2. Organisation logicielle en relation avec l'organisation matérielle détaillée.

Le seul langage compris par l'unité centrale (CPU) est le langage machine. Le programme exécutable est donc obligatoirement, en langage machine. Ce programme est exécuté instruction par instruction, par la CPU à l'aide de ses registres internes.

2.1.) Déroulement du programme.

Chacune des instructions du programme est rangée à une adresse dans la mémoire du système. La CPU doit, par conséquent, gérer l'adressage de cette mémoire. C'est à dire connaître l'adresse de l'instruction à traiter. Ceci est fait grâce au compteur programme (registre PC: Program Counter) et à son circuit d'incrémentation.

Registre "PC"= adresse de l'instruction à exécuter

A la mise sous tension du système, le contenu du registre PC est forcé à une valeur particulière (0000h pour la famille 8051 et les PICs (noté 0x000), chez MOTOROLA (MC6800 à MC68HC11), un vecteur est placé en $FFFE et $FFFF et indique sur deux octets l'adresse de démarrage). La mémoire programme ne peut donc être placée n'importe où dans l'espace adressable.

Un décodeur d'instructions et un séquenceur ("Instruction register" et "Timing and control") permettent l'exécution des séquences élémentaires nécessaires au déroulement de l'instruction.

Une instruction comporte souvent plusieurs octets consécutifs (Dans le cas des microprocesseurs CISC), ou est codée sur 12, 14 ou 16 bits correspondant à la taille du bus d'instructions sur les processeurs RISC (PIC). Le (ou les) premier(s) octet(s) forme(nt) le code opératoire. Celui-ci indique au décodeur d'instructions la tâche qui devra être effectuée et le nombre d'octets composant l'instruction complète. Le (ou les) dernier(s) octet(s) constitue(nt) l'opérande. Pour les processeurs RISC le code opératoire et l'opérande sont codés sur n Bits correspondant à la taille du Bus d'instruction pour pouvoir être exécutés en 1 seul cycle. Dans ce cas le bus d'instruction est obligatoirement plus grand (12 bits pour les PIC 12CXXX). Le nombre de possibilité de codage (instruction et opérande) se trouve donc réduit.

Instruction = code opératoire + opérande

Instruction:	En langage assembleur	En langage machine
Famille 8051	**MOV A,#31h**	74 31
	Code opératoire / Opérande	(codé en hexadécimal sur 2 Octets)
Famille 68HC11	**LDAA #$31**	86 31
Famille PIC (Ex: PIC16F84)	**MOVLW 0x31**	3031 (codé en hexadécimal sur 14 bits)

Attention: *Le codage des nombres différe suivant les familles de microprocesseurs ou microcontroleurs: 0F3h chez INTEL, $F3 chez MOTOROLA, 0xF3 Chez MICROCHIP.*

L'opérande précise le registre ou la donnée à prendre en compte pour réaliser l'instruction. La donnée est définie sous l'une des formes suivantes:

- une valeur (adressage immédiat)
- une adresse où se trouve la donnée (adressage direct ou étendu si l'adresse est sur 16 bits).
- un registre contenant l'adresse de la donnée (adressage indirect ou indexé).

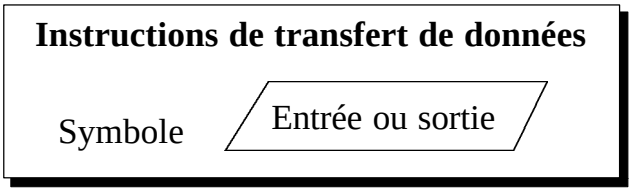
D'une manière générale, la présence ou l'absence d'un opérande, et sa nature (valeur, adresse, registre contenant une adresse) constitue le *mode d'adressage* de l'instruction.

2.2) Les types d'instructions de base et leurs modes d'adressage.

Les différentes instructions de base sont classées en 4 types, en rapport avec les besoins exprimés par l'analyse fonctionnelle logicielle (algorigramme, ordinogramme). Chaque instruction étant souvent disponible dans plusieurs modes d'adressage.

