pile est repéré par un registre appelé **pointeur de pile** (SP : Stack Pointer) qui contient l'adresse de la dernière donnée empilée.

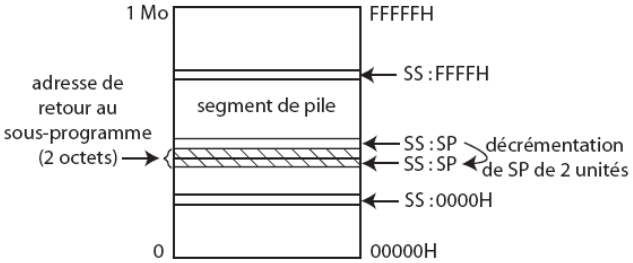
La pile est définie dans le **segment de pile** dont l'adresse de départ est contenue dans le registre SS.



Remarque :

La pile et le programme croissent en sens inverse pour diminuer le risque de collision entre le code et la pile dans le cas où celle-ci est placée dans le même segment que le code (SS = CS).

Lors de l'appel à un sous-programme, l'adresse de retour au programme appelant (contenu de IP) est empilée et le pointeur de pile SP est automatiquement **décrémenté**. Au retour du sous-programme, le pointeur d'instruction IP est rechargé avec la valeur contenue sommet de la pile et SP est **incrémenté**.



La pile peut également servir à sauvegarder le contenu de registres qui ne sont pas automatiquement sauvegardés lors de l'appel `a un sous programme :

- Instruction d'empilage : PUSH opérande
- Instruction de dépilage : POP opérande

où opérande est un registre ou une donnée sur 2 octets (on ne peut empiler que des mots de 16 bits).

Exemple :

```
push ax        ; empilage du registre AX ...
push bx        ; ... du registre BX ...
push [1100H]   ; ... et de la case mémoire 1100H-1101H
     ⋮
pop  [1100H]   ; dépilage dans l'ordre inverse de l'empilage
pop  bx
pop  ax
```

Remarque : la valeur de SP doit être initialisée par le programme principal avant de pouvoir utiliser la pile.

VI.4. Utilisation de la pile pour le passage de paramètres :

Pour transmettre des paramètres à une procédure, on peut les placer sur la pile avant l'appel de la procédure, puis celle-ci les récupère en effectuant un adressage basé de la pile en utilisant le registre BP.

Exemple :

Soit une procédure effectuant la somme de deux nombres et retournant le résultat dans le registre AX :

- **Programme principal :**

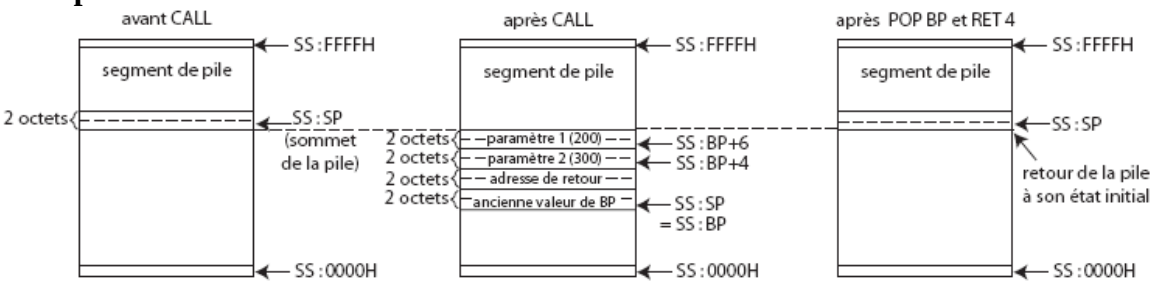
```
mov  ax,200
push ax        ; empilage du premier paramètre
mov  ax,300
push ax        ; empilage du deuxième paramètre
call somme     ; appel de la procédure somme
```

- **Procédure somme :**

```
somme    proc
         push bp        ; sauvegarde de BP
         mov  bp,sp     ; faire pointer BP sur le sommet de la pile
         mov  ax,[bp+4] ; récupération du deuxième paramètre
         add  ax,[bp+6] ; addition au premier paramètre
         pop  bp        ; restauration de l'ancienne valeur de BP
         ret  4         ; retour et dépilage des paramètres
somme    endp
```

L'instruction ret 4 permet de retourner au programme principal et d'incrémenter le pointeur de pile de 4 unités pour dépiler les paramètres afin de remettre la pile dans son état initial.

Etat de la pile :



VII. Méthodes de programmation :

VII.1. Etapes de la réalisation d'un programme :

- Définir le problème à résoudre : que faut-il faire exactement ?
- Déterminer des algorithmes, des organigrammes : comment faire ? Par quoi commencer, puis poursuivre ?
- Rédiger le programme (code source) :
  - Utilisation du jeu d'instructions (mnémoniques) ;
  - création de documents explicatifs (documentation).
- Tester le programme en réel ;
- Corriger les erreurs (bugs) éventuelles : déboguer le programme puis refaire des tests jusqu'à obtention d'un programme fonctionnant de manière satisfaisante.

VII.2. Langage machine et assembleur :

- Langage machine : codes binaires correspondant aux instructions ;
- Assembleur : logiciel de traduction du code source écrit en langage assembleur (mnémoniques).

VII.3. Réalisation pratique d'un programme :

- Rédaction du code source en assembleur à l'aide d'un éditeur (logiciel de traitement de texte ASCII) :
  - edit sous MS-DOS,
  - notepad (bloc-note) sous Windows,
- Assemblage du code source (traduction des instructions en codes binaires) avec un assembleur :
  - MASM de Microsoft,
  - TASM de Borland,
  - A86 disponible en shareware sur Internet, ...
  - SIM 8086

  Pour obtenir le **code objet** : code machine exécutable par le microprocesseur ;
- Chargement en mémoire centrale et exécution : rôle du système d'exploitation (ordinateur) ou d'un moniteur (carte de développement à base de microprocesseur).

Pour la mise au point (débogage) du programme, on peut utiliser un programme d'aide à la mise au point (comme DEBUG sous MS-DOS) permettant :

- L'exécution pas à pas;
- La visualisation du contenu des registres et de la mémoire ;
- La pose de points d'arrêt ...

VII.4. Structure d'un fichier source en assembleur :

Pour faciliter la lisibilité du code source en assembleur, on le rédige sous la forme suivante :

```
labels        instructions        commentaires
label1 :      mov ax,60H          ; ceci est un commentaire ...
     ⋮            ⋮                   ⋮
sous_prog1    proc near           ; sous-programme
     ⋮            ⋮                   ⋮
sous_prog1    endp
     ⋮            ⋮                   ⋮
```

VII.5. Directives pour l'assembleur :

- Origine du programme en mémoire : ORG offset

  Exemple : org 1000H
- Définitions de constantes : nom constante EQU valeur

  Exemple : escape equ 1BH
- Réservation de cases mémoires :

```
nom_variable DB valeur_initiale
nom_variable DW valeur_initiale
```

DB : **Define Byte**, réservation d'un octet ;

DW : **Define Word**, réservation d'un mot (2 octets).

Exemples :

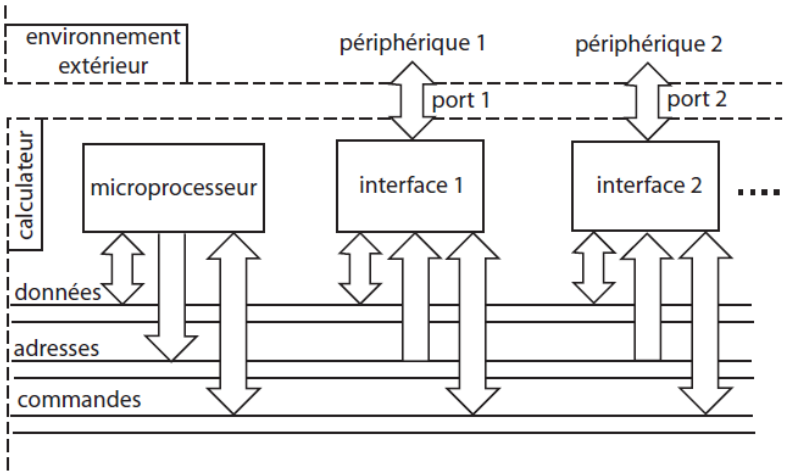
```
nombre1  db   25
nombre2  dw   ?              ; pas de valeur initiale
buffer   db   100 dup (?)    ; réservation d'une zone mémoire
                             ; de 100 octets
```

Chapitre IV : Les interfaces d'entrées/sorties

I. Définitions :

Une **interface d'entrées/sorties** est un circuit intégré permettant au microprocesseur de communiquer avec l'environnement extérieur (périphériques) : clavier, écran, imprimante, modem, disques, processus industriel, ...

Les interfaces d'E/S sont connectées au microprocesseur à travers les bus d'adresses, de données et de commandes.

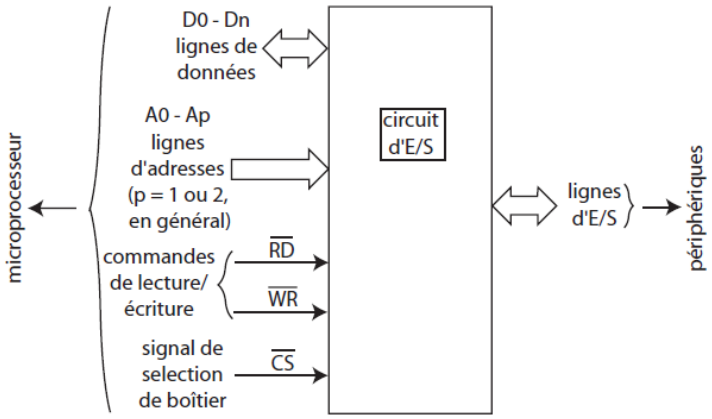


Les points d'accès aux interfaces sont appelés ports.

Exemples :

interface	port	exemple de périphérique
interface parallèle	port parallèle	imprimante
interface série	port série	modem

Schéma synoptique d'un circuit d'E/S :



II. Adressage des ports d'E/S :

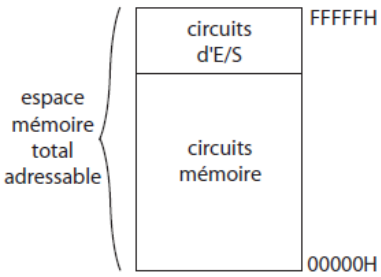
Un circuit d'E/S possède des registres pour gérer les échanges avec les périphériques :

- Registres de configuration ;
- Registres de données.

A chaque registre est assigné une adresse : le microprocesseur accède à un port d'E/S en spécifiant l'adresse de l'un de ses registres.

Le microprocesseur peut voir les adresses des ports d'E/S de deux manières :

- **Adressage cartographique** : les adresses des ports d'E/S appartiennent au même espace mémoire que les circuits mémoire (on dit que les E/S sont mappées en mémoire) :



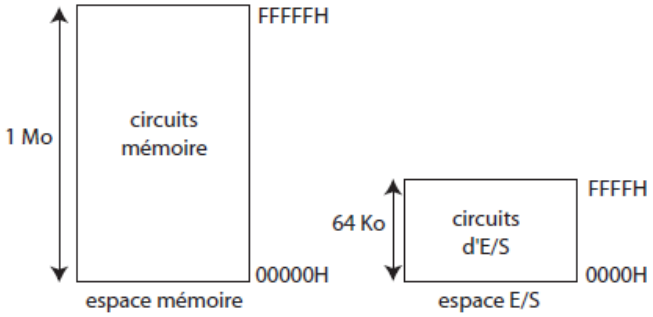
Conséquences :

- L'espace d'adressage des mémoires diminue ;
- L'adressage des ports d'E/S se fait avec une adresse de même longueur (même nombre de bits) que pour les cases mémoires ;
- Toutes les instructions employées avec des cases mémoires peuvent être appliquées aux ports d'E/S : les mêmes instructions permettent de lire et écrire dans la mémoire et les ports d'E/S, tous les modes d'adressage étant valables pour les E/S.

➢ **Adressage indépendant :** le microprocesseur considère deux espaces distincts :

- L'espace d'adressage des mémoires ;
- L'espace d'adressage des ports d'E/S.

C'est le cas du microprocesseur 8086 :



Conséquences :

- Contrairement à l'adressage cartographique, l'espace mémoire total adressable n'est pas diminué ;
- L'adressage des port d'E/S peut se faire avec une adresse plus courte (nombre de bits inférieur) que pour les circuits mémoires ;
- Les instructions utilisées pour l'accès à la mémoire ne sont plus utilisables pour l'accès aux ports d'E/S : ceux-ci disposent d'instructions spécifiques ;
- Une même adresse peut désigner soit une case mémoire, soit un port d'E/S : le microprocesseur doit donc fournir un signal permettant de différencier l'adressage de la mémoire de l'adressage des ports d'E/S.

Remarque : L'adressage indépendant des ports d'E/S n'est possible que pour les microprocesseurs possédant un signal permettant de différencier l'adressage de la mémoire de l'adressage des ports d'E/S ainsi que les instructions spécifiques pour l'accès aux ports d'E/S. Par contre, l'adressage cartographique est possible pour tous les microprocesseurs.

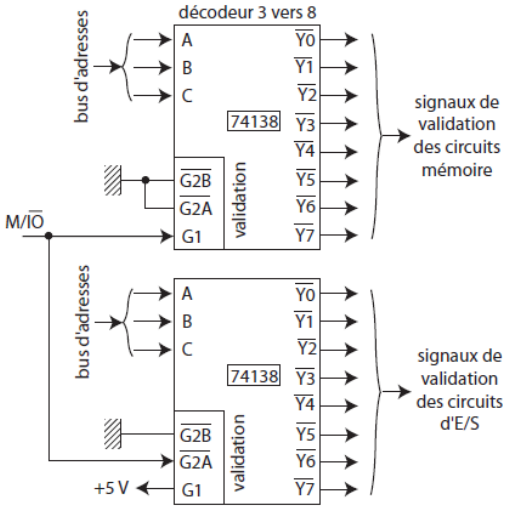
III. Gestion des ports d'E/S par le 8086 :

Le 8086 dispose d'un espace mémoire de 1 Mo (adresse d'une case mémoire sur 20 bits) et d'un espace d'E/S de 64 Ko (adresse d'un port d'E/S sur 16 bits).

Le signal permettant de différencier l'adressage de la mémoire de l'adressage des ports d'E/S est la ligne **M/IO** :

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

Ce signal est utilisé pour valider le décodage d'adresse dans les deux espaces :



Les instructions de lecture et d'écriture d'un port d'E/S sont respectivement les instructions **IN** et **OUT**. Elles placent la ligne M/IO à 0 alors que l'instruction MOV place celle-ci à 1.

III.1. Lecture d'un port d'E/S :

- Si l'adresse du port d'E/S est sur un octet :
  - IN AL,adresse : lecture d'un port sur 8 bits ;
  - IN AX,adresse : lecture d'un port sur 16 bits.
- Si l'adresse du port d'E/S est sur deux octets :
  - IN AL,DX : lecture d'un port sur 8 bits ;
  - IN AX,DX : lecture d'un port sur 16 bits.
  - Où le registre DX contient l'adresse du port d'E/S à lire.

III.2. Ecriture d'un port d'E/S :

- Si l'adresse du port d'E/S est sur un octet :
  - OUT adresse,AL : écriture d'un port sur 8 bits ;
  - OUT adresse,AX : écriture d'un port sur 16 bits.
- Si l'adresse du port d'E/S est sur deux octets :
  - OUT DX,AL : écriture d'un port sur 8 bits ;
  - OUT DX,AX : écriture d'un port sur 16 bits.

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

Exemples :

- Lecture d'un port d'E/S sur 8 bits à l'adresse 300H :

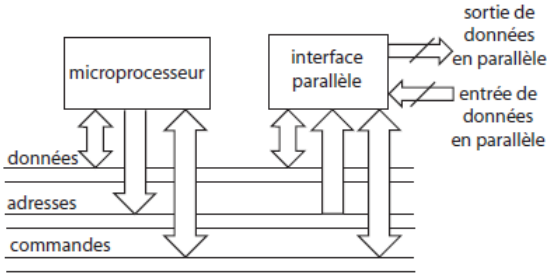
```
mov dx,300H
in al,dx
```

- Ecriture de la valeur 1234H dans le port d'E/S sur 16 bits à l'adresse 49H :

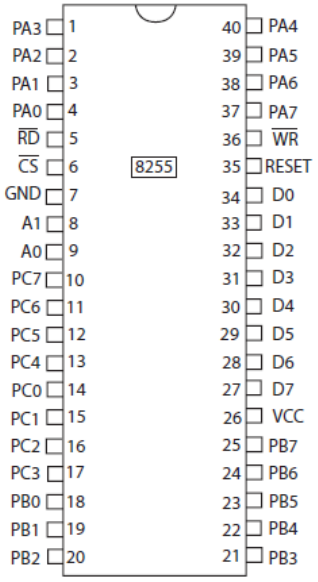
```
mov ax,1234H
out 49H,ax
```

IV. L'interface parallèle 8255 :

Le rôle d'une interface parallèle est de transférer des données du microprocesseur vers des périphériques et inversement, tous les bits de données étant envoyés ou reçus simultanément.



Le 8255 est une interface parallèle programmable : elle peut être configurée en entrée et/ou en sortie par programme.



Brochage du 8255

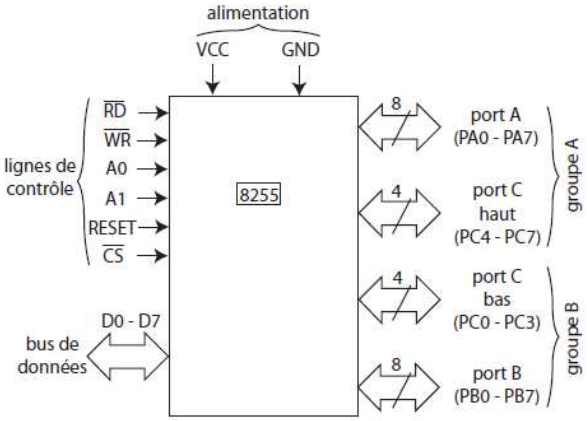


Schéma fonctionnel :

Le 8255 contient 4 registres :

- Trois registres contenant les données présentes sur les ports A, B et C;
- Un registre de commande pour la configuration des ports A, B et C en entrées et/ou en sorties.

IV.1. Accès aux registres du 8255 :

Les lignes d'adresses A0 et A1 définissent les adresses des registres du 8255 :

A1	A0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	opération
0	0	0	1	0	lecture du port A
0	1	0	1	0	lecture du port B
1	0	0	1	0	lecture du port C
0	0	1	0	0	écriture du port A
0	1	1	0	0	écriture du port B
1	0	1	0	0	écriture du port C
1	1	1	0	0	écriture du registre de commande
X	X	X	X	1	pas de transaction
1	1	0	1	0	illégal
X	X	1	1	0	pas de transaction

Remarque : Le registre de commande est accessible uniquement en écriture, la lecture de ce registre n'est pas autorisée.

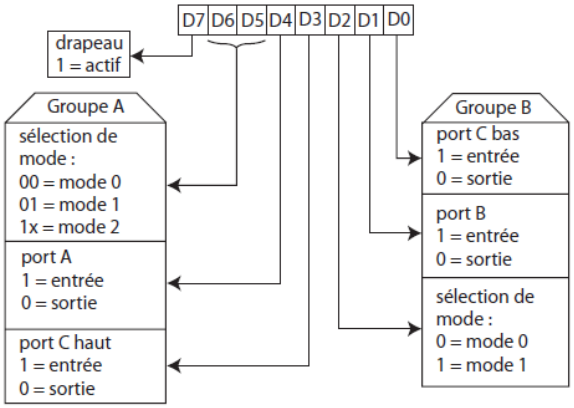
IV.2. Configuration du 8255 :

Les ports peuvent être configurés en entrées ou en sorties selon le contenu du registre de commande. De plus le 8255 peut fonctionner selon 3 modes : **mode 0, mode 1 ou mode 2.**

Le mode 0 est le plus simple : les ports sont configurés en entrées/sorties de base. Les données écrites dans les registres correspondants sont mémorisées sur les lignes de sorties ; l'état des lignes d'entrées est recopié dans les registres correspondants et n'est pas mémorisé.

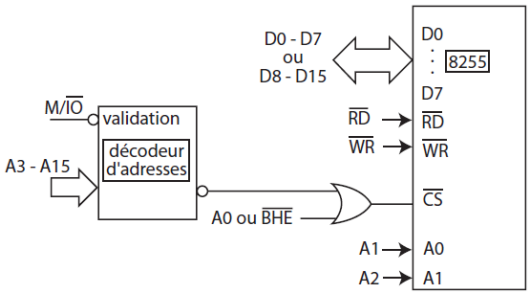
Les modes 1 et 2 sont plus complexes. Ils sont utilisés pour le dialogue avec des périphériques nécessitant un asservissement.

IV.3. Structure du registre de commande :



Connexion du 8255 sur les bus du 8086 : le bus de données du 8255 est sur 8 bits alors que celui du microprocesseur 8086 est sur 16 bits. On peut donc connecter le bus de données du 8255 sur les lignes de données de poids faible du 8086 (D0 - D7) ou sur celles de poids fort (D8 - D15). Une donnée est envoyée (ou reçue) par le microprocesseur 8086 :

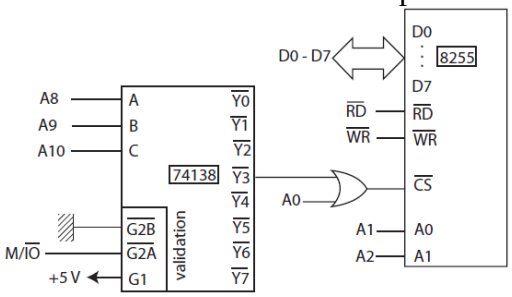
- Sur la partie faible du bus de données lorsque l'adresse à écrire (ou à lire) est paire : validation par A0 ;
- Sur la partie haute lorsque l'adresse est impaire : validation par BHE. Ainsi l'un de ces deux signaux A0 ou BHE doit être utilisé pour sélectionner le 8255 :



Conséquence : les adresses des registres du 8255 se trouvent à des adresses paires (validation par A0) ou impaires (validation par BHE).

Le décodeur d'adresses détermine l'adresse de base du 8255 ; les lignes A1 et A2 déterminent les adresses des registres du 8255.

