Assembleur CALM

Les notations de l'assembleur CALM (Common Assembly Language for Microprocessors) ont été définies en 1975 pour unifier les notations entre les nouveaux microprocesseurs de l'époque. Elles ont été modifiées en 1984 pour tenir compte des processeurs 32 bits et Partick Faeh a ré-écrit l'assembleur pour PC. Les notations Calm sont simples et logiques. Elles notent explicitement les opérandes, ce que les notations de Microchip ne font pas, induisant souvent en erreur le débutant.
Voir les spécifications complètes sous http://www.didel.com/calm/
L'assembleur Calm est actuellement réécrit pour mieux coller à la famille PIC de Microchip. Voir http://sourceforge.net/projects/calm/

L'environnement de développement SmileNG permet d'éditer des programmes avec les ordres de mise en page permettant même de coller des figures comme commentaires. L'assembleur est appelé sur un clic et les erreurs apparaissent dans le source.
SmileNG n'est plus intéressant pour les processeurs 16 et 32 bits, mais il est bien supporté, utilisé et apprécié pour les PICs de Microchip, dont les notations d'assembleur sont aberrantes. Voir http://www.didel.com/calm/Calm4Pics.pdf (en anglais).
Avec le Pickit2 pour programmer n'importe quel PIC, les boucles de déverminage d'un programme sont rapides et motivantes.
L'exécutable de SmileNG avec des assembleurs CALM pour PIC 10F 12F 16F se trouve sous www.didel.com/pic/SmileNG.zip
Pour la documentation de l'éditeur SmileNG et ses ordres de mise en page, voir http://www.didel.com/pic/SmileNG.pdf .

Ce document ne présente que les aspects de CALM utiles à un utilisateur écrivant des programmes courts ayant une structure simple. Les programmes complexes se font maintenant en C. L'assembleur reste essentiel pour programmer des processeurs qui ont des contraintes de prix ou d'encombrement, et il est plus facile à maîtriser que certains le prétendent.
Le dernier cours (2009) pour expliquer les techniques de programmation en assembleur sur PICs se trouve en http://www.didel.com/pic/Programmer.pdf

Pour une vision plus générale, lire http://en.wikipedia.org/wiki/Assembly_language (en anglais)

2.1 Instructions du PIC 16F877

L'argument publicitaire de Microchip est que le processeur n'a que 33 instructions. La compatibilité entre les différents processeurs de Microchip est plus intéressante, on peut développer pour un processeur simple en utilisant le 16F877 qui est le plus complet de la famille (documentation en prèparation).
Les instructions seront vues, analysées et testées progressivement dans les documents expliquant la programmation des PICs. Elles sont résumées dans la feuille de référence CALM, dont il est bon d'avoir une copie à portée de main pour vérifier l'existence d'une instruction ou son effet sur les fanions (carry, equal). Pour être à l'aise dans les fichiers assembleurs, il est bon d'avoir compris d'abord la sintaxe qui va envelopper les instructions qui nous intéressent.

2.2 Fichier assembleur

Le programme édité dans SmileNG est sauvé comme fichier avec l'extension **.asm** pour le programme principal et **.asi** pour les fichier insérés.
Ce fichier contient des instructions, des pseudo-instructions et des commentaires.

2.3 Assemblage

L'assemblage d'un programme génère un fichier de listage avec l'extension **.lst** et selon la configuration de SmileNG un fichier binaire en format Intel-hex (extension **.hex**) ou une table de symboles (extension **.ref**). La structure conseillée pour un programme ressort des exemples donnés dans notre documentation.
Le listage du programme **.lst** associe les instructions éditées avec l'adresse de ces instructions en mémoire et leur codage binaire, naturellement exprimé en hexa.
SmileNG permet des listages sur plusieurs colonnes qui permettent d'avoir une vision claire du programme, s'il a été écrit pour profiter des possibilités de mise en page.
Le fichier **.hex** est accepté par les chargeurs ou programmateurs de processeurs pour mettre le programme en mémoire du processeur et pouvoir l'exécuter.
Les fichiers **.pro** et **.ref** sont associés aux instructions et registres internes du processeur.

