Programmation en assembleur PIC : directives, macro-instructions, boucles, switch-case

Objectifs

Le but de ce second TME est de vous faire appréhender le problème lié à la gestion de la mémoire dans un microcontrôleur en général et dans un pic en particulier. Vous aborderez une méthode pour l'écriture de fonctions concernant le problème lié aux passages des paramètres. Vous verrez également une méthode pour introduire le temps réel dans vos programmes par l'intermédiaire de boucles de temporisation.

Vous trouverez le programme dont le code est donné en fichier attaché hello4

mkpic: un script pour la compilation de projets pic

Vous utiliserez désormais un script nommé **mkpic** pour toutes les opérations d'assemblage, de simulation ou de programmation. Nous verrons les avantages de ce script au fur et à mesure des TME, pour l'heure contentez-vous de lancer:

```
mkpic a        -> pour faire l'assemblage
mkpic s        -> pour faire l'appel du simulateur
mkpic u        -> pour faire le chargement (upload) du programme dans le pic
```

mkpic impose quelques contraintes méthodologiques. La plus importante est qu'il suppose qu'il n'y a **qu'un seul programme par répertoire** (au sens d'une seule application), et que le **nom du programme doit être le nom du répertoire** avec l'extension .asm en plus. Un programme ne signifie pas obligatoirement un seul fichier, puisque le fichier racine, du nom du programme peut inclure d'autres fichiers de noms quelconques .asm ou .inc. Grâce à cette contrainte, il est inutile de donner le nom du programme à mkpic, il le trouve tout seul.

L'appel à mkpic sans paramètre donne:

```
Script d'assemblage - simulation - programmation pour PIC ver 2.10
------------------------------------------------------------------

Usage   : mkpic <command> [F=file] [P=processor] [M=on|off|debug] [W=0|1|2]
                          [B="labels..."] [A="nbcycles..."] [C=cmdfile]
                          [I="include_dir..."] [D="SYM=VAL SYM=VAL..."]
                          [E=0|1] [X=0|1]

... la suite en annexe.
```

La mémoire

Les différents types de mémoire du pic16f877

Nous avons déjà vu que le pic 16f877 dispose de quatre types de mémoire: pour le code programme, pour les données volatiles, pour les adresses de retours (la pile), pour des données persistantes.

1. La mémoire de programme (8kmots de 14bits): La mémoire de programme est composée de 4 pages de 2k-instructions de 14 bits.

2. La mémoire non volatile de données (256mots de 8bits): La mémoire non volatile de données (RAM) est constituée d'une eeprom flash de 256 octets. Son principal avantage est que ses données sont conservées en l'absence de courant, de même pour la mémoire de programme. En revanche l'accès à cette mémoire se fait par une procédure nécessitant plusieurs cycles en lecture et plusieurs dizaines de cycles en écriture. Cette caractéristique réduit l'usage de cette mémoire à contenir des paramètres généraux et pas des variables. Sachez aussi que, dans le pic que vous utilisez, la mémoire de programme peut également servir de mémoire de données. En effet, elle est de même nature, c.-à-d. de la flash. En outre, sa lecture est plus rapide et elle est plus grande. C'est pourquoi elle est souvent utilisée pour y placer des données. Cependant, le nombre d'écritures est limité de 100000 à un million de fois.

3. La pile des adresses de retour (8mots de 13bits): Le pic dispose d'une instruction call qui commence par sauvegarder dans une mémoire spécifique, la pile, l'adresse de l'instruction suivante et qui saute ensuite à l'adresse passée en paramètre. La pile est également utilisée lors d'une interruption, lorsque le programme en cours est stoppé temporairement, pour exécuter le gestionnaire d'interruption. Cette pile, ne fait pas partie des registres, elle ne peut être ni lue ou ni écrite normalement. Elle fait 8 cases. Le seul moyen de la lire est de dépiler l'adresse enregistrée au sommet par une instruction, ret, retlw, ou retfie qui toutes trois restaurent le compteur ordinal (PC) avec l'adresse lue. Disons tout de suite, que ce mode de fonctionnement limité de la pile empêche son utilisation pour le passage des paramètres aux fonctions.

