CH-1092 Belmont, Switzerland
info@didel.com, www.didel.com

PIC*Génial* - Microcontrôleur PIC 16F84/16F870

Introduction à la programmation avec le PIC*Génial* - suite

2. Programmation du PIC 16F84/16F870

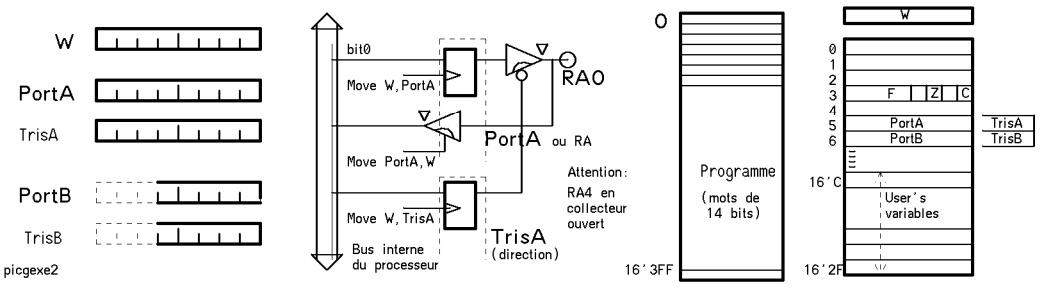
L'architecture, le répertoire d'instruction et les périphériques internes du PIC seront expliquées en plusieurs fois, pour éviter des longs commentaires et des listes rébarbatives d'instructions.

2.1. Un premier modèle simplifié du PIC

Le PIC 16F84/16F870 peut être vu en première approche comme un registre W, deux ports A et B, leurs registres de direction associés. Le portA est à l'adresse 5 et le portB à l'adresse 6 (l'assembleur le sait, il n'y a pas besoin de le lui dire). Les registres de direction sont dans une 2e banque, mais peuvent être accédés directement. Le PIC 16F84/16F870 a heureusement des instructions spéciales pour les accéder. Avec CALM, les registres de direction s'apellent TrisA, TrisB (et TrisC pour le 16F870) et l'instruction Move permet de transférer la valeur préparée dans W. On ne peut par contre pas lire le contenu ou faire des opérations comme avec les autres registres et variables. Un 0 dans un bit de direction met la ligne correspondante en sortie (truc mnémotechnique: 0 -> O out 1 -> I In).

Lorsque le processeur exécute par exemple l'instruction "Move W,PortA", une impulsion charge le registre appelé PortA ou RA (register A). Les sorties qui ont été préparées en sortie copient cette information. W --> registre interne --> sorties. Si on relit avec un "Move PortA,W", on lit directement l'état des broches. Le passeur qui met en communication la broche avec le bus interne, lui-même relié au registre W, n'est activé que pendant quelques dizaines de nanosecondes. C'est une photo instantanée de l'état des broches du port; si la tension sur une broche est inférieure à 0.7V, le fabricant garantit qu'un zéro sera enregistré dans W. Si c'est supérieur à 2V (alimentation 5V), c'est un 1. Entre les deux, ce sera 0 ou 1. La frontière dépend du circuit et de sa température.

A partir de la position 16'C, l'utilisateur peut placer ses variables, compteurs, bits d'état, etc. Il faut passer par le registre W (Work register) pour initialiser une variable à une valeur différente de zéro. Le résultat des opérations entre W et un registre n'est pas nécessairement dans W; la destination peut être le registre, ce qui est souvent très efficace.



picgexe2

Fig. 1 Modèle pour les ports du PIC

Le registre F (flags, état) mémorise en particulier deux bits que l'on trouve dans tous les processeurs: C est le "Carry", activé par une addition avec dépassement de capacité, ou un décalage. Z est le "zéro bit", activé si le résultat d'une opération est nul (les 8 bits du résultat transféré dans W ou dans un registre sont nuls). Toutes les instructions n'agissent pas sur Z et sur C: la feuille de codage verte précise pour chaque intruction les bits modifiés.

Le programme est dans une mémoire séparée (architecture "Harvard"). Le processeur démarre en 0 (nous parlerons des interruptions plus tard) et exécute chaque instruction en 1 microseconde (à 4MHz), sauf les sauts qui "cassent le pipeline" et demandent deux microsecondes.

