Apprendre à programmer avec le 16F628 – 1ere partie

Vous préférez le 16F877 ? → www.didel.com/pic/Cours877.pdf

-- en travail – dire votre intérêt à info@didel.com

Le but est d'apprendre à écrire des programmes en assembleur pour des application utilisant un code efficace sur des microcontrôleurs en général plus petits. Notre démarche est de tester les programmes sur un 16F628 (ou 627) qui a 2 ports d'entrée-sortie.
Le 16F628 est la version améliorée du 16F84 bien connu et abondamment documentés. Le 628 est compatible avec le 84, sauf quelques instruction différentes à l'initialisation. Le 627 est identique, avec moins de mémoire. Plusieurs autres PICs sont compatibles, ils diffèrent par leurs quantités de mémoire, et par des fonctionnalités supplémentaires non étudiées dans ce document, parfois testées par les programmes PicTest.

Matériel nécessaire

Pickit2 65.-
Kit Microdule M18Eval 100.-
Doc sous www.didel.com/08micro/M18Eval.pdf

M18Eval

La doc générale sur les Microdules est sous
www.didel.com/pic/Microdules.pdf

Table des matières

1ere partie

1 Introduction

Avant de pouvoir faire les exercices de ce document, il faut installer
SmileNG www.didel.com/pic/SmileNG.zip
Pour la documentation de l'éditeur SmileNG et ses ordres de mise en page, voir
http://www.didel.com/pic/SmileNG.pdf .
Pickit2 www.bricobot.ch/docs/Microchip.zip
La procédure d'installation se trouve en www.didel.com/pic/InstallPic.pdf
A peu près la même chose, avec le brochage du connecteur de programmation sous
www.didel.com/pic/Pickit2.pdf
Les **exemples de programmes** associés à ce document se trouvent sous
www.didel.com/pic/Cours628.zip Mettre ces fichiers dans un dossier Cours628.

Il faut ensuite connecter le Microdule M18 avec son 16F628, tester les sorties avec le programme T628.hex (ce programme sera expliqué plus loin) avec un afficheur sur chaque port pour vérifier que cela clignote partout ou cela a un sens. Si non, vérifiez les soudures.

1.1 Comment travailler ?

Chacun a besoin d'explications différentes pour bien comprendre. Ce document s'adresse à des débutants sans connaissances initiales, et prévoit beaucoup d'exercices progressifs avec très peu d'explications générales avant de faire les exercices.

1.2 Architecture des PICs

Les PICs ont une architecture simple, avec une mémoire d'instructions 12 ou 14 bits, des variables 8 bits et des périphériques 8 bits dans le même *espace* que les variables.
Les exercices permettront de comprendre la différence entre constantes et variables, adresses et données, et de se familiariser progressivement avec les instructions.

1.3 L'assembleur CALM

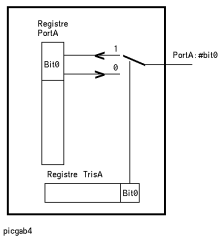
Les règles de l'assembleur et les pseudo-instructions apparaîtront petit à petit, parfois sans explications. L'expérience a montré que trop d'explications découragent. C'est en écrivant des programmes simples que la compréhension se développe.

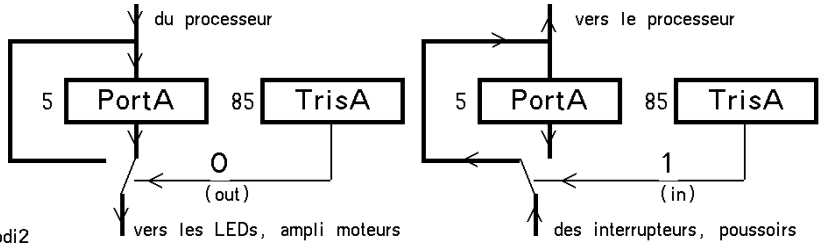
2. Entrées et sorties

L'intérêt des microcontrôleurs est qu'il ont des sorties sur lesquelles on peut brancher des diodes lumineuses, des amplis pour commander des moteurs, un haut-parleur. Ils ont des entrées pour connecter des poussoirs et interrupteurs, pour lire des signaux analogiques, pour mesurer des durées d'impulsions. Ces fonctions sont programmables et c'est notre premier souci : configurer le microcontrôleur dans le mode compatible avec l'exercice.

