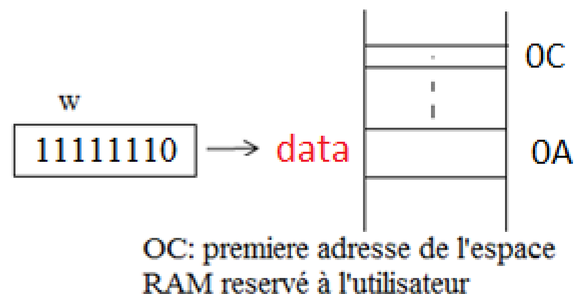
Université Badji Mokhtar – Licence Electronique
Matière : Systèmes à microprocesseurs
Enseignant : Dr S.BENSAOULA

Pic16f84 : Initiation à la programmation assembleur

1. Transfert entre mémoire RAM et le registre accumulateur w

```
valeur   EQU  0xfe   ; valeur est équivalent à 0xfe (en décimale 254, en binaire : 11111110)
data     EQU  0x0a   ; data est équivalent à 0x0a
movlw    valeur      ; charge la constante valeur (254) dans w
movwf    data        ; copie (transfert) le contenu de w dans la position mémoire data
                     ; d'adresse 0a , c'est une ecriture en RAM.
```

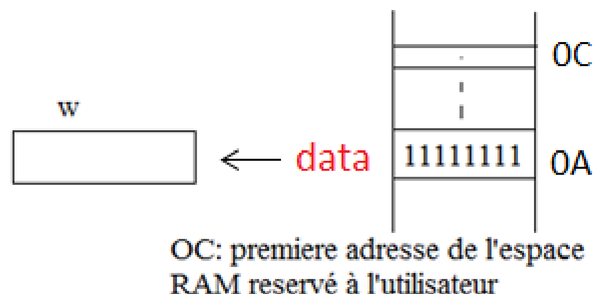
On vient de charger la constante 254 dans la position mémoire **data**
data a pour adresse 0x0A



On poursuit note exemple :

```
incf    data      ; incrementation du contenu de data : 254 +1 = 255

movf    data, w   ; copie le contenu de data dans w
                  ; c'est une lecture en RAM : w contient 255
```



Tous les transferts de données transitent par le registre w

2. Configuration des E/S

La configuration des lignes d'entrées-sorties des ports consiste à fixer le sens de ces lignes :
ligne en entrée : elle transporte une information de l'extérieur vers l'intérieur du pic.
ligne en sortie : elle transmis une information de l'intérieur vers l'extérieur du pic.

Pour configurer le PORTA on utilise le registre TRISA, et pour le PORTB on utilise le registre TRISB.

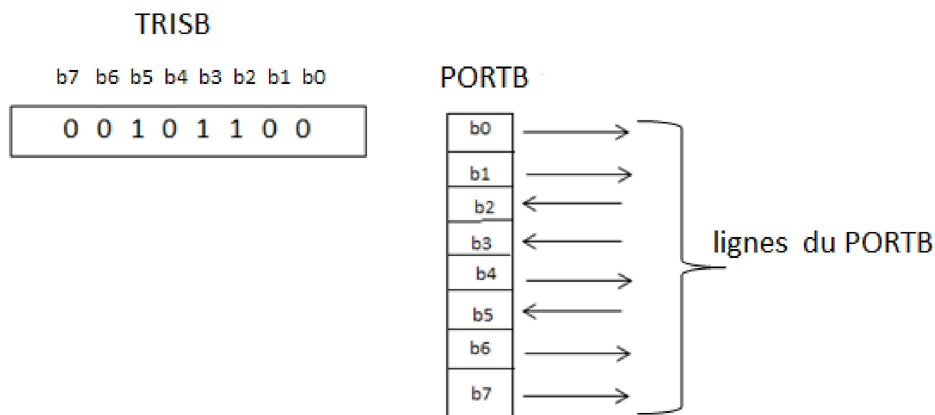
exemple de configuration du PORTB :