2.2.1) Instructions de transfert de données (présentation des registres accumulateurs).

Ces instructions correspondent principalement aux entrées/sorties de données entre la CPU et les circuits environnants (mémoires et circuits périphériques). La CPU utilise de manière privilégiée un registre appelé accumulateur (A) ou (W = Work pour les PICs). Toutes les manipulations de données sont possibles, en passant par ce registre.



Exemple 1: *(Codage famille 8051).*

MOV A,#31h (en langage machine, exprimé en hexadécimal: 74 31)
Cette instruction signifie charger l'accumulateur A (MOVE TO ACCUMULATOR A) avec la valeur exprimée en hexadécimal 31. Le symbole **#** indique le mode d'adressage *immédiat*: la donnée est exprimée par sa valeur.

MOV A,31h (en langage machine, exprimé en hexadécimal: E5 31)
Cette instruction signifie charger l'accumulateur A, avec la donnée située à l'adresse (exprimée en hexadécimal) 31. Le mode d'adressage est appelé *direct* et n'est représenté par aucun symbole particulier dans le champ de l'opérande.
L'instruction réciproque est **MOV 31h,A** (code machine hexadécimal: F5 31). Il n'existe, bien sûr, aucune instruction réciproque en adressage immédiat !

Dans de nombreuses CPU de µP ou de µC (microcontrôleur), un registre supplémentaire destiné à travailler plus particulièrement avec l'accumulateur A a été implanté. Ce registre s'appelle B et est souvent utilisé avec A dans des opérations arithmétiques (exemple en 8051: **DIV AB** [A4], **MUL AB** [84]. Le mode d'adressage de ces instructions est appelé *par registre*).

Exemple 2: *(Codage famille 68HC11).*

LDAA #$30 signifie charger l'accu A (LOAD ACCUMULATEUR A) avec la valeur exprimée en héxadécimal $30. (LDAA est la partie appelée mnémonique correspondant au

code opératoire, le symbôle # indique le mode d'adressage (ici mode immédiat (en abrégé: imm)-> l'opérande correspond à une valeur).

LDAA $4000 signifie charger l'accu A, avec la donnée située à l'adresse exprimée en héxadécimal $4000. L'opérande correspond alors à une adresse. Le mode d'adressage est appelé direct étendu (ou simplement étendu (en abrégé: ext=extended)) et n'est représenté par aucun symbôle avant l'opérande.

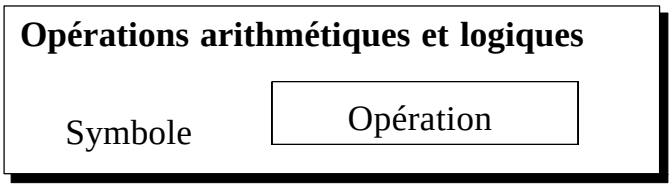
Attention: ***LDAA $30*** signifie donc charger A avec la donnée située à l'adresse $0030.

L'instruction ***STAA*** (inverse de LDAA) store accu A, n'est pas possible en mode immédiat, l'opérande devant obligatoirement indiquer une adresse pour sauvegarder le contenu de A. donc par ex: ***STAA $4000***

L'instruction TAB permet de recopier le contenu de A dans B (Transfert de A dans B). Cette instruction ne necessite aucun opérande. Le mode d'adressage est appelé implicite ou inhérent (abrégé: imp ou inh).

2.2) Opérations arithmétiques ou logiques (rôle de l'ALU et du registre d'état).

Ces opérations sont réalisées par une unité arithmétique et logique (UAL ou ALU "Arithmetic & Logic Unit"). Les deux nombres nécessaires sont en principe constitués par l'accumulateur A et par un registre (B, par exemple) ou par une donnée. Le résultat est placé pour la plus part des opérations dans l'accumulateur A. Un registre d'état (registre PSW: Program Status Word, pour le µC 8051) donne des renseignements sur l'opération effectuée ou sur le résultat (exemple: dépassement de capacité, résultat nul, négatif....).



