République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE IBN KHALDOUN – TIARET –

Faculté des Sciences Appliquées

EXTRAIT DU PROCES VERBAL

de la réunion du Conseil Scientifique de la Faculté

du 14/06/2020

L'an deux Mille Vingt et le Quatorze du Mois de Juin à 10H00, s'est tenue une réunion ordinaire du Conseil Scientifique de la Faculté des Sciences Appliquées de l'Université Ibn Khaldoun de Tiaret sous la présidence du Professeur **ABDICHE Ahmed.**

- Validation des Polycopiés :

Dans ce point d'ordre du jour « Validation des Polycopiés », le conseil scientifique de la faculté a validé la proposition du polycopié présenté par le docteur **SAHLI Belgacem,** de grade MC «A» au département Génie Electrique.
En effet, suite aux rapports des experts désignés par le Conseil scientifique de la faculté (CSF) réuni en date du 27/05/2020, le CSF a validé le polycopié présenté par le susnommé.

N°	Auteur du Polycopié	Titre du Polycopié	Experts	Avis du CSF
01	**SAHLI Belgacem**	Processeurs de Signaux Numériques (DSP)	**Dr BELARBI Mustapha** Université de Tiaret **Dr NASRI Djillali** Université de Tiaret	Avis Favorable

Le Doyen

Le Président du Conseil Scientifique

Ministère de l'Enseignement Supérieure et de la Recherche Scientifique
Université Ibn Khaldoun de Tiaret
Faculté des Sciences Appliquées - Département de Génie Electrique
Laboratoire de Génie Physique

POLYCOPIE : Processeurs de Signaux Numériques (DSP)

Destiné aux étudiants de 1ère année Master de la spécialité Electronique des Systèmes Embarqués.

Présenté par :

Dr. SAHLI Belgacem
Maître de conférences A

Année Universitaire : 2019 /2020

PREFACE

La famille de processeurs DSP TMS320C6000, telle qu'elle a été suggérée dans le programme du semestre deux de la première année master de la spécialité Electronique des Systèmes embarqués, a été introduite par Texas Instruments (TI) pour répondre aux exigences de haute performance dans les applications de traitement du signal. L'objectif de ce polycopié est de fournir les éléments pour la mise en œuvre et l'optimisation d'algorithmes de traitement de signal intensifs en calcul sur la famille de processeurs DSP (Digital Signal Processors) TMS320C6x.

Le polycopié est rédigé de sorte qu'il peut être utilisé comme manuel pour les cours de DSP pour être dispensés dans de nombreuses universités du territoire. Le manuel présenté est principalement écrit pour ceux qui sont déjà familiers avec les concepts DSP et qui sont intéressés par la conception de systèmes DSP basés sur les produits TI C6x DSP. Notez qu'une grande partie des informations contenues dans ce polycopié apparaissent dans les manuels TI de la famille C6000 DSP. Cependant, ces informations ont été restructurées, modifiées et condensées pour être utilisées pour l'enseignement d'un cours de DSP au courant du semestre.

En conséquence, le polycopié peut être utilisé comme un guide d'auto-apprentissage pour la mise en œuvre d'algorithmes sur les DSP C6x. Les chapitres sont organisés pour créer une corrélation étroite entre les sujets et les étudiants s'ils sont utilisés comme supports pour un cours de DSP. La connaissance du langage de programmation C est requise pour comprendre et exécuter les mécanismes DSP.

Table des matières

Chapitre 1 : Généralités sur les Processeurs DSP

1.1 **Introduction** :

En général, les capteurs génèrent des signaux analogiques en réponse à divers phénomènes physiques qui se produisent de manière analogique (c'est-à-dire en temps continu et en amplitude). Le traitement des signaux peut se faire dans le domaine analogique ou numérique. Pour effectuer le traitement d'un signal analogique dans le domaine numérique, il est nécessaire qu'un signal numérique soit formé par échantillonnage et quantification (numérisation) du signal analogique. Par conséquent, contrairement à un signal analogique, un signal numérique est discret en temps et en amplitude. Le processus de numérisation est réalisé via un convertisseur analogique-numérique (A/D).

Le traitement numérique du signal (DSP) implique la manipulation de signaux numériques afin d'en extraire des informations utiles. Bien qu'une quantité croissante de traitement du signal soit effectuée dans le domaine numérique, il reste le besoin de s'interfacer avec le monde analogique dans lequel nous vivons. Les convertisseurs de données analogiques-numériques (A/D) et numériques-analogiques (D/A) sont les dispositifs qui permettent cette interface. La figure 1-1 illustre les principaux composants d'un système DSP, comprenant des périphériques A/D, DSP et D/A.

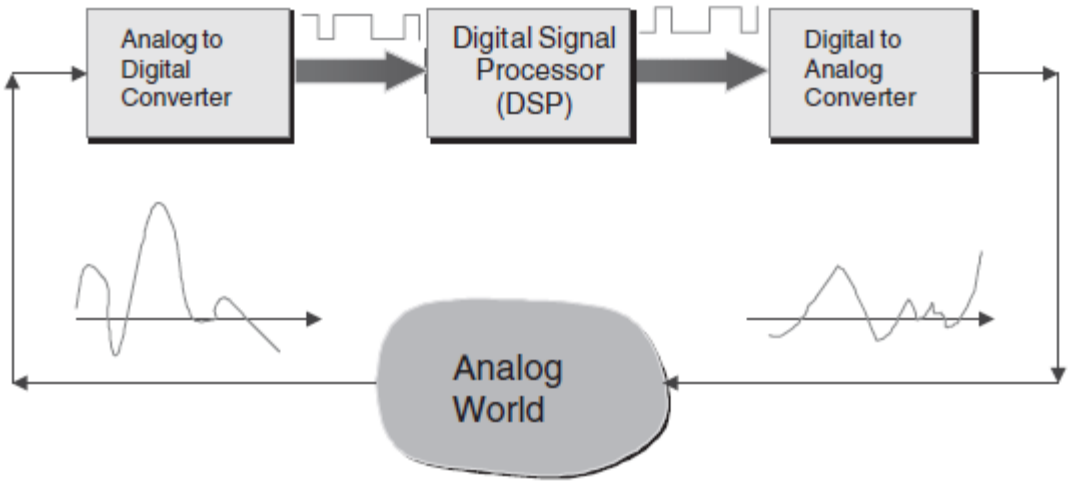


Fig. 1.1 : Composants principaux d'un système DSP

Il existe de nombreuses raisons pour lesquelles on voudrait traiter un signal analogique de manière numérique en le convertissant en un signal numérique. La raison principale est que le traitement numérique permet la programmabilité. Le même matériel DSP peut être utilisé pour de nombreuses applications différentes en changeant simplement le code résidant en mémoire. Une autre raison est que les circuits numériques fournissent une sortie plus stable et tolérante que les circuits analogiques, par exemple lorsqu'ils sont soumis à des changements de température. De plus, l'avantage de fonctionner dans le domaine numérique peut être intrinsèque. Par exemple, un filtre de phase linéaire ou un filtre coupe-bande à coupure abrupte ne peut être réalisé qu'en utilisant des techniques de traitement du signal numérique, et de nombreux systèmes adaptatifs ne sont réalisables dans un produit pratique que par la manipulation numérique des signaux. En substance, la représentation numérique (0 et 1) permet aux données vocales, audio, image et vidéo d'être traitées de la même manière à des fins de transmission et de stockage numériques tolérantes aux erreurs. Par conséquent, le traitement numérique, et donc les processeurs de signaux numériques (également appelés DSP), devraient jouer un rôle majeur dans la prochaine génération d'infrastructures de télécommunications, y compris les lignes sans fil 3 et 4G (troisième et

quatrième générations), les câbles (modems câble) et les lignes téléphoniques (ligne d'abonné numérique - modems DSL).

Le traitement d'un signal numérique peut être mis en œuvre sur diverses plates-formes telles qu'un processeur DSP, un circuit intégré à très grande échelle personnalisé (VLSI) ou un microprocesseur à usage général. Certaines des différences entre un DSP et une implémentation VLSI à fonction unique sont les suivantes :

1. La mise en œuvre du DSP offre une grande souplesse d'application, car le même matériel DSP peut être utilisé pour différentes applications. En d'autres termes, les processeurs DSP sont programmables. Ce n'est pas le cas pour un circuit numérique câblé.

2. Les processeurs DSP sont économiques car ils sont fabriqués en série et peuvent être utilisés pour de nombreuses applications. Une puce VLSI personnalisée est normalement conçue pour une seule application et un client spécifique.

3. Dans de nombreuses situations, les nouvelles fonctionnalités constituent une mise à niveau logicielle sur un processeur DSP ne nécessitant pas de nouveau matériel. De plus, les corrections de bogues sont généralement plus faciles à effectuer.

4. Souvent des taux d'échantillonnage très élevés peuvent être atteints par une puce personnalisée, alors qu'il existe des limitations de taux d'échantillonnage associées aux puces DSP en raison de leurs contraintes périphériques et de la conception de l'architecture.

Les processeurs DSP partagent certaines caractéristiques communes qui les séparent également des microprocesseurs à usage général. Certaines de ces caractéristiques sont les suivantes :

Ils sont optimisés pour faire face à la répétition ou au bouclage d'opérations courantes dans les algorithmes de traitement du signal. Relativement parlant, les jeux d'instructions des DSP sont plus petits et optimisés pour les opérations de traitement du signal, telles que la multiplication et l'accumulation en un seul cycle.

2. Les DSP permettent des modes d'adressage spécialisés, comme l'adressage indirect et circulaire. Ce sont des mécanismes d'adressage efficaces pour implémenter de nombreux algorithmes de traitement du signal.

3. Les DSP possèdent des périphériques appropriés qui permettent une interface d'entrée / sortie (E / S) efficace avec d'autres périphériques.

4. Dans les processeurs DSP, il est possible d'effectuer plusieurs accès à la mémoire en un seul cycle d'instructions. En d'autres termes, ces processeurs ont une bande passante relativement élevée entre leurs unités centrales de traitement (CPU) et la mémoire.

La majeure partie de la part de marché des DSP appartient aux systèmes intégrés en temps réel et rentables, par exemple les téléphones cellulaires, les modems et les lecteurs de disque. En temps réel signifie terminer le traitement dans le temps autorisé ou disponible entre les échantillons. Ce temps disponible, bien sûr, dépend de l'application. Comme illustré sur la figure 1-2, le nombre d'instructions pour qu'un algorithme s'exécute en temps réel doit être inférieur au nombre d'instructions pouvant être exécutées entre deux échantillons consécutifs.

Par exemple, pour un traitement audio fonctionnant à une fréquence d'échantillonnage de 44,1 kHz, soit un intervalle d'échantillonnage d'environ 22,6 µs, le nombre d'instructions doit être inférieur à près de 4500, en supposant un temps de cycle d'instructions de 5 ns. Le

traitement en temps réel comporte deux aspects: a) la fréquence d'échantillonnage et b) les latences (retards) du système. Les taux d'échantillonnage et les latences typiques pour plusieurs applications différentes sont indiqués dans le Tableau 1-1.

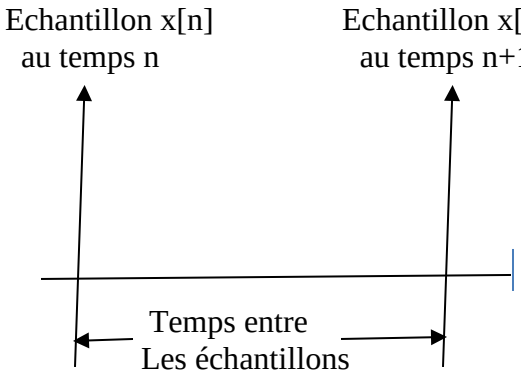


Fig. 1.2 : Nombre maximal d'instructions à respecter en temps réel = temps entre échantillons / temps de cycle d'instruction.

Tableau 1.1 : Taux d'échantillonnage et latences typiques pour certaines applications :

Application	Taux d'échantillonnage E/S	Latence
Instrumentation	1 Hz	*Dépend du système
Commande	> 0.1 KHz	*Dépend du système
Voix	8 KHz	< 50 ms
Audio	44.1 KHz	*< 50 ms
Vidéo	1 – 14 MHz	*< 50 ms

* Dans de nombreux cas, on n'a pas à se préoccuper de la latence, par exemple, un signal TV dépend plus de la synchronisation avec l'audio que de la latence. Dans chacun de ces cas, la latence dépend de l'application.

1.2 Exemples de systèmes DSP

Pour que le lecteur puisse apprécier l'utilité des DSP, un exemple de systèmes DSP actuellement utilisé est présenté ici.

Au cours des dernières années, le marché du sans-fil a connu une croissance considérable. La figure 1-3 illustre un système DSP de communication sans fil pour téléphone cellulaire.

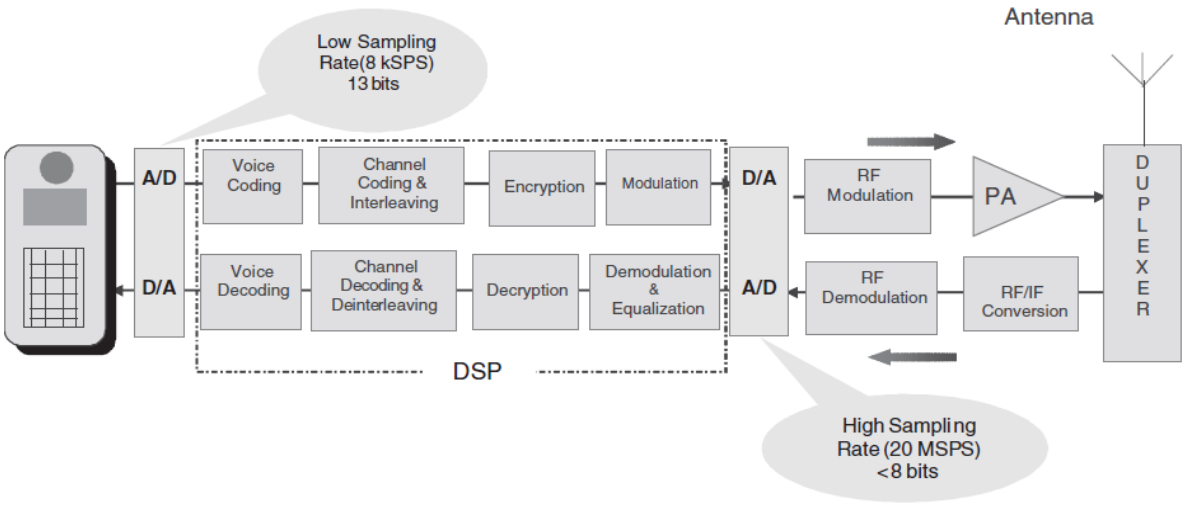


Fig. 1.3 : Système DSP de communication sans-fil pour téléphone cellulaire

Comme le montre cette figure, il existe deux ensembles de convertisseurs de données. Du côté de la bande vocale, un faible taux d'échantillonnage (par exemple, 8 KSPS [kilo d'échantillons par seconde]) et un convertisseur haute résolution (par exemple, 13 bits) sont utilisés, tandis que du côté de la modulation RF, un débit relativement élevé (par exemple, 20 MSPS) et un convertisseur basse résolution (par exemple, 8 bits) est utilisé. Les concepteurs de systèmes préfèrent intégrer davantage de fonctionnalités dans les DSP plutôt que dans des composants analogiques afin de réduire le nombre de composants et donc le coût global. Cette stratégie d'intégration accrue dans les DSP dépend de spécifications réalisables pour une faible consommation d'énergie dans les appareils portables.

Chapitre 2 : Arithmétique à virgule fixe et à virgule flottante

2.1 Virgule fixe ou flottante

Une caractéristique importante qui distingue les différents processeurs DSP est de savoir si leurs processeurs effectuent une arithmétique à virgule fixe ou à virgule flottante. Dans un processeur à virgule fixe, les nombres sont représentés et manipulés au format entier. Dans un processeur à virgule flottante, en plus de l'arithmétique entière, l'arithmétique à virgule flottante peut être gérée. Cela signifie que les nombres sont représentés par la combinaison d'une mantisse (ou d'une partie fractionnaire) et d'une partie exposante, et le CPU possède le matériel nécessaire pour manipuler ces deux parties. En conséquence, en général, les processeurs à virgule flottante sont plus chers et plus lents que les processeurs à virgule fixe.

Dans un processeur à virgule fixe, il faut se préoccuper de la plage dynamique des nombres, car une plage de nombres beaucoup plus étroite peut être représentée au format entier par rapport au format à virgule flottante. Pour la plupart des applications, une telle préoccupation peut être pratiquement ignorée lors de l'utilisation d'un processeur à virgule flottante. Par conséquent, les processeurs à virgule fixe exigent généralement plus d'efforts de codage que leurs homologues à virgule flottante.

2.2 Représentation des nombres au format Q sur les DSP à virgule fixe

La valeur décimale d'un nombre complément à 2 $B = b_{N-1}b_{N-2} \dots b_1b_0$, $b_i \in \{0, 1\}$,est donné par :

$$D(B) = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \dots + b_12^1 + b_02^0 \qquad (2.1)$$

La représentation du complément à 2 permet à un processeur d'effectuer l'addition et la soustraction d'entiers en utilisant le même matériel. Lorsque vous utilisez une représentation entière non signée, le bit de signe est traité comme un bit supplémentaire. De cette façon, seuls les nombres positifs peuvent être représentés.

Il y a une limitation à la plage dynamique du schéma de représentation entier précédent. Par exemple, dans un système à 16 bits, il n'est pas possible de représenter des nombres supérieurs à $+2^{15} - 1 = 32767$ et inférieurs à $-2^{15} = -32768$. Pour faire face à cette limitation, les nombres sont normalisés entre –1 et 1. En d'autres mots, ils sont représentés comme des fractions. Cette normalisation est réalisée par le programmeur en déplaçant le point binaire implicite ou imaginaire (notez qu'il n'y a pas de mémoire physique allouée à ce point) comme indiqué sur la figure 6-1. De cette façon, la valeur fractionnaire est donnée par :

$$F(B) = -b_{N-1}2^0 + b_{N-2}2^1 + \ldots + b_1 2^{-(N-2)} + b_0 2^{-(N-1)}$$

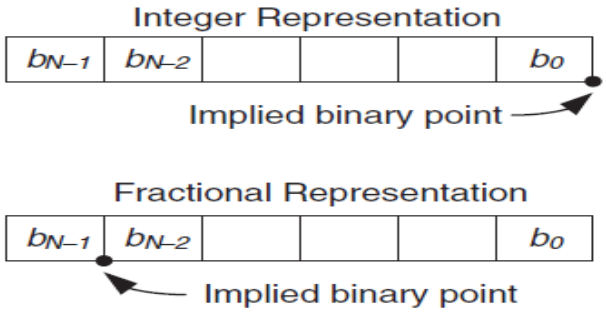


Fig. 2.1: Représentation des nombres

Ce schéma de représentation est appelé format Q ou représentation fractionnée.

Le programmeur doit garder une trace du point binaire implicite lors de la manipulation des nombres au format Q. Par exemple, considérons deux nombres au format Q-15, étant donné que nous avons une mémoire large de 16 bits. Chaque nombre se compose d'un bit de signe plus 15 bits fractionnaires. Lorsque ces nombres sont multipliés, un nombre au

format Q-30 est obtenu (le produit de deux fractions est toujours une fraction), le bit 31 étant le bit de signe et le bit 32 un autre bit de signe (appelé bit de signe étendu). S'il n'y a pas suffisamment de bits disponibles pour stocker les 32 bits et que seuls 16 bits peuvent être stockés, il est logique de stocker les bits les plus significatifs. Cela se traduit par le stockage de la partie supérieure du registre de produit 32 bits en effectuant un décalage à droite (SHR) de 15 bits. De cette manière, le produit serait stocké au format Q-15. (Voir figure 2.2.)