2.1 Registre de direction

Pour décider si une broche du processeur est une entrée ou sortie, il faut que le contrôleur aie des aiguillages internes commandés par un registre de direction appelé *Tris*. Si le bit *Tris* est à 1 (input) la broche du circuit est connectée au bus et amène l'information dans le registre W, lorsque l'instruction `Move Port,W` est exécutée. Si le bit *Tris* vaut 0 (output), c'est l'information qui vient du bus qui est mémorisée dans le registre et activée en sortie.





pdi2

Au début de chaque programme, il y a quelques instructions pour configurer les ports. On se contentera au début de comprendre leur effet.

2.2 Un premier exercice : copier le portA sur le PortB

Le 16F628 a deux ports A et B. Le programme le plus simple que l'on puisse écrire copie l'état des lignes d'entrées du PortA sur les sorties du PortB. Les microdules Sw8 et Lb8 facilitent le test : les interrupteurs du module Sw8 assignent des états logiques 0 ou 1 sur le portA et les diodes du module Lb8 montrent du rouge pour l'état 0 et du vert pour l'état 1.
Le cœur du programme est :

```
Loop:
      Move  PortA,W
      Move  W,PortB
      Jump  Loop
```

Peut-on imaginer plus simple ? Move est en fait une copie, on ne déplace pas l'information, on en fait une copie. On ne peut pas copier directement du portA vers le portB, il faut passer par le registre W, qui joue un rôle de plaque tournante.
Loop est un nom inventé (une étiquette pour dire où est la prochaine instruction) ; on aurait pu mettre Boucle, Toto, mais pas Jump ou PortA qui sont des noms réservés. Il faut mettre le : sans espace devant.
Pour vérifier l'exécution, il faut charger le programme **Ex628-101.asm** dans SmileNG et assembler avec F5.
A noter que le programme complet contient des instructions supplémentaires qui seront expliquées plus loin. Seul le cœur des programmes, dont on vérifie le comportement sur l'affichage binaire nous intéresse pour le moment.

2.3 Valeur fixe dans le portB

Si on regarde le mot binaire de la photo précédente, on lit `01101010`. Tous les processeurs travailent en binaire (base 2) que nous noterons 2'01101010. L'hexadécimal est préféré car plus compact, la valeur équivalente est 16'6A. Si ce codage ne vous est pas familier, lire la première page de www.didel.com/pic/Bases.pdf
Le cœur du programme **Ex628-103.asm** est

```
Loop:
      Move  #16'6A,W    ; Move #2'01101010
      Move  W,PortB
      Jump  Loop
```

Faire les exercices proposés au bas du programme (ce qui suit le .end n'est pas assemblé).
A noter que le portA a des lignes un peu spéciales qui conviennent moins bien pour afficher des valeurs. On va donc utiliser en général le portB pour afficher des valeurs.

2.4 Exemple : osciller le portB

Activer-désactiver le portB, donc créer une impulsion positive sur tous ses bits, s'écrit de plusieurs façons. La première et la 3e solutions sont identiques (les mêmes instructions sont générées).

```
Move  #2'11111111,W
Move  W,PortB
Move  #2'00000000,W
Move  W,PortB
```

```
Move  #2'11111111,W
Move  W,PortB
Clr   PortB
```

```
Move  #-1,W
Move  W,PortB
Move  #0,W
Move  W,PortB
```

Pour répéter ces instructions d'activation/désactivation, on crée une boucle en mettant une étiquette au début de la boucle et en ajoutant une instruction qui saute à cette étiquette, c'est à dire à l'adresse correspondante dans la mémoire du processeur, comme dans le programme précédent.

```
Loop:   Move    #2'11111111,W     ; 1 µs
        Move    W,PortB
        Move    #2'00000000,W
        Move    W,PortB
```

Les PICs à 4 MHz exécutent une instruction par microseconde (2 µs pour les sauts). Les 4 instructions précédentes vont activer le port C pour 2 microscondes seulement, et désactiver pour 3 microsecondes. On ne verra rien sans utiliser un oscilloscope. Un programme utilisable sera vu dans la section suivante.

