Prise en mains de l'assembleur pic et des outils de développement

Objectif

Le but de ce premier TME est de vous faire une présentation progressive des outils de développement pic sur la base d'un programme très simple, un chenillard, dont vous allez construire et simuler 3 variantes. Vous pourrez aussi tester les séquences de programme écrits la semaine dernière. Nous n'utiliserons pas la maquette au cours de ce premier TME.

Flot de conception

Aide mémoire sur l'assembleur pic16f877

- Noms des symboles du pic16f877
- Liste des instructions.

Les outils

On peut distinguer 5 étapes pour la conception d'un programme en assembleur, avec des rebouclages à chaque étape.

1. `vi ou emacs` spécification et saisie du programme
2. gpasm Assemblage pic (c'est en fait un des outils de la suite logicielle gputils)
3. gpsim Simulation / debugage graphique PIC
4. picprog programmation et chargement de code dans le µC
5. test

Spécification

L'analyse du problème est une étape nécessaire quelque soit le langage de programmation utilisé. Toutefois, dans le cas d'un système embarqué et en particulier dans cas d'un microcontrôleur, l'analyse ne peut être faîte correctement que si on a une connaissance précise du microcontrôleur visé et du matériel qui l'environne. Ce module a pour but de vous donner une méthode de travail vous permettant d'aborder la programmation en assembleur du pic16, mais la méthode est valable pour n'importe quel microcontrôleur. Il faut connaitre :

- le microcontrôleur et son langage d'assemblage,
- les modules périphériques internes et externes utilisés,
- la méthode de prise en compte des évènements,
- et avoir une méthode de programmation.

La saisie du programme se fait avec un éditeur classique. Comme tous les programmes ont une structure commune, il est bien de fabriquer des modèles de programmes que l'on adapte au besoin.

Assemblage

Le but d'un assembleur est de traduire un programme écrit sous forme d'une séquence d'instructions simples et de sauts conditionnels ou pas, en code machine pour un processeur donné. Un programme source est constitué d'un fichier avec l'extention .asm (par convention pour le pic), fichier qui peut en inclure d'autres avec les extentions .asm ou .inc (encore par convention).

L'assembleur est un langage plus complexe qu'il n'y parrait au départ. À la base, un programme en assembleur est constitué de lignes comportant au plus une instruction. Une ligne se découpe en quatre colonnes :

1. la première colonne commence en tout début de ligne et contient au plus un mot. Ce mot, que l'on appelle aussi symbole ou étiquette, est associé à une valeur numérique représentant le plus souvent l'adresse de la case mémoire de l'instruction ou de la donnée courante. On peut aussi attribuer une valeur quelconque à une étiquette.
2. la deuxième colonne commence après un espace ou une tabulation, elle contient le plus souvent une instruction, mais elle peut contenir une directive, dont on verra le rôle un peu plus loin. Les instructions, comme les directives peuvent avoir des paramètres en nombre quelconque mais fixe.
3. la troisième colonne est séparée de la seconde par un espace ou une tabulation et contient la désignation des opérandes (s'il y en a) associées à l'instruction.
4. la quatrième colonne et dernière commence par un ; et s'achève au retour chariot. Elle contient un commentaire. {\bf Les commentaires sont très importants en assembleur}. Mais, il ne s'agit pas de décrire le comportement de chaque ligne, mais de décrire l'algorithme et la façon dont on l'utilise, c'est-à-dire la manière dont on passe les paramètres et ce qu'il en fait.

Une fois qu'un programme est écrit, on utilise un logiciel assembleur, ici **gpasm**, qui produit essentiellement trois fichiers:

- `.hex` code machine à écrire dans la mémoire programme du processeur.
- `.cod` code machine plus des informations sur le source destiné au simulateur.
- `.lst` contenant un journal de l'assemblage.

Simulation

Un programme pic est destiné à être exécuté sur un pic, mais il est très difficile de faire une mise au point directement sur le pic, car il n'est normalement pas possible d'afficher l'état de la mémoire en cours d'exécution du programme. Pour faire la mise au point on utilise donc un simulateur, qui se comporte comme le pic mais qui

permet une exécution en pas à pas et montre l'état des registres à chaque instant. Le simulateur que vous allez utiliser s'appelle **gpsim** et vous verrez qu'il permet des choses très utiles pour la mise au point des programmes.

Notez toutefois que le pic dispose d'un mode de fonctionnement qui permet de le stopper à chaque instruction pour lire les registres internes. Il existe aussi des émulateurs de pic qui se comportent comme des pic mais qui permettent de lire les registres à tout momment ou après qu'une condition soit réalisée. Nous n'utiliserons pas ses possibilités.