Un programme commencera toujours par une initialisation des registres de direction des ports A et B. Si on ne fait rien, ils sont en entrée après un Reset. Il faudra aussi initialiser les variables, compteurs dont la valeur à l'enclenchement est importante. On donnera un nom aux variables, en déclarant au début les équivalences entre ces noms et les adresses mémoire assignées (en évitant de mettre deux variables différentes dans la même adresse mémoire). Les noms des registres du PIC sont connus par l'assembleur et n'ont pas besoin d'être déclarés.

2.2. Un premier programme

Nous voici enfin à pied d'oeuvre pour devenir des experts dans la programmation du PIC. Comme pour tous les processeurs, il faudrait déjà connaître tout le processeur avant de pouvoir expliquer clairement la moindre instruction. Avec des exemples, nous allons progressivement nous familiariser avec l'architecture et le répertoire d'instruction. Toute l'information utile est résumée dans une feuille de codage qui deviendra notre instrument de travail. La documentation complète du fabricant est essentielle pour comprendre toutes les possibilités du processeur (timer, interruptions), mais en première étape on ne peut que s'y perdre. Le document en anglais "Programming the Microchip-PIC microcontrollers" (www.didel.com/picg/PicE.html) détaille en anglais les instructions en notation CALM et Microchip-PIC, et donne plusieurs exemples de programmes avancés.

Chargeons et exécutons le programme PicgT. Il copie le port A, initialisé en entrée, sur le port B, initialisé en sortie. Si on presse sur l'un des deux poussoirs du module PicgExe, l'effet sera visible sur deux LEDs. Pouvez-vous prévoir lesquelles?

Le programme PicgT (le PC ne fait pas de distinction entre majuscules et minuscules) a l'extension .ASM comme tous les programmes en assembleur, mais cette extension n'a pas besoin d'être précisée à SMILE NG. Il se trouve dans le répertoire PicGenial et se transfère pour programmer le PIC avec F5, comme expliqué dans la section 1.1.3. Ce même programme peut être chargé dans le 16F870. Le code est le même pour les deux processeurs, mais pas la séquence de programmation. C'est donc très important que SmileNG soit présélectionné sur le bon processeur (16F84-RC, 16F84-XT ou 16F870-XT). DebVar est choisi dans les exemples pour être compatible avec les deux processeurs.

```
Program PicgT   Copie le port A sur le port B
.proc 16F84

Constant Ports  Ports A et B
DirA   = 2'11111       ; Tout le port A en entrée
DirB   = 2'00000000    ; Tout en sortie
ToutEteint       = 2'11111111   ; Lampes éteintes
.Loc 0

Program Initialisation
Debut:Move      #DirA,W
      Move      W,TrisA
      Move      #DirB,W
      Move      W,TrisB

Program Boucle  On boucle sans cesse pour copier RA dans RB
Boucle:
      Move      PortA,W
      Move      W,PortB

      Jump      Boucle
.End
```

0.95

Reprenons ce programme instruction par instruction pour comprendre la raison et le sens de chaque instruction. Ce programme est traduit par l'assembleur en codes d'instructions qui sont chargées en mémoire. On peut voir les codes générés en chargeant le fichier PicgT.lst, mis dans le répertoire Picgenial par l'assembleur.

\prog;PicgT|Copie le port A sur le port B

Dans SmileNG il y a, en plus des instructions pour le processeur, des ordres de mise en page et des pseudo-instructions pour l'assembleur. Pour avoir sur l'écran un joli graphisme en début de programme, il faut taper la séquence \prog;xx|yy.

Cet ordre est interprété par Smile NG et par le programme d'impression. Il ne perturbe pas les autres outils d'édition, de transfert et d'impression. C'est donc une solution simple et compatible. Les ordres "LILA" interprétés pas Smile NG sont donnés dans l'annexe 3. Le nom du programme, PicgT est naturellement le même que celui du fichier sur disque dans le répertoire PicGénial: PicgT.asm. L'extension .asm est automatique. Les minuscules et majuscules sont identifiées.

.proc 16F84

Cette pseudo-instruction signale à l'assembleur le processeur utilisé (16C84 et 16F84 ont le même jeu d'instructions). Un fichier 16F84.pro doit exister sur le disque, dans le répertoire "Proc" de SmileNG, en plus de l'assembleur Ascalmc.exe contenu dans le répertoire "Exe".