4. La mémoire volatile de données (512mots de 8bits, dont 368 pour l'utilisateur): La mémoire volatile des données est composée de 4 bancs de 128 octets. La mémoire des données est en fait un ensemble de registres organisé en 4 bancs correspondant à 128 adresses soit 512 adresses de registres pour le pic16f877, de `0` à `0x7F`, de `0x80` à `0xFF`, de `0x100` à `0x17F`, et de `0x180` à `0x1FF`. Les 512 adresses dipsonibles ne correspondent pas à 512 registres, et ce pour deux raisons. La première est qu'à certaines adresses, il n'y a pas de registres. La seconde est que certains registres répondent à plusieurs adresses, par exemple le registre STATUS est accessible via 4 adresses distinctes: `0x03`, `0x83`, `0x103`, `0x183`, en fait il a une adresse par banc, pour toutefois un seul registre.

On distingue 2 types de registres

Les registres «spéciaux»
: comme `TRISD` ou `PORTD`, dont la fonction est prédéfinie. Ils permettent d'accéder aux ressources internes du microcontrôleur. Nous les verrons au fur et à mesure des TME. Je vous conseille toutefois de faire une lecture «diagonale» (i.e. rapide) de l'ensemble des registres dès maintement afin de vous faire une idée de l'organisation du pic16f877.

Les registres «normaux»
: l'utilisateur peut les utiliser à sa guise. Le pic16f877 en compte 368. Sur les 368, 16 ont 4 adresses, une par banc, ce sont les registres de 0x70 à 0x7F. Bien que rien ne l'impose, nous réserverons ces registres partagés à un usage spécifique pour profiter de ces adresses multiples.

Remarque

Notez bien que l'architecture du pic peut surprendre pour ceux qui sont habitués à une organisation de la mémoire plus classique, avec d'un coté les registres internes (quelques dizaines au plus) et de l'autre la mémoire externe où se trouve programmes et données. En effet, les pics sont des microcontrôleurs qui cherchent à se suffire à eux-mêmes (c.-à-d. sans besoin de composants externes si ce n'est pour des raisons d'adaptation électrique). C'est pourquoi toute la mémoire est interne, et le choix à été fait d'augmenter le nombre de registres afin de se passer de mémoire externe. En fait, il est tout de même possible d'avoir de la mémoire externe mais avec un accès plus lent.

Les deux modes d'accès aux registres

L'accès direct
: pour lequel le numéro du registre se trouve directement dans l'instruction. Malheureusement, et c'est là une petite difficulté que vous avez déjà vue, l'instruction ne contient que les 7 bits de poids faibles de l'adresse absolue (0 à 511). Les 2 bits de poids forts sont dans le registre STATUS, aux places RP1(bit 6) et RP0(bit 5). Ces bits sont concaténés aux bits contenus dans l'instruction, pour former l'adresse absolue sur 9 bits.

L'accès indirect registres
: pour lequel le numéro du registre à accéder se trouve dans un autre registre. Il n'y a qu'un seul registre qui peut être utilisé pour faire un accès indirect, c'est le registre FSR (dont l'adresse est 0x04, mais aussi 0x84, 0x104 et 0x184). Plus précisément, pour faire un accès indirect, on place le numéro du registre à accéder dans le registre FSR, et on peut alors accéder en lecture ou én écriture à ce registre en utilisant le registre symboliquement notée INDF (0, mais aussi 0x80, 0x100, 0x180). Autrement dit quand on accède à l'adresse INDF, on accède en fait à l'adresse contenue dans le registre FSR. Toutefois, encore une difficulté, le registre FSR a un format de 8 bits et il en faut 9 pour désigner un registre. Là encore, on va chercher le bit manquant dans le registre STATUS, à la place IRP (bit 7).