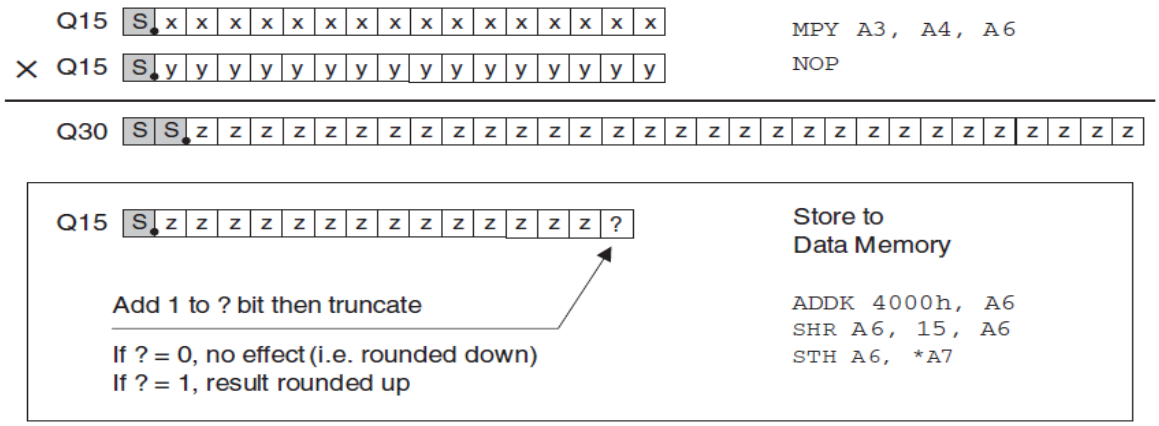


Fig. 2.2 : Multiplication et stockage des nombres Q-15.

Sur la base de la représentation du complément à 2, une plage dynamique de - $(2^{N-1}) \leq D(B) < 2^{N-1} - 1$ peut être obtenue, où N désigne le nombre de bits. À des fins d'illustration, considérons un système à 4 bits où le nombre le plus négatif est –8 et le nombre le plus positif 7. Les représentations décimales des nombres sont illustrées à la figure 2.3. Remarquez comment les nombres changent du plus positif au plus négatif avec le bit de signe.

Étant donné que seuls les nombres entiers compris dans les limites –8 et 7 peuvent être représentés, il est facile de voir que toute multiplication ou addition entraînant un nombre supérieur à 7 ou inférieur à –8 entraînera un débordement. Par exemple, lorsque 6 est

multiplié par 2, nous obtenons 12. Par conséquent, le résultat est supérieur aux limites de représentation et sera enroulé autour de 1100, soit –4.

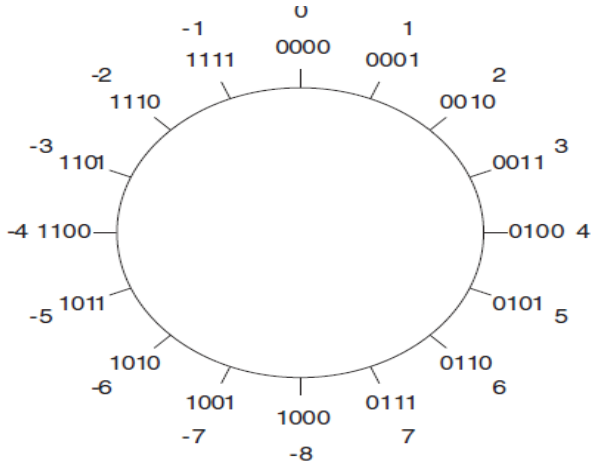


Fig. 2.3 : Représentation binaire à 4 bits

La représentation au format Q résout ce problème en normalisant la plage dynamique entre –1 et 1. Toute multiplication résultante sera dans les limites de cette plage dynamique. En utilisant une représentation au format Q, la plage dynamique est divisée en 2^N sections, où $2^{-(N-1)}$ est la taille d'une section. Le nombre le plus négatif est toujours –1 et le nombre le plus positif est $1 - 2^{-(N-1)}$.

L'exemple suivant permet de voir la différence entre les deux schémas de représentation. Comme le montre la figure 2.4, la multiplication de 0110 par 1110 en binaire équivaut à multiplier 6 par –2 en décimal, ce qui donne un résultat de –12, un nombre dépassant la plage dynamique du système à 4 bits. Sur la base de la représentation Q-3, ces chiffres correspondent respectivement à 0,75 et –0,25. Le résultat est –0,1875, ce qui se situe dans la plage fractionnaire. Notez que le matériel génère les mêmes 1 et 0, ce qui est différent est l'interprétation des bits.

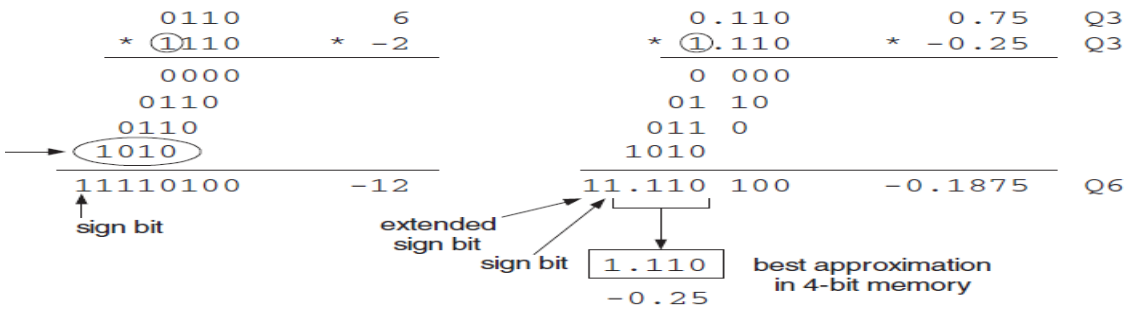


Fig. 2.4 : Multiplication binaire et fractionnelle

Lors de la multiplication des nombres Q-N, il ne faut pas oublier que le résultat consistera en 2N bits fractionnaires, un bit de signe et un ou plusieurs bits de signe étendus. En fonction du type de données utilisé, le résultat doit être décalé en conséquence. Si deux nombres Q-15 sont multipliés, le résultat aura une largeur de 32 bits, le MSB étant le bit de signe étendu suivi du bit de signe. Le point décimal imaginaire sera après le 30ème bit.

Un décalage à droite de 15 est donc nécessaire pour stocker le résultat dans un emplacement de mémoire de 16 bits en tant que nombre Q-15. Il doit être réalisé qu'une certaine précision est perdue, bien sûr, à la suite de l'élimination des bits fractionnaires plus petits. Comme seuls 16 bits peuvent être stockés, le décalage permet de conserver les bits fractionnaires de plus haute précision. Si une capacité de stockage 32 bits est disponible, un décalage vers la gauche de 1 peut être effectué pour supprimer le bit de signe étendu et stocker le résultat sous le format d'un numéro Q-31.

Pour mieux comprendre une éventuelle perte de précision lors de la manipulation de nombres au format Q, considérons un autre exemple où deux nombres Q12 correspondant à 7,5 et 7,25 sont multipliés. Comme le montre la figure 2.5, le produit résultant doit être

décalé de 4 bits vers la gauche pour stocker tous les bits fractionnaires correspondant au format Q12.

Cependant, cela entraîne une valeur de produit de 6,375, ce qui est différent de la valeur correcte de 54,375. Si le produit est stocké dans un format Q de moindre précision, par exemple au format Q8, la valeur correcte du produit peut être stockée.

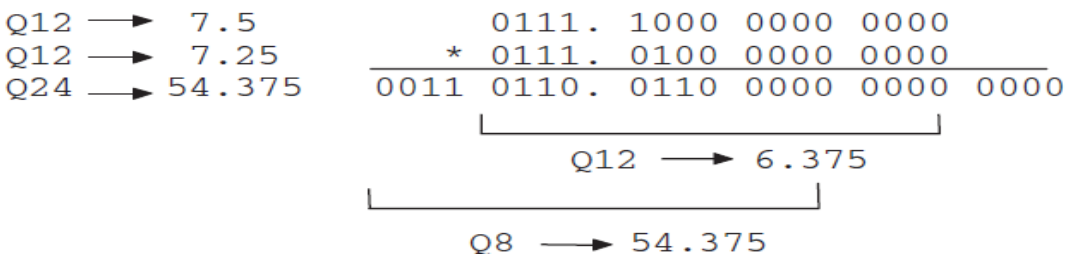


Fig. 2.5 : Exemple de perte de précision dans le format Q

Bien que le format Q résout le problème du débordement dans la multiplication, l'addition et la soustraction posent toujours un problème. Lors de l'ajout de deux nombres Q15, la somme dépasse la plage de représentation Q15. Pour résoudre ce problème, l'approche de mise à l'échelle, discutée plus tard, doit être utilisée.

2.3 Représentation des nombres en virgule flottante

La norme IEEE 754 pour la représentation des nombres à virgule flottante est la norme la plus couramment utilisée dans les processeurs DSP. Elle utilise un format simple précision avec 32 bits et un format double précision avec 64 bits.

Le standard IEEE 754 (**I**nstitute of **E**lectronic and **E**lectrical **E**ngineers) de simple précision est le suivant : le premier bit (*s*) est le signe du nombre ; suivent ensuite 8 bits représentant *e*, l'exposant en code excédant 127. Dans ce code, 127 représente 0, tout

nombre inférieur à 127 est considéré comme négatif, et tout nombre entre 127 et 255 est positif. On peut ainsi exprimer des exposants allant de -126 (00000001) à +127 (11111110). Les exposants -127 et +128 sont réservés aux représentations de 0 et de ∞ respectivement. Les 23 bits suivants représentent la partie fractionnaire *f* de la mantisse normalisée. Il s'agit d'une mantisse normalisée *m* de la forme

$$m = 1,f \text{ telle que } 1 \leq m < 2. \qquad (2.2)$$

Comme le premier bit d'une telle mantisse normalisée est toujours 1, on n'a pas besoin de l'écrire, ce qui permet de gagner un bit et d'atteindre une précision de 24 bits. Cette représentation nécessite donc un total de 32 bits, soit 4 octets. Ceci représente une précision décimale de l'ordre de 7 chiffres significatifs et peut exprimer des nombres allant d'environ 10^{-38} à 10^{+38}.

La valeur du nombre est donc donnée par l'expression :

$$N = (-1)^s \times 2^{(e - 127)} \times 1,f \qquad (2.3)$$

où *s* est le signe de la mantisse, et *e* est l'exposant tel qu'inscrit dans le format suivant :

0	100 0011 1	101 1000 0000 0000 0000 0000
signe	*e* = exposant	partie fractionnaire *f*
s	excédant 127	de la mantisse normalisée
(+)	= 135	
	e - 127 = 8	

Fig. 2.6 : Représentation de 432 en virgule flottante IEEE de simple précision

Le nombre 432 = $(110110000)_2$ = $1,10110000 \times 2^8$ s'écrit donc en simple précision : $43D80000_{IEEE}$. On utilise ici la notation hexadécimale comme abréviation de la

représentation binaire du nombre IEEE. On met l'indice IEEE pour bien indiquer qu'il ne s'agit pas d'un nombre hexadécimal.

Exemples :

$+0 = 00000000_{IEEE}$,

$-0 = 80000000_{IEEE}$,

$+1 = 1{,}0 \times 2^0 = 3F800000_{IEEE}$,

$+2 = 1{,}0 \times 2^1 = 40000000_{IEEE}$,

$+\infty = 1{,}0 \times 2^{128} = 7F800000_{IEEE}$ et

$-\infty = FF800000_{IEEE}$.

En double précision, on utilise 8 octets, soit 64 bits, dont 11 servent à l'exposant (en code excédant 1023) et 52 à la partie fractionnaire de la mantisse. Il en résulte une précision équivalant à 15 chiffres significatifs en décimal et des exposants pouvant aller de 10^{-308} à 10^{+307}.

La valeur d'un nombre *N* en double précision est donc donnée par l'expression :

$$N = (-1)^s \times 2^{(e-1023)} \times 1,f \qquad (2.4)$$

En précision étendue, on utilise 10 octets, soit 80 bits, dont 15 servent à l'exposant (en code excédant 16 384) et 63 à la partie fractionnaire de la mantisse. Il en résulte une précision équivalant à 19 chiffres significatifs en décimal et des exposants pouvant aller de 10^{-4933} à 10^{+4931}.

La valeur d'un nombre *N* en précision étendue est donc donnée par l'expression :

$$N = (-1)^s \times 2^{(e-16383)} \times 1,f \qquad (2.5)$$

2.4 Échantillonnage

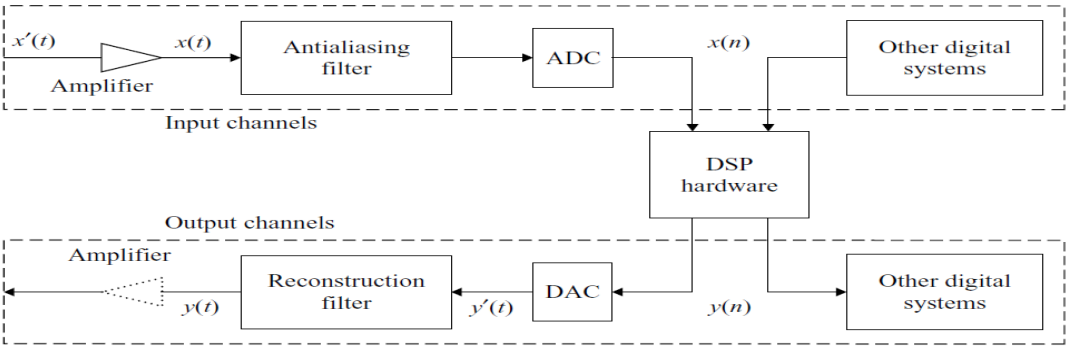


Fig. 2.7 : Schéma fonctionnel de base d'un système DSP en temps réel

Comme le montre la figure 2.7, le CAN convertit le signal analogique x (t) en signal numérique x (n). La conversion analogique-numérique, communément appelée numérisation, comprend les processus d'échantillonnage (numérisation en temps) et de quantification (numérisation en amplitude) comme illustré à la figure 2.8. Le processus d'échantillonnage représente un signal analogique comme une séquence de valeurs. La fonction d'échantillonnage de base peut être exécutée avec un circuit « échantillonnage et maintien » idéal, qui maintient le niveau du signal échantillonné jusqu'à ce que l'échantillon suivant soit prélevé.

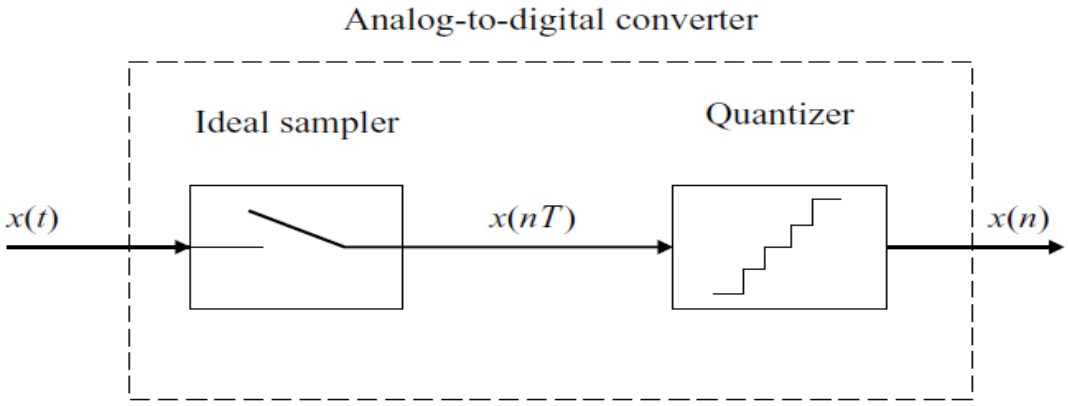


Fig. 2.8 : Schéma fonctionnel d'ADC

Le processus de quantification se rapproche d'une forme d'onde en attribuant un numéro à chaque échantillon. Par conséquent, la conversion analogique-numérique effectuera les étapes suivantes:

1. Le signal à bande limitée x (t) est échantillonné à des instants uniformément espacés du temps nT, où n est un entier positif et T est la période d'échantillonnage en secondes. Ce processus d'échantillonnage convertit un signal analogique en un signal à temps discret x (nT) avec une valeur d'amplitude continue.

2. L'amplitude de chaque échantillon à temps discret est quantifiée dans l'un des 2 niveaux B, où B est le nombre de bits que l'ADC doit représenter pour chaque échantillon. Les niveaux d'amplitude discrets sont représentés (ou codés) en mots binaires distincts x (n) avec une longueur de mot fixe B.

La raison de cette distinction est que ces processus introduisent des distorsions différentes. Le processus d'échantillonnage entraîne un aliasing ou une distorsion de pliage, tandis que le processus de codage entraîne un bruit de quantification. Comme le montre la figure 2.2, l'échantillonneur et le quantificateur sont intégrés sur la même puce. Cependant, les ADC haute vitesse nécessitent généralement un dispositif d'échantillonnage et de maintien externe.

Un échantillonneur idéal peut être considéré comme un interrupteur qui s'ouvre et se ferme périodiquement toutes les Ts (secondes).

La période d'échantillonnage est définie comme :

$$T = \frac{1}{f_S} \qquad (2.6$$

où f_S est la fréquence d'échantillonnage (ou taux d'échantillonnage) en hertz (ou cycles par seconde). Le signal intermédiaire x (nT) est un signal à temps discret avec une valeur continue (un nombre avec une précision infinie) à un temps discret nT, n = 0, 1,. . . , ∞, comme illustré à la figure 2.9. Le signal analogique x (t) est continu en temps et en amplitude. Le signal à temps discret échantillonné x (nT) est continu en amplitude, mais n'est défini qu'à des instants d'échantillonnage discrets t = nT.

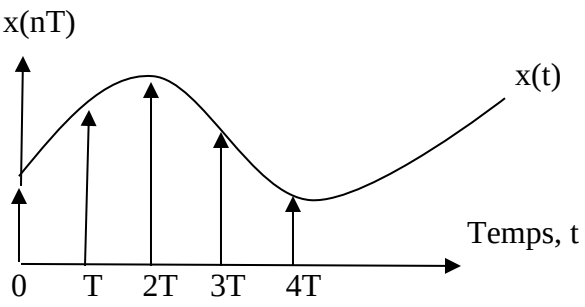


Fig. 2.9 : Exemple de signal analogique x (t) et de signal à temps discret x (nT)

Afin de représenter avec précision un signal analogique x (t) par un signal à temps discret x (nT), la fréquence d'échantillonnage f_S doit être au moins le double de la composante de fréquence maximale (f_M) dans le signal analogique x (t).

