Laboratoire 3 :   
Optimisation des programmes pour les DSP TMS320C6x

Le but de ce laboratoire est de se familiariser avec l’optimisation en assembleur sur la carte TMS320C6713. Il fait suite au cours et permet de rentrer concrètement dans le sujet. Code Composer Studio a été installé sur <http://processors.wiki.ti.com/index.php/Download_CCS> et il s’agit de la version.   
Nous verrons différentes méthodes d’optimisation en 1ère partie suivi de l’optimisation Pipeline en 2ème partie.

1. **Méthodes d’optimisation**

Un programme qui marche n’est pas forcément le résultat final, il est souvent nécessaire d’optimiser son programme pour qu’il soit le plus performant possible. C’est le but de cette partie, on verra au total 4 programmes différents :

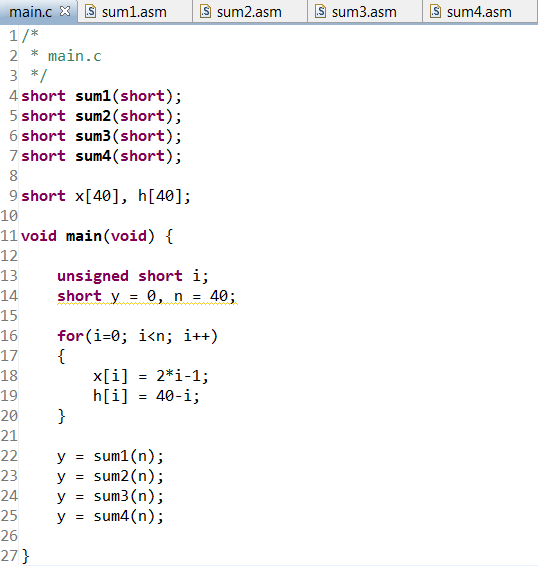
1. Non optimisé
2. Optimisation par parrallélisme  ||
3. Optimisation des NOPs
4. Optimisation World - Wide

Les temps de process seront ensuite analysés.

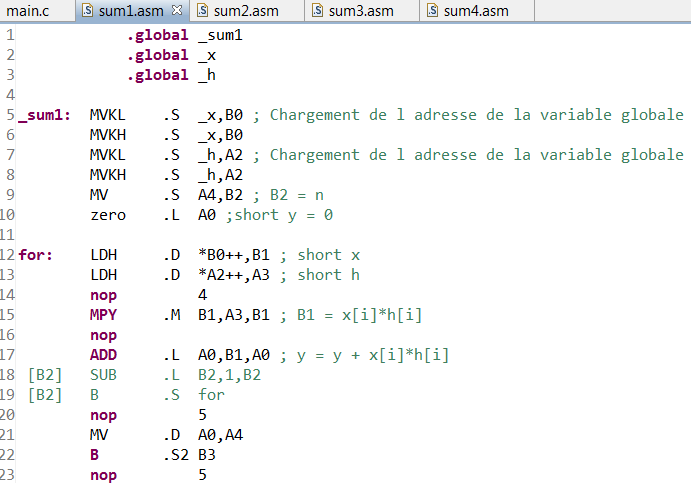
**I.1 Non optimisé :**

Il faut commencer par créer le « main », j’ai déclaré 2 tableaux *short* statiques et respectivement de taille 40, et ceci de façon globale. Ces tableaux sont initilialisés dans la boucle principale du programme.