1. les registres TRISA et TRISB sont en mémoire dans la bank1
2. pour écrire dans ces deux registres on doit sélectionner la bank1
3. instructions de sélection :

bsf : mise à 1 d'un bit
bcf : mise à 0 d'un bit

```
bsf    STATUS, RP0    ;  mise à  1 du bit RP0  du registre STATUS, sélection de la bank1
movlw    B'00101100'    ; charge la constante '00101100' dans w
movwf    TRISB            ; copie w dans TRISB
bcf         STATUS, RP0    ; mise à 0 du bit RP0, on revient dans la bank0
```

Avec l'instruction **bcf** on revient en bank0 car le PORTA et le PORTB sont en bank0 (on ne peut les utiliser que si on est en bank0).



Configuration du PORTB :

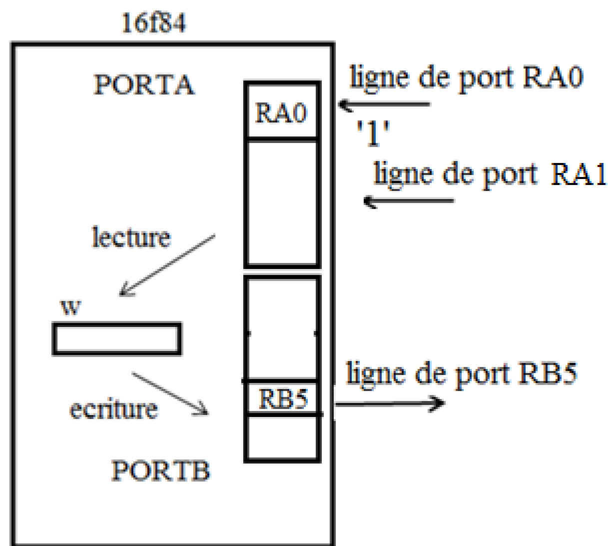
bit du registre TRISB à 0 ligne correspondante du PORTB en sortie
bit du registre TRISB à 1 ligne correspondante du PORTB en entrée

3. Configurer une seule ligne de port

Lorsqu'une seule ligne nous intéresse, on peut la configurer individuellement.

```
bsf    STATUS, RP0   ; passage en bank1
bsf    TRISA, 0          ; mise à 1 du bit 0 de TRISA : ligne RA0 du PORTA en entrée
bcf    TRISB, 5      ; mise à 0 du bit 5 de TRISB : ligne RB5 du PORTB en sortie
bcf    STATUS, RP0   ; revenir en bank0
```

4. Lecture PORTA- On propose le schéma suivant :




```
bsf      STATUS, RP0   ; passage en bank1
movlw    B'00000011'   ; on fixe la valeur des bits
movwf    TRISA         ; on charge w dans TRISA : RA0 et RA1 en entrée
bcf      STATUS, RP0   ; revenir en bank0 car PORTA en bank0
movf     PORTA, w      ; lecture du PORTA : le contenu du PORTA est copié dans w
```

maintenant le bit 0 du registre w est à 1, comme la ligne RA0.

Seules les lignes configurées en entrée seront recopiés (lecture) dans les bits correspondants du registre w : RA0 dans le bit0 et RA1 dans le bit1.

5. Ecriture PORTB

On considère le schéma précédent, et on suppose que la ligne RB5 est à l'état logique 0 (bit RB5 = 0) et w contient 00101100 :

```
bsf      STATUS, RP0 ; passage en bank1
movlw    B'11011101'   ; on fixe la valeur des bits
movwf    TRISB         ; lignes RB5 et RB1 en sortie
bcf      STATUS, RP0   ; revenir en bank0 car PORTB en bank0
movwf    PORTB         ; écriture dans le PORTB
```

maintenant la ligne RB5 est à 1, comme le bit 5 du registre w

Seuls les bits 1 et 5 du registre w seront recopiés (écriture) dans les bits respectivement 1 et 5 du PORTB.