Tests

Les tests réels se font directement sur le processeur. Pour que ce soit possible, il faut que le processeur dispose d'une mémoire spéciale pour le code machine, et qu'un programme associé à un matériel, que l'on nomme un programmateur, puisse envoyer le code issu de l'assembleur dans le processeur.

Le pic que vous utilisez dispose d'une mémoire flash pour le programme et a la capacité de se faire reprogrammer en place (sans devoir le retirer de la plaque). Le programmateur utilise le port RS232 du PC et est piloté par le logiciel **picprog**. Nous utiliserons par la suite un bootloader.

Travaux pratiques

Mise en place de l'environnement

- Pour bash, ajouter dans le fichier `$HOME/.bashrc`

```
. ~encadr/micro/env/micro.env.bash
```

- Pour tcsh, ajouter dans le fichier `$HOME/.tcshrc`

```
source ~encadr/micro/env/micro.env.tcsh
```

Le but principal de ce script est d'ajouter à la variable PATH, le chemin où se trouve les outils de développement pic. Ne recopiez pas le script dans votre compte car celui-ci va évoluer en cours d'année.

- Vous allez maintenant creer un répertoire micro dans votre HOMEDIR

```
mkdir ~/micro ; cd ~/micro
mkdir tme1 ; cd tme1
```

- Vous créerez ensuite un répertoire par projet. Pour ce premier TME, vous avez trois programmes à écrire, plus deux autres si vous êtes rapide.

```
mkdir hello1 hello2 hello3
```

- Placez-vous enfin dans hello1 et éditez le fichier `hello1.asm`

```
cd hello1 ; vim hello1.asm
```

Le premier programme

L'objectif de ce premier programme est de réaliser un chenillard sur les 8 bits du port D. Plus précisément, un bit du port D est mis à 1, et les autres sont mis à 0, puis le 1 se déplace à gauche ou à droite, jusqu'à ce qu'il disparaisse pour se retrouver dans le bit carry du registre STATUS, et réapparaisse à l'autre extrémité du port D.

Le programme que nous allons écrire ne respecte pas les conventions d'écriture pic (pourtant obligatoire), mais ce n'a ici aucune importance, le simulateur va quand même l'accepter. Votre programme fait les choses suivantes:

1. Placer le PORTD en sortie, en écrivant dans le registre TRISD.
2. Mettre la valeur 1 dans le bit 1 du PORTD.
3. Faire une rotation du registre PORTD.
4. Boucler à l'étape 3 (le programme ne s'arrête jamais).

PORTD et TRISD sont deux registres spéciaux (Description de l'espace mémoire, rôle des registres spéciaux.) aux adresses respectives: 0x8 et 0x88.

Le code du programme se trouve en annexe. Vous devez le recopier, intelligemment (c.-à-d. en comprenant ce que vous faîtes), l'assembler et le simuler pas à pas en observant bien les changements d'état des registres.

- Pour faire l'assemblage vous utilisez la commande `gpasm`:

```
gpasm -pp16f877 hello1.asm
```

- simuler vous utilisez `gpsim`:

```
gpsim -pp16f877 -shello1.cod
```

Mieux, vous allez écrire un Makefile très simple dans le répertoire hello1, avec trois cibles .PHONY a, s et c, qui réalise respectivement l'assemblage, le lancement du simulateur, et le nettoyage du répertoire. Notez qu'à partir de la semaine prochaine vous utiliserez un script pour l'assemblage.

- Fichier `Makefile` dans le répertoire hello1

```
.PHONY: a s c
F=hello1
a:;  gpasm -pp16f877 $(F).asm
s:a; gpsim -pp16f877 -s$(F).cod
c:;  \rm *.cod *.hex *.lst 2> /dev/null || true
```

Vous allez voir que gpasm donne des messages lors de l'assemblage. Ici, gpasm vous demande de vérifier que vous avez bien effectué un changement de banc mémoire (initialisation des bits RP0 et RP1 du registre STATUS) avant d'accéder au registre 0x88. En effet, 0x88 est une adresse exprimée en hexadécimal (notation 0x) (ou b'10001000' en binaire) or seuls les 7 bits de poids faible de cette adresse sont codés dans l'instruction. Ces 7 bits sont complétés avec les 2 bits RP0 et RP1 du registre STATUS. Dès que votre programme fonctionne, passez au suivant. Ne faîtes pas trop d'essais avec gpsim sur hello1, si vous le faîtes tourner trop longtemps, gpsim plante car vous n'avez pas configuré le processeur correctement.

Profitez ce ce premier programme pour regarder ce qui est produit par l'assembleur, c'est à dire les fichiers .hex et .lst (pas le .cod qui est un binaire non éditable).

Le second, pareil mais en mieux

Le premier programme ne fait pas trois choses qui sont pourtant indispensables dans un programme.

1. l'utilisation des noms symboliques des registres et champs du pic utilisé.
2. la configuration du processeur.
3. la programmation du gestionnaire d'interruption.