2.4 Syntaxe des lignes

Les lignes doivent respecter un format facilitant la traduction. Chaque ligne a au maximum 256 caractères. Dans la pratique, on se limitera à 60 pour être lisible à l'impression. Contrairement à d'autres assembleurs, il n'y a pas de contrainte en colonne1 (1er caractère de la ligne). Des espaces et tabulateurs peuvent être utilisés librement pour augmenter la lisibilité.
2.4.1 Commentaires; Les commentaires sont précédés d'un ; ou \. L'assembleur ignore les commentaires, mais l'éditeur interprète les ordres Lila (on/off selon F8).
2.4.2 Etiquettes: Les étiquettes sont des noms logiques pour des repères d'adressage du programme. Un "deux-point" est collé à la fin des étiquettes pour les distinguer des déclarations et instructions.
2.4.3 Instructions: Les instructions sont formées d'un code mnémotechnique suivi en général par un ou des opérandes. L'ordre des opérandes est source, destination ou 1^{e} op, 2^{e} op, destination.
Par exemple on peut avoir la ligne

```
Etiq:  Move #34,W  ; On ne devrait pas mettre un nombre dans une instr.
```

2.5 Nombres, symboles et expressions

Les nombres sont décimaux (34 ou 10'34), binaires (2'11010110) ou hexadécimaux (16'D6). L'assembleur convertit tout en binaire, représenté en hexa dans les listages.
Un symbole remplace un nombre en lui donnant sa signification. Par exemple si la durée d'un temps d'attente est de 52 unités, on déclarera (c'est une déclaration)

```
Attente = 52
```

Si une variable représentant une vitesse doit être déclarée dans la zone des variables, à la suite, on écrit

```
Vitesse: .16   1
```

Les minuscules et majuscules sont identifiées lors de la traduction. Les caractères spéciaux _ et ? sont admis. Nommer les symboles de façon logique et cohérente est la première priorité pour une bonne lisibilité des programmes.
Une expression est une combinaison arithmétique de nombres et symboles. On a les opérateurs + - * / ** (exposant) plus d'autres opérateurs: .and. .or. .xor. .ne. (non equal) .eq. (equal) .hi. (higher) .hs. (higher same) .ls. .lo. Si l'expression est complexe, on utilisera des parenthèses plutôt que de tester les règles de précédence. On peut écrire par exemple

```
Move  #2**Exp+Toto,W   (l'assembleur n'acceptera pas si le résultat
                        est supérieur à 255=16'FF)
.If   APC/256 .EQ. 0   (teste si l'adresse est en page 0)
```

A noter que A est une lettre qui peut être déclarée comme un symbole, donc un nombre (A = 3), "A" est un nombre, le code ASCII de la lettre A.

2.6 Pseudo-instructions

Les pseudo-instructions (pseudos pour être plus court) sont des commandes à l'assembleur, elles agissent sur l'insertion des fichiers, la mise en page et la traduction des instructions. Toutes les pseudos commencent par un point. Elles sont composées soit d'une pseudo-opération seulement, soit d'une pseudo-opération suivie par une ou plusieurs expressions (séparées par des virgules).
Seules les pseudos CALM qui ont un sens avec les processeurs 8 bits PIC et qui sont utilisées dans les programmes documentés par Didel sont documentées par la suite. Pour une liste plus exhaustive, voir les documents de 1983.

2.6.1 .Proc

Insère la description du processeur mentionné et génère le code machine pour les instructions de ce processeur.

```
        .Proc   16F877  ; Charge le fichier 16F877.pro
```

2.6.2 .Ref File

Insère la table des symboles qui correspond aux noms des registres et des bits du processeur choisi.

```
        .Ref    16F877 ; Charge le fichier  16F870.ref
```

Les fichiers .ref sont créés à partir de fichiers .asi dans un mode spécial de l'assembleur. On pourrait faire des .ref à partir des définitions et variables, pour accélérer l'assemblage (bien assez rapide par ailleurs).