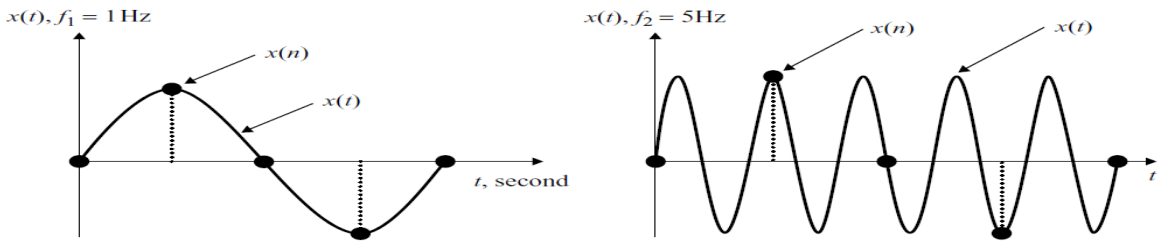
C'est,

$$f_S \geq 2f_M$$

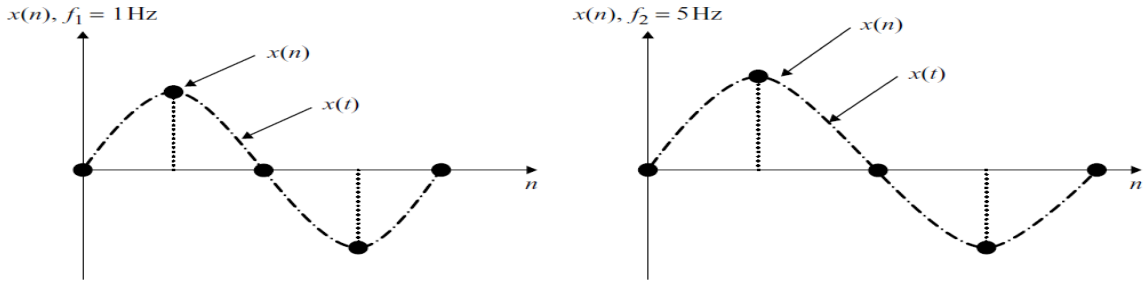
où f_M est également appelé largeur de bande du signal x (t). Il s'agit du théorème d'échantillonnage de Shannon, qui stipule que lorsque la fréquence d'échantillonnage est supérieure à deux fois la composante de fréquence la plus élevée contenue dans le signal analogique, le signal d'origine x (t) peut être parfaitement reconstruit à partir du signal à temps discret correspondant x (nT).

Le taux d'échantillonnage minimal $f_S = 2\ f_M$ est appelé taux de Nyquist. La fréquence $f_N = f_S / 2$ est appelée fréquence de Nyquist ou fréquence de repliement. L'intervalle de fréquence $[- f_S/2, f_S/2]$ est appelé intervalle de Nyquist. Lorsqu'un signal analogique est échantillonné à f_S, les composantes de fréquence supérieures à $f_S/2$ se replient dans la plage de fréquences $[0, f_S/2]$. Les composantes de fréquence repliées se chevauchent avec les composantes de fréquence d'origine dans la même gamme. Par conséquent, le signal analogique d'origine ne peut pas être récupéré à partir des données échantillonnées. Cet effet indésirable est appelé aliasing (repliement).

Exemple 1: Considérons deux ondes sinusoïdales de fréquences $f_1 = 1$ Hz et $f_2 = 5$ Hz qui sont échantillonnées à $f_S = 4$ Hz, plutôt qu'à 10 Hz selon le théorème d'échantillonnage. Les formes d'onde analogiques sont illustrées à la figure 2.10 (a), tandis que leurs échantillons numériques et formes d'onde reconstruites sont illustrés dans la figure 2.4 (b). Comme le montrent les figures, nous pouvons reconstruire la forme d'onde d'origine à partir des échantillons numériques pour l'onde sinusoïdale de fréquence $f_1 = 1$ Hz. Cependant, pour l'onde sinusoïdale d'origine de fréquence $f_2 = 5$ Hz, le signal reconstruit est identique à l'onde sinusoïdale de fréquence 1 Hz. Par conséquent, f_1 et f_2 sont censés être liés l'un à l'autre, c'est-à-dire qu'ils ne peuvent pas être distingués par leurs échantillons à temps discret...



(a) Formes d'onde analogiques originales et échantillons numériques pour $f_1 = 1$ Hz et $f_2 = 5$ Hz.



(b) Échantillons numériques pour $f_1 = 1$ Hz et $f_2 = 5$ Hz et formes d'onde reconstruites.

Fig. 2.10 : Exemple du phénomène de repliement: (a) formes d'onde analogiques originales et échantillons numériques pour $f_1 = 1$ Hz et $f_2 = 5$ Hz; (b) échantillons numériques de $f_1 = 1$ Hz et $f_2 = 5$ Hz et formes d'onde reconstruites dans la figure 2.10 (b).

Comme le montrent les figures, nous pouvons reconstruire la forme d'onde d'origine à partir des échantillons numériques pour l'onde sinusoïdale de fréquence $f_1 = 1$ Hz. Cependant, pour l'onde sinusoïdale d'origine de fréquence $f_2 = 5$ Hz, le signal reconstruit est identique à l'onde sinusoïdale de fréquence 1 Hz. Par conséquent, f_1 et f_2 sont dits repliés (aliased) l'un sur l'autre, c'est-à-dire qu'ils ne peuvent pas être distingués par leurs échantillons à temps discret.

Exemple 2: La plage de fréquence d'échantillonnage requise par les systèmes DSP est large, allant d'environ 1 GHz dans le radar à 1 Hz dans l'instrumentation. Étant donné un taux d'échantillonnage pour une application spécifique, la période d'échantillonnage peut être déterminée. Certaines applications réelles utilisent les fréquences et périodes d'échantillonnage suivantes :

1. Dans les normes de compression de la parole de l'Union internationale des télécommunications (UIT), le taux d'échantillonnage de l'UIT-T G.729 et G.723.1 est f_S = 8 kHz, donc la période d'échantillonnage T = 1/8000 s = 125µs. Notez que 1 µs = 10^{-6} s.

2. Les systèmes de télécommunication à large bande, tels que l'UIT-T G.722, utilisent un taux d'échantillonnage de f_S = 16 kHz, donc T = 1/16 000 s = 62,5 µs.

3. Dans les CD audio, la fréquence d'échantillonnage est f_S = 44,1 kHz, donc T = 1/44 100 s = 22,676µs.

4. Les systèmes audio hautefidélité, tels que la norme AAC (codage audio avancé) MPEG-2 (groupe d'experts en images animées), la norme de compression audio MP3 (couche MPEG 3) et Dolby AC-3, ont un taux d'échantillonnage de f_S = 48 kHz, et donc T = 1/48 000 s = 20,833µs. La fréquence d'échantillonnage pour MPEG-2 AAC peut atteindre 96 kHz.

Exemple 3 :

On considère les deux signaux analogiques :

$x_1(t) = Cos2\Pi\ (10)t$ et $x_2(t) = Cos2\Pi\ (50)t$

Qui sont échantillonnés à un taux de F_S = 40 Hz. Les signaux à temps discret ou séquences correspondants sont :

$x_1(n) = cos2\Pi\ (10/40)n = cos(\Pi/2)n$

$x_2(n) = cos2\Pi\ (50/40)n = cos(5\Pi/2)n = cos(2\Pi n+\Pi/2n) = cos(\Pi/2)n \Rightarrow x_1(n) = x_2(n)$.

Nous disons que la fréquence $F_2 = 50$ Hz est un repliement de la fréquence $F_1 = 10$ Hz à un taux d'échantillonnage de 40 échantillons par seconde.

Exemple 4 : Considérons le signal analogique : $x_a(t) = A \cos(2\Pi Ft+\theta) = 3\cos 100\Pi t$

a) Déterminer le taux minimal d'échantillonnage nécessaire pour éviter le repliement (aliasing) ;
b) Supposons que le signal est échantillonné au taux $F_S = 200$ Hz. Quel est le signal à temps discret obtenu après échantillonnage ?
c) On suppose que le signal est échantillonné à un taux de $F_S = 75$ Hz. Quel est le signal à temps discret obtenu ?
d) Quelle est la fréquence $0 < F \leq F_S/2$ d'une sinusoïde qui donne des échantillons identiques à ceux obtenus dans C) ?

Solution :

a) La fréquence du signal analogique est F = 50 Hz. Ainsi le taux d'échantillonnage nécessaire pour éviter le repliement est $F_S = 100$ Hz.
b) Si le signal est échantillonné à $F_S = 200$ Hz, le signal à temps discret est : $x(n) = 3\cos(100/200)\Pi n = 3\cos(\Pi/2)n$.
c) Si le signal est échantillonné à $F_S = 75$ Hz, le signal à temps discret est : $x(n) = 3\cos(100/75)\Pi n = 3\cos(4/3)\Pi n = 3\cos(2\Pi-2/3\Pi)n = 3\cos(2/3)\Pi n$.
d) Pour le taux d'échantillonnage de $F_S = 75$ Hz, nous avons $F = fF_S = 75f$, la fréquence de la sinusoïde dans c) est $f = 1/3 \Rightarrow F = 25$ Hz.

2.5 Quantification uniforme

L'ADC suppose que les valeurs d'entrée couvrent une plage pleine échelle, disons R. Les valeurs typiques de R sont comprises entre 1 et 15 volts. Étant donné que la valeur échantillonnée quantifiée xQ (nT) est représentée par des bits B, elle ne peut prendre qu'un seul des 2B niveaux de quantification possibles. Si l'espacement entre ces niveaux est le même sur toute la plage R, alors nous avons un quantificateur uniforme. L'espacement entre les niveaux de quantification est appelé largeur de quantification ou résolution du quantificateur.

Pour une quantification uniforme, la résolution est donnée par :

$$Q = \frac{R}{2^B} \qquad (2.7)$$

Le nombre de bits requis pour atteindre une résolution requise de Q est donc

$$B = \log_2 \frac{R}{Q} \qquad (2.8)$$

La plupart des CAN peuvent accepter des entrées bipolaires, ce qui signifie que les valeurs échantillonnées se situent dans la plage symétrique :

$$-\frac{R}{2} \leq x(nT) \prec \frac{R}{2} \qquad (2.9)$$

Pour les entrées unipolaires, $0 \leq x(nT) < R$

En pratique, le signal d'entrée x (t) doit être préconditionné pour se situer dans la plage pleine échelle du quantificateur. La figure 2.11 montre les niveaux de quantification d'un quantificateur 3 bits pour les entrées bipolaires.

Si la conversion est unipolaire, la grandeur de sortie est toujours de même signe et peut donc prendre les valeurs comprises entre 0 et 2^n-1.

Si on dispose, par exemple, d'un CNA unipolaire 12 bits-10 volts. Si la conversion est bipolaire, la grandeur de sortie peut être négative ou positive, et peut donc prendre les valeurs comprises entre -2^{n-1} et 2^{n-1} -1.

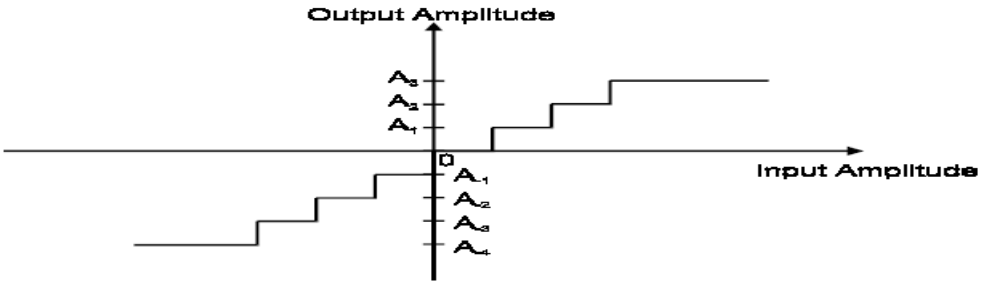


Fig. 2.11 : Une fonction de transfert de quantificateur uniforme à 3 bits

Nous discutons maintenant une méthode de représentation du signal à temps discret échantillonné x (nT) comme un nombre binaire avec un nombre fini de bits. Il s'agit du processus de quantification et d'encodage. Si la longueur de mot d'un CAN est de B bits, il existe 2^B valeurs (niveaux) différentes qui peuvent être utilisées pour représenter un échantillon. Si x (n) se situe entre deux niveaux de quantification, il sera soit arrondi, soit tronqué. L'arrondi remplace x (n) par la valeur du niveau de quantification le plus proche, tandis que la troncature remplace x (n) par la valeur du niveau inférieur. Étant donné que l'arrondi produit une représentation moins biaisée des valeurs analogiques, il est largement utilisé par les ADC. Par conséquent, la quantification est un processus qui représente un échantillon à valeur analogique x (nT) dont le niveau le plus proche correspond au signal numérique x (n).

Nous pouvons utiliser 2 bits pour définir quatre niveaux également espacés (00, 01, 10 et 11) pour classer le signal dans les quatre sous-plages comme illustré dans la figure 2.12. Sur cette figure, le symbole «o» représente le signal à temps discret x (nT) et le symbole «•» représente le signal numérique x (n). L'espacement entre deux niveaux de quantification consécutifs est appelé la largeur de quantification, le pas ou la résolution. Si l'espacement entre ces niveaux est le même, alors nous avons un quantificateur uniforme. Pour la quantification uniforme, la résolution est donnée en divisant une plage pleine échelle avec le nombre de niveaux de quantification, 2^B.

Dans la figure 2.12, la différence entre le nombre quantifié et la valeur d'origine est définie comme l'erreur de quantification, qui apparaît sous forme de bruit dans la sortie du convertisseur. Il est également appelé bruit de quantification, qui est supposé être des variables aléatoires qui sont uniformément réparties. Si un quantificateur B bits est utilisé, le rapport signal sur bruit de quantification (SQNR) est approximé par SQNR ≈ 6B dB.

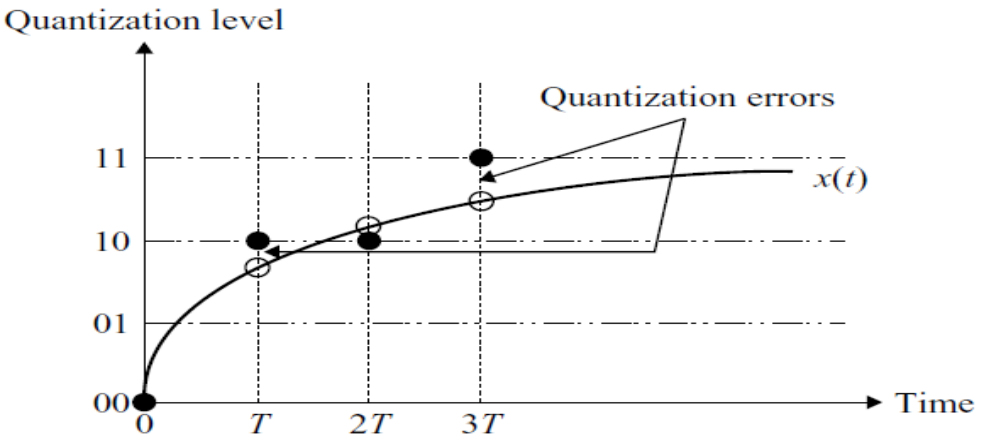


Fig. 2.12 : Échantillons numériques utilisant un quantificateur 2 bits

L'erreur de quantification est la différence entre la valeur réelle échantillonnée et la valeur quantifiée. Mathématiquement, c'est

$$e(nT) = x(nT) - x_Q(nT) \qquad (2.8)$$

ou son équivalent,

$$e(n) = x(n) - x_Q(n) \qquad (2.10)$$

Si x (n) se situe entre deux niveaux de quantification, il sera soit arrondi soit tronqué.

L'arrondi remplace x (n) par la valeur du niveau de quantification le plus proche. La troncature remplace x (n) par la valeur du niveau en dessous.

Pour l'arrondi, l'erreur est donnée par :

$$-\frac{Q}{2} \prec e \prec \frac{Q}{2} \qquad (2.11)$$

Pour la troncature, l'erreur est

$$0 \leq e < Q \qquad (2.12)$$

Il est évident que l'arrondi produit une représentation moins biaisée des valeurs analogiques.

L'erreur moyenne est donnée par :

$$\bar{e} = \frac{1}{Q}\int_{-\frac{Q}{2}}^{\frac{Q}{2}} e\,de = 0 \qquad (2.13)$$

Ce qui signifie qu'en moyenne la moitié des valeurs sont arrondies vers le haut et la moitié vers le bas.

La valeur quadratique moyenne de l'erreur nous donne une idée de la puissance moyenne du signal d'erreur. Elle est donnée par :

$$\bar{e} = \frac{1}{Q}\int_{-\frac{Q}{2}}^{\frac{Q}{2}} e^2 de = \frac{Q^2}{12} \qquad (2.14)$$

L'erreur quadratique moyenne est donc :

$$e_{rms} = \sqrt{\bar{e}^2} = \frac{Q}{\sqrt{12}} \qquad (2.15)$$

Le rapport signal - bruit de quantification est :

$$\text{SQNR} = 20 \log_{10} [R/Q] = 20 \log_{10} (2^B) = 20B \log_{10}2 = 6B \text{ db} \qquad (2.16)$$

Ainsi, si nous augmentons le nombre de bits de l'ADC d'un bit, le rapport signal sur bruit de quantification s'améliore de 6 dB. L'équation ci-dessus nous donne la plage dynamique du quantificateur.

Exemple :

La plage dynamique de l'oreille humaine est d'environ 100 dB. Si un système audio numérique est nécessaire pour correspondre à cette plage dynamique, il faudra

$$100/6 = 16{,}67 \text{ bits}$$

Un quantificateur 16 bits atteindra une plage dynamique de 96 dB.

Si la fréquence la plus élevée que l'oreille humaine peut entendre est de 20 kHz, une fréquence d'échantillonnage d'au moins 40 kHz est requise. Si le taux d'échantillonnage réel est de 44 kHz, le débit binaire de ce système sera

$$16x44 = 704 \text{ kbits / sec}$$

Il s'agit du débit binaire typique d'un lecteur de disque compact.

Étant donné que l'erreur de quantification est un nombre aléatoire dans la plage donnée, elle est généralement modélisée comme un signal aléatoire (ou bruit) avec une distribution uniforme, comme le montre la figure 2.13.

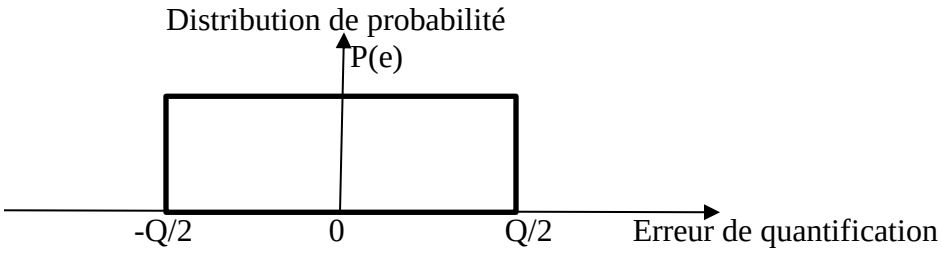


Fig. 2.13 : Distribution uniforme de l'erreur de quantification

2.6 Codage binaire des niveaux de quantification

Considérons un nombre binaire à n bits représentant un intervalle R pleine échelle. En d'autres termes, l'intervalle R est quantifiée en 2^n niveaux de quantification. Si R est unipolaire, la valeur quantifiée x_Q se situe dans l'intervalle [0, R]. S'il est bipolaire, x_Q se situe dans l'intervalle [−R / 2, R / 2].

Le modèle à n bits est représenté par un vecteur $b = [b_{n-1}, b_{n-2}, \ldots, b_1, b_0]$ où b_{n-1} est appelé le bit le plus significatif (MSB) et b_0 est le bit le moins significatif (LSB). Il

existe de nombreuses façons d'utiliser ce modèle à n bits pour coder x_Q. Les trois moyens les plus courants sont :

- Binaire naturel unipolaire :

$$x_Q = R(b_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n}) \qquad (2.17)$$

- Binaire décalé bipolaire :

$$x_Q = R(b_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n} -0.5) \qquad (2.18)$$

- Complément à 2 bipolaire :

$$x_Q = R(Б_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n} -0.5) \qquad (2.19)$$

Ici, $Б_{n-1}$ désigne le complément à 2 de b_{n-1}.

Exemple :

Pour R=2 v et une quantification à 3 bits (8 niveaux), la correspondance entre la représentation binaire et la valeur quantifiée est :

b_2 b_1 b_0	Binaire naturel	Binaire décalé	Complément à 2
111	1.75	0.75	-0.25
110	1.50	0.50	-0.50
101	1.25	0.25	-0.75
100	1.00	0.00	-1.00
011	0.75	-0.25	0.75
010	0.50	- 0.50	0.50
001	0.25	- 0.75	0.25
000	0.00	-1.00	0.00

La représentation binaire naturelle unipolaire code les niveaux compris entre 0 et 2 V. Le binaire décalé et le complément à 2 codent les niveaux compris entre –1 V et 1 V.