La réservation des registres «normaux» (utilisateur)

Les 368 registres utilisateur sont libres, au sens où le matériel n'impose pas de contrainte sur leur rôle. Toutefois nous allons devoir les allouer aux programmes et nous allons nous imposer des contraintes de programmation (autant le dire tout de suite, ces contraintes sont discutables, c.-à-d. que d'autres programmeurs en auraient imposés des différentes). D'ailleurs nous nous autoriserons quelquefois à ne pas les respecter. Enfin, quelles qu'elles soient, les contraintes de programmation ont pour but de rendre les programmes plus cohérents, plus lisibles et donc plus fiables. Elles permettent également de faire du partage avec les autres programmeurs, ou du «design reuse» très à la mode dans certains millieux.

Nous introduirons nos contraintes au fur et à mesure des séances, disons pour l'heure que les registres partagés seront utilisés entre autre pour le passage des paramètres aux fonctions, et pour la sauvegarde du contexte lors des interruptions.

L'allocation des registres utilisateur pose un problème d'organisation. Il y a deux choix possibles:

1. On fait une allocation statique et globale de tous les registres du programme, en prenant garde à ce qu'il n'y ait pas de collisions entre adresses allouées.
2. On fait une allocation locale, par fonction ou par tâche, en laissant un programme ou une méthode faire en sorte qu'il n'y ait pas de collisions. Dans ce cas, il y a encore deux possibilités:
    1. On dispose d'un assembleur relogeable, qui ne choisit pas les adresses des registres alloués, qui produit du code objet (comme le fait un compilateur C) et qui utilise un éditeur de liens pour produire le code exécutable en résolvant le problème d'allocation mémoire.
    2. On dispose d'une méthode d'allocation qui assure la non collision par construction.

Ces deux approches sont possibles puisque, gpasm peut produire du code relogeable, et nous vous proposerons une méthode de non collision. Pour ce TME, nous allons utiliser l'allocation statique et globale, en utilisant la directive

assembleur `CBLOCK`

```
CBLOCK adr
  nom1 : long1     ; réserve nom1 à l'adresse adr
  nom2 : long2     ; réserve nom2 à l'adresse adr + long1
ENDC
CBLOCK
  nom3 : long3     ; réserve nom3 à l'adresse adr + long1 + long2
  nom4 : long4     ; réserve nom4 à l'adresse adr + long1 + long2 + long3
ENDC
```

Le comptage du temps

Nous allons reprendre le programme `hello3` vu lors de la précédente séance, mais nous voulons maintenant ralentir le déplacement du bit à 1 dans le PORTD afin de le rendre visible à l'oeil, en supposant que l'on relie des leds sur les broches associées au PORTD. Pour ce faire, nous allons réaliser une boucle d'attente active. Cette attente est dite active car le processeur est effectivement occupé à compter et ne peut rien faire d'autre. Nous verrons qu'il existe un moyen plus efficace de réaliser les attentes en utilisant des périphériques spécialisés nommés *timers* qui évitent de monopoliser le processeur.

La boucle d'attente se fait dans la routine wait200us. Comme vous le voyez sur le listing la boucle consiste à remplir un registre avec une valeur calculée de manière précise, puis à le décrémenter jusqu'à ce qu'il atteigne 0. La question est combien faut-il faire de tours de boucle pour que la fontion dure 200µs.

Le pic de notre maquette est cadencé par une horloge à 20MHz, soit une période élémentaire de 50ns, mais le pic utilise 4 cycles élémentaires pour exécuter la plupart des instructions. En fait on parle de macro-cycles pour désigner ces 4 cycles élémentaires et par abus de langage quand on parle de cycles, on fait référence à ces macro-cycles. En conséquence une instruction standard, comme `movlw`, est exécutée en un macro-cycle, soit en 200ns. Dans la documentation technique du pic, à la page 56[feuille 6 recto], vous avez la liste des instructions avec leur durée respective: 1 ou 2 cycles (c.-à-d. macro-cycles). Remarquez que la description des instructions dans cette documentation est celle des pic de série 16f8x et pas 16f8xx. En fait il s'agit des mêmes instructions mais la documentation des 16f8x est plus détaillée lorsqu'elle décrit le comportement des instructions.

Le programme hello4.asm doit être copié dans le répertoire `hello4` que vous allez créer. L'assemblage se fera avec mkpic.

Question 1

Déterminer la valeur à mettre dans XXX pour faire 200µs.