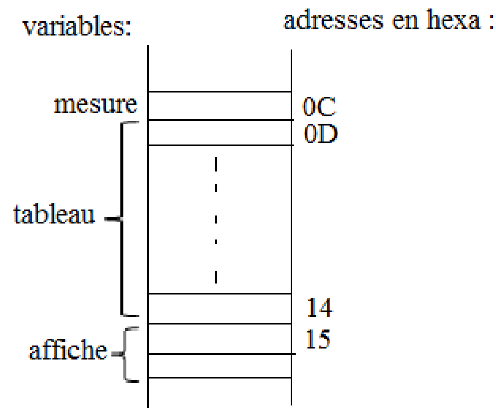
6. Réservation d'espace mémoire

Les variables suivantes sont utilisées dans un programme assembleur :

tableau :   occupe 8 octets
affiche :   //   2 octets
mesure :   //   1 octet

Ces variables vont occuper un espace mémoire en RAM. Dans un programme assembleur on réserve un espace mémoire comme suit :

```
cblock          0x0C            ; c'est le début de l'espace utilisateur
        mesure : 1
        tableau : 8
        affiche : 2
endc
```



La variable **mesure** a pour adresse 0x0C : occupe un seul octet
La variable **tableau** a pour adresse 0x0D : occupe l'espace mémoire de l'adresse 0x0D jusqu'à l'adresse 0x14 : soit 8 octets
La variable **affiche** a pour adresse 0x15 : occupe 2 adresses, 0x15 et 0x16.

7. Corps d'un programme

Nous allons voir à travers un exemple les parties constituants un programme assembleur.
Le programme suivant allume une led branchée sur la ligne RB7.

```
        list            p=16f84         ; déclaration que c'est le pic 16f84 qui est utilisé
        # include       <p16F84.inc>    ;  appelle du fichier qui contient les définitions
                                        ; des différentes constantes propres au pic 16f84
tab     EQU   0x0f                      ; si on a plusieurs variables on emploie cblock

                org     0x00            ; l'adresse de départ après un reset c'est 0x00
; partie initialisation
      bsf       STATUS, RP0     ; 1er instruction logée à l'adresse 0
      movlw   0
```

```
        movf    TRISB           ; PORTB en sortie
        bcf     STATUS, RP0
        clrf    PORTB           ;  remise à zéro du PORTB
; le programme principale
loop    bsf     PORTB, 0
        goto    loop            ; le programme  reboucle,  la led reste indéfiniment allumée.
        end                     ; fin du  programme
```

-Question : Tracer le schéma correspondant à ce programme

Remarque

Les directives : **list**, **include**, **org** et **end** sont obligatoires dans un programme assembleur. ***Ce ne sont pas des instructions : elles ne sont pas exécutées par le pic***. Elles sont utiles uniquement lors de la compilation (génération fichier exécutable).

8. Masquage de bits

Le masquage (ou forçage) de bits consiste à changer l'état logique d'un ou plusieurs bits d'un octet. On peut forcer l'état d'un bit soit à 0 ou à 1.

1. Masquage à 0

Soit la variable a = 11101111

On veut forcer à 0 tous les bits sauf le bit de poids 0

Pour cela on se donne un masque : 00000001

```
     1110 1111  : a
ET   0000 0001  : masque
    --------------
     0000 0001
```

Le masquage à 0 consiste à faire un ET logique bit-à-bit

Soit une variable Y = xxxxxxxx : x = 0 ou 1

On veut forcer à 0 le bit 4, pour cela on se donne le masque : 11101111

```
xxxx  xxxx
1110 1111
-------------
xxx0  xxxx
```

Tous les bits restent inchangés, sauf le bit 4 a été forcé à 0.