2.5 Boucles d'attente

Pour que le processeur modifie les sorties à une vitesse ''humaine'', il faut insérer des boucles d'attente. On initialise un compteur et on décompte jusqu'à la valeur zéro, que le processeur sait bien reconnaître.

```
      Move  #250,W
      Move  W,Cx1
Att:  Dec   Cx1   ; 1 µs
      Skip,EQ     ; 1 µs
      Jump  Att   ; 2 µs
```

L'instruction Dec diminue de 1 la variable Cx1. Cx1 est un nom inventé, et il faut dire ou cette variable se trouve en m.moire. C'est libre à partir de l'adresse 16'20, donc il faut déclarer avant le début du programme

```
Cx1   = 16'20
```

L'instruction `Skip,EQ` se prononce Skip If Equal, saute si c'est égal, si l'opération précédente a un résultat égal à zéro. Si ce n'est pas égal à zéro, on passe à l'instruction suivante, décompte à nouveau. A noter que le Skip,EQ, comme le Jump, s'exécute en 2 microsecondes.
Ce bout de programme va faire 250 fois la boucle qui dure 4 microsecondes, donc la durée totale est de 1 milliseconde (plus les 2 microsecondes d'initialisation du décompteur).
A noter les trois nouvelles instructions. Dec Cx1 décompte la variable mentionnée. L'instruction Skip,EQ permet de savoir si le compteur est à zéro (EQ equal). Si oui, le processeur saute l'instruction suivante. Si non, il exécute le saut qui lui fait attendre encore un tour.
Ce nouveau cœur de programme s'écrit

```
Loop:    Move     #2'11111111,W     ; on allume
         Move     W,PortB
; attente de 1ms
            Move  #250,W
            Move  W,Cx1
      Att:  Dec   Cx1    ; 1 µs
            Skip,EQ      ; 1 µs
            Jump  Att    ; 2 µs

         Move     #2'00000000,W     ; on éteint
         Move     W,PortB
; attente de 1ms
            Move  #250,W
            Move  W,Cx1
      Att:  Dec   Cx1    ; 1 µs
            Skip,EQ      ; 1 µs
            Jump  Att    ; 2 µs
         Jump     Loop
```

C'est encore trop rapide pour notre oeil. On ne peut pas augmenter la valeur 250 au delà de 255, car la variable Cx1 est un mot de 8 bits et les PICs n'ont pas d'instructions pour des compteurs ou décompteurs plus grands que 8 bits.
On fait alors une 2e boucle qui appelle la première et utilise une 2e variable. Vous voulez un dixième de secondes ? Il suffit de répéter 100 fois la boucle de 1ms.

```
; Boucle 100 millisecondes
      Move  #100,W
      Move  W,Cx2
Att2 :
      Move  #250,W
      Move  W,Cx1
Att1: Dec   Cx1   ; Boucle 1ms
      Skip,EQ
      Jump  Att1
      Dec   Cx2
      Skip,EQ
      Jump  Att2
```

On pourrait naturellement compter et comparer quand on atteint la valeur 100 ou 250. C'est moins efficace avec le PIC.

2.6 Programme complet

Il faut donc insérer nos boucles d'attente de 100ms après avoir écrit et après avoir effacé le motif sur le PortB. Le programme doit aussi initialiser le PortB en sortie, et déclarer l'adresse des variables Cx1 et Cx2.

```
\prog;Ex628-104|Clignote le portB
\b; RB0..RB7 oscille,
;période 0.65 s a 4 Mhz
; Ne pas utiliser si des sorties
; sont court-circuitées ou
; connectées a des sorties
; Voir T628Cli pour une écriture
plus élégante
.Proc   16F84
.Ref    16F628
;Configuration
.Loc    16'2007
.16     16'3F39

Cx1     = 16'20 ; Début des var
Cx2     = 16'21

Motif1  = 16'55 ; 2'01010101
Motif2  = 16'AA ; 2'10101010
.Loc    0
Deb:
        Move    #0,W    ; out
        Move    W,TrisB
Loop:
        Move    #Motif1,W
        Move    W,PortB
; attente 100ms
        Move    #100,W
        Move    W,Cx2
Att2:
        Move    #250,W
        Move    W,Cx1
Att:    Dec     Cx1
        Skip,EQ
        Jump    Att
        Dec     Cx2
        Skip,EQ
        Jump    Att2
        Move    #Motif2,W
        Move    W,PortB
; attente 100ms
        Move    #100,W
        Move    W,Cx2
Att2b:
        Move    #250,W
        Move    W,Cx1
Attb:   Dec     Cx1
        Skip,EQ
        Jump    Attb
        Dec     Cx2
        Skip,EQ
        Jump    Att2b

        Jump    Loop
.End
```