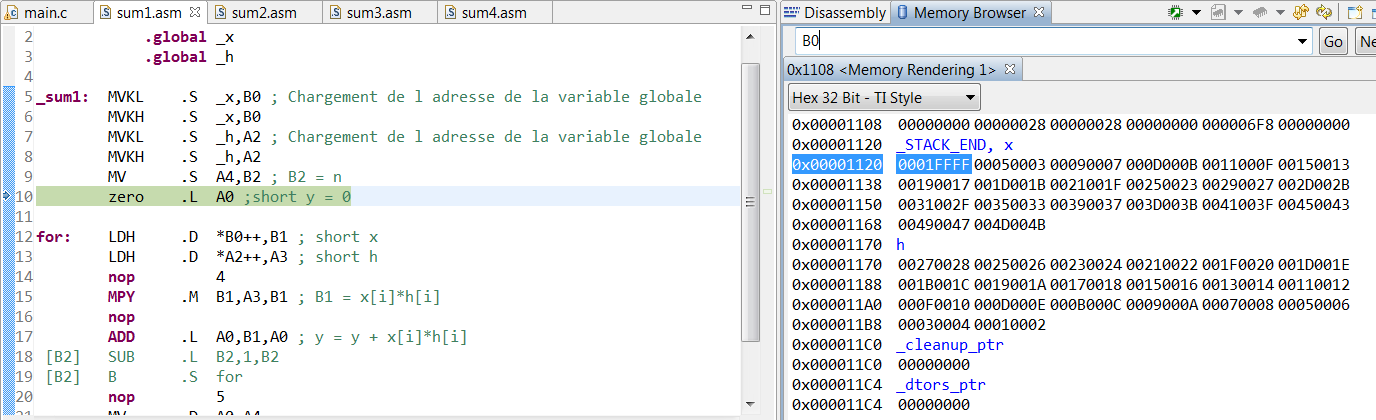
J’ai décidé pour la suite de ne faire rentrer qu’un seul paramètre dans les fonctions, *short* qui correspond à la taille des tableaux.



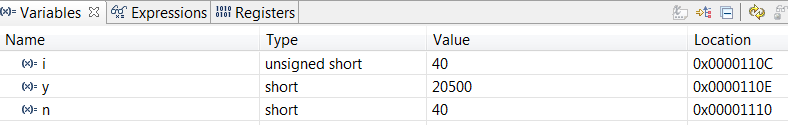
Pour accéder aux contenus des tableaux, j’utilise donc les variables globales. Là où cette méthode rajoute plus de cycles (comme nous le verrons par la suite), elle permet d’économiser du temps dans la transistion entre le *C*  et l’assembleur. En effet car on ne fait rentrer qu’un seul paramètre dans les fonctions.



En mode debug, j’ai commencé par vérifier les variables globales (les tableaux et ), ainsi que le paramètre (dans le registre A4).

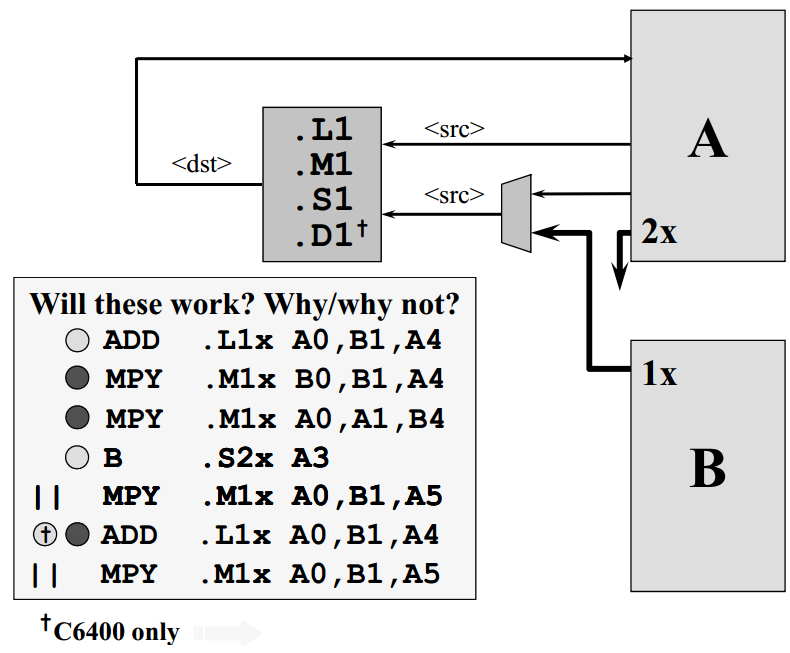
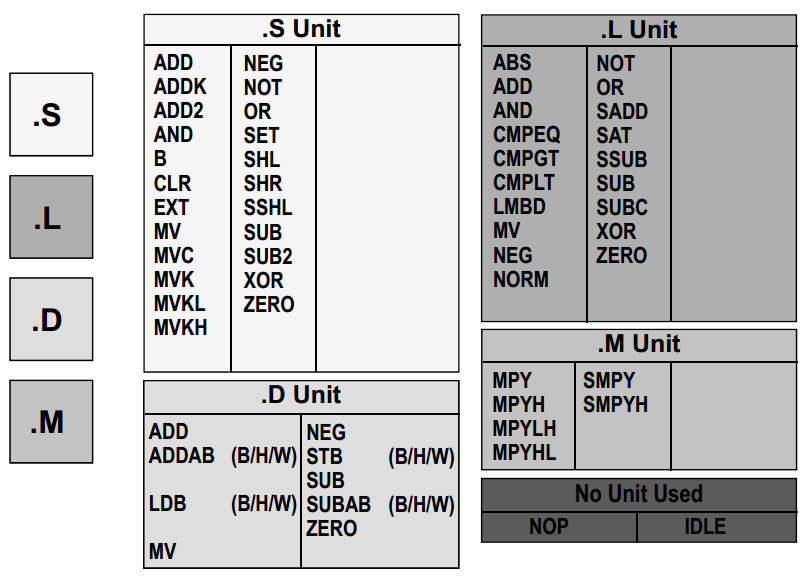