Exemple : connexion du 8255 sur la partie faible du bus de données du 8086, avec décodage d'adresses incomplet (les lignes d'adresses A3 - A15 ne sont pas toutes utilisées) :



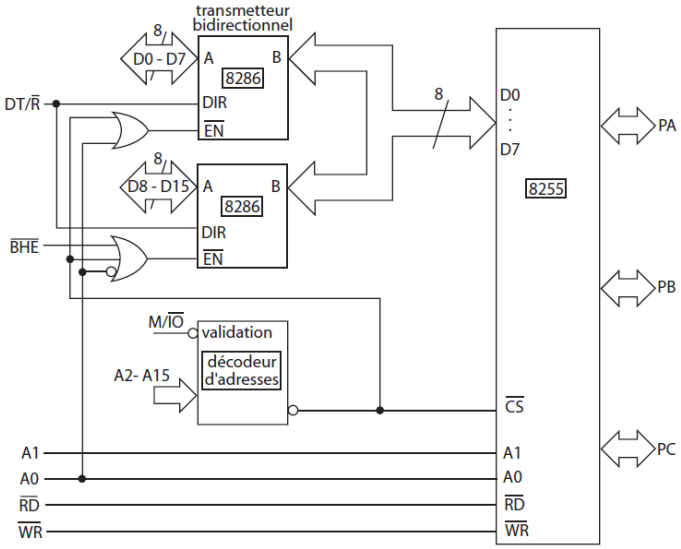
IV.4. Détermination de l'adresse du 8255 :

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

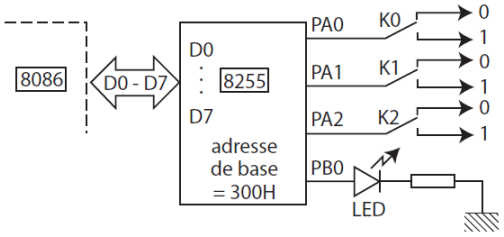
- A2 = 0 et A1 = 0 : adresse du port A = adresse de base + 0 = 300H ;
- A2 = 0 et A1 = 1 : adresse du port B = adresse de base + 2 = 302H ;
- A2 = 1 et A1 = 0 : adresse du port C = adresse de base + 4 = 304H ;
- A2 = 1 et A1 = 1 : adresse du registre de commande = adresse de base + 6 = 306H.

Remarque : le décodage d'adresses étant incomplet, le 8255 apparaît dans plusieurs plages d'adresses selon les valeurs des bits d'adresses non décodés (A7 - A13 et A12 - A15). Dans cet exemple, l'adresse de base 300H correspond à la première adresse possible (bits d'adresses non décodés tous égaux à 0).

Remarque : si on veut que le 8255 possède des adresses consécutives (par exemple 300H, 301H, 302H et 303H), on peut utiliser le schéma suivant qui exploite tout le bus de données (D0 - D15) du 8086 :



Exemple de programmation : soit le montage suivant :



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

Programme :

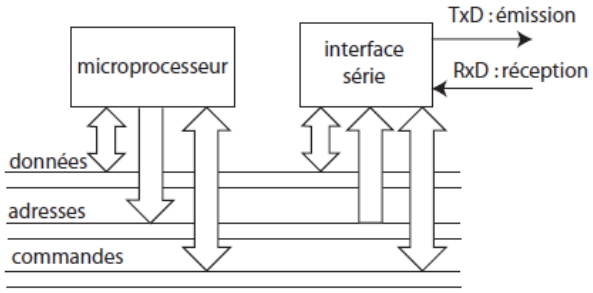
```
portA       equ   300H          ; adresses des registres du 8255
portB       equ   302H
portC       equ   304H
controle    equ   306H

            mov   dx,controle   ; initialisation du port A en entrée
            mov   al,90H        ; et du port B en sortie (mode 0) :
            out   dx,al         ; controle = 10010000B = 90H

boucle :    mov   dx,portA      ; lecture du port A
            in    al,dx
            and   al,00000111B  ; masquage PA0, PA1 et PA2
            cmp   al,00000101B  ; test PA0 = 1, PA1 = 0 et PA2 = 1
            jne   faux          ; non -> aller au label ''faux'' ...
            mov   al,00000001B  ; oui -> mettre PB0 à 1
            jmp   suite         ; et continuer au label ''suite''
faux :      mov   al,00000000B  ; ... mettre PB0 à 0
suite :     mov   dx,portB      ; écriture du port B
            out   dx,al
            jmp   boucle        ; retourner lire le port A
```

V. L'interface série 8250 :

Une interface série permet d'échanger des données entre le microprocesseur et un périphérique bit par bit.

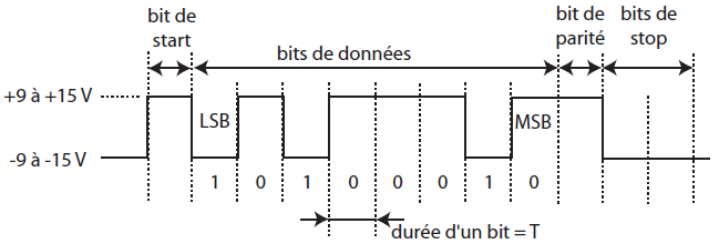


Avantage : diminution du nombre de connexions (1 fil pour l'émission, 1 fil pour la réception).
Inconvénient : vitesse de transmission plus faible que pour une interface parallèle.
Il existe deux types de transmissions séries :

- Asynchrone : chaque octet peut être émis ou reçu sans durée déterminée entre un octet et le suivant ;
- Synchrone : les octets successifs sont transmis par blocs séparés par des octets de synchronisation.

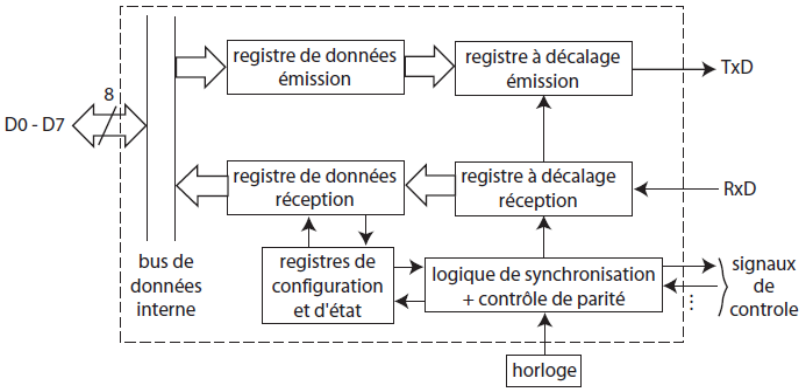
La transmission asynchrone la plus utilisée est celle qui est définie par la norme RS232.

Exemple : transmission du caractère 'E' (code ASCII 45H = 01000101B) sous forme série selon la norme RS232 :



- L'état 1 correspond à une tension négative comprise entre −9 et −15 V, l'état 0 à une tension positive comprise entre +9 et +15 V. Au repos, la ligne est à l'état 1 (tension négative) ;
- Le bit de start marque le début de la transmission du caractère ;
- Les bits de données sont transmis l'un après l'autre en commençant par le bit de poids faible. Ils peuvent être au nombre de 5, 6, 7 ou 8. Chaque bit est maintenu sur la ligne pendant une durée déterminée T. L'inverse de cette durée définit la fréquence de bit = nombre de bits par secondes = vitesse de transmission. Les vitesses normalisées sont : 50, 75, 110, 134.5, 150, 300, 600, 1200, 2400, 4800, 9600 bits/s ;
- Le bit de parité (facultatif) est un bit supplémentaire dont la valeur dépend du nombre de bits de données égaux à 1. Il est utilisé pour la détection d'erreurs de transmission ;
- Les bits de stop (1, 1.5 ou 2) marquent la fin de la transmission du caractère.

V.1. Principe d'une interface série :



Un circuit intégré d'interface série asynchrone s'appelle un UART : Universal Asynchronous Receiver Transmitter) ; une interface série synchrone/asynchrone est un USART.

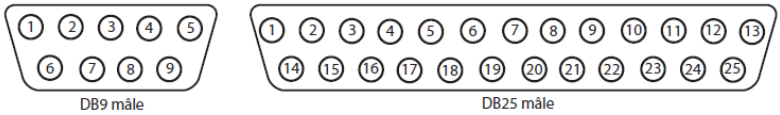
V.2. Exemples d'interfaces séries :

- 8251 (Intel) ;
- 8250 (National Semiconductor) ;
- 6850 (Motorola).

Connexion de deux équipements par une liaison série RS232 : les équipements qui peuvent être connectés à travers une liaison série RS232 sont de deux types :

- Les équipements terminaux de données (DTE : Data Terminal Equipment) qui génèrent les données à transmettre, exemple : un ordinateur ;
- Les équipements de communication de données (DCE : Data Communication Equipment) qui transmettent les données sur les lignes de communication, exemple : un modem.

Pour connecter ces équipements, on utilise des connecteurs normalisés DB9 ou DB25 :

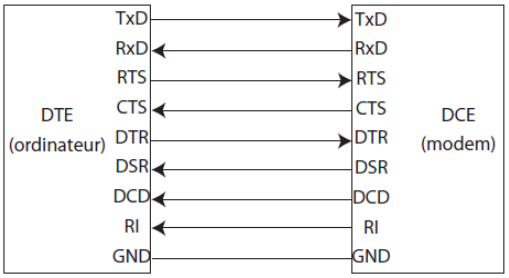


Différents signaux sont transportés par ces connecteurs :

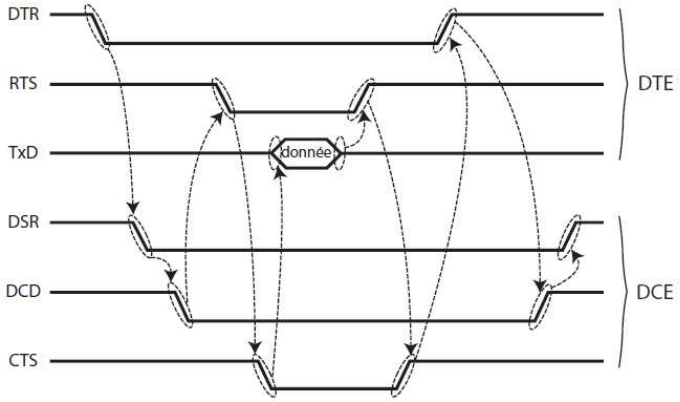
signal	n° broche DB9	n° broche DB25	description	sens	
				DTE	DCE
TxD	3	2	Transmit Data	sortie	entrée
RxD	2	3	Receive Data	entrée	sortie
RTS	7	4	Request To Send	sortie	entrée
CTS	8	5	Clear To Send	entrée	sortie
DTR	4	20	Data Terminal Ready	sortie	entrée
DSR	6	6	Data Set Ready	entrée	sortie
DCD	1	8	Data Carrier Detect	entrée	sortie
RI	9	22	Ring Indicator	entrée	sortie
GND	5	7	Ground	—	—

Seuls les 2 signaux TxD et RxD servent à transmettre les données. Les autres signaux sont des signaux de contrôle de l'échange de données.

V.3. Connexion entre DTE et DCE :



Dialogue entre DTE et DCE :

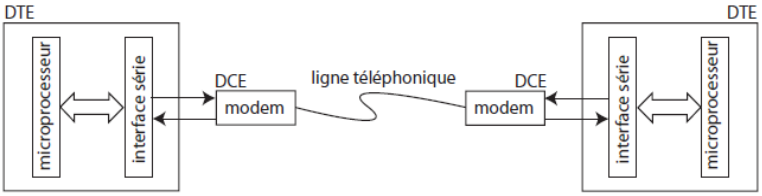


(les signaux de contrôle sont actifs à l'état bas = tension positive)

- Quand le DTE veut transmettre des données, il active le signal DTR. Si le DCE est prêt à recevoir les données, il active le signal DSR puis le signal DCD : la communication peut débuter ;
- Lorsque le DTE a une donnée à émettre, il active le signal RTS. Si le DCE peut recevoir la donnée, il active CTS : le DTE envoie la donnée sur la ligne TxD;
- Si le DCE veut demander une pause dans la transmission, il désactive CTS : le DTE arrête la transmission jusqu'à ce que CTS soit réactivé. C'est un contrôle matériel du flux de données;
- Lorsque la transmission est terminée, les signaux RTS, CTS, DTR, DCD et DSR sont successivement désactivés.

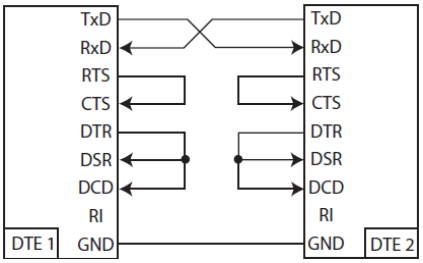
V.4. Applications des liaisons séries :

- Transmission de données à travers une ligne téléphonique :



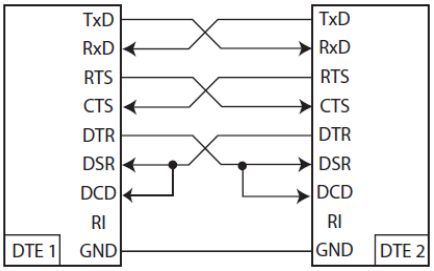
Le modem transforme les signaux numériques produits par l'interface série en signaux analogiques acceptés par la ligne téléphonique et inversement (modulations numériques FSK, PSK, ...)

- Liaison série directe entre deux DTE :
- Liaison simple à 3 fils : rebouclage (strapping) des signaux de contrôle :



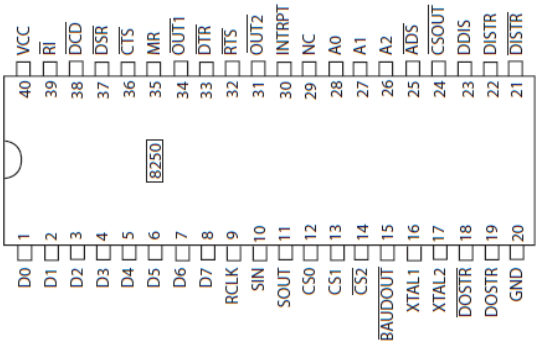
Ce câblage ne permet pas le contrôle matériel du flux entre les deux DTE.

- Liaison complète : câble Null Modem :



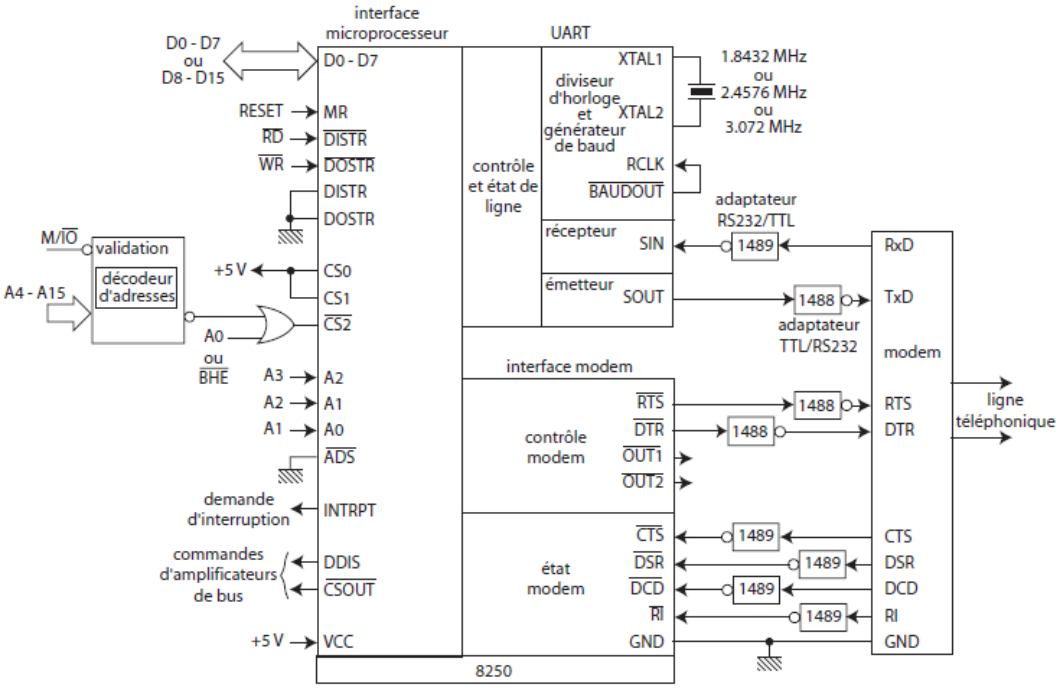
Ce câblage simule la présence d'un modem (DCE) en croisant les signaux de contrôle et permet le contrôle matériel du flux.

V.5. Mise en oeuvre d'une interface série, l'UART 8250 :



Brochage du 8250 :

Schéma fonctionnel :



Accès aux registres du 8250 : le 8250 possède 11 registres. Comme il n'y a que 3 bits d'adresses (A0, A1 et A2), plusieurs registres doivent se partager la même adresse :

DLAB	A2	A1	A0	registre
0	0	0	0	RBR : Receiver Buffer Register, registre de réception (accessible seulement en lecture)
0	0	0	0	THR : Transmitter Holding Register, registre d'émission (accessible seulement en écriture)
1	0	0	0	DLL : Divisor Latch LSB, octet de poids faible du diviseur d'horloge
1	0	0	1	DLM : Divisor Latch MSB, octet de poids fort du diviseur d'horloge
0	0	0	1	IER : Interrupt Enable Register, registre d'autorisation des interruptions
X	0	1	0	IIR : Interrupt Identification Register, registre d'identification des interruptions
X	0	1	1	LCR : Line Control Register, registre de contrôle de ligne
X	1	0	0	MCR : Modem Control Register, registre de contrôle modem
X	1	0	1	LSR : Line Status Register, registre d'état de la ligne
X	1	1	0	MSR : Modem Status Register, registre d'état du modem
X	1	1	1	SCR : Scratch Register, registre à usage général

En fonction de l'état de DLAB (Divisor Latch Access Bit = bit de poids fort du registre LCR), on a accès soit au registre d'émission/réception, soit au diviseur d'horloge, soit au masque d'interruptions.

Structure des registres :

- **Line Control Register (LCR) :**
  bits 0 et 1 : longueur du mot transmis,

bit 1	bit 0	
0	0	→ 5 bits
0	1	→ 6 bits
1	0	→ 7 bits
1	1	→ 8 bits

bit 2 : nombre de bits de stop, 0 → 1 bit de stop,
1 → 1.5 bits de stop si 5 bits sont transmis, 2 bits de stop sinon ;
bit 3 : autorisation de parité, 0 → pas de parité,
1 → parité générée et vérifiée ;
bit 4 : sélection de parité, 0 → parité impaire,
1 → parité paire;
bit 5 : forçage de parité, 0 → parité non forcée
1 → parité fixe;
bit 6 : contrôle de l'état de la ligne TxD, 0 → ligne en fonctionnement normal,
1 → forçage de TxD à l'état 0 (break) ;
bit 7 : DLAB (Divisor Latch Access bit), 0 → accès aux registres d'émission, de réception et IER,
1 → accès au diviseur d'horloge.

- **Line Status Register (LSR) :**

bit 0 : 1 → donnée reçue ;
bit 1 : 1 → erreur d'écrasement de caractère ;
bit 2 : 1 → erreur de parité ;
bit 3 : 1 → erreur de cadrage (bit de stop non valide) ;
bit 4 : 1 → détection d'un état logique 0 sur RxD pendant une durée supérieure à la durée d'un mot ;
bit 5 : 1 → registre de transmission vide ;
bit 6 : 1 → registre à décalage vide ;
bit 7 : non utilisé, toujours à 0.

- **Modem Control Register (MCR) :**

bit 0 : $\overline{DTR}$
bit 1 : $\overline{RTS}$
bit 2 : $\overline{OUT1}$
bit 3 : $\overline{OUT2}$
} activation (mise à 0) des lignes correspondantes en mettant à 1 ces bits ;
bit 4 : 1 → fonctionnement en boucle : TxD connectée à RxD (mode test) ;
bit 5
bit 6
bit 7
} : inutilisés, toujours à 0.

- **Modem Status Register (MSR) :**

bit 0 : 1 → changement de CTS depuis la dernière lecture : delta CTS ;
bit 1 : 1 → delta DSR ;
bit 2 : 1 → delta RI (uniquement front montant sur $\overline{RI}$) ;
bit 3 : 1 → delta DCD ;
bit 4 : $\overline{CTS}$
bit 5 : $\overline{DSR}$
bit 6 : $\overline{RI}$
bit 7 : $\overline{DCD}$
} ces bits indiquent l'état des lignes correspondantes.

- **Diviseur d'horloge (DLM,DLL) :** la vitesse de transmission est fixée par la valeur du diviseur d'horloge :

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

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

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

- Receiver Buffer Register (RBR) : contient la donnée reçue.
- Transmitter Holding Register (THR) : contient la donnée à transmettre.
- Interrupt Identification Register (IIR) :

  bit 0 : 0 → interruption en cours,
  1 → pas d'interruption en cours ;

  bits 1 et 2 : source d'interruption,

bit 2	bit 1	
1	1	→ erreur
1	0	→ donnée reçue
0	1	→ registre d'émission vide
0	0	→ changement d'état modem

(ordre de priorité décroissant) ;

  bit 3, bit 4, bit 5, bit 6, bit 7 : inutilisés, toujours à 0.

➢ **Interrupt Enable Register (IER) : autorisation des interruptions**

  bit 0 : 1 → donnée reçue ;
  bit 1 : 1 → registre d'émission vide ;
  bit 2 : 1 → erreur ;
  bit 3 : 1 → changement d'état modem ;
  bit 4, bit 5, bit 6, bit 7 : inutilisés, toujours à 0.

➢ **Scratch Register (SCR) : registre à usage général pouvant contenir des données temporaires.**

Exemple de programmation : soit un UART 8250 dont le bus de données est connecté sur la partie faible du bus de données du microprocesseur 8086. L'adresse de base du 8250 est fixée à la valeur 200H par un décodeur d'adresses. La fréquence d'horloge du 8250 est de 1.8432 MHz. On veut :

- Ecrire une procédure init qui initialise le 8250 avec les paramètres suivants : 2400 bits/s, 8 bits par caractère, parité paire, 1 bit de stop (2400, 8, P, 1) ;
- Ecrire une procédure envoi qui émet un message contenu dans la zone de données msg. L'émission s'arrête lorsque le caractère EOT (End Of Text, code ASCII = 03H) est rencontré ;
- Ecrire une procédure réception qui reçoit une ligne de 80 caractères et la range dans une zone de données appelée ligne. En cas d'erreur de réception, envoyer le caractère NAK (No Acknowledge, code ASCII = 15H) sinon envoyer le caractère ACK (Acknowledge, code ASCII = 06H).