2.6.3 .Ins File.xxx

Insère le fichier mentionné. Pour un programme complexe et de grande taille il est souvent commode de diviser le programme en plusieurs fichiers. Le programme principal insère ensuite ces modules séparés. Une bonne modularité permet de récupérer des blocs de programmes et gagner du temps dans des nouvelles applications.

```
        .Ins    Definitions.asi
```

2.6.4 .Loc

Assigne une nouvelle valeur au compteur d'adresse de l'assembleur (APC pour Assembler Program Counter).

```
        .Loc    16'10 ; la prochaine instruction sera en 16'10
```

A noter que les variables sont aussi assignées par un .loc 16'20 (ou .Loc DebVar), ce qui n'est pas très correct, puisque les variables sont dans une zone de variables qui n'a rien à voir avec la zone de programme.

2.6.5 .16 Valeur

Insère la valeur donnée dans le programme objet. Chaque valeur a la taille 14 ou 12 bits selon le PIC (et les 18F ont 16 bits). Uniformiser à 16 simplifie.

```
.16  "M","y","p","r","o","g"   ; L'assembleur met le code des lettres dans les
                                 positions successives de la mémoire.
```

2.6.6 .List Booleen (0 ou 1)

Les lignes d'instructions suivantes sont copiées dans le listage, si l'expression est vraie (TRUE). Les pseudo-instructions .LIST peuvent être imbriquées.
L'intérêt de cette pseudo-instruction est que l'on peut avec un .List 0 décider qu'une partie bien connue ne sera pas imprimée.

```
        .List   All
            ...  ; cette partie du programme apparait si All = 1
        .Endlist
```

2.6.7 .Endlist
Termine une section de listage conditionnel.
On peut ajouter un texte après la pseudo-instruction .Endlist, sur la même ligne. Ce texte est ignoré par l'assembleur, mais augmente la lisibilité si le programme est long et/ou des pseudo-instructions .List sont imbriquées. Ci-dessus, on aurait pu écrire **`.Endlist All`**

2.6.8 .End
Termine les instructions du programme. Des commentaires supplémentaires peuvent se trouver après cette pseudo-instruction, sans contrainte de longueur de lignes.
L'assembleur ignore toutes les lignes d'instructions éventuelles après cette pseudo-instruction. La pseudo-instruction .End peut aussi terminer les fichiers insérés (.asi) et c'est conseillé de terminer tous les modules de programme par un .End..

```
      .End
Blalbla, plus besoin de ; et de lignes de moins de 256 caractères
```

2.6.9 .IF
L'assemblage conditionnel est très utile et permet qu'un même programme génère par exemple du code pour différents processeurs.
Après un .IF, les lignes d'instructions suivantes jusqu'au .ELSE ou .ENDIF correspondant sont assemblées seulement si l'expression est vraie (TRUE). Les .IF peuvent être imbriqués.

```
        .If     Debug
             ...   ; est assemblé si Debug = 1 (en fait ≠0)
        .Endif Debug    ...
```

2.6.10 .Else
Les lignes d'instructions suivantes jusqu'au .Endif correspondant sont assemblées, si l'expression du .If correspondant était fausse (False). Un texte quelconque suivant .Else est ignoré par l'assembleur.
Le texte qui suit la pseudo-instruction .Else est seulement une aide supplémentaire pour le programmeur. C'est surtout le cas, quand les pseudo-instructions .If, .Else et .Endif sont très éloignées et imbriquées.