2.7 Optimisation ou simplifications

L'instruction DecSkip,EQ remplace les deux instructions de la boucle d'attente.
Si on n'a pas besoin d'une durée précise, on peut ne pas recharger les compteurs. Après être arrivés à zéro, ils recommencent un tour de 256 pas. La façon la plus simple de faire une attente de 256x256x3 µs =~ 0.2 secondes est d'écrire

```
A$: DecSkip,EQ  Cx1
    Jump  A$          ; A$ est une étiquette dite locale
    DecSkip,EQ  Cx2
    Jump  A$
```

Le programme **T628.asm** utilise cette nouvelle attente et clignote tous les ports en inversant le motif avec un ou-exclusif que l'on comprendra plus loin. Le programme est nettement plus court.
T628Cli.asm est un programme de test que l'on recharge toutes les fois que l'on veut vérifier que le processeur fonctionne et que toutes les sorties sont connectées. S'il ne tourne pas, c'est que l'oscillateur interne n'est pas initialisé.

2.8 Routines

Ce n'est pas très élégant, ni très efficace, de devoir écrire deux fois la même boucle d'attente, comme dans le programme **C628Ex26.asm**. On écrit une seule fois la boucle d'attente, on la termine avec l'instruction Ret et c'est une routine, et on peut l'appeler autant de fois que l'on veut. L'instruction Call fait l'appel et à la fin de la routine, l'instruction Ret (return) retourne à l'instruction qui suit le Call.
Ecrivons une routine **DelWx1ms** avec un paramètre en entrée.

```
\rout:AttWx1ms|Attente de (W) fois 1ms
\in:W nombre de ms
\mod:W Cx1 Cx2
AttWx1ms:  Move    W,Cx2
A$:   Move  #250,W
      Move  W,CX1
  B$: Nop
      DecSkip,EQ  CX1   ; boucle 1ms
      Jump  B$
      DecSkip,EQ Cx2    ; boucle (W) fois 1ms
      Jump  A$
      Ret
```

Cette routine est le premier élément d'une bibiothèque de routines que chaque programmeur se constitue. Les 3 premières lignes contiennent toute l'information nécessaire pour l'utiliser.
Vous pouvez vérifier avec le programme **T628-105.asm** et changer le motif ou
la période en corrigeant une seule valeur au début du programme.

Exercices – délais et instructions logiques

Le dossier **Exos628/Ex1Logic** www.didel.com/pic/Exos628.zip contient des exercises pour comprendre et modifier des programmes simples. C'est le moment de vérifier que c'est facile !
Rappelons que ce qui suit le **.end** n'est pas assemblé. Les questions pour modifier les programmes se trouvent à cet endroit.
Exercices : Logic1 Logic2 Logic3 Logic4 Logic5 Logic6

2.9 Agir sur une seule ligne de sortie

On veut souvent modifier une sortie sur un port sans modifier celles qui sont à côté sur le même port. Une première solution est de lire le port, modifier avec les bonnes instructions, et ré-écrire la valeur. Par exemple, si on veut mettre à 1 le signal Led du portB (sur le bit2 en 3^{e} position) on peut écrire

```
Move  PortB,W
Or    #2'00000100,W     ; force le bit 2 à l'état 1,
                        ;   ne modifie pas les autres bits
Move  W,PortB
```

Une instruction fait exactement cela ; c'est bien de donner un nom au bit concerné :

```
bLed     = 2   ; 3e position car on numérote 0 1 2 .. depuis la droite
    Set   PortB:#bLed   ;
```

Pour mettre le bit 2 à zéro sans modifier les autres :

```
Move  PortB,W
And   #2'11111011,W  ; force le bit 2 à 0, ne modifie pas les autres
Move  W,PortB
```

Une instruction fait exactement cela :

```
Clr   PortB:#bLed
```

Le programme **C628Ex29** permet de vérifier ces instructions. Comme pour le programme de la section 2.2, on copie le portA sur le portB, mais en forçant le bit 2 à 0 et le bit 5 à un.
Si le ET (And) et le OU (Or) logique ne vous sont pas familiers, regardez le document www.didel.com/pic/Bases.pdf. Le Ou exclusif du programme **T628Cli.asm** y est aussi expliqué.