Notez, dans le code qui suit, l'utilisation du caractère `$`. Dans un programme en assembleur `$` référence l'adresse de l'instruction courante. En conséquence, `$+1` est l'adresse qui suit l'instruction courante, `$-1` est l'adresse qui précède. `goto $+1` saute à l'adresse suivante, et donc semble ne servir à rien, mais elle dure 2 cycles (i.e. 2 macro-cycles) et c'est là son intérêt.

```
; ------------------------------------------------------------------------
; programme : hello4
; Date      : 20050108:2328
; Version   : 1
; Auteurs   : franck
; Notes     : suppose que l'horloge externe est à 20MHz
; toujours le chenillard mais avec une routine de délai qui ralentit le
; décalage du portd, l'attente est de 200 microsecondes.
; ------------------------------------------------------------------------

        list    p=16f877        ; definit le processeur cible
        include "p16f877.inc"   ; declaration des noms de registres
```

```
        ; Definition du registre de configuration du PIC
        ; _CP_OFF   : le code n'est pas protege et peut etre relu
        ; _WDT_OFF  : pas de timer watch dog
        ; _PWRTE_ON : attente d'un délai apres le power on
        ; _HS_OSC   : oscillateur à quartz
        ; _LVP_OFF  : pas de mode programmation basse tension
        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF

        CBLOCK  0x20            ; première adresse de la zone
           wait200us_arg : 1    ; 1 octet de nom wait200us_arg
        ENDC

        org     0               ; adresse du reset
        call    initialisation
        goto    main

        org     4               ; adresse du vecteur d'interruption
        retfie                  ; par défaut ne rien faire

initialisation
        BANKSEL TRISD           ; aller dans le banc de registre TRSID
        clrf    TRISD           ; place tous les signaux du port D en sortie
        BANKSEL PORTD           ; revenir dans le banc de registre PORTD
        return

main    movlw   1               ; met 0x01  sur port D
        movwf   PORTD
loop    rrf     PORTD,f
        call    wait200us
        goto    loop            ; on boucle sur la rotation

wait200us ; routine d'attente active
        movlw   d'XXX'          ; XXX -> W              ? cycle
        movwf   wait200us_arg   ; W -> wait200us_arg    ? cycle
        goto    $+1             ; nop2                  ? cycles
        decfsz  wait200us_arg,f ; wait200us_arg--       ? cycle si faux sinon ?
        goto    $-2             ; goto adr courante -2  ? cycles
        return                  ;                       ? cycles
; --------------------------------------------------------------------------
        END                     ; directive terminant un programme
```

Question 2

Maintenant, nous allons rendre la routine wait200us paramètrable. Le paramètre sera placé dans le registre W. Après avoir copié le projet `hello4` en `hello5`, modifiez le programme `hello5` en conséquence, renommer la routine `wait200us` en `waitus` (et toutes les étiquettes formées sur ce nom).

Question 3

On veut que W contienne précisément le nombre de micro-secondes (µs) qu'attendra la routine `waitus`. Les deux routines ci-dessous font une attente de Wµs, où W contient un nombre entre 2 et 255. C'est à dire que pour attendre 200us, on devra mettre 200 dans W et appeler `waitus`. Solution 1:

```
waitus ; routine d'attente active
        movwf   waitus_arg      ; W -> waitus_arg
        decf    waitus_arg,f    ; waitus_arg--
        goto    $+1             ; nop2
        decfsz  waitus_arg,f    ; waitus_arg--
        goto    $-2             ; goto adr courante -2
        return                  ;
```

Solution 2:

```
waitus ; routine d'attente active
        addlw   -1              ; -1 + W -> W
```

```
movwf   waitus_arg      ; W -> waitus_arg
goto    $+1             ; nop2
decfsz  waitus_arg,f    ; waitus_arg--
goto    $-2             ; goto adr courante -2
return                  ;
```

Les deux solutions semblent équivalentes et pourtant elles ne le sont pas: la première fonctionne (c.-à-d. un bit à 1 circule sur le registre port D), la seconde pas (c.-à-d. que le bit à 1 ne circule plus). Essayez de comprendre pourquoi et d'en tirer une conclusion.