2.6.11 .Endif
Termine une section d'un .If ou .Else. Un texte quelconque suivant .Endif est ignoré par l'assembleur.
Le texte qui suit la pseudo-instruction .Endif est seulement une aide supplémentaire pour le programmeur. C'est surtout le cas, quand les pseudo-instructions .If, .Else et .Endif sont très éloignées et imbriquées.

2.7 Macroinstructions
Les macros sont pratiques pour remplacer des groupes d'instructions répétitifs. Ce qui suit ne documente que leur utilisation simple. Lorsque la macro a été définie, une instruction unique, appelée macro-instruction, copie dans le programme une ou plusieurs instructions d'assembleur et le programmeur est libéré de répétitions encombrantes.

`.Macro Nom`

Commence la définition d'une macro et définit son nom. Une autre macro ne peut pas être définie à l'intérieur de la définition d'une macro. Mais on peut appeler d'autres macros dans une macro, si elles sont définies avant.

`.Endmacro`

Termine la définition d'une macro. Exemple:

```
      .Macro  Delai4us; délai de 4 microsecondes
      Nop
      Nop
      Nop
      Nop
      .EndMacro
```

Dans le programme, on écrit

```
Delai4us
```
et l'assembleur insère les 4 instructions de la macro. Les commentaires d'une macro sont supprimés du listage.

On peut définir des macros comme des instructions d'un langage de plus haut niveau, ou créer des instructions manquantes. Ceci est surtout intéressant pour le programmeur puisqu'il conçoit lui-même ce langage. En abuser réduit la lisibilité pour d'autres programmeurs. Il y a deux autres problèmes avec le macros :
On perd le contrôle du temps d'exécution du programme, puisque qu'une macro prend un temps dépendant du nombre de ses instructions
On n'a plus la visibilité de l'effet de la macro sur le registre W, sur les variables utilisées et sur les fanions. Dans le listage, les macros sont explicitées et ces deux critiques disparaissent.

Les macros sont fortement recommandées pour les instructions d'entrée-sortie qui peuvent être amenées à changer de place d'une implémentation à l'autre ou selon le processeur utilisé.

```
.Macro SkipIfPoussoir
TestSkip,BC PortA:#bPoussoir
.Endmacro
```

Cette macro permettra d'écrire SkipIfPoussoir dans le programme, et si dans une autre application le poussoir est sur un autre port, il suffira de changer dans la macro, et pas partout dans le programme.

2.8 Sous-programmes et macro-instructions
Il ne faut pas confondre sous-programme et macro.
Un sous-programme (routine) est caractérisé par le fait qu'il n'y a qu'une copie du texte de ce sous-programme dans le programme. Le sous-programme est exécuté si on l'appelle (instruction Call). Les paramètres transmis au sous-programme modifient sa fonction. Les sous-programmes diminuent la longueur totale d'un programme et permettent une bonne modularité. Mais pour chaque appel, en plus de la préparation de paramètres éventuellement inutiles, il faut un Call et un Ret qui prennent chacun 2 microsecondes. La macro ou "le code de ligne" offre une solution plus rapide mais coûte de la place mémoire et de l'effort.
Les macros doivent toujours être définies avant d'être utilisées, ce qui n'est pas le cas de routines.

****

2.9 Autres pseudos
Les pseudos suivantes sont pour des spécialistes et seront ré-expliquées dans nos programmes si elles apparaissent..

2.9.1 .Align n
Aligne le compteur d'adresse APC sur un multiple de n, en général une puissance de 2.

2.9.2 .Fill.16 n,d
Remplit n positions mémoire avec la donnée d de 12 ou 14 bits selon le PIC

2.9.3 .Blk.16 n
Réserve une place de n mots mémoire. Chaque valeur a la taille 14 ou 12 bits selon le PIC.

```
.Loc  DebVar
Var1: .Blk.16    1     ; Variable 1 octet
Tabl: .Blk.16    4     ; Tableau de 4 variables d'un octet
```

2.9,4 .Base
Définit la nouvelle base par défaut pour les nombres.