J’ai ensuite executé le programme pour vérifier la valeur de qui doit être égal à

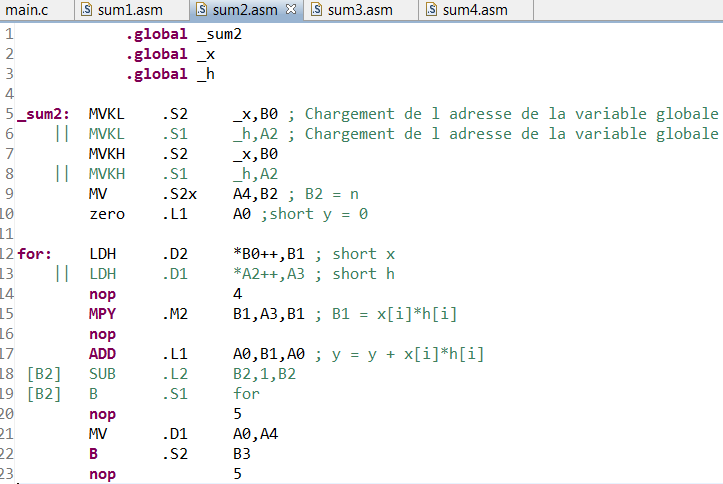


**I.2 Optimisation par parrallélisme**  **:**

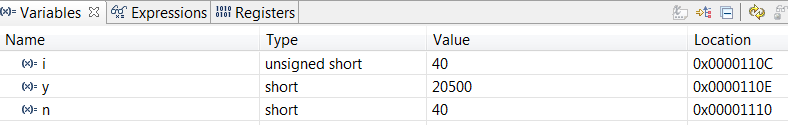
La parrallélisation permet de faire travailler plusieurs unités de calcul en même temps. Lorsque l’on veut parralléliser les tâches, il faut bien définir les unités ainsi que leur numéro. Rajouter un « x » à la fin des unités indique que l’on fait un cheminement croisé, les opérandes de registre passent d’un côté à un autre.



J’en ai profité pour mettre en parrallèle les *MVK* nécessaires pour le chargement des variables globales.



La procédure de debug est exactement la même que précedemment,

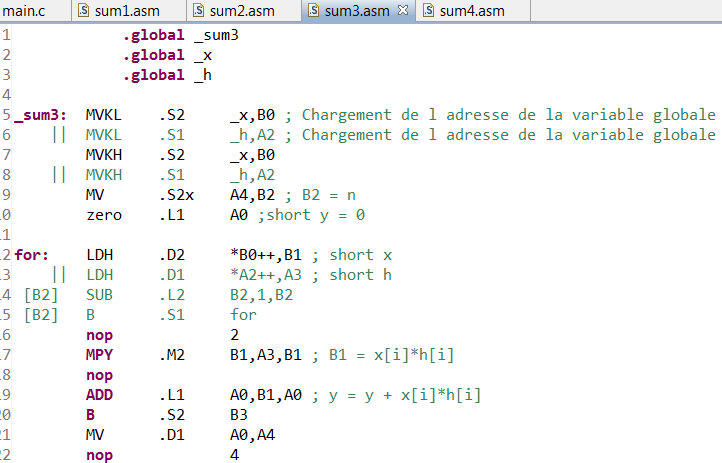


**I.3 Optimisation des NOPs**  **:**

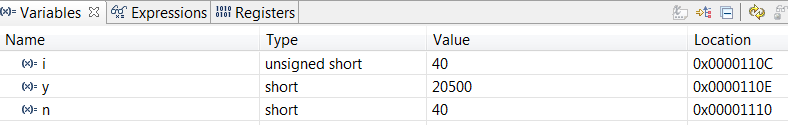
Pour cette partie, il faut bien analysé les temps de calcul de chaque instruction :