2.7 Quantification non uniforme

L'une des hypothèses que nous avons formulées lors de l'analyse de l'erreur de quantification est que l'amplitude du signal échantillonné est uniformément répartie sur toute la gamme. Cette hypothèse peut ne pas être valable pour certaines applications. Par exemple, les signaux de parole sont connus pour avoir une large gamme dynamique. La parole vocale (par exemple, les sons vocaux) peut avoir des amplitudes qui couvrent tout l'intervalle de pleine échelle, tandis que la parole non plus douce (par exemple les consonnes telles que les fricatives) a généralement des amplitudes beaucoup plus petites. De plus, une personne moyenne ne parle que 60% du temps lorsqu'elle parle. Les 40% restants sont du silence avec une amplitude de signal négligeable.

Si une quantification uniforme est utilisée, les sons plus forts seront correctement représentés. Cependant, les sons plus doux n'occuperont probablement qu'un petit nombre de niveaux de quantification avec des valeurs binaires similaires. Cela signifie que nous ne pourrions pas distinguer les sons plus doux. En conséquence, la parole analogique reconstruite à partir de ces échantillons numériques ne sera presque pas aussi intelligible que l'original.

Pour contourner ce problème, une quantification non uniforme peut être utilisée. Plus de niveaux de quantification sont attribués aux amplitudes inférieures tandis que les amplitudes plus élevées auront moins de niveaux. Ce schéma de quantification est illustré dans la figure 2.14.

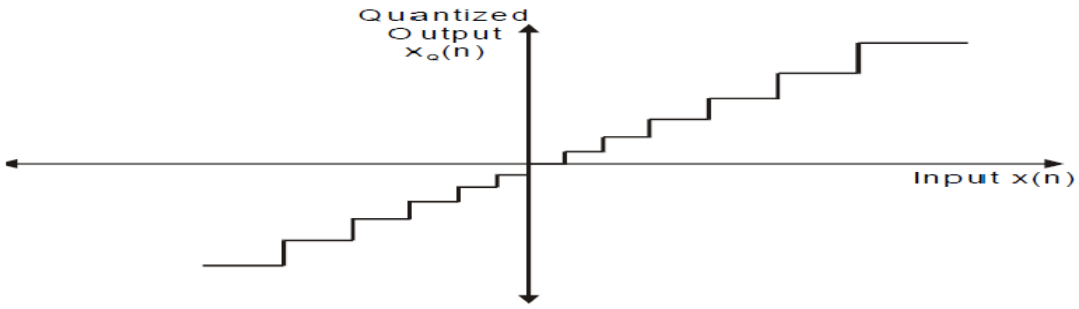


Fig. 2.14 : Quantification non uniforme

Alternativement, un quantificateur uniforme peut toujours être utilisé, mais le signal d'entrée est d'abord compressé par un système avec une relation entrée-sortie (ou fonction de transfert) similaire à celle illustrée dans la figure 2.15.

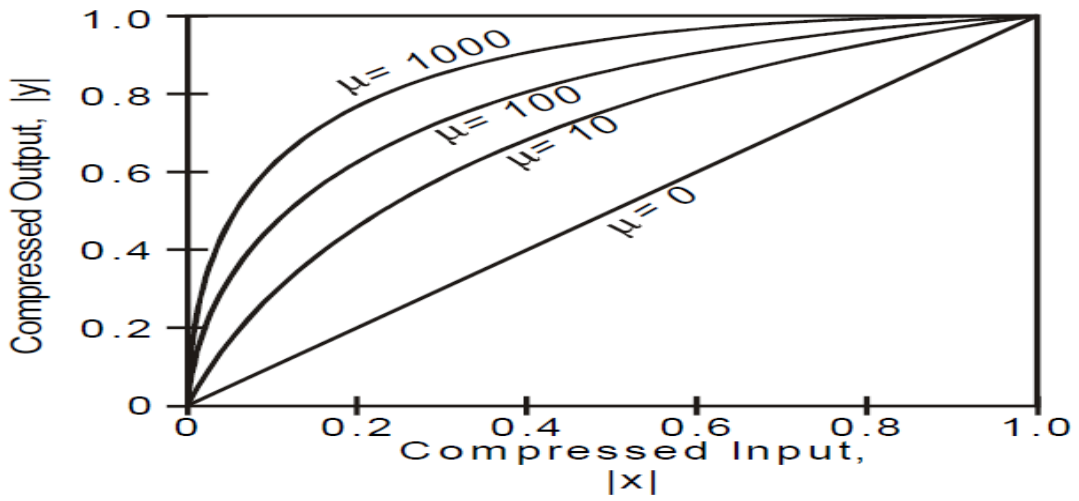


Fig. 2.15 : Caractéristiques de loi μ de compression

Les amplitudes plus élevées du signal d'entrée sont comprimées, ce qui réduit efficacement le nombre de niveaux qui lui sont attribués. Les signaux d'amplitudes inférieures sont étendus (ou amplifiés de manière non uniforme), ce qui les fait occuper un grand nombre de niveaux de quantification. Après le traitement, une opération inverse est appliquée au signal de sortie (en l'étendant). Le système qui étend le signal a une relation entrée-sortie qui est l'inverse du compresseur. L'extenseur étend les hautes amplitudes et comprime les

faibles amplitudes. L'ensemble du processus est appelé companding (COMpressing et exPANDING) (comprimer et étendre).

Le companding est largement utilisé dans les systèmes téléphoniques publics. Il existe deux schémas de compression-extension distincts. En Europe, la compression A-law est utilisée et aux États-Unis, la compression μ-law est utilisée.

La caractéristique de compression de loi μ est donnée par la formule :

$$y = y_{\max} \frac{\ln\left[1+\mu\left(\frac{|x|}{x_{\max}}\right)\right]}{\ln(1+\mu)} \operatorname{sgn}(x) \qquad (2.20)$$

Où $\operatorname{sgn}(x) = \begin{cases} +1 & si\ x \geq 0 \\ -1 & si\ x < 0 \end{cases}$

Ici, x et y représentent les valeurs d'entrée et de sortie, et x_{max} et y_{max} sont respectivement les valeurs maximales de l'entrée et de la sortie. μ est une constante positive. La norme nord-américaine spécifie μ par 255. Notez que μ = 0 correspond à une relation linéaire entrée-sortie (c'est-à-dire une quantification uniforme). La caractéristique de compression est illustrée dans la figure 2.15.

La caractéristique de compression de la loi A est donnée par :

$$y = \begin{cases} y_{\max} \dfrac{A\left(\frac{|x|}{x_{\max}}\right)}{1+\ln A} \operatorname{sgn}(x), & 0 < \dfrac{|x|}{x_{\max}} \leq \dfrac{1}{A} \\ y_{\max} \dfrac{1+\ln\left[A\left(\frac{|x|}{x_{\max}}\right)\right]}{1+\ln A} \operatorname{sgn}(x), & \dfrac{1}{A} < \dfrac{|x|}{x_{\max}} < 1 \end{cases} \qquad (2.21)$$

Ici, A est une constante positive. La norme européenne spécifie A comme étant 87,6. La figure 2.16 montre graphiquement la caractéristique.

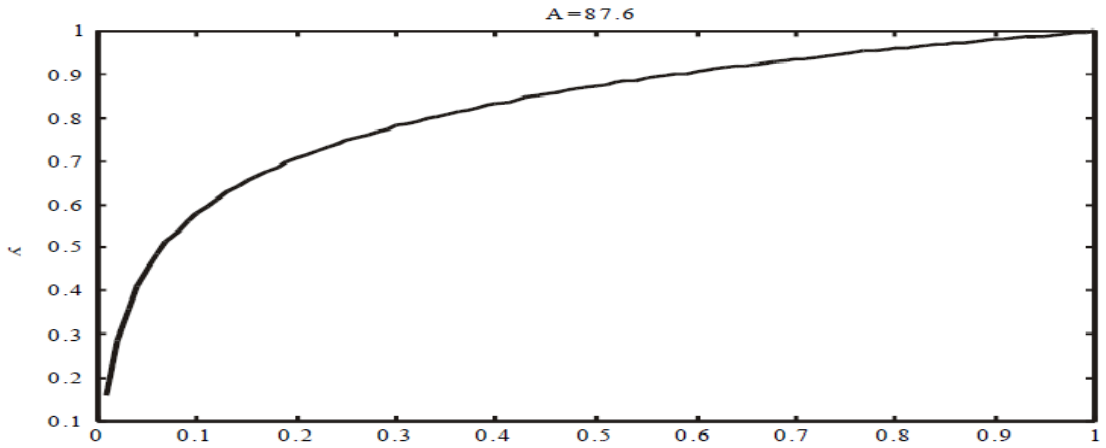


Fig. 2.16 : Les caractéristiques de compression de la loi A

Chapitre 3 : Architecture des DSP TMS320 C6x

La famille de processeurs TMS320C6x, fabriqués par Texas Instruments, est conçue pour offrir de la vitesse. Ils sont conçus pour des millions d'instructions par seconde (MIPS), telles que l'imagerie sans fil et numérique de troisième et quatrième génération (3 et 4G). Il existe de nombreuses versions de processeurs appartenant à cette famille, différents par le temps de cycle d'instruction, la vitesse, la consommation d'énergie, la mémoire, les périphériques, l'emballage et le coût. Par exemple, la version C6416-600 à virgule fixe fonctionne à 600 MHz (temps de cycle de 1,67 ns), offrant une performance de pointe de 4800 MIPS. La version à virgule flottante C6713-225 fonctionne à 225 MHz (temps de cycle de 4,4 ns), offrant une performance de pointe de 1350 MIPS.

La figure 3-1 montre un schéma de principe de l'architecture générique C6x. L'unité centrale de traitement (CPU) du 6x se compose de huit unités fonctionnelles divisées en deux côtés: A et B. Chaque côté a une unité .M (utilisée pour l'opération de multiplication), une unité .L (utilisée pour

Les opérations logiques et arithmétiques), une unité .S (utilisée pour le branchement, la manipulation des bits et les opérations arithmétiques) et une unité .D (utilisée pour le chargement, le stockage et les opérations arithmétiques). Certaines instructions, telles que ADD, peuvent être effectuées par plusieurs unités. Il y a seize registres de 32 bits associés à chaque côté. L'interaction avec le CPU doit se faire via ces registres.

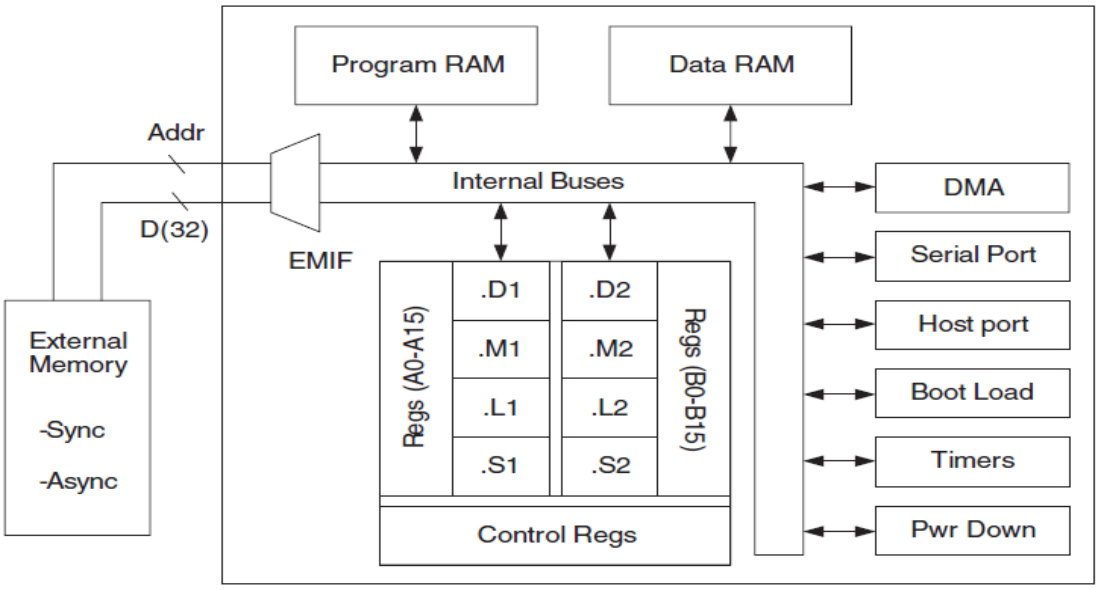


Fig. 3.1 : Architecture générique du C6x

Comme le montre la figure 3-2, les bus internes se composent d'un bus d'adresse de programme 32 bits, d'un bus de données de programme 256 bits pouvant accueillir huit instructions 32 bits, de deux bus d'adresses de données 32 bits (DA1 et DA2), de deux bus de données de 32 bits pour le chargement (64 bits pour la version C64) (LD1 et LD2) et deux bus de stockage de données 32 bits (64 bits pour la version à virgule flottante) (ST1 et ST2). De plus, il existe un bus de données d'accès direct à la mémoire (DMA) de 32 bits et un bus d'adresses DMA 32 bits. La mémoire hors puce, ou externe (off-chip memory), est accessible via un bus d'adresse 20 bits et un bus de données 32 bits.

Les périphériques d'un processeur C6x typique comprennent une interface de mémoire externe (EMIF), un DMA (Accès Mémoire Directe), un chargeur d'amorçage (de démarrage), un port série multi-canaux tampon (McBSP), une interface de port hôte (HPI), un temporisateur et une unité de désamorçage (Power Down Unit). EMIF fournit le délai nécessaire pour accéder à la mémoire externe. Le DMA permet le mouvement des données d'un endroit en mémoire à un autre sans interférer avec le fonctionnement du CPU. Le

chargeur d'amorçage démarre le code de la mémoire hors puce ou HPI vers la mémoire interne. McBSP fournit une liaison de communication série multicanal haute vitesse. HPI permet à un hôte d'accéder à la mémoire interne. Le temporisateur fournit deux compteurs 32 bits. L'unité de mise hors tension (désamorçage) est utilisée pour économiser de l'énergie pendant des durées lorsque le CPU est inactif.

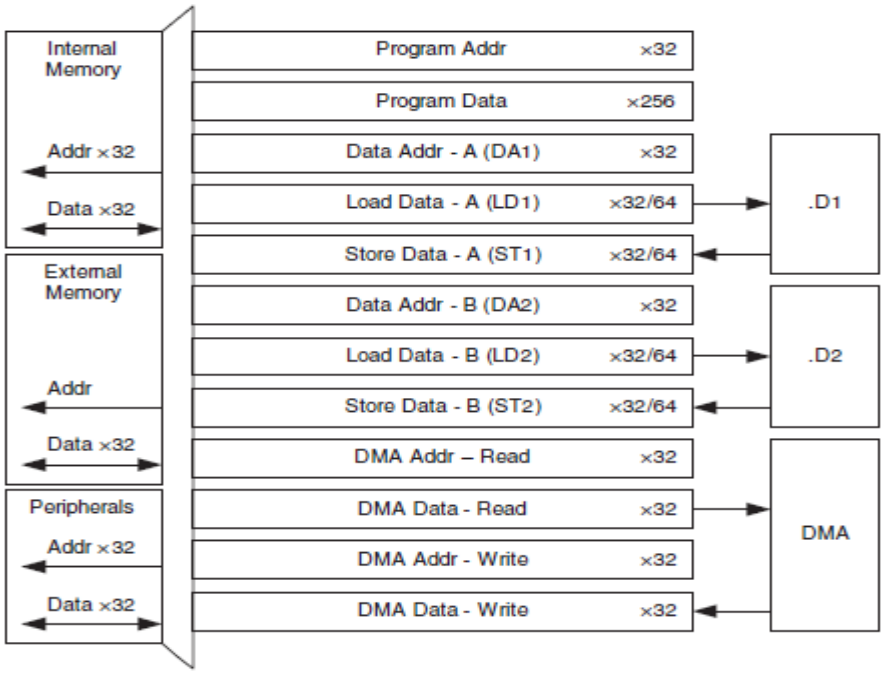


Fig. 3.2 : Bus internes du C6x

3.1 Opération sur l'unité de commande (CPU)

Exemple du Dot Product :

Comme le montre la figure 3.1, le processeur C6x est scindé en deux chemins de données, le chemin de données A (ou 1) et le chemin de données B (ou 2). Un moyen efficace de comprendre le fonctionnement du processeur consiste à passer par un exemple. La figure 3.3 montre le code assembleur pour un produit scalaire à 40 points y (entre deux vecteurs a et x), $y = \sum_{=1}^{40} a_n * \mathrm{x}_n$

À ce stade, il convient de mentionner que l'assembleur n'est pas sensible à la casse (c'est-à-dire que les instructions et les registres peuvent être écrits en minuscules ou en majuscules).

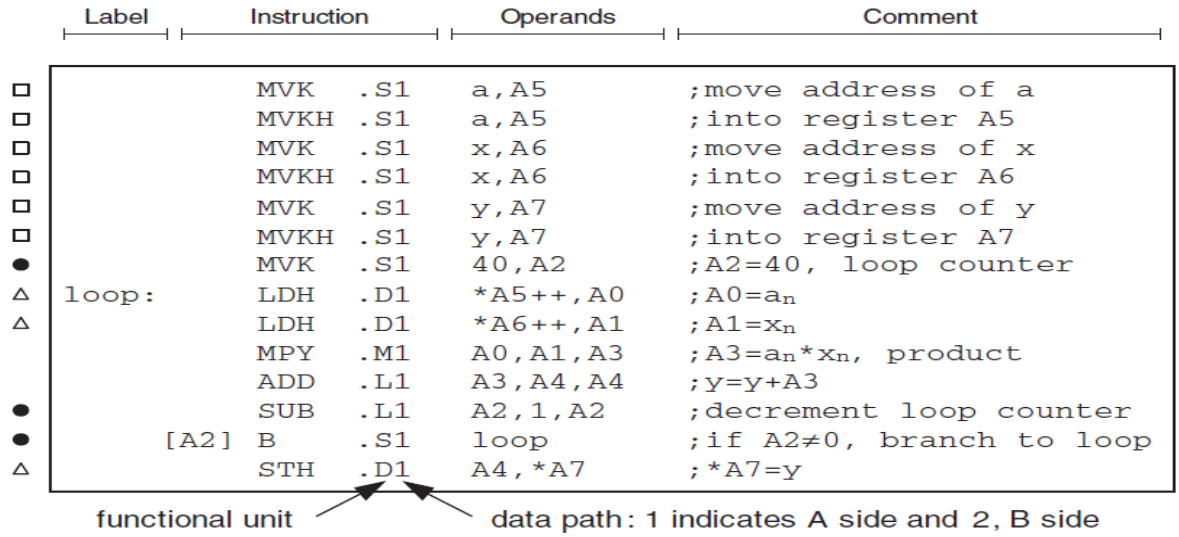


Fig. 3.3 : Code assembleur du Dot product

Les registres affectés à a_n, x_n, compteur de boucles, produit, y, &a[n] (adresse de a_n), &x[n] (adresse de x_n), et &y[n] (adresse de y_n) sont illustrés à la figure 3.4. Dans cet exemple, seules les unités fonctionnelles et les registres du côté A sont utilisés.

Une boucle est créée par les instructions indiquées par • 's. Tout d'abord, un compteur de boucle est configuré à l'aide de l'instruction constante de déplacement MVK. Cette instruction utilise l'unité .S1 pour placer la constante 40 dans le registre A2. Le début de la boucle est indiqué par la boucle d'étiquette et la fin par une instruction de soustraction SUB pour décrémenter le compteur de boucle suivie d'une instruction de branchement B pour revenir à la boucle.