D'autre part, pour la solution qui marche quelle durée obtient-on si W est à 1 ou 0 au départ?

Question 4

Nous allons ajouter une deuxième routine nommée `waitms` qui invoque la première (`waitus`).

◊ La routine `waitms` décompte un nombre de 16 bits pour compter les tours de boucle et son corps de boucle appelle la routine `waitus`.

◊ La routine `waitms` a un pas de 0.1 ms, c'est à dire 100us. **On ne vous demande pas d'être précis au cycle près pour le calcul de temps.**

◊ Elle prend son paramètre dans une variable, nommée `waitms_arg` de 2 octets. Le placement des nombres entiers de plusieurs octets suit la stratégie «little endian» comme le MIPS (poids faible à l'adresse basse).

◊ À titre d'exemple pour attendre 0.13 seconde (130 ms), une fois écrite la routine `waitms` il faut écrire ({\bf notez à cette ocasion une méthode pour initialiser une variable de 2 octets avec une valeur}):

```
movlw  LOW D'1300'  ; met les 8 bits de poids faible de 1300 dans W
movwf  waitms_arg   ; met W dans le registre waitms_arg
movlw  HIGH D'1300' ; met les 8 bits de poids fort de 1300 dans W
movwf  waitms_arg+1 ; met W dans le registre waitms_arg+1
call   waitms       ; attend 1300 * 0.1 ms = 130ms
```

◊ La routine `waitms` doit décrémenter un nombre sur 16 bits et boucler tant qu'il n'est pas nul. L'opération de décrémentation sur 16 bits n'existe pas dans les instructions natives du pic, c'est pourquoi vous ajouterez une macro-instruction au debut de votre programme (après la config) qui définit une pseudo instruction 16bits. Pour ne pas vous prendre du temps, nous vous donnons cette macro (bonne candidate lors d'un examen):

```
subwf2  macro @r ; r <- r-w sur 2 octets ; positionne /Z ; 5 cycles
           subwf @r,f
           btfss STATUS,C ; C=0 si r passe en dessous de 0
               decf  @r+1,f    ; si C==0 alors décrémenter le poids fort
               movf  @r,w      ; positionne Z si et seulement si
               iorwf @r+1,w    ; r ET r+1 sont à 0
        endm
```

◊ En outre, vous allez désormais faire une rotation sur 8 bits et non plus sur 9 bits dans le programme principal. Pour faire la rotation, vous utiliserez la macro `rr8f r` dont la définition vous est donnée ci-dessous (autre bon candidat d'exercice à l'examen).

```
rr8f    macro @r ; r <- (r&1)|((r>>1)&0x7F) : rot. droite sur 8 bits
           rrf   @r,w       ; initialise carry avec le bit n°0 de r
           rrf   @r,f       ; r[7..0] <- r[6..0],carry
        endm
```

◊ Pour résumer, vous devez:

· Copier le projet `hello5` en `hello6`.

· Ajouter les macro `subwf2` et `rr8f`.

· Réserver la variable `waitms_arg` sur 2 octets, dans la section `CBLOCK`.

· Ajouter la routine `waitms` (disons avant `waitus`).

· Modifier «main» pour qu'il utilise la macro `rr8f` et appelle la routine `waitms` et de telle

sorte que le déplacement du bit sur le port D se fasse à la vitesse d'environ une seconde par bit.

L'écriture des fonctions

Quel est le problème

Maintenant, on aimerait pouvoir mettre dans un fichier toutes ces routines, afin de se constituer une bibliothèque de fonctions réutilisables. On va d'ailleurs en profiter pour changer de nom. On appellera fonction une routine placée dans une bibliothèque pour laquelle on a défini une stratégie standardisée de passage des paramètres et de remise des résultats. On appellera une routine un morceau de programme rendant un service particulier, référencé par son nom (correspondant à l'adresse de sa première instruction) auquel on «saute» au moyen d'une instruction (ici l'instruction call) qui sauvegarde le compteur ordinal+1 (adresse de l'instruction suivante) dans une pile spéciale de 8 mots de 13 bits, avant de lui affecter l'adresse de la routine et dont on revient au moyen d'une instruction (ici l'instruction return) qui restaure le compteur ordinal. Une routine peut prendre des paramètres mais aucune convention n'est définie, on fait comme on veut en somme.