* Les opérations de calcul simples *zero, ADD* ou encore *SUB* prennent 1 cycle
* Les transferts simples *MV, MVKL* ou *MVKH* prennent 1 cycle
* Les opérations de multiplication terme à terme *MPY* prennent 2 cycles
* Les opérations de chargement *LD* prennent 5 cycles d’horloges du CPU
* Les branch *B* sont les plus coûteux et prennent 6 cyles

Le but est de réduire au minimum l’utilisation des NOP. On remarque par exemple que le temps que le branching se fasse, on a le temps d’executer des opérations.

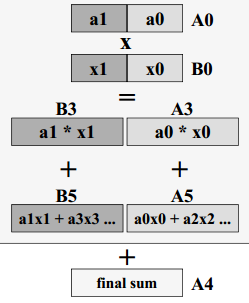


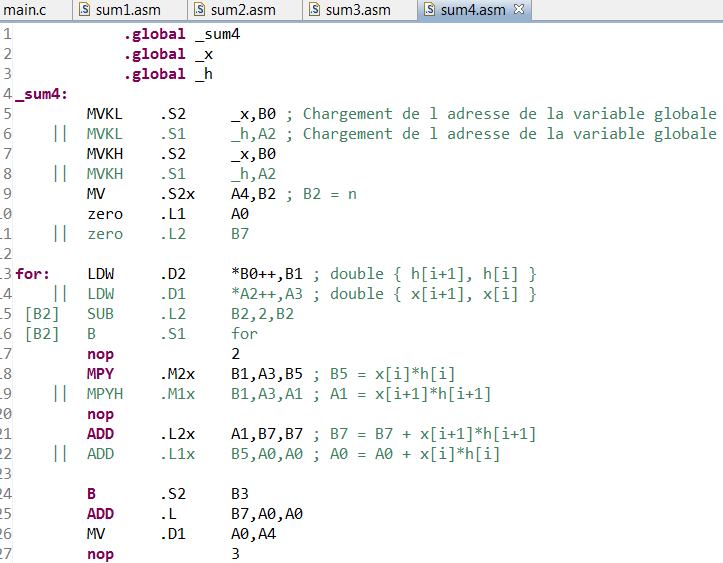
Avec la même procédure de debug,



**I.4 Optimisation World - Wide**  **:**

Dans cette optimisation, il est question de réduire le temps d’éxecution de la boucle principale. Pour cela on va utiliser des *LDW* au lieu de *LDH* qui permet de déplacer un mot de 32 bits en une seule fois. Voici l’idée générale de l’algorithme :

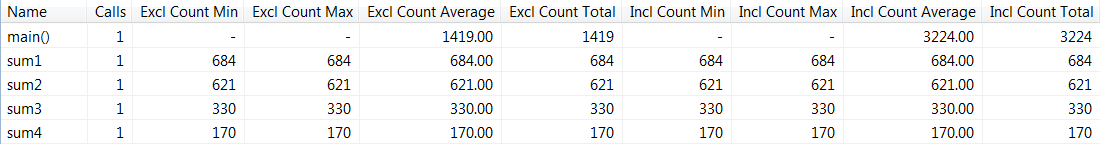


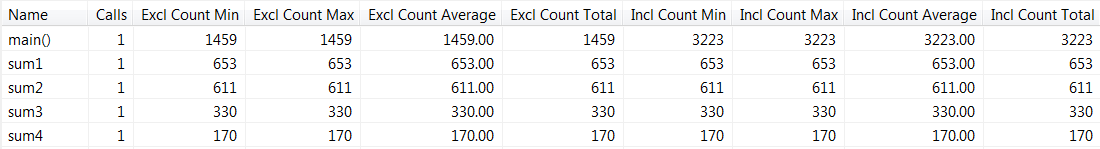


Au début j’ai déroulé la boucle pour économiser des ressources au niveau de la gestion de la boucle. Mais il s’est avéré que le résultat n’est pas si significatif que ça par apport à la taille du programme (qui commence à deveir très gros). De plus ce déroulement n’est vraiment pas pratique si l’on veut changer la taille de la boucle (il faut recopier le code à chaque fois..). Au final même en enlevant le branch *B*, les *LD* au début prennent du temps à se finir donc on ne gagne pas là-dessus. On gagne au total 2 cyles d’instructions par rapport à une optimisation world wide « normale » ce qui ne vaut vraiment pas le coup.