Programme :

```
RBR      equ   200H              ; adresses des registres du 8250
THR      equ   200H
DLL      equ   200H
DLM      equ   202H
IER      equ   202H
IIR      equ   204H
LCR      equ   206H
MCR      equ   208H
LSR      equ   20AH
MSR      equ   20CH
SCR      equ   20EH
EOT      equ   03H               ; caractère End Of Text
ACK      equ   06H               ; caractère Acknowledge
NAK      equ   15H               ; caractère No Acknowledge
LIGNE    db    80 dup(?)         ; zone de rangement des caractères reçus
MSG      db    'Test 8250', EOT  ; message à envoyer

INIT     PROC  NEAR              ; procédure d'initialisation du 8250
         mov   dx,LCR            ; DLAB = 1 pour accéder au diviseur
         mov   al,80H            ; d'horloge
         out   dx,al
         mov   dx,DLL            ; vitesse de transmission = 2400 bit/s
         mov   al,48             ; => DLL = 48 ...
         out   dx,al
         mov   dx,DLM            ; ... et DLM = 0
         mov   al,0
         out   dx,al
         mov   dx,LCR            ; DLAB = 0 , 8 bits de données,
         mov   al,00011011B      ; parité paire, 1 bit de stop
         out   dx,al
         ret
INIT     ENDP

ENVOI_CARACTERE PROC NEAR        ; procédure d'émission du contenu de AH
         mov   dx,LSR            ; lecture du registre d'état de la ligne
attente_envoi :                  ; attente registre de transmission vide
         in    al,dx
         and   al,20H            ; masquage bit 5 de LSR
         jz    attente_envoi     ; si bit 5 de LSR = 0 => attente ...
         mov   dx,THR            ; ... sinon envoyer le caractère
         mov   al,ah             ; contenu dans le registre AH
         out   dx,al
         ret
ENVOI_CARACTERE ENDP
```

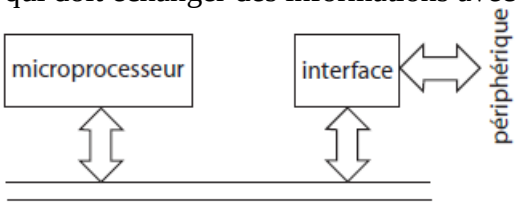
```
ENVOI    PROC  NEAR                ; procédure d'émission du message
         mov   si,offset MSG       ; pointer vers le début du message
boucle : mov   ah,[si]             ; AH <- caractère à envoyer
         cmp   AH,EOT              ; fin du message ?
         jz    fin_envoi           ; oui => fin procédure
         call  ENVOI_CARACTERE     ; non => envoyer caractère ...
         inc   si                  ; ... et passer au caractère suivant
         jmp   boucle
fin_envoi :
         ret
ENVOI    ENDP

RECEPTION PROC NEAR                ; procédure de réception d'une ligne
         mov   di,offset LIGNE     ; pointer vers début zone de réception
         mov   cx,80               ; compteur de caractères reçus
attente_reception :
         mov   dx,LSR              ; lecture du registre d'état de la ligne
         in    al,dx
         test  al,01H              ; test de l'état du bit 0 de LSR
         jz    attente_reception   ; pas de caractère reçu => attente
         test  al,00001110B        ; sinon test erreurs : bits 1,2,3 de LSR
         jz    suite               ; pas d'erreurs => continuer
         mov   ah,NAK              ; erreurs => envoyer NAK ...
         call  ENVOI_CARACTERE
         jmp   attente_reception   ; ... et retourner attendre un caractère
suite :  mov   dx,RBR              ; lire caractère reçu ...
         in    al,dx
         mov   [di],al             ; ... et le ranger dans LIGNE
         mov   ah,ACK              ; puis envoyer ACK
         call  ENVOI_CARACTERE
         dec   cx                  ; décrémenter compteur de caractères
         jz    fin_reception       ; si compteur = 0 => fin réception
         inc   di                  ; sinon incrémenter DI
         jmp   attente_reception   ; et aller attendre caractère suivant
fin_reception :
         ret
RECEPTION ENDP
```

Chapitre V : Les interruptions

I. Définition d'une interruption :

Soit un microprocesseur qui doit échanger des informations avec un périphérique :



Il y a deux méthodes possibles pour recevoir les données provenant des périphériques :

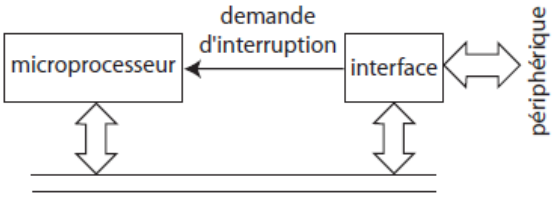
• **Scrutation périodique (ou polling)** : le programme principal contient des instructions qui lisent cycliquement l'état des ports d'E/S.

Avantage : facilité de programmation.

Inconvénients :

- perte de temps s'il y a de nombreux périphériques à interroger ;
- de nouvelles données ne sont pas toujours présentes ;
- des données peuvent être perdues si elles changent rapidement.

• **Interruption** : lorsqu'une donnée apparaît sur un périphérique, le circuit d'E/S le signale au microprocesseur pour que celui-ci effectue la lecture de la donnée : c'est une demande d'interruption (IRQ : Interrupt Request) :



Avantage : le microprocesseur effectue une lecture des ports d'E/S seulement lorsqu'une donnée est disponible, ce qui permet de gagner du temps et d'éviter de perdre des données.

Exemples de périphériques utilisant les interruptions :

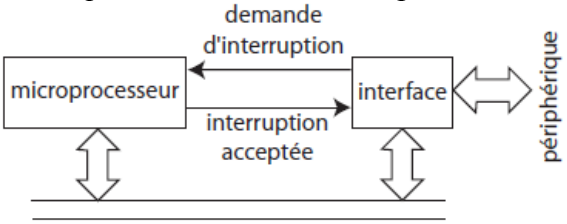
- Clavier : demande d'interruption lorsqu'une touche est enfoncée ;
- Port série : demande d'interruption lors de l'arrivée d'un caractère sur la ligne de transmission.

Remarque : les interruptions peuvent être générées par le microprocesseur lui-même en cas de problèmes tels qu'une erreur d'alimentation, une division par zéro ou un circuit mémoire défectueux (erreurs fatales). Dans ce cas, la demande d'interruption conduit à l'arrêt du microprocesseur.

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

A la suite d'une demande d'interruption par un périphérique :

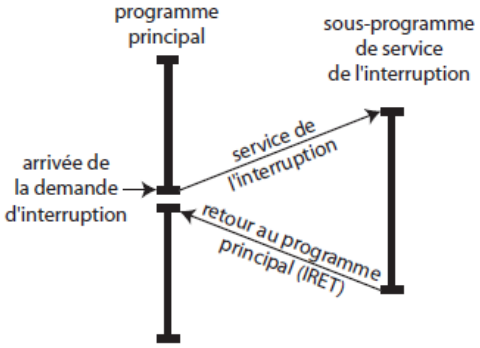
- Le microprocesseur termine l'exécution de l'instruction en cours ;
- Il range le contenu des principaux registres sur la pile de sauvegarde : pointeur d'instruction, flags, ...
- Il émet un accusé de réception de demande d'interruption (Interrupt Acknowledge) indiquant au circuit d'E/S que la demande d'interruption est acceptée :



Remarque : le microprocesseur peut refuser la demande d'interruption : celle-ci est alors masquée. Le masquage d'une interruption se fait généralement en positionnant un flag dans

le registre des indicateurs d'état. Il existe cependant des interruptions non masquables qui sont toujours prises en compte par le microprocesseur.

- Il abandonne l'exécution du programme en cours et va exécuter un sous-programme de service de l'interruption (ISR : Interrupt Service Routine) ;
- Après l'exécution de l'ISR, les registres sont restaurés à partir de la pile et le microprocesseur reprend l'exécution du programme qu'il avait abandonné :

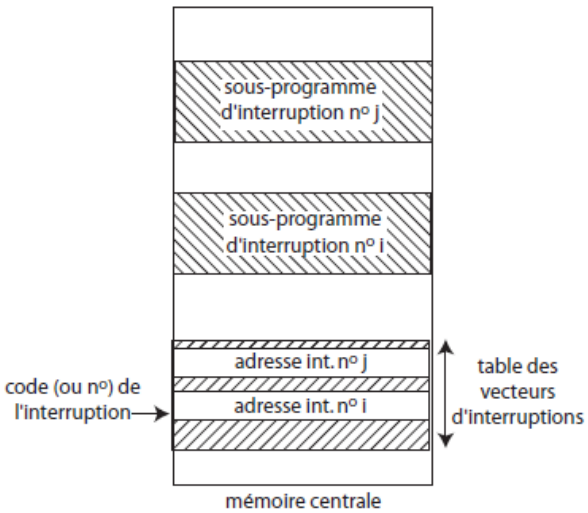


Remarque : la dernière instruction d'un sous-programme de service d'interruption doit être l'instruction IRET : retour d'interruption.

Si plusieurs interruptions peuvent se produire en même temps, on doit leur affecter une priorité pour que le microprocesseur sache dans quel ordre il doit servir chacune d'entre elle.

III. Adresses des sous-programmes d'interruptions :

Lorsqu'une interruption survient, le microprocesseur a besoin de connaître l'adresse du sous-programme de service de cette interruption. Pour cela, la source d'interruption place sur le bus de données un code numérique indiquant la nature de l'interruption. Le microprocesseur utilise ce code pour rechercher dans une table en mémoire centrale l'adresse du sous-programme d'interruption à exécuter. Chaque élément de cette table s'appelle un vecteur d'interruption :



Lorsque les adresses des sous-programmes d'interruptions sont gérées de cette manière, on dit que les interruptions sont vectorisées.

Avantage de la vectorisation des interruptions : l'emplacement d'une ISR peut être n'importe où dans la mémoire, il suffit de spécifier le vecteur d'interruption correspondant.

IV. Les interruptions du 8086 :

Le microprocesseur 8086 peut gérer jusqu'à 256 interruptions. Chaque interruption reçoit un numéro (compris entre 0 et 255) appelé type de l'interruption.

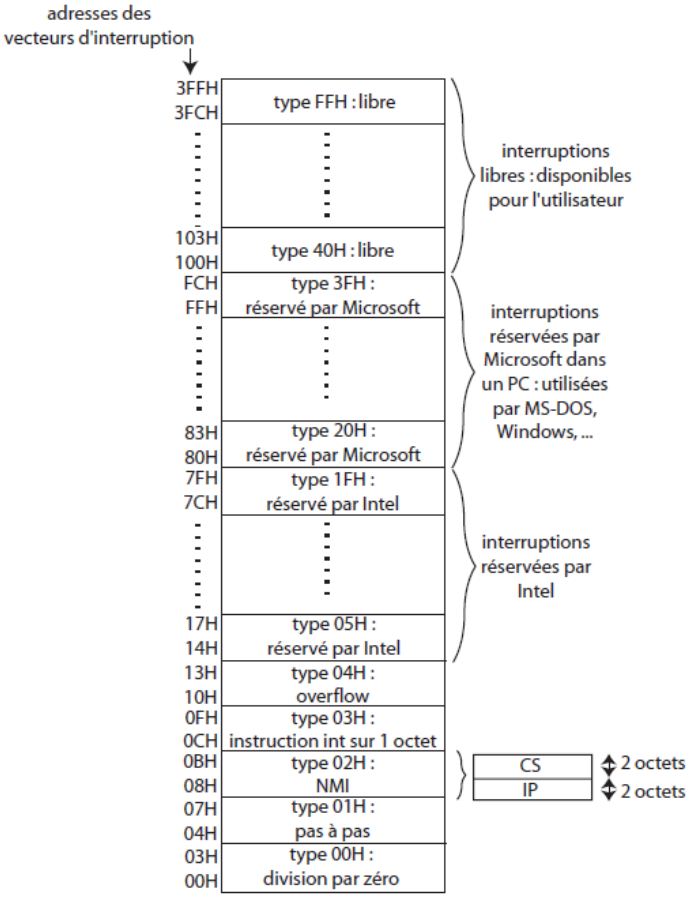
Trois sortes d'interruptions sont reconnues par le 8086 :

- Interruptions matérielles produites par l'activation des lignes INTR et NMI du microprocesseur ;
- Interruptions logicielles produites par l'instruction INT n, où n est le type de l'interruption ;

- Interruptions processeur générées par le microprocesseur en cas de dépassement, de division par zéro ou lors de l'exécution pas à pas d'un programme.

Les interruptions du 8086 sont vectorisées. La table des vecteurs d'interruptions doit obligatoirement commencer à l'adresse physique 00000H dans la mémoire centrale.

Chaque vecteur d'interruption est constitué de 4 octets représentant une adresse logique du type CS : IP.



Remarque : correspondance entre le type de l'interruption et l'adresse du vecteur correspondant :

adresse vecteur d'interruption = 4 × type de l'interruption

Exemple : interruption 20H, adresse du vecteur = 4 × 20H = 80H.

La table des vecteurs d'interruptions est chargée par le programme principal (carte à microprocesseur) ou par le système d'exploitation (ordinateur) au démarrage du système.

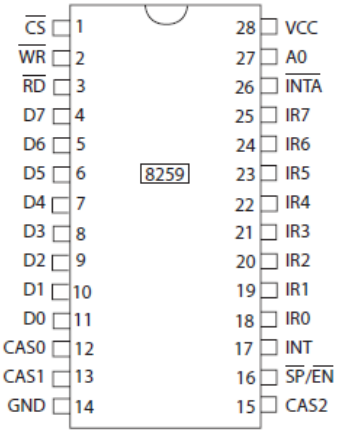
Elle peut être modifiée en cours de fonctionnement (détournement des vecteurs d'interruptions).

V. Le contrôleur programmable d'interruptions 8259 :

Le microprocesseur 8086 ne dispose que de deux lignes de demandes d'interruptions matérielles (NMI et INTR). Pour pouvoir connecter plusieurs périphériques utilisant des interruptions, on peut utiliser le contrôleur programmable d'interruptions 8259 dont le rôle est de :

- Recevoir des demandes d'interruptions des périphériques ;
- Résoudre les priorités des interruptions ;
- Générer le signal INTR pour le 8086 ;
- Emettre le numéro de l'interruption sur le bus de données.

Un 8259 peut gérer jusqu'à 8 demandes d'interruptions matérielles.



Brochage du 8259

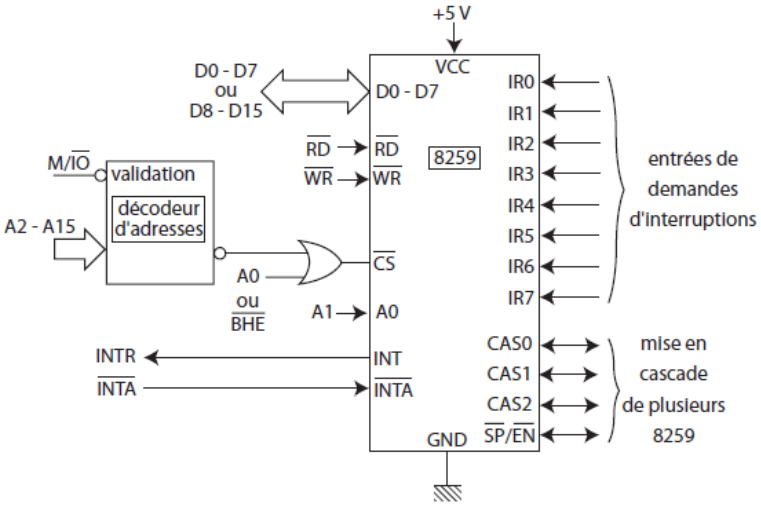
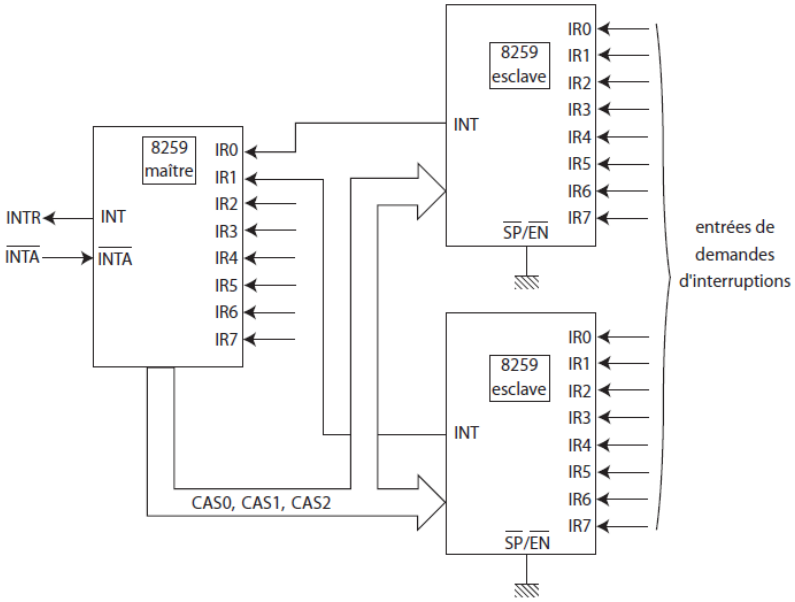


Schéma fonctionnel :

Remarque : si le nombre de demandes d'interruptions est supérieur à 8, on peut placer plusieurs 8259 en cascade :



Chapitre VI : Le Microcontrôleur PIC16F84

I. Introduction :

Un microcontrôleur est une unité de traitement de l'information de type microprocesseur à laquelle nous ajoutons des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants annexes. Un microcontrôleur peut donc fonctionner de façon autonome après programmation. Un microcontrôleur est réalisé en technologie CMOS, en le regardant pour la première fois, il fait davantage penser à un banal circuit intégré logique TTL ou MOS, plutôt qu'à un microcontrôleur.

Un PIC (Programmable Interface Controler) est un microcontrôleur fabriqué par la Société américaine Arizona MICROCHIP Technology. Les PIC sont dérivés du PIC1650 développé à l'origine par la division microélectronique de General Instrument.

Un PIC est fourni en boîtier DIL (Dual In Line) de 2x9 pattes. En dépit de sa petite taille, ils sont caractérisés par une architecture interne qui lui confère une souplesse et une vitesse incomparables. Ses caractéristiques principales sont :

- Séparation des mémoires de programme et de données (architecture Harvard) : On obtient ainsi une meilleure bande passante et des instructions et des données pas forcément codées sur le même nombre de bits.
- Communication avec l'extérieur seulement par des ports : il ne possède pas de bus d'adresses, de bus de données et de bus de contrôle comme la plupart des microprocesseurs.
- Utilisation d'un jeu d'instructions réduit, d'où le nom de son architecture : RISC (Reduced Instructions Set Construction). Les instructions sont ainsi codées sur un nombre réduit de bits, ce qui accélère l'exécution (1 cycle machine par instruction sauf pour les sauts qui requirent 2 cycles). En revanche, leur nombre limité oblige à se restreindre à des instructions basiques, contrairement aux systèmes d'architecture CISC (Complex Instructions Set Construction) qui proposent plus d'instructions donc codées sur plus de bits mais réalisant des traitements plus complexes.

Il existe trois familles de PIC :

- Base-Line : Les instructions sont codées sur 12 bits ;
- Mid-Line : Les instructions sont codées sur 14 bits ;
- High-End : Les instructions sont codées sur 16 bits.

Par rapport à des systèmes électroniques à base de microprocesseurs et autres composants séparés, les microcontrôleurs permettent de diminuer la taille, la consommation électrique et le coût des produits. Ils sont fréquemment utilisés dans les systèmes embarqués, comme les contrôleurs des moteurs automobiles, les télécommandes, les appareils de bureau, l'électroménager, les jouets, la téléphonie mobile, etc.

II. Le PIC16F84 de Microchip

Le PIC16F84 est un microcontrôleur 8 bits à 18 pattes. Le numéro 16 signifie qu'il fait partie de la famille "MID-RANGE". La lettre F indique que la mémoire programme de ce PIC est de type "Flash". Les deux derniers chiffres permettent d'identifier précisément le PIC, ici c'est un PIC de type 84. La référence 16F84 peut avoir un suffixe du type "-XX" dans lequel XX représente la fréquence d'horloge maximal que le PIC peut recevoir.

Remarque : La lettre L indique que le PIC peut fonctionner avec une plage de tension beaucoup plus tolérante. La lettre C indique que la mémoire programme est une EPROM ou plus rarement une EEPROM et la lettre CR indique une mémoire de type ROM. Notez à ce niveau que seule une mémoire FLASH ou EEPROM est susceptible d'être effacée.

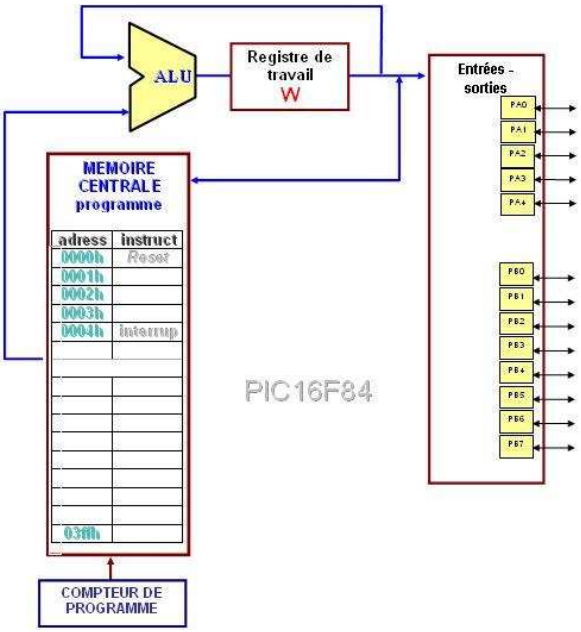
Device	Program Memory (words)	Data RAM (bytes)	Data EEPROM (bytes)	Max. Freq (MHz)
PIC16F83	512 Flash	36	64	10
PIC16F84	1 K Flash	68	64	10
PIC16CR83	512 ROM	36	64	10
PIC16CR84	1 K ROM	68	64	10

Tableau n°1: Liste des composants présentés dans la documentation technique n° DS30430C du PIC.

II.1. Détails des principales caractéristiques du PIC16F84

Les principales caractéristiques d'un PIC16F84 sont :

- 35 instructions (composant RISC) ;
- 2Ko de mémoire Flash pour le programme (RAM de 1019 mots de 14 bits pour les instructions), cette mémoire allant de l'adresse 005 à l'adresse 3FF ;
- 68 octets de RAM (Données sur 8 bits) allant de l'adresse 0C à l'adresse 4F) ;
- Une mémoire RAM de 2x12 emplacements réservée aux registres spéciaux
- 64 octets de d'EEprom ;
- 1 compteur/timer de 8 bits ;
- Une horloge interne, avec pré diviseur et chien de garde (Watch dog) ;
- 4 sources d'interruption ;
- 13 entrées/sorties configurables individuellement, réparties en un port de 5 lignes (Port A) et un port de 8 lignes (Port B) ;
- Mode SLEEP ;
- 1 cycle machine par instruction, sauf pour les sauts (2 cycles machine) ;
- Vitesse maximum 10 MHz soit une instruction en 400 ns (1 cycle machine = 4 cycles d'horloge) ;
- 1000 cycles d'effacement/écriture pour la mémoire flash, 10.000.000 pour la mémoire de donnée EEPROM ;
- Vecteur de Reset situé à l'adresse 000 ;
- Bus d'adresses de 13 lignes ;
- Alimentation sous 5 Volts.



