heig-vd
Haute Ecole d'Ingénierie et de Gestion
Du Canton du Vaud

Hes·so
Haute Ecole Spécialisée
de Suisse Occidentale

MICROINFORMATIQUE

NOTE D'APPLICATION 1 (REV. 2011)

ARITHMETIQUE EN ASSEMBLEUR ET EN C

Programmation en mode simulation

1. Documents de référence

Les documents de références pour cette note d'application sont :

1. ***Slides* Bases de numération**.
2. ***Slides* Introduction au TI MSP430**
3. ***Slides* Outil de développement.**
4. ***EW430_UserGuide.pdf*** : MSP430 IAR Embedded Workbench® IDE User Guide.
5. ***MSP430x4xx Family.pdf*** : chapitre 4 – 16-Bit MSP430X CPU, §4.6.2 MSP430 Instructions, pages 4.62 à 4.112.

2. But de cette note d'application

Le but de cette note d'application est de

1. Découvrir l'outil de développement IAR.
2. Expérimenter avec les mécanismes de base du calcul arithmétique, plus précisément l'addition, la soustraction et la multiplication en simple et multiprécision. A la base, le microcontrôleur utilisé possède une unité arithmétique et logique (ALU) de 16 bits. Le multiplicateur est un périphérique intégré.

3. Installation de IAR Embedded Workbench

Installer le programme sur votre PC.

Un installateur ainsi que le manuel de IAR peuvent être téléchargées à la page

http://php.iai.heig-vd.ch/~lzo/pmwiki/pmwiki.php/Microinformatique/Downloads

4. Organisation de l'espace de travail sur votre PC

Il est **très important** de **bien organiser l'espace de travail** pour la programmation du MSP430.

1. Vous commencerez par créer sur votre PC un directoire appelé **MSP430.**
2. Vous créerez ensuite dans celui-ci plusieurs sous-directoires qui correspondront aux diverses phases de ce cours :

   MSP430 →

   ASM (pour assembleur)
   C_simul (C en mode simulateur)
   EZ430
   Carte_IAI

   Chacun de ces **quatre directoires** constituera ensuite le répertoire pour un ***Workspace* IAR** distinct, dans lequel se trouvera (à la fin du point 9. du chapitre 5.2) un fichier de type **`.eww`** .

3. A son tour chaque directoire de *Workspace* comprendera **un sous-directoire pour chaque application (projet) développée**.
4. **En règle générale**, vous devrez donc ensuite **créer un nouveau projet** dans **un directoire distinct** pour **chaque nouvelle application** et programme demandés à défaut de cela on risque de perdre le programme précédent.

   Commencez doc par créer déjà dans chacun de ces directoires un premier directoire de projet (par exemple appelé **Projet_1**). On aura donc une arborescence de directoires de type suivant

   MSP430 → ASM → Projet_1

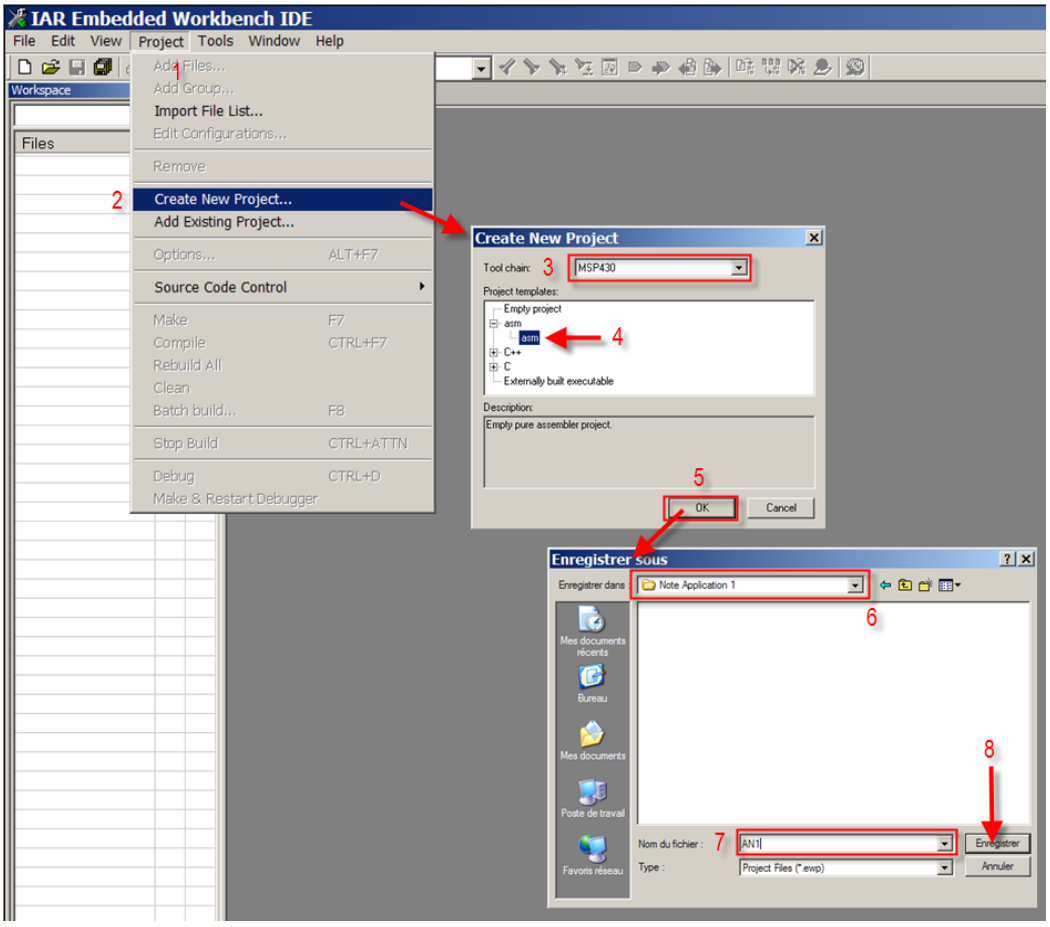
5. Création d'un *workspace* pour les projets en assembleur

