II2 TD n°1

Voir WIKI en français
(http://fr.wikibooks.org/wiki/Comment_d%C3%A9marrer_avec_un_PIC16F84)
Les PICs ont été introduits en 1977 par General Instrument Corporation (PIC1650 datasheet de l'époque disponible sur Internet http://www.rhoent.com/pic16xx.pdf).
Les PICs représentent une famille tellement diversifiée qu'il est difficile de réaliser un ensemble de TDs suffisamment généraux pour les présenter. Nous avons choisi de présenter la série des 18F4550 et 16F877 qui seront utilisés en TPs. Pour la partie assembleur, seules seront présentées les instructions du 16F, le 16F84 étant encore très utilisé. Dans la partie description matérielle, on essaiera de montrer les architectures communes des deux modèles.

1°) La représentation des nombres

La donnée de base de PIC (18FXX8) est l'octet. L'octet comporte 8 bits. Le bit 0 est le bit de poids faible et le bit 7 est le bit de poids fort.

Mémoire programme : la mémoire programme du PIC (18FXX8) est organisée en mots de 16 bits. Le bit 0 est aussi le bit de poids faible et le bit 15 est le bit de poids fort.
L'adressage de cette mémoire programme se fait sur 20 bits. La mémoire mémoire programme comporte donc 2Mo (2 Méga octets). Les 18FX48 disposent de 16ko de mémoire programme et les 18FX58 disposent de 32ko donc loin des 2Mo théoriques.

Mémoire EEPROM : le PIC (18FXX8) dispose de 256 octets de mémoire EEPROM.
L'accès est réalisé à l'aide de la directive ORG 0xF00000.

Mémoire donnée : le PIC (18FXX8) dispose jusqu'à 4ko de mémoire donnée répartie en quinze banques subdivisées en deux parties :
- cases mémoires spéciales (registres spéciaux) en banque 15
- cases mémoires libres que vous pouvez utiliser à votre guise 768 (18FX48) ou 1536 (18FX58) octets.

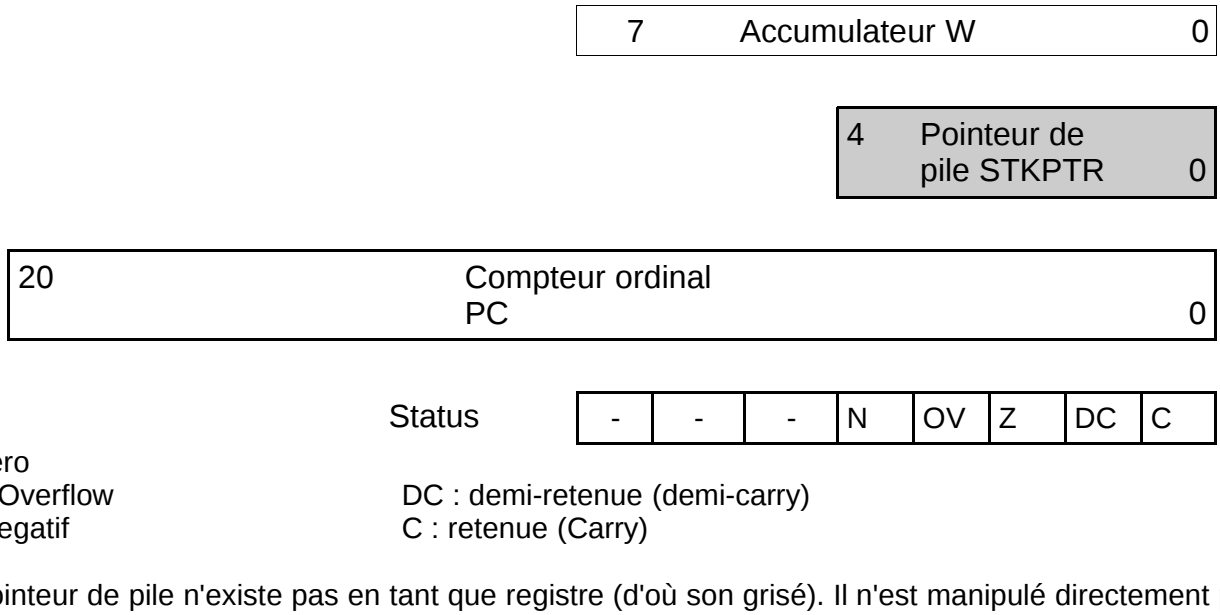
L'unité centrale connaît trois représentations des nombres :
Nombres entiers non signés (Unsigned Integer): Un octet peut avoir des valeurs entre 0 et 255 ($FF), un mot entre 0 et 65535 ($FFFF).
Nombres entiers signés complément à 2 (Two's complement). Un octet peut avoir des valeurs entre -128 ($80) et +127 ($7F), un mot entre -32768 ($8000) et 32767 ($7FFF). Le bit de poids fort indique toujours le signe.
Décimal codé binaire (BCD Binary Coded Decimal): Un octet contient deux chiffres décimaux. Les bits 7,6,5,4 contiennent le chiffre de poids fort, les bits 3,2,1,0 contiennent le chiffre de poids faible. Un octet peut donc avoir des valeurs entre 0 ($00) et 99 ($99). Comme beaucoup de micro-contrôleur, le PIC (16F84) gère ce format à travers la demi-retenue (bit DC du registre status).

Exercice 1

Convert the following decimal numbers to binary : +11, -11, -23 (two's complement representation with 8 bits). Then convert these numbers to hexadecimal.
Convert the following binary numbers to decimal : 111011, 11010101.
Perform the following binary substraction using (i) the « ordinary » substraction technique ; (ii) the two's complement method.
11011 - 10101
11100 - 1001

2°) Le modèle de programmation du PIC (18FXX8)

Il peut être présenté de la manière très simplifiée suivante :



Z : zéro
OV : Overflow
N : Negatif
DC : demi-retenue (demi-carry)
C : retenue (Carry)

Le pointeur de pile n'existe pas en tant que registre (d'où son grisé). Il n'est manipulé directement par aucune instruction (mais indirectement par PUSH et POP). Il ne comporte qu'une profondeur de 31 niveaux. Le sommet de la pile TOS (Top Of Stack) est accessible à travers 3 registres **TOSU**, **TOSH** et **TOSL**.
Le compteur programme est en fait composé de trois registres 8 bits :
- **PCL** (FF9h) pour les 8 bits de poids faibles (accessible en lecture/écriture)
- **PCH** pour les 8 bits de poids moyens (inaccessible en lecture/écriture)
- **PCU** pour les 5 bits de poids forts (inaccessible en lecture/écriture)

Lors d'une écriture dans **PCL** (movwf, addwf ...) **PCLATH** (FFAh)est transféré dans **PCH** et **PCLATU** (FFBh) dans **PCU**.

Le modèle de programmation complet est plus complexe, en particulier à cause des registres spéciaux. Il est présenté en annexe à la fin du document.

3°) Quelques instructions

Toute zone définie par l'utilisateur commence avec la DIRECTIVE CBLOCK, suivie par l'adresse du début de la zone. En ce qui concerne le PIC (18FXXXX) la zone mémoire libre commence en adresse 0x0C on aura donc

```
UDATA_ACS ; début de la zone variables
     w_temp :1      ; Zone de 1 byte
     montableau : 8 ; zone de 8 bytes
      ; Fin de la zone
```

ou

```
CBLOCK 0x00 ; début de mémoire ACCESS RAM :96 octets
     w_temp :1      ; Zone de 1 byte
     montableau : 8 ; zone de 8 bytes
ENDC  ; Fin de la zone
```

Opérations littérales (adressage direct) et de contrôles pour 18FXXXX et 16FXXX					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
GOTO k	aller à l'adresse k (sur 11bits pour les 16F et sur 20 bits pour les 18F)	2	10 1kkk kkkk kkkk 1110 1111 kkkk kkkk 1111 kkkk kkkk kkkk		

Voici en résumé le fonctionnement du goto :
- L'adresse de saut sur 20 bits est chargée dans le PC.
- La suite du programme s'effectue à la nouvelle adresse du PC.

Souvenez-vous, que pour le 18FXXXX : Adresse de saut = adresse réelle.

Opérations orientées octets entre registre et mémoire (File en anglais) pour 18FXXXX et 16FXXX					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
INCF f,d INCF f[,d[,a]]	Incrémente f	1	00 1010 dfff ffff 0010 10da ffff ffff	C,DC,N,OV,Z	1,2,3,4
DECF f,d DECF f[,d[,a]]	Décrémente f	1	00 0011 dfff ffff 0000 01da ffff ffff	C,DC,N,OV,Z	1,2,3,4

Sauf spécification contraire, d vaut toujours, au choix :
- 0 la destination est W et le contenu de l'emplacement mémoire n'est pas modifié.
- 1 la destination est f (la lettre f) : dans ce cas le résultat est stocké dans l'emplacement mémoire.

a est le RAM access bit :
- 0 l'emplacement mémoire est en access RAM
- 1 l'emplacement mémoire est en access Bank (déterminé par le registre BSR)

Exemples

```
incf mavariable , 1 ; le contenu de ma variable est augmenté de 1
                    ; le résultat est stocké dans mavariable.
incf mavariable , 0 ; Le contenu de mavariable est chargé dans w et
                    ; augmenté de 1. W contient donc le contenu de
                    ; mavariable + 1. mavariable n'est pas modifiée
```

Les modes d'adressage

Le PIC (18FXXXX) connaît neuf modes d'adressage. Nous n'en verrons qu'un dans ce TD.
Immédiat ou littéral (Immediate) : c'est l'adressage le plus facile. L'opérande se trouve directement dans le programme derrière le code de l'instruction.

Opérations littérales (adressage immédiat) et de contrôles pour 18FXXXX et 16FXXX					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
ADDLW k	Addition de W et k	1	11 1110 kkkk kkkk 0000 1111 kkkk kkkk	N,OV,C, DC,Z	
ANDLW k	Et logique de W et k	1	11 1001 kkkk kkkk 0000 1011 kkkk kkkk	N,Z	
IORLW k	OU inclusif logique de W et k		11 1000 kkkk kkkk 0000 1001 kkkk kkkk	N,Z	
MOVLW k	chargement du littéral dans W	1	11 00xx kkkk kkkk 0000 1110 kkkk kkkk		

```
0E 7F      movlw d'127'    ;charger 127 dans W
0F 10      addlw H'10'     ;additionner 16
```

Expliquer assemblage et désassemblage.

4°) Bien comprendre le modèle de programmation

Il est important de bien comprendre le fonctionnement général d'un micro-contrôleur pour bien se persuader qu'une connaissance parfaite d'un état permet de trouver ses états suivants. Pour un micro-

contrôleur on appelle état la connaissance (partielle) des valeurs des registres et de celles des mémoires.

Exercice 2 (données en hexadécimal)
On donne les états (connaissance partielle : on ne connaît pas tout, mais c'est suffisant) suivants :

	PC=120
adresses	mémoire
11F	30C6
	30FF
	3E09
	3908
	38F9
124	30FF

PC=040	***W=FF***	***DC=0***	***C=0***

03F	30FF
	30C8
	3E08
	3E40

1°) D'après le schéma donné ci-dessus, désassembler la partie utile du code.
2°) Exécuter ensuite les instructions que vous pouvez, et donner les états successifs.

II2 : TD n°2

1°) Les banques mémoires du 16F84

La mémoire du 16FXXX est organisée en banques sélectionnées par le(s) bit(s) RP0 (et RP1) du registre **STATUS**. Seul RP0 est utilisé pour le 16F84, soit deux banques.

	File Address	Banque 0	Banque 1	File Address	
	00h	Indirect addr.	Indirect addr.	80h	
`bcf status,rp0`	01h	TMR0	OPTION_REG	81h	`bsf status,rp0`
passe en	02h	PCL	PCL	82h	passe en
banque 0	03h	STATUS	STATUS	83h	banque 1
	04h	FSR	FSR	84h	
	05h	PORTA	TRISA	85h	
	06h	PORTB	TRISB	86h	
	07h	---	---	87h	
	08h	EEDATA	EECON1	88h	
	09h	EEADR	EECON2	89h	
	0Ah	PCLATH	PCLATH	8Ah	
	0Bh	INTCON	INTCON	8Bh	
	0Ch ...	68 cases mémoires SRAM	correspond à la banque 0	8Ch	
	4Fh			CFh	
	7Fh	inutilisé	inutilisé	FFh	

2°) Les banques des 18FXXXX

La série des 18F dispose de 16 banques sélectionnées par les 4 bits de poids faible du registre **BSR**. Mais le changement de banque se fait maintenant en une seule instruction :

```
movlb 0x01; en banque 1
```

Probablement pour des raisons de compatibilité, l'adressage direct sur plusieurs banques du 16F est encore présent. La terminologie microchip est loin d'être claire à ce sujet : ce qui se passait sur la série des 16F est appelé "banked" et la nouveauté du 18F "access bank". Pour éviter toute confusion dans ce document j'utiliserai "access RAM" pour le nouveau mode d'adressage et "banked "pour l'ancien mode. Le mode "access RAM" permet d'avoir 256 emplacements toujours identiques :
- de l'adresse 0 à l'adresse 0x5F on dispose de cases mémoires (soit 60h=96 cases mémoire)
- de l'adresse 0x60 à 0xFF les registres SFR (de la banque 15)

Tout ceci se traduira par l'apparition d'un a dans les instructions (voir ci-après)

3°) Les modes d'adressage (suite du TD 1)

Direct : l'adresse de l'opérande se trouve dans l'instruction.