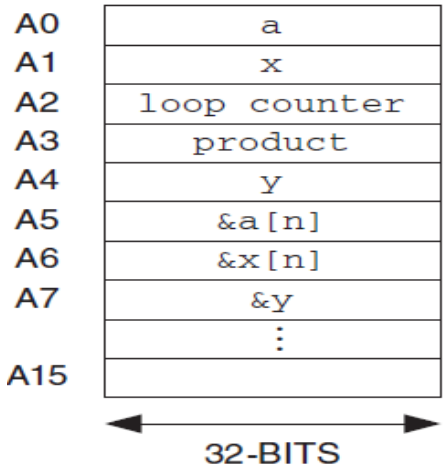


Fig. 3.4 : Registres côté A

La soustraction est effectuée par l'unité .L1 et le branchement par l'unité .S1. Les crochets dans le cadre de l'instruction branch indiquent qu'il s'agit d'une instruction conditionnelle. Toutes les instructions du C6x peuvent être conditionnées sur la base d'une valeur nulle ou non nulle dans l'un des registres : A1, A2, B0, B1 et B2. La syntaxe [A2] signifie "exécuter l'instruction si A2 ≠ 0" et [! A2] signifie "exécuter l'instruction si A2 = 0". À la suite de ces instructions, la boucle est répétée 40 fois.

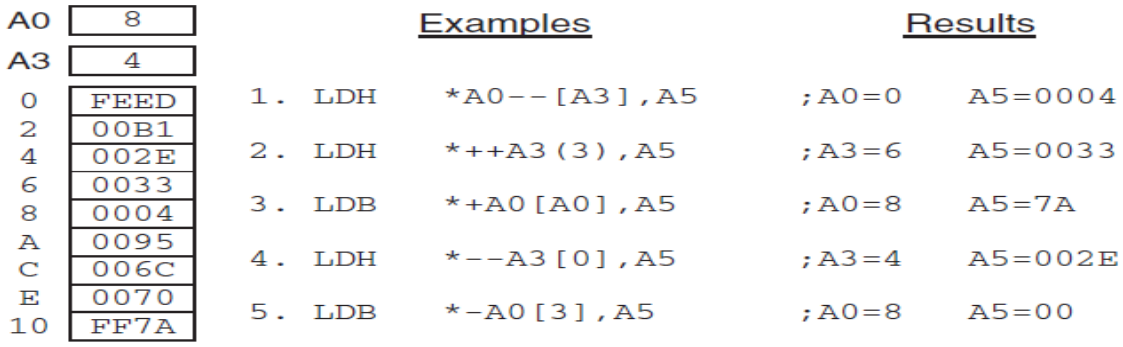
Étant donné que l'interaction avec les unités fonctionnelles se fait via les registres du côté A, ces registres doivent être configurés afin de démarrer la boucle. Les instructions marquées d'un □ indiquent les instructions nécessaires pour le faire. MVK et MVKH sont utilisés pour charger l'adresse de a_n, x_n et y dans les registres A5, A6 et A7, respectivement. Ces instructions doivent être exécutées dans l'ordre indiqué pour charger les 16 bits inférieurs de l'adresse 32 bits complète en premier, suivis des 16 bits supérieurs. Ces registres sont utilisés comme pointeurs pour charger a_n, x_n dans les registres A0, A1 et stocker y à partir du registre A4 (instructions marquées par Δ). La notation * du langage de programmation C est utilisée pour indiquer qu'un registre est utilisé comme pointeur.

Selon le type de données, l'une des instructions de chargement suivantes peut être utilisées : LDB octets (8 bits), LDH demi-mots (16 bits) ou mots (LDW 32 bits). Ici, les données sont supposées être des demi-mots. Le chargement / stockage est effectué par l'unité .D1, car les unités .D sont les seules unités capables d'interagir avec la mémoire de données.

Notez que les pointeurs A5 et A6 doivent être post-incrémentés (notation C), afin qu'ils pointent vers les valeurs suivantes pour la prochaine itération de la boucle. Lorsque les registres sont utilisés comme pointeurs, il existe plusieurs façons d'effectuer l'arithmétique des pointeurs. Ceux-ci incluent des options de pré et post-incrémentation / décrémentation d'une certaine valeur de déplacement, où le pointeur est modifié avant ou après son utilisation (par exemple, * A1 [disp] et * A1 ++ [disp]). De plus, une option de pré-décalage peut être effectuée sans modification du pointeur (par exemple, * + A1 [disp]). Le déplacement entre crochets spécifie le nombre d'éléments de données (selon le type de données), tandis que le déplacement entre parenthèses spécifie le nombre d'octets. Ces options de décalage de pointeur sont répertoriées dans la figure 3.5 avec quelques exemples.

Syntaxe	Description	Pointeur modifié
*R	Pointeur	Non
*+R[disp]	+Pré-décalage	Non
*-R[disp]	– Pré-décalage	Non
*++R[disp]	Pré-incrémentation	Oui
*--R[disp]	Pré-décrémentation	Oui
*R++[disp]	Post-incrémentation	Oui
*R--[disp]	Post-décrémentation	Oui

[disp] spécifie # éléments - taille en W, H ou B
(disp) spécifie # octets
(a)



(b)

Fig. 3.5 : (a) Décalages de pointeurs, (b) exemples de pointeurs
(Remarque : les instructions sont indépendantes et non séquentielles)

Enfin, les instructions MPY et ADD dans la boucle effectuent l'opération du produit scalaire. L'instruction MPY est effectuée par l'unité .M1 et ADD par l'unité .L1.

3.2 Mise en œuvre de la somme des produits (SOP) :

Il a été démontré que la SOP est l'élément clé de la plupart des algorithmes DSP.

Écrivons donc le code de cet algorithme et découvrons en même temps l'architecture du C6000.

$$y = \sum_{n=1}^{N} a_n * x_n = a_1 * x_1 + a_2 * x_2 + \ldots + a_N * x_N \qquad (3.1)$$

Deux opérations de base sont requises pour cet algorithme.

(1) Multiplication

(2) Addition

Alors implémentons l'algorithme SOP! L'implémentation dans ce module se fera en assembleur.

La multiplication de a1 par x1 se fait en assemblage par l'instruction suivante :

MPY a1, x1, Y

Cette instruction est exécutée par un multiplicateur appelé «.M», L'unité .M effectue des multiplications dans le matériel :

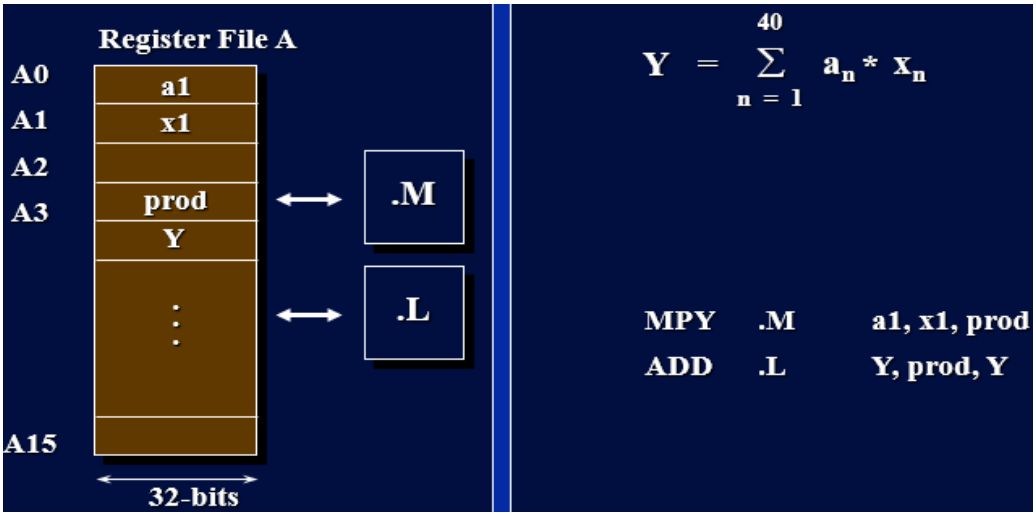
MPY .M a1, x1, Y

Remarque : le multiplicateur 16 bits par 16 bits fournit un résultat de 32 bits.

Le multiplicateur 32 bits par 32 bits fournit un résultat de 64 bits.

MPY .M a1, x1, prod

ADD .L Y, prod, Y



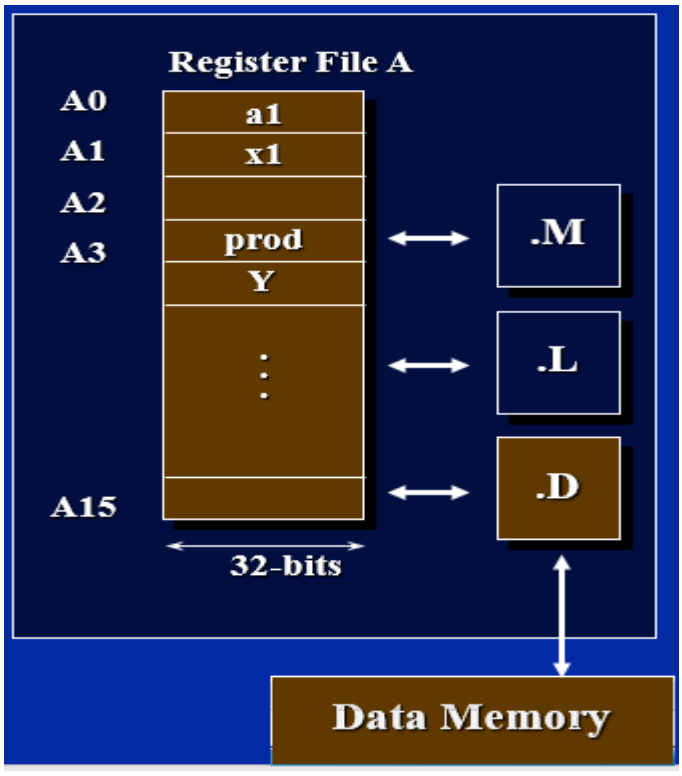
Les processeurs RISC tels que le C6000 utilisent des registres pour contenir les opérandes, modifions donc ce code. Corrigeons cela en remplaçant a, x, prod et Y par les registres comme indiqué ci-dessus.

```
MPY     .M     A0, A1, A3

ADD     .L     A4, A3, A4
```

Les registres A0, A1, A3 et A4 contiennent les valeurs à utiliser par les instructions. Le fichier de registre A contient 16 registres (A0 -A15) d'une largeur de 32 bits.

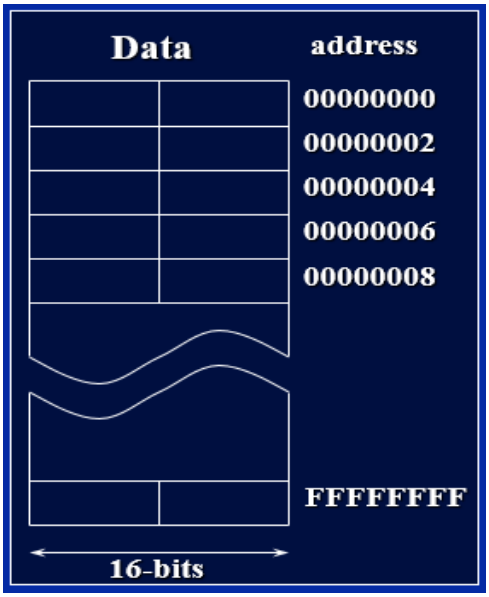
Comment charger les opérandes dans les registres ? Les opérandes sont chargés dans les registres en les chargeant depuis la mémoire à l'aide de l'unité .D.



Il convient de noter à ce stade que le seul moyen d'accéder à la mémoire est via l'unité .D.

Quelle(s) instruction(s) peut-on utiliser pour charger des opérandes de la mémoire dans les registres ? L'instruction de chargement LOAD

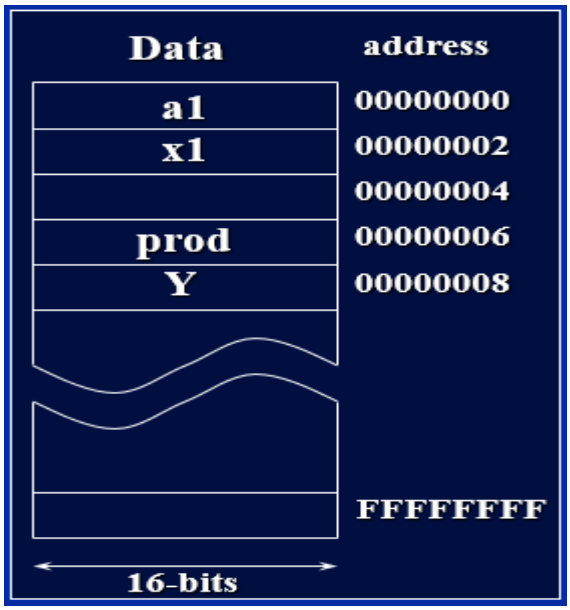
Avant d'utiliser l'unité de chargement LOAD, vous devez savoir que ce processeur est adressable en octets, ce qui signifie que chaque octet est représenté par une adresse unique. Les adresses ont également une largeur de 32 bits.



La syntaxe de l'instruction de chargement est la suivante :

LD *Rn, Rm

Où : Rn est un registre qui contient l'adresse de l'opérande à charger et Rm est le registre de destination.



La question est maintenant de savoir combien d'octets vont être chargés dans le registre de destination ?

La réponse est que cela dépend de l'instruction que vous choisissez :

LDB: charge un octet (8 bits)

LDH: charge un demi-mot (16 bits)

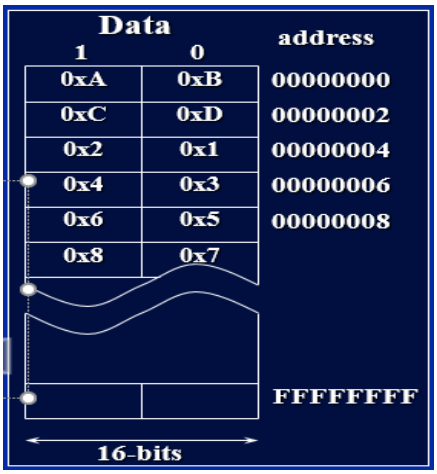
LDW: charge un mot (32 bits)

LDDW: charge un double mot (64 bits)

Remarque : LD seul n'existe pas.

Exemple:

Si nous supposons que A5 = 0x4, alors:
(1) LDB *A5, A7; donne A7 = 0x00000001
(2) LDH *A5, A7; donne A7 = 0x00000201
(3) LDW *A5, A7; donne A7 = 0x04030201
(4) LDDW *A5, A7: A6; donne A7: A6 = 0x0807060504030201



Si les données ne sont accessibles que par l'instruction de chargement et l'unité .D, comment pouvons-nous charger le pointeur de registre Rn en premier lieu ?

L'instruction MVKL permettra le déplacement d'une constante de 16 bits dans un registre comme indiqué ci-dessous :

MVKL.? a, A5

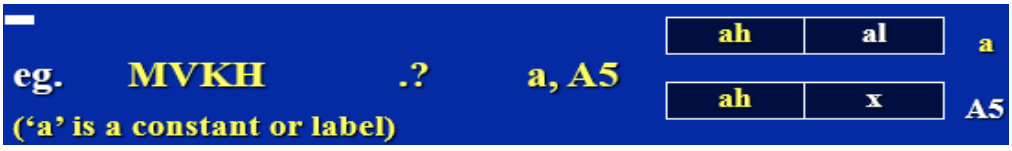
(«a» est une constante ou une étiquette)

Combien de bits représentent une adresse complète ? 32 bits

Alors pourquoi l'instruction n'autorise-t-elle pas un déplacement de 32 bits ?

(Toutes les instructions ont une largeur de 32 bits)

Pour résoudre ce problème, une autre instruction est disponible : MVKH



Enfin, pour déplacer l'adresse de 32 bits vers un registre, nous pouvons utiliser :

```
MVKL    a, A5
MVKH    a, A5
```

Utilisez toujours MVKL puis MVKH, regardez les exemples suivants :

Exemple 1 :

A5 = 0x87654321

```
MVKL        0x1234FABC, A5
```

A5 = 0xFFFFFABC (sign extension)

```
MVKH        0x1234FABC, A5
```

A5 = 0x1234FABC ; OK

Exemple 2:

```
MVKH        0x1234FABC, A5
```

A5 = 0x12344321

```
MVKL        0x1234FABC, A5
```

A5 = 0xFFFFFABC ; Faux

```
MVKL        pt1, A5
MVKH        pt1, A5
MVKL        pt2, A6
MVKH        pt2, A6
```

```
LDH     .D      *A5, A0

LDH     .D      *A6, A1

MPY     .M      A0, A1, A3

ADD     .L      A4, A3, A4
```

pt1 et pt2 pointent vers certains emplacements dans la mémoire de données.

Jusqu'à présent, nous n'avons implémenté le SOP que pour un seul cycle, c'est-à-dire

$$Y = a1 * x1$$

Créons donc une boucle afin de pouvoir implémenter le SOP pour N cycles.

Avec les processeurs C6000, il n'y a pas d'instructions dédiées telles que la répétition de bloc. La boucle est créée à l'aide de l'instruction B.

Quelles sont les étapes pour créer une boucle ?

Créez une étiquette vers laquelle vous branchez.

2. Ajoutez une instruction de branchement, B.

3. Créez un compteur de boucles.

4. Ajoutez une instruction pour décrémenter le compteur de boucles.

5. Rendez la branche conditionnelle en fonction de la valeur du compteur de boucles.

```
                MVKL    .S    pt1, A5
                MVKH    .S    pt1, A5
                MVKL    .S    pt2, A6
                MVKH    .S    pt2, A6
                MVKL    .S    count, B0
loop            LDH     .D    *A5, A0
                LDH     .D    *A6, A1
                MPY     .M    A0, A1, A3
                ADD     .L    A4, A3, A4
                SUB     .S    B0, 1, B0
                B       .S    loop
```

Quelle est la syntaxe pour rendre l'instruction conditionnelle ?

[Condition] Instruction Étiquette

par exemple. [B1] B Boucle

(1) La condition peut être l'un des registres suivants : A1, A2, B0, B1, B2.

(2) Toute instruction peut être conditionnelle.

La condition peut être inversée en ajoutant le symbole d'exclamation "!" comme suit :

[! condition] d'instruction Étiquette

par exemple :

```
[!B0]   B    loop   ;branch if B0 = 0

[B0]    B    loop   ;branch if B0 != 0
```

Ce code effectue désormais les opérations suivantes :

a0 * x0 + a1 * x1 + a2 * x2 + ... + aN * xN

Le pointeur A7 n'a pas été initialisé.

```
MVKL   .S   pt1, A5

MVKH   .S   pt1, A5

MVKL   .S   pt2, A6

MVKH   .S   pt2, A6

MVKL   .S2  pt3, A7 (Le pointeur A7 est maintenant initialisé.)

MVKH   .S2  pt3, A7

MVKL   .S   count, B0
```

```
loop        LDH     .D      *A5, A0
            LDH     .D      *A6, A1
            MPY     .M      A0, A1, A3
            ADD     .L      A4, A3, A4
            SUB     .S      B0, 1, B0
[B0]        B       .S      loop
            STH     .D      A4, *A7
```

A4 est utilisé comme accumulateur, il doit donc être remis à zéro.

(MIPS = IPC * f * 10^6)...

3.3 Unité de commande Pipelinée

En général, il faut plusieurs étapes pour exécuter une instruction. Fondamentalement, ces étapes sont la recherche, le décodage et l'exécution. Si ces étapes sont effectuées en série, toutes les ressources du processeur, telles que plusieurs bus ou unités fonctionnelles, ne sont pas entièrement utilisées.