En plus des caractéristiques suivantes :

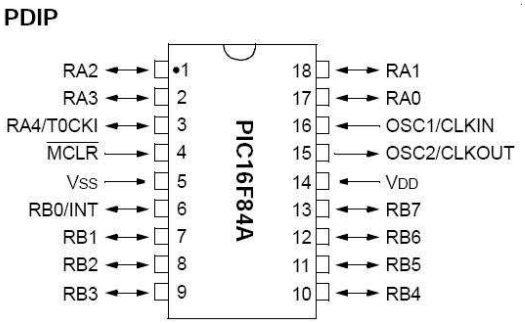
- Architecture interne révolutionnaire lui conférant une extraordinaire rapidité ;
- Possibilité d'être programmé in-circuit, c'est à dire sans qu'il soit nécessaire de le retirer du support de l'application ;

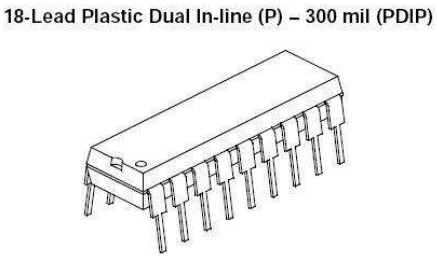
- Un vecteur d'interruption, situé à l'adresse 004 ;
- Présence d'un code de protection permettant d'en empêcher la duplication ;
- Facilité de programmation ;
- Simplicité ;
- Une faible consommation électrique ;
- Faible prix.

II.2. Brochage du PIC16F84

Le PIC16F84 est un microcontrôleur réalisé en technologie CMOS. Les signaux sont compatibles TTL.

- V_{SS} et V_{DD} : broches d'alimentation (3 à 5,5V) ;
- OSC1 et OSC2 : signaux d'horloges, ces broches peuvent recevoir un circuit RC ou un résonateur ;
- CLKIN : peut être connectée à une horloge externe (0 à 4, 10 ou 20 MHz) ;
- MCLR : Reset (Master Clear) ;
- RA0, ... , RA4 : 5 entrées/sorties du port A ;
- RB0, ... , RB7 : 8 entrées/sorties du port B ;
- T0CKI : Entrée d'horloge externe du timer TMR0 ;
- INT : entrée d'interruption externe.

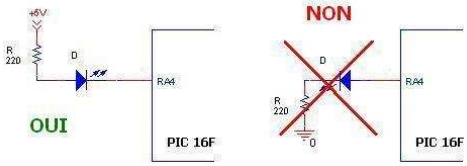




Remarque1: RA = Register A, RB = Register B.

Remarque2: A remarquer que RB0 (pin 6) et RA4 (pin 3), outre qu'à pouvoir servir d'entrées/sorties, selon la façon dont on les programme peuvent respectivement servir l'une comme entrée d'interruption et l'autre comme entrée d'horloge externe pour le pilotage du timer (TMR0).

Remarque3: Le 16F84 possède 13 entrées/sorties (5 dans le port A et 8 dans le port B). Chaque entrée/sortie est configurable individuellement (en entrée ou bien en sortie). On peut par exemple configurer les broches RB0, RA2 et RA3 en entrée et les broches RB1, RB2, RB3, RA0 et RA1 en sortie. Le choix de la configuration des entrées/sorties non utilisées n'a évidemment aucune importance. Notez le cas particulier de la broche RA4 configurée en sortie. Cette broche possède une sortie de type drain ouvert. Cela veut dire qu'elle ne peut pas fournir de courant. Par contre, elle peut en consommer :



II.3. Vitesse des PIC

Tous les PIC Mid-Range ont un jeu de 35 instructions, stockent chaque instruction dans un seul mot de programme, et exécutent chaque instruction (sauf les sauts) en 1 cycle. On atteint donc des très grandes vitesses, et les instructions sont de plus très rapidement assimilées. L'exécution en un seul cycle est typique des composants RISC.

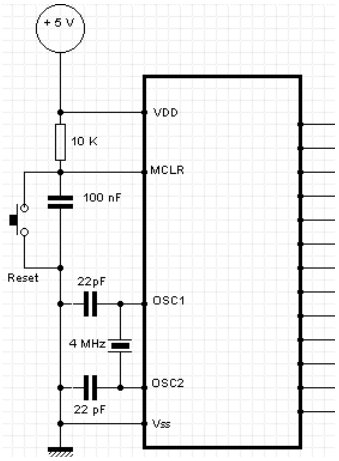
L'horloge fournie au PIC est prédivisée par 4 au niveau de celle-ci. C'est cette base de temps qui donne la durée d'un cycle. Si on utilise par exemple un quartz de 4MHz, on obtient donc 1000000 de cycles/seconde, or, comme le PIC exécute pratiquement 1 instruction par cycle, hormis les sauts, cela vous donne une puissance de l'ordre de 1MIPS (1 Million d'Instructions Par Seconde). Pensez que les PIC peuvent monter à 20MHz. C'est donc une vitesse de traitement plus qu'honorable.

II.4. Fonctionnement d'un PIC16F84

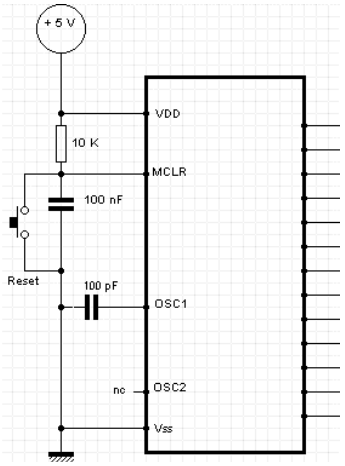
Indépendamment de ce qu'on veut faire des 13 lignes (que l'on définit par lignes d'entrée/sortie) et quelle que soit l'application à laquelle on le destine, un microcontrôleur PIC 16F84, pour pouvoir fonctionner, a nécessairement besoin de :

- Une alimentation de 5 Volts ;
- Un quartz et deux condensateurs (si un pilotage précis par base de temps à quartz est nécessaire), ou une résistance et un condensateur (pour une base de temps de type RC, économique, utilisable dans les cas ne demandant pas une extrême précision de cadencement) ;
- Un condensateur de découplage (pour réduire les transitoires se formant inévitablement dans tout système impulsionnel) ;
- Un bouton poussoir et une résistance, pour la mise en place d'une commande de Reset.

Ces éléments - qu'il convient de considérer comme des invariants devant nécessairement figurer dans tout montage - représentent le cortège obligatoire de tout microcontrôleur PIC 16F84, de la même façon - pourrais-je dire - qu'un transistor demande, pour fonctionner, une résistance de Base et une résistance de Collecteur. Les applications type sont celles des deux pages suivantes :



Pilotage par quartz



Pilotage par oscillateur RC

III. Organisation du PIC16F84

La Figure suivante présente l'architecture générale du circuit. Il est constitué des éléments suivants :

- Un système d'initialisation à la mise sous tension (power-up timer, …) ;
- Un système de génération d'horloge à partir du quartz externe (timing génération) ;
- Une unité arithmétique et logique (ALU) ;
- Une mémoire flash de programme de 1k "mots" de 14 bits ;
- Un compteur de programme (program counter) et une pile (stack) ;
- Un bus spécifique pour le programme (program bus) ;
- Un registre contenant le code de l'instruction à exécuter ;
- Un bus spécifique pour les données (data bus) ;
- Une mémoire RAM contenant :
  - Les SFR ;
  - 68 octets de données ;

- Une mémoire EEPROM de 64 octets de données ;
- 2 ports d'entrées/sorties ;
- Un compteur (timer) ;
- Un chien de garde (watchdog).

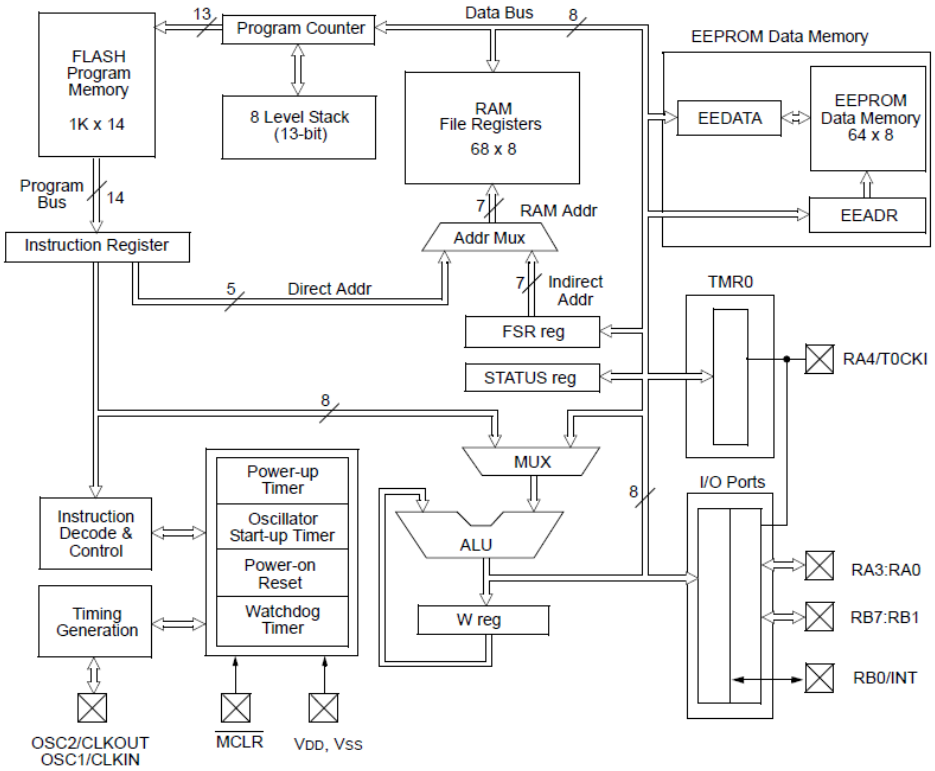


Diagramme blocs d'un PIC16F84

III.1. Organisation de la mémoire du PIC16F84

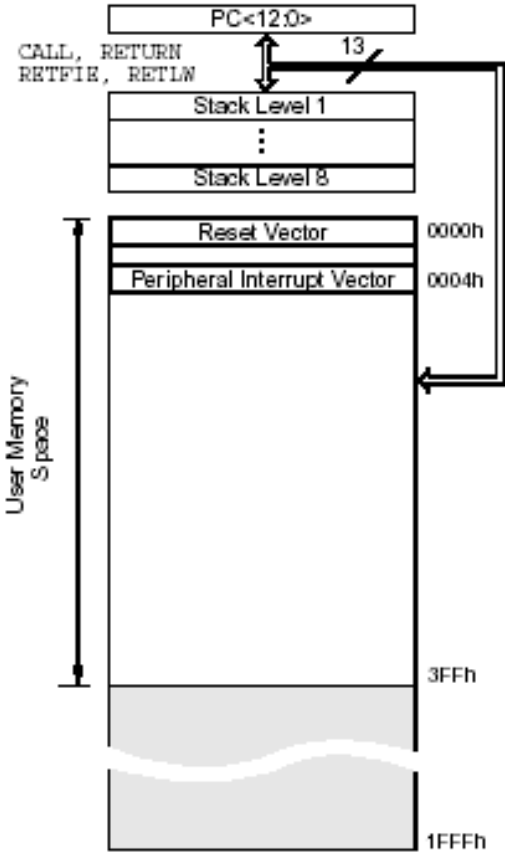
Le PIC contient de la mémoire de programme et de la mémoire de données. La structure Harvard des PICs fournit un accès séparé à chacune. Ainsi, un accès aux deux est possible pendant le même cycle machine. La mémoire du PIC16F84 est répartie en trois espaces, logés sur la même pastille de silicium :

III.1.1. La mémoire programme

La mémoire programme est constituée de 1K mots de 14 bits. C'est une mémoire EEPROM de type flash, de 1 K mots de 14 bits, allant de l'adresse 0000 à l'adresse 03FF. C'est dans cette zone que nous allons écrire notre programme d'où le nom 'mémoire programme'. Ce dernier est téléchargé par une liaison série. Ceci explique aussi pourquoi les fichiers sur un PC font 2Ko (1 Kibioctets). Le plan de cette mémoire est le suivant :

5 adresses réservées au µC (adresses à ne pas utilisée)	000 001 002 003 004	Vecteur de Reset Vecteur d'Interruption
1019 adresses restantes, disponibles pour y loger les instructions de programme	005 . . . . 3FF	Début du programme utilisateur Fin de l'espace mémoire disponible

La Figure suivante montre encore l'organisation de la mémoire programme. Elle contient 1k "mots" de 14 bits dans le cas du PIC 16F84, même si le compteur de programme (PC) de 13 bits peut en adresser 8k. Il faut se méfier des adresses images ! L'adresse 0000h contient le vecteur du reset, l'adresse 0004h l'unique vecteur d'interruption du PIC. La pile contient 8 valeurs. Comme le compteur de programme, elle n'a pas d'adresse dans la plage de mémoire. Ce sont des zones réservées par le système.



Organisation de la mémoire de programme et de la pile

a) L'adresse 000 correspond au vecteur de Reset :
A la mise sous tension, ou à chaque fois que des instructions spécifiques l'obligent, le Program Counter (PC) se rend à cette adresse et c'est là que le système trouve la première instruction à exécuter. C'est une case devant obligatoirement être remplie et contenir l'origine du programme (ORG). Si cette adresse était vide, le microcontrôleur ne ferait rien, car aucun programme ne serait exécuté.
b) l'adresse 004 correspond au vecteur d'interruption :
C'est l'adresse « point de rencontre » définie par le fabricant, à laquelle système et utilisateur se rendent lorsqu'un problème surgit, pour se dire ce qu'il se passe et quel sont les remèdes d'urgence à apporter.

III.1.2. La mémoire de données

Elle se décompose en deux parties de RAM (RAM statique SRAM) et une troisième zone EEPROM. La première contient les SFRs (Special Function Registers) qui permettent de contrôler les opérations sur le circuit. La seconde contient des registres généraux, libres pour l'utilisateur. La dernière contient 64 octets que nous pouvons lire et écrire depuis notre programme. Ces octets sont conservés après une coupure de courant et sont très utiles pour conserver des paramètres semi-permanents. Leur utilisation implique une procédure spéciale, car ce n'est pas de la RAM, mais bien une ROM de type spécial. Il est donc plus rapide de la lire que d'y écrire.

Les instructions orientées octets ou bits contiennent une adresse sur 7 bits pour désigner l'octet avec lequel l'instruction doit travailler. D'après la Figure suivante, l'accès au registre TRISA d'adresse 85h, par exemple, est impossible avec une adresse sur 7 bits. C'est pourquoi le constructeur a défini deux banques. Le bit RP0 (et RP1 pour d'autre PICs) du registre d'état (STATUS) permet de choisir entre les deux. Ainsi, une adresse sur 8 bits est composée de RP0 en poids fort et des 7 bits provenant de l'instruction à exécuter.

- RP1:RP0 = 00 pour la sélection Bank0 ;
- RP1:RP0 = 01 pour la sélection Bank1.

(RP1 est utilisé pour la compatibilité avec d'autres PICS de la famille et restera à 0 sur le 16F84.)

Adr.	Banque0	Banque1	Adr.
00h	Indirect addr.	Indirect addr.	80h
01h	TMR0	OPTION_REG	
02h	PCL	PCL	82h
03h	STATUS	STATUS	
04h	FSR	FSR	84h
05h	PORTA	TRISA	
06h	PORTB	TRISB	86h
07h	--	--	
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	
0Ch - 4Fh	68 cases mémoires	idem banque 0	8Ch – CFH
50h - 7Fh	Inutilisé	inutilisé	D0H – FFH

RAM et Registres du 16F84

a) Registres internes

Les registres SFR (Registres internes à Fonctions Spéciales) sont répartis en deux banques. Cette partie de mémoire est suivie par 68 registres d'usage général pouvant être utilisés comme mémoire simple qui sont identiques sur les deux banques. Ensuite vient une série de registre généraux qui eux dépendent de la banque mémoire choisie et qui ne sont pas implémenté dans le 16F84 original.
Les SFR vont de 00h à 0Bh en Bank0 et de 80h à 8Bh en Bank1. Certains registres sont accessibles en Bank0 et également en Bank1 (ils sont remappés). La fonction de chacun des bits de ces registres est détaillée dans l'annexe A3.

Bank 0

```
00h  INDF     Utilisé en adressage indirect avec FSR, il ne s'agit pas d'un registre physique
01h  TMR0     Timer/Compteur 8 bits
02h  PCL      Poids faible du compteur programme (PC)
03h  STATUS   Registre d'état dont les bits sont : IRP RP1 RP0 TO PD Z DC C
04h  FSR      Pointeur d'adresse en adressage indirect.
05h  PORTA    Port d'Entrée/Sortie A dont les bits sont : x x x RA4/T0CKI RA3 RA2  RA1 RA0
06h  PORTB    Port d'Entrée/Sortie B dont les bits sont: RB7 RB6 RB5 RB4 RB3 RB2 RB1 RB0/INT
07h  Pas utilisé
08h  EEDATA   Registre de données de l' Eeprom
09h  EEADR    Registre d'adresses de l' Eeprom
0Ah  PCLATH   5 bits de poids fort du compteur programme
0Bh  INTCON   Registre des Interruptions dont les bits sont: GIE EEIE T0IE INTE RBIE T0IF INTF RBIF
```

Bank 1

```
80h  INDF     Utilisé en adressage indirect avec FSR, il ne s'agit pas d'un registre physique
81h  OPTION_REG    Registre d'options, bits : RBPU INTEDG T0CS T0SE PSA PS2 PS1 PS0
82h  PCL      Poids faible du compteur programme (PC)
83h  STATUS   Registre d'état dont les bits sont : IRP RP1 RP0 TO PD Z DC C
84h  FSR      Pointeur d'adresse en adressage indirect.
85h  TRISA    Registre de Direction du Port A
86h  TRISB    Registre de Direction du Port B
87h  Pas utilisé
88h  EECON1   Registre de Contrôle n°1 de l' Eeprom : x x x EEIF WRERR WREN WR RD
89h  EECON2   Registre de contrôle n°2 de l' Eeprom (il ne s'agit pas d'un registre physique)
0Ah  PCLATH   5 bits de poids fort du compteur programme
0Bh  INTCON   Registre des ITs dont les bits sont: GIE EEIE T0IE INTE RBIE T0IF INTF RBIF
```

b) La RAM

La RAM GPR (RAM utilisateur d'usage Général ou encore General Purpose Registers) de 68 octets va de 0Ch à 4Fh, soient 68 octets disponibles pour les variables du programme. Cette RAM est remappée en Bank0 et Bank1, il s'agit donc des mêmes emplacements mémoires qui sont accessibles par les deux Banks à deux adresses différentes. Ceci permet d'éviter d'avoir à changer de bank avant d'accéder à une variable ou à une autre.

Lors de la programmation il faut toujours indiquer l'adresse de la zone RAM à partir de laquelle le µC doit commencer à écrire, ainsi que le nombre d'emplacements à réserver pour chaque variable.

Comme par exemple :

```
ORG OC
Compteur RES 3
```

Ce qui revient à dire : réserve trois emplacements à la variable Compteur, dans l'ordre suivant :

Compteur à l'adresse 0C
Compteur+1 à l'adresse 0D
Compteur+2 à l'adresse 0E .

Ainsi, par exemple, pour effacer les données de l'adresse OE, on écrira :

```
CLRF Compteur+2.
```

c) La mémoire EEPROM

La mémoire EEPROM de 64 octets est indirectement mappée dans la zone de données et est donc accessible par un pointeur d'adresse indirect (accessibles en lecture et en écriture par le programme). On peut y sauvegarder des valeurs, qui seront conservées même si l'alimentation est éteinte, et les récupérer lors de la mise sous tension. Cette mémoire de données est une mémoire de type flash de 64 emplacements à 8 bits, allant de l'adresse 00 à l'adresse 3F, auxquels on accède uniquement par l'intermédiaire de quatre registres spéciaux :

- EEADR (EEprom ADRess) pour ce qui concerne les adresses ;
- EEDATA (EEprom DATA) pour ce qui concerne les données ;
- EECON1 et EECON2 (EEprom CONtrol) permettant de définir le mode de fonctionnement de cette mémoire.

III.1.3. La mémoire EEPROM réservées au microcontrôleur

Une autre petite mémoire EEPROM, contenant seulement 8 cases, de l'adresse 2000 à l'adresse 2007, est réservée au microcontrôleur. Les adresses 2000, 2001, 2002 et 2003 correspondent aux emplacements dans lesquels l'utilisateur peut stocker un code d'identification (en n'utilisant que les quatre bits de poids faible de chacun de ces mots à 14 bits). L'adresse 2007 correspond au registre de configuration du microcontrôleur.

III.2. Les Ports d'entrées/sorties

Le PIC 16F84 est doté de deux ports d'entrées/Sorties appelés PortA et PortB.

III.2.1. Le Port A: RA0 … RA4

Il comporte 5 pattes d'entrée/sortie bi-directionnelles, notées RAx avec x={0,1,2,3,4}. Le registre PORTA, d'adresse 05h dans la banque 0, permet d'y accéder en lecture ou en écriture (une copie des lignes RA0..RA4). En effet, lire le PORTA revient à lire l'état des pins alors qu'une écriture place le niveau correspondant sur les pins qui auront été configurées en sorties (dans une séquence interne au PIC de Read-modify-write). Le registre TRISA, d'adresse 85h dans la banque 1, permet de choisir le sens de chaque patte (entrée ou sortie) : un bit à 1 positionne le port en entrée, un bit à 0 positionne le port en sortie.

Les lignes RA0 .. RA3 sont des entrées à niveaux compatibles TTL et des sorties CMOS standards. La ligne RA4 est une entrée à Trigger de Schmitt et une sortie à drain ouvert qui est multiplexée avec l'entrée de Timer TMR0.

Il n'y a aucune instruction permettant d'écrire directement dans le registre TRISA (ou TRISB pour le port B) : on y accède en transitant par le registre de travail W. En programmation, on commence donc par charger l'octet de configuration dans le registre W, puis on copie celui-ci dans TRISA (ou TRISB).