\const;Ports|Ports A et B

Un ordre de mise en page que l'assembleur ignore

```
DirA        = 2'11111       ; Tout le port A est en entrée
DirB        = 2'00000000    ; Tout le port B est en sortie

ToutEteint          = 2'11111111   ; Lampes éteintes
```

Les constantes sont déclarées au début du programme. Ici le port A (5 bits) est voulu en entrée. Les 5 bits de direction doivent être mis à un. 2' signifie que l'on travaille en binaire, ce qui est naturel ici. Les 8 bits du port B sont en sortie. Les bits de direction devront être mis à zéro. Ces pseudo-instructions de déclaration n'ont pas d'effet sur le processeur. Avec ces déclarations en début de programme, on établit un dictionnaire entre un langage qui nous est naturel (DirA pour la configuration de direction du port A) et le langage du processeur, formé de bits seulement.

On voit dans le schéma qu'il faut que le bit en sortie soit à zéro pour que la diode correspondante s'allume. Tous les bits du port B doivent être à un pour que les diodes soient éteintes. La constante "ToutEteint" qui permettra de s'assurer que toutes les diodes sont éteintes à la fin de l'initialisation, est donc un mot avec des "1" partout.

\prog:Initialisation|

```
Debut:  Move    #DirA,W
        Move    W,TrisA
        Move    #DirB,W
        Move    W,TrisB
```

Ce sont les 4 premières instructions exécutées, à partir de l'adresse 0. Elles initialisent la direction des ports. TrisA et TrisB sont les registres de direction. Il faut les initialiser avec les valeurs DirA et DirB, ce qui implique de copier la valeur à transférer dans le registre de travail W (workspace register). On pourrait écrire Move #2'11111,W et ne pas déclarer DirA. Mais c'est prendre une mauvaise habitude que de ne pas déclarer les constantes et variables. Pour les programmes simples, cela simplifie un peu, mais dès que les programmes sont compliqués, on s'empêtre, fait des fautes, et un programme que l'on a écrit soi-même est difficile à comprendre après quelques jours.

Le signe # (lu "valeur" plutôt que "dièze") montre que DirA est une valeur immédiate, fixe dans le programme. La valeur peut être donnée en décimal, en binaire (2'01101) ou en hexadécimal (16'F3); l'assembleur traduira en binaire pour mettre en mémoire programme la valeur que le processeur comprend. TrisA par contre est un registre, dont le contenu est variable. W également. Move W,TrisA transfère le contenu de W dans le registre TrisA. W n'est pas modifié.

A noter encore qu'il n'y a pas de commentaires pour expliquer ces 4 instructions. Leur lecture est évidente. Ce qui est important, c'est que DirA et DirB soient bien expliqués au début du programme, avec tous les détails nécessaires pour que l'on puisse câbler le système conformément au programme.

```
        Move    #ToutEteint,W
        Move    W,PortB
```

Les bits en sortie doivent être assignés, autrement l'état initial sera quelconque (en fait le PIC initialise ses registres à zéro à la remise à zéro (Raz). Ici, on veut que les diodes soient éteintes à l'enclenchement. La réflexion pour savoir s'il faut mettre des 1 ou 0 a été faite au début, dans les déclarations. Il n'y a plus besoin de se reposer des questions. Si on change de système et qu'il faut un 0 pour éteindre, il suffit de changer les déclarations, et rien dans le programme.

\prog;Boucle|On boucle sans cesse pour copier RA dans RB

```
Boucle:
```

L'assembleur note l'adresse mémoire de boucle. Plus loin, l'instruction "Jump Boucle" permet de revenir ici pour répéter l'opération

```
        Move    PortA,W
        Move    W,PortB
```

Ces deux instructions copient le port A (qui comporte en particulier deux poussoirs sur les entrées RA3 et RA4) sur le port B. La copie se fait sur les bits de même numéro.

Attention pour le 16F870, RA3 n'est pas copié. Voir www.didel.com/picg/Picg0.pdf

```
        Jump    Boucle
.End
```

La pseudo-instruction .END dit à l'assembleur que son travail de traduction et terminé. Les lignes suivantes sont ignorées. Cela peut être le mode d'emploi de votre programme. Les fichiers se perdent moins facilement que les feuilles de papier. Documentez ce que vous faites autant que possible dans votre programme et pas ailleurs. Après le .End, il n'y a plus besoin de ; devant les lignes, et les lignes peuvent avoir plus de 96 caractères.

S'il n'y a pas d'erreurs, l'assembleur génère le code binaire, transféré automatiquement dans le PIC via le port parallèle du PC si vous l'avez demandé.