2.10 Lire un poussoir isolé

Pour lire un bit isolé, par exemple un poussoir relié à une entrée, on peut naturellement lire tout le port et ensuite isoler le bit pour savoir s'il vaut un ou zéro. L'instruction AND permet ce masquage d'un ou plusieurs bits dans un mot.
Par exemple, un poussoir est connecté sur le bit3 du port A initialisé en entrée. Le poussoir actif ferme le contact et impose un zéro sur l'entrée.

```
bPous = 7
Move  PortB,W
And   #2'1 0000000,W    ; seul le bit 4 reste dans son état
Skip,EQ     ; le résultat est nul si le bit 4 est à zéro
Jump  PousInactif
Jump  PousActif
```

Comme on doit très souvent savoir si un bit d'un port ou d'une variable est à un ou zéro, une instruction unique remplace les 3 instructions précédentes:

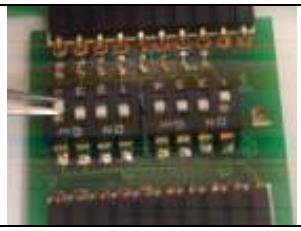
```
TestSkip,BC PortB:#bPous  (skip if bit clear) saute si le bit 4=0
Jump  PousInactif
Jump  PousActif
```

Pour tester si un bit est à un, on a l'instruction

```
TestSkip,BS PortB:#bPous  (skip if bit set)
Jump  PousActif
Jump  PousInactif
```

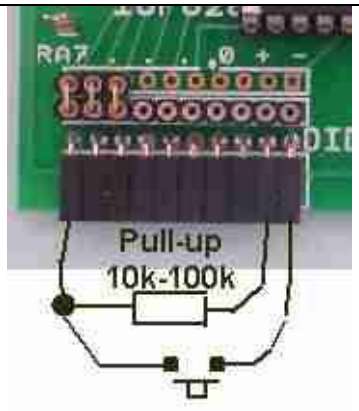
Les trois instructions (Move – And – Move) sont toujours utilisées si on veut tester un groupe de bits. Pare exemple, si 4 poussoirs sont pressés, le AND permet de savoir qu'il y en a un qui est activé, mais il faut ensuite décider lequel.

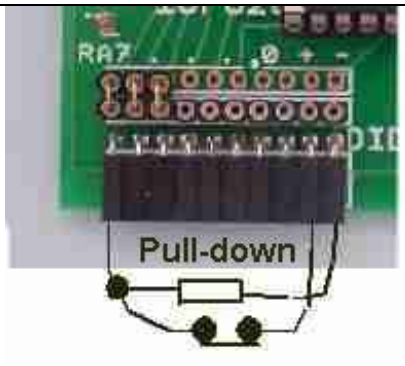
Le cœur du programme doit dire ce que l'on fait si le bit déclaré est actif ou non. Exécuter le programme **C628Ex2A** qui montre un motif différent sur le portB si on bouge l'interrupteur de gauche.



Câblons un poussoir réel pour bien comprendre les techniques d'interface. Enlevons le module Sw8. Avec un fil relions le bit 7 avec le + ou le -. Le comportement est correct. Le problème est que si l'entrée RA7 n'est pas câblée, le résultat n'est pas prévisible car le potentiel n'est pas défini. Il faut câbler une pull-up ou une pull-down et un seul fil, ou un poussoir simple, permet de changer d'état.