**I.5 Mesure de performances**

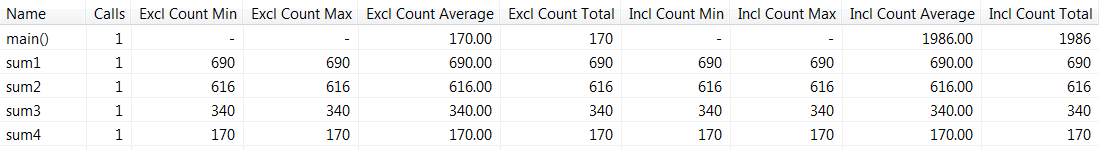
J’ai mesuré les performances en mode Debug et Release. Le premier taleau correspond à la mesure de performance lorsque l’on lance le programme pour la première fois :

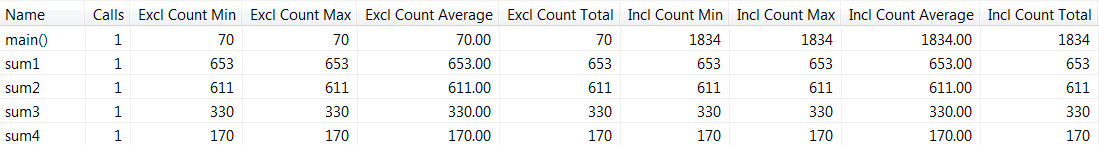




Ainsi l’optimisation World Wide permet de diviser par plus de 3 le nombre de cycles !

Mode Release





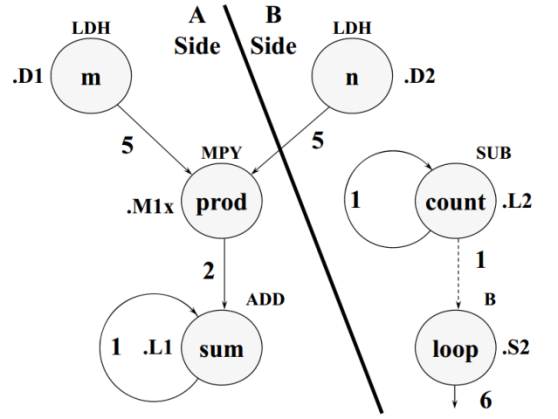
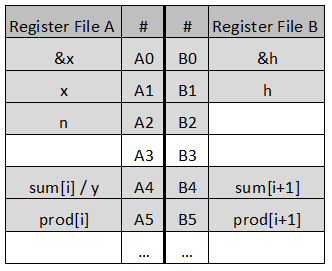
Le mode Release est surtout utile pour le language C, en effet en assembleur c’est en quelque sorte à nous de faire l’optimisation.

1. **Pipeline**

L’optimisation par Pipeline est une approche qui permet de maximiser le travail des unités fonctionnels, en implémentant une bonne parrallélisation des tâches et en remplissant les délais. Il y a plusieurs étapes pour faire du pipeline :

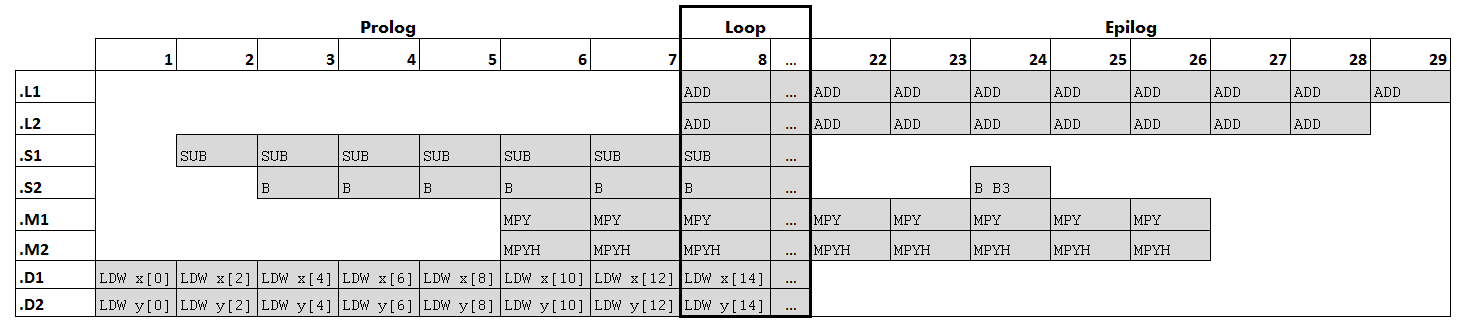
* Ecrire le programme en C
* Ecrire le programme en assembleur linéaire
* Créer le graphe de dépendance
* Allouer de manière optimale les registres
* Ecrire la table de temps
* Programmer