5.1 OUVRIR LE PROGRAMME IAR

5.2 CRÉER LE PREMIER PROJET

Suivre les instructions des slides **Outil de développement**.

Créer donc un projet en assembleur dans un sous-directoire de votre workspace ASM.
Le code de base est généré automatiquement.



1. Pour créer un nouveau projet, sélectionner **Project**.
2. Dans le menu déroulant, sélectionner **Create New Project**.

3. S'assurer que la toolchain est MSP430.
4. Choisir un projet assembleur *asm* → *asm.*
5. Cliquer sur ok.
6. Cherchez le directoire ad hoc (voir la page 4) que vous aurez préalablement créé: par ex. *MSP430 / ASM / Projet_1* .
7. Donner un nom au projet (par exemple *prog*_1).
8. Et enregistrer le nouveau projet. Ceci créera dans *MSP430/ASM/Projet_1* le fichier *prog_1.ewp.*
9. Il faut maintenant encore sauver le nouveau Workspace: sélectionnez **File → Save Workspace**. Donnez un nom au Workspace (par ex. ASM, comme son directoire) et sauvez-le dans le directoire prédéfini (MSP430 / ASM).

   Ceci créera dans *MSP430/ASM* le fichier *ASM.eww* .

5.3 COMPILATION ET EXÉCUTION

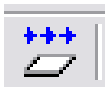
Compiler

 ,

lancer le mode Debug

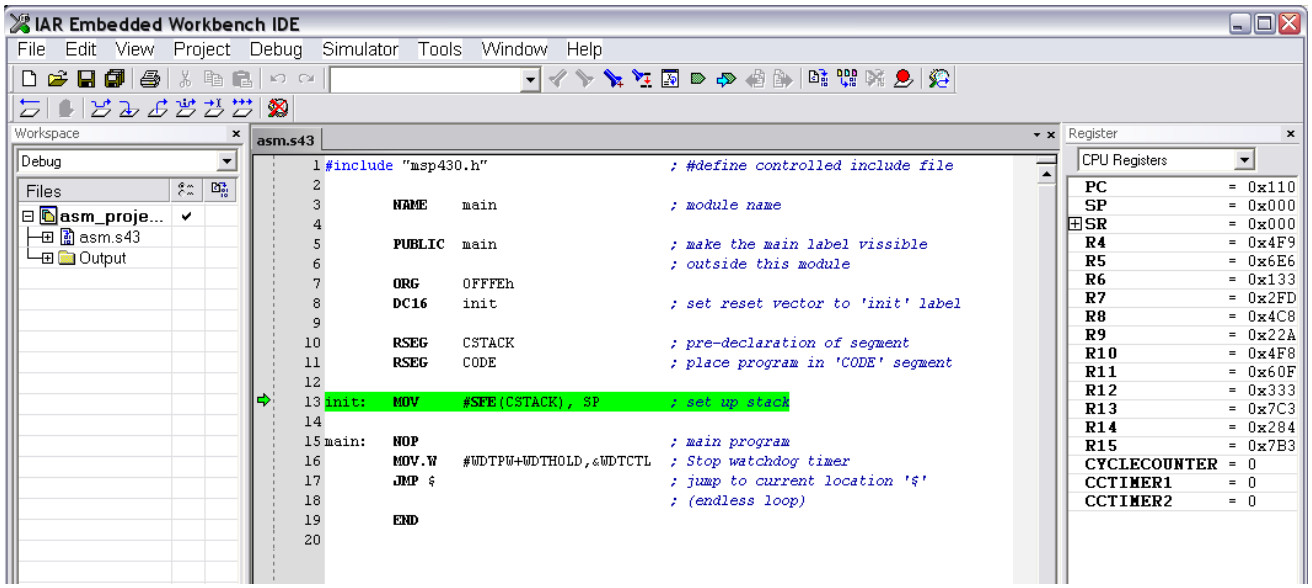
 ,

et exécuter

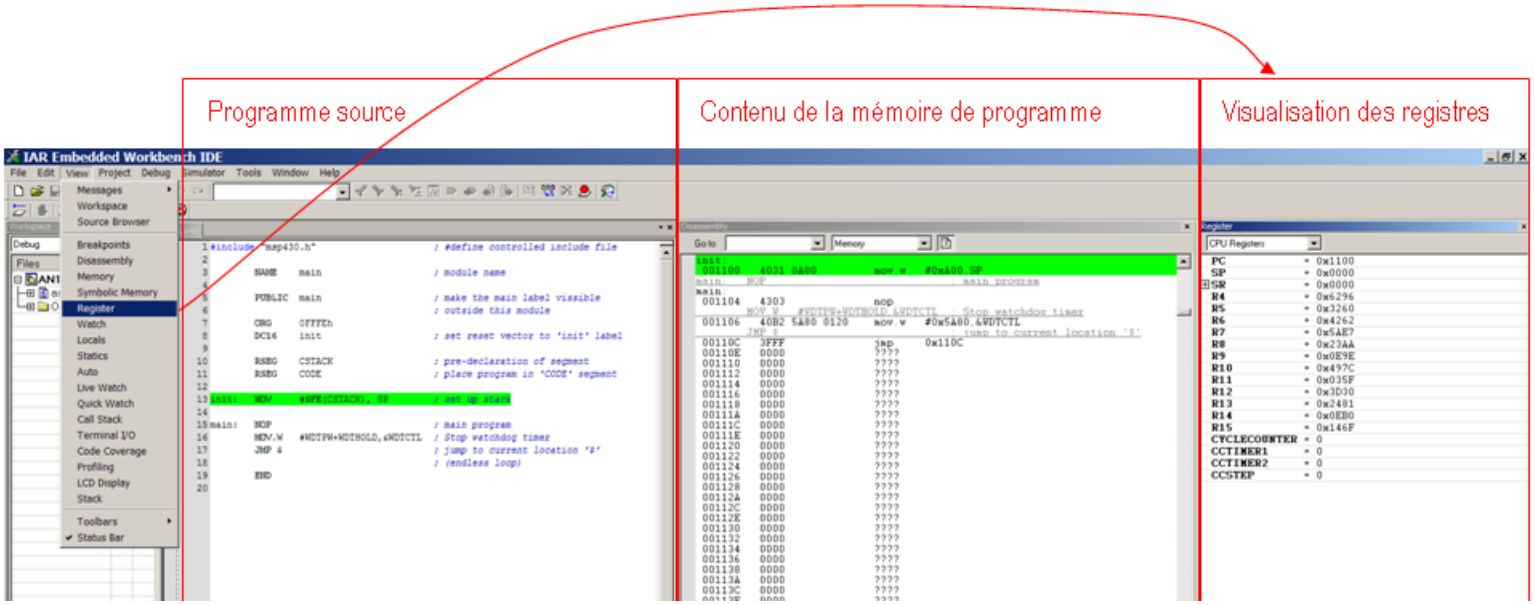
 ou  en mode pas à pas

le programme en assembleur en mode simulateur (menu Project -> Options, etc.).