Les instructions du pic 16f84 qui réalisent ce masquage sont :

1) **andlw** : réalise un ET logique entre le contenu du registre w et une constante (le masque)

exemple: andlw B'11101111' ; ET logique entre w et le masque : le résultat dans w

2) **andwf :** réalise un ET logique entre le contenu du registre w et une variable

exemple :

```
andwf  var, f    ; ET logique entre w et var : le résultat dans var
andwf  var, w    ; ET logique entre w et var : le résultat dans w
```

2. Masquage à 1

Soit la variable a = 10010100

On veut forcer à 1 le bit 6 seulement

Pour cela on se donne un masque : 01000000

```
OU  10010100  : a
    01000000  : masque
    --------
    11010100
```

Le masquage à 1 consiste à faire un OU logique bit-à-bit

Soit une variable Z = xxxxxxxx

On veut forcer à 1 le bit 2, pour cela on se donne le masque : 00000100

```
x x x x x x x x
0 0 0 0 0 1 0 0
---------------
x x x x x 1 x x
```

Tous les bits restent inchangés, sauf le bit 1 a été forcé à 1.

Les instructions du pic 16f84 qui réalisent ce masquage sont :

1) **iorlw :** réalise un OU logique entre le contenu du registre w et une constante (le masque)

exemple: iorlw B'00000100' ; OU logique entre w et le masque : le résultat dans w

2) **iorwf** : réalise un OU logique entre le contenu du registre w et une variable

exemple :

```
iorwf  var, f    ; OU logique entre w et var : le résultat dans var
iorwf  var, w    ; OU logique entre w et var : le résultat dans w
```

Exercices corrigés

(w) : signifie le contenu de w :   (w) = 0x0f : alors w contient 15 $(00001111)_2$

Ex1

Soit le programme :

```
var     EQU     0x0d
                org  0x00
        movf     PORTB, 0   ; copie le PORTB dans w
        movwf    0x0d
        movlw     var
        incf      var
        addwf     var, w    ; faire (var) + (w), et place le résultat dans w : c'est l'addition
        end
```

Questions

1. Si le PORTB contient 10 (valeur décimale), quel est le résultat final dans w en décimale. **(R : 24)**
2. Quelle est l'adresse de l'instruction addwf **(R : 0x04)**
3. Sur combien de bits est codée une instruction du pic 16f84 **(R : 14bits)**
4. Sur combien de bits est codée une adresse de la mémoire flash **(R : 13bits)**

Ex2

On demande de compléter le remplissage du tableau suivant en indiquant le contenu de chaque variable (port, registre, etc.) après l'exécution de l'instruction correspondante.

Données initiales : (PORTA) = 0, (w) = 0, (var) = 0, (temp) = 0x0F

Réservation mémoire

```
        cblock   0x0C
                temp : 1
                var   : 1
        endc
```

Instruction	PORTA	w	var
1 : movlw var	0	0D	0
2 : movwf var	0	0D	0D
3 : movf temp,w	0	0F	0D
4 : movwf PORTA	0F	0F	0D
5 : movwf var	0F	0F	0F
6 : addwf var,w	0F	1E	0F

Commentaires :
D'après la réservation, **var** a pour **adresse** 0x0D : le mot var est équivalent à 0D

1. movlw var : charge la constante équivalente à var (0D) dans w, donc (w) = 0D

Les contenus du PORTA et de la position mémoire var restent inchangés

2. dans l'instruction 2 on travaille avec les résultats de 1, ainsi de suite.

Tableau : adressage indirect

Ex3 – Ecriture en RAM

Le pic 16f84 exécute la lecture de 10 valeurs sur le PORTB. Ces valeurs sont stockées en mémoire RAM à partir de l'adresse 0x30. Ecrire le programme assembleur qui s'exécute une seule fois.

Corrigé

Nous avons 10 valeurs à lire et à stocker en RAM à partir de l'adresse 0x30.
On a donc un tableau définit par sa taille (n=10) et son adresse 0x30.
Le remplissage de ce tableau est réalisé à l'aide de l'adressage indirect.
Pour tester la fin de l'opération on utilise un **compteur**, initialisé avec n=10, ensuite après chaque lecture d'une donnée on décrémente ce compteur. Lorsque ce compteur atteint 0, c'est la fin de l'opération.