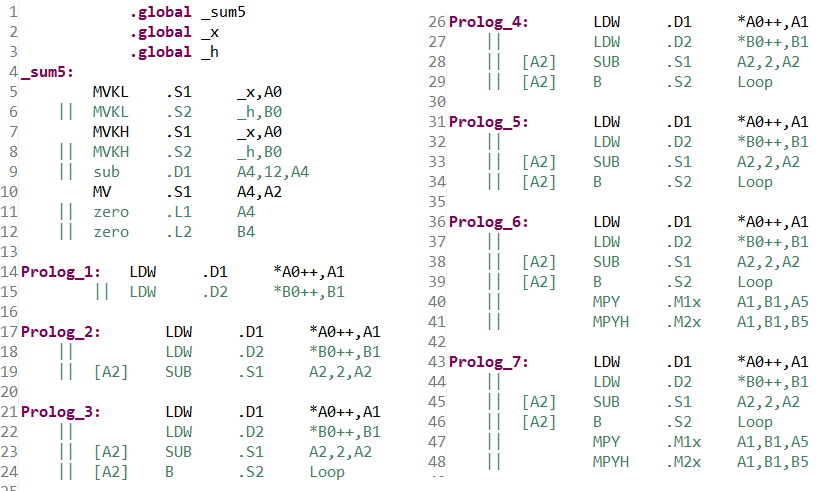
Après avoir créer le programme en C et en assembleur, il est nécessaire de créer ce que l’on appelle un « graphe de dépendance ». J’ai décidé d’utiliser des *LDW* au lieu de *LDH* pour diminuer le temps de cycle. **Si l’on se concentre sur la boucle**, le graphe et l’allocation des registres sont de la forme suivante :

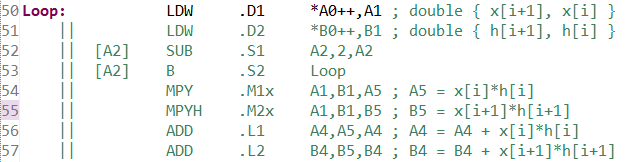
 

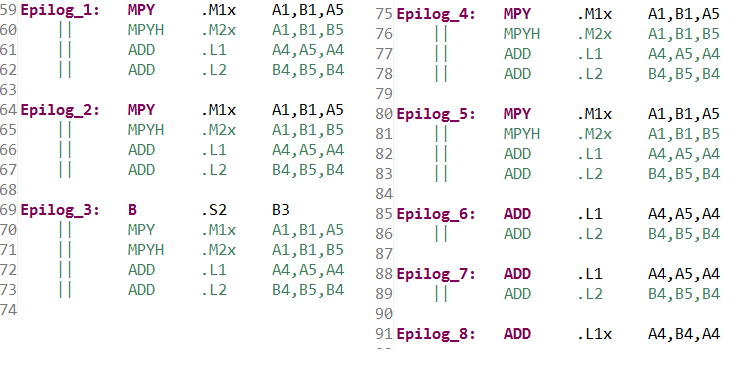
On peut découper le programme en 3 séquences le prologue (initialisation nécessaire pour que la boucle fonctionne), la boucle principale et l’épilogue (qui évite de faire des instructions inutiles en trop comme des *LDH*).

La taille du prologue est determiné par la plus longue séquence d’instructions (ici *LDW, MPY, ADD* pour 7 cycles). L’épilogue revient à terminer les calculs sans faire des *LDW* en trop dans la boucle qui seraient inutiles. Il ne faut pas oublier de soustraire pour avoir le bon nombre de boucles.