Exemple

```
MOVLW 00000001        ;(en binaire, sinon 01 en hexa)
MOVWF TRISA
```

Le bit 0 du port A est défini comme entrée, tandis que les sept autres lignes sont définies comme sorties.

Exemple de configuration du PORTA

```
BSF   STATUS, RP0     ; Acces Bank1
MOVLW 0xFB            ; TRISA.b2 à 0 pour RA2 en sortie   (% 1111.1011)
MOVWF TRISA
BCF   STATUS, RP0     ; Acces Bank0
…
…
BCF   STATUS, RP0     ; Acces Bank0
BSF   PORTA, RA2      ; Allume la Led connectée à la ligne RA2
```

III.2.1. Le Port B: RB0 … RB7

Il comporte 8 pattes d'entrée/sortie bi-directionnelles, notées RBx avec x={0,1,2,3,4,5,6,7}. Le registre PORTB, d'adresse 06h dans la banque 0, permet d'y accéder en lecture ou en écriture. Le registre TRISB, d'adresse 86h dans la banque 1, permet de choisir le sens de chaque patte (entrée ou sortie) : un bit à 1 positionne le port en entrée, un bit à 0 positionne le port en sortie.

Le câblage interne d'une porte du port B ressemble beaucoup à celui du port A. On peut noter la fonction particulière pilotée par le bit RBPU (OPTION_REG.7) qui permet d'alimenter (RBPU=0) ou non (RBPU=1) les sorties.

Les quatre bits de poids fort (RB7-RB4) peuvent être utilisés pour déclencher une interruption sur changement d'état. RB0 peut aussi servir d'entrée d'interruption externe.

III.2. Les registres spéciaux

Nous avons dit que dans l'espace mémoire RAM que Microchip appelle Register File, une zone est réservée aux registres spéciaux. Le mot registre est utilisé ici pour désigner un emplacement mémoire, tandis que le mot file signifie groupement. Ils ont des noms et des usages spécifiques, et servent à commander le microcontrôleur. Il y en a 16 en tout, et sont si importants qu'ils conditionnent véritablement la programmation.

Ils sont utilisés constamment, et constamment tenus présents dans la tête du programmeur. Celui qui veut écrire ne fût-ce qu'un petit programme de quelques lignes, ne peut pas les ignorer.

A. Détail du registre STATUS (Bank 0 en 03h et Bank 1 en 83h)

Les cinq premiers bits de ce registre (bits 0 à 4) correspondent à des flags que le programmeur peut interroger pour obtenir des informations lui permettant d'écrire correctement la suite des instructions de son programme ; tandis que les bits 5, 6 et 7 (RP0, RP1, RP2), d'après la façon dont on les programme, pourraient sélectionner 8 pages de registres internes (chacune de 128 octets). Les détails des différents bits du registre STATUS sont donnés comme ci-après. Ces bits reflètent le status de l'ALU du PIC, ils permettent également de basculer d'une Bank de registres à une autre.

7	6	5	4	3	2	1	0
		RP0	TO	PD	Z	DC	C

```
b7:     IRP             non utilisé dans le 16F84
b6-5:   RP1 RP0         Sélection de la bank de registres active
                           00:  Bank0
                           01:  Bank1
                           10:  non utilisé sur le 16F84
                           11:  non utilisé sur le 16F84
                           Sert à sélectionner l'une des deux pages de registres (Page 0 ou Page 1).
b4:     TO\             Time Out du watch dog
```

```
                                1:  après un démarrage, CLRWDT ou SLEEP.
                                0:  après un time-out du watch dog
                                Dépassement de délai. Passe à 0 si le timer du Watch-Dog (chien de garde)
                                déborde. Est mis à 1 par les instructions CLWDT et SLEEP, ainsi qu'à la mise sous
                                tension.

b3:     PD\             Power down bit
                                1:  après un démarrage ou après l'instruction CLRWDT
                                0:  après l'exécution de l'instruction SLEEP
                                Mise en veilleuse de l'alimentation, effectuée par l'instruction SLEEP. Passe à 1
                                lorsqu'on utilise l'instruction CLWDT, ou à la mise sous tension.
b2:     Z               bit Zero
                                1:  Résultat de l'opération précédente nul
                                0:  Résultat de l'opération non nul
                                Ce flag passe à 1 si le résultat d'une opération (arithmétique ou logique) est 0.

b1:     DC              Digital carry / borrow\
                                1:  Débordement du 4° bit du résultat de l'opération précédente
                                0:  Pas de débordement
                                Flag fonctionnant comme le bit de Carry, sauf qu'ici la surveillance de la
                                retenue s'exerce non pas sur l'octet entier, mais sur le premier demi-octet. Ce
                                flag se positionne à 1 si une retenue est générée du bit 3 (bit de poids fort du
                                quartet inférieur) vers le bit 0 du quartet supérieur. Il est utile pour corriger
                                le résultat d'opérations effectuées en code BCD.
b0:     C               Carry / borrow\
                                1:  Débordement bit du résultat de l'opération précédente
                                0:  Pas de débordement
                                Flag indiquant si une retenue a eu lieu dans un octet lors d'une addition ou
                                d'une soustraction. Si une retenue a été générée, ce bit passe à 1.
```

B. Détail du registre OPTION_REG (Bank 1 en 81h)

Permet de configurer les résistances de rappel internes du PortB, et aussi l'INT externe, le Timer0 et le prescaler du Timer0 ou du watch-dog

```
b7:     RBPU\           Résistances de rappel des entrées du PortB
                             1: Les résistances sont désactivées
                             0: Les résistances du PortB sont activées
b6:     INTEDG          Sélection du front actif de l'INT externe
                             1: Interruption sur le front montant de RB0/INT
                             0: Interruption sur le front descendant de RB0/INT
b5:     TOCS            Source de l'horloge du Timer0
                             1: Comptage sur la pin RA4/TOCKI
                             0: Comptage sur l'horloge interne CLKOUT
b4:     TOCE            Sélection du front actif pour le comptage sur RA4/TOCKI
                             1: Comptage sur front descendant
                             0: Comptage sur front montant
b3:     PSA             Assignation du Prescaler
                             1: Prescaler assigné au chien de garde WDT
                             0: Assigné au Timer TMR0
b2-b1-b0:               PS2:PS1:PS0     Valeur du Prescaler du TMR0 ou du WDT
```

Valeurs du prescaler

PS2:PS1:PS0	Prescaler TMR0	Prescaler WDT
000	1/2	1/1
001	1/4	1/2
010	1/8	1/4
011	1/16	1/8
100	1/32	1/16
101	1/64	1/32
110	1/128	1/64
111	1/256	1/128

C. Détail du registre INTCON (Bank 0 en 0Bh et Bank 1 en 8Bh)

Bits d'autorisation et Flags d'interruptions

```
b7:     GIE             Autorisation Globale des Interruptions
                             1: Autorise toutes les interruptions
                             0: Interdit toutes les interruptions
b6:     EEIE            Autorisation de l'interruption de fin d'écriture en Eeprom
                             1: Autorise l'interruption de fin d'écriture en Eeprom
                             0: Interdit l'interruption de fin d'écriture en Eeprom
b5:     TOIE            Autorisation de l'interruption de débordement de TMR0
                             1: Autorise l'interruption de TMR0
                             0: Interdit l'interruption de TMR0
b4:     INTE            Autorisation de l'interruption sur RB0/INT
                             1: Autorise l'interruption sur RB0/INT
                             0: Interdit l'interruption sur RB0/INT
b3:     RBIE            Autorisation de l'interruption lors d'un changement d'état sur le PortB
                             1: Autorise l'interruption sur RB7:RB4
                             0: Interdit l'interruption sur RB7:RB4
b2:     TOIF            Flag de débordement de TMR0
                             1: Le TMR0 a débordé  (à effacer par programme)
                             0: Le TMR0 n'a pas débordé
b1:     INTF            Flag d'interruption sur RB0/INT
                             1: Il y a eu une demande d'interruption sur RB0/INT
                             0: Il n'y a pas d'interruption sur RB0/INT
```

```
b0:     RBIF            Flag d'interruption sur le PortB
                             1: Au moins une pin RB7:RB4 a changé d'état (doit être effacé par  programme)
                             0: Il n'y a pas de changement d'état sur RB7:RB4
```

D. EEADR (EEprom ADRess)

Registre dans lequel on écrit l'adresse de la mémoire de données EEPROM (mémoire flash de 64 octets, allant de l'adresse 00 à l'adresse 3F) à laquelle on veut accéder pour y lire ou pour y écrire. Contrairement à l'EEPROM de programme qui - en plus de la tension d'alimentation du microcontrôleur - nécessite une tension externe pour la programmation, cette EEPROM fonctionne avec la seule tension d'alimentation, dans toute sa plage.

E. EECON1 (EEprom CONtrol 1)

C'est un registre de contrôle permettant de définir le mode de fonctionnement de la mémoire de données EEPROM (mémoire flash de 64 octets, allant de l'adresse 00 à l'adresse 3F). En plus, c'est registre à 8 bits, mais dont 5 seulement sont utilisés:

Bit 0 : RD (ReaD) : Normalement à 0. Il se met dans cet état de lui-même. Le programmeur ne peut y écrire que un 1. N'accepte pas d'être programmé à zéro.

Bit 1 : WR (WRite) : Normalement à 0. Il se met dans cet état de lui-même. Le programmeur ne peut écrire que un 1. N'accepte pas d'être programmé à zéro.

Bit 2 : WREN (WRite ENable) : Mis à zéro, interdit toute écriture en mémoire. Mis à 1, autorise une écriture en mémoire.

Bit 3 : WRERR (WRite ERRor) : Flag d'erreur. Normalement à zéro. Passe à 1 pour signaler qu'une erreur s'est produite juste au moment où une écriture était en cours (Celle-ci n'a pu aboutir parce qu'un événement inopiné s'est produit ; par exemple un Reset).

Bit 4 : EEIF (EEprom Interrupt Flag) : Flag d'interruption. Il est automatiquement mis à 1 lorsque la programmation de l'EEPROM de données est terminée. Doit être mis à zéro par programmation.

F. EECON2 (EEprom CONtrol 2)

Registre n'ayant aucune consistance physique, et dont le seul rôle consiste à obliger le programmeur à vérifier les données qu'il envoie dans l'EEPROM.

G. EEDATA (EEprom DATA)

- Pendant une opération de lecture : registre dans lequel est disponible la donnée qu'on est allé chercher à une certaine adresse de la mémoire EEPROM.
- Pendant une opération d'écriture : registre dans lequel on place la donnée qu'on veut y écrire.

H. FSR (File Select Register)

Sert à sélectionner la mémoire de données, pour pouvoir y accéder.

I. INTCON (INTerrupt CONtrol)

C'est le registre qui préside au fonctionnement des interruptions. Dans le 16F84 il y a quatre sources possibles d'interruptions. Chaque fois que l'une d'elles surgit, le microcontrôleur (après avoir noté dans la pile l'adresse de retour) abandonne momentanément (interrompt) le programme qu'il avait en cours d'exécution et saute à l'adresse 004 (adresse prédéfinie par le fabricant, de la même façon que l'adresse 000 a été prédéfinie pour la fonction Reset).

En lisant le contenu de ce registre, on peut déterminer la provenance de la demande d'interruption et aiguiller le programme de manière à y répondre de façon adéquate.

L'interruption peut être commandée soit par un flanc montant, soit par un flanc descendant : cela dépend de la façon dont on a préalablement programmé le bit 6 (INTEDG) du registre OPTION :

1 = l'interruption est générée à l'apparition d'un front montant ;

0 = l'interruption est générée à l'apparition d'un front descendant.

Les quatre sources d'interruption possibles sont :

1) la fin d'une programmation de l'EEPROM de données ;
2) le débordement du timer interne ;
3) une commande externe appliquée sur la pin 6 (RB0/INT) ;
4) un changement d'état sur l'une des pins 10, 11, 12 ou 13 (respectivement RB4, RB5, RB6, RB7).

Dans ce cas, seule une configuration des lignes en entrée peut donner lieu à une éventuelle demande d'interruption.

J. PCL (Program Counter Low)

Il s'agit du compteur qui fournit au programme la partie basse de l'adresse. Dans les microcontrôleurs de Microchip les lignes d'adresses sont réparties en deux bytes : le PCL (fourni par ce registre Program Counter Low), et le PCH (fourni par le registre PCLATH Program Counter LATch High).

Il s'agit d'un compteur dont la tâche est d'adresser la mémoire dans laquelle sont logées les instructions du programme. L'organisation de ce type de compteur, dans les microcontrôleurs PIC, veut que l'adresse soit composée de deux parties : la partie basse (fournie par PCL, sur dix lignes d'adresse) et la partie haute (fournie par PCLATH).

K. PCLATH (Program Counter LATch High)

L'adresse du compteur de programme est obtenue en mettant ensemble la partie basse fournie par PCL (Program Counter Low) et la partie haute fournie par PCLATH.

Contrairement à ce qu'on pourrait penser, ce registre ne fournit pas un nombre complémentaire fixe de bits, mais un nombre de bits variable, en fonction des instructions qui sont traitées.

L. TMR0 (TiMeR zero)

C'est le registre de contrôle de l'horloge interne (timer) du microcontrôleur. Ce timer peut soit fonctionner seul, soit être précédé par un pré diviseur programmable à 8 bits dont la programmation se fait par l'intermédiaire du registre OPTION. Ce registre peut être lu et modifié à tout moment, soit pour connaître sa position courante, soit pour le déclencher à partir d'une valeur déterminée.

Une quelconque opération d'écriture dans ce registre met automatiquement à zéro le comptage en cours dans le pré diviseur. Se rappeler que le timer compte sur 8 bits, et qu'une fois que le comptage est arrivé à FF, celui-ci revient à 00 (ce qui provoque le passage à 1 du bit 2 du registre INTCON appelé T0IF).

En programmation on peut écrire :

1) pour le lire :

```
MOVF TMR0 ,W - (porte la valeur de TMR0 dans W)
```

2) pour lui donner une valeur de départ :

```
MOVLW valeur
MOVWF TMR0
```

3) pour le mettre à zéro :

```
CLRF TMR0
```

Chapitre VI : Le jeu d'instructions d'un PIC16F84

I. Introduction

Les PIC 16F84A, 16F628A, 16F88, 16F876A, 16F886 (famille mid-range) ont le même jeu d'instructions, constitué de seulement 35 instructions (architecture RISC : Reduced Instruction-Set Computer). Une instruction est codée par un mot de 14 bits. La mémoire programme (de type Flash) a une taille de :

- 1792 octets (16F84A)
- 3584 octets (16F628A)
- 7168 octets (16F88)
- 14 336 octets (16F876A - 16F886)

Ce qui permet de stocker un programme de :

- 1024 instructions (16F84A)
- 2048 instructions (16F628A)
- 4096 instructions (16F88)
- 8192 instructions (16F876A - 16F886)

Une instruction nécessite 1 cycle, ou bien 2 cycles dans le cas d'une instruction de branchement (GOTO, CALL ...). Avec une horloge à quartz de 20 MHz, un cycle correspond à 4/(20.106) = 200 nanosecondes. Le microcontrôleur peut donc exécuter jusqu'à 5 millions d'instructions par seconde (5 MIPS) !

Vous pouvez expérimenter ces instructions avec MPLAB et son simulateur, en insérant ces instructions après l'étiquette « start » de votre programme.

Sur le tableau 5.1, nous présentons l'ensemble des 35 instructions du PIC 16F84 codées sur 14 bits. Vous trouverez en colonne de gauche le code binaire de l'instruction, les mnémoniques des instructions se trouvant au centre puis une courte explication de ce que fait l'instruction. Les opérandes peuvent être de plusieurs types:

- f : adresse mémoire de registres (register file address) de 00 à 7F
- W : registre de travail
- d : sélection de destination : d=0 vers W, d=1 vers f
- pp : numéro de PORT entre 1 et 3 sur deux bits
- bbb : adresse de bit dans un registre 8 bits (sur 3 bits)
- k : champ littéral (8, ou 11 bits)
- PC compteur programme

Opcode (binary)	Mnemonic	Description
00 0000 0xx0 0000	NOP	Pas d'opération
00 0000 0110 0011	SLEEP	arrête le processeur
00 0000 0110 0100	CLRWDT	Reset du timer watchdog
00 1000 dfff ffff	MOVF f,d	Recopie de W dans f (adressage direct)
00 0000 1fff ffff	MOVWF	déplacement de W vers f
00 0001 0xxx xxxx	CLRW	Positionne W à 0 (idem à CLR x, W)
00 0001 1fff ffff	CLRF f	Positionne f à 0 (idem à CLR f, F)
00 0010 dfff ffff	SUBWF f, d	Soustrait W de (d = f − W)
00 0011 dfff ffff	DECF f, d	Décrément f (d = f − 1)
00 0100 dfff ffff	IORWF f, d	OU Inclusif W avec F (d = f OR W)
00 0101 dfff ffff	ANDWF f, d	ET entre W et F (d = f AND W)
00 0110 dfff ffff	XORWF f, d	OU Exclusif W avec F (d = f XOR W)
00 0111 dfff ffff	ADDWF f, d	Additionne W avec F (d = f + W)
00 1000 dfff ffff	MOVF f, d	recopie F (d = f)
00 1001 dfff ffff	COMF f, d	Complement f (d = NOT f)

00 1010 dfff ffff	INCF f, d	Incrément f (d = f + 1)
00 1011 dfff ffff	DECFSZ f, d	Decrément f (d = f − 1) et saut si zero
00 1100 dfff ffff	RRF f, d	Rotation droite F (rotation droite avec la retenue)
00 1101 dfff ffff	RLF f, d	Rotation gauche F (rotation gauche avec la retenue)
00 1110 dfff ffff	SWAPF f, d	échange de groupes 4-bit de f (d = lsb:f msb:f)
00 1111 dfff ffff	INCFSZ f, d	Incrément f (d = f + 1) et saut si zero
01 00bb bfff ffff	BCF f, b	RaZ d'un bit de f (Clear bit b of f)
01 01bb bfff ffff	BSF f, b	Mise à 1 d'un bit de f (Set bit b of f)
01 10bb bfff ffff	BTFSC f, b	test de bit de f, saute si zéro (Test bit b of f)
01 11bb bfff ffff	BTFSS f, b	test de bit de f, saute si un (Test bit b of f)
11 01xx kkkk kkkk	RETLW k	Positionne W à k et retour
10 0kkk kkkk kkkk	CALL k	Sauve l'adresse de retour, charge PC avec k
00 0000 0000 1001	RETFIE	retour d'interruption
00 0000 0000 1000	RETURN	retour de sous-programme
10 1kkk kkkk kkkk	GOTO k	saut à l'adresse k (9 bits!)
11 111x kkkk kkkk	ADDLW k	Addition de W et k
11 110x kkkk kkkk	SUBLW k	Soustraction de W et k
11 00xx kkkk kkkk	MOVLW k	Chargement littéral de W (W = k)
11 1000 kkkk kkkk	IORLW k	OU Inclusif littéral avec W (W = k OR W)
11 1001 kkkk kkkk	ANDLW k	ET littéral avec W (W = k AND W)
11 1010 kkkk kkkk	XORLW k	OU exclusif or littéral avec W (W = k XOR W)

Tableau 5.1. Jeu d'instructions 14-bit pour PIC 16F84.

Exemples

```
addlw  k      ;ajoute une constante k à W. Résultat dans W
addwf f,d     ;ajoute W à f. Résultat dans W si d=0 ou dans f si d=1
andlw  k      ;effectue un ET entre une la constante k et W. Résultat dans W
andwf f,d     ;effectue un ET entre W et f. Résultat dans W si d=0 ou dans f si
bcf f,b       d=1
bsf f,b       ;fait passer le bit b de f à 0
btfsc f,b     ;fait passer le bit de f à 1
btfss f,b     ;teste le bit b de f. Incrémente PC si b=0
call  k       ;teste le bit b de f. Incrémente PC si b=1
clrf  f       ;empile PC et affecte PC de l'adresse d'un sous programme
clrw          ;place zero dans f
clrwdt        ;place zero dans W et fait passer STATUS.Z à 1
comf f,d      ;initialise le timer du chien de garde
decf f,d      ;complemente f à 1. Résultat dans W si d=0 ou dans f si d=1
decfsz f,d    ;décrémente f. Résultat dans W si d=0 ou dans f si d=1
goto  k       ;décrémente f. Si f=0, incrémente PC. Résultat dans W ou dans f
incf f,d      ;charge une adresse dans PC
incfsz f,d    ;incrémente F
iorlw  k      ;inrémente f. Résultat dans W si d=0 ou dans f si d=1
iorwf f,d     ;effectue un OU entre une constante et W
movf f,d      ;effectue un OU entre W et f. Résultat dans W si d=0 ou dans f si
rlf f,d       d=1
rrf f,d       ;si d=0, écrit f dans W , sinon réécrit f dans f
sleep         ;effectue une rotation de bits à gauche à travers Carry
sublw  k      ;effectue une rotation de bits à droite à travers Carry
subwf f,d     ;fait passer le PIC en mode veille
 swapf f,d    ;sustrait W de d'une constante k .  ( W := k-W )
 xorlw  k     ;soustrait W de f. Résultat dans W si d=0 ou dans f si d=1
xorwf f,d     ;permute les deux quartets de f. Résultat dans W si d=0 ou dans f
const         si d=1
              ;effectue un OU exclusif entre une constante et W. Resultat dans W
              ;OU exclusif entre W et f. Résultat dans W si d=0 ou dans f si d=1
              ;est obsolète : ne plus l'utiliser
```

II. Les types d'instructions

Pour la programmation du PIC16F84, il existe 4 types d'instructions :

II.1. Les instructions « orientées octet »

Ce sont des instructions qui manipulent les données sous forme d'octets. Elles sont codées de la manière suivante :

- 6 bits pour l'instruction : logique, car comme il y a 35 instructions, il faut 6 bits pour pouvoir les coder toutes ;
- 1 bit de destination(d) pour indiquer si le résultat obtenu doit être conservé dans le registre de travail de l'unité de calcul (W pour Work) ou sauvé dans l'opérande (F pour File) ;
- Reste 7 bits pour encoder l'opérande (File).

Remarque : premier problème, 7 bits ne donnent pas accès à la mémoire RAM totale, donc voici ici l'explication de la division de la RAM en deux banques en utilisant le bit RP0 du registre STATUS.

II.2. Les instructions « orientées bits »

Ce sont des instructions destinées à manipuler directement des bits d'un registre particulier. Elles sont codées de la manière suivante :

- 4 bits pour l'instruction (dans l'espace resté libre par les instructions précédentes) ;
- 3 bits pour indiquer le numéro du bit à manipuler (bit 0 à 7 possible), et de nouveau ;
- 7 bits pour indiquer l'opérande.