En effet, le principal problème pour l'écriture des fonctions concerne le choix d'une méthode pour le passage des paramètres, l'allocation des registres de travail et la remise du, ou des, résultats. Quand on écrit une fonction quelconque, on ne veut pas se soucier de qui va l'utiliser, or si on réserve en dur (absolu) les registres (par exemple les registres 0x20 et 0x21) pour une fonction, ils ne sont plus, dans le cas général, utilisables par une autre fonction.

Dans un processeur doté d'une vraie pile, les paramètres des fonctions sont passés par la pile et la fonction y accède généralement en utilisant le pointeur de pile. Dans ce pic, la pile ne peut pas servir à passer les paramètres car il n'existe aucun moyen d'y accéder autrement qu'avec les instructions call et return. On peut bien sûr simuler une pile et utiliser des accès mémoire indirects grâce au registre FSR, mais c'est assez coûteux en instructions.

La façon de résoudre ce problème d'une manière générale passe par le choix judicieux des cases mémoire utilisées par les fonctions au moment où l'on connait toutes les adresses nécessaires. C'est le travail d'un éditeur de liens. Si on n'en dispose pas, il faut faire ce travail à la main. Nous verrons ultérieurement comment s'y prendre sans trop pénaliser la lisibilité du code.

Les fonctions terminales, sans état et non réentrantes

Maintenant, il existe des fonctions particulières pour lesquelles le problème de la gestion de la mémoire est plus simple. Ce sont les fonctions teminales, sans état et non réentrantes.

fonction teminale
: Une fonction qui n'en appelle pas d'autres.

fonction sans état
: Une fonction qui ne mémorise pas les précédents appels.

fonction non réentrante
: Une fonction qui ne peut pas être exécutée plusieurs fois en même temps. La réentrance n'est possible que si le processeur est capable d'exécuter deux taches en parallèle (ou plutôt en pseudo parallélisme). C'est possible même en l'absence de système d'exploitation, par exemple, si un programme est interrompu en raison d'un événement quelconque alors qu'il était en train d'exécuter une fonction, et que le gestionnaire d'interruption veut lui aussi exécuter la même fonction. La fonction en question doit être impérativement réentrante.

Si une fonction n'est pas réentrante, alors elle ne peut pas être utilisée par le programme principal et par le gestionnaire d'interruption en même temps. On aura, fondamentalement 3 possiblités pour respecter cette contrainte:

1. interdire son usage pour l'un ou l'autre, c'est facile, c'est au programmeur de respecter cette contrainte.
2. masquer les interruptions avant d'utiliser la fonction, ainsi si le programme principal commence à l'utiliser, il est sûr que le gestionnaire d'interruption ne le fera pas puisqu'il est bloqué. Cette solution est simple mais brutale, car elle bloque toutes les interruptions, même si le gestionnaire des interruptions n'avait pas l'intention d'utiliser la fonction en question.
3. mettre en place un mécanisme permettant de devenir propriétaire de la fonction considérée comme une ressource, le temps de son utilisation.

Dans un permier temps, on utilisera la première solution, et plus tard on utilisera la troisième

Il n'est pas difficile de comprendre que toutes les fonctions ayant ces trois propriétés (terminale, sans état, non réentrante) peuvent partager la même zone mémoire que ce soit pour le passage de paramètres ou pour le travail.

Première petite librairie

Nous allons écrire une petite librairie de fonctions teminales, sans état et non réentrantes contenant pour le moment une seule fonction: `waitms`. La mémoire utilisée par ces fonctions sera prise dans une zone partagée par tous les bancs. Nous nous limiterons à 4 octets à partir de `0x70` que nous nommerons A0 à A3 (ils sont réservés dans picmips.inc, mais vous pouvez le faire vous même). Cette zone va servir au passage des arguments, à la récupération des résultats et aux variables locales. Certes 4 octets c'est peu, mais c'est souvent suffisant pour un grand nombre de fonctions de base. Les fonctions peuvent utiliser d'autres registres si c'est nécessaire.