Programme

```
        list        p=16f84             ; définition de processeur
        #include   <p16F84.inc>         ; définitions de variables
count   EQU  0x0c     ; déclaration variable compteur

                org     0x00            ; adresse de départ après reset
        bsf         STATUS, RP0         ; 1er instruction logée à l'adresse 0
        movlw   0xff
        movwf   TRISB                   ; port en entrée
        bcf         STATUS, RP0
        movlw   .10
        movwf   count           ; initialise compteur avec 10
        movlw   0x30
        movwf   FSR             ; initialise pointeur avec l'adresse du tableau
loop
        movf    PORTB, 0        ; lecture de la donnée
        movwf   INDF            ; écriture de la donnée en RAM : adressage indirect
        incf    FSR             ; incrémente  pointeur
        decfsz  count           ; décrémente compteur : count -1 et test si count = 0
        goto    loop            ; counter ≠ 0 :  retour
        end                     ; counter = 0 : on sort
```

Ex4 –Lecture en RAM

Le pic 16f84 réalise la lecture de 12 valeurs en mémoire rangées à partir de l'adresse 0x0c. Chaque valeur lue est envoyée sur le PORTB. Ecrire le programme assembleur qui réalise cette opération une seule fois.

Corrigé

On a un tableau définit par sa longueur n=12 et son adresse 0x0c.
On doit donc définir un compteur pour détecter la fin des lectures dans le tableau, et utiliser l'adressage indirect pour la lecture.

Programme

```
        list        p=16f84A
        #include   <p16f84A.inc>

count   EQU  0x40     ; déclaration variable compteur

        org      0x00
        bsf      STATUS, RP0
        clrf     TRISB                ; port en sortie
        bcf      STATUS, RP0
        movlw    .12
        movwf    count
        movlw    0x0c
        movwf    FSR          ; initialisation pointeur
suite   movf     INDF, w      ; lecture donnée en RAM : adressage indirect
        movwf    PORTB        ; sortie donnée sur le port
        incf     FSR          ; incrémente  pointeur
        decfsz   count        ; décrémente count et test si count = 0
        goto     suite        ; count ≠ 0 : retour
        end                   ; count = 0 : on quitte
```

Ex5 : masquage de bits

Le pic 16f84 exécute la lecture de 10 valeurs sur le PORTB. On force à 0 le bit7 et à 1 le bit0 de chacune de ces valeurs avant de l'enregistrer en mémoire. L'adresse de début de chargement est 0x30. Ecrire le programme assembleur.

Corrigé

Nous avons un forçage à 0 du bit 7 donc on utilise le masque 01111111, et un forçage à 1 du bit 0, on utilise le masque 00000001.
Les données lues sur le PORTB sont traitées (masquage) ensuite stockées dans un tableau d'adresse 0x30 et de taille 10 en utilisant l'adressage indirect.

Programme

```
         list   p=16f84                    ; définition de processeur
        #include   <p16F84.inc>            ; définitions de variables

count   EQU  0x0c     ; déclaration variable compteur

                org      0x00              ; adresse de départ après reset
        bsf          STATUS, RP0           ; 1er instruction logée à l'adresse 0
        movlw    0xff
        movwf    TRISB                     ; port en entrée
        bcf         STATUS, RP0
        movlw   .10
        movwf   count              ; initialise compteur avec 10
        movlw   0x30
        movwf   FSR                ; initialise pointeur avec l'adresse du tableau
loop
        movf     PORTB, 0          ; lecture de la donnée
        andlw    B'01111111'       ;  bit7 à 0 : résultat  dans w
        iorlw     B'00000001'      ;  bit0 à 1 : résultat  dans w
        movwf    INDF              ; écriture de la donnée en RAM : adressage indirect
        incf        FSR            ; incrémente  pointeur
        decfsz    count            ; count -1 et test si count =0
        goto        loop           ; count ≠ 0 :  retour
        end                        ; count = 0 : on sort
```