Opérations orientées octets entre registre et mémoire (File en anglais)					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes

MOVF f,d MOVF f,d,a	déplacement de W vers f si d=1. Si d=0 déplacement W->W mais en positionnant Z	1	00 1000 dfff ffff 0101 00da ffff ffff	Z	1,2

Sauf spécification contraire, d vaut toujours, au choix :
- 0 la destination est W et le contenu de l'emplacement mémoire n'est pas modifié.
- 1 la destination est f (la lettre f) : dans ce cas le résultat est stocké dans l'emplacement mémoire.

a est le RAM access bit :
- 0 l'emplacement mémoire est en access RAM
- 1 l'emplacement mémoire est en banked (déterminé par le registre BSR)

Exemple :

```
; 96 octets en access RAM
CBLOCK 0x00 ; début de la zone variables
; CBLOCK 0x0C sur 16F84 ; voir doc section 1°)
     w_temp :1      ; Zone de 1 byte
     montableau : 8 ; zone de 8 bytes
     mavariable :1  ; zone de 1 byte
ENDC ; Fin de la zone
...
movf mavariable ,0        ;movf mavariable ,W (mavariable) -> W
; l'assembleur complète l'instruction en movf mavariable ,0, 0
```

Exercice 1
Assembler l'instruction movf du programme ci-dessus.

Exercice 2 (données en hexadécimal)

On donne l'état (connaissance partielle : on ne connaît pas tout, mais c'est suffisant) suivant :

PC=01F

C=0	***Z=0***

adresses	mémoire Prog.
01F	088C
	0A0D
021	3E01

adresses	Mémoire RAM
0C	FF

1°) D'après le schéma donné ci-dessus, désassembler la partie utile du code à l'aide du tableau d'instructions du TD1 et TD2.
2°) Exécuter ensuite les instructions que vous pouvez et donner les états successifs.
Vous disposez d'autre part d'un adressage direct long (seulement sur le 18F)

Opérations orientées octets entre registre et mémoire (File en anglais)					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
MOVFF fs,fd	déplacement de fs (source) en fd (destination)	2	1100 ffff ffff ffff (s) 1111 ffff ffff ffff (d)		

Exemple :

```
; 96 octets en access RAM
movff var1,PORTB ; Pas de probleme de banque
```

4°) Mon premier programme en assembleur

Après un reset ou un démarrage le PIC (16F84) démarre en adresse 0. L'adresse 0x004 étant réservée aux interruptions, un programme aura une structure

```
      org 0x000    ; Adresse de départ après reset
      goto    init ; Adresse 0: initialiser
      ...
init
      END
```

init est une étiquette. Tout programme se termine par la directive END.
Pour le 18F deux interruptions sont possibles, en adresse 0x008 pour la haute priorité et en adresse 0x018 pour la basse priorité

```
      ; 96 octets en access RAM
      CBLOCK 0x00    ; début de la zone variables
      ; CBLOCK 0x00C sur 16F84
           w_temp :1       ; Zone de 1 byte
           status_temp : 1 ; zone de 1 byte
           mavariable : 1  ; je déclare ma variable
      ENDC            ; Fin de la zone
      org 0x000    ; Adresse de départ après reset
      goto    start; Adresse 0: initialiser
      org 0x019    ; après l'interruption simple
      start
           clrf mavariable
      boucle
           incf mavariable,1 ;incf mavariable,f ->f
           goto boucle
      END
```

On distingue dans ce programme une étiquette, une définition symbolique, un commentaire et la définition de l'origine du programme.

5°) Les structures de contrôle

On présente maintenant les structures de contrôle dans le cas du PIC (16F84).

5-1 Structure si-alors-sinon

Cette structure est réalisée par deux instructions élémentaires de base btfsc et btfss :

Opérations orientées bits sur les registres					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
BCF f,b BCF f,b,a	mise à 0 du bit b dans f	1	01 00bb bfff ffff 1001 bbba ffff ffff		1,2
BSF f,b BSF f,b,a	mise à 1 du bit b dans f	1	01 01bb bfff ffff 1000 bbba ffff ffff		1,2
BTFSC f,b BTFSC f,b,a	test du bit b 0 de f saute si 0	1,(2)	01 10bb bfff ffff 1011 bbba ffff ffff		1,2
BTFSS f,b BTFSS f,b,a	test du bit b 0 de f saute si 1	1,(2)	01 11bb bfff ffff 1010 bbba ffff ffff		1,2

a est le RAM access bit :
- 0 l'emplacement mémoire est en access RAM
- 1 l'emplacement mémoire est en banked (déterminé par le registre BSR)

Voici un exemple dans lequel on doit exécuter une seule instruction supplémentaire si

le bit vaut 1 :

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

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

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

5-2 Structure for (boucle avec compteur)

On utilise l'une des deux instructions suivantes :

Opérations orientées octets entre registre et mémoire (File en anglais)					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
DECFSZ f,d DECFSZ f,d,a	Décrémente f (saute si 0)	1,(2)	00 1011 dfff ffff 0010 11da ffff ffff	Z	1,2,3
INCFSZ f,d INCFSZ f,d,a	Incrémente f (saute si 0)	1,(2)	00 1111 dfff ffff 0011 11da ffff ffff	Z	1,2,3

d vaut toujours, au choix :
- 0 la destination est W et le contenu de l'emplacement mémoire n'est pas modifié.
- 1 la destination est f (la lettre f) : dans ce cas le résultat est stocké dans l'emplacement mémoire.

a est le RAM access bit :
- 0 l'emplacement mémoire est en access RAM
- 1 l'emplacement mémoire est en banked (déterminé par le registre BSR)

Voici un exemple qui utilise un compteur de boucle :

```
       movlw 3              ; charger 3 dans w
       movwf compteur       ; initialiser compteur
       movlw 0x5            ; charger 5 dans w
boucle                      ; étiquette
       addwf mavariable , 1 ; ajouter 5 à ma variable
       decfsz compteur , 1  ; décrémenter compteur et tester sa valeur
       goto boucle          ; si compteur pas 0, on boucle
       movf mavariable , 0  ; on charge la valeur obtenue dans w
```

5-3 Structure répéter ... tant que ... et structure tant que...

Par exemple :

```
     ; tant que C=0 on boucle
       movlw 0x5            ; charger 5 dans w
boucle                      ; étiquette
       addwf mavariable , 1 ; ajouter 5 à ma variable
       btfsc STATUS,C    ; tester si le bit C du registre STATUS vaut 0
        goto suite        ; non (C=1), alors aller à suite
       goto boucle       ; oui, on boucle
suite
       movf mavariable , 0  ; on charge la valeur obtenue dans w
```

On peut mieux faire avec un seul goto :

```
     ; tant que C=0 on boucle
       movlw 0x5            ; charger 5 dans w
boucle                      ; étiquette
       addwf mavariable , 1 ; ajouter 5 à ma variable
       btfss STATUS,C    ; tester si le bit C du registre STATUS vaut 1
        goto boucle       ; non (C=0), alors aller à boucle
       movf mavariable , 0  ; on charge la valeur obtenue dans w
```

Exercice 3

On suppose qu'en **PORTB** se trouvent des valeurs positives (octets) provenant de l'extérieur du PIC (18FXXXX). Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. Faire un programme qui fait sans arrêt l'acquisition de 4 de ces valeurs, fait la somme et divise par 4 le résultat pour le ranger dans une variable moyenne.

Indications
La difficulté de ce problème réside dans le fait que l'on a à additionner un ensemble de valeur 8 bits qui peut donner un résultat sur plus de 8 bits. La division par 4 sera fait avec décalages vers la droite.

II2 TD n°3

I) L'adressage indirect du 16FXXX

Cet adressage fait appel à 2 registres, dont un est particulier, car il n'existe pas vraiment. Examinons-les donc :
Les registres **FSR** et **INDF**
INDF signifie INDirect File. C'est le fameux registre de l'adresse 0x00. Ce registre n'existe pas vraiment, ce n'est qu'un procédé d'accès particulier à **FSR** utilisé par le PIC® pour des raisons de facilité de construction électronique interne.
Le registre **FSR** est à l'adresse 0x04 dans les 2 banques. Il n'est donc pas nécessaire de changer de banque pour y accéder, quelle que soit la banque en cours d'utilisation.
Le contenu du registre **FSR** pointe sur une adresse en 8 bits. Or, sur certains PIC®, la zone RAM contient 4 banques (16F876). L'adresse complète est donc une adresse sur 9 bits. Le registre **FSR** est alors complété alors par le bit **IRP** du registre status.

Exemple :

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

Les puristes peuvent se demander pourquoi ne pas écrire directement

```
movf SFR,w
```

au lieu de

```
movf INDF,w
```

C'est tout simplement parce que `movf FSR,w` existe mais a une autre signification, mettre le registre **FSR** dans w ! On a bien besoin d'un pseudo registre pour l'adressage indirect.

II) L'adressage indirect du 18FXXX

Le concept précédent a été étendu dans les PIC 18F pour pouvoir accéder à de plus grandes quantités de mémoires. Il y a maintenant 3 registres **FSR**. Puisqu'ils doivent permettre d'adresser toute la mémoire ils doivent avoir 12 bits chacuns, d'où la présence de **FSR2H:FSR2L**, **FSR1H:FSR1L** et **FSR0H:FSR0L**. Pour compliquer un peu les choses, il y a cinq équivalent à **INDF** : **INDFn**, **POSTINCn**, **POSTDECn**, **PREINCn** et **PLUSWn**. Evidemment cette complexité est faite pour simplifier la construction des compilateurs sur cette architecture.

Exemple :

```
movlw LOW(mavariable)
movwf FSR0L
movlw HIGH(mavariable)
movwf FSR0H
movf  INDF0,w     ; charger le CONTENU de INDF0 dans W.
```

peut être remplacé par :

```
      lfsr FSR0,mavariable
      movf  INDF0,w     ; charger le CONTENU de INDF0 dans W.
```

Microchip a donc ajouté une instruction pour faciliter ce mode d'adressage.

III) Les tableaux

Les tableaux se gèrent avec cet adressage indirect naturellement. Contrairement à ce qui se passe en langage C, il faut gérer la mémoire des tableaux vous-même. On se limitera aux cas de la série des 16FXXX plus simple.

1°) Les tableaux en mémoire programme

Comme aucune instruction ne permet de lire un octet en mémoire programme, l'astuce est d'utiliser l'instruction retlw qui permet de retourner une valeur passée en argument. Ecrivons donc notre tableau sous forme de retlw (voir TD 4) : cela donne :

```
      retlw      0              ;  premier élément = 0
      retlw      1              ;  deuxième élément = 1
      ...
      retlw      225            ;  nième élément = 225
```

Nous pouvons ensuite utiliser **PCL** (poids faible du compteur programme) pour accéder à ce tableau. Si l'index d'accès au tableau est dans W

```
tableau
      addwf PCL , f ; ajouter w à PCL
      retlw      0              ;  premier élément = 0
      retlw      1              ;  deuxième élément = 1
      ...
      retlw      225            ;  nième élément = 225
```

Donc, si on charge 1 dans W et qu'on effectue un appel « call tableau ». Le programme se branche sur la bonne ligne, le **PCL** est incrémenté de 1 et le programme exécute donc alors la ligne « retlw 1 ». Si l'origine du tableau est au-delà des 8 bits, par exemple :

```
ORG 0x300
tableau
      addwf PCL , f ; ajouter w à PCL
      retlw      0              ;  premier élément = 0
```

ne pas oublier de positionner **PCLATH** correctement. Pour l'exemple ci-dessus, cela donne :

```
      movlw 03
      movwf PCLATH
```

2°) Tableau en RAM

On complète l'adressage indexé de la section I

```
      movf  INDF,w  ; W<-(FSR)
```

avec

```
      movwf INDF     ;(FSR)<-W
```

Nous pouvons ainsi utiliser l'adressage indexé dans les deux directions.

IV) Exercices

1) On suppose qu'en **PORTB** se trouvent des valeurs (octets) provenant de l'extérieur du PIC (16F84). Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. Faire un programme qui fait l'acquisition de 4 de ces valeurs et les stocke dans un tableau. Ensuite on calculera la somme de ces 4 cases divisée par 4 et le résultat sera rangé à l'adresse $14.

2) Modifier ce programme pour qu'il calcule le maximum des 4 valeurs du tableau et le range en mémoire à la suite du tableau.

3) Modifier ce programme pour que le passage d'un tableau de 4 cases à un tableau de 10 (ou autre) cases se fasse seulement par la changement d'une seule valeur dans votre programme.

II2 TD n°4

1°) Les sous-programmes le switch

Les instructions CALL et RETLW et RETURN sont pour l'appel de sous-programme.

Opérations littérales (adressage direct) et de contrôles					
Mnémonique Opérande	Description	Cycles	14/16 bits Opcode	status affected	notes
CALL k	appel du sous programme k	2	10 0kkk kkkk kkkk		
CALL n [,s]	appel du sous programme n (sur 20 bits) (s fast bit)	2	1110 110s kkkk kkkk 1111 kkkk kkkk kkkk		
GOTO k	aller à l'adresse k (sur 11bits pour les 16F et sur 20 bits pour les 18F)	2	10 1kkk kkkk kkkk 1110 1111 kkkk kkkk 1111 kkkk kkkk kkkk		
RETLW k	retour avec le littéral dans W	2	11 01xx kkkk kkkk 0000 1100 kkkk kkkk		
RETURN	retour de sous-programme	2	00 0000 0000 1000		
RETURN s	retour de sous-programme	2	0000 0000 0001 001s		

s=0 pas de restauration automatique des registres **W**, **STATUS** et **BSR**.

Le rôle de la pile pour les appels imbriqués
Avec le PIC 16F on ne peut pas abuser des sous-programmes car il n'y a que 8 niveaux d'imbrications. Pour le 18F les choses se compliquent un peu : un pile d'adresse de retour a 31 niveaux mais aussi une autre pile pour les registres **W**, **STATUS** et **BSR** (Fast Register Stack) d'où la présence du bit s dans les instructions d'appel et de retour.