Exemple 1: *(Codage famille 8051)*

ADD A,#0BAh (en langage machine, exprimé en hexadécimal: 24 BA)
Cette instruction réalise l'addition (ADDition) du contenu de l'accumulateur A avec la valeur exprimée en hexadécimal BA. Le résultat de l'opération est placé dans l'accumulateur A, qui voit son contenu modifié. La présence du zéro devant la valeur BA, dans l'opérande, est obligatoire et permet à l'assembleur de différencier une valeur exprimée en hexadécimal commençant par une lettre (A à F) d'une étiquette appelée BAh !

Les principales autres opérations arithmétiques sont:

- l'addition avec retenue (ADDC, ADDition with Carry)
- la soustraction avec retenue (SUBB, SUBtract with Borrow)
- la multiplication (MUL, MULtiplication) et la division (DIV, DIVision)
- l'incrémentation (INC, INCrementation) et la décrémentation (DEC, DECrementation)

Certaines de ces opérations arithmétiques sont utilisables avec plusieurs modes d'adressages (par registre, direct, immédiat ou indirect).

ANL 31h,A (en langage machine, exprimé en hexadécimal: 52 31)
Cette instruction réalise un ET logique (ANd Logic) entre la variable située à l'adresse (exprimée en hexadécimal) 31 et le contenu de l'accumulateur A. Le résultat de l'opération est placé à l'adresse hexadécimale 31.

Les principales autres opérations logiques sont:

- le OU logique (ORL, OR Logic)
- le OU exclusif (XRL, eXclusive oR Logic)
- la complémentation (CPL, ComPLementation)
- les rotations de bits (RL, Rotate Left; RR, Rotate Right...)

Certaines de ces opérations logiques sont également, utilisables avec plusieurs modes d'adressages (par registre, direct, immédiat ou indirect).

Remarque: Les instructions ET logique et OU logique permettent le forçage de tout ou partie des bits de la variable manipulée. Le ET logique permet le forçage de bit(s) par un 0, alors que le OU logique permet le forçage de bit(s) par un 1. La fonction ainsi réalisée est appelée *masquage de bit(s)*.

Exemple 2: *(Codage famille 68HC11).*

ex: ***ANDA #$01*** fait un ET logique de A avec la valeur $01 (force à 0 tous les bits de D1 à D7). Le résultat étant dans A.

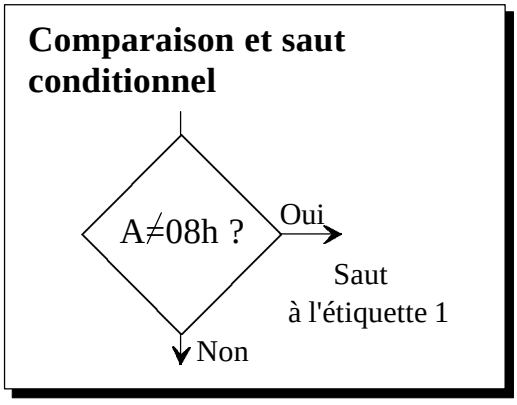
Il sera donc possible d'utiliser un certain nombre d'opérations arithmétiques et / ou logiques avec A ou B, et souvent avec plusieurs modes d'adressages. (ADD=addition, SUB = soustraction, MUL = multiplication, AND = ET logique, OR, EOR= ou exclusif, COM = complément, LSL et LSR= décalage à gauche ou à droite, ROL et ROR= rotation,...etc).

ex: ***ADDA $4000*** signifie ajouter à A le contenu de la case mémoire d'adresse $4000.

2.3) Prise de décision (comparaisons et sauts conditionnels).

Les instructions de ce type permettent d'effectuer des traitements différents (saut dans le programme) en fonction du résultat d'une comparaison entre deux données ou d'une condition sur les indicateurs du registre d'état (indicateur = bit du registre PSW chez INTEL ou CCR = code condition register chez MOTOROLA, registre STATUS sur les PICs).