Le problème est que si l'entrée RA7 n'est pas câblée, le résultat n'est pas prévisible car le potentiel n'est pas défini. Il faut câbler une pull-up ou une pull-down et un seul fil, ou un poussoir simple, permet de changer d'état.
Avec une pull-down, ce qui est en général préféré, la pression sur le poussoir force un zero et on change le **.BS** en **.BC**





A noter encore que les 7 instructions du cœur du programme **C628Ex2A** se simplifient :

```
Loop:  Move   #MotifOne,W
       TestSkip,BS  PortA:#bPous
```

```
        Move   #MotifZero,W
        Move   W,PortB
        Jump   Loop
```

2.11 Mélange d'entrées et sorties sur un même port

Le registre Tris permet de mélanger des entrées et sorties, mais il peut y avoir des directions prioritaires, par exemple si le convertisseur analogique est déclaré. Les programmes de test mettront en évidence quelques cas simples. La documentation du fabricant sur 300 pages les décrit tous !

Comme exemple, câblons une LED sur le bit 2 du PortB et un poussoir sur le bit 4. Décidons que les 4 bits de poids faibles sont des sorties, et les 4 bits de poids forts sont des entrées. Les instructions spécifiques à cet exemple sont :

```
      bLed  = 2
      bPous = 4
      DirB  = 2'11110000
...
      Move  #DirB,W
      Move  W,TrisB     ; en banque 1
...
      TestSkip,BS PortB: bPous
      Clr   PortB :#bLed
      TestSkip,BC PortB: bPous
      Set   PortB :#bLed
```

Le programme **C628ExB.asm** est plus efficace, car la moitié gauche du portB est en entrée et la moitié droite en sortie. L'instruction

```
      Swap PortB,W
```

permute les 2 moitiés en transférant dans W.

Question : peut on remplacer les 2 instructions

```
      Swap  PortB,W
      Move  W,PortB
```

par

```
      Swap  PortB
```

Essayez, ca marche en fait pour toutes les variables ; c'est parfois pratique de pouvoir permuter les deux moitiés..

2.12 Tester la durée d'action sur un poussoir

On veut savoir pendant combien de temps on a pressé sur un poussoir. On a vu dans la section 2.5 comment attendre des mutiples de 10ms par exemple Décidons d'attendre l'action sur le poussoir. Ensuite, toutes les 10ms, on compte tant que le poussoir est actif. On interdit alors au processeur de modifier l'affichage, pour avoir le temps de le lire.

```
1ere étape : attendre
      AtActif :   TestSkip,BC  PortA :#bPous ; poussoir actif à zéro.
                  Jump     AtActif
2e étape : compter tant que le poussoir est actif
                  Clr      Compteur
      Compte :    Inc      Compteur
                  Move     #10,W
                  Call     DelWx1ms
                  TestSkip,Bs  PortA :#bPous
                  Jump     Compteur ; si BS, donc
2e étape : on affiche le compteur
                  Move     Compteur,W
                  Move     W,PortB
      Fini :      Jump     Fini
```

Le programme C628Ex28 a une amélioration : le compteur est affiché à chaque incrément, ce qui permet de le voir travailler. Ce n'est plus nécessaire d'afficher quand on bloque le processeur.

Avec un comptage toutes les 10ms, la durée d'action mesurable est de 2.56 secondes. Si on veut plus, on peut modifier le temps d'échantillon (100 ms serait en core assez précis).
On peut saturer le compteur, c'est à dire faire en sorte qu'il se bloque à la valeur maximale, ou que l'affichage clignote pour signaler l'erreur.
Pour détecter la valeur maximale, le plus simple est d'écrire

```
      Inc      Compteur        ; comme avant
      Inc      Compteur,W     ; ne modifie pas Compteur mais on sait si le
résultat est nul
      Skip,NE
      Jump     Depasse        ; on est à 16'FF, il faut arrêter de compter
```

On peut encore agrandir le compteur et afficher lsur les 4 bits de poids faible du portA.
On écrira

```
      Inc      Compteur
      Skip,NE
      Inc      CompteHigh
      Move     Compteur,W
      Move     W,PortB
      Move     CompteHigh,W
      Move     W,PortA
```