La programmation de suivant le cas faire... peut se faire de différentes manières. Nous commençons par en présenter une :

```
      CBLOCK 0x00    ; début de la zone variables en ACCESS RAM
            w_temp :1       ; Zone de 1 byte
            status_temp : 1 ; zone de 1 byte
            Choix : 1  ; je déclare ma variable
      ENDC            ; Fin de la zone
      ...
;si Choix=0 faire le sous programme spgm0 si Choix=1 faire spgm1
      ; Choix contient ici une valeur <=N
      movlw 0x0
      subwf Choix,w          ;Choix - W -> W
      btfsc STATUS,Z         ;Z=1 si = 0 et on saute
      goto spgm0
      movlw 0x1
      subwf Choix,w          ;Choix - W -> W
      btfsc STATUS,Z         ;Z=1 si = 0 et on saute
      goto spgm1
      ...
      movlw 0xN
      subwf Choix,w          ;Choix - W -> W
      btfsc STATUS,Z         ;Z=1 si = 0 et on saute
      goto spgmN
      ; erreur ici !!!
suite
```

```
      ....

spgm0       ;code ici
      goto suite
      ....
```

Remarque : les sous-programmes ne sont pas ici de vrais sous-programmes (appelés avec goto et non call finissant par goto au lieu de return). D'autre part, cette technique ne traite pas le dépassement de l'index signalé par "erreur ici". A vous d'être créatifs...

2°) suivant le cas faire... avec bits de registres

Sans rentrer dans le détail des interruptions (voir bimestre B4) il s'agit d'un domaine où il est utile de réaliser un suivant le cas faire... à partir du test de bits d'un (ou plusieurs) registres. Ceci est réalisé de la manière suivante (avec de vrais appels de sous-programmes) :

```
      ; switch vers différentes interrupts
      ; inverser ordre pour modifier priorités
      ;--------------------------------------
      btfsc INTCON,T0IE        ; tester si interrupt timer autorisée
      btfss INTCON,T0IF        ; oui, tester si interrupt timer en cours
      goto  intsw1             ; non test suivant
      call  inttimer           ; oui, traiter interrupt timer
      goto  restorereg         ; et fin d'interruption
intsw1
      btfsc INTCON , INTE  ; tester si interrupt RB0 autorisée
      btfss INTCON , INTF  ; oui, tester si interrupt RB0 en cours
      goto  intsw2         ; non sauter au test suivant
      call  intrb0         ; oui, traiter interrupt RB0
      bcf   INTCON,INTF    ; effacer flag interupt RB0
      goto  restorereg     ; et fin d'interruption
intsw2
      btfsc INTCON,RBIE    ; tester si interrupt RB4/7 autorisée
      btfss INTCON,RBIF    ; oui, tester si interrupt RB4/7 en cours
      goto  intsw3         ; non sauter
      call  intrb4         ; oui, traiter interrupt RB4/7
      bcf   INTCON,RBIF    ; effacer flag interupt RB4/7
      goto  restorereg     ; et fin d'interrupt
intsw3
       ; ici, la place pour l'interruption eeprom
```

3°) Exercices

Exercice 1
Modifier le premier exemple du « suivant le cas faire ... » pour que ce que l'on doive faire se trouve dans des sous-programmes. Le fait que les choix se font sur une instruction complique un peu la situation.
Modifier le programme en utilisant une technique similaire aux tableaux en mémoire programme.

Exercice 2
Écrire un programme qui lit sans arrêt le **PORTB** pour le mettre dans Choix et qui suivant la valeur décimale (1, 2, 4, 8, 16, 32, 64, 128) appelle un des sous programme do, re, mi, fa, sol, la, si et do2. On ne vous demande pas de détailler les sous-programmes .

Exercice 3
Écrire un programme qui, suivant la valeur présente en **PORTB** soit fait la somme des cases d'un tableau de 4 cases, soit calcule la moyenne de ces cases, soit cherche le maximum du tableau.

II2 - TD n°5

Nous allons poursuivre par un autre thème : les fonctions logiques booléennes.

Exercice
Given a truth table, find differents methods to perform this boolean function (input address $1000, output address $1004).

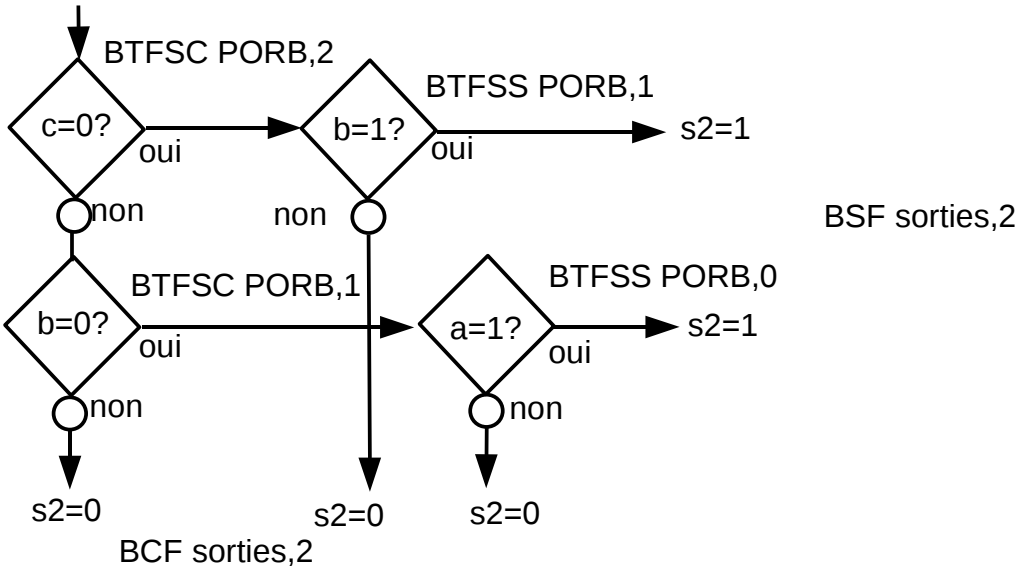
c	*b*	*a*	*s2*	*s1*	*s0*
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	0	1	1

1°) Show that we can perform this boolean function with an array. Write the complete program which have to initilize this array.
2°) Show that we can perform this boolean function with use of logical operation instructions
or Branch Instructions.

Write a program for s2 with branch instructions.

Write a program for s1 with logical operation instructions.

Hint :



Principe : chaque test est suivi par deux goto, celui du non puis celui du oui.

II2 TD n°6

1°) Arithmétique binaire et expressions en C PIC (16F84)

Pour bien comprendre la signification des expressions, il est essentiel d'avoir 2 notions en tête : la priorité et l'associativité. Nous donnons ci-après un tableau des opérateurs par priorité décroissante :

Catégorie d'opérateurs	Opérateurs	Associativité
fonction, tableau, membre de structure, pointeur sur un membre de structure	( ) [ ] . ->	Gauche -> Droite
opérateurs unaires	- ++ -- ! ~ * & sizeof (type)	Droite ->Gauche
multiplication, division, modulo	* / %	Gauche -> Droite
addition, soustraction	+ -	Gauche -> Droite
décalage	<< >>	Gauche -> Droite
opérateurs relationnels	< <= > >=	Gauche -> Droite
opérateurs de comparaison	== !=	Gauche -> Droite
et binaire	&	Gauche -> Droite
ou exclusif binaire	^	Gauche -> Droite
ou binaire	|	Gauche -> Droite
et logique	&&	Gauche -> Droite
ou logique	||	Gauche -> Droite
opérateur conditionnel	? :	Droite -> Gauche
opérateurs d'affectation	= += -= *= /= %= &= ^= |= <<= >>=	Droite -> Gauche
opérateur virgule	,	Gauche -> Droite

Exercice 1

Enlever les parenthèses des expressions suivantes lorsqu'elles peuvent être retirées.
a=(25*12)+b;
if ((a>4) &&(b==18)) { }
((a>=6)&&(b<18))||(c!=18)
c=(a=(b+10));
Evaluer ces expressions pour a=6, b=18 et c=24

Les types du C PIC sont :

```
unsigned char a;//8 bits, 0 to 255
signed char b;  //8 bits, -128 to 127
unsigned int c; //16 bits, 0 to 65535
signed int d;   //16 bits, -32768 to 32767
long e;         //32 bits, -2147483648 to 2147483647
float f;    //32 bits
```

Exercice 2

b7	*b6*	*b5*	*b4*	*b3*	*b2*	*b1*	*b0*

Si une variable p1 de type signed char (8 bits signés) est déclarée écrire les expressions en C permettant de :
- mettre à 1 le bit b2
- mettre à 1 le bit b3 et b6
- mettre à 0 le bit b0
- mettre à 0 le bit b4 et b5
- inverser le bit b3 (se fait facilement avec un ou exclusif)
- mettre à 1 le bit b2 et à 0 le bit b0
- mettre à 1 les bits b0 et b7 et à 0 les bits b3 et b4

Exercice 3

On donne le sous-programme suivant (tiré d'un ancien projet inter-semestre) :

```
void conversion(char nb,char result[8]){
  char i;
  for(i=7;i>=0;i--) if ((nb & (1<<i)) == (1<<i)) result[i]=1;
                    else result[i]=0;
}
```

1) Peut-on retirer des parenthèses dans l'expression booléenne du if ?
2) Peut-on écrire `(nb & (1<<i))` au lieu de `((nb & (1<<i)) == (1<<i))`?
3) Construire l'expression booléenne qui permettrait l'écriture

```
for(i=7;i>=0;i--) if (E.B.?????) result[i]=0;
                    else result[i]=1;
```

en donnant toujours le même le même résultat de conversion.
4) Modifier le programme pour qu'il fasse une conversion d'entier (16 bits) vers le binaire.
5) Est-ce que l'algorithme suivant donne le même résultat de conversion :

```
 for(i=0;i<8;i++) {
      result[i]=nb%(2);
      nb = nb / 2;
  }
```

Exercice 4

Soit une variable :

```
      char nb;
```

Écrire les expressions permettant de calculer les centaines, les dizaines et les unité de cette variable.

Il existe plusieurs autres méthodes pour positionner les bits
(http://www.microchipc.com/HiTechCFAQ/index.php)
1° méthode pour positionner bit à bit :

```
      unsigned char x=0b0001;
      bit_set(x,3); //now x=0b1001;
      bit_clr(x,0); //now x=0b1000;*/
      #define bit_set(var,bitno) ((var) |= 1 << (bitno))
      #define bit_clr(var,bitno) ((var) &= ~(1 << (bitno)))
```

2° méthode pour positionner plusieurs bits

```
      unsigned char x=0b1010;
      bits_on(x,0b0001);   //now x=0b1011
      bits_off(x,0b0011);  //now x=0b1000 */
      #define bits_on(var,mask) var |= mask
      #define bits_off(var,mask) var &= ~0 ^ mask
```

II2 TD n°7

1°) Expression booléenne vraie (PIC 18FXXXX et autres)

Dans la suite du cours on appellera vraie expression booléenne une expression qui si elle était affectée à une variable donnerait soit la valeur 0 ou soit la valeur 1. Le C est un langage qui permet de fabriquer de fausses expressions booléennes. C'est très pratique, mais la confusion des deux peut conduire à des erreurs. Donnons un exemple :

Fausse expression booléenne	*Vraie expression booléenne*

`char n=10;`
`while(n) {`
`.....;`
`n--; }`
	`char n=10;`
	`while(n!=0) {`
	`.....;`
	`n--; }`

Ces deux constructions marchent en C. Si on se rappelle que dans une parenthèse d'un while on a une Expression Bolléenne (E.B.), d'un côté n et de l'autre n!=0 sont des E.B. La différence entre les deux est que pour n elle peut prendre toutes les valeurs entre -128 et +127 (car n est de type char) alors que n!=0 ne prendra que deux valeurs 0 ou 1. La deuxième sera donc appelée E.B. vraie et la première E.B.
Le problème du langage C est que l'oubli d'un & ou d'un | conduit en général à une fausse expression booléenne ce qui est tout à fait autorisé. Par exemple écrire a & b au lieu de a && b.

Exercice 1
Différence entre && et &
Évaluer les expressions :
a&b
a&&b
pour a= 0xF0 et b=0x0F
En déduire les valeurs booléennes correspondantes (si ces expressions étaient utilisées dans un if par exemple).
Construire des vraies expressions booléennes sur les tests suivants
expression vraie si
le bit b6 est à 1
le bit b3 est à 0
le bit b6 est l'inverse du bit b3
le bit b2 est à 1 et le bit b4 est à 0
le bit b2 est à 1 ou le bit b7 est à 0

Les tests d'un bit particulier en C peuvent aussi être réalisés de la manière suivante (http://www.microchipc.com/HiTechCFAQ/index.php)

```
x=0b1000; //decimal 8 or hexadecimal 0x8
if (testbit_on(x,3)) a(); else b(); //function a() gets executed
if (testbit_on(x,0)) a(); else b(); //function b() gets executed
if (!testbit_on(x,0)) b(); //function b() gets executed
#define testbit_on(data,bitno) ((data>>bitno)&0x01)
```

Exercice 2

Quelle opération arithmétique est réalisée par un décalage ? Évaluer pour cela les expressions suivantes (avec a=12 et b=23) :
- a = a >> 1 (ou a >>= 1)
- a = a >> 2 (ou a >>= 2)
- b = b << 1 (ou b <<=1)
- b = b << 2 (ou b <<=2)

Généralisation.
Construire une vraie expression booléenne avec opérateur de décalage, & et ^ qui reprend le test de l'exercice précédent : le bit b6 est l'inverse du bit b3

Exercice 3

Soit le programme suivant :

```
#include <stdio.h>
main() {
  int n=10,p=5,q=10,r;
  r= n == (p = q);
  printf("A : n = %d p = %d q= %d r = %d\n",n,p,q,r);
  n = p = q = 5;
  n += p += q;
  printf("B : n = %d p = %d q= %d\n",n,p,q);
  q = n++<p || p++ != 3;
  printf("C : n = %d p = %d q= %d\n",n,p,q);
  q = ++n == 3 && ++p == 3;
  printf("D : n = %d p = %d q= %d\n",n,p,q);
  return 0;
}
```

Que donnera-t-il comme affichage sur l'écran ?