Evidemment ce programme ne fait rien ... sauf tourner à vide, ce qui permet toutefois d'ouvrir toutes sortes de fenêtres de visualisation (menu *View*) de registres, etc..



- Ensuite familiarisez-vous avec tous le menus et les diverses fenêtres : édition du programme, options du projet, visualisation des registres, etc..
- Familiarisez-vous avec l'utilisation des **breakpoints**  pour arrêter le programme afin d'examiner l'état des ses variables.



6. Rapport

Le **rapport** sera un **compte-rendu** mis au propre **de vos propres notes** prises durant les principales taches à effectuées, décrites dans les chapitres suivante de cette note.

Au minimum il comprendra les **réponses à toutes les questions posée**s dans les chapitres suivants ainsi que

- Quelques additions et soustractions de nombres à 16 bits **en assembleur**, en notant les résultats (registres et bit d'état) par un copier-coller de la fenêtre des registres. Incluez svp quelques cas avec dépassement et vos observations des bits d'état.
- Au moins deux additions de deux nombres fractionnaires **en assembleur** (section 7.8).
- Quelques multiplications non-signées et signées, ainsi qu'une multiplication de deux nombres fractionnaires, **en assembleur**.
- **En C** des exemples d'utilisation de **tous les opérateurs binaires et de décalage**, en notant les résultats de la fenêtre des *locals*, ensuite traduits, et donc vérifiés, en binaire.