II.3. Les instructions générales

Ce sont les instructions qui manipulent des données qui sont codées dans l'instruction directement. Nous verrons ceci plus en détail lorsque nous parlerons des modes d'adressage. Elles sont codées de la manière suivante :

- L'instruction est codée sur 6 bits
- Elle est suivie d'une valeur IMMEDIATE codée sur 8 bits (donc de 0 à 255).

II.4. Les sauts et appels de sous-routines

Ce sont les instructions qui provoquent une rupture dans la séquence de déroulement du programme. Elles sont codées de la manière suivante :

- Les instructions sont codées sur 3 bits
- La destination codée sur 11 bits

Nous pouvons déjà en déduire que les sauts ne donnent accès qu'à 2Ki de mémoire programme (2^{11}). Ceci ne pose aucun problème, le 16F84 ne disposant que de 1K mots de mémoire. Pour coder une adresse de saut à l'intérieur de la mémoire programme, il faut donc 10 bits (2^{10} = 1024 = 1Ki).

III. Panoramique des instructions

Le tableau suivant vous permet d'un simple regard de vous informer de la manière dont fonctionne chaque instruction du PIC16F84. La première colonne indique le MNEMONIQUE et les OPERANDES pour chaque opération. Les mnémoniques sont des mots réservés (donc que vous ne pouvez utiliser que pour cet usage) compris et interprétés par le programme d'assemblage.

Notez ici la confusion de langage commune pour le terme ASSEMBLEUR, qu'on utilise à la fois pour indiquer le programme qui permet d'assembler le code (programme d'assemblage), et le langage utilisé dans l'éditeur (langage d'assemblage).

La syntaxe doit être la suivante pour l'assembleur MPLAB (dans l'ordre) :

- Etiquette (facultative)
- Espace(s) ou tabulation(s)
- Mnémonique (en majuscules ou minuscules)
- Tabulation ou Espace(s)

- Opérande ou la valeur
- Virgule éventuelle de séparation
- Bit de destination W ou F ou éventuellement numéro du bit de 0 à 7 si nécessaire
- Espace(s) ou tabulation(s)
- Point-virgule. (facultatif si pas de commentaire)
- Commentaire. (facultatif)

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

Notez que le mnémonique ne peut pas se trouver en première colonne, et que tout ce qui suit le point-virgule est ignoré de l'assembleur (donc c'est de la zone commentaire). La première colonne est réservée pour les étiquettes (repères). Vous disposez également de la possibilité d'insérer un ou plusieurs espace(s) ou tabulation(s) de chaque côté de la virgule.

Voici à titre d'exemple deux lignes valides, les mots en vert sont des mots réservés, en bleu l'instruction, ceux en jaune étant libres, le reste est du commentaire :

```
Ma_ligne                ; Ceci est une étiquette
      MOVF  STATUS,W    ; charge le registre status dans le registre de travail
```

La seconde colonne du tableau donne un bref descriptif de l'instruction. La troisième colonne donne le nombre de cycles nécessaires pour exécuter l'instruction. Notez que toutes les instructions nécessitent un seul cycle, sauf les sauts qui en nécessitent 2, et les opérations de test avec saut, lorsque le résultat du test engendre le saut (instructions notées 1(2)).
La 4ème colonne du tableau donne ce qu'on appelle l'OPCODE, c'est à dire le mot binaire que MPLAB va générer pour vous au départ du mnémonique. Vous ne vous en servirez donc pas, mais sachez que vous pourriez programmer directement le PIC sans passer par un assembleur, directement en construisant un fichier .hex et en entrant les valeurs trouvées ici. Cette technique pour construire des programmes auto-modifiés pour cause de restriction mémoire.
La 5ème colonne du tableau est primordiale, car elle donne les INDICATEURS D'ETATS ou STATUS FLAGS affectés (modifiés) une fois l'instruction effectuée.
La dernière colonne renvoie à des notes en bas de page. La note 1 est très importante, elle fait allusion à la méthode « lecture/modification/écriture » propre aux ports d'entrées/sortie (I/O). La note 2 indique qu'une modification d'un timer remet à zéro son prédiviseur. La troisième note indique que si vous vous servez de l'instruction pour modifier le compteur de programme, il y aura un cycle supplémentaire. C'est logique car cela équivaut à un saut.

IV. Les indicateurs d'état

Ces indicateurs sont indispensables pour la programmation. Il est donc absolument nécessaire d'avoir compris leur fonctionnement (du moins pour Z et C). Tous les indicateurs sont des bits du registre STATUS. Nous aborderons ici les flags Z et C.

IV.1. L'indicateur d'état « Z »

C'est l'indicateur Zéro, il fonctionne de la manière suivante : Si le résultat d'une opération POUR LEQUEL IL EST AFFECTE, donne un résultat égal à 0, le flag Zéro passe à 1. Donc, dire «si Z=1» correspond à dire «si résultat = 0». La colonne 5 du tableau précédent indique les instructions qui modifient Z.
Donc, si vous faites une addition avec ADDWF et que le résultat obtenu est 0, le bit Z sera à 1. Si le résultat est <>0 (différent de 0), le bit Z vaudra 0. Dans les 2 cas il est modifié.
Par contre, si vous stockez une valeur avec l'instruction MOVWF, le bit Z ne sera pas modifié, même si la valeur vaut 0. Ces remarques sont valables pour les autres flags.

IV.2. L'indicateur d'état « C »

C'est l'indicateur pour Carry (report). Si le résultat d'une opération entraîne un débordement, le bit C sera positionné à 1. Il s'agit en fait du 9ème bit de l'opération. Exemple :
Si vous ajoutez B'11111110' (254)
+ B'00000011' (3)
Vous obtenez B'100000001', (257) donc 9 bits.
Comme les registres du PIC ne font que 8 bits, vous obtiendrez B'00000001' (1) et C positionné à 1

V. Détails des instruction du PIC16F84

V.1. L'instruction « GOTO » (aller à)

Cette instruction effectue ce qu'on appelle un saut inconditionnel, encore appelé rupture de séquence synchrone inconditionnelle. Rappelez-vous que l'instruction goto contient les 11 bits de l'emplacement de destination. Les 2 bits restants sont chargés depuis le registre PCLATH. Rappelez-vous en effet la manière dont est construit le PC dans le cas des sauts. On ne peut sauter qu'à l'intérieur d'une même PAGE de 2^{11}, soit 2048 mots. Ceci n'a aucune espèce d'importance pour le 16F84, qui ne dispose que de 1Ki mots de mémoire programme, mais devra être considéré pour les PIC de plus grande capacité (16F876). Voici en résumé le fonctionnement du goto :

- L'adresse de saut sur 11 bits est chargée dans le PC ;
- Les 2 bits manquants sont chargés depuis PCLATH (b3 et b4), pas pour le 16F84 ;

- Le résultat donne l'adresse sur 13 bits (10 bits pour le 16F84) ;
- La suite du programme s'effectue à la nouvelle adresse du PC.

Souvenez-vous, que pour le 16F84 : Adresse de saut = adresse réelle. Vous ne devez donc vous préoccuper de rien. Pour les autres, en cas de débordement, MPLAB vous le signalera.

Syntaxe

```
goto etiquette
```

Exemple

```
Start
goto plusloin  ; le programme saute à l'instruction qui suit l'étiquette « plusloin »
xxxxxxxx
plusloin
xxxxxxxx       ; instruction exécutée après le saut : le programme se poursuit ici
```

Remarquez que vous pouvez sauter en avant ou en arrière. goto nécessite 2 cycles d'horloge, comme pour tous les sauts.

V.2. L'instruction « INCF » (INCrement File)

Cette instruction provoque l'incrémentation de l'emplacement spécifié (encore appelé File).

Syntaxe

```
incf f,d
```

Comme pour toutes les instructions, « f » représente « File », c'est à dire l'emplacement mémoire concerné pour cette opération, « d », quant à lui: représente la Destination. Sauf spécification contraire, d vaut toujours, au choix :

- f (la lettre f) : dans ce cas le résultat est stocké dans l'emplacement mémoire.
- W (la lettre w) : dans ce cas, le résultat est laissé dans le registre de travail «accumulateur», et le contenu de l'emplacement mémoire n'est pas modifié.

La formule est donc (f) + 1 → (d) : Les parenthèses signifient « le contenu de ». Soit, en français : Le contenu de l'emplacement spécifié est incrémenté de 1, le résultat est placé dans l'emplacement désigné par « d ». Emplacement qui pourra être soit l'emplacement spécifié par « f », soit l'accumulateur, (f) restant dans ce cas inchangé.

Bit du registre STATUS affecté : Le seul bit affecté par cette opération est le bit Z.

Etant donné que la seule manière, en incrémentant un octet, d'obtenir 0, est de passer de 0xFF à 0x00. Le report n'est pas nécessaire, puisqu'il va de soi. Si, après une incrémentation, vous obtenez Z=1, c'est que vous avez débordé. Z vaudra donc 1 si (f) avant l'exécution valait 0xFF.

Exemples

```
incf mavariable , f   ; le contenu de ma variable est augmenté de 1
                      ; le résultat est stocké dans mavariable.
incf mavariable , w   ; Le contenu de mavariable est chargé dans w et ; augmenté de 1. W
                      ; contient donc le contenu de mavariable + 1. mavariable n'est pas modifiée
```

V.3. L'instruction « DECF » (DECRement File)

Décrémente l'emplacement spécifié. Le fonctionnement est strictement identique à l'instruction précédente.

Syntaxe

```
decf f , d ; (f) - 1 -> (d)
```

Bit du registre STATUS affecté : Le seul bit affecté par cette opération est le bit Z. Si avant l'instruction, (f) vaut 1, Z vaudra 1 après l'exécution (1-1 = 0)

Exemples

```
decf mavariable , f       ; décrémente mavariable, résultat dans mavariable
decf mavariable , w       ; prends (mavariable) - 1 et place le résultat dans w
```

V.4. L'instruction « MOVLW » (MOVe Literal to W)

Cette instruction charge la valeur spécifiée (valeur littérale, ou encore valeur immédiate), dans le registre de travail W.

Syntaxe

```
movlw k      ; k-> w : k représente une valeur de 0x00 à 0xFF.
```

Bit du registre STATUS affecté : Aucun (donc même si vous chargez la valeur '0'.)

Exemple

```
movlw 0x25     ; charge 0x25 dans le registre w
```

V.5. L'instruction « MOVF » (MOVe File)
Charge le contenu de l'emplacement spécifié dans la destination
Syntaxe
```
movf f , d      ; (f) -> (d)
```
Bit du registre STATUS affecté : Une fois de plus, seul le bit Z est affecté (si f vaut 0, Z vaut 1).
Exemple 1
Pour cette instruction, je vais me montrer beaucoup plus explicite. Vous allez comprendre pourquoi
```
movf mavariable,w    ; charge le contenu de mavariable dans w.
```
ATTENTION
Il est impératif ici de bien faire la distinction entre movlw k et movf f,w.
Dans le premier cas, c'est la VALEUR qui est chargée dans w, dans le second c'est le CONTENU de l'emplacement spécifié. Si nous exécutons l'instruction suivante :
```
movlw mavariable
```
L'assembleur va traduire en remplaçant mavariable par sa VALEUR. Attention, la valeur, ce n'est pas le contenu. Nous parlerons dans ce cas d'un ADRESSAGE IMMEDIAT. Le déroulement est du type f -> (d) (f n'est pas entre parenthèses).
Si nous exécutons par contre l'instruction suivante :
```
movf mavariable , w
```
L'assembleur va traduire également en remplaçant mavariable par sa valeur. Nous parlerons ici d'un ADRESSAGE DIRECT.
Exemple 2
```
movf mavariable , f
```
Elle place le CONTENU de mavariable dans mavariable. Dire que cela ne sert à rien est tentant mais prématuré. En effet, si le contenu de mavariable reste bien inchangé, par contre le bit Z est positionné à 1. Cette instruction permet donc de vérifier si (mavariable) = 0.
V.6. L'instruction « MOVWF » (MOVe W to File)
Permet de sauvegarder le contenu du registre de travail W dans un emplacement mémoire.
Syntaxe
```
movwf f ; (W) -> (f)
```
Bit du registre STATUS affecté : Aucun
Exemple
```
movlw 0x50 ; charge 0x50 dans W
movwf mavariable ; « mavariable » contient maintenant 0x50.
```
V.7. L'instruction « ADDLW » (ADD Literal and W)
Cette opération permet d'ajouter une valeur littérale (adressage immédiat) au contenu du registre de travail W.
Syntaxe
```
addlw k ; (W) + k -> (W)
```
Bits du registre STATUS affectés :
Z : Si le résultat de l'opération vaut 0, Z vaudra 1
C : Si le résultat de l'opération est supérieur à 0xFF (255) , C vaudra 1
DC : Si le résultat de l'opération entraîne en report du bit 3 vers le bit 4, DC vaudra 1
DC n'est utilisé que pour les opérations sur les quartets, par exemple, les nombres Binary Coded Decimal.
Exemple
```
movlw 253       ; charger 253 en décimal dans W
addlw 4         ; Ajouter 4. W contient 1, Z vaut 0, C vaut 1(déborde)
addlw 255       ; ajouter 255. W vaut 0, C vaut 1, Z vaut 1
```
V.8. L'instruction « ADDWF » (ADD W and F)
Ne pas confondre avec l'instruction précédente. Une nouvelle fois, il s'agit ici d'un ADRESSAGE DIRECT. Le CONTENU du registre W est ajouté au CONTENU du registre F
Syntaxe
```
addwf f , d ; (w) + (f) -> (d)
```

Bits du registre STATUS affectés : C, DC, et Z

Exemple

```
movlw 12              ; charger 12 dans W
movwf mavariable      ; mavariable vaut maintenant 12
movlw 25              ; charger 25 dans W
addwf mavariable,f    ; résultat : (W) + (mavariable), donc 25+12
                      ; résultat = 37 sauvé dans mavariable (,f).
```

V.9. L'instruction « SUBLW » (SUBtract W from Literal)

Attention, ici il y a un piège. L'instruction aurait du s'appeler SUBWL. En effet, on soustrait W de la valeur littérale, et pas l'inverse.

Syntaxe

```
sublw k ; k - (W) -> (W)
```

Bits du registre STATUS affectés : C, DC, Z

Notez ici que le bit C fonctionne de manière inverse que pour l'addition. Ceci est commun à la plupart des microprocesseurs du marché. Les autres utilisent parfois un bit spécifique pour la soustraction, bit le plus souvent appelé « borrow » (emprunt).

Si le résultat est positif, donc, pas de débordement : C =1. S'il y a débordement, C est forcé à 0.

Ceci est logique, et s'explique en faisant une soustraction manuelle. Le bit C représente le 9ème bit ajouté d'office à la valeur initiale. Si on effectue une soustraction manuelle donnant une valeur <0, on obtient donc une valeur finale sur 8 bits, le report obtenu venant soustraire le bit C. Si le résultat est >0, il n'y a pas de report, le résultat final reste donc sur 9 bits.

La formule de la soustraction est donc : k précédé d'un neuvième bit à 1 – contenu de W = résultat sur 8 bits dans W avec 9ème bit dans C.

Exemple 1

```
movlw 0x01      ; charger 0x01 dans W
sublw 0x02      ; soustraire W de 2
                ; résultat : 2 - (W) = 2-1 = 1
                ; Z = 0, C = 1, donc résultat positif
```

Effectuons cette opération manuellement :

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0	0	0	0	0	0	1	0^{10}	2
-	0	0	0	0	0	0	0	0^{1}	1	1
=	1	0	0	0	0	0	0	0	1	1

Comment procéder ? Et bien, comme pour une soustraction décimale. On commence par les bits de droite : 0-1, ça ne passe pas, donc on emprunte 10 (noté en exposant violet), et on soustraira évidemment une unité supplémentaire au bit b1. On a donc :B '10' – B'1', car souvenez-vous qu'on a emprunté 10 en BINAIRE. Résultat 1. On soustrait ensuite les b1 : on aura 1 – 0 – l'emprunt, donc 1-0-1 = 0. On continue de droite à gauche jusqu'au 9ème bit qui est le carry. Résultat final : B'00000001' et carry à 1, C.Q.F.D.

Exemple 2

```
movlw 0x02  ; charger 0x02 dans W
sublw 0x02  ; soustraire W de 2
```

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0	0	0	0	0	0	1	0	2
-	0	0	0	0	0	0	0	1	0	2
=	1	0	0	0	0	0	0	0	0	0

On procède toujours de la même manière.

Exemple 3

```
movlw 0x03  ; charger 0x03 dans W
sublw 0x02  ; soustraire W de 2
```

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0^{10}	0^{10}	0^{10}	0^{10}	0^{10}	0^{10}	1^{10}	0^{10}	2
-	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	1^{1}	1	3
=	0	1	1	1	1	1	1	1	1	-1

Procédons de la même manière, et nous obtenons B'11111111', avec le bit C à 0. Et là, me dites-vous, B'11111111', c'est FF, pas –1. Et bien, rappelez-vous ceci : Vous DEVEZ lire le bit C pour interpréter le résultat de votre soustraction.
Comme ce dernier vaut 0, vous êtes AVERTI que le résultat de l'opération est négatif. Or, comment savoir la valeur absolue d'un nombre négatif ? En prenant son complément à 2.
Rappelez-vous :
Complément à 1 de B'11111111' = B'00000000' (on inverse tous les bits)
La preuve, si vous ajoutez 1 à –1, vous obtenez B'11111111' + B'00000001' = B'00000000' = 0.
Vous maîtrisez maintenant les soustractions. Certains auront sans doute pensé que j'expliquais trop en détail, mais mon expérience m'a appris que les soustractions représentaient souvent un écueil dans la réalisation de trop de programmes.
Encore un dernier détail : Pour effectuer une soustraction de 1, vous pouvez bien entendu effectuer une addition de –1. Le résultat sera strictement le même, à votre charge d'interpréter les bits Z et C. Je vous laisse le faire vous-même pour vous convaincre.

V.10. L'instruction « SUBWF » (SUBtract W from F)

Nous restons dans les soustractions, mais, cette fois, au lieu d'un adressage immédiat, nous avons un ADRESSAGE DIRECT.

Syntaxe :

```
subwf f , d ; (f) - (W) -> (d)
```

Bits du registre STATUS affectés : C , DC , Z

Exemple

```
movlw 0x20            ; charger 0x20 dans w
movwf mavariable      ; mettre w dans (mavariable) (0x20)
movlw 0x1F            ; charger 0x1F dans w
subwf mavariable,w    ; (mavariable) - (w) -> (w)
                      ; 0x20 - 0x1F = 0x01
                      ; résultat dans w, C=1, Z=0
movwf autrevariable   ; sauver 0x01 dans une autre variable
```

V.11. L'instruction « ANDLW » (AND Literal with W)

Cette instruction effectue une opération « AND » BIT A BIT entre le contenu de W et la valeur littérale qui suit.

Syntaxe

```
andlw k ; avec k = octet : (w) AND k -> (w)
```

Bit du registre STATUS affecté : Z

Exemple

```
movlw B'11001101' ; charger w
andlw B'11110000' ; effectuer un 'and' (&)
```

	b7	b6	b5	b4	b3	b2	b1	b0
	1	1	0	0	1	1	0	1
And	1	1	1	1	0	0	0	0
=	1	1	0	0	0	0	0	0

Rappelez-vous qu'on effectue un AND entre chaque bit de même rang. Seuls restent donc positionnés à 1 les bits dont les 2 opérandes valent 1. Donc, le fait d'effectuer un AND avec la valeur B'11110000' MASQUE les bits 0 à 3, et ne laisse subsister que les bits 4 à 7.
Tant que vous ne jonglerez pas avec l'hexadécimal, je vous conseille de toujours traduire les nombres en binaires pour toutes les instructions concernant les bits.

V.12. L'instruction « ANDWF » (AND W with F)

Maintenant, vous devriez avoir bien compris. De nouveau la même opération, mais en ADRESSAGE DIRECT. Je vais donc accélérer les explications.

Syntaxe

```
andwf f , d ; (f) AND (w) -> (d)
```

Bit du registre STATUS affecté : Z

Exemple

```
movlw 0xC8 ; charger 0XC8 dans w
movwf mavariable ; sauver dans mavariable
movlw 0xF0 ; charger le masque
andwf mavariable,f ; (mavariable) = 0xC0 (on a éliminé le quartet faible)
```

V.13. L'instruction « IORLW » (Inclusive OR Literal with W)

Et oui, les mêmes instructions, mais pour le OU inclusif. Inclusif signifie simplement le contraire d'exclusif, c'est à dire que le bit de résultat vaudra 1 si un des bits, OU LES DEUX BITS, opérandes =1.

Syntaxe

```
iorlw k ; (w) OR k -> (w)
```

Bit du registre STATUS affecté : Z

Exemple

```
movlw 0xC3 ; charger 0xC3 dans W
iorlw 0x0F ; FORCER les bits 0 à 3
           ; résultat ; (w) = 0xCF
```

	b7	b6	b5	b4	b3	b2	b1	b0
	1	1	0	0	0	0	1	1
OR	0	0	0	0	1	1	1	1
=	1	1	0	0	1	1	1	1

Donc, avec un ou inclusif (OR), on peut FORCER n'importe quel bit à 1 (pour rappel, avec AND, on peut forcer n'importe quel bit à 0).

V.14. L'instruction « IORWF » (Inclusive OR W with File)

Effectue un OR entre (w) et l'emplacement spécifié. C'est donc une instruction en ADRESSAGE DIRECT. Je ne donnerai pas d'exemple, vous devriez avoir compris.

Syntaxe

```
iorwf f , d ; (w) OR (f) -> (d)
```

Bit du registre STATUS affecté : Z

V.15. L'instruction « XORLW » (eXclusive OR Literal with W)

Par opposition au ou inclusif, voici maintenant le OU EXCLUSIF. Sa table de vérité est la même que le ou inclusif, excepté que lorsque les 2 bits sont à 1, le résultat est 0.

Cette instruction peut donc servir à INVERSER n'importe quel bit d'un octet. En effet, si vous effectuez « 1 xor 0 », vous obtenez 1, si vous effectuez « 0 xor 0 », vous obtenez 0.

Donc, si vous appliquez xor 0, la valeur de départ est inchangée.

Si par contre vous appliquez « 0 xor 1 » , vous obtenez 1, et avec « 1 xor 1 », vous obtenez 0. En appliquant xor 1, vous inversez le bit, quelque soit son état initial.

Maintenant, vous pouvez donc FORCER un bit à 1 avec OR, MASQUER un bit (le mettre à 0) avec AND, et l'INVERSER avec XOR.