Lorsque la condition est vérifiée (condition vraie), le saut est effectué.



Exemple 1: *(Codage famille 8051).*

CJNE A,#08,LABEL (en langage machine hexadécimal: B4 08 dépl.)
Cette instruction réalise la comparaison du contenu de l'accumulateur A avec la valeur 08 et le saut, à l'étiquette LABEL, si ce contenu est différent de 08 (Compare & Jump if Not Equal). Dans le cas contraire, le traitement suit son cours normal.

L'écriture en assembleur (écriture sous forme mnémonique) permet d'indiquer les sauts à l'aide d'étiquettes. Le logiciel d'assemblage (Assembleur) calcule alors automatiquement le déplacement nécessaire pour effectuer le saut. Ce déplacement correspond au nombre d'octets à sauter pour trouver la première instruction à exécuter (lorsque la condition est vérifiée). Il est exprimé en code complément à 2 sur un octet, afin de pouvoir représenter un saut négatif (recul) ou un saut positif (avance). Le mode d'adressage de ces instructions est appelé *relatif*.

JC LABEL1 (en langage machine, exprimé en hexadécimal: 40 dépl.)
Cette instruction réalise le saut à l'étiquette LABEL1, si l'indicateur C (Carry) du registre d'état est égal à 1, sinon le programme suit son cours normalement (Jump if Carry flag is set).

Exemple 2: *(Codage famille 68HC11).*

Rem: Lorsque la condition est vérifiée, le branchement (saut) sera effectué.

CMPA #$08 ;compare A avec la valeur $08

BEQ Etiquette1 ;saute à l'Etiquette1 si A est égal à $08

(Suite du traitement si A différent de $08)

Rem: BEQ correspond à Branch if EQual to 0.

Attention: L'écriture en assembleur (écriture sous forme mnémonique) permet d'indiquer les sauts à l'aide d'étiquettes.

L'assembleur calculera alors automatiquement le code de l'opérande pour effectuer le saut. Ce code correspond au nombre d'octets à sauter pour trouver la première instruction à exécuter (lorsque la condition est vérifiée). Il sera exprimé en complément à 2, afin de pouvoir représenter un saut négatif (recul). (voir plus loin pour le détail). Le mode d'adressage est alors appelé relatif. Ce mode d'adressage est particulier aux branchements. Le 68HC11 propose un ensemble complet d'instructions de branchement conditionnel (<, >, < ou =, > ou =, ...) pour des nombre signés ou non.

Exemple 3: *(Codage famille PIC).*

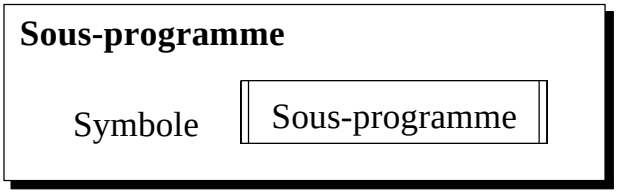
Le PIC de par son jeu d'instructions réduit est le plus pauvre et ne propose que: (DECFSZ = "*Decremente F Skip if Zero*", INCFSZ = "*Incremente F Skip if Zero*", BTFSC = "*Bit Test F Skip if Clear*" et BTFSS = "*Bit Test F Skip if Set*".

Il faut donc se référer au registre d'état en utilisant au paravant une instruction permettant le positionnement d'un indicateur (Z, C,...)

Ex: ***BTFSC STATUS,Z*** Test du bit Z du registre (File) STATUS et saut de l'instruction suivante si Z = 0 (Clear). L'instruction suivante doit en principe être un ***GOTO ETIQUETTE***

2.4) Sous-programmes (la pile et son pointeur "SP").

Un sous-programme permet l'utilisation d'une même tâche sans écrire plusieurs fois les instructions, constituant cette tâche, dans le programme. Afin d'autoriser des appels successifs ou des appels imbriqués de sous-programmes, il est nécessaire d'utiliser une zone mémoire pour sauvegarder les adresses de retour des sous-programmes. Cette zone est appelée *pile* (stack, en anglo-saxon). De plus un registre, appelé *pointeur de pile* (registre SP: Stack Pointer) gère la pile. (Voir détail dans description des sous-programmes).