Afin d'augmenter le débit, les processeurs DSP sont conçus pour être pipelinés. Cela signifie que les étapes précédentes sont effectuées simultanément. La figure 3.6 illustre la différence de temps de traitement pour trois instructions exécutées sur une CPU série ou non pipelinée et une CPU pipelinée. Comme on peut le voir, une CPU en pipeline nécessite moins de cycles d'horloge pour exécuter le même nombre d'instructions.

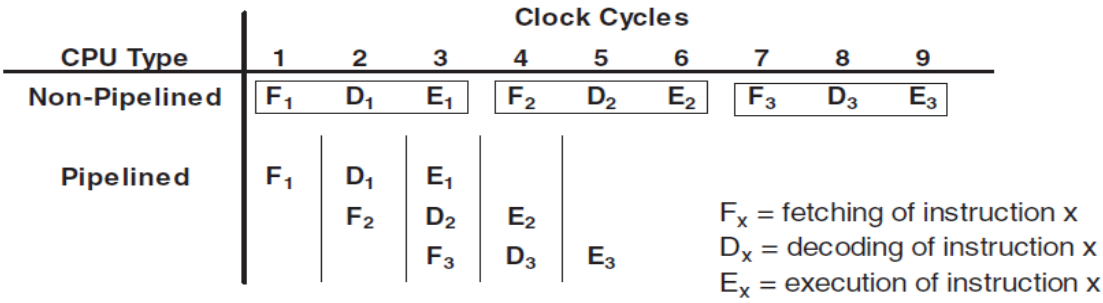


Fig. 3.6 : Unité de commande pipelinée et non pipelinée

Sur le processeur C6x, la recherche (fetching) se compose de quatre phases, chacune nécessitant un cycle d'horloge.

Il s'agit notamment de générer l'adresse de recherche (notée F1), d'envoyer l'adresse à la mémoire (F2), d'attendre les données (F3) et de lire l'opcode dans la mémoire (F4). Le décodage se compose de deux phases, chacune nécessitant un cycle d'horloge. Celles-ci sont envoyées aux unités fonctionnelles appropriées (désignées par D1) et décodage (D2).

En raison des retards associés à la multiplication des instructions (MPY - 1 retard), chargement (LDx - 4 retards) et au la branchement (B - 5 retards), l'étape d'exécution peut comprendre jusqu'à six phases (désignées par E1 à E6) , acceptant un maximum de 5 retards. Par conséquent, comme le montre la figure 3.7, l'étape F se compose de quatre, l'étape D de deux et l'étape E de six sous-étapes ou phases possibles.

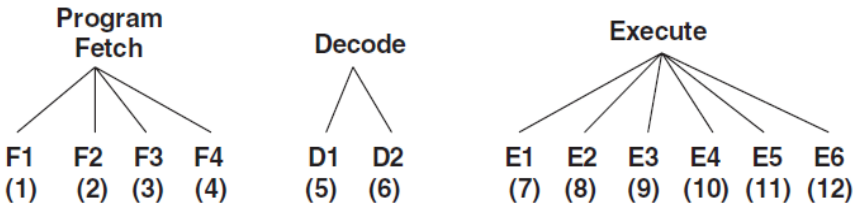


Fig. 3.7 : Les étapes du pipeline

Lorsque le résultat d'une instruction est utilisé par l'instruction suivante, un nombre approprié de NOP (aucune opération ou délai) doit être ajouté après la multiplication (un NOP), le chargement (quatre NOP / ou NOP 4) et le branchement (cinq NOP / ou NOP 5) afin de permettre au pipeline de fonctionner correctement. Par conséquent, pour que l'exemple ci-dessus s'exécute sur le processeur C6x, les NOP appropriés, comme le montre la figure 3.8, doivent être ajoutés après les instructions MPY, LDH et B.

```
          MVK   .S1    40,A2
loop:     LDH   .D1    *A5++,A0
          LDH   .D1    *A6++,A1
          NOP          4
          MPY   .M1    A0,A1,A3
          NOP
          ADD   .L1    A3,A4,A4
          SUB   .L1    A2,1,A2
[A2]      B     .S1    loop
          NOP          5
          STH   .D1    A4,*A7
```

Fig. 3.8 : Code pipeliné avec des NOP insérés

La figure 3.9 illustre un exemple de situation de pipeline qui nécessite l'ajout d'un NOP.

Les signes plus indiquent le nombre de sous-étapes ou de latences nécessaires à l'exécution de l'instruction. Dans cet exemple, on suppose que l'opération d'addition est effectuée avant que l'un de ses opérandes soit rendu disponible à partir de l'opération de multiplication précédente, d'où la nécessité d'ajouter un NOP après le MPY. Plus tard, on verra que dans le cadre de l'optimisation du code, les NOP peuvent être réduits ou supprimés conduisant à une amélioration de l'efficacité.

Recherche Programme	Décodage	Exécution						Terminé
F1-F4	D1-D2	E1	E2	E3	E4	E5	E6	
MPY								

MPY est recherché (fetched).

	MPY							
ADD								

MPY est décodée (decoded) et ADD est recherchée (fetched).

		MPY						
	ADD							

MPY est exécutée (executed) et ADD est décodée.

			MPY					
		ADD						

MPY est toujours en cours d'exécution tandis que ADD est également exécuté.

				→				MPY
				→				ADD

Les deux instructions se terminent en même temps, le résultat du MPY n'est pas utilisé dans l'instruction ADD.

(a)

Recherche Programme	Décodage	Exécution						Terminé
F1-F4	D1-D2	E1	E2	E3	E4	E5	E6	
MPY								

MPY est recherché (fetched).

	MPY							
NOP								

MPY est décodée et NOP est recherchée.

		MPY						
	NOP							
ADD								

MPY est exécutée, NOP est décodée et ADD est recherchée.

			MPY					
		NOP						
	ADD							

MPY est toujours en cours d'exécution tandis que NOP bloque le pipeline et ADD est décodé.

				→				MPY
		ADD						

MPY se termine, ADD est exécuté en utilisant le résultat de MPY.

				→				ADD

ADD se termine.

(b)

Fig. 3.9 : (a) Multiplication puis addition, et (b) besoin d'insertion de NOP.

3.4 Paquet de recherche (FP)

L'architecture C6x est basée sur l'architecture de mot d'instruction très long (VLIW). Dans une telle architecture, plusieurs instructions sont capturées et traitées simultanément. C'est ce qu'on appelle un paquet de recherche (FP). (Voir figure 3.10).

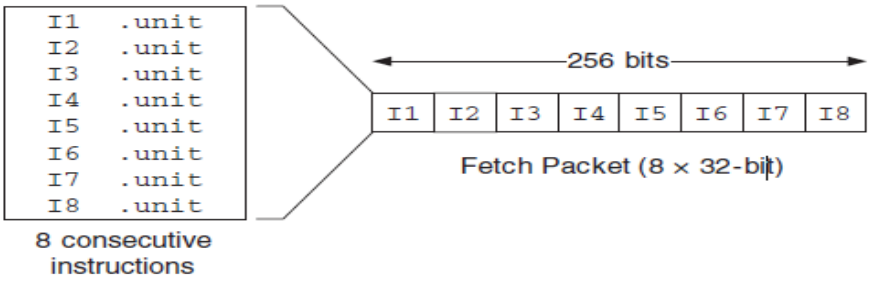


Fig. 3.10 : C6x fetch packet: le C6x recherche huit instructions de 32 bits à chaque cycle.

Le C6x utilise VLIW, permettant à huit instructions d'être capturées simultanément de la mémoire sur puce sur son bus de données de programme de 256 bits. L'architecture VLIW d'origine a été modifiée par TI(Texas Instrument) pour permettre à plusieurs soi-disant paquets d'exécution (EP) d'être inclus dans le même paquet de recherche, comme le montre la figure 3.11. Un EP constitue un groupe d'instructions parallèles. Les instructions parallèles sont indiquées par des symboles de double tube (||) et, comme leur nom l'indique, elles sont exécutées ensemble ou en parallèle.

Les instructions d'un EP se déplacent ensemble à chaque étape du pipeline. Cette modification VLIW est appelée VelociTI. Par rapport à VLIW, VelociTI réduit la taille du code et augmente les performances lorsque les instructions résident dans une puce extérieure.

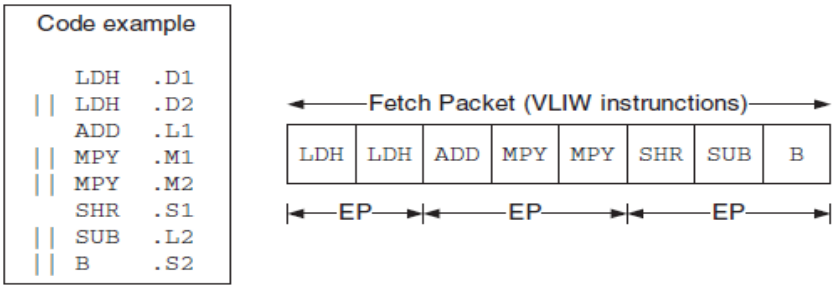


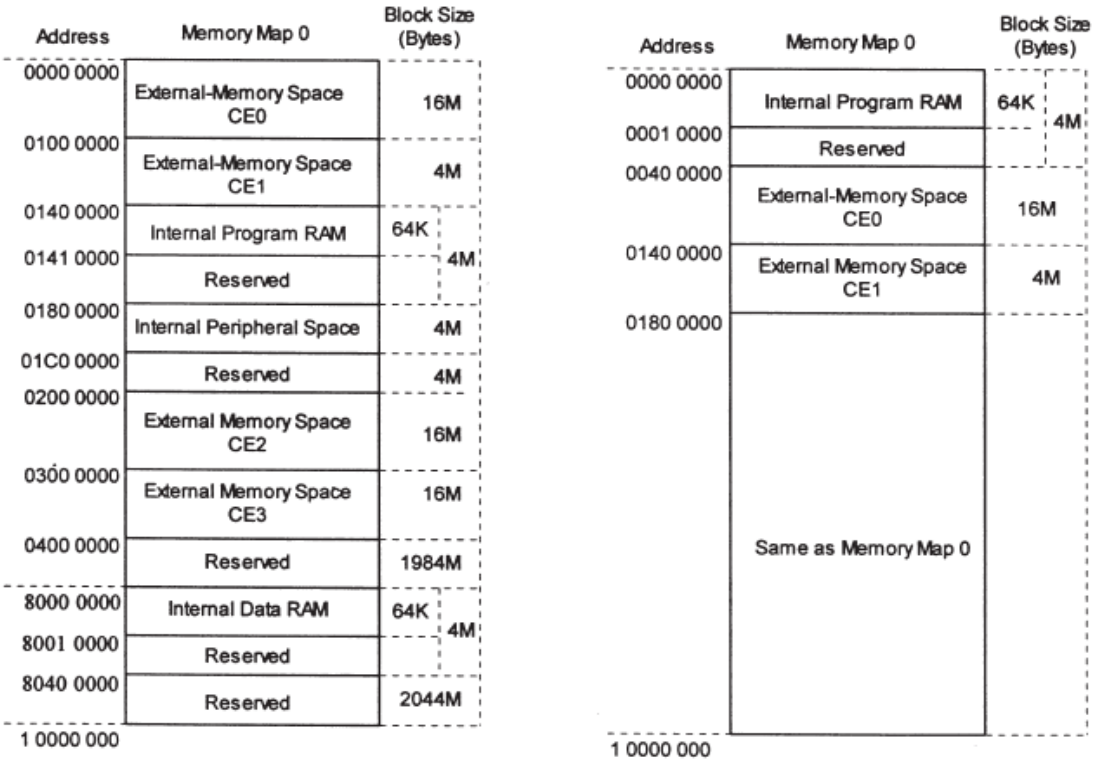
Fig. 3.11 : Un paquet de recherche contenant trois paquets d'exécution.

Chapitre 4 : Gestion de la mémoire

La mémoire externe utilisée par un processeur DSP peut être statique ou dynamique. La mémoire statique (SRAM) est plus rapide que la mémoire dynamique (DRAM), mais elle est plus chère, car elle prend plus de place sur le silicium. Les DRAM doivent également être actualisées périodiquement.

Un bon compromis entre coût et performances est obtenu en utilisant SDRAM (Synchronous DRAM). La mémoire synchrone nécessite une synchronisation, contrairement à la mémoire asynchrone, qui ne le fait pas.

Étant donné que le bus d'adresse a une largeur de 32 bits, l'espace mémoire total se compose de 2^{32} = 4 Go. Sur l'EVM (EValuation Module=DSK), cet espace est divisé, selon une carte mémoire, en mémoire de programme interne (PMEM), mémoire de données interne (DMEM), périphériques internes et espaces de mémoire externe nommés CE0, CE1, CE2 et CE3. Il existe deux configurations de carte mémoire : la carte mémoire 0 et la carte mémoire 1. Les figures 4.1 (a) et 4.1 (b) illustrent ces deux cartes mémoire. Sur le DSK (DSP Starter Kit), il n'y a pas de séparation entre le programme interne et la mémoire de données. Pour les exercices pratiques de ce polycopié, la carte EVM est configurée en fonction de sa carte mémoire 1, comme illustré à la figure 4.1 (c), et la carte DSK, en fonction de sa carte mémoire 1, comme illustré à la figure 4.1 (d) .



Address	Memory Map 0	Block Size (Bytes)
0000 0000	External-Memory Space CE0	16M
0100 0000	External-Memory Space CE1	4M
0140 0000	Internal Program RAM	64K
0141 0000	Reserved	4M
0180 0000	Internal Peripheral Space	4M
01C0 0000	Reserved	4M
0200 0000	External Memory Space CE2	16M
0300 0000	External Memory Space CE3	16M
0400 0000	Reserved	1984M
8000 0000	Internal Data RAM	64K
8001 0000	Reserved	4M
8040 0000	Reserved	2044M
1 0000 000		

(a)

Address	Memory Map 0	Block Size (Bytes)
0000 0000	Internal Program RAM	64K
0001 0000	Reserved	4M
0040 0000	External-Memory Space CE0	16M
0140 0000	External Memory Space CE1	4M
0180 0000	Same as Memory Map 0	
1 0000 000		

(b)

Address	Memory Map 1	Block Size (Bytes)
0000 0000	Internal Program RAM	64K
0001 0000	Reserved	4M
0040 0000	SBSRAM	256K
0044 0000	Unused	Unused
0140 0000	Asynchronous Expansion Memory	3M
0170 0000	PCI add-on registers	64
0170 0040	Unavailable	64K
0171 0000	PCI FIFO	4
0171 0004	Unavailable	64K
0172 0000	Audio Codec Registers	16
0172 0010	Unavailable	64K
0173 0000	Reserved	320K
0178 0000	DSP control/status registers	32
0178 0020	Unavailable	64K
0179 0000	Reserved	448K
0180 0000	Internal Peripheral Space	4M
01C0 0000	Reserved	4M
0200 0000	SDRAM (Bank 0)	4M
0240 0000	Reserved	12M
0300 0000	SDRAM (Bank 1)	4M
0340 0000	Reserved	12M
0400 0000	Reserved	1984M
8000 0000	Internal Data RAM	64K
8001 0000	Reserved	4M
8040 0000	Reserved	2044M
10000 0000		

(c)

Address	Memory Map 1	Block Size (Bytes)
0000 0000	Internal RAM (L2)	64K
0001 0000	Reserved	24M
0180 0000	EMIF control regs	32
0184 0000	Cache Configuration regs	4
0184 4000	L2 base addr & count regs	32
0184 4020	L1 base addr & count regs	32
0184 5000	L2 flush & clean regs	32
0184 8200	CE0 mem attribute regs	16
0184 8240	CE1 mem attribute regs	16
0184 8280	CE2 mem attribute regs	16
0184 82C0	CE3 mem attribute regs	16
0188 0000	HPI control regs	4
018C 0000	McBSP0 regs	40
0190 0000	McBSP1 regs	40
0194 0000	Timer0 regs	12
0198 0000	Timer1 regs	12
019C 0000	Interrupt selector regs	12
01A0 0000	EDMA parameter RAM	2M
01A0 FFE0	EDMA control regs	32
0200 0000	QDMA regs	20
0200 0020	QDMA pseudo-regs	20
3000 0000	McBSP0 data	64M
3400 0000	McBSP1 data	64M
8000 0000	CE0, SDRAM	16M
9000 0000	CE1, 8-bit ROM	128K
9008 0000	CE1, 8-bit I/O port	4
A000 0000	CE2-Daughtercard	256M
B000 0000	CE3-Daughtercard	256M
10000 0000		

(d)

Fig. 4.1 : (a) Carte mémoire C6x 0, (b) carte 1, (c) Carte EVM 1, et (d) Carte DSK 1.

Les plages de mémoire externe CE0, CE1, CE2 et CE3 prennent en charge la mémoire synchrone (SBSRAM, SDRAM) ou asynchrone (SRAM, ROM, etc.), accessible en octets (8 bits), en demi-mots (16 bits) ou en mots (32 bits). Les périphériques sur puce et les registres de contrôle sont mappés dans l'espace mémoire.

La mémoire de données interne est organisée en banques de mémoire afin que deux chargements ou stockages puissent être effectués simultanément. Tant que les données sont accessibles à partir de différentes banques, aucun conflit ne se produit. Cependant, si les données sont accessibles à partir de la même banque dans une instruction, un conflit de mémoire se produit et la CPU est bloquée par un cycle.

Si un programme s'intègre dans la mémoire interne ou sur la puce, il doit être exécuté à partir de là pour éviter les retards associés à l'accès à la mémoire externe ou à la puce. Si un programme est trop volumineux pour être inséré dans la mémoire interne, la plupart de ses portions chronophages doivent être placées dans la mémoire interne pour une exécution efficace. Pour les codes répétitifs, il est recommandé de configurer la mémoire interne comme mémoire cache.

Cela permet d'accéder à la mémoire externe aussi rarement que possible et donc d'éviter les retards associés à de tels accès.

4.1 Alignement de données en mémoire

Le C6x permet l'adressage d'octets, de demi-mots ou de mots. Consirérons une représentation en format Word de la mémoire, comme illustré à la figure 4.2. Il y a quatre limites d'octets, deux demi-mots (ou short) et une limite de mot par mot. Le C6x accède

toujours aux données sur ces limites en fonction du type de données spécifiées ; c'est-à-dire qu'il accède toujours aux données alignées. Lors de la spécification d'une section de variable non initialisée .usect, il est nécessaire de spécifier l'alignement ainsi que le nombre total d'octets.

Les exemples qui apparaissent à la figure 4.3 montrent l'alignement des données pour les constantes et les variables.

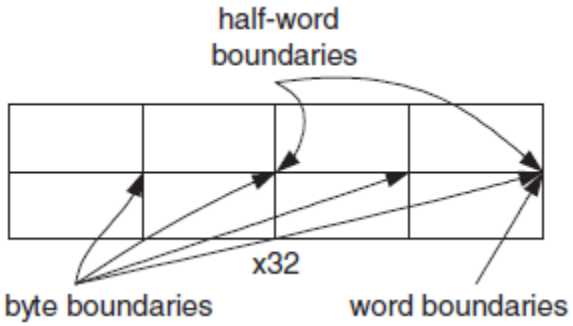


Fig. 4.2 : Limites des données

Constants are automatically aligned

```
    .sect "my_const"
A   .byte   11h
B   .short  2222h
C   .int    33333333h
```

Variables need an alignment field

```
;label    .usect  "section",  #bytes,  alignment
d         .usect  "vars",     1,       1
ee        .usect  "vars",     2          ;byte alignment by default
ffff      .usect  "vars",     4,       4
g_array   .usect  "vars",     8,       2
```

22	22	--	11
33	33	33	33
--	e	e	d
ff	ff	ff	ff
g1		g0	
g3		g2	

Note 1: vars and `my_const` sections are assumed contiguous.

Note 2: First declare words, then shorts and bytes to save memory space.