TD8 - Les ports A et B (PIC® 16F et 18F)

1°) Généralités

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset
FF1h	INTCON2	/RBPU	INTEDG0	INTEDG1	-	-	TMR0IP	-	RBIP	111- -1-1
F93h	TRISB									1111 1111
F92h	TRISA	-								-111 1111
F8Ah	LATB									xxxx xxxx
F89h	LATA	-								-xxx xxxx
F81h	PORTB	RB7/PGD	RB6/PGC	RB5/PGM	RB4	RB3/ CANRX	RB2/CAN TX/INT2	RB1/INT1	RB0/INT0	xxxx xxxx
F80h	PORTA	-	RA6/CLK O/OSC2	RA5/ AN4	RA4/ TOCKI	RA3/AN3 /Vref+	RA2/AN 2/Vref-	RA1/ AN1	RA0/AN0/ CVref	-x0x 0000

La manipulation des bits de registre dépend des compilateurs utilisés. Nous présentons un exemple pour comprendre :

MikroC	C18

`INTCON.TMR0IF = 0; // Clear TMR0IF`
 	
`// If RB0 is set, set RC0:`
`if (PORTB.F0) PORTC.F0 = 1;`
	`INTCONbits.TMR0IF=0;// Clear TMR0IF`
	
	`// If RB0 is set, set RC0:`
	`if (PORTBbits.RB0) PORTCbits.RC0 = 1;`

Les masques peuvent être utilisés pour écrire du code portable : si le compilateur est bien fait il optimisera avec les instructions assembleurs correspondantes.
Les PORTS fonctionnent toujours sur le principe
lecture->modification->écriture.
Par exemple, si vous écrivez bsf PORTA,1, le PIC® procède de la manière suivante :
1) Le **PORTA** est lu en intégralité (pins en entrée et en sortie)
2) Le bit 1 est mis à 1
3) Tout le **PORTA** est réécrit (concerne les pins en sortie).
Ainsi, supposons par exemple que le RA4 soit en sortie et mis à 1. Comme il est à drain ouvert, si une électronique externe le force à 0, si vous effectuez l'opération suivante :

```
bsf          PORTA   , 1          ; mettre RA1 à 1
```

Lorsque le **PORTA** va être lu, RA4 sera lu comme 0, bien qu'il soit à 1. RA1 sera forcé à 1, et le tout sera replacé dans **PORTA**. RA4 est donc maintenant à 0, sans que vous l'ayez explicitement modifié. Pour éviter ce problème propre aux 16F, les 18F ont un registre supplémentaire **LATA** et **LATB** associés aux **PORTA** et **PORTB** qui contiennent toujours les valeurs correctes.

2°) Le registre TRISA

Ce registre est situé à l'adresse 0xF92. Ce registre est d'un fonctionnement très simple et est lié au fonctionnement du **PORTA**.

Chaque bit positionné à 1 configure la pin correspondante en entrée Chaque bit à 0 configure la pin en sortie

Au reset du PIC®, toutes les pins sont mises en entrée, afin de ne pas envoyer des signaux non désirés sur les pins. Les bits de **TRISA** seront donc mis à 1 lors de chaque reset.
Notez également que, comme il n'y a que 7 pins utilisées sur le **PORTA**, seuls 7 bits (b0/b6) seront utilisés sur **TRISA**. Les bits de **TRISA** sont désignés par TRISAbits.TRISA0 ... TRISAbits.TRISA7

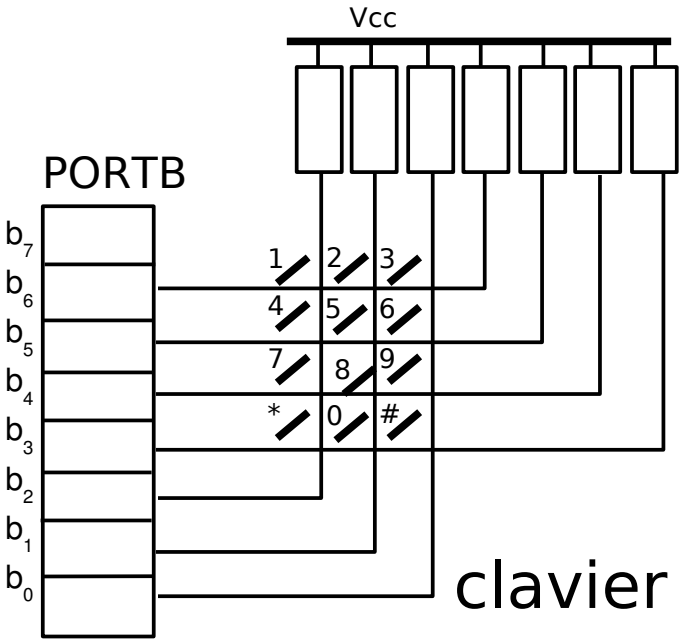
3°) Les registres PORTB et TRISB

Ces registres fonctionnent exactement de la même manière que **PORTA** et **TRISA**, mais concernent bien entendu les 8 pins RB. Tous les bits sont donc utilisés dans ce cas.
Voyons maintenant les particularités du **PORTB**. Les entrées du **PORTB** peuvent être connectées à une résistance de rappel au +5V de manière interne, cette sélection s'effectuant par le bit 7 du registre **OPTION** (effacement du bit7 RBPU pour valider les résistances de rappel au +5V).
Les bits de **TRISB** sont désignés par TRISBbits.TRISB0 ... TRISBbits.TRISB7

4°) Interfaçage d'un clavier

Sur un PC, le clavier est complètement décodé. C'est à dire que lorsqu'une touche est appuyée, sa position sur le clavier est envoyée sur la liaison PS2. Le fait d'envoyer la position et non le code ASCII permet de gérer les claviers en divers langues.
Pour de petites applications, on utilise un clavier à 12 touches. Il est composé de simples contacts et le décodage est réalisé par le système informatique. Avec seulement 8 touches, un PORT de 8 bits en entrée suffit. Si le clavier possède plus de 8 touches, il faut:
- soit utiliser d'avantage d'entrées,
- soit multiplexer les entrées en deux étapes.

En utilisant 4 fils pour les lignes et 4 fils pour les colonnes, on peut différencier par croisement 16 touches. On utilise donc 8 fils reliés à 8 bits d'un PORT pour 16 touches. Pour nos 12 touches on peut câbler comme indiqué ci-dessus. Il s'agit ensuite de procéder en deux phases, une pour la détection de la colonne et une autre pour la détection de ligne.



Question 1 : détermination du numéro de colonne
Programmer les directions avec **TRISB** (PB6-PB3 en sortie et PB2-PB0 en entrée).
Tester si une touche est appuyée. Si oui sauvegarder dans une variable la valeur lue sur **PORTB** puis transformer cette valeur en numéro de colonne (0 à gauche et 2 à droite)
Quel est le code correspondant : sous-programme `char lecture_colonne()`

Question 2 : détermination du numéro de ligne
Programmer les directions avec **TRISB** (PB6-PB3 en entrée et PB2-PB0 en sortie).
Tester si une touche est appuyée. Si oui sauvegarder dans une variable la valeur lue sur **PORTB** puis transformer cette valeur en numéro de ligne (0 en haut et 3 en bas)
Quel est le code correspondant : sous-programme `char lecture_ligne()`

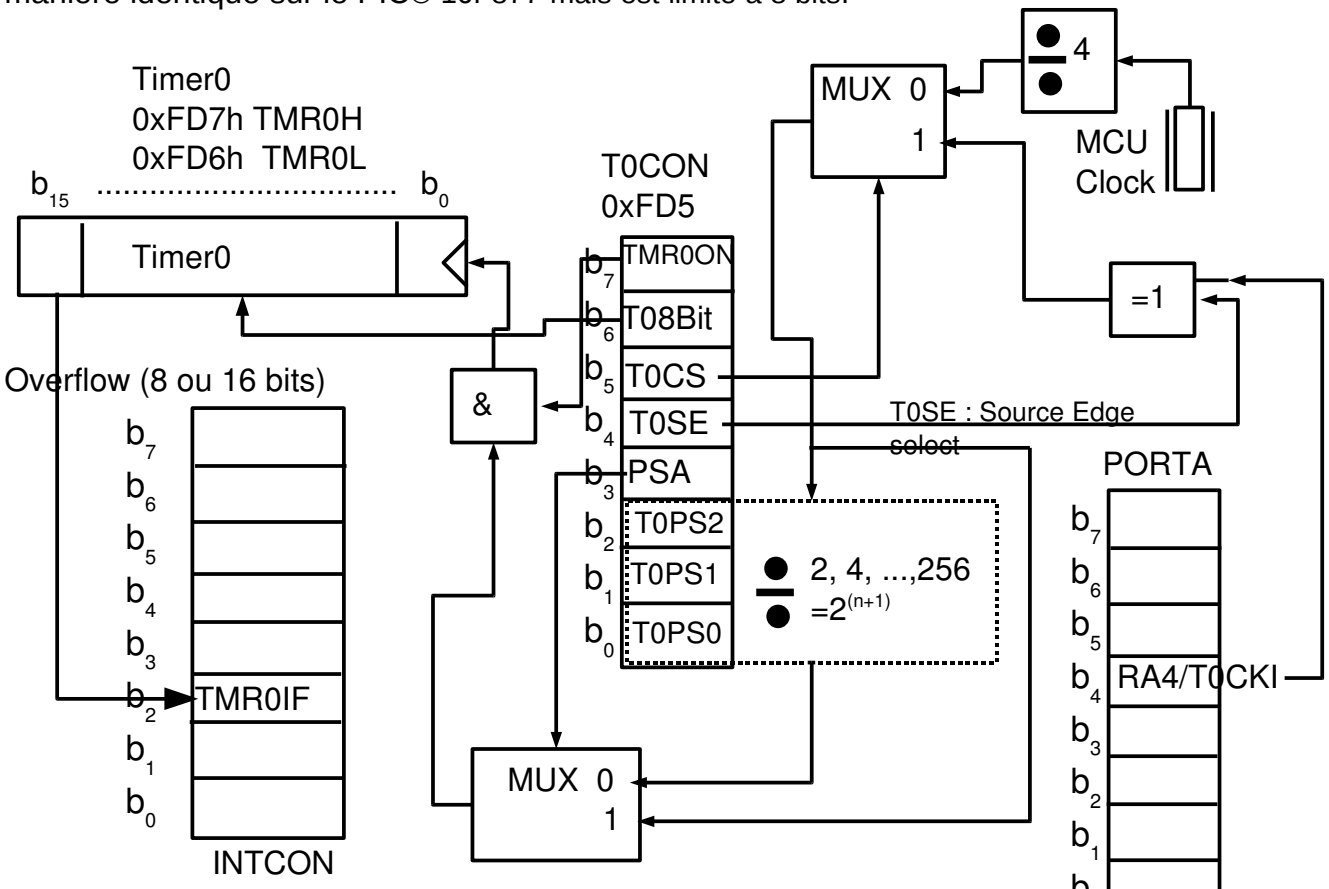
Question 3 : détermination du caractère
A partir des deux informations précédentes transformer le numéro de colonne et le numéro de ligne en caractère correspondant sur le clavier : '1' ou '2' ou ... ou '0' ou '#'

Remarque : les bits RB4-RB7 peuvent servir à déclencher une interruption (RB port change Interrupt). Si l'on veut utiliser cette interruption il faudrait câbler notre clavier autrement. Les interruptions seront abordées plus loin.

TD 9 : Le timer0 des PIC® 18F4550 et PIC® 16F877

I) Le timer 0

La documentation du Timer0 du PIC® 18F4550 est présentée maintenant. Il fonctionne de manière identique sur le PIC® 16F877 mais est limité à 8 bits.



Timer0 dans 18F4550

II) Les différents modes de fonctionnement

Le timer0 est en fait un compteur. Mais qu'allez-vous compter avec ce timer ? Et bien, vous avez deux possibilités :

- En premier lieu, vous pouvez compter les impulsions reçues sur la pin RA4/T0CKI. Nous dirons dans ce cas que nous sommes en mode compteur
- Vous pouvez aussi décider de compter les cycles d'horloge du PIC® lui-même. Dans ce cas, comme l'horloge est fixe, nous compterons donc en réalité du temps. Donc, nous serons en mode « timer ».

La sélection d'un ou l'autre de ces deux modes de fonctionnement s'effectue par le bit 5 du registre **T0CON** : T0CS pour Tmr0 Clock Source select bit.
T0CS = 1 : Fonctionnement en mode compteur
T0CS = 0 : Fonctionnement en mode timer
Dans le cas où vous décidez de travailler en mode compteur, vous devez aussi préciser lors de quelle transition de niveau le comptage est effectué. Ceci est précisé grâce au bit 4 du registre **T0CON** : T0SE pour Timer0 Source Edge select bit.
T0SE = 0 : comptage si l'entrée RA4/TOKI passe de 0 à 1 (front montant)
T0SE = 1 : comptage si l'entrée RA4/TOKI passe de 1 à 0 (front descendant)