Pour chaque tache effectuée:

- Description sommaire de la tache, opération, calcul, etc.
- Le cas échéant copier-coller du listing produit et/ou des résultats visualisés.
- Constatations et suggestions si applicable.

7. Programmation élémentaire en Assembleur

7.1 GÉNÉRALITÉS

En simple précision, les opérations d'addition et de soustraction se font sur 16 bits. Les opérandes peuvent être signées ou non-signées. Selon la valeur de ces opérandes, il est possible que le résultat ne puisse pas s'exprimer sur 16 bits.

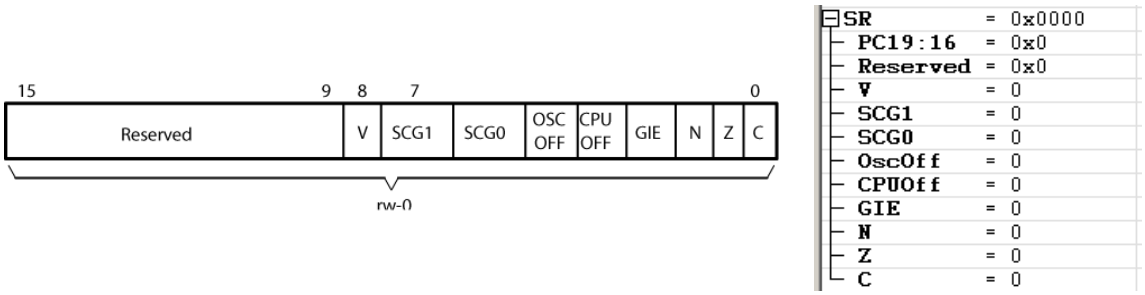


Figure 7-1 : Registre d'état SR

Un registre d'état nommé SR (*status register*) permet de savoir, à l'aide de quatre bits si le résultat de l'opération d'addition ou de soustraction est :

1. N=1 : résultat négatif,
2. Z=1 : résultat nul,
3. C=1 : report de la retenue à des fins de calcul multi précision,
4. V=1 : dépassement.

Réaliser et exécuter des courts programmes en mode simulateur avec les opérations suivantes

- **Exécuter** les programme en mode pas-à-pas
- **Observer** et commenter les valeurs prises par les registres (menu *View* → *Register*).

7.2 ASSIGNATION DE REGISTRE

Chargement d'une valeur de 8 bit (c.à.d. un *Byte*) dans le registre Rn: **MOV.B #valeur, Rn**

Chargement d'une valeur de 16 bit (c.à.d. un *Word*) dans le registre Rn: **MOV.W #valeur, Rn**

Dans chaque cas, observer le résultat dans le registre Rm et les bits d'état V, N, Z, C dans le registre SR. Pour visualiser SR, sélectionner dans la barre d'état de l'outil de développement (IAR Embedded Workbench IDE): *View* → *Register* et dans le menu déroulant de la fenêtre Register: CPU register.

Exemple :

```
MOV.W   #12, R12                ; Mettre une valeur à registre R12
```

```
R11 = 0x570D
R12 = 0x000C
R13 = 0x384B
R14 = 0x089D
```