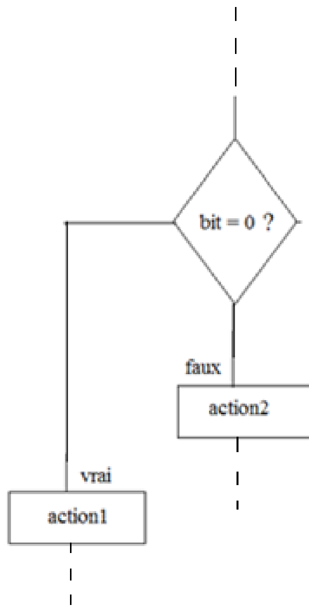
Ex6- Lecture interrupteur et bouton poussoir

Les instructions btfsc et btfss

Ces deux instructions vont servir pour lire l'état logique d'un interrupteur ou d'un bouton poussoir. Elles servent aussi pour la lecture de l'état logique d'une ligne de port configurée en entrée. Ces deux instructions exécutent un **test** de bit suivi d'un **saut si ce test est vrai.**

btfsc : « **b**it **t**est **f**ile **s**kip if **c**lear » : test le bit et saut si ce bit est 0
btfss : « **b**it **t**est **f**ile **s**kip if **s**et » : test le bit et saut si ce bit est 1

On présente le fonctionnement de btfsc, pour l'autre instruction c'est le même mécanisme. Le fonctionnement de btfsc est expliqué par l'organigramme suivant :



<u>Exemple : **btfsc**</u>

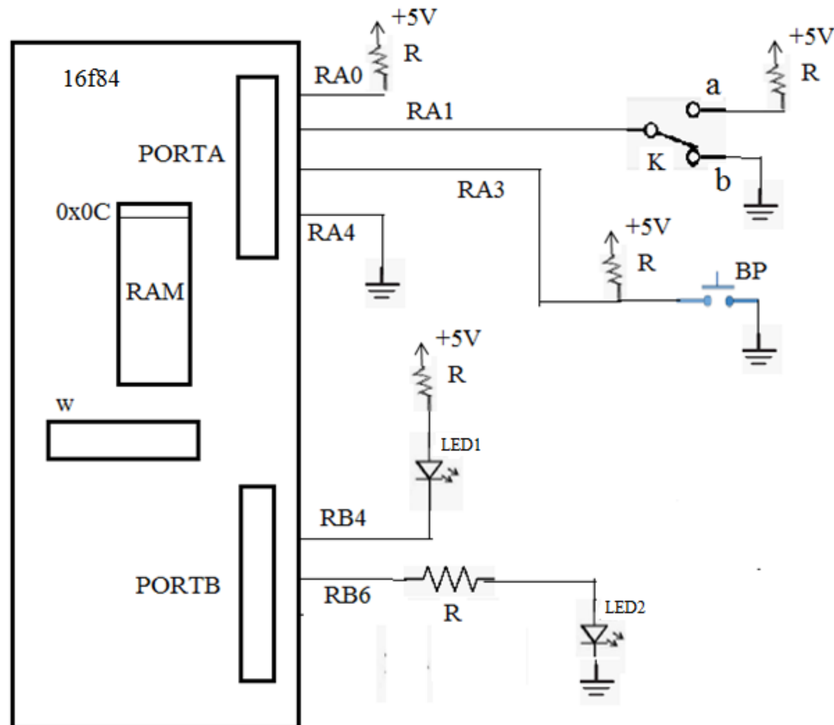
```
début     ............
          ...........
          ............
          ...........
          btfsc  PORTB, 7    ;  test du bit 7 du PORTB : RB7
          goto   action1     ;  RB7=1 : saut vers action1
          goto   action2     ;  RB7=0 : saut vers action2
          ............
          ...........
action1   ...........
          ...........
          goto  début
action2   ...........
          ...........
          goto  début
```