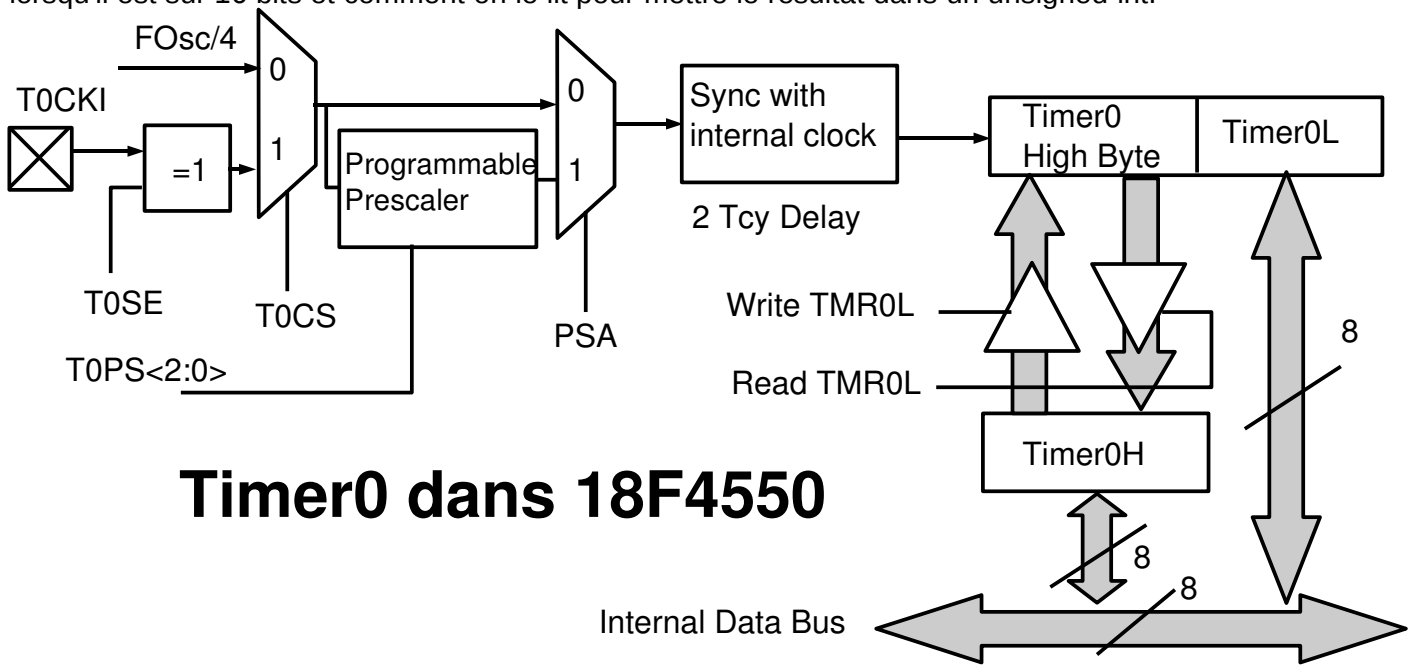
III) Mesure du temps d'exécution d'un algorithme

L'optimisation d'un algorithme en vitesse (ou en taille) est très importante dans les systèmes embarqués réalisés par des micro-contrôleurs. Une recherche d'algorithmes sur Internet vous donnera des résultats qu'il vous faudra évaluer. Par exemples, le site :
http://www.piclist.com/techref/language/ccpp/convertbase.htm
vous propose un algorithme de division par 10 que voici :

```
unsigned int A;
unsigned int Q; /* the quotient */
        Q = ((A >> 1) + A) >> 1; /* Q = A*0.11 */
        Q = ((Q >> 4) + Q)     ; /* Q = A*0.110011 */
        Q = ((Q >> 8) + Q) >> 3; /* Q = A*0.00011001100110011 */
        /* either Q = A/10 or Q+1 = A/10 for all A < 534,890 */
```

Exercice 1

1°) Sans chercher à comprendre l'algorithme de division, on vous demande de le transformer en une fonction `unsigned int div10(unsigned int A);`
2°) A l'aide de la documentation complémentaire ci-dessous, préciser comment on initialise le timer0 lorsqu'il est sur 16 bits et comment on le lit pour mettre le résultat dans un unsigned int.



3°)Initialiser le timer0 pour une division par 256 de la fréquence du quartz et un travail sur 16 bits.

IV) Le mode de scrutation du flag

Nous devons savoir à ce niveau, que tout débordement du timer0 (passage de 0xFF à 0x00 en mode 8 bits ou de 0xFFFF à 0x0000 en mode 16 bits) entraîne le positionnement du flag T0IF, bit b_2 du registre **INTCON**. Vous pouvez donc utiliser ce flag pour déterminer si vous avez eu débordement du timer0, ou, en d'autres termes, si le temps programmé est écoulé. Cette méthode à l'inconvénient de vous faire perdre du temps inutilement dans une boucle d'attente.
Petit exemple :

```
     clrf       tmr0                ; début du comptage dans 2 cycles
                                    ; (voir remarque plus bas)
     bcf        INTCON , T0IF       ; effacement du flag
loop
     btfss      INTCON , T0IF       ; tester si compteur a débordé
     goto       loop                ; non, attendre débordement
     xxx                            ; poursuivre : 256/65536 événements écoulés
```

Exercice 2
Question 1

Écrire en langage C un programme qui fait la même chose que le programme assembleur ci-dessus : initialise le timer0, efface le flag et attend, à l'aide d'une boucle, le positionnement de ce dernier.

Mais vous pourriez vous dire que vous ne désirez pas forcément attendre 256 incrémentations de tmr0. Supposons que vous désiriez attendre 100 incrémentations. Il suffit dans ce cas de placer dans tmr0 une valeur telle que 100 incrémentations plus tard, tmr0 déborde.
exemple :

```
       ; timer0 en mode 8 bits
     movlw       256-100             ; charger 256 – 100
     movwf       tmr0                ; initialiser tmr0
     bcf         INTCON,T0IF         ; effacement du flag
loop
     btfss       INTCON,T0IF         ; tester si compteur a débordé
     goto        loop                ; non, attendre débordement
     xxx                             ; oui, poursuivre : 100 événements écoulés
```

Question 2
Écrire en langage C un programme qui fait la même chose que le programme assembleur ci-dessus : initialise le timer0, efface le flag et attend à l'aide d'une boucle le positionnement de ce dernier.
Remarque : ceci pourrait sembler correct, mais en fait toute modification de TMR0 entraîne un arrêt de comptage de la part de celui-ci correspondant à 2 cycles d'instruction multipliés par la valeur du pré diviseur. Autrement dit, un arrêt correspondant toujours à 2 unités TMR0. Il faut donc tenir compte de cette perte, et placer « 256-98 » et non « 256-100 » dans le timer.

Question 3
Générer un signal de fréquence 1 KHz (avec un quartz de 4 MHz). Pour cela :
- calculer la valeur de la pré division
- calculer la valeur de décomptage
- Écrire le programme.

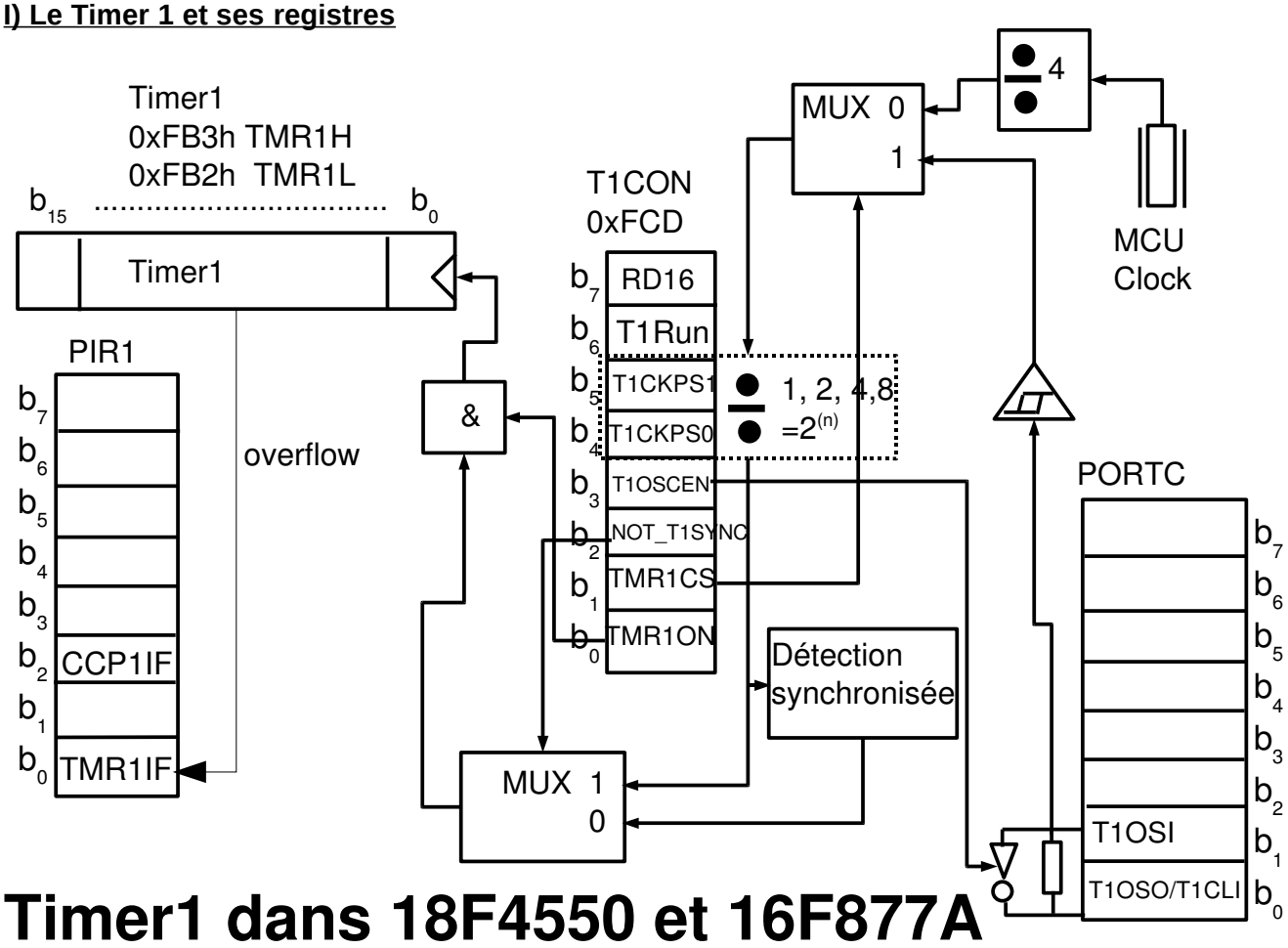
Question 4
Même chose mais il faut que la période diminue progressivement

Question 5
Générer un signal de sortie de rapport cyclique 1/4 sur le même principe.
Il y a mieux à faire avec les PICs, utiliser le module CCP ou le module ECCP.

TD 10 : Le mode comparaison du module CCP (Capture/Compare/PWM)

L'objectif de ce TD est de générer des signaux de fréquences déterminées. Ce travail est réalisé en général avec le module CCP (Capture/Compare/PWM) et plus exactement son mode comparaison. Nous allons commencer par décrire le fonctionnement du timer 1 du 18F4550 et 16F877.

I) Le Timer 1 et ses registres



Timer1 dans 18F4550 et 16F877A

Remarque : les adresses des registres présentes dans le dessin ne correspondent pas à celles du PIC 16F877 mais au 18F4550. Ces adresses sont données pour information et n'ont que peu d'utilité pour la programmation.

Les registres utilisés par le Timer1 sont donc :
- **TMR1H** et **TMR1L**,
- **T1CON**
- éventuellement le **PORTC**, et **PIR1**.

La détection synchronisée doit être obligatoirement utilisée dans les modes compare (de ce TD) et le mode capture du TD suivant.

Exercice 1
The PIC® has an oscillator frequency of 4 MHz. What is the slowest Timer 1 interrupt rate it can generate with this internal clock ?

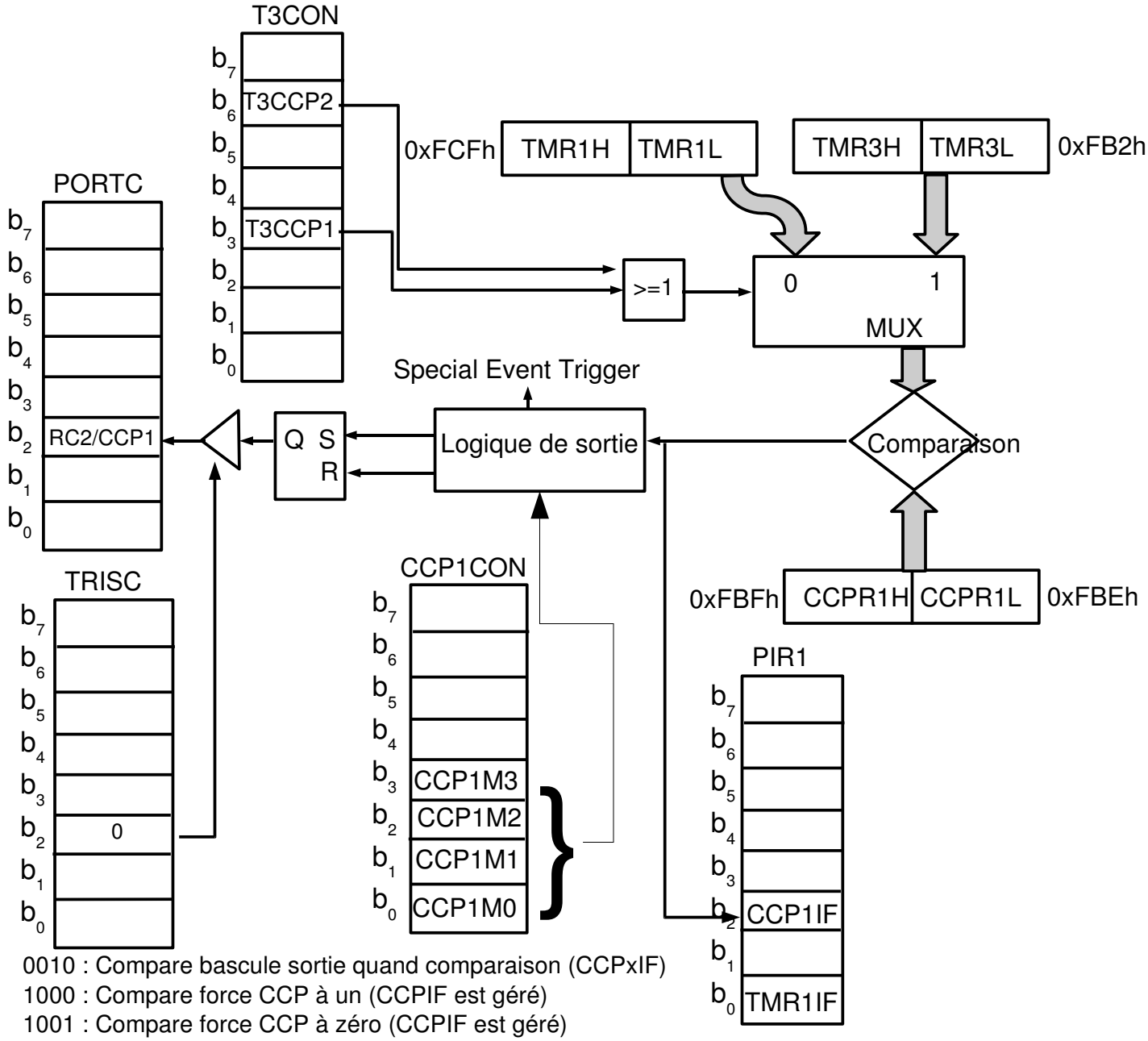
II) Le mode comparaison

Le mode comparaison va nous permettre de générer des signaux de fréquence déterminée.