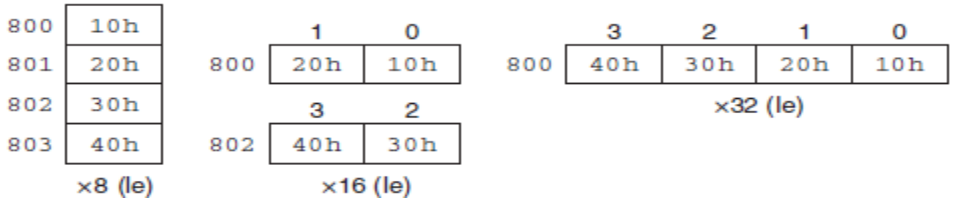
Fig. 4.3 : Exemples d'alignement de constantes et variables.

Les données en mémoire peuvent être organisées au format little-endian ou big-endian. Little-endian (le) signifie que l'octet le moins significatif est stocké en premier. La figure 4.4 (a) montre le stockage de .int 40302010h au format little-endian pour l'adressage d'accès octet, demi-mot et mot. Au format big-endian (be), illustré à la figure 4.4 (b), l'octet

le plus significatif est stocké en premier. Little-endian est le format normalement utilisé dans la plupart des applications.

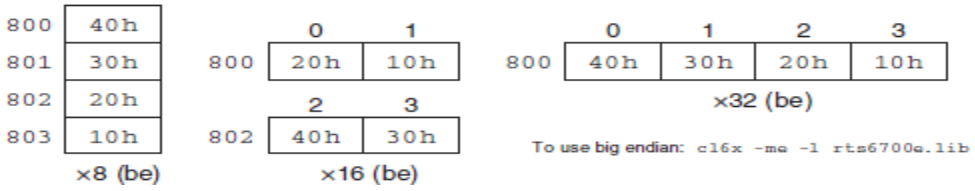
Des exemples supplémentaires d'alignement des données sont présentés sur la figure 4.4 (c), sur la base du format de données little-endian apparaissant dans la figure 4.4 (a).

Little endian (par défaut-LSB en premier)

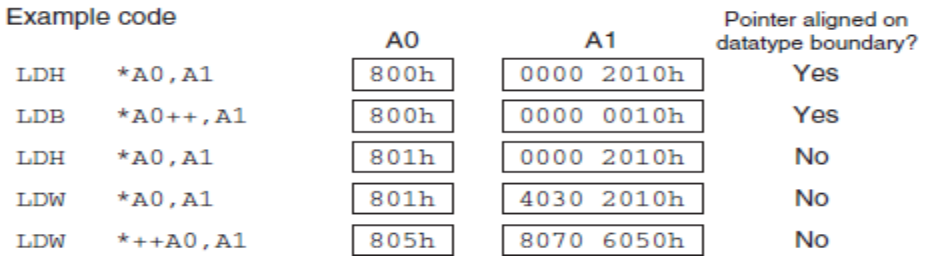


(a)

Big endian (MSB en premier)



(b)

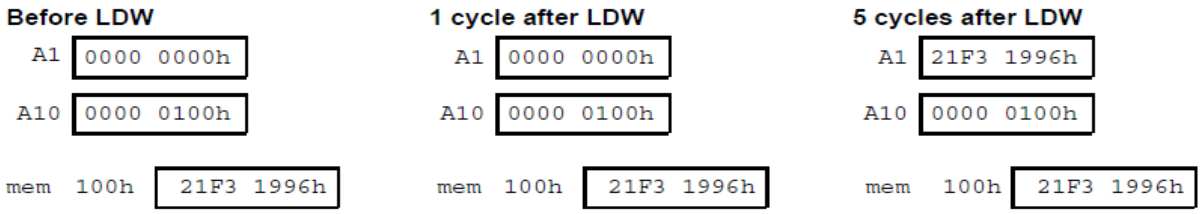


(c)

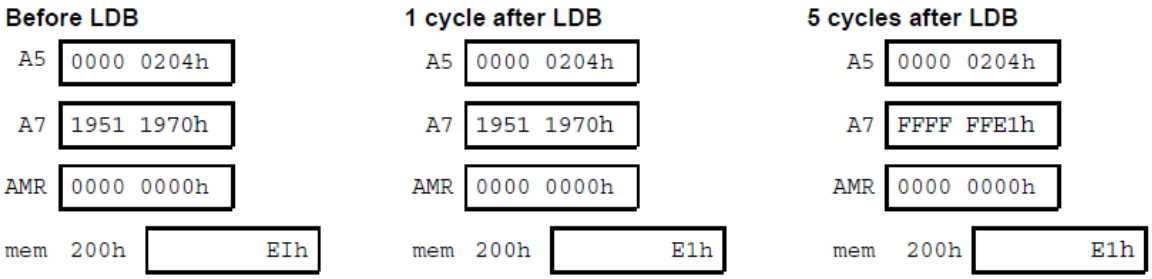
Fig. 4.4 : (a) Little endian, (b) big endian, et (c) plus d'exemples d'alignement de données.

4.2 Exemples d'alignements

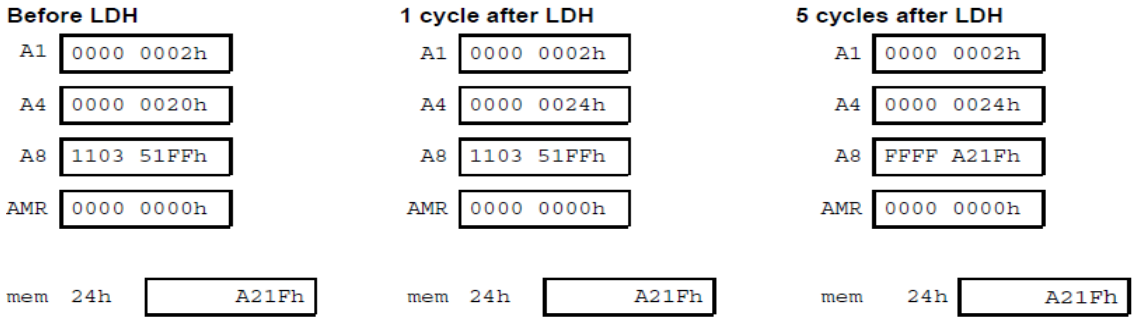
Exemple 1 : `LDW .D1 *A10,A1`



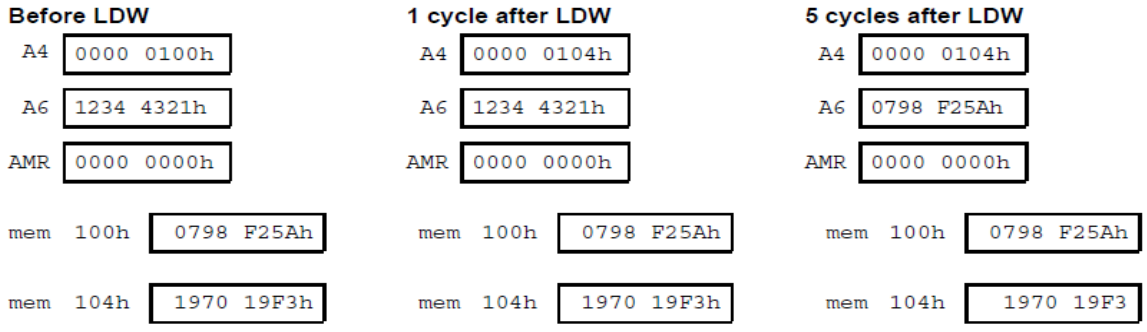
Exemple 2 : `LDB .D1 *-A5[4],A7`



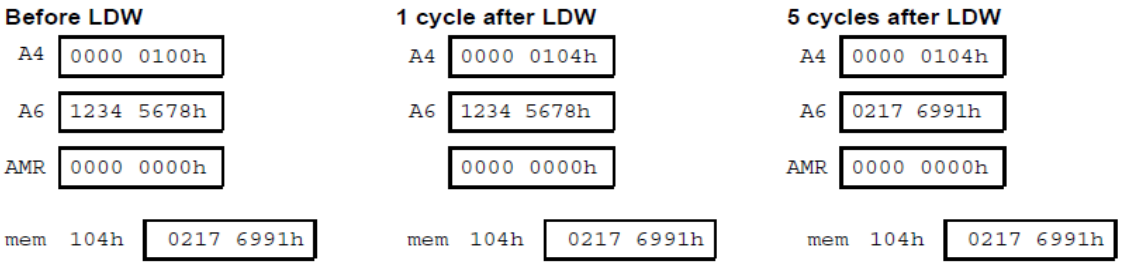
Exemple 3 : `LDH .D1 *++A4[A1],A8`



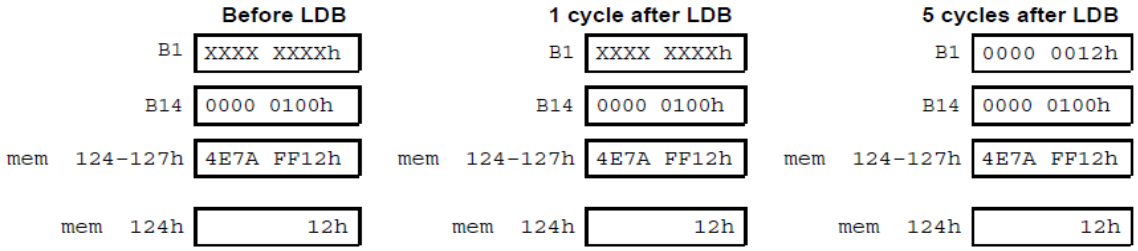
Exemple 4 : `LDW .D1 *A4,++[1],A6`



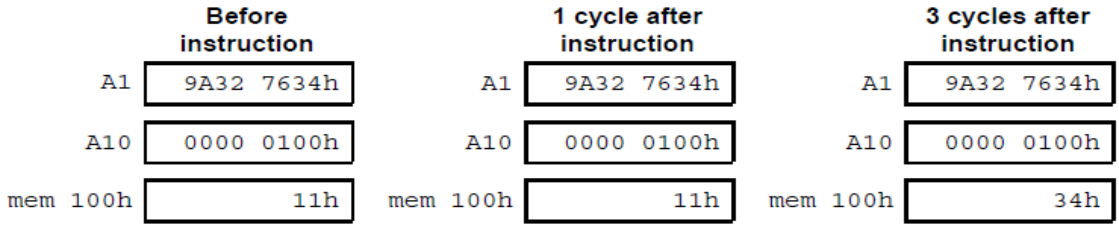
Exemple 5 : `LDW .D1 *++A4(4),A6`



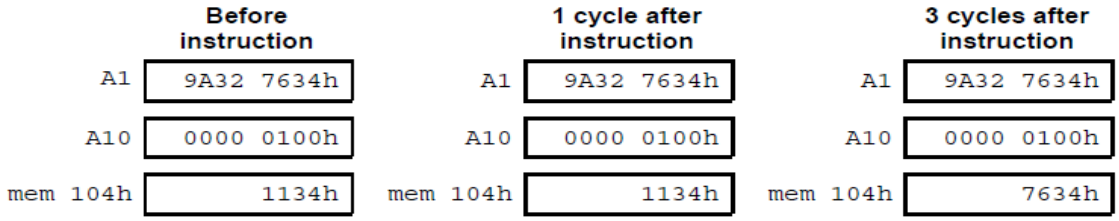
Exemple 6 : `LDB .D2 *+B14[36],B1`



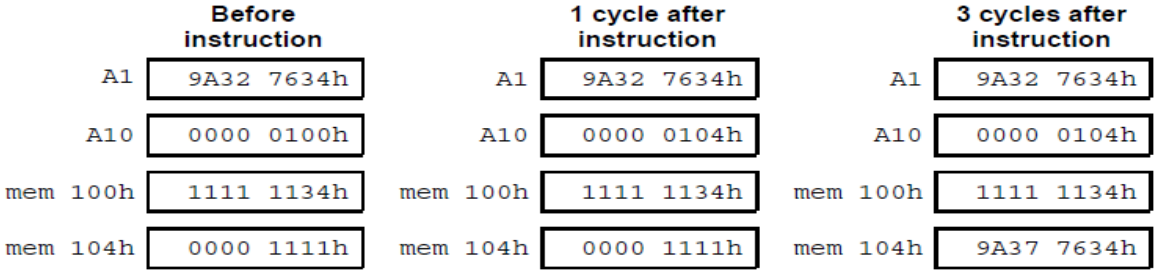
Exemple 1 : `STB .D1 A1,*A10`



Exemple 2 : `STH .D1 A1,*+A10(4)`



Exemple 3 : `STW .D1 A1,*++A10[1]`



Exemple 4 : `STH .D1 A1,*A10--[A11]`

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 2634h	9A32 2634h	9A32 2634h
A10	0000 0100h	0000 009Ch	0000 009Ch
A11	0000 0004h	0000 0004h	0000 0004h
mem 9Ch	0000h	0000h	0000h
mem 100h	0000	0000h	2634h

Exercice 5 : `STB .D2 B1,*+B14[40]`

	Before instruction	1 cycle after instruction	3 cycles after instruction
B1	1234 5678h	1234 5678h	1234 5678h
B14	0000 1000h	0000 1000h	0000 1000h
mem 1028h	42h	42h	78h

Chapitre 5 : Environnement de développement : 'Code Composer Studio' (CCS)

5.1 Logiciel et matériel requis

L'outil logiciel nécessaire pour générer des fichiers exécutables TMS320C6x s'appelle Code Composer Studio (CCS). CCS intègre les utilitaires assembleur, éditeur de liens, compilateur, simulateur et débogueur. En l'absence d'une carte cible, qui permet d'exécuter un fichier exécutable sur un processeur C6x réel, le simulateur peut être utilisé pour vérifier la fonctionnalité du code en utilisant des données déjà stockées dans un fichier de données. Cependant, lors de l'utilisation du simulateur, une routine de service d'interruption (ISR) ne peut pas être utilisée pour lire des échantillons de signal à partir d'une source de signal. Pour pouvoir traiter les signaux en temps réel sur un processeur C6x réel, un kit de démarrage DSP (DSK) ou une carte EVM (EValuation Module) est nécessaire pour le développement du code. L'équipement de test recommandé est un générateur de fonctions, un oscilloscope, un microphone, un boom box et des câbles avec prises audio.

Une carte DSK peut facilement être connectée à un PC hôte via son port parallèle ou USB. L'interface du signal avec la carte DSK se fait via ses deux prises audios standard. Une carte EVM doit être installée dans un emplacement PCI pleine longueur à l'intérieur d'un hôte PC. L'interface du signal avec la carte EVM se fait via ses trois prises audios standard.

Pour effectuer les travaux pratiques, la connaissance du langage C est supposée.

5.2 Outils logiciels

La programmation de la plupart des processeurs DSP peut être effectuée en C ou en assembleur. Bien que l'écriture de programmes en C nécessite moins d'efforts, l'efficacité obtenue est normalement inférieure à celle des programmes écrits en assembleur.

L'efficacité signifie avoir le moins d'instructions ou le moins de cycles d'instructions possible en utilisant au maximum les ressources de la puce.

En pratique, on commence par le codage en langage C pour analyser le comportement et la fonctionnalité d'un algorithme. Ensuite, si le taux de traitement requis n'est pas atteint à l'aide de l'optimiseur du compilateur C, les parties chronophages du code C sont identifiées et converties en assembleur, où le code entier est réécrit en assembleur. En plus du langage C et de l'assembleur, le C6x permet d'écrire le code en assembleur linéaire. La figure 5-1 illustre l'efficacité du code par rapport à l'effort de codage pour trois types de fichiers source sur le C6x: C, assembleur linéaire et assembleur optimisé à la main. Comme on peut le voir, l'assembleur linéaire offre un bon compromis entre l'efficacité du code et l'effort de codage.

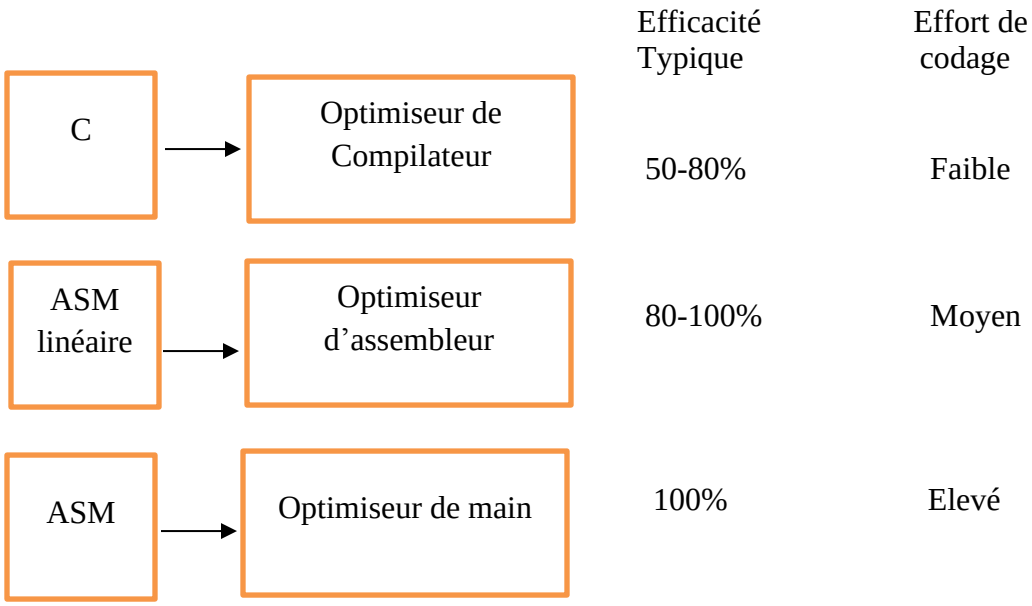
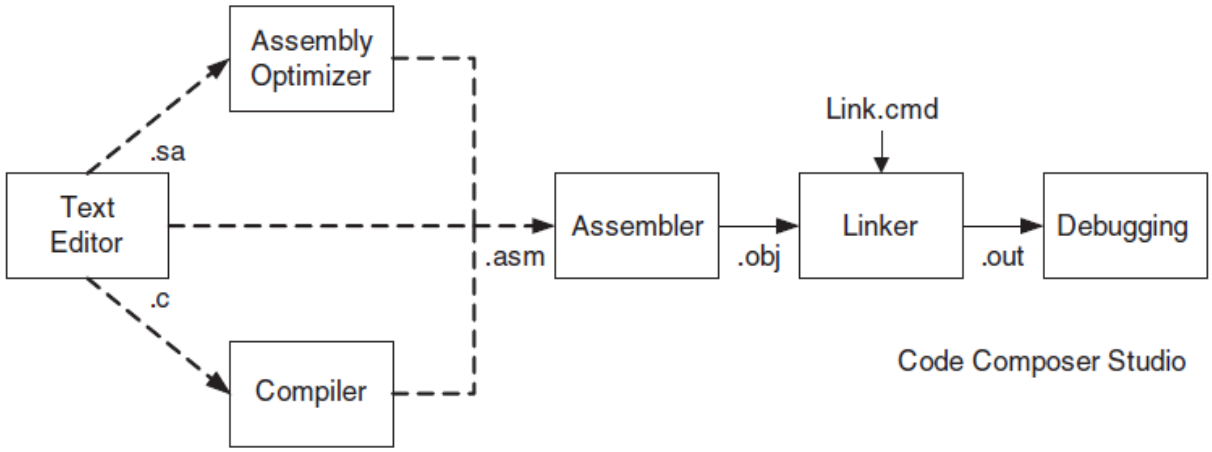


Fig. 5.1 : Efficacité du code et effort de codage.

La figure 5.2 montre les étapes à suivre pour passer d'un fichier source (extension .c pour C, .asm pour l'assembleurs et .sa pour l'assembleur linéaire) à un fichier exécutable (extension .out).