Une instruction d'appel de sous-programme est placée dans le programme appelant, alors que la dernière instruction du sous-programme appelé est une instruction de retour au programme appelant.

Exemple 1: *(Codage famille 8051)*

LCALL NAME (en langage machine, exprimé en hexadécimal: 12 addr.)
Cette instruction est un appel (Long subroutine CALL) au sous-programme repéré par l'étiquette NAME (Nom du sous-programme). L'opérande de l'instruction est l'adresse d'implantation de la première instruction du sous-programme.

RET (en langage machine, exprimé en hexadécimal: 22)
Cette instruction permet le retour (RETurn from subroutine) au programme appelant.

Exemple 2: *(Codage famille 68HC11).*

JSR (jump to subroutine), l'opérande précisera alors l'adresse du sous programme (mode étendu) (ou une adresse par rapport à un registre d'index= mode indéxé).

BSR (branch to subroutine), c'est un appel de sous-programme en mode relatif.

RTS (return subroutine)= retour de sous-programme.

Exemple 3: *(Codage famille PIC).*

CALL (branch to subroutine), c'est un appel de sous-programme en mode relatif.

RETURN (return subroutine)= retour de sous-programme.

3. Utilisation et programmation des périphériques (PIA, ACIA, TIMER, CAN, CNA, afficheur LCD).

Voir polycopié spécifique sur chaque périphérique.

4. Exercices d'application divers.

4.1.) Exercices avec défilement lumineux sur 8 LEDs reliées à un port entrée sortie.

4.2.) Exercice complet sur le Filancemètre (Organisation matérielle et logicielle, utilisation du PIA).

4.3.) Communication série entre deux micro. Contrôle d'une carte "système minimum" à partir d'un programme développé sur PC.

4.4) Lecteur de codes à barres (utilisation PIA, ACIA et d'un afficheur LCD "LTN211".

4.5.) Temporisateur programmable (PIA, TIMER, afficheur).

4.6.) Oscilloscope numérique (PIA, ACIA, TIMER, CAN, CNA, échantillonneur bloqueur, afficheur).

4.1.) Exercices avec défilement lumineux sur 8 LEDs reliées à un port entrée sortie.

a) Programme pour 68HC11 (Clignotement des LEDs paires ou impaires, valeur $55 et $AA).

```
**********Programme ESPORTB1 HC11**********
* Partie à remplir obligatoirement pour apprendre
* à faire une en tête correcte
*
* Société: Lycée Maurice GENEVOIX (INGRE)
* Auteur: Mr COTTET                Service (Classe): TS1
* Date: 15/4/99                    Rev: 1.0
* Programme (Nom du fichier):  ESportB1.asm
* Titre: Essai du port B clignotement des LEDs en sortie du port B
*
* Description des entrées / sorties
* Entrées: Pas d'entrée utilisée
*
* Sorties:
*    PB7  PB6 PB5 PB4 PB3 PB2 PB1 PB0
*     D7   D6  D5  D4  D3  D2  D1  D0
*************************************************************************

* PortB   equ    $1004   ; Constante déja déclarées dans UMPS

**********programmeprincipal**********
       ORG    $F000    ; Début du programme dépend de la vesion de HC11
                       ; correspond au début de la ROM (ou EPROM, EEPROM)
                       ; $F800 pour le HC811E2 (EEPROM)

debut  lds    #$00FF   ; Positionnement de la Pile en RAM ($0000 à $00FF)
depart ldaa   #$55
       staa   PortB    ; Conduction des LEDs paires
       jsr    Tempo    ; Temporisation de durée T0 (à ajuster)
       ldaa   #$AA
       staa   PortB    ; Conduction des LEDs impaires
       jsr    Tempo
       bra    depart   ; Retour à depart

********** Sous Programme de Temporisation **********
Tempo  ldx    #$FFFF
decrem dex
       bne    decrem
       rts

;******** Vecteurs d'interruptions ********
       org    $FFFE
RESET  fdb    debut

       end
```