COMPARAISON PIC 18F4550 et 16F877 (sans timer 3)

Special Event Trigger déclenchera:
- un Reset du Timer1 ou Timer3 (mais pas un positionnement du drapeau d'interruption de Timer1 ou Timer3)
- positionnement du bit GO/DONE qui démarre une conversion A/D (ECCP1 seulement)

Note : T3CCP2 est parfois appelé T3ECCP1



0010 : Compare bascule sortie quand comparaison (CCPxIF)
1000 : Compare force CCP à un (CCPIF est géré)
1001 : Compare force CCP à zéro (CCPIF est géré)
1010 : Compare sort rien sur CCP mais sur CCPIF
1011 : Compare force Special Event Trigger reset sur timer1 (CCP1IF est géré)

Remarque : Cette partie comparaison est identique pour le 18F4550 et le 16F877 sauf que le 18F a la possibilité d'utiliser le timer3 (géré par T3CON) contrairement au 16F.

Exercice 2 Réalisation d'un signal de 1Hz et son amélioration.

1°) Notre horloge quartz a une fréquence de 20 MHz. Utiliser le résultat de l'exercice 1 pour répondre : par combien doit-on diviser la fréquence de CCP1IF minimale pour obtenir un temps de 500000 µs (0,5 s)?

2°) On choisit une division par 5. Le principe sera donc d'attendre CCP1IF cinq fois sans oublier de le remettre à 0 et d'inverser par logiciel la sortie RC2. Il nous faut donc choisir un mode de fonctionnement logiciel (1010 qui n'agit pas sur RC2). Écrire le programme complet.

Remarque : on pourrait utiliser le flag de débordement du Timer1 (TMR1IF du registre **PIR1**) pour faire la même chose, ce qui n'utiliserait pas le module CCP.

3°) Avec la technique de la question précédente, il nous est absolument impossible de régler exactement la fréquence de sortie. On va essayer de palier à cet inconvénient en utilisant un autre mode qui n'agit pas sur RC2 mais a l'avantage de déclencher une remise à 0 du timer1 quand il y a comparaison : c'est le mode 1011. Calculer la valeur à mettre dans **CCPR1**.
Écrire le programme.

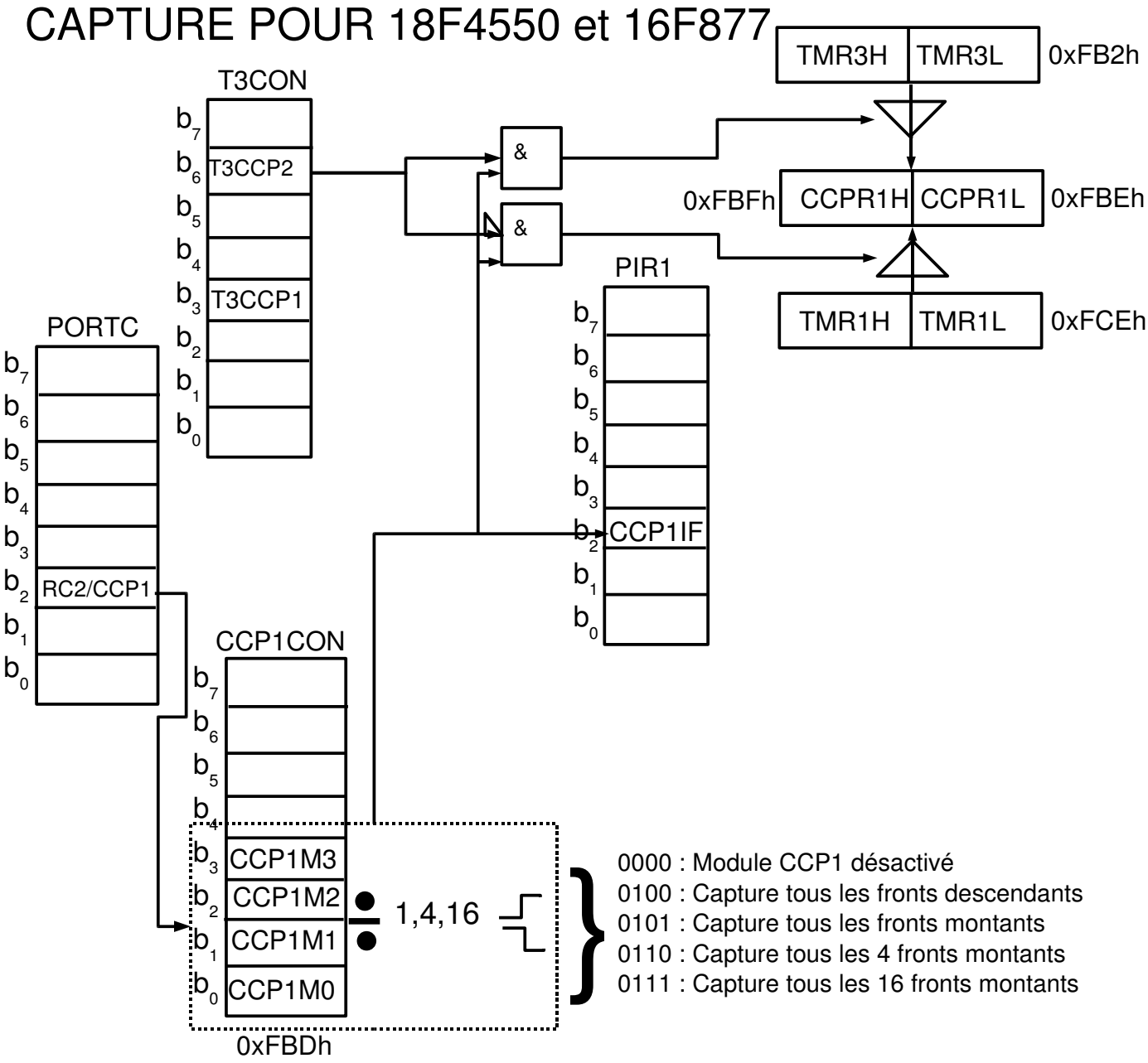
4°) Compléter le programme précédent pour qu'il envoie sur une liaison série (avec un printf) le temps en heure minute seconde.

On verra plus tard qu'une méthode plus précise consiste à utiliser un quartz horloger.

TD 11 : Le mode capture du module CCP (Capture/Compare/PWM)

Dans le mode capture, **CPPR1H** et **CPPR1L** capturent la valeur du timer 1 ou du timer 3 quand un événement se produit sur RC2/CPP1 du port C. Un événement est défini comme :

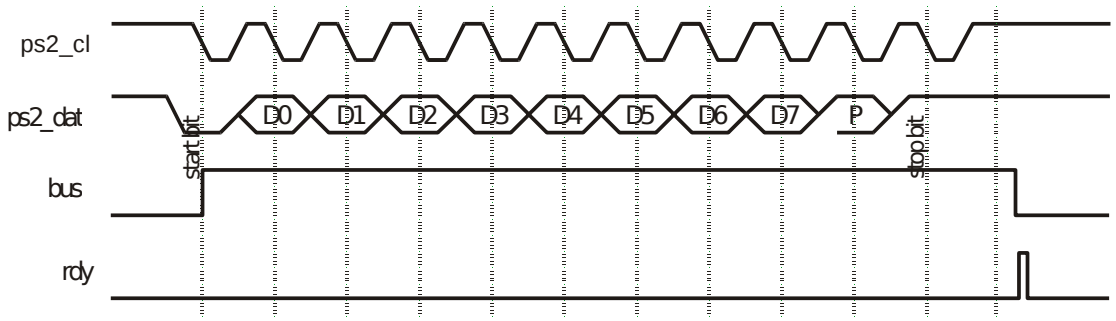
- un front descendant
- un front montant
- tous les 4 fronts montants
- tous les 16 fronts montants



Remarques : Cette partie capture est identique pour le 18F4550 et le 16F877 sauf que le 18F a la possibilité d'utiliser le timer3 (géré par T3CON) contrairement au 16F.
Le PIC® 18F dispose d'un autre module CCP2.
C'est bien le bit T3CCP2 qui sélectionne le timer 3 dans le module CCP1 d'après la doc du 18F4550 !
Attention le PORTC doit être configuré en entrée pour le bit b_2.

Exercice : Utilisation du mode capture pour lire un clavier PS2
Remarque préliminaire : en toute rigueur, il serait préférable de réaliser cette capture avec une interruption. Mais les interruptions ne sont abordées qu'au TD 12.
Présentation du protocole PS2 :

Le protocole PS2 complet est un mélange de protocole synchrone et asynchrone :



où seules les premiers signaux ps2_clk et ps2_data vont nous intéresser. Comme c'est le clavier qui émet les données c'est à lui de fabriquer l'horloge. La fréquence d'horloge est comprise entre 10kHz et 16kHz et on va utiliser la partie capture non pas pour mesurer la fréquence de l'horloge (dans un premier temps en tout cas), mais pour détecter les fronts descendants de l'horloge grâce au bit CCP1IF du registre PIR1.

1°) Écrire le sous-programme "`void initKBD(void)`" destiné à initialiser le fonctionnement du module CCP pour qu'il intercepte les fronts descendants de PS2_clk, si les données ps2_data sont sont sur le bit b_1 du PORTC. Ne pas oublier de mettre les deux bits du PORTC en entrée.

2°) Écrire le programme principal qui lit les informations à chaque fois qu'un front descendant arrive et construit le bit du scan-code après lecture pour le sortir sur le PORTB (positionné en sortie).

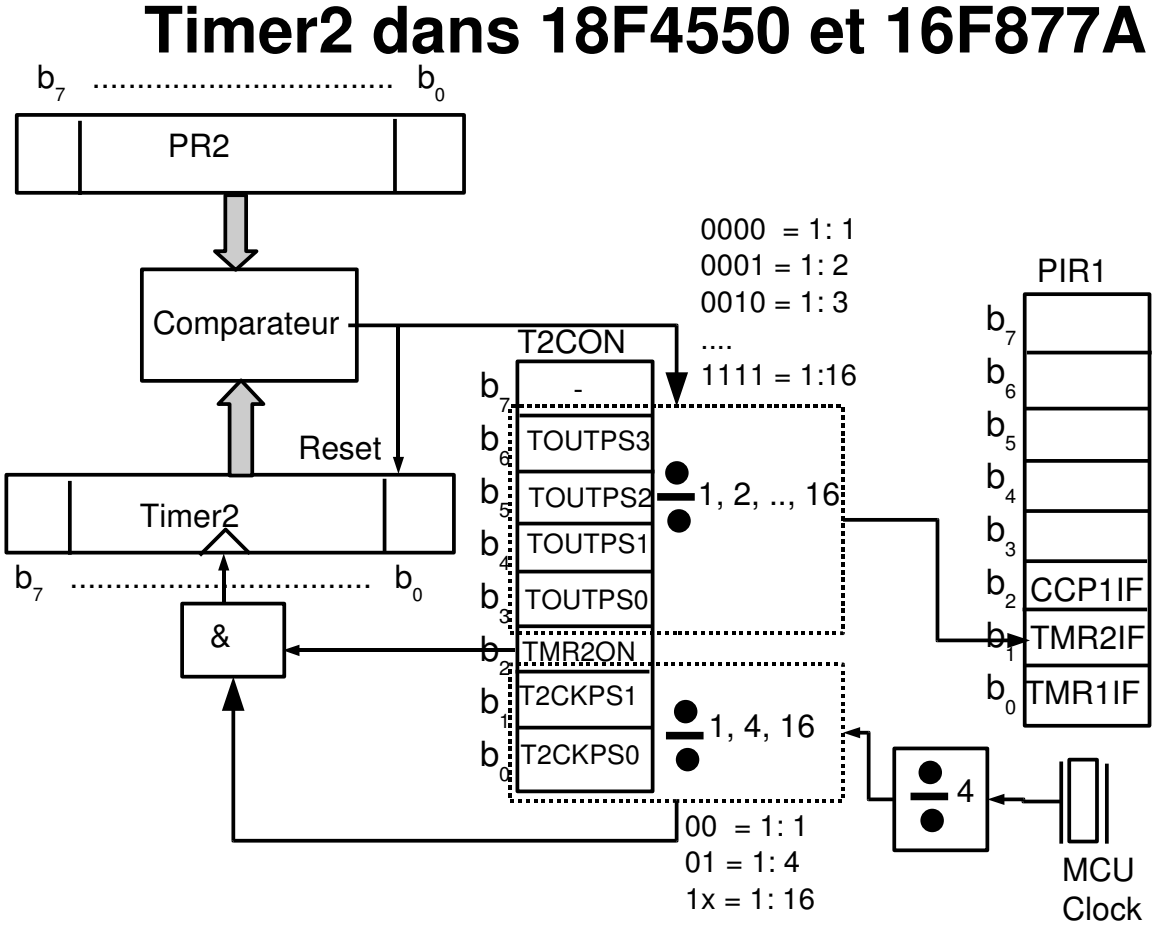
3°) Utiliser maintenant complètement le mode capture pour calculer la fréquence moyenne de l'horloge ps2_clk.

TD 12 : Le mode PWM du module CCP (Capture/Compare/PWM)

PWM signifie « Pulse Width Modulation », ce qu'on pourrait traduire par modulation de largeur d'impulsion. Son objectif est de changer une valeur moyenne en jouant sur le rapport cyclique. Puisque ce module utilise essentiellement le timer 2, nous commençons donc à le décrire.