Syntaxe

```
xorlw k ; (w) xor k -> (w)
```

Bit du registre STATUS affecté : Z

Exemple

```
movlw B'11000101'    ; charger W
xorlw B'00001111'    ; xor avec la valeur
                     ; résultat : B '11001010'
                     ; les 4 bits de poids faible ont été inversés
```

	b7	b6	B5	B4	b3	b2	b1	b0
	1	1	0	0	0	1	0	1
Xor	0	0	0	0	1	1	1	1
=	1	1	0	0	1	0	1	0

Remarquez que tous les bits de l'octet initial ont été inversés par chaque bit du second opérande qui était à 1.

V.16. L'instruction « XORWF » (eXclusive OR W with F)

C'est exactement la même opération que XORLW, mais en ADRESSAGE DIRECT.

Syntaxe

```
xorwf f , d ; (w) xor (f) -> (d)
```

Bit du registre STATUS affecté : Z

V.17. L'instruction « BSF » (Bit Set F)

C'est une instruction qui permet tout simplement de forcer directement un bit d'un emplacement mémoire à 1.

Syntaxe

```
bsf f , b   ; le bit n° b est positionné dans la case mémoire (f)
            ; b est évidemment compris entre 0 et 7
```

Bit du registre STATUS affecté : Aucun

Exemples

```
bsf STATUS , C        ; positionne le bit C à 1 dans le registre STATUS
bsf mavariable , 2    ; positionne bit 2 de (mavariable) à 1
```

V.18. L'instruction « BCF » (Bit Clear F)

C'est une instruction qui permet tout simplement de forcer directement un bit d'un emplacement mémoire à 0.

Syntaxe

```
bcf f , b   ; le bit n° b est mis à 0 dans la case mémoire (f)
            ; b est évidemment compris entre 0 et 7
```

Bit du registre STATUS affecté : Aucun

Exemples

```
bcf STATUS , C ; positionne le bit C à 0 dans le registre STATUS
bcf mavariable , 2 ; positionne b2 de (mavariable) à 0
```

V.19. L'instruction « RLF » (Rotate Left through Carry)

Rotation vers la gauche en utilisant le carry. Les opérations de décalage sont des opérations très souvent utilisées. Les PIC® ont la particularité de ne disposer que d'instructions de ROTATION. Vous allez voir qu'avec ces instructions, on peut très facilement réaliser des décalages.

L'opération de rotation effectue l'opération suivante : Le bit de carry C est mémorisé. Ensuite chaque bit de l'octet est déplacé vers la gauche. L'ancien bit 7 sort de l'octet par la gauche, et devient le nouveau carry. Le nouveau bit 0 devient l'ancien carry. Il s'agit donc d'une rotation sur 9 bits.

Syntaxe

```
rlf f , d ; (f) rotation gauche avec carry-> (d)
```

Bit du registre STATUS affecté : C

Exemple1

Un petit exemple vaut mieux qu'un long discours.

```
bsf STATUS,C ; positionne le carry à 1
movlw B'00010111'    ; charge la valeur dans w
movwf mavariable     ; initialise mavariable
rlf mavariable,f     ; rotation vers la gauche
```

	C	b7	b6	b5	b4	b3	b2	b1	b0
F	1	0	0	0	1	0	1	1	1
Rlf	0	0	0	1	0	1	1	1	1

Vous voyez que tous les bits ont été décalés vers la gauche. « C » a été réintroduit dans b0. Le résultat reste sur 9 bits.

Exemple 2

```
bcf STATUS,C         ; positionne le carry à 0
movlw b'00010111'    ; charge la valeur dans w
movwf mavariable     ; initialise mavariable
rlf mavariable,f     ; rotation vers la gauche
```

Si vous avez compris, le résultat sera B'00101110', avec le carry à 0. Si le carry était à 0 au départ, on effectue un simple décalage vers la gauche. Que se passe-t-il si, en décimal, on effectue ce type d'opération ?

Prenons le nombre 125 et décalons-le vers la gauche en décimal, nous obtenons 1250. Nous avons multiplié le nombre par sa base (décimal = base 10). Et bien c'est la même chose en binaire (une fois de plus).
Prenons B'00010111 ', soit 23 en décimal. Décalons-le, nous obtenons B'00101110', soit 46. Nous avons donc effectué une multiplication par 2. Retenez ceci, cela vous sera très utile par la suite.

V.20. L'instruction « RRF » (Rotate Right through Carry)

Rotation vers la droite en utilisant le carry. L'opération de rotation vers la droite effectue l'opération suivante : Le bit de carry « C » est mémorisé. Ensuite chaque bit de l'octet est déplacé vers la droite. L'ancien bit 0 sort de l'octet par la droite, et devient le nouveau carry. L'ancien carry devient le nouveau bit7. Il s'agit donc également d'une rotation sur 9 bits.

Syntaxe

```
rrf f , d ; (f) rotation droite avec carry-> (d)
```

Bit du registre STATUS affecté : C

Exemple1

```
bsf STATUS,C          ; positionne le carry à 1
movlw B'00010111'     ; charge la valeur dans w
movwf mavariable      ; initialise mavariable
rrf mavariable,f      ; rotation vers la droite
```

	b7	b6	b5	B4	b3	b2	b1	b0	C
F	0	0	0	1	0	1	1	1	1
Rrf	1	0	0	0	1	0	1	1	1

Vous voyez que tous les bits ont été décalés vers la droite. C a été réintroduit dans b7. Le résultat reste sur 9 bits.

Exemple 2

```
bcf STATUS,C          ; positionne le carry à 0
movlw b'00010111'     ; charge la valeur dans w
movwf mavariable      ; initialise mavariable
rrf mavariable,f      ; rotation vers la droite
```

Si vous avez compris, le résultat sera B'00001011', avec le carry à 1. Si le carry est à 0 au départ, on effectue un simple décalage vers la droite.
Que s'est-il passé ? Et bien notre nombre de départ, soit 23 en décimal est devenu 11. Le carry représente le bit « -1 », donc, la moitié du bit 0, donc ½. En effet, en décimal, le chiffre « 1 » derrière les unités a comme valeur 1/base, donc 1/10. En binaire ce sera donc ½.
Si nous regardons alors les 9 bits, nous obtenons 11 ½ . Nous avons donc effectué une DIVISION PAR 2. Retenez ceci, cela vous sera également très utile par la suite.

V.21. L'instruction « BTFSC » (Bit Test F, Skip if Clear)

Traduit littéralement, cela donne : Teste le bit de l'emplacement mémoire et saute s'il vaut 0. Il s'agit ici de votre premier SAUT CONDITIONNEL, ou RUPTURE DE SEQUENCE SYNCHRONE CONDITIONNELLE. En effet, il n'y aura saut que si la condition est remplie.
Notez que dans ce cas l'instruction prendra 2 cycles, sinon, elle n'utilisera qu'un cycle. De plus, il faut retenir que pour tous ces types de saut, ON NE SAUTE QUE l'INSTRUCTION SUIVANTE. En effet, la syntaxe ne contient pas d'adresse de saut, comme nous allons le voir

Syntaxe

```
btfsc f, b ; on teste le bit b de la mémoire (f).
; si ce bit vaut 0, on saute l'instruction suivante, sinon
; on exécute l'instruction suivante.
Instruction exécutée si faux ; si le bit vaut 0, ne sera pas exécutée (skip)
Poursuite du programme ; le programme continue ici
```

Bit du registre STATUS affecté : Aucun

Exemple1

Voici un exemple dans lequel on doit exécuter une seule instruction supplémentaire si le bit vaut 1.

```
btfsc STATUS,C        ; tester si le bit C du registre STATUS vaut 0
bsf mavariable,2      ; non (C=1), alors bit 2 de mavariable mis à 1
xxxx                  ; la suite du programme est ici dans les 2 cas
```

Exemple 2

Que faire si les traitements nécessitent plusieurs instructions ? Et bien, on combine les sauts conditionnels avec les saut inconditionnels (par exemple goto).

```
movlw 0x12          ; charger 12 dans le registre de travail
subwf mavariable,f  ; on soustrait 0x12 de mavariable
btfsc STATUS,C      ; on teste si le résultat est négatif (C=0)
goto positif        ; non, alors au saute au traitement des positifs
xxxx                ; on poursuit ici si le résultat est négatif
```

Ces procédures sont les mêmes pour tous les sauts inconditionnels, je ne les détaillerai donc pas avec autant d'explications.

V.22. L'instruction « BTFSS » (Bit Test F, Skip if Set)

Traduit littéralement, cela donne : Teste le bit de l'emplacement mémoire et saute s'il vaut 1. Toutes les remarques de l'instruction « BTFSC » restent valables.

Syntaxe

```
btfss f, b     ; on teste le bit b de la mémoire (f).
               ; si ce bit vaut 1, on saute l'instruction
               ; suivante, sinon
               ; on exécute l'instruction suivante.
xxxx           ; si le bit vaut 1, ne sera pas exécutée (skip)
xxxx           ; Le programme continue ici
```

Bit du registre STATUS affecté : Aucun

Exemple

```
btfss STATUS,C      ; tester si le bit C du registre STATUS vaut 1
bsf mavariable,2    ; non (C=0), alors bit 2 de mavariable mis à 1
xxxx                ; la suite du programme est ici dans les 2 cas
```

V.23. L'instruction « DECFSZ » (DECrement F, Skip if Z)

Nous poursuivons les sauts conditionnels avec une instruction très utilisée pour créer des boucles. Cette instruction décrémente un emplacement mémoire et saute l'instruction suivante si le résultat de la décrémentation donne une valeur nulle.

Syntaxe

```
decfsz f, d ; (f) -1 -> (d). Saut si (d) = 0
```

Bit du registre STATUS affecté : Aucun

Exemple1

```
movlw 3             ; charger 3 dans w
movwf compteur      ; initialiser compteur
movlw 0x5           ; charger 5 dans w
boucle              ; étiquette
addwf mavariable , f ; ajouter 5 à ma variable
decfsz compteur , f  ; décrémenter compteur et tester sa valeur
goto boucle         ; si compteur pas 0, on boucle
movf mavariable , w  ; on charge la valeur obtenue dans w
```

Comment écrit-on ce type de programme ? Et bien tout simplement de la manière suivante :

- on initialise le compteur de boucles.
- on place une étiquette de début de boucle
- on écrit les instructions qui doivent s'exécuter plusieurs fois
- l'instruction decfsz permet de déterminer la fin de la boucle
- l'instruction goto permet de localiser le début de la boucle.

ATTENTION / Si vous aviez mis

```
ecfsz compteur , w ; décrémenter compteur et tester sa valeur
```

La boucle n'aurait jamais de fin, car la variable compteur ne serait jamais modifiée.

- Si vous placez 0 dans le compteur de boucles, elle sera exécutée 256 fois. Si vous ne désirez pas qu'elle soit exécutée dans cette circonstance, vous devez ajouter un test AVANT l'exécution de la première boucle, comme dans l'exemple suivant :

Exemple 2

```
movf compteur,f     ; permet de positionner Z
```

```
btfsc STATUS , Z     ; sauter si Z = 0, donc si compteur >0
goto suite           ; compteur = 0, ne pas traiter la boucle
boucle               ; étiquette de début de boucle
addwf mavariable , f ; ajouter 5 à ma variable
decfsz compteur , f  ; décrémenter compteur et tester sa valeur
goto boucle          ; si compteur pas 0, on boucle
suite                ; on saute directement ici si compteur = 0
movf mavariable , w  ; on charge la valeur obtenue dans w
```

V.24. L'instruction « INCFSZ » (INCrement F, Skip if Zero)

Je ne vais pas détailler cette instruction, car elle est strictement identique à la précédente, hormis le fait qu'on incrémente la variable au lieu de la décrémenter.

Syntaxe

```
incfsz f , d ; (f) + 1 -> (d) : saut si (d) = 0
```

Bit du registre STATUS affecté : Aucun

V.25. L'instruction « SWAPF » (SWAP nibbles in F)

Nous pouvons traduire cette instruction par «inverser les quartets dans F ». Cette opération inverse simplement le quartet (demi-octet) de poids faible avec celui de poids fort.

Syntaxe

```
swapf f , d ; inversion des b0/b3 de (f) avec b4/b7 -> (d)
```

Bit du registre STATUS affecté : Aucun : cette particularité nous sera très utile lorsque nous verrons les interruptions.

Exemple

```
movlw 0xC5           ; charger 0xC5 dans w
movwf mavariable     ; placer dans mavariable
swapf mavariable , f ; (mavariable) = 0x5C
```

V.26. L'instruction « CALL » (CALL subroutine)

Cette opération effectue un saut inconditionnel vers un sous-programme. Voyons ce qu'est un sous-programme. Et bien, il s'agit tout simplement d'une partie de programme qui peut être appelé depuis plusieurs endroits du programme dit « principal ».

Le point de départ est mémorisé automatiquement, de sorte qu'après l'exécution du sous-programme, le programme continue depuis l'endroit où il était arrivé. Cela paraît un peu ardu, mais c'est extrêmement simple. Voyez les exemples dans la description de l'instruction RETURN.

Syntaxe

```
call etiquette ; appel de la sous-routine à l'adresse etiquette.
```

Mécanisme

Lors de l'exécution de l'instruction, l'adresse de l'instruction suivante est sauvegardée sur le sommet d'une pile (exactement comme une pile d'assiettes). Lorsque la sous-routine est terminée, l'adresse sauvegardée est retirée de la pile et placée dans le PC. Le programme poursuit alors depuis l'endroit d'où il était parti.

Notez que si le sous-programme (ou sous-routine) appelle lui-même un autre sous programme, l'adresse sera également sauvée au dessus de la pile.

Attention, cette pile a une taille limitée à 8 emplacements. Il n'existe aucun moyen de tester la pile, vous devez donc gérer vos sous-programmes pour ne pas dépasser 8 emplacements, sinon, votre programme se plantera.

Notez que lorsque vous sortez d'un sous-programme, l'emplacement est évidemment libéré. La limite n'est donc pas dans le nombre de fois que vous appelez votre sous- programme, mais dans le nombre d'imbrications (sous-programme qui en appelle un autre qui en appelle un autre) etc.

Bit du registre STATUS affecté : Aucun

V.27. L'instruction « RETURN » (RETURN from subroutine)

Retour de sous-routine. Va toujours de pair avec une instruction call. Cette instruction indique la fin de la portion de programme considérée comme sous-routine (SR). Rappelez-vous que pour chaque instruction « call » rencontrée, votre programme devra rencontrer une instruction «return».

Syntaxe

```
return   ; fin de sous-routine. Le PC est rechargé depuis la pile, le
         ; programme poursuit à l'adresse qui suit la ligne call.
```

Bit du registre STATUS affecté : Aucun

Exemples

Comme ceci est un concept très important, je vais détailler un peu plus. Imaginons un programme qui a besoin d'une petite temporisation (comme chaque instruction prend du temps, on peut s'arranger pour en faire perdre volontairement au programme afin de retarder son fonctionnement. Ecrivons-la :

```
movlw 0xCA          ; valeur du compteur
movwf compteur      ; initialiser compteur de boucles
boucle
decfsz compteur,f   ; décrémenter compteur, sauter si 0
goto boucle         ; boucler si pas 0
xxx                 ; suite du programme
```

Imaginons maintenant que cette petite temporisation soit appelée régulièrement par notre programme principal. Ecrivons à quoi ressemble le programme principal :

```
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      tempo       ; ici, on a besoin d'une tempo
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      tempo       ; ici aussi
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      tempo       ; et encore ici
      xxx         ; instruction quelconque
```

La première chose qui vient à l'esprit, est d'effectuer un copier/coller de notre temporisation. On obtient donc un programme comme ceci :

```
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      movlw 0xCA  ; valeur du compteur
      movwf compteur    ; initialiser compteur de boucles
      boucle
      decfsz compteur,f ; décrémenter compteur, sauter si 0
      goto boucle       ; boucler si pas 0
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      xxx         ; instruction quelconque
      movlw 0xCA        ; valeur du compteur
      movwf compteur    ; initialiser compteur de boucles
      boucle2
      decfsz compteur,f ; décrémenter compteur, sauter si 0
      goto boucle2      ; boucler si pas 0
      xxx               ; instruction quelconque
      xxx               ; instruction quelconque
      movlw 0xCA        ; valeur du compteur
      movwf compteur    ; initialiser compteur de boucles
boucle3
      decfsz compteur,f ; décrémenter compteur, sauter si 0
      goto boucle3      ; boucler si pas 0
```

Ceci n'est pas élégant, car, si nous devons changer la valeur de notre tempo, nous devons la changer partout dans le programme, sans oublier un seul endroit. De plus, ça prend beaucoup de place.

On peut également se dire : « utilisons une macro », qui, rappelez-vous, effectue une substitution au moment de l'assemblage. C'est vrai, qu'alors, il n'y a plus qu'un endroit à modifier, mais, dans notre PIC®, le code se retrouvera cependant autant de fois qu'on a utilisé la temporisation. Que de place perdue, surtout si la portion de code est grande et utilisée plusieurs fois.
Pour remédier à ceci, nous utiliserons la technique des sous-programmes. Première étape, modifions notre temporisation pour en faire une sous-routine :

```
tempo             ; étiquette de début de la sous-routine
movlw 0xCA        ; valeur du compteur
movwf compteur    ; initialiser compteur de boucles
boucle
decfsz compteur,f ; décrémenter compteur, sauter si 0
goto boucle       ; boucler si pas 0
return            ; fin de la sous-routine.
```

Deuxième étape, nous modifions notre programme principal pour que chaque fois que nous avons besoin d'une tempo, il appelle le sous-programme. Nous obtenons :

```
xxx         ; instruction quelconque
xxx         ; instruction quelconque
xxx         ; instruction quelconque
call tempo  ; appel du sous-programme
xxx         ; instruction quelconque, le programme continue ici
xxx         ; instruction quelconque
xxx         ; instruction quelconque
call tempo  ; appel du sous-programme
xxx         ; instruction quelconque, le programme continue ici
xxx         ; instruction quelconque
call tempo  ; appel du sous-programme
xxx         ; instruction quelconque, le programme continue ici
```

Dans ce cas, la routine tempo n'est présente qu'une seule fois en mémoire programme. On peut améliorer : supposons que nous désirons une temporisation à durée variable. On modifie la sous-routine en supprimant la valeur d'initialisation, et on place celle-ci dans le programme principal. Cela s'appelle un sous-programme avec passage de paramètre(s).
Exemple, notre sous-programme devient :

```
tempo               ; étiquette de début de la sous-routine
movwf compteur      ; initialiser compteur de boucles
boucle
decfsz compteur,f   ; décrémenter compteur, sauter si 0
goto boucle         ; boucler si pas 0
return              ; fin de la sous-routine.
```

Quant à notre programme principal :

```
xxx         ; instruction quelconque
xxx         ; instruction quelconque
xxx         ; instruction quelconque
xxx         ; instruction quelconque
movlw 0x25  ; charger w avec 0x25
call tempo  ; appel du sous programme tempo d'une durée de 0x25
xxx         ; instruction quelconque, le programme continue ici
xxx         ; instruction quelconque
xxx         ; instruction quelconque
movlw 0x50  ; charger w avec 0x50
call tempo  ; appel du sous programme tempo d'une durée de 0x50
xxx         ; instruction quelconque, le programme continue ici
xxx         ; instruction quelconque
movlw 0x10  ; charger w avec 0x10
call tempo  ; appel du sous programme tempo d'une durée de 0x10
xxx         ; instruction quelconque, le programme continue ici
```

Voilà, maintenant vous savez ce qu'est un sous-programme. Enfantin, n'est-ce pas ?

V.28. L'instruction « RETLW » (RETurn with Literal in W)

Retour de sous-routine avec valeur littérale dans W. C'est une instruction très simple : elle équivaut à l'instruction return, mais permet de sortir d'une sous-routine avec une valeur spécifiée dans W.

Syntaxe

```
retlw k ; (w) = k puis return
```

Bit du registre STATUS affecté : Aucun

Exemple

test ; étiquette de début de notre sous-programme
btfss mavariable,0 ; teste le bit 0 de mavariable
retlw 0 ; si vaut 0, fin de sous-programme avec (w)=0
retlw 1 ; sinon, on sort avec (w) = 1

Le programme qui a appelé la sous-routine connaît donc le résultat de l'opération en lisant le registre «w ».

V.29. L'instruction « RETFIE » (RETurn From IntErrupt)

Cette instruction indique un retour d'interruption (nous verrons ultérieurement ce que sont les interruptions). Cette instruction agit d'une manière identique à RETURN, excepté que les interruptions sont remises automatiquement en service au moment du retour au programme principal.

Syntaxe

```
retfie ; retour d'interruption
```

Bit du registre STATUS affecté : Aucun

V.30. L'instruction « CLRF » (CLeaR F)

Cette instruction efface l'emplacement mémoire spécifié

Syntaxe

```
clrf f ; (f) = 0
```

Bit du registre STATUS affecté : Z : Vaut donc toujours 1 après cette opération.

Exemple

```
Clrf mavariable ; (mavariable) = 0
```

V.31. L'instruction « CLRW » (CLeaR W)

Cette instruction efface w

Syntaxe

```
clrw ; (w) = 0
```

C'est une instruction qui n'est pas vraiment indispensable, car on pourrait utiliser l'instruction « movlw 0 ». Cependant, à la différence de movlw 0, clrw positionne le bit Z.

Bit du registre STATUS affecté : Z : Vaut donc toujours 1 après cette opération.

V.32. L'instruction « CLRWDT » (CLeaR WatchDog)

Remet à 0 le chien de garde (watchdog) de votre programme. Nous aborderons la mise en œuvre du watchdog ultérieurement. Sachez cependant que c'est un mécanisme très pratique qui permet de provoquer un reset automatique de votre PIC en cas de plantage du programme (parasite par exemple).

Le mécanisme est simple à comprendre : il s'agit pour votre programme d'envoyer cette instruction à intervalles réguliers. Si la commande n'est pas reçue dans le délai imparti, le PIC redémarre à l'adresse 0x00. C'est exactement le mécanisme utilisé par les conducteurs de train qui doivent presser un bouton à intervalle régulier. Si le bouton n'est pas pressé, le train s'arrête. On détecte ainsi si le conducteur est toujours dans l'état d'attention requis.

Syntaxe

```
clrwdt ; remet le timer du watchdog à 0
```

Bit du registre STATUS affecté : Aucun

V.33. L'instruction « COMF » (COMplement F)

Effectue le complément à 1 de l'emplacement mémoire spécifié. Donc, inverse tous les bits de l'octet désigné.

Syntaxe

```
comf f , d ; NOT (f) -> (d)
```

Bit du registre STATUS affecté : Z

Exemple

```
movlw B'11001010'    ; charge valeur dans W
movwf mavariable     ; initialise mavariable
comf mavariable,w    ; charge l'inverse de mavariable dans W
                     ; (W) = B'00110101'
```

Astuce : en utilisant cette instruction, on peut également tester si mavariable vaut 0xFF. En effet, si c'est le cas, W vaudra 0 et Z vaudra donc 1.