A l'exécution, le processeur va initialiser les ports, et ensuite exécuter sans cesse la boucle qui copie le port A sur le port B. La durée de la boucle est 4 microsecondes. Si le poussoir est pressé, l'entrée sur le port A passe à zéro. Ce zéro est copié sur le bit correspondant du port B, et allume la lampe correspondante. Puisque les poussoirs sont sur

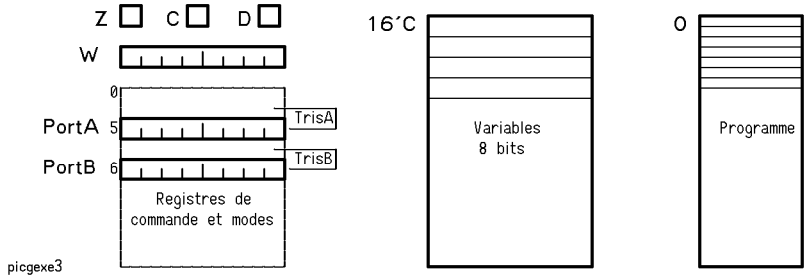
les lignes RA3 et RA4 du processeur (port A bits 3 et 4), ce sont les LEDs 3 et 4 (donc 4e et 5e depuis la gauche) qui vont être activées. Comme un zéro est actif à la fois pour les poussoirs et pour les LEDs, l'effet est correct. Si on veut que les LEDs non concernées ne s'allument pas, il faut forcer les autres lignes du port B à un, ce qui se prépare dans W avec un OU logique. Il faut insérer entre la lecture du port A et l'écriture sur le port B l'instruction "Or #2'11100111,W". Ceci sera expliqué avec plus de détails plus loin, mais il faut dès maintenant s'habituer à compter les positions de bits correctement (de 0 à 7 même si on dit premier à 8e).

2.3. Un peu plus sur l'architecture du PIC

Le PIC 16F84/16F870 contient en plus de ses deux ports A et B, et leurs registres de direction associés, des registres de commande et de mode que nous découvrirons petit à petit. Ils utilisent les adresses 0 à 16'B sur le 16F84 et 0 à 16'1F sur le 16F870. Une zone de registres est prévue pour les variables à partir de l'adresse DebVar=16'20. Il y a deux façons de déclarer les adresses des variables. On peut les assigner avec des déclarations Var1= 16'20 (mieux: DebVar+0), Var2 = 16'21 (DebVar+1), ce que nous ferons dans les premiers programmes simples.

Les bits Z, C et D, réunis dans un registre de "fanions" (flag register F) sont très importants et nous donneront bien du fil à retordre. Le bit Z est activé par certaines opérations pour signaler que le résultat de l'opération est nul. Par exemple l'instruction "XOR Var,W" calcule le ou exclusif de la variable Var et du contenu du registre W. Z est activé si le résultat est nul, ce qui veut dire que Var et W avaient le même contenu (après exécution de l'instruction la variable Var est inchangée et W vaut zéro dans ce cas). "Move W,Var" ne modifie pas Z, mais "Move Var,W" modifie Z (Z=1 si Var contient zéro). Il faut, au début, regarder chaque fois la feuille de codage verte (voir plus loin) pour vérifier ce que fait exactement l'instruction.

Le bit C est activé s'il y a dépassement de capacité dans une addition, soustraction ou décalage. Là aussi, il faudra faire très attention. Le bit D est utile pour calculer en décimal, mais il est rarement utilisé.



picgexe3 *Fig. 2 Modèle simple du PIC*

2.4. Initialiser les ports

Comme on l'a vu, tout programme commence par des déclarations: quel est le processeur, quels noms sont donnés aux bits des ports, aux variables. Au début du programme il faut initialiser les directions de ports, en entrée ou en sortie. Avec les PIC, des instructions spéciales permettent d'initialiser ces registres de direction TrisA et TRisB. En CALM, elles s'écrivent Move W,TrisA, mais Move TrisA,W ou Set TrisA:#bit n'est pas possible. Un zéro programme une sortie. Par défaut (si on oublie de définir les registres TrisA ou TrisB), les ports sont en entrée après une Raz (remise à zéro), ce qui est logique puisque l'on veut éviter que le processeur entre en conflit avec des signaux allant vers le processeur. Pour initialiser un registre, une variable, il faut toujours avec le PIC passer par le registre de travail W. On initialise les 4 bits de poids faible du port B en sortie avec les instructions:

```
Move    #2'11110000,W    ; 2' pour les nombres binaire, 2'11110000 = 16'F0
Move    W,TrisA
```

A noter que comme pour tous les processeurs, l'instruction Move (en CALM) copie la valeur source donnée dans le registre destination. La 2e instruction ne modifie pas le contenu de W. Le signe # (prononcé dièze ou valeur devant un nom) indique que la valeur est immédiate (paramètre d'assemblage). Il serait préférable de déclarer au début du

programme "DirB = 2'11110000" et d'écrire dans le programme "Move #DirB,W". On aime bien avoir toutes les définitions de constantes au début, et il faut toujours nommer et déclarer au début du programme les constantes que l'on peut être amené à modifier par la suite.