7.3 ADDITION

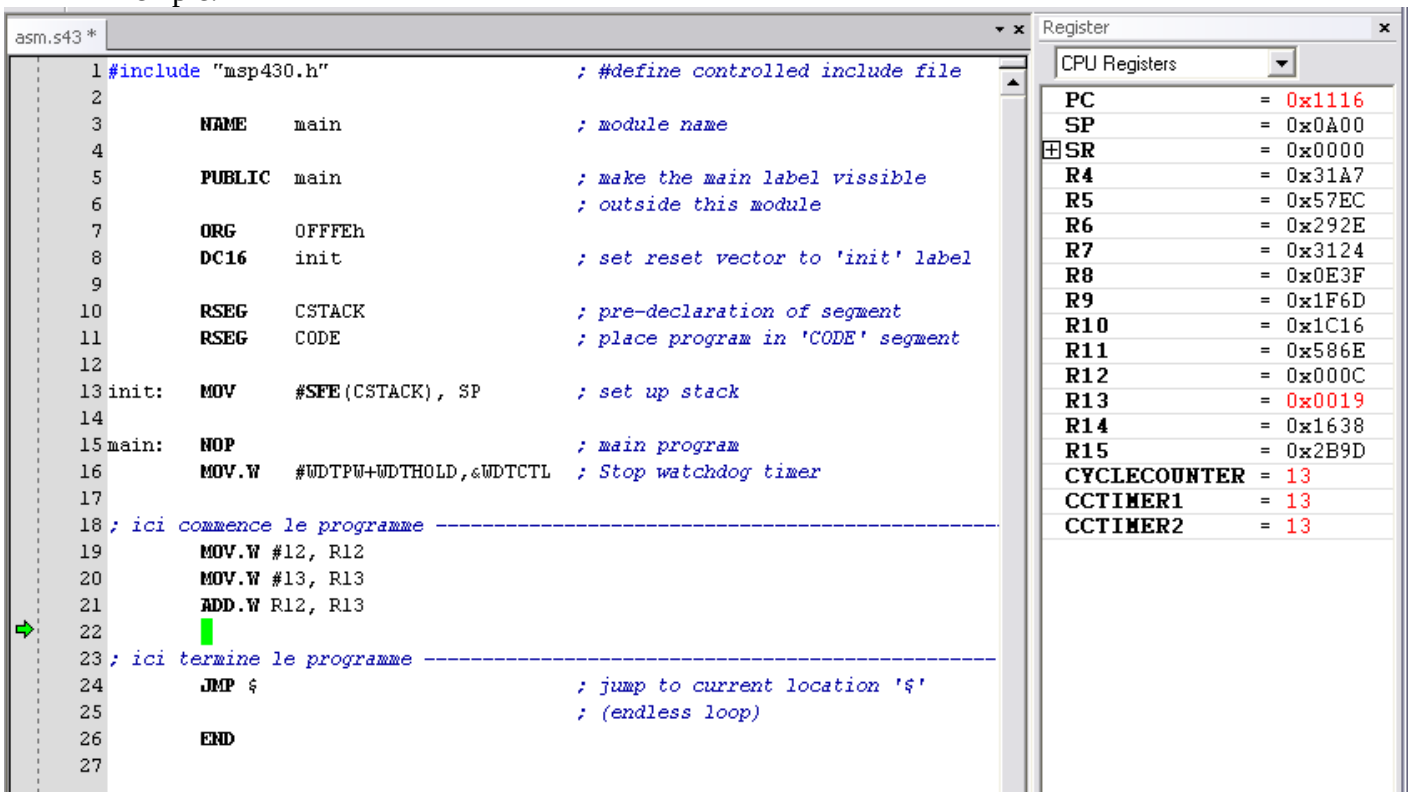
Addition de deux nombres de 8 bits: **ADD.B Rn, Rm**

Addition avec report de deux nombres de 8 bits **ADDC.B Rn, Rm**
(si le *carry* de l'opération précédente est 1, on le rajoute au résultat)

Addition de deux nombres de 16 bits: **ADD.W Rn, Rm**

Addition avec report de deux nombres de 16 bits **ADDC.W Rn, Rm**

Exemple:



Les valeurs en rouge sont celles modifiées lors de la dernière opération.
La valeur positive +12 plus la valeur positive +13, pas de dépassement V=0, pas de carry C=0, résultat pas égale zéro Z=0 et résultat est positif +25, donc N=0.

7.4 SOUSTRACTION

Soustraction de deux nombres de 8 bits: **SUB.B Rn, Rm**

Soustraction avec report de deux nombres de 8 bits: **SUBC.B Rn, Rm**

Soustraction de deux nombres de 16 bits: **SUB.W Rn, Rm**

Soustraction avec report de deux nombres de 16 bits: **SUBC.W Rn, Rm**

Attention : ceci calcule : [val(Rm) ─ val(Rn)], résultat trouvé dans Rm .

7.5 COMPARAISON

Comparaison d'un registre avec une valeur immédiat: **CMP.W #X, Rn**

Rn > X: V=0, **N=0**, Z=0, C=1
Rn = X: V=0, **N=0**, Z=1, C=1
Rn < X: V=0, **N=1**, Z=0, C=0

7.6 TEST

Comparaison d'un registre avec zéro: **TST.W Rn**

Rn>0: V=0, **N=0**, Z=0, C=1
Rn=0: V=0, **N=0**, Z=1, C=1
Rn<0: V=0, **N=1**, Z=0, C=1

7.7 ADDITIONS D'OPÉRANDES EN FORMAT ENTIER SIGNÉ DE 16 BITS

- Réaliser des exemples de **toutes les opérations d'additions**, avec des opérandes **en format entier signé**, donnant des résultats positifs, négatifs, sans et avec dépassement. Faire un tableau et expliquer les valeurs de chaque bit d'état.

  En cas de dépassement modifier ensuite les valeurs entrées pour limiter les valeurs positives à 2^{15}-1 (0x7FFF) et les valeurs négatives à $-(2^{15})$ (0x8000).

7.8 ADDITION D'OPÉRANDES EN FORMAT FRACTIONNAIRE SUR 16 BITS

- Réaliser ensuite l'addition de **deux nombres fractionnaires (0.75) + (4.1875)**.