Vous vous placez dans le répertoire `hello2`, et vous recopiez en le renommant le fichier `hello1.asm` en `hello2.asm` et le `Makefile`. Vous allez ensuite faire les modifications de ce programme suivant les indications qui vous sont données dans la suite.

Les noms symboliques de registres

Le pic dispose d'un grand nombre de registres spéciaux permettant d'accéder aux ressources internes. Ces registres portent des noms et sont présentés dans la documentation technique du processeur (noms des symboles). On peut bien sûr utiliser directement leur numéro mais au détriment de la lisibilité et de la portabilité du programme. C'est pourquoi, pour chaque pic, il existe un fichier contenant la définition des noms symboliques de chaque registre spécial. Ce fichier est inclus par la directive `include`. De plus, il peut être utile d'indiquer le nom du processeur cible dans le programme, ceci se fait par la directive `list`. Il est alors inutile de donner le nom du processeur lors de l'assemblage.

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

Vous pouvez aussi donner des noms aux registres destinés aux programmes avec la directive CBLOCK vue en cours.

La configuration du processeur

Le processeur dispose d'un registre de configuration à l'adresse 0x2007 dans la mémoire de programme permettant de lui donner certains comportements particuliers. Par exemple, on peut protéger en lecture la mémoire de programme, protéger en écriture la mémoire data eeprom, demander la mise en place du watchdog timer (nous verrons l'utilité de ce chien de garde ultérieurement), etc. Vous trouverez le détail de cette configuration à la page 120(5_pic16f877_conf_reset_int_wd.pdf) de la documentation technique 16f877. Pour demander une configuration particulière vous devez utiliser la directive `__CONFIG` de gpasm. En fait, vous utiliserez presque toujours la même configuration quel que soit vos programmes (sauf cas particulier), en l'occurence vous ajouterez en tête de vos programmes (au début de `hello2.asm`), mais après l'`include` et le `list`.

```
; Definition du registre de configuration du pic
; _CP_OFF   : le code n'est pas protege et peut être relu
; _WDT_OFF  : pas de timer watch dog
; _PWRTE_ON : attente d'un délai de 72ms apres le power on
; _HS_OSC   : le pic utilise un oscillateur à quartz
; _LVP_OFF  : pas de mode programmation basse tension
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF
```

Pour voir, l'effet de cette commande, après avoir fait l'assemblage, ouvrez le fichier `hello2.lst`, faites le décodage binaire du contenu de l'adresse 0x2007 et vérifiez page 120(5_pic16f877_conf_reset_int_wd.pdf).

La programmation du gestionnaire d'interruptions

Le pic16f877 réserve deux adresses pour les interruptions, l'adresse 0 et l'adresse 4. L'adresse 0 est utilisée pour l'interruption reset, l'adresse 4 est utilisée pour les interruptions normales (hors reset).

L'interruption reset n'est pas masquable, elle survient pour différentes raisons définies par le mot de configuration. Dans tous les cas, le programme en cours d'exécution s'interrompt et le pic va à l'adresse 0. Le registre STATUS permet de savoir la cause pour laquelle le code du reset a démarré. Les raisons sont entre autres:

- On active le signal MCLR, en mettant à 0 la broche du pic correspondante.
- Une baisse de tension d'alimentation a été detectée.
- Le compteur associé au watchdog a débordé.

L'interruption normale du pic16f877 survient pour 14 causes différentes. Ces interruptions sont masquables. Nous les verrons en détail plus tard. Dans tous les cas, le programme en cours d'exécution s'interrompt et le pic va à l'adresse 4. Différents registres permettent de connaitre la ou les causes de l'interruption. Quoi qu'il en soit, comme

tous les programmes réels utilisent les interruptions, il faut réserver l'adresse 4 à cet usage.

Modifiez votre programme, en utilisant la directive ORG, de manière à ce qu'à l'adresse 0, on fasse un appel à une routine INITIALISATION qui mette le port D en sortie, puis fasse un goto au programme MAIN qui réalise le chenillard sur le port D. La routine d'interruption sera reduite à l'instruction `retfie`.

Ce programme se trouve également en annexe. Essayez de ne pas regarder la correction tout de suite!