Si on veut mettre tous les ports en sortie, et les initialiser à zéro, l'instruction Clr (Clear) existe, mais elle ne peut pas agir sur les registres de direction TrisA et TrisB. On doit donc écrire:

```
Clr     W
Move    W,TrisA
Move    W,TrisB
Clr     PortA           ; ou Move       W,PortA car W contient 0
Clr     PortB           ; ou Move       W,PortB
```

2.5. Copie de bits

Nous avons jusqu'à présent manipulé les 8 bits du port B simultanément. Pour le port A, ce serait la même chose, mais seuls les 5 bits de poids faibles sont disponibles, avec un 5e bit (RA4) un peu spécial puisqu'il est en collecteur ouvert: une résistance garantit l'état 1 si le port est utilisé en entrée. Sur un port, chaque bit doit en général être géré individuellement.

Le PIC est très performant pour agir directement sur l'un des bits d'un port ou d'une variable en mémoire (mais pas sur les registres de direction si on les accède par les instructions Tris), sans modifier les autres bits. Les bits sont numérotés de 0 (poids faible) à 7 (poids fort). Le signe : indique une sous-adresse; PortA:#3 est le bit 3 du port A. On nomme naturellement les numéros de bits selon leur fonction.

```
Clr     PortA:#BitNumber  ; Met le bit à zéro
Set     PortA:#BitNumber  ; Met le bit à un
```

On peut de même lire (en fait tester) un bit d'un port initialisé en entrée, sur lequel on a câblé un interrupteur par exemple, ou un bit dans une variable. Tester un bit veut dire que l'on va prendre une décision. Tout ce que sait faire le PIC est de sauter conditionnellement l'instruction suivante. Le nom de l'instruction est TestSkip et la condition est Bit Set ou Bit Clear (BS, BC). L'adresse du registre ou port suit avec comme sous-adresse le numéro du bit (de 0 à 7), qui est une valeur immédiate, donc avec le signe #.

```
TestSkip,BC  PortA:#bNumber  ; Skip si le bit testé est à zéro
TestSkip,BS  PortA:#bNumber  ; Skip si le bit testé est à un
```

Par exemple, si on veut copier l'état du poussoir câblé sur RA3 sur le bit de poids fort de l'affichage, RB7, on initialisera au moins RA3 en entrée et RB7 en sortie avant de tester et activer les bons bits. Le programme s'écrit

```
Program  Picgt00   Lit et copie un interrupteur
.Proc 16F84

Variables  Néant

Constant  Port A
bInter = 3          ; interrupteur en RA3
DirA   = 2'11111            ; RA4..0 en entrée

Constant  Port B
bLed   = 7          ; Led en RB7
DirB   = 2'00000000         ; Tout en sortie
```

```
Program  Début du programme
.Loc  0
Debut: Move         #DirA,W
       Move         W,TrisA
       Move         #DirB,W
       Move         W,TrisB
Boucle:
       TestSkip,BS  PortA:#bInter  ; Si 0 (bit clear), on
       Clr          PortB:#bLed
       TestSkip,BC  PortA:#bInter  ; dans l'autre cas, on
       Set          PortB:#bLed
;... rien d'autre à faire dans ce programme
       Jump         Boucle
.End
```

0.94

Notons que c'est assez pratique de mettre un b minuscule comme première lettre lorsqu'il s'agit d'un bit et non pas d'un mot de 8 bits. Il ne faut pas avoir peur de passer du temps à choisir et taper des noms explicites; ce temps reste négligeable vis-à-vis du temps passé à trouver des erreurs de programmation.