  Ici pour illustration un exemple avec (-0.475) + (+12.73) :

```
*/
; 1.3.5 Addition d'opérandes en format fractionnaire sur 16 bits

; Pour calculer l'addition de nombre réel sur 16 bits, je reserve 1 bit pour
; signe, 4 bits pour entier et 11 bits pour fractionnaire.
; -0.475 = -1 + 0.515
; 12.73 + (-0.475) = 12.73 + (-1) + 0.515

; 12.73 = 01100.10111010111
; -1 = 11111.00000000000
; 0.515 = 00000.10000011110

        MOV.W   #0110010111010111b, R11
        MOV.W   #1111100000000000b, R12
        MOV.W   #0000010000011110b, R13
        ADD.W   R11, R12
        ADD.W   R12, R13                ; Resultat R13

        JMP $                           ; jump to current location '$'

        END                             ; (endless loop)
```

```
SR          = 0x0000
  Reserved  = 0x00
  V         = 0
  SCG1      = 0
  SCG0      = 0
  OscOff    = 0
  CPUOff    = 0
  GIE       = 0
  N         = 0
  Z         = 0
  C         = 0
R4          = 0x49F0
R5          = 0x653B
R6          = 0x3142
R7          = 0x7168
R8          = 0x0A74
R9          = 0x204B
R10         = 0x3387
R11         = 0x65D7
R12         = 0x5DD7
R13         = 0x61F5
R14         = 0x0EC3
R15         = 0x24DF
```

Réaliser dans un programme commenté ensuite l'addition de **deux nombres fractionnaires de votre choix** avec au moins 3 chiffres (décimales) après la virgule.

7.9 MULTIPLICATION EN ASSEMBLEUR

7.9.1 Généralités

En principe, pour couvrir tout les cas, la multiplication de deux nombres de 16 bits donne un résultat correct sur 32 bits. Programmation en assembleur

7.9.1.1 Instructions de multiplication

- Multiplication non signée: MOV.W *src*, &MPY
  MOV.W *src*, &OP2
- Multiplication signée: MOV.W *src*, &MPYS
  MOV.W *src*, &OP2
- Registres résultats: RESHI **et** RESLO

Avec *src*, opérande source, pouvant être attribué à une valeur immédiate: #nombre, ou un registre Rn (n=4 … 15) .

7.9.1.2 Instruction d'assignation de registre

- On récupère ensuite le résultat dans deux registres Rn.
- Par exemple: MOV.W &RESLO, R12
  MOV.W &RESHI, R11

7.9.1.3 Instruction de décalage

Parfois, il peut être utile de décaler le résultat afin de le rendre dans un format plus optimal.

- RLA.W *dst*; Décalage arithmétique à gauche

- RLC.W *dst*; Rotation à gauche à travers le Carry

Avec *dst*, registre de source (et destination)

7.9.2 Multiplication en format entier 16 bits x 16 bits avec résultat sur 32 bits

Réaliser quelques opérations de multiplication avec **multiplicande et multiplicateur positifs et négatifs**.

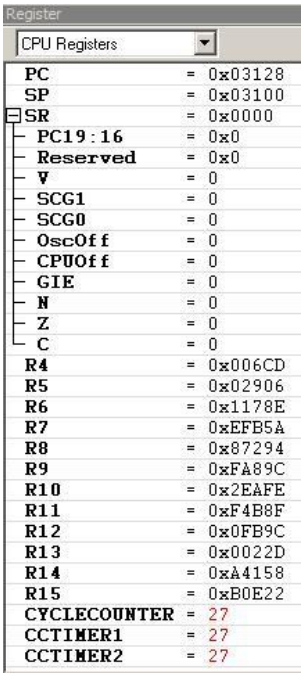
Vérifier et justifier vos résultats.

7.9.3 Multiplication en format fractionnaire 16 bits x 16 bits avec résultat sur 16 bits

Réaliser la multiplication de deux nombres fractionnaires, soit dans l'exemple qui suit (0.425)*(2.564).

Les formats des opérandes (multiplicateur et multiplicande) doivent être choisis de manière à avoir la précision maximum sur les nombres. Donner le résultat sous la forme d'un nombre fractionnaire de 16 bits au format optimal.

```
MOV.W   #0x06CD, R4             ; 0.425
MOV.W   #0x2906, R5             ; 2.564
MOV.W   R4, &MPY
MOV.W   R5, &OP2
MOV.W   &RESLO, R12             ;
MOV.W   &RESHI, R13
RLA.W   R12                     ; Décalage gauche arithmétique
RLC.W   R13                     ; Décalage gauche avec carry
```



Programmez ensuite **(0.75)*(4.1875)**.