Si on veut se bloquer sur la valeur 12 bits que l'on peut afficher au maximum, il suffit de vérifier si son 13^{e} bit à passé à un :

```
      Testskip,BS  CompteHigh:#4    ; 4   3 2 1 0  7 6 5 4 3 2 1 0
```

pas encore modifié le 4 janvier 2010

Il faut dans la boucle d'attente, surveiller le signal du poussoir, toutes les 50 millisecondes au moins si on veut être assez précis, et pas trop souvent si on ne veut pas mesurer la durée d'un rebond de contact. Si on veut mesurer une durée, pour voir par exemple comment on peut être rapide, on met un compteur à zéro, on attend quelques secondes avant d'allumer une LED et on compte à partir de cet instant. Le cœur du programme est donc :

```
TestReflexe:
      Clr     PortB:#bLed
      Call    Attente     ; attente 1-5 secondes
      Set     PortB:#bLed
      Clr     PortD
AtPous:       Call  Del20ms
      Inc     PortD ; Durée en multiple de 20ms
      TestSkip,BS  PortB :#bPous
      Jump    AtPous
Fini: Jump    Fini  ; Jump  APC idem, car APC = adresse de l'instruction
```

On se pose naturellement beaucoup de questions avec ce programme. Comment compter en décimal et pas en binaire, comment mettre un affichage 7 segments, comment faire un programme attractif. Cela viendra !
On voit à la fin de ce programme l'instruction Jump Fini ; il n'y a plus rien à faire pour nous, mais si on ne met pas cette instruction, le processeur va prendre les positions mémoire suivantes comme des instructions, et on ne peut pas deviner ce qui va se passer.