Pour le passage des parametres, c'est-à-dire la copie dans les registres A0 à A3 et la récupération du résultat, nous pouvons utiliser des macros. Par exemple pour une fonction qui calcul la moyenne de deux entiers sur 2 bits.

La fonction:

```
f_moy:  movf    a1,w
        addwf   a0,f
        bcf     STATUS,c
        rrf     a0,f
        return
```

La macro d'appel:

```
moy macro @d, @s0, @s1
        movf    @s0,w
        movwf   a0
        movf    @s1,w
        movwf   a1
        call    f_moy
        movf    a0,w
        movwf   @d
    endm
```

De sorte que l'usage de la fonction se réduit à

```
     CBLOCK BANK1
        v0
        v1
        v2
     ENDC
     ...
     moy   v2, v1, v0
```

Question 6

Vous allez réécrire le programme `hello6` que nous nommerons maintenant `hello7` (vous devez commencer à vous y habituer). Etant donné qu'il n'est pas facile d'expliquer exactement ce que nous souhaitons vous voir programmer, nous vous avons préparé un cadre que vous allez remplir. Faites les ajouts et testez.

hello7.asm
: contient le programme principal et l'inclusion des macros et des fonctions de l'utilisateur. Ce fichier est inclus dans le fichier juste après le gestionnaire d'interruption. Réfléchissez à la raison de l'avoir inclus à cet endroit et pas au tout début du programme.

myfun.asm
: contient les macros d'appels des fonctions avec le passage des paramètres et les fonctions de l'utilisateur. Tout est donné hormis le code de la fonction `waitms` et la déclaration des registres A0 à A3.

Deuxième petite librairie

Afin de voir si vous avez compris comment marche cette petite librairie, vous allez ajouter quelques fonctions supplémentaires. Faites au moins les deux premières.

Pour chaque fonction, on vous dit quels sont les paramètres, quelle est son action et ce qu'elle rend en résultat. Les paramètres sont mis dans les registres A0 à A3, dans l'ordre et le résultat est mis à partir du registre A0. Cherchez sur le site \underline{www.microchip.com}, les algorithmes du random ou de la multiplication.

A0 est le nom symbolique d'un registre de travail choisi dans la zone partagée (entre 0x70 et 0x7F). W pour Working. Si A0 est à l'adresse 0x70, A1 est à 0x71, A2 est à 0x72 et enfin 0x73.

Question 7

`f_add2`	`[A1:A0] <- [A1:A0] + [A3:A2]`	addition signée sur 2 octets
`f_sub2`	`[A1:A0] <- [A1:A0] - [A3:A2]`	soustraction signée sur 2 octets
`f_sra2`	`[A1:A0] <- [A1:A0] >> A2`	décalage droite arithmétique de 0 à 16
`f_srl2`	`[A1:A0] <- [A1:A0] >> A2`	décalage droite logique de 0 à 16
`f_sll2`	`[A1:A0] <- [A1:A0] >> A2`	décalage gauche de 0 à 16 décalage
`f_ran2`	`[A1:A0] <- nombre aléatoire`	tire un entier entre 0 et 65535
`f_mul`	`[A1:A0] <- A1 * A0`	multiplication sur des nombres signés