I) Description du timer 2

Le timer 2 est identique sur le PIC® 16F877 et le 18F4550. Il s'agit d'un timer 8 bits dont la description schématique est présentée maintenant.



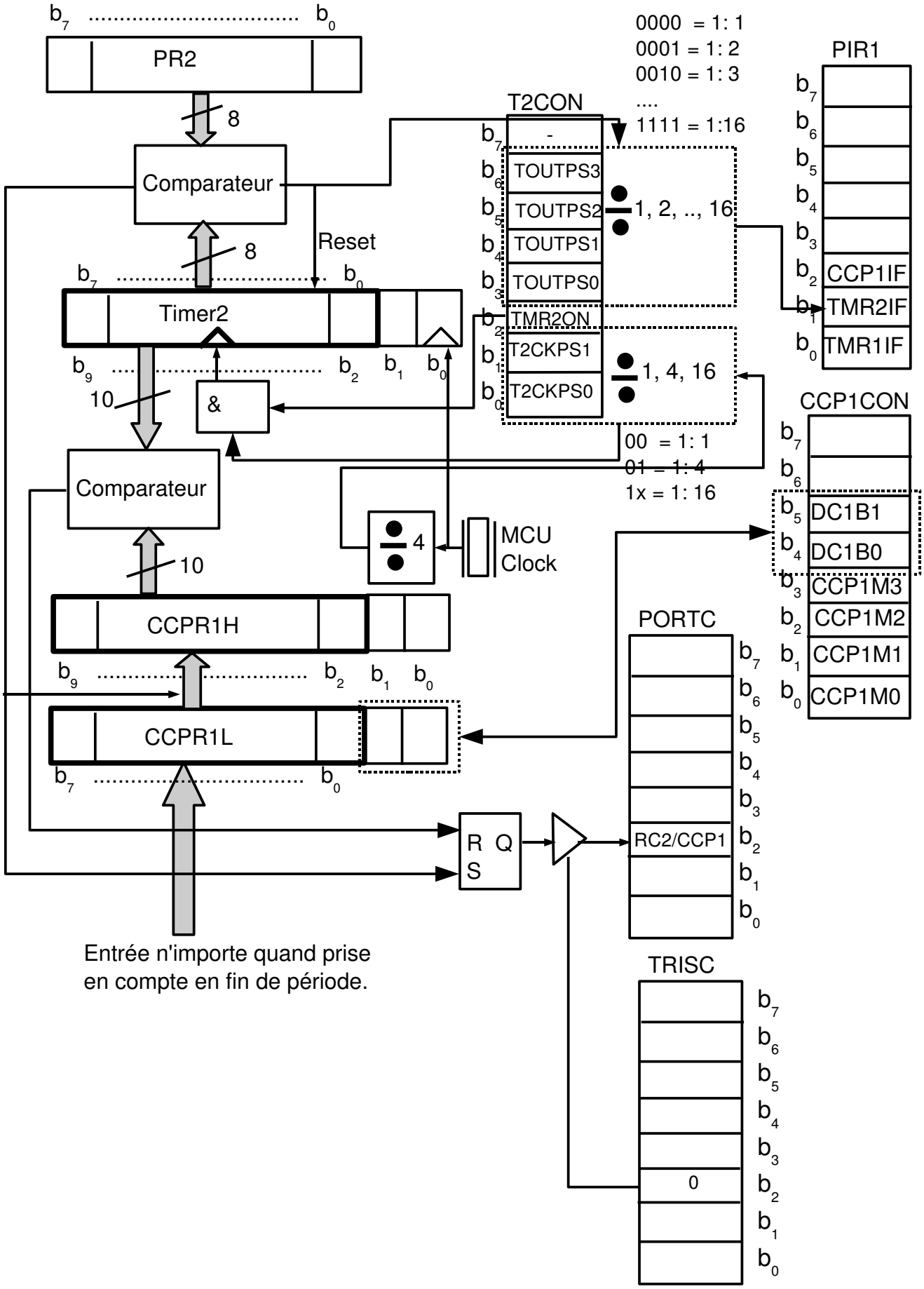
Sa seule entrée est l'entrée horloge du PIC® divisée par 4 suivie d'un pré-diviseur géré par deux bits. Un comparateur donne un signal qui sert de reset au timer2 et d'entrée à un post-diviseur capable de diviser par un nombre entre 1 et 16.
La période T peut être calculée par :
T = (**PR2** + 1) x (Timer2 input clock period) soit :

T = (**PR2**+1) x (T_{osc} x 4 Timer 2 prescale value)

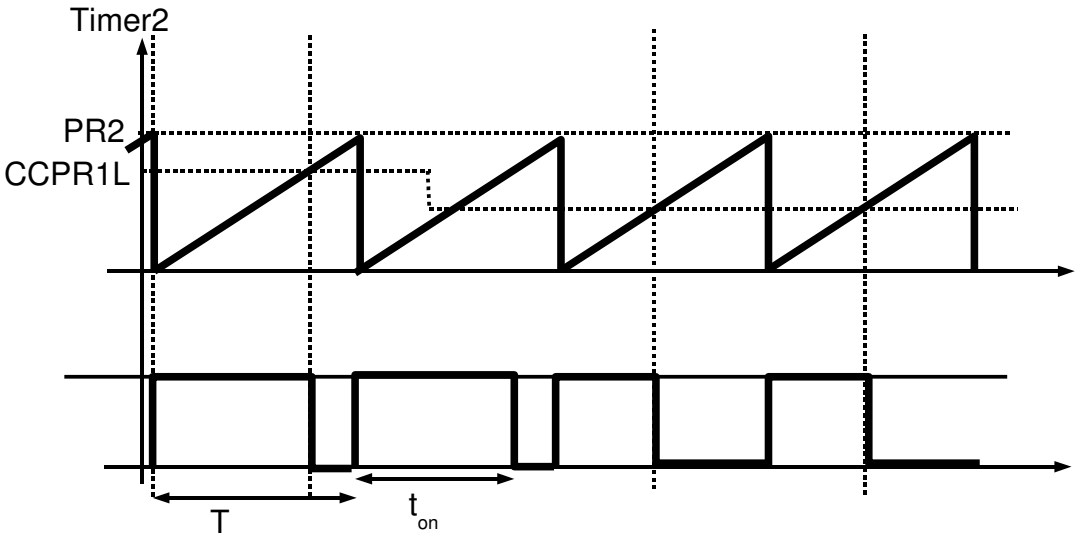
Exercice 1
The PIC® has an oscillator frequency of 4 MHz. What is the slowest Timer 2 interrupt rate it can generate ?
By writing to the **T2CON** register, Timer 2 is switched on, with neither pre- nor postscale. What is the value you have to write to **PR2** to realize a 4,00 kHz frequency ?

Quand on s'intéresse au PWM le choix de la période est réalisé simplement comme dans l'exercice 1. Puisque le tipmer2 est sur 8 bits, la résolution de la période est de 256. Le choix du rapport cyclique est par contre défini par le contenu d'un registre 8 bits **CCPR1L** avec deux bits supplémentaires DC1B1 et DC1B0. Ceci est montré dans la documentation ci-dessous :

PWM dans 18F4550 et 16F877A



Le PWM ne fonctionnera correctement que si $CCPR1L \leqslant PR2$.



La formule permettant de calculer t_{on} est :

t_{on} = (pulse width register) x (T_{osc} x Timer2 prescale value)

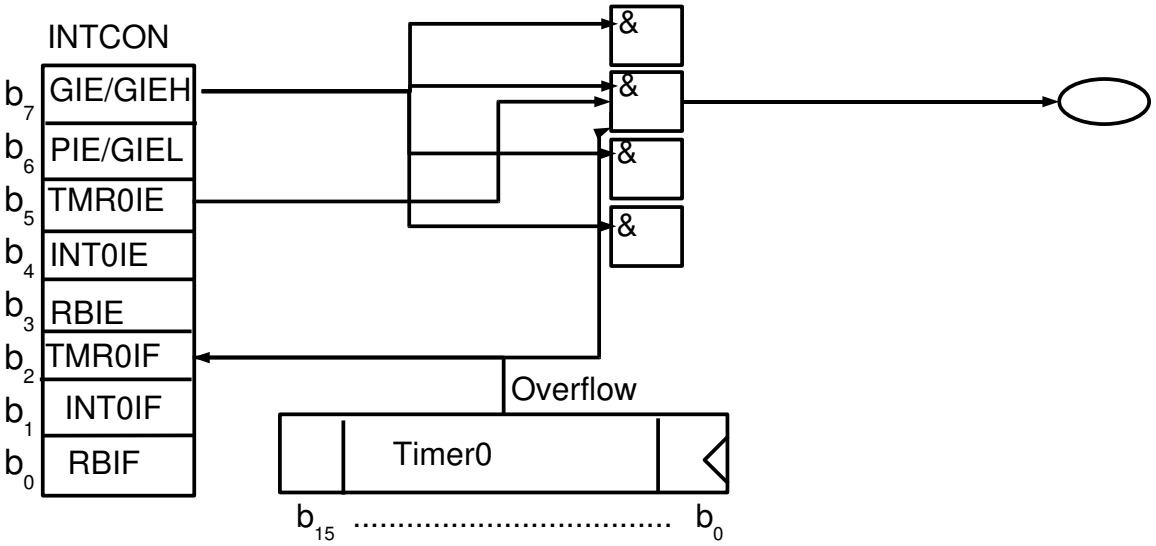
Exercice 2

TD 13 : Interruption timer0

I) Le timer 0

La documentation du Timer0 du PIC® 18F4550 a été présentée dans le TD9.
Nous passons donc à la description de son interruption associée.

Interruption timer0 du 16F877



Ce schéma de principe fonctionne de la manière suivante. Il y aura interruption si on arrive à réaliser un 1 dans l'ellipse, sachant que le matériel ne gère que l'overflow (bit TMR0IF du registre **INTCON**).
Le programme qui gère l'interruption sera en adresse 0x004.

II) Les interruptions en C

La gestion des interruptions en C est fortement dépendante des compilateurs, comme le montre le tableau ci-dessous :

MikroC	C18

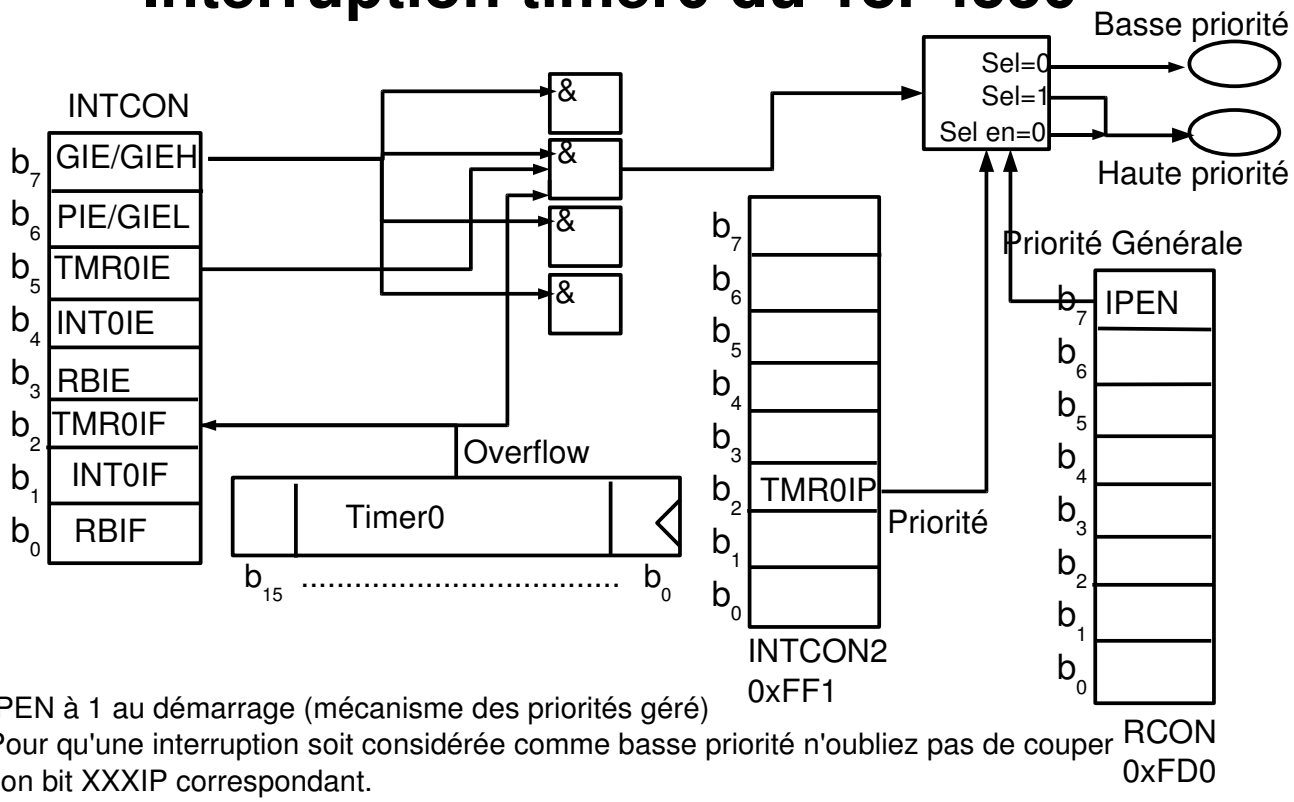
`// interruption haute priorite seulement`
`void interrupt() {`
`  // traitement interruption`
`  // positionnement du FLAG`
`}`
 	
`void main() {`
`  // traitement`
`  while(1);`
`}`
	`#include<p18f452.h>`
	`#pragma config WDT = OFF`
	`#pragma config DEBUG=ON`
	`void it_prioritaire(void) ;`
	`#pragma code address_it=0x08`
	`//0x18 pour basse priorité`
	`void int_toto(void) {`
	`  _asm GOTO it_prioritaire _endasm`
	`}`
	`#pragma code`
	`#pragma interrupt it_prioritaire`
	`void it_prioritaire(void) {`
	`  // traitement interruption`
	`  // positionnement du FLAG`
	`}`
	`main(){`
	`  // traitement`
	`  while(1);`
	`}`

Avec le compilateur mikroC le nom de l'interruption est fixé.
L'écriture d'une interruption se fait normalement en trois étapes :

- spécifier que le sous-programme n'est pas un sous-programme normal, mais un sous-programme d'interruption,
- fixer l'adresse du sous-programme d'interruption
- initialiser le mécanisme d'interruption dans le programme principal.