8. Programmation en C

8.1 INTRODUCTION

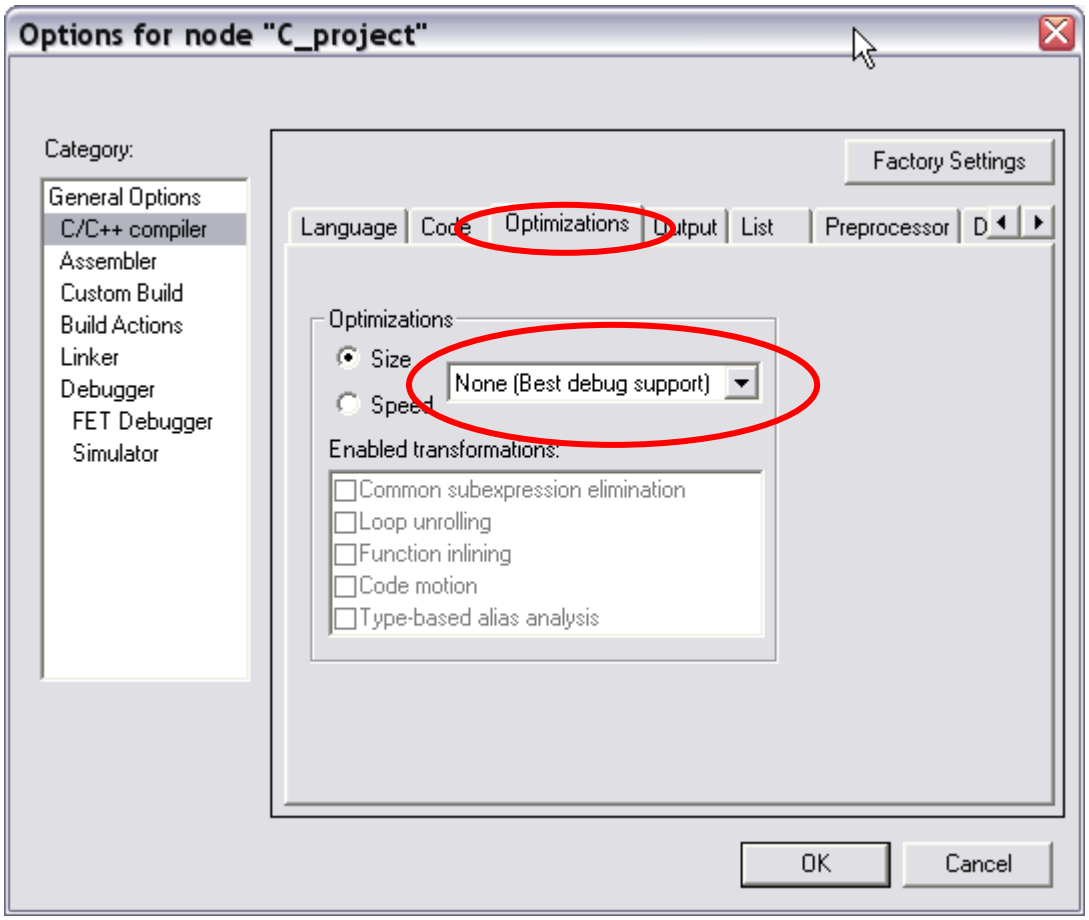
Les variables 16 bits signées sont déclarées en *signed int*. Les variables 32 bits signées sont déclarées en *signed long*

Le code assembleur généré est affiché de manière automatique dans une fenêtre nommée *Disassembly*. Si cette fenêtre n'apparaît pas, sélectionner dans la barres d'état de l'outil de développement (IAR Embedded Workbench IDE): *View→ Disassembly*.

Selon la manière dont est construit le programme, les variable déclarées sont directement assignée aux registres (R1, … R15). La lecture du code assembleur généré permet clairement de voir ou sont assignés les variables.

A l'aide de la dans la barres d'état de l'outil de développement (IAR Embedded Workbench IDE): *View→ Locals*, **il est possible de voir en cours d'exécution pas à pas le contenu des variables déclarées**.

Important: pour cela il est **nécessaire** que le code compilé **ne soit pas optimalisé**:



8.2 OPÉRATEURS

Les six opérateurs plus importants sont :

Signe	Nom	Exemple	
+	addition	int a = 4 + 10	// a vaut 14
-	soustraction	int a = 9 - 6	// a vaut 3
*	multiplication	int a = 8 * 9	// a vaut 72
/	division	int a = 9 / 3	// a vaut 3
%	modulo (reste de la division entière)	int a = 15 % 9	// a vaut 6
=	affectation	var1 = 3	

Les opérateurs possèdent des priorités à ne pas négliger.
Pour éviter les erreurs, ne pas hésiter à préciser l'ordre des opérations par des parenthèses.

Opérateurs d'accumulation

Les opérateurs d'accumulation sont des opérateurs qui ajoutent une valeur à la variable.
Par exemple, ma variable var1 vaut 10. Je veux lui ajouter 5. Je peux écrire *var1 = var1 + 5* ou *var1 += 5*. Cette expression est très souvent utilisée dans les boucles.