<u>Exemple : **btfss**</u>

```
début     ............
          ...........
          ............
          ...........
     btfss   PORTA, 0   ;   test du bit 0 du PORTA : RA0
     goto   suite1   ;   RA0 = 0 : saut vers  suite1
     goto   suite2   ;   RA0 = 1 : saut vers suite2
     ............
     ...........
```

Exemple d'utilisation

Soit le schéma suivant:



On veut tester la ligne RA1 de l'interrupteur K.

K = 0 : éteindre LED1

K = 1 : allume LED1

Le programme tourne indéfiniment : son exécution ne s'arrête pas

Lorsqu'on commande des composants qui ont une consommation élevée de courant, on préfère utiliser le PORTB à cause du courant qu'il est capable de débiter.

Sur le schéma de la figure les lignes du PORTA et du PORTB ont pour fonctions :

RA0 : état logique '1', car branchée au 5V, sens : **entrée**

RA1 : branchée à l'interrupteur K, sens : **entrée**

- si K est dans la position **a** : la ligne est à l'état logique '1'
- si K est dans la position **b** : la ligne est à l'état logique '0'

RA3 : branché au bouton poussoir BP, sens : **entrée**

RA4 : état logique '0', car branchée à la masse, sens : **entrée**

Ces lignes sont en entrée car chacune d'elles est la **source** d'une information

RB4 : commande une led : LED1 sens : **sortie**

RB6 : commande une autre led : LED2 sens : **sortie**

Lorsqu'une ligne commande un actionneur (moteur, pompe, etc.) ou un composant dissipateur de puissance (lampe, résistance de puissance, etc.), elle est configurée en sortie.

Principe du bouton poussoir BP

Au repos BP est ouvert, la ligne est branchée au 5V, donc état haut. Une action sur BP ferme le contact momentanément (quelques ms), la ligne passe à l'état bas. Lorsque BP est relâché la ligne repasse à l'état haut. Il existe des boutons poussoirs qui sont fermés à l'état de repos.

Programme

```
                list   p=16f84
                #include   <p16F84.inc>
mavariable   EQU   0x0C    ; on déclare mavariable équivalent  à 0C

                        org       0x00            ; adresse de départ après reset
; initialisation  des lignes de port
             bsf       STATUS, RP0       ;  passage en bank1
             movlw   B'00011011'
             movwf   TRISA                ; RA0, RA1, RA3 et RA4  entrées
             movlw   B'00000000'
             movwf   TRISB               ; PORTB  en sortie,
             bcf       STATUS, RP0       ;  retour en bank0

             movf    PORTA, w          ; lecture état interrupteur K et bouton poussoir BP
             movwf   mavariable        ; sauvegarde état  en  mémoire

encore       btfsc   PORTA, 1      ; bit RA1= 0 ? (K sur la position b?)
             goto    action2          ;  RA1= 1 (K sur la position a) : aller à action2
             goto    action1          ;  RA1= 0 (K sur la position b) : aller à action1

action2      bcf     PORTB, 4       ; RB4 à 0 : LED1 ON
             bcf     PORTB, 6       ;  RB6 à 0 : LED2  OFF
             goto  encore          ; on revient au début : car l'exécution ne s'arrête pas

action1      bsf    PORTB, 4     ; RB4 à 1 : LED1  OFF
             bsf    PORTB, 6     ;  RB6 à 1 : LED2  ON
             goto  encore
             end
```

Remarque

Les noms communs dans le programme: **encore**, **action1**, **action2**, sont appelés **étiquettes**. Il faut éviter de choisir des noms propres (personne, ville, etc.)

Travail personnel

Ecrire le programme qui teste le commutateur K en utilisant l'instruction btfss :

si K = 1 : allume LED2

si K = 0 : éteindre LED2