Pour certains compilateurs (MikroC) les deux premières étapes se font simplement en donnant un nom prédéterminé au sous-programme d'interruption. Cela a une conséquence importante, c'est qu'il n'est pas possible de gérer les priorités : toute interruption est de haute priorité sur le 18F.
Nous décrivons maintenant le mécanisme d'interruption du 18F, un peu plus sophistiqué car il gère deux priorités en adresse 0x008 pour la haute priorité et en adresse 0x018 pour la basse priorité.
Le bit priorité est propre à la série des PIC® 18F et ne retrouve pas dans le composant PIC® 16F

Interruption timer0 du 18F4550



IPEN à 1 au démarrage (mécanisme des priorités géré)
Pour qu'une interruption soit considérée comme basse priorité n'oubliez pas de couper son bit XXXIP correspondant.
IPEN à 0 pas de mécanisme de priorité, toutes les interruptions sont alors considérées comme haute priorité.

Exercice

TD 14 : Interruption Capture du module CCP

Ou conversion Analogique/Numérique

```
/* This code snippet reads analog value from channel 2 and displays
   it on PORTD (lower 8 bits) and PORTB (2 most significant bits). */
// mikroC
unsigned temp_res;
void main() {
  ADCON1 = 0x80;    // Configure analog inputs and Vref
  TRISA = 0xFF;     // PORTA is input
  TRISB = 0x3F;     // Pins RB7, RB6 are outputs
  TRISD = 0;        // PORTD is output
  do {
    temp_res = Adc_Read(2);      // Get results of AD conversion
    PORTD = temp_res;            // Send lower 8 bits to PORTD
    PORTB = temp_res >> 2;       // Send 2 most significant bits to RB7, RB6
  } while(1);
}
```

et dessin p 163 du miKroC_manual.pdf

ANNEXE : les instructions du PIC (18FXX8)

Opérandes :
- f : register file address (adresse mémoire + registres) de 00 à 7F
- W : registre de travail
- d : sélection de destination : d=0 vers W, d=1 vers f
- a : a=0 Access RAM sélectionnée sans tenir compte de **BSR**, a=1 **BSR** est utilisé (banked)
- bbb : adresse de bit dans un registre 8 bits (sur 3 bits)
- k : champ littéral (8, 12 ou 20 bits)
- s : sélection du mode rapide : s=0 pas de restauration des registres **W**, **STATUS** et **BSR**
- PC compteur programme
- TO Time Out bit
- PD Power Down bit

Opérations orientées octets entre registre et mémoire (File en anglais)					
Mnémonique Opérande	Description	Cycles	16 bits Opcode	status affected	notes
ADDWF f[,d[,a]]	Additionne W et f	1	0010 01da ffff ffff	N,OV,C, DC,Z	1,2
ADDWFC f[,d[,a]]	Additionne W et f et retenue		0010 00da ffff ffff	N,OV,C, DC,Z	1,2
ANDWF f[,d[,a]]	ET bit à bit W et f	1	0001 01da ffff ffff	N,Z	1,2
CLRF f[,a]	mise à 0 de f	1	0110 101a ffff ffff	Z	2
COMF f,[d[,a]]	Complément de f	1	0001 11da ffff ffff	N,Z	1,2
CPFSEQ f[,a]	compare f avec W, saute si =	1,(2 ou3)	0110 001a ffff ffff		4
CPFSGT f[,a]	compare f avec W, saute si >	1, (2 ou3)	0110 010a ffff ffff		4
CPFSLT f[,a]	compare f avec W, saute si <	1, (2 ou3)	0110 000a ffff ffff		1,2
DECF f[,d[,a]]	Décrémente f	1	0000 01da ffff ffff	C,DC,N, OV,Z	1,2,3, 4

DECFSZ f[,d[,a]]	Décrémente f (saute si 0)	1,(2)	0010 11da ffff ffff		1,2,3,4
DECFSNZ f[,d[,a]]	Décrémente f (saute si non 0)	1,(2)	0100 11da ffff ffff		1,2,3,4
INCF f[,d[,a]]	Incrémente f	1	0010 10da ffff ffff	C,DC,N,OV,Z	1,2,3,4
INCFSZ f[,d[,a]]	Incrémente f (saute si 0)	1,(2 ou 3)	0011 11da ffff ffff		1,2,3
INFSNZ f[,d[,a]]	Incrémente f (saute si non 0)	1,(2 ou 3)	0100 10da ffff ffff		1,2,3
IORWF f[,d[,a]]	Ou inclusif de f	1	0001 00da ffff ffff	Z,N	1,2
MOVF f[,d[,a]]	déplacement de f (adressage direct)	1	1001 00da ffff ffff	Z,N	1,2
MOVFF fs,fd	déplacement de fs (source) en fd (destination)	2	1100 ffff ffff ffff (s) 1111 ffff ffff ffff (d)		
MOVWF f[,a]	déplacement de W vers f	1	0110 111a ffff ffff		
MULWF f[,a]	multiplication de W par f	1	0000 001a ffff ffff		
NEGF f[,a]	négation de f	1	0110 110a ffff ffff	C,DC,N,OV,Z	1,2
RLCF f[,d[,a]]	Rotation gauche avec la retenue	1	0011 01da ffff ffff	C,Z,N	
RLNCF f[,d[,a]]	Rotation gauche sans la retenue	1	0100 01da ffff ffff	Z,N	1,2
RRCF f[,d[,a]]	Rotation droite avec la retenue	1	0011 00da ffff ffff	C,Z,N	
RRNCF f[,d[,a]]	Rotation droite avec la retenue	1	0100 00da ffff ffff	Z,N	1,2
SETF f[,a]	positionne f à 0xFF (tout le monde à 1)	1	0110 100a ffff ffff		
SUBFWB f[,d[,a]]	soustrait f de W	1	0101 01da ffff ffff	C,DC,N,OV,Z	1,2
SUBWF f[,d[,a]]	soustrait W de f	1	0101 11da ffff ffff	C,DC,N,OV,Z	1,2
SUBWFB f[,d[,a]]	soustrait W de f avec retenue	1	0101 10da ffff ffff	C,DC,N,OV,Z	
SWAPW f[,d[,a]]	inverser les quartets dans f	1	0011 10da ffff ffff		4
TSTFSZ f[,a]	teste f saute si 0	1	0110 011a ffff ffff		1,2
XORWF f[,d[,a]]	Ou exclusif de f	1	0001 010da ffff ffff	Z	

Opérations orientées bits sur les registres					
Mnémonique Opérande	Description	Cycles	16 bits Opcode	status affected	notes
BCF f,b[,a]	mise à 0 du bit b dans f	1	1001 bbba ffff ffff		1,2
BSF f,b[,a]	mise à 1 du bit b dans f	1	1000 bbba ffff ffff		1,2
BTFSC f,b[,a]	test du bit b 0 de f saute si 0	1,(2)	1011 bbba ffff ffff		3,4
BTFSS f,b[,a]	test du bit b 0 de f saute si 1	1,(2)	1010 bbba ffff ffff		3,4
BTG f,b[,a]	inverser le bit du registre	1	0111 bbba ffff ffff		1,2

Opérations de contrôles

Mnémonique Opérande	Description	Cycles	16 bits Opcode	status affected	notes
BC n	saut relatif court si retenue est positionnée	1,(2)	1110 0010 nnnn nnnn		
BN n	brancher si résultat négatif	1,(2)	1110 0110 nnnn nnnn		
BNC n	brancher si retenue non positionnée	1,(2)	1110 0011 nnnn nnnn		
BNN n	brancher si résultat non négatif	1,(2)	1110 0111 nnnn nnnn		
BNOV n	brancher si pas dépassement (overflow)	1,(2)	1110 0101 nnnn nnnn		
BOV n	brancher si dépassement (overflow)	1,(2)	1110 0100 nnnn nnnn		
BNZ n	brancher si résultat non nul	1,(2)	1110 0001 nnnn nnnn		
BZ n	brancher si résultat nul	1,(2)	1110 0000 nnnn nnnn		
BRA n	brancher toujours	1,(2)	1101 0nnn nnnn nnnn		
CALL n [,s]	appel du sous programme n (sur 20 bits) (s fast bit)	2	1110 110s kkkk kkkk 1111 kkkk kkkk kkkk		
CLRWDT -	mise à 0 du timer watchdog	1	0000 0000 0000 0100	/TO,/PD	
DAW	ajustement décimal dans W	1	0000 0000 0000 0111		
GOTO n	aller à l'adresse n (sur 20 bits)	2	1110 1111 kkkk kkkk 1111 kkkk kkkk kkkk		
NOP -	pas d'opération	1	0000 0000 0000 0000		
NOP -	pas d'opération	1	1111 xxxx xxxx xxxx		4
POP	dépile le sommet de la pile dans PC	1	0000 0000 0000 0110		
PUSH	empile PC sur le sommet de la pile	1	0000 0000 0000 0101		
RCALL n	appel en relatif	2	1101 1nnn nnnn nnnn		
RESET	reset en programmation	1	0000 0000 1111 1111		
RETFIE s	Retour d'interruption	2	0000 0000 0001 000s		
RETLW k	retour avec le littéral dans W	2	000 1100 kkkk kkkk		
RETURN s	retour de sous-programme	2	0000 0000 0001 001s		
SLEEP	aller au mode standby	1	0000 0000 0000 0011	/TO,/PD	

Opérations littérales (adressage immédiat)					
Mnémonique Opérande	Description	Cycles	16 bits Opcode	status affected	notes
ADDLW k	Addition de W et k	1	0000 1111 kkkk kkkk	N,OV,C, DC,Z	
ANDLW k	Et logique de W et k	1	0000 1011 kkkk kkkk	N,Z	
IORLW k	OU inclusif logique de W et k		0000 1001 kkkk kkkk	N,Z	
LFSR f,k	chargement de l'adresse 12 bits dans FSR (ff numéro de registre FSR concerné)	2	1110 1110 00ff kkkk 1111 kkkk kkkk kkkk		
MOVLB k	copier la valeur dans BSR	1	0000 0001 0000 kkkk		
MOVLW k	chargement du littéral dans W	1	0000 1110 kkkk kkkk		
MULLW k	multiplication du littéral par W	1	0000 1101 kkkk kkkk		
RETLW k	retour avec littéral dans W	2	0000 1100 kkkk kkkk		
SUBLW k	soustraction de W du littéral	1	0000 1000 kkkk kkkk	N,OV,C, DC,Z	
XORLW k	ou exclusif du littéral avec W	1	0000 1010 kkkk kkkk	N,Z	

Opérations de programmation de la mémoire					
Mnémonique Opérande	Description	Cycles	16 bits Opcode	status affected	notes
TBLRD*	lecture octet Flash pointé par TBLPTR et le copie dans TABLAT	2	0000 0000 0000 1000		
TBLRD*+	... avec post increment	2	0000 0000 0000 1001		
TBLRD*-	... avec post decrement	2	0000 0000 0000 1010		
TBLRD+*	... avec pre increment	2	0000 0000 0000 1011		
TBLWT*	écriture octet Flash pointé par TBLPTR avec ce qu'il y a dans TABLAT	2	0000 0000 0000 1100		
TBLWT*+	... avec post increment	2	0000 0000 0000 1101		
TBLWT*-	... avec post decrement	2	0000 0000 0000 1110		
TBLWT+*	... avec pre increment	2	0000 0000 0000 1111		

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

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

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

Description de la mémoire RAM

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset
FFFh	TOSU	-	-	-	TOS<20:16>					---0 0000
FFEh	TOSH	TOS<15:8>								0000 0000
FFDh	TOSL	Haut de pile (Top of stack = TOS) : poids faible TOS<7:0>								0000 0000
FFBh	PCLATU	-	-	bit21	PC<20:16>					---0 0000
FFAh	PCLATH	PC<15:8>								0000 0000
FF9h	PCL	Compteur programme (8 bits de poids faible) PC<7:0>								0000 0000
FF2h	INTCON	GIE/ GIEH	PEIE/ GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x
FF1h	INTCON2	/RBPU	INTEDG0	INTEDG1	-	-	TMR0IP	-	RBIP	111- -1-1
FF0h	INTCON3	INT2IP	INT1IP	-	INT2IE	INT1IE	-	INT2IF	INT1IF	11-0 0-00
FE8h	WREG									xxxx xxxx
FD8h	STATUS	-	-	-	N	OV	Z	DC	C	---x xxxx
FD7h	TMR0H	TIMER0 <15:8>								0000 0000
FD6h	TMR0L	TIMER0 <7:0>								xxxx xxxx
FD5h	T0CON	TMR0ON	T08Bit	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111

FCFh	TMR1H	TIMER1 <15:8>								
FCEh	TMR1L	TIMER1 <7:0>								
F9Eh	PIR1	SPPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	
	TRISC									
F93h	TRISB									1111 1111
F92h	TRISA	-								-111 1111
F8Ah	LATB									xxxx xxxx
F89h	LATA	-								-xxx xxxx
	PORTC						RC2/CCP1			
F81h	PORTB	RB7/PGD	RB6/PGC	RB5/PGM	RB4	RB3/ CANRX	RB2/CAN TX/INT2	RB1/INT1	RB0/INT0	xxxx xxxx
F80h	PORTA	-	RA6/CLK O/OSC2	RA5/ AN4	RA4/ TOCKI	RA3/AN3 /Vref+	RA2/AN 2/Vref-	RA1/ AN1	RA0/AN0/ CVref	-x0x 0000
07h	----	Non implémenté								
08h	EEDATA	Registre de données EEPROM								xxxx xxxx
09h	EEADR	Registre d'adresses EEPROM								xxxx xxxx