L'assembleur CALM démarre avec une base décimale. Les bases courantes sont: binaire, octale, décimale et sexadécimale (hexadécimale). La nouvelle base doit être exprimée dans l'ancienne base sauf quand la base est explicitement placée en tête (p.ex. 16'10).

```
.Base   10'16
     Move  #0AB,W  ; un 0 est nécessaire si le 1er chiffre est hexa
.Base 10'2
     Move  #10101011,W
.Base 10'10
     Move  171,W
```

Chaque fois, la même valeur binaire est chargée dans W.

2.9.5 Macroinstructions à paramètre

Une macro peut avoir des paramètres. On peut regretter par exemple de devoir toujours écrire deux instructions pour déplacer une variable dans une autre:

```
Move Var1,W
Move W,Var2
```

Si on écrit

```
.Macro  Mov
Move %1,W
Move W,%2
,Endmacro
```

on peut écrire dans la suite du programme

```
Mov  Var1,Var2
```

Ceci cache malheureusement que le registre W est détruit.

Les paramètres de la macro sont appelés dans la partie principale de la macro par les symboles %1 à %8, dans l'ordre des paramètres de la macro-instruction. Les espaces et les tabulateurs sont permis à l'intérieur d'un paramètre.

Un meilleur exemple est une comparaison à trois paramètres: les deux opérandes et l'adresse de saut.

```
\b;JumpEQ  #Val,Var,Adr  ;saute à Adr si la valeur
                           ; de Var est égale à Val
.macro  JumpEQ
     Move  %1,W
     Xor   %2,W
     Skip,NE
     Jump  %3
.endmacro
```

En insérant l'instruction

```
JumpEQ     #Limit, PortA, Stop
```

le programme ne continue à l'instruction suivante que si la valeur lue sur le port A est différente de la limite donnée.

2.9.6 .MACRO avec paramètres par défaut

Les paramètres de la macro sont appelés dans la partie principale de la macro par les symboles %1 à %8, dans l'ordre des paramètres de la macro-instruction. Les espaces et les tabulateurs sont permis à l'intérieur d'un paramètre.

On peut en déclarant une macro donner une liste de paramètres correspondant aux valeurs par défaut.

```
.MACRO Nom, Param1,Param2, ...
```

(il peut y avoir un espace après la virgule, mais pas avant)
Par exemple, la Macro "Copy" peut par défaut copier le PortA sur le PortB, et si on précise le paramètre source ou destination, en tenir compte.

```
.Macro Copy, PortA, PortB
Move %1,W
Move W,%2
.EndMacro
```

Cette macro n'est en fait pas conseillée car elle ne facilite pas la relecture du programme. Par exemple, elle permet des paramètres par défaut, ce qui est amusant, mais génère facilement des erreurs, et ne facilite pas la compréhension.

2.9.7 Macros avec étiquettes locales

Dans une macro, on peut avoir besoin d'étiquettes. Si la macro est appelée plusieurs fois, ces étiquettes ne peuvent pas être identiques. Il faut les déclarer avec la pseudo-instruction

```
.Localmacro étiquette1,...,étiquette8
```

Cette pseudo-instruction définit jusqu'à huit étiquettes qui peuvent être utilisées dans la partie principale d'une macro. Ces étiquettes seront converties en étiquettes locales (M_0$..M_999$) si cette macro est appelée. La syntaxe permise pour les étiquettes est la même que pour les étiquettes dans l'assembleur CALM. Une virgule sépare les étiquettes. Si cette pseudo-instruction est utilisée, elle doit directement le .Macro.

```
.Macro Delai    ; paramètre: délai en multiple de 3us
.Localmacro  A
     Move  %1,W
     Move  W,C1
  A: DecSkip,EQ C1
     Jump  A
.EndMacro
```

L'appel se fait par

```
Delai 30   ; 30 * 3us
```
(espace ente Delai et le paramètre)

jdn060902