PIC 16F628 - CALM instructions

www.didel.com/pic/Pic84Calm.pdf

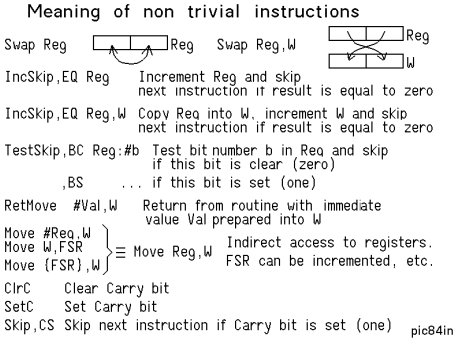
Code 14 bits	Microchip	CALM		Flags	Operation
110000 kkkkkkkk	movlw val	**Move**	**#Val,W**	[none]	#Val → W
000000 1fffffff	movwf reg	**Move**	**W,Reg**	[none]	W → Reg
001000 0fffffff	movf reg,0	**Move**	**Reg,W**	[Z]	Reg → W
001000 1fffffff	movf reg,1	**Test**	**Reg**	[Z]	Reg → Reg
000000 01100010	option	**Move**	**W,Option**	[none]	
000000 01100101	tris porta	**Move**	**W,TrisA** [1]	[none]	
000000 01100110	tris portb	**Move**	**W,TrisB** [1]	[none]	
000000 01100111	tris portc	**Move**	**W,TrisC** [1]	[none]	
111110 kkkkkkkk	addlw val	**Add**	**#Val,W**	[C,D,Z]	#Val+W → W
000111 0fffffff	addwf reg,0	**Add**	**Reg,W**	[C,D,Z]	Reg+W → W
000111 1fffffff	addwf reg,1	**Add**	**W,Reg**	[C,D,Z]	
111100 kkkkkkkk	sublw val	**Sub**	**W,#Val,W**	[C[2],D,Z]	#Val+(-W) → W
000010 0fffffff	subwf reg,0	**Sub**	**W,Reg,W**	[C[2],D,Z]	Reg+(-W) → W
000010 1fffffff	subwf reg,1	**Sub**	**W,Reg**	[C[2],D,Z]	Reg+(-W) → Reg
111001 kkkkkkkk	andlw val	**And**	**#Val,W**	[Z]	
000101 0fffffff	andwf reg,0	**And**	**Reg,W**	[Z]	
000101 1fffffff	andwf reg,1	**And**	**W,Reg**	[Z]	
111000 kkkkkkkk	iorlw val	**Or**	**#Val,W**	[Z]	
000100 0fffffff	iorwf reg,0	**Or**	**Reg,W**	[Z]	
000100 1fffffff	iorwf reg,1	**Or**	**W,Reg**	[Z]	
111010 kkkkkkkk	xorlw val	**Xor**	**#Val,W**	[Z]	
000110 0fffffff	xorwf reg,0	**Xor**	**Reg,W**	[Z]	
000110 1fffffff	xorwf reg,1	**Xor**	**W,Reg**	[Z]	
001110 0fffffff	swapf reg,0	**Swap**	**Reg,W**	[none]	Reg → Reg swapped Reg → W
001110 1fffffff	swapf reg,1	**Swap**	**Reg**	[none]	Reg → swapped Reg
000001 1fffffff	clrf reg	**Clr**	**Reg**	[Z=1]	0 → Reg
000001 00000000	clrw	**Clr**	**W**	[none]	0 → W
000000 01100100	clrwdt	**ClrWDT**		[none]	
000000 01100011	sleep	**Sleep**		[none]	
001001 1fffffff	comf reg,1	**Not**	**Reg**	[Z]	Reg → .not.Reg
001001 0fffffff	comf reg,0	**Not**	**Reg,W**	[Z]	Reg → Reg .not.Reg → W
001010 1fffffff	incf reg,1	**Inc**	**Reg**	[Z]	Reg+1 → Reg
001010 0fffffff	incf reg,0	**Inc**	**Reg,W**	[Z]	Reg → Reg Reg+1→ W
000011 1fffffff	decf reg,1	**Dec**	**Reg**	[Z]	Reg-1 → Reg
000011 0fffffff	decf reg,0	**Dec**	**Reg,W**	[Z]	Reg → Reg Reg-1→ W
001101 1fffffff	rlf reg,1	**RLC**	**Reg**	[C]	Reg → .rlc.Reg
001101 0fffffff	rlf reg,0	**RLC**	**Reg,W**	[C]	Reg → Reg .rlc.Reg → W
001100 1fffffff	rrf reg,1	**RRC**	**Reg**	[C]	Reg → .rrc.Reg
001100 0fffffff	rrf reg,0	**RRC**	**Reg,W**	[C]	Reg → Reg .rrc.Reg → W
0100bb bfffffff	bcf reg,b	**Clr**	**Reg:#b**	[none]	
0101bb bfffffff	bsf reg,b	**Set**	**Reg:#b**	[none]	
001111 1fffffff	incfsz reg,1	**IncSkip,EQ**	**Reg**	[none]	Reg+1 → Reg
001111 0fffffff	incfsz reg,0	**IncSkip,EQ**	**Reg,W**	[none]	Reg → Reg Reg+1→ W
001011 1fffffff	decfsz reg,1	**DecSkip,EQ**	**Reg**	[none]	Reg-1 → Reg
001011 0fffffff	decfsz reg,0	**DecSkip,EQ**	**Reg,W**	[none]	Reg → Reg Reg-1→ W
0110bb bfffffff	btfsc reg,b	**TestSkip,BC**	**Reg:#b**	[none]	
0111bb bfffffff	btfss reg,d	**TestSkip,BS**	**Reg:#b**	[none]	
1010kk kkkkkkkk	goto addr	**Jump**	**Addr**	[none]	
1000kk kkkkkkkk	call addr	**Call**	**Addr**	[none]	
110100 kkkkkkkk	retlw val	**RetMove**	**#Val,W**	[none]	
000000 00001000	return	**Ret**		[none]	
000000 00001001	retfie	**RetI**		[int]	
000000 00000000	nop	**Nop**		[none]	

bcf status,0	**CLRC**	Clr F:#0
bsf status,0	**SETC**	Set F:#0
btfss status,0	**Skip,CS**	TestSkip,BS Status:#C
btfsc status,0	**Skip,CC**	TestSkip,BC Status:#C
btfss status,2	**Skip,EQ**	TestSkip,BS Status:#Z
btfsc status,2	**Skip,NE**	TestSkip,BC Status:#Z

F same name as Status

Note 2: Processor add the 2-s complement; Carry value is inverted compared to most other µP

Meaning of non trivial instructions



All instructions execute in 4 clock cycles (1 µS at 4 MHz) except the Skip, Jump, Call and Ret that takes the double
Compare for equality: Xor W,Reg or Xor Reg,W Compare for Inequality: Sub instruction, but be careful
Except I/O, 16F87x, and other 14-bit instruction PICs have the same instruction set. jdn090823