Voici un tableau des différents opérateurs d'accumulation :

Opérateur	Description	Exemple (au départ int i vaut 10)	
+=	addition	i += 23	/* i vaut 33 */
-=	soustraction	i -= 5	/*i vaut 5 */
*=	multiplication	i *= 10	/* i vaut 100 */
/=	division	i /= 2	/* i vaut 5 */
%=	modulo	i %= 6	/* i vaut 4 */

Attention: cette notation peut gêner les programmeurs habitués à d'autres langages.

Incrémentation

L'**incrémentation** est une autre forme d'accumulation :
Elle ajoute 1 à un nombre (cela reviendrait a écrire: var1 = var1 + 1) :

```
var1 += 1 ;
```

La syntaxe est donc plus courte.

8.3 ADDITION

Réaliser quelques opérations d'addition et vérifier les résultats obtenus. **Observer** et analyser **le code assembleur généré**.

8.4 MULTIPLICATION

Réaliser quelques opérations de multiplication et vérifier les résultats obtenus. Observer et analyser le code assembleur généré.

8.5 OPÉRATEURS BINAIRES

En plus des opérateurs arithmétiques usuels, que nous utilisons tout le temps, le C définit des opérateurs travaillant sur les bits.

Certains de ces opérateurs permettent de travailler bit à bit sur des valeurs; d'autres permettent de décaler, vers la droite ou la gauche, les bits d'une donnée.

Nous commencerons cette partie par un bref rappel de logique booléenne, puis nous présenterons les opérateurs correspondants.

8.5.1 Rappels de logique booléenne

La logique booléenne n'utilise que deux valeurs, 0, et 1, qui correspondent tout à fait à ce que l'on emploie lorsque l'on travaille en binaire.

Trois opérations, voire quatre, d'algèbre de Boole nous seront utiles lorsque nous programmerons en C:

- le OU (*or*, en anglais),
- le OU exclusif (*xor*),
- le ET (*and*),
- le NON (*not*).

Voici les tables de ces opérations:

a	b	a OR b	a XOR b	a AND b	NOT a
0	0	0	0	0	1
0	1	1	1	0	1
1	0	1	1	0	0
1	1	1	0	1	0

a OR b vaut 1 si a, ou b, ou les deux, valent 1.

a XOR b vaut si a ou b, mais pas les deux à la fois, valent 1.

a AND b vaut 1 si a et b valent tous les deux 1.

NOT a vaut 1 si a vaut 0, et, inversement, 0 si a vaut 1.

8.5.2 Opérateurs C bits à bits

Le C propose des opérateurs bits à bits permettant de faire ceci directement sur les bits composant une, ou deux, valeurs, selon l'opérateur. Notez que ces opérateurs ne travaillent qu'avec des valeurs, on des variables, de type entier: ils ne peuvent pas être utilisés sur des flottants !

Opérateur	Rôle
&	AND binaire bit à bit
|	OR binaire bit à bit
^	XOR exclusif bit à bit
~	Complement à un

Notez que |, ^, et & sont des opérateurs binaires, alors que le ~ est un opérateur unaire, qui n'a besoin que d'un seul opérateur.. Les trois premiers, en tant qu'opérateurs binaires, disposent d'une écriture abrégée pour les affectations, telle &=, par exemple.

Ci-dessous, un extrait de code source présentant quelques emplois de ces quatre opérateurs:

```
int a=10,
    b=20,
    c;

c = a | b;
c = a & b;
c = a ^ b;
c = ~a;
```

Quelle devient la valeur de c en binaire et en décimal ?

8.5.3 Opérateurs de décalage de bits

Un autre type permet de manipuler directement les bits d'une variable, ou d'une donnée.

Il s'agit des deux opérateurs de décalage de bits.

Opérateur	Rôle
<<	Décalage de N bits vers la gauche
>>	Décalage de N bits vers la droite

Le premier, >> (deux fois de suite le signe supérieur) permet de décaler les bits d'un nombre vers la droite, et le second, << (deux signes inférieur successifs), est son réciproque, c'est-à-dire qu'il décale vers la gauche.

Ces opérateurs s'utilisent de la façon suivante:

valeur OP N

Où *valeur* est une variable ou une valeur, OP est << ou >> , et N est le nombre de bits dont on souhaite décaler les bits de la donnée.

Notez que décaler de N bits vers la droite **revient à diviser par 2^N**, et décaler de N bits vers la gauche correspond à une **multiplication par 2^N**.

Voici un petit exemple, pour illustrer la syntaxe d'utilisation:

```
int a = 0x1C ;
int b;
b = a << 2;
b = a >> 4;
a <<= 3;
```

Quelles sont les valeurs de a et b en décimal, binaire et hexadécimal après chaque instruction ?

Un autre cas à étudier:

```
int a = 0x2A ;
int b;
b = a << 1;
b = b >> 3;
a <<= b;
```