V.34. L'instruction « SLEEP » (Mise en sommeil)

Place le PIC en mode de sommeil. Il ne se réveillera que sous certaines conditions que nous verrons plus tard.

Syntaxe

```
sleep ; arrêt du PIC
```

Bit du registre STATUS affecté : T0, PD

V.35. L'instruction « NOP » (No Operation)

Comme vous devez être fatigué, et moi aussi, je vous présente l'instruction qui ne fait rien, qui ne positionne rien, et qui ne modifie rien. On pourrait croire qu'elle ne sert à rien. En fait elle est surtout utilisée pour perdre du temps, par exemple pour attendre une ou deux instructions, le temps qu'une acquisition ai pu se faire, par exemple. Nous l'utiliserons donc à l'occasion.

Syntaxe

```
nop ; tout simplement
```

Ceci termine l'analyse des 35 instructions utilisées normalement dans les PIC® mid-range.

Ceci peut vous paraître ardu, mais en pratiquant quelque peu, vous connaîtrez très vite toutes ces instructions par coeur. Pensez pour vous consoler que certains processeurs CISC disposent de plusieurs centaines d'instructions.

V.36. Les instructions obsolètes

Il reste 2 instructions qui étaient utilisées dans les précédentes versions de PIC®. Elles sont encore reconnues par le PIC16F84 mais leur utilisation est déconseillée par Microchip®. En effet, leur compatibilité future n'est pas garantie. Il s'agit de l'instruction OPTION, qui place le contenu du registre « W » dans le registre OPTION_REG, et de l'instruction TRIS, qui place le contenu de « W » dans le registre TRISA ou TRISB suivant qu'on utilise TRIS PORTA ou TRIS PORTB.

Ces instructions ne sont plus nécessaires actuellement, car ces registres sont désormais accessibles directement à l'aide des instructions classiques. Je vous conseille donc fortement d'éviter de les utiliser, sous peine de rencontrer des problèmes avec les futures versions de PIC de Microchip.

Je ne vous les ai donc présentées que pour vous permettre de comprendre un éventuel programme écrit par quelqu'un d'autre qui les aurait utilisées.

VI. Les modes d'adressage

Les instructions utilisent toutes une manière particulière d'accéder aux informations qu'elles manipulent. Ces méthodes sont appelées « modes d'adressage ».

Je vais simplement donner un petit exemple concret de ce qu'est chaque mode d'adressage. Supposons que vous vouliez mettre de l'argent dans votre poche :

VI.1. L'adressage littéral ou immédiat

Avec l' ADRESSAGE IMMEDIAT ou ADRESSAGE LITTERAL, vous pouvez dire : 'je mets 100DA en poche'. La valeur fait IMMEDIATement partie de la phrase. J'ai donné LITTERALlement la valeur concernée. Pas besoin d'un autre renseignement.

Exemple

```
movlw 0x55 ; charger la valeur 0x55 dans W
```

VI.2. L'adressage direct

Avec l' ADRESSAGE DIRECT, vous pouvez dire : je vais mettre le contenu du coffre numéro 10 dans ma poche. Ici, l'emplacement contenant la valeur utile est donné DIRECTement dans la phrase. Mais il faut d'abord aller ouvrir le coffre pour savoir ce que l'on va effectivement mettre en poche. On ne met donc pas en poche le numéro du coffre, mais ce qu'il contient. Pour faire l'analogie avec les syntaxes précédentes, je peux dire que je mets (coffre 10) dans ma poche.

Exemple

```
movf 0x10 , W ; charger le contenu de l'emplacement 0x10 dans W
```

VI.3. L'adressage indirect

Avec l'ADRESSAGE INDIRECT, vous pouvez dire : Le préposé du guichet numéro 3 va me donner le numéro du coffre qui contient la somme que je vais mettre en poche. Ici, vous obtenez le numéro du coffre INDIRECTement par le préposé au guichet.

Vous devez donc allez demander à ce préposé qu'il vous donne le numéro du coffre que vous irez ouvrir pour prendre l'argent. On ne met donc en poche, ni le numéro du préposé, ni le numéro du coffre que celui-ci va vous donner. Il y a donc 2 opérations préalables avant de connaître la somme que vous empocherez.

Cet adressage fait appel à 2 registres, dont un est particulier, car il n'existe pas vraiment. Examinons-les donc :

VI.3.1. Les registres FSR et INDF

Ceux qui suivent sont déjà en train de chercher dans le tableau 4-2 après INDF. INDF signifie INDirect File. Vous le voyez maintenant ? Et oui, c'est le fameux registre de l'adresse 0x00. Ce registre n'existe pas vraiment, ce n'est qu'un procédé d'accès particulier à FSR utilisé par le PIC pour des raisons de facilité de construction électronique interne.

Le registre FSR est à l'adresse 0x04 dans les 2 banques. Il n'est donc pas nécessaire de changer de banque pour y accéder, quelle que soit la banque en cours d'utilisation. Dans l'exemple schématique précédent, le préposé au guichet, c'est le registre FSR. L'adressage indirect est un peu particulier sur les PIC, puisque c'est toujours à la même adresse que se trouvera l'adresse de destination. En somme, on peut dire qu'il n'y a qu'un seul préposé (FSR) dans notre banque. Comment cela se passe-t-il ?

Premièrement, nous devons écrire l'adresse pointée dans le registre FSR. Ensuite, nous accédons à cette adresse pointée par le registre INDF. On peut donc dire que INDF est en fait le registre FSR utilisé pour accéder à la case mémoire. Donc, quand on veut modifier la case mémoire pointée, on modifie FSR, quand on veut connaître l'adresse de la case pointée, on accède également à FSR. Si on veut accéder au CONTENU de la case pointée, on accède via INDF. Nous allons voir tout ceci par un petit exemple, mais avant,

ATTENTION

Le contenu du registre FSR pointe sur une adresse en 8 bits. Or, sur certains PIC®, la zone RAM contient 4 banques (16F876). L'adresse complète est donc une adresse sur 9 bits.

L'adresse complète est obtenue, en adressage DIRECT, par l'ajout des bits 7 et 8 sous forme de RP0¨et RP1 (RP1 est inutilisé pour le 16F84 car seulement 2 banques), et par l'ajout du bit IRP dans le cas de l'adressage INDIRECT (inutilisé sur le 16F84). Veillez donc à toujours laisser IRP (dans le registre STATUS) et RP1 à 0 pour assurer la portabilité de votre programme.

Exemple

```
movlw 0x50        ; chargeons une valeur quelconque
movwf mavariable  ; et plaçons-la dans la variable « mavariable »
movlw mavariable  ; on charge l'ADRESSE de mavariable, par ; exemple, dans
                  les leçons précédentes, c'était ; 0x0E. (W) = 0x0E
movwf FSR         ; on place l'adresse de destination dans FSR.
```

```
                        ; on dira que FSR POINTE sur mavariable
      movf INDF,w       ; charger le CONTENU de INDF dans W.
```

LE CONTENU DE INDF EST TRADUIT PAR LE PIC® COMME ETANT LE CONTENU DE L'EMPLACEMENT MEMOIRE POINTE PAR FSR (W) = 0X50

VI.4. Quelques exemples

Pour les habitués des processeurs divers, excusez ces répétitions. Les registres sont initialisés avec les valeurs précédentes.

```
      movlw mavariable
```

C'est de l'adressage immédiat ou littéral ; donc on charge la VALEUR de mavariable, ce qui correspond en réalité à son ADRESSE. Donc 0x0E est placé dans (W). Ceci se reconnaît au « l » de l'instruction movlw. Attention, la valeur de mavariable ce n'est pas son contenu.

```
      movf mavariable , w
```

Cette fois, c'est de l'adressage DIRECT, donc, on va à l'adresse mavariable voir ce qu'il y a à l'intérieur. On y trouve le CONTENU de mavariable, donc (w) = 0x50 (dans notre exemple). Pour l'assembleur, « mavariable » sera remplacée par « 0x0E », donc movf 0x0E,w

```
      movf INDF , w
```

Maintenant, c'est de l'adressage INDIRECT. Ce mode d'adressage se reconnaît immédiatement par l'utilisation du registre INDF. Le PIC va voir dans le registre FSR, et lit l'adresse contenue, dans ce cas 0X0E. Il va ensuite à l'emplacement visé, et lit le CONTENU. Donc, dans (W) on aura le contenu de 0x0E, soit 0x50.

```
      movf FSR , w
```

Ceci est un piège. C'est en effet de l'adressage DIRECT. On placera donc dans (W) le CONTENU du registre FSR, donc 0X0E (l'adresse pointée) sera mis dans (W).

Vous aurez donc constaté qu'il n y a nulle part des modes d'adressage de type indexé (pré et post, avec ou sans offset). En fait, c'est tout simplement parce que ce mode d'adressage n'existe pas dans les PIC de type 16F. Il vous faudra donc faire preuve d'astuce pour compenser ces lacunes. Il est logique que la facilité d'apprentissage et de mise en œuvre se paye par une moins grande souplesse de création des applications.

Annexe : Jeu d'instructions du 8086

Transfert de données			
Général		**Entrées/Sorties**	
MOV	Déplacement d'un octet ou d'un mot	**IN**	Lecture d'un port d'E/S
PUSH	Ecriture d'un mot au sommet de la pile	**OUT**	Ecriture d'un port d'E/S
POP	Lecture d'un mot au sommet de la pile	**Transfert d'adresses**	
XCHG	Echange d'octets ou de mots	**LEA**	Chargement d'une adresse effective
XLAT ou XLATB	Traduction d'un octet à l'aide d'une table	**LDS**	Chargement d'un pointeur utilisant DS
		LES	Chargement d'un pointeur utilisant ES
Transfert des flags			
LAHF	Transfert des 5 flags bas dans AH		
SAHF	Transfert de AH dans les 5 flags bas		
PUSHF	Sauvegarde des flags sur la pile		
POPF	Restauration des flags à partir de la pile		

Instructions arithmétiques			
Addition		**Soustraction**	
ADD	Addition d'octets ou de mots	**SUB**	Soustraction d'octets ou de mots
ADC	Addition d'octets ou de mots avec retenue	**SBB**	Soustraction d'octets ou de mots avec retenue
INC	Incrémentation de 1 d'un octet / d'un mot	**DEC**	Décrémentation de 1 d'un octet ou d'un mot
AAA	Ajustement ASCII de l'addition	**NEG**	Complémentation à 2 d'un octet ou d'un mot
DAA	Ajustement décimal de l'addition	**CMP**	Comparaison d'octets ou de mots
Division		**AAS**	Ajustement ASCII de la soustraction
DIV	Division non signée d'octets ou de mots	**DAS**	Ajustement décimal de la soustraction
IDIV	Division signée d'octets ou de mots	**Multiplication**	
AAD	Ajustement ASCII de la division	**MUL**	Multiplication non signée d'octets ou de mots
CBW	Conversion d'octet en mot	**IMUL**	Multiplication signée d'octets ou de mots
CWD	Conversion de mot en double mot	**AAM**	Ajustement ASCII de la multiplication

Instructions logiques			
Logique		**Décalages**	
NOT	Complément à 1 d'un octet ou d'un mot	**SHL/SAL**	Décalage à gauche arithmétique ou logique (octet ou mot)
AND	ET logique de deux octets / deux mots	**SHR**	Décalage logique à droite d'un octet ou d'un mot
OR	OU logique de deux octets / deux mots	**SAR**	Décalage arithmétique à droite d'un octet ou d'un mot
XOR	OU exclusif logique de 2 octets / 2 mots		
TEST	Comparaison, à l'aide d'un ET, d'octets ou de mots		
Rotations			
ROL	Rotation à gauche d'un octet / d'un mot		
ROR	Rotation à droite d'un octet ou d'un mot		
RCL	Rotation à gauche incluant CF (octet/mot)		
RCR	Rotation à droite incluant CF (octet/mot)		

Instructions sur les chaînes de caractères			
Préfixes		**Instructions**	
REP	Répétition tant que CX n'est pas nul	**MOVS ou MOVSB/MOVSW**	Déplacement de blocs d'octets ou de mots
REPE ou REPZ	Répétition tant qu'il y a égalité et que CX n'est pas nul	**CMPS ou CMPSB/CMPSW**	Comparaison de blocs d'octets ou de mots
REPNE ou REPNZ	Répétition tant qu'il n'y a pas égalité et que CX n'est pas nul	**SCAS ou SCASB/SCASW**	Exploration d'un bloc d'octets ou de mots
		LODS ou LODSB/LODSW	Transfert d'un octet ou d'un mot dans AL ou AX
		STOS ou STOSB/STOSW	Chargement d'un bloc d'octets ou de mots par AL ou AX

Instructions de branchements				
Branchements inconditionnels			**Contrôles d'itérations**	
CALL	Appel de procédure		**LOOP**	Bouclage tant que CX _= 0
RET	Retour d'une procédure		**LOOPE ou LOOPZ**	Bouclage tant que CX _= 0 et ZF = 1 égalité)
JMP	Saut inconditionnel		**LOOPNE ou LOOPNZ**	Bouclage tant que CX _= 0 et ZF = 0 (inégalité) JCXZ Saut si CX est nul
Interruptions				
INT	Interruption logicielle			
INTO	Interruption si OF = 1 (overflow)			
IRET	Retour d'une interruption			

Instructions de branchements conditionnels	
Sauts conditionnels	
JA ou JNBE (1)	Saut si « supérieur » (si CF + ZF = 0)
JAE ou JNB (1)	Saut si « supérieur ou égal » (si CF = 0)
JB ou JNAE (1)	Saut si « inférieur » (si CF = 1)
JBE ou JNA (1)	Saut si « inférieur ou égal » (si CF + ZF = 1)
JC	Saut en cas de retenue (si CF = 1)
JE ou JZ	Saut si « égal » ou « nul » (si ZF = 1)
JG ou JNLE (2)	Saut si « plus grand » (si (SF $\oplus$ OF) + ZF = 0)
JGE ou JNL (2)	Saut si « plus grand ou égal » (si SF $\oplus$ OF = 0)
JL ou JNGE (2)	Saut si « plus petit » (si SF $\oplus$ OF = 1)
JLE ou JNG (2)	Saut si « plus petit ou égal » (si (SF $\oplus$ OF) + ZF = 1)
JNC	Saut si « pas de retenue » (si CF = 0)
JNE ou JNZ	Saut si « non égal » ou « non nul » (si (ZF = 0)
JNO	Saut si « pas de dépassement » (si OF = 0)
JNP ou JPO	Saut si « parité impaire » (si PF = 0)
JNS	Saut si « signe positif » (si SF = 0)
JO	Saut si « dépassement » (si OF = 1)
JP ou JPE	Saut si « parité paire » (si PF = 1)
JS	Saut si « signe négatif » (si SF = 1)
(1) concerne des nombres non signés. (2) concerne des nombres signés.	

Instructions de contrôle du 8086				
Opérations sur les flags			**Synchronisation avec l'extérieur**	
STC	Met le flag de retenue à 1		**HLT**	Arrêt du microprocesseur (sortie de cet état par interruption ou reset)
CLC	Efface le flag de retenue		**WAIT**	Attente tant que TEST n'est pas à 0
CMC	Inverse l'état du flag de retenue		**ESC**	Préfixe = instruction destinée à un coprocesseur
STD	Met le flag de direction à 1 (décrémentation)		**LOCK**	Préfixe= réservation du bus pour l'instruction
CLD	Met le flag de direction à 0 (incrémentation)		**Pas d'opération**	
STI	Autorise les interruptions sur INTR		**NOP**	Pas d'opération
CLI	Interdit les interruptions sur INTR			

Annexe 2 : Jeu d'instructions du 8086

Notes :

- AL = 8-bit accumulateur
- AX = 16-bit accumulateur
- CX = Count register
- DS = Data segment
- ES = Extra segment
- Above/Beow refers to unsigned value
- Greater = more positive
- Less = less positive (more negative) signed values
- If d = 1 then «to» reg ; if d = 0 then from reg
- If w = 1 then word instruction ; if w = 0 then byte instruction
- If mod = 11 then r/m is treaded as a REG field
- If mod = 00 then DISP = 0* ; disp-low and disp-high are absent
- If mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
- If mod = 10 then DISP = disp-high ; disp-low
- If r/m = 000 then EA = (BX) + (SI) + DISP
- If r/m = 001 then EA = (BX) + (DI) + DISP
- If r/m = 010 then EA = (BP) + (SI) + DISP
- If r/m = 011 then EA = (BX) + (DI) + DISP
- If r/m = 100 then EA = (SI) + DISP
- If r/m = 101 then EA = (DI) + DISP
- If r/m = 110 then EA = (BP) + DISP*
- If r/m = 111 then EA = (BX) + DISP

DISP follow 2nd byte of instruction (before data if required)
* except if mod = 00 and r/m = 110 then EA = disp-high ; disp-low

- If sw = 01 then 16 bits of immediate data from the operand
- If sw = 11 then an immediate data byte is sign extended to from the 16-bit operand
- If v = 0 then ''count'' = 1 ; if v = 1 then ''count'' in (CL) x = don't care
- Z is used for string primitives comparaison with ZF FLAG

001 reg 110 Segment override prefix

REG is assigned according to the following table ;

16-bit (w=1)	8-bit (w=0)	Segment
000 AX	000 AL	00 AL
001 CX	001 CL	01 CL
010 DX	010 DL	10 DL
011 BX	011 BL	11 BL
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions with reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file :
FLAGS = X :X :X :X (OF) : (DF) : (IF) : (SF) : (ZF) : X : (AF) : X : (PF) : X (CF)

Annexe A3 : Registres spéciaux - SFRs

Ils permettent la gestion du circuit. Certains ont une fonction générale, d'autres un fonction spécifique attachée à un périphérique donné. La Figure* III.3 donne la fonction de chacun des bits de ces registres. Ils sont situés de l'adresse 00h à l'adresse 0Bh dans la banque 0 et de l'adresse 80h à l'adresse 8Bh dans la banque 1. Les registres 07h et 87h n'existent pas.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets (Note 3)
Bank 0											
00h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
01h	TMR0	8-bit real-time clock/counter								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8 bits of the Program Counter (PC)								0000 0000	0000 0000
03h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
07h		Unimplemented location, read as '0'								---- ----	---- ----
08h	EEDATA	EEPROM data register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM address register								xxxx xxxx	uuuu uuuu
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
Bank 1											
80h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	0000 0000
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA data direction register					---1 1111	---1 1111
86h	TRISB	PORTB data direction register								1111 1111	1111 1111
87h		Unimplemented location, read as '0'								---- ----	---- ----
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM control register 2 (not a physical register)								---- ----	---- ----
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged, - = unimplemented read as '0', q = value depends on condition.

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> is never transferred to PCLATH.

2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ reset.

3: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

Description des SFR.

INDF (00h - 80h) : Utilise le contenu de FSR pour l'accès indirect à la mémoire (V.3).

TMR0 (01h) : Registre lié au compteur (VII).

PCL (02h - 82h) : Contient les poids faibles du compteur de programmes (PC). Le registre PCLATH (0Ah-8Ah) contient les poids forts.

STATUS (03h - 83h) : Il contient l'état de l'unité arithmétique et logique ainsi que les bits de sélection des banques (Figure* III.4).

FSR (04h - 84h) : Permet l'adressage indirect (V.3)

PORTA (05h) : Donne accès en lecture ou écriture au port A, 5 bits. Les sorties sont à drain ouvert. Le bit 4 peut être utilisé en entrée de comptage.

PORTB (06h) : Donne accès en lecture ou écriture au port B. Les sorties sont à drain ouvert. Le bit 0 peut être utilisé en entrée d'interruption.

EEDATA (08h) : Permet l'accès aux données dans la mémoire EEPROM.

EEADR (09h) : Permet l'accès aux adresses de la mémoire EEPROM.

PCLATCH (0Ah - 8Ah) : Donne accès en écriture aux bits de poids forts du compteur de programme.

INTCON (0Bh - 8Bh) : Masque d'interruptions (VI).

OPTION_REG (81h) : Contient des bits de configuration pour divers périphériques.

TRISA (85h) : Indique la direction (entrée ou sortie) du port A.

TRISB (86h) : Indique la direction (entrée ou sortie) du port B.

EECON1 (88h) : Permet le contrôle d'accès à la mémoire EEPROM (VIII).

EECON2 (89h) : Permet le contrôle d'accès à la mémoire EEPROM (VIII).

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **IRP**: Register Bank Select bit (used for indirect addressing)
0 = Bank 0, 1 (00h - FFh)
1 = Bank 2, 3 (100h - 1FFh)
The IRP bit is not used by the PIC16F8X. IRP should be maintained clear.

bit 6-5: **RP1:RP0**: Register Bank Select bits (used for direct addressing)
00 = Bank 0 (00h - 7Fh)
01 = Bank 1 (80h - FFh)
10 = Bank 2 (100h - 17Fh)
11 = Bank 3 (180h - 1FFh)
Each bank is 128 bytes. Only bit RP0 is used by the PIC16F8X. RP1 should be maintained clear.

bit 4: **$\overline{TO}$**: Time-out bit
1 = After power-up, `CLRWDT` instruction, or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3: **$\overline{PD}$**: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1: **DC**: Digit carry/$\overline{borrow}$ bit (for `ADDWF` and `ADDLW` instructions) (For $\overline{borrow}$ the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0: **C**: Carry/$\overline{borrow}$ bit (for `ADDWF` and `ADDLW` instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred
Note: For $\overline{borrow}$ the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

Registre d'étai du PIC - STATUS.

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **RBPU**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled (by individual port latch values)

bit 6: **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5: **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4: **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3: **PSA**: Prescaler Assignment bit
1 = Prescaler assigned to the WDT
0 = Prescaler assigned to TMR0

bit 2-0: **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Registre de configuration de périphériques - OPTION_REG.

Bibliographie

- Le microprocesseur INTEL 8086 et ses périphériques (Architecture et programmation, M. T. Benhabiles, A. FAROUKI, T. LAROUSSI, université Mentouri Constantine.
- MICROPROCESSEUR, Support de cours, J.Y. Haggège, Institut Supérieur des Etudes Technologiques de Radès, 2003.
- Architecture (1ère partie), Serge Boada, Ecole d'Ingénieurs du Canton de Vaud, HES.SO Haute Ecole Spécialisée de Suisse Occidentale, Janvier 2003
- Sites d'Internet.

- PIC16F8X, 18-pin Flash/EEPROM 8-Bit Microcontrollers. Documentation technique n° DS30430C du PIC. Microchip Technology Inc, 1998.