2.6. Supression de rebonds

Nous voulons compter les actions sur le poussoir. Le premier problème est de suivre un signal qui passe à un, puis passe à zéro, et d'exécuter une action (compter et afficher le résultat) pour l'une de ces transitions. Il suffit d'écrire

```
A$:     TestSkip,BS  PortA:#bSwitch
        Jump    A$       ; Exécuté si bit clear (on attend que le signal monte)
B$:     TestSkip,BC  PortA:#bSwitch
        Jump    B$       ; Exécuté si bit set (on attend que le signal redescende)
        ; Tâche à exécuter
```

Le deuxième problème à résoudre est que notre poussoir a des rebonds de contact; la lame du contact rebondit pendant quelques millisecondes et le processeur est assez rapide pour voir ces contacts successifs. Une solution est de ne pas lire trop souvent, en insérant une boucle d'attente de 2 ms au moins (et 50ms au plus pour ne pas rater une action très rapide sur le poussoir). Le programme complet appelle une routine d'attente dans laquelle le délai en 100 microsecondes est un paramètre. Il est facile de le réduire pour voir quelle est la durée maximale des rebonds.

Puisque les diodes sont allumées par un zéro, il faut inverser la valeur du compteur avant de la transférer vers le port B. Le PIC sait faire cela en une seule instruction. L'instruction NOT inverse tous les bits d'un registre. Avec le PIC, "NOT Compteur" inverse tous les bits de la variable compteur, ce qui n'est pas souhaité ici. "NOT Compteur,W" ne modifie pas Compteur, car l'inversion est faite au moment de la copie dans W.

Le programme suivant a besoin de 3 variables. C0 C1 sont des compteurs pour boucle d'attente. Compteur est la variable qui compte lentement et est copiée sur le port B pour être visualisée à chaque changement. On pourrait déclarer

```
Compteur  = 16'20
C0        = 16'21 ; ou Compteur + 1
C1        = 16'22
```

On préfère comme dans le programme ci-dessous, réserver un position mémoire avec la pseudoinstruction .Blk.16 1 (la variable est 8 bits, mais l'assembleur PIC doit considèrer que les variables ont la même taille que les instructions 12, 14 ou 16 bits des processeurs de la famille PIC). Il faut naturellement dire alors à partir de quelle adresse on trouve ce bloc de variables (.Loc 16'0) et mettre au début du programme un .Loc 0 pour dire que le programme commence en zéro. Ceci sera fait dès que les programmes deviennent plus complexes.

```
Program PicGT0   Comptage des actions sur le poussoir gauche

.Proc 16F84
DebVar= 16'20

Constant Ports  bSwitch      = 3       ; sur RA3, a
DirA    = 2'11000            ; RA4 RA4 en entrée
DirB    = 0       ; tout en sortie, LEDs actives à zéro

Variables Variables
Compteur        = DebVar
C1              = DebVar+1
C2              = DebVar+2
.Loc  0

Program Début
Debut:Move      #DirA,W      ; RA3 en sortie
      Move      W,TrisA
      Move      #DirB,W
      Move      W,TrisB                ; PortB en so

Boucle:
 A$:  Call      Delai
      TestSkip,BS  PortA:#bSwitch
      Jump      A$              ; Exécuté si bit clear
 B$:  Call      Delai
      TestSkip,BC  PortA:#bSwitch
      Jump      B$
      Inc       Compteur
      Not       Compteur,W
      Move      W,PortB
      Jump      Boucle

Routine Delai  Delai multiple de 100µs
  in:   W delai 0, 0,1 ... 25,5 ms
  mod: C1 C2 W
Delai: Move     #20,W           ; Exemple avec 2 ms
       Move     W,C1
  A$:  Move     #32,W           ;Boucle interne 100µs
       Move     W,C2
  B$:           DecSkip,EQ C2
       Jump     B$
       DecSkip,EQ C1
       Jump     A$
       RetMove  #1,W
.End
```

Sauriez-vous remettre le compteur à zéro avec l'autre poussoir? C'est plus simple, car il n'y a pas besoin de se préoccuper des rebonds. Si le poussoir n'est pas activé on passe par dessus une instruction "Clr Compteur" à rajouter. Cette instruction doit être mise dans la boucle qui attend que l'on presse sur le poussoir de comptage. Il faut aussi faire la mise à jour du port B immédiatement, car c'est lui qu'on voit. C'est un peu plus compliqué, et il y a plusieurs solutions. A vous de trouver la plus élégante.

2.7. Feuille de codage

La feuille de codage CALM du PIC16F84 en www.didel.com/picg/Pic84Calm.pdf contient toute l'information nécessaire pour se rafraîchir la mémoire quand on programme. Pour les fonctions spéciales, la documentation du fabricant est indispensable.