Le troisième, le même avec des macros

L'assembleur pic est très frustre. Il y a peu d'instructions et celles-ci font peu de chose. C'est à la fois, un avantage car il y a peu de choses à apprendre et un inconvénient car le texte des programmes est assez long et donc source d'erreur. Pour améliorer un peu les choses, il est possible de définir des macros instructions (ou simplement macros). Il est très simple de créer une macro. Les macros sont placées au début, avant de les utiliser.

syntaxe d'une macro

```
coop     macro   [args]...
            séquence d'instruction utilisant les args s'il y en a
         endm
```

exemple

Deux macros pour initialiser une valeur immédiate dans un registre, et pour faire la somme de deux registres sur 1 octet.

```
li1      macro   @reg, @val     ; @reg <- @val
            movlw @val          ; la valeur @val est placée dans w
            movwf @ref          ; w est recopié dans le registre reg
         endm

add1     macro   @rd, @rs, @rt  ; @rd <- @rs + @rt
            movf  @rs,w
            addwf @rt,w
            movwf @rd
         endm
```

Dans le code on pourra alors écrire

```
CBLOCK 0x20
   A : 1
   B : 1
   C : 1
ENC

li1  A, 46
li1  B, 56
add1 C, A, B
```

Certaines macros, particulièrement utiles, ont été intégrées à l'assembleur gpasm. C'est le cas de la macro `BANKSEL <r>` qui prend comme paramètre `<r>` un numéro de registre (entre 0 et 511[0x1FF] ou plus simplement sous sa forme symbolique) et qui initialise les bits RP0 et RP1 du registre STATUS en fonction du numéro du banc du registre. Entre 0 et 127[0x7F], les bits RP1 et RP0 sont mis à 0; entre 128[0x80] et 255[0xFF], les bits RP1 et RP0 sont mis respectivement à 0 et 1; entre 256[0x100] et 383[0x17F], les bits RP1 et RP0 sont mis respectivement à 1 et 0; enfin entre 384[0x180] et 511[0x1FF] les bits RP1 et RP0 sont mis tous les deux à 1.

Recopiez en le renommant le fichier hello2.asm en hello3.asm dans le répertoire hello3, ainsi que le Makefile, puis utilisez la macro BANKSEL avant d'utiliser TRISD et PORTD et utiliser la macro li1.

La suite?

k2000
: une variante de hello3 avec un test en plus. Vous devez faire un balayage du port D avec le bit à 1, de gauche à droite puis de droite à gauche. Pour celà, il vous faut faire un test pour savoir si vous êtes au bit 0 (à droite) ou au bit 7 (à gauche) afin de changer de sens de décalage. Vous devez utiliser un registre pour gérer le balayage, utilisez le registre 0x20.

k2000_onoff
: k2000 mais le balayage dépend de la valeur de la broche B0. Si B0 vaut 0, on stoppe le balayage, si B0 vaut 1 on reprend. Vous pouvez aussi essayer les séquences de programme demandées lors du premier TD.

Annexe

hello1

```
; --------------------------------------------------------------------------
; - ce programme est volontairement petit et ne fait appel a aucun fichier
; - Il ne doit pas etre considéré comme un modèle pour d'autres programmes
; --------------------------------------------------------------------------

        ; ces deux lignes permettent de sélectionner le bank n°1
        bcf     0x3,6       ; STATUS(RP1) <- 0
        bsf     0x3,5       ; STATUS(RP0) <- 1  pour désigner le banc 1

        ; programme le PORTD en sortie
        clrf    0x88        ; place tous les signaux du port D en sortie

        ; on change à nouveau de bank
        bcf     0x3,5       ; STATUS(RP0) <- 0 revient sur le banc 0

        ; on place 1 dans le registre relié au port D
        movlw   1           ;
        movwf   0x8         ; met 1 sur le bit 0 sur port D

loop
        ; rotation vers la droite sur 9 bits car utilise le bit carry
        rrf     0x8,f       ; rotation vers la droite du port D

        ; on boucle sans fin
        goto    loop        ; on boucle sur la rotation

; --------------------------------------------------------------------------
        END                 ; directive de fin de programme
```

hello2

```
; --------------------------------------------------------------------------
; - ce programme a le même comportement que hello1 mais il réserve l'usage
;   des adresses 0 et 4 pour le reset et l'interruption
; --------------------------------------------------------------------------

        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 de 72ms apres le power on
        ; _HS_OSC   : le pic utilise un oscillateur à quartz
```

```
        ; _LVP_OFF  : pas de mode programmation basse tension
        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF

        ORG     0
        call    initialisation
        goto    main

        ORG     4
        retfie

initialisation
        bcf     STATUS,RP1      ; STATUS(RP1) <- 0
        bsf     STATUS,RP0      ; STATUS(RP0) <- 1 permet de désigner le banc 1
        clrf    TRISD           ; place tous les signaux du port D en sortie
        bcf     STATUS,RP0      ; STATUS(RP0) <- 0 revient sur le banc 0
        return

main
        movlw   1               ;
        movwf   PORTD           ; met 1 sur le bit 0 sur port D
loop
        rrf     PORTD,f         ; rotation vers la droite du port D
        goto    loop            ; on boucle sur la rotation

; ------------------------------------------------------------------------------
        END                     ; directive de fin de programme
```