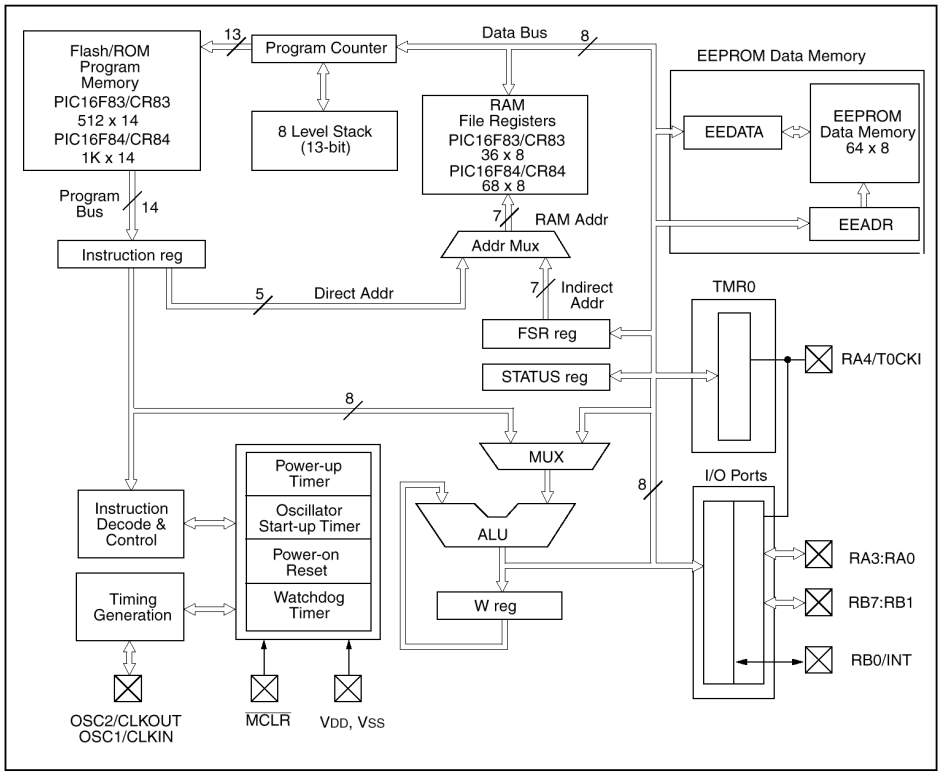
b) Programme pour 68HC11 (animation sur les LEDs suivant une table de valeurs).

```
*************************************************
*  Partie à remplir obligatoirement pour apprendre
*  à faire une en tête correcte
*
*  Société (établissement):
*
*  Nom:                  Service (Classe):
*  Date:                 Rev:
*     Titre du projet et nom du fichier
*     Animation des leds de la carte
*     d'affichage par une table de
*     donnée.           ESPortB3.asm
*
* Description des entrées / sorties
* Entrées:
*    Aucune
* Sorties:
*    PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0
*     D7  D6  D5  D4  D3  D2  D1  D0
*
* Clignottement des LEDs sur le port B
*      entrées: non utilisées
*      sorties: 8 leds sur le port B (1 => allumage)
*************************************************
* PortB   equ $1004         ; Adresse du port B

        org $F800
debut   lds #$00FF; Positionnement de la pile
depart  ldx #Table
        ldaa 0,X
suite   staa PortB
        jsr Tempo
        inx
         ldaa 0,x
        cmpa #0
        bne suite
        bra depart

Tempo   ldy #$0fff; Sous programme de Tempo
decrem  dey
        cpy #$0
        bne decrem
        rts

Table   FCB $01,$03,$07,$0E,$1C
         FCB $38,$70,$E0,$C0,$80
         FCB $00

;********* Vecteurs d'interruption **************
        org   $FFFE
RESET   fdb   debut
        end
```

c) Programme pour PIC16F84 (Clignotement des LEDs, valeur $55 et $AA).