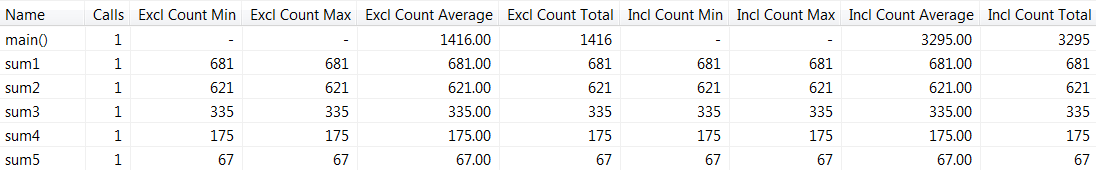


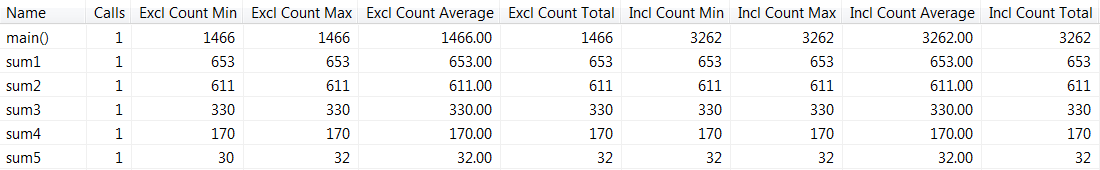




Pour débuger, j’ai utilisé le « Memory browser » et les informations des registres. Ceci pour vérifier quand est-ce que les registres sont chargés et calculés pour chaque itération. Voici les résultats du profiler en mode debug :

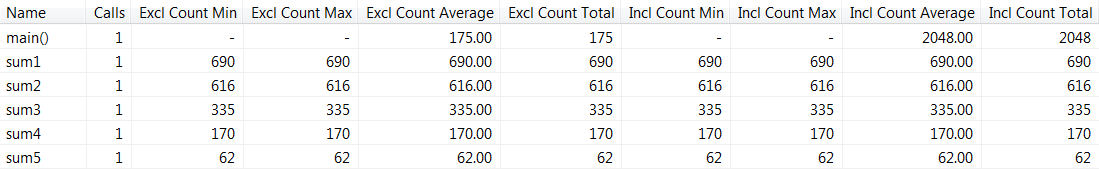
1ère fois

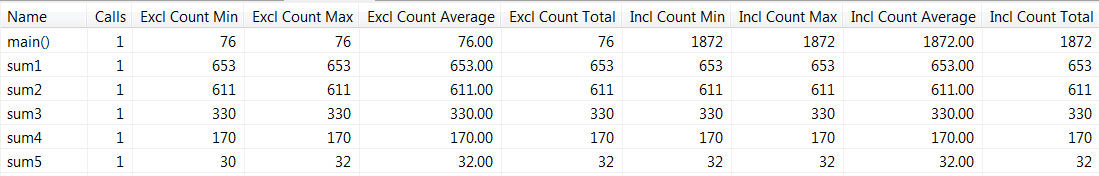




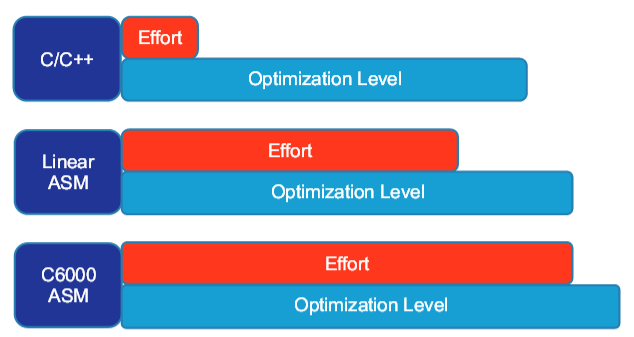
J’atteint un minimum de 30 boucles pour le calcul d’un produit scalaire, ce qui est un très bon résultat quand on compare au nombre de cycles en C du lab 2 (169 cycles de CPU en mode Release).

Release :





Pour conclure ces laboratoires, voici un petit schéma qui résume bien la programmation :



Ca a en effet été très rapide pour la programmation en C (quelques minutes) mais assez long pour l’optimisaton Pipeline qui nécessite beaucoup de rigueur (une à deux heures). A voir si les heures en plus sont justifiés par rapport au nombre de cycles.

Quelques minutes 🡺 169 cycles

1 à 2 heures 🡺 30 cycles