Notation

- `[A1:A0]` : désigne un couple de registre contenant un mot de 16 bits. Les 8 bits de poids faibles sont dans le registre `A0`, les 8 bits de poids forts dans le registre `W1`.

```
; ------------------------------------------------------------------------------
; programme : hello7
; Date      : 20130208:1521
; Version   : 1
; Auteurs   : franck
; Notes     : suppose que le Quartz est à 20MHz
; Premier programme utilisant des fonctions et des macros mises en bibliothèque
; sinon même comportement que hello6
; ------------------------------------------------------------------------------
```

```
        list    p=16f877        ; définit le processeur cible
        include "p16f877.inc"   ; déclaration des noms de registres
        include "picmips.inc"   ; macros à la mips
        include "myfun.asm"     ; fonctions utilisateurs

        ; Définition du registre de configuration du PIC
        ; _CP_OFF   : le code n'est pas protégé et peut être relu
        ; _WDT_OFF  : pas de timer watch dog
        ; _PWRTE_ON : attente d'un délai après le power on
        ; _HS_OSC   : oscillateur à quartz
        ; _LVP_OFF  : pas de mode programmation basse tension
        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF

; reset + gestionnaire d'interruption
; ------------------------------------------------------------------------------
        org     0
        call    initialisation
        goto    main            ; sauter le code du gestionnaire d'interruption

        org     4               ; adresse du vecteur d'interruption
        retfie                  ; par défaut ne rien faire

; initialisation générale
; ------------------------------------------------------------------------------
initialisation
        BANKSEL TRISD           ; aller dans le banc de registre TRSID
        clrf    TRISD           ; place tous les signaux du port D en sortie
        BANKSEL PORTD           ; revenir dans le banc de registre PORTD

; programme principal
; ------------------------------------------------------------------------------
main    movlf   1,PORTD         ; met 0x01 sur port D
loop    rr8f    PORTD           ; rotation vers la droite du port D
        movlf2  D'10000', A0    ; 10000 * 0.1ms = 1s
        call    waitms          ;
        goto    loop            ; on boucle sur la rotation

; ------------------------------------------------------------------------------
        END                     ; directive terminant un programme

; ------------------------------------------------------------------------------
; library   : myfun
; Date      : 20130208:1526
; Version   : 1
; Auteurs   : franck
; Notes     :
; - mapping mémoire réservé
; - macros basiques extension de l'assembleur pic
; - fonctions non réentrantes, terminales, sans état.
; ------------------------------------------------------------------------------

          ORG .....
; ==============================================================================
; macros basiques
; ==============================================================================

movlf   macro l,r ;------------- r <- l
           movlw l
           movwf r
        endm

movlf2  macro l,r ;------------- r <- l sur 2 octets
           movlf low(l),r
           movlf high(l),r+1
        endm
```

```
subwf2  macro r ;--------------- r <- r-w sur 2 octets ; positionne le bit Z (5 cycles)
           subwf r,f
           btfss STATUS,C  ; C=0 si r passe de 0 à FF
           decf  r+1,f     ; si C==0 alors décrémenter le poids fort
           movf  r,w       ; positionne Z uniquement...
           iorwf r+1,w     ; ...si r ET r+1 sont à 0
        endm

rr8f    macro r ;---------------- r <- (r&1)|((r>>1)&0x7F) : rot. droite sur 8 bits
           rrf   r,w       ; initialise carry avec le bit n°0 de r
           rrf   r,f       ; r[7..0] <- r[6..0],carry
        endm

; ============================================================================
; fonctions non réentrantes, terminales, sans état.
; ------------------------------------------------
; - Les fonctions de cette bibiothèque utilisent exclusivement les registres
;   de A0 à A3, à la fois pour le passage de paramètres, la remise de résultat
;   et les variables locales.
; - Il n'est pas interdit d'utiliser des routines mais il faut rester dans la
;   limite des registre de A0 à A3
; - attention les labels doivent être locaux a ce fichier de manière à ne pas
;   crééer de conflits de noms avec d'autres fichiers
; ============================================================================

; waitms
; ----------------------------------------------------------------------------
; IN     : [A1,A0] = mombre sur 16 bits (A0 octet de poid faible)
; OUT    : rien
; ACTION : routine d'attente active millisecondes
;          attente de [A1,A0] * 0.1ms
; TAILLE : 13 octets
; DUREE  : [A1,A0] * 0.1ms
; ----------------------------------------------------------------------------
f_waitms
; ****************************************************************************
; QUESTION : PLACER LE CODE DE LA FONCTION waitms QUI TRAVAILLE SUR A1 et A0
; ****************************************************************************
        return

f_waitus ; routine d'attente active (W)us (W>=2)
        movwf   A2              ; W -> A2
        decf    A2,f            ; pour attendre précisement W us
        goto    $+1             ; nop2
        decfsz  A2,f            ; A2--
        goto    $-2             ; goto adr courante -2
        return                  ;
```