PIC16F8X

FIGURE 3-1: PIC16F8X BLOCK DIAGRAM




```
; Fichier:  es_portb.asmvers: 1.0
; Essai de clignotement des Leds
; sur le port B d'un pic 16F84
        list    p=16f84
        include <p16f84.inc>
temp    equ     0x0c            ; adresse du compteur temp
temp2   equ     temp+1
        org     0x00            ;depart du processeur adr 000
reset   goto    start           ;saut au debut du prog principal
        org     0x04            ;prevu pour interruption
start
;       bsf     STATUS,RP0      ;selectionner bank 1
        banksel TRISB
        clrf    TRISB           ;port B en sortie pour Leds
        bcf     STATUS,RP0      ;selection bank 0
suite
        movlw   0x55
        movwf   PORTB
        call    tempo
        movlw   0xAA
        movwf   PORTB
        call    tempo
        goto    suite
tempo
        movlw   0xFF
        movwf   temp2
tempBC2 movwf   temp
tempBC1 decfsz  temp,f
        goto    tempBC1
        decfsz  temp2,f
        goto    tempBC2
        return
        end
```

d) Programme pour PIC16F84 (animation sur les LEDs suivant une table de valeurs).

```
;*******************************************************
;*  Partie à remplir obligatoirement pour apprendre à faire une en tête
;*  Société (établissement): Lycée Maurice GENEVOIX
;*  Nom: COTTET JJ          Service (Classe): 1STS
;*  Date: 12/02/2002        Rev: 1.0   Version revue pour un PIC 16F84
;*     Titre du projet et nom du fichier   Animation des leds de la carte
;*     d'affichage par une table de  données.             ESPortB3.asm
;*    ;* Description des entrées / sorties
;* Entrées:     Aucune
;* Sorties:    RB7 RB6 RB5 RB4 RB3 RB2 RB1 RB0
;*                  D7  D6  D5  D4  D3  D2  D1  D0
;* Clignottement des LEDs sur le port B
;* suivant une séquence définie en mémoire programme
;*      sorties: 8 leds sur le port B (1 => allumage)
;*******************************************************
          list     p=16f84
          include <p16f84.inc>

;******************** variables RAM **********************
temp      equ      0x0c              ; adresse du compteur temp
temp2     equ      0x0d        ; et temp2 définissant la tempo
pointeur equ       0x0e        ; pointeur dans la table (offset)

;******************** debut programme ********************
          org      0x00        ;depart du processeur adr 000
reset
          goto     start       ;saut au debut du prog principal
          org      0x04        ;prevu pour interruption
;******************** Programme Principal ***************
          org      0x05
start
          call     InitPortB
depart
          movlw    0x00        ;initialisation sur la première
                               ;valeur de la table (offset 0)
suite     movwf    pointeur    ;sauve valeur du pointeur
          call     Table       ;retourne dans w la valeur
          andlw    0xFF        ;positionne indicateur Z si 0
btfsc     STATUS,Z
          goto     depart
          movwf    PORTB
          call     tempo
          movf     pointeur,w
          addlw    1
          goto     suite

;******************** Sous Programmes *******************
InitPortB
;         bsf      STATUS,RP0  ;selectionner bank 1
          banksel TRISB
          clrf     TRISB       ;port B en sortie pour Leds
;         bcf      STATUS,RP0  ;selection bank 0
          banksel PORTB
          return
tempo
          movlw    0xC0
          movwf    temp2
tempBC2   movwf    temp
tempBC1   decfsz   temp,f
          goto     tempBC1
          decfsz   temp2,f
          goto     tempBC2
          return
;******************** Table de constantes ***************
```

```
Table
          addwf    PCL,f
          retlw    1
          retlw    3
          retlw    7
          retlw    0E
          retlw    0x1C
          retlw    0x38
          retlw    0x70
          retlw    0xE0
          retlw    0xC0
          retlw    0x80
          retlw    0

          end
```