.c = fichier source C

.sa = fichier source d'assemblage linéaire

.asm = fichier source de l'assembly

.obj = fichier objet

.out = fichier exécutable

.cmd = fichier de commande de l'éditeur de liens

Fig. 5.2 : Outils logiciels du C6x

La figure 5.3 répertorie les versions .c et .sa de l'exemple de produit scalaire pour voir à quoi elles ressemblent. L'assembleur est utilisé pour convertir un fichier d'assembleur en un fichier objet (extension .obj). L'optimiseur d'assembleur et le compilateur sont utilisés pour convertir, respectivement, un fichier d'assembleur linéaire et un fichier C en un fichier objet. L'éditeur de liens est utilisé pour combiner des fichiers d'objets, comme indiqué par le fichier de commandes de l'éditeur de liens (extension .cmd), dans un fichier exécutable. Toutes les étapes d'assembleur, de liaison, de compilation et de débogage ont été intégrées dans un environnement de développement intégré (IDE) appelé Code Composer Studio

(CCS ou CCStudio). CCS fournit un environnement utilisateur graphique facile à utiliser pour créer et déboguer des codes C et d'assembleur sur divers DSP cibles.

```
void main()
{
    y = DotP( (int *) a, (int *) x, 40);
}

int DotP(int *m, int *n, short count)
{
    int sum, i;
    sum = 0;

    for(i=0;i<count;i++)
        sum += m[i] * n[i];

    return(sum);
  }
```

(a)

```
                .title "dot product"
                .def dotp
                .sect code
dotp: .proc   A4,B4,A6,B6,A8,B3
                .reg  a, ai, b,bi,r,prod,sum,c,ci,i;
                MV    A4,c
                MV    B4,b
                MV    A6,a
                MV    B6,r
                MV    A8,i
 loop:          .trip 40
                LDH   *a++, ai
                LDH   *b++,bi
                MPY    ai,bi,prod
                SHR    prod,15,sum
               ADD    ai,sum,ci
               STH    ci, *c++
        [i]    SUB    i,1,i
        [i]    B      loop
               .endproc B3
```

(b)

Fig. 5.3 : (a) .c (b) .sa version d'un exemple de produit scalaire.

5.3 Cartes cibles C6x DSK / EVM

Lors de la disponibilité d'une carte DSK ou EVM, un fichier exécutable peut être exécuté sur un processeur C6x réel. En l'absence de telles cartes, CCS peut être configuré pour simuler le processus d'exécution. Comme le montre la figure 5.4, la carte DSK C6713 est un système DSP qui comprend une puce DSP C6713 fonctionnant à 225 MHz avec une mémoire de 4/4/256 Ko pour le cache de données L1D / cache de programme L1P / mémoire L2, respectivement, 8 Mo de la SDRAM intégrée (RAM dynamique synchrone), 512 Ko de mémoire flash et un codec stéréo 16 bits AIC23 avec une fréquence d'échantillonnage de 8 kHz à 96 kHz. La carte DSK C6416 comprend une puce DSP C6416 fonctionnant à 600 MHz avec 16/16/1024 Ko de mémoire pour le cache de données L1D / cache de programme L1P / cache L2, respectivement, 16 Mo de SDRAM intégrée, 512 Ko de mémoire flash et codec AIC23. La carte DSK C6711 comprend une puce DSP C6711 fonctionnant à 150 MHz avec 4/4/64 Ko de mémoire pour le cache de données L1D / cache de programme L1P / cache L2, 16 Mo de SDRAM intégrée, 128 Ko de mémoire flash, un codec 16 bits AD535 ayant une fréquence d'échantillonnage fixe de 8 kHz et une interface de carte fille à laquelle une carte fille audio PCM3003 peut être connectée pour changer la fréquence d'échantillonnage.

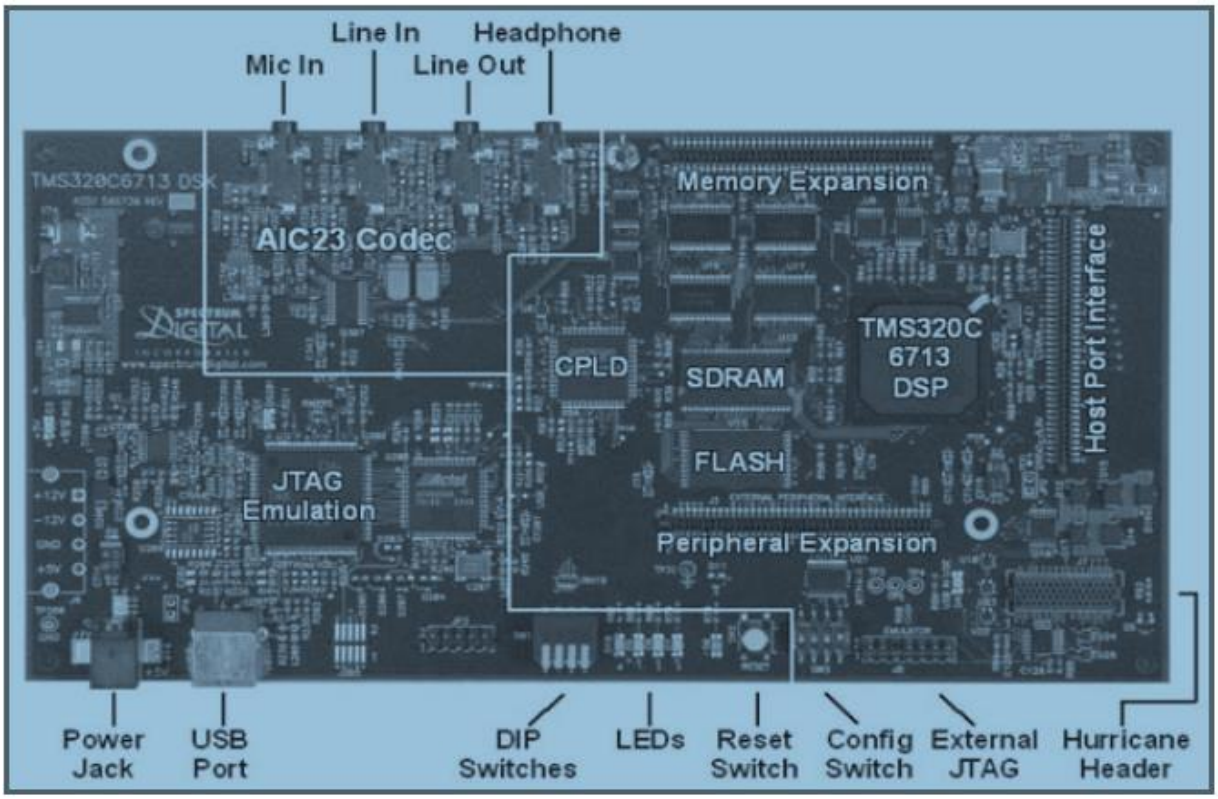
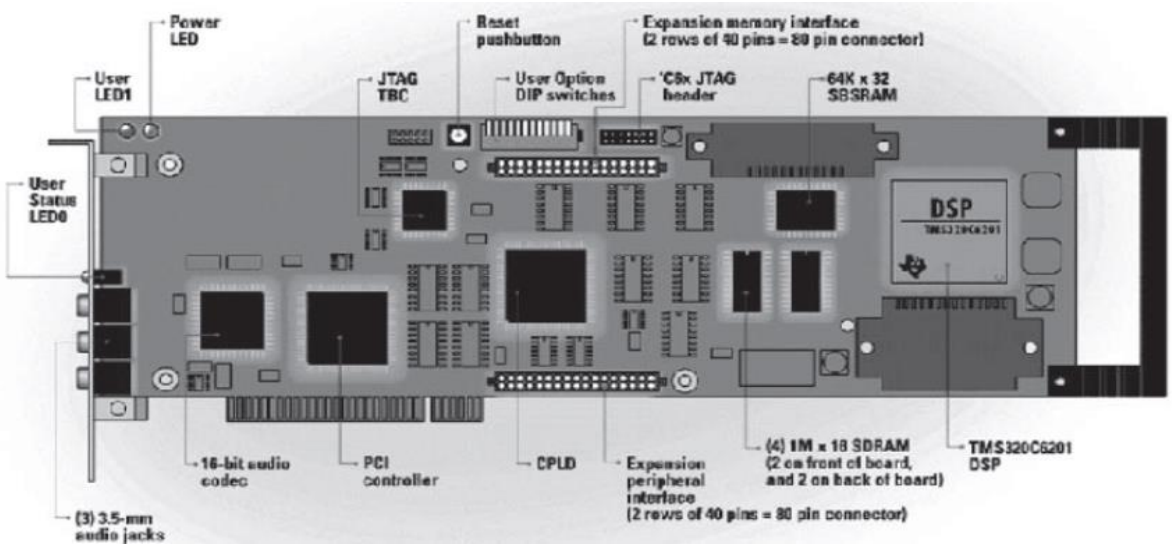
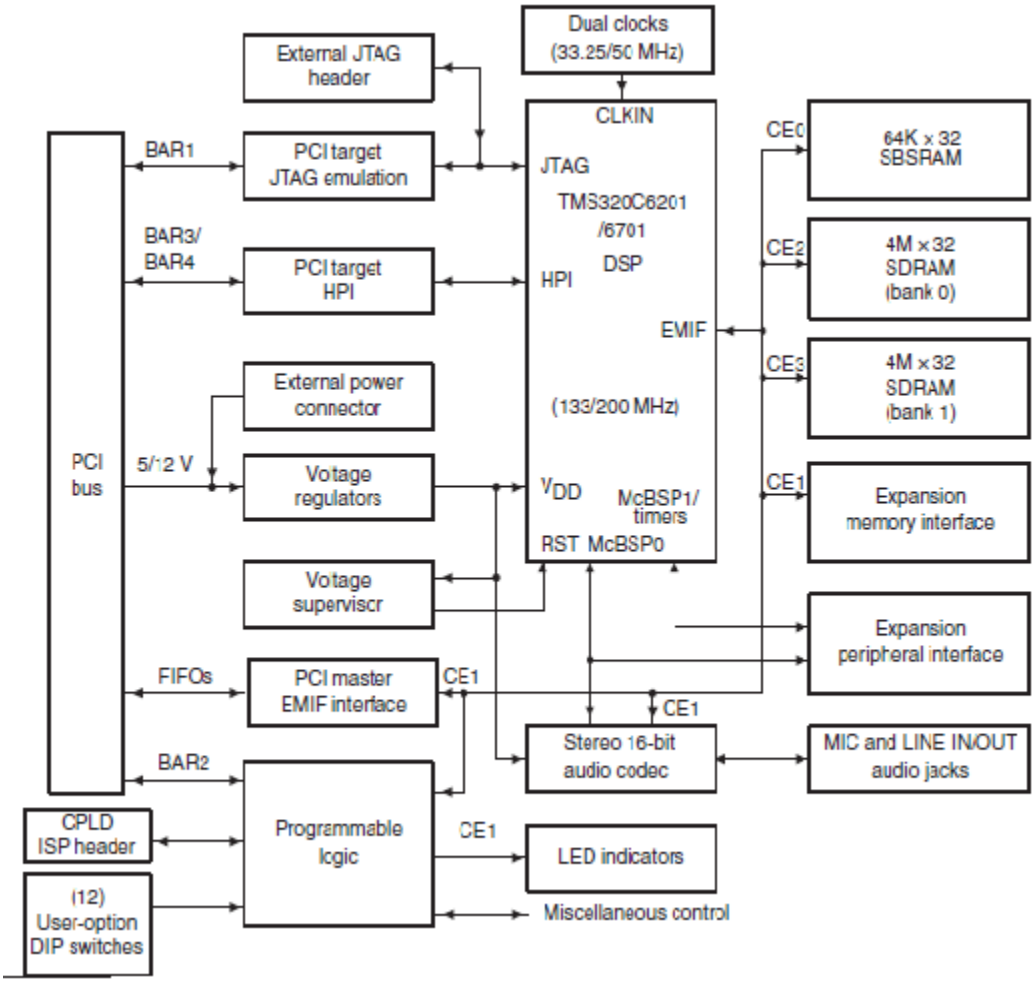


Fig. 5.4 : Carte DSK C6713.

Comme le montre la figure 5.5 (a), la carte EVM C6701 / C6201 est un système DSP qui comprend une puce C6701 (ou C6201), une mémoire externe, des capacités A / D et des composants d'interface hôte PC. Le schéma fonctionnel de la carte EVM apparaît à la figure 4-5 (b). La carte possède un codec 16 bits CS4231A dont la fréquence d'échantillonnage peut être modifiée de 5,5 kHz à 48 kHz.



(a)



(b)

Fig. 5.5 : (a) Carte EVM C6201/C6701, (b) son schéma fonctionnel.

La mémoire résidant sur la carte EVM se compose de 32 Mo de SDRAM fonctionnant à 100 MHz et de 256 Ko de SBSRAM (RAM statique synchrone en rafale) fonctionnant à 133 MHz, une mémoire plus rapide mais plus chère par rapport à la SDRAM. Un régulateur de tension sur la carte est utilisé pour fournir 1,8 V ou 2,5 V pour le noyau C6x et 3,3 V pour sa mémoire et ses périphériques, et 5 V pour les composants audio.

5.4 Fichier assembleur

Semblable à d'autres langages assembleurs, l'assembleur C6x se compose de quatre champs: étiquette, instruction, opérandes et commentaire. (Voir figure 3.3.) Le premier champ est le champ d'étiquette. Les étiquettes doivent commencer dans la première colonne et doivent commencer par une lettre. Une étiquette, si elle est présente, indique un nom attribué à un emplacement de mémoire spécifique qui contient une instruction ou des données. Un mnémonique ou une directive constitue le champ d'instruction. Il est facultatif que le champ d'instruction comprenne l'unité fonctionnelle qui exécute cette instruction particulière. Cependant, pour rendre les codes plus compréhensibles, l'affectation d'unités fonctionnelles est recommandée. Si une unité fonctionnelle est spécifiée, le chemin de données doit être indexé par 1 pour le côté A et 2 pour le côté B. Une instruction parallèle est indiquée par un symbole double pipe (||), et une instruction conditionnelle par un registre apparaissant entre parenthèses dans le champ d'instruction. Comme le nom de l'opérande l'indique, le champ d'opérande contient les arguments d'une instruction. Les instructions nécessitent deux ou trois opérandes. À l'exception des instructions de stockage, l'opérande de destination doit être un registre. L'un des opérandes source doit être un registre, l'autre un registre ou une constante. Après le champ opérande, il y a un champ de commentaire facultatif qui, s'il est indiqué, doit commencer par un point-virgule (;).

5.5 Directives

Les directives sont utilisées pour indiquer les sections de code d'assembleur et pour déclarer les structures de données. Il convient de noter que les instructions d'assembleur apparaissant comme des directives ne produisant aucun code exécutable. Elles contrôlent simplement le processus de l'assembleur par l'assembleur. Certaines des directives assembleur largement utilisées sont :

Directive .sect "nom", qui définit une section de code ou de données nommée "nom".

Directive .int, .long ou .word, qui réserve 32 bits de mémoire initialisés à une valeur.

Directive .short ou .half, qui réserve 16 bits de mémoire initialisés à une valeur.

Directive .byte, qui réserve 8 bits de mémoire initialisés à une valeur.

Notez que dans le format de fichier d'objet commun TI (COFF), les directives .text, .data, .bss sont utilisées pour indiquer le code, les données constantes initialisées et les variables non initialisées, respectivement. D'autres directives souvent utilisées incluent la directive .set, pour attribuer une valeur à un symbole, la directive .global ou .def, pour déclarer un symbole ou un module comme global afin qu'il puisse être reconnu en externe par d'autres modules, et la directive .end, pour signaler la résiliation du code d'assembleur. La directive .global agit comme une directive .def pour les symboles définis et comme une directive .ref pour les symboles non définis.

À ce stade, il convient de mentionner que le compilateur C crée diverses sections indiquées par les directives .text, .switch, .const, .cinit, .bss, .far, .stack, .sysmem, .cio. La figure 5.6 répertorie certaines sections courantes du compilateur.

Nom de la section	description
.text	Code
.switch	Tableaux pour les instructions de commutation
.const	Littéraux de chaîne globaux et statiques
.cinit	Valeurs initiales pour les variables globales / statiques
.bss	Variables globales et statiques
.far	Global et statique déclarés plus loin
.stack	Stack (variables locales)
.sysmem	Mémoire pour les mallc fcns (tas)
.cio	Tampons pour les fonctions stdio

Fig. 5.6 : Sections communes du compilateur.

5.6 Utilitaire de compilation

La fonction de génération de CCS peut être utilisée pour effectuer l'ensemble du processus de compilation, d'assemblage et de liaison en une seule étape via l'activation de l'utilitaire cl6x et en indiquant les bonnes options. La commande suivante montre comment cet utilitaire est utilisé dans CCS pour créer les fichiers source file1.c, file2.asm et file3.sa:

`cl6x -gs` *file1.c file2.asm file3.sa* `-z -o` *file.out* `-m` *file.map* `-l` *rts6700.lib*

L'option -g ajoute des informations spécifiques au débogueur au fichier objet à des fins de débogage. L'option -s fournit une inter-liste de C et d'assembleur. Pour file1.c, le compilateur C, pour file2.asm l'assembleur et pour file3.sa, l'optimiseur d'assemblage (assembleur linéaire) sont appelés. L'option –z appelle l'éditeur de liens, plaçant le code exécutable dans file.out si l'option -o est utilisée. Sinon, le fichier par défaut a.out est créé.

L'option -m fournit un fichier de carte (file.map), qui comprend une liste de toutes les adresses des sections, symboles et étiquettes. L'option -l spécifie la bibliothèque de prise en charge au moment de l'exécution rts6700.lib pour la liaison de fichiers sur le processeur C6713. Le tableau 5.1 répertorie certaines options fréquemment utilisées.

Options	Description	Moyen
-mv6700	Génére le 'code C67x (' C62x est la valeur par défaut)	Comp/Asm
-g	Active le débogage symbolique au niveau src	Comp/Asm
--mg	Permet un débogage minimum pour permettre le profilage	Compilateur
-s	Inscrit des instructions C dans la liste d'assembleur	Compilateur
-o	Invoque l'optimiseur (-o0, -o1, -o2 / -o, -o3)	Compilateur
-pm	Combine tous les fichiers source C avant de les compiler	Compilateur
-mt	Aucun alias utilisé	Compilateur
-ms	Réduit la taille du code (-ms0 / -ms, -ms1, -ms2)	Compilateur
-z	Appelle l'éditeur de liens	Éditeur de liens
-o	Nom du fichier de sortie	Éditeur de liens
-m	Nom du fichier de carte	Éditeur de liens
-c	Variables C Auto -Init (-cs désactive l'autoinit)	Éditeur de liens
-l	Bibliothèques Link -in (petit -L)	Éditeur de liens

Tableau 5.1 : Options courantes du compilateur

Le compilateur permet d'appeler quatre niveaux d'optimisations en utilisant -o0, -o1, -o2, -o3. Le débogage et l'optimisation à grande échelle ne peuvent pas être effectués ensemble, car ils tombent en conflit; c'est-à-dire que dans le débogage, des informations sont ajoutées pour améliorer le processus de débogage, tandis que dans l'optimisation, les informations sont minimisées ou supprimées pour améliorer l'efficacité du code. En principe, l'optimiseur modifie le flux du code C, ce qui rend le débogage du programme très difficile.

Comme le montre la figure 4-9, une bonne approche de programmation serait d'abord de vérifier que le code fonctionne correctement en utilisant le compilateur sans optimisation (option -gs). Ensuite, utilisez l'optimisation complète pour générer un code efficace (option

-o3). Il est recommandé de prendre une étape intermédiaire dans laquelle une optimisation est effectuée sans interférer avec le débogage au niveau source (option -go). Cette étape intermédiaire peut revérifier la fonctionnalité du code avant d'effectuer une optimisation complète. Il convient de noter qu'une optimisation complète peut modifier les emplacements de mémoire en dehors de la portée du code C. Ces emplacements de mémoire doivent être déclarés « volatils » pour éviter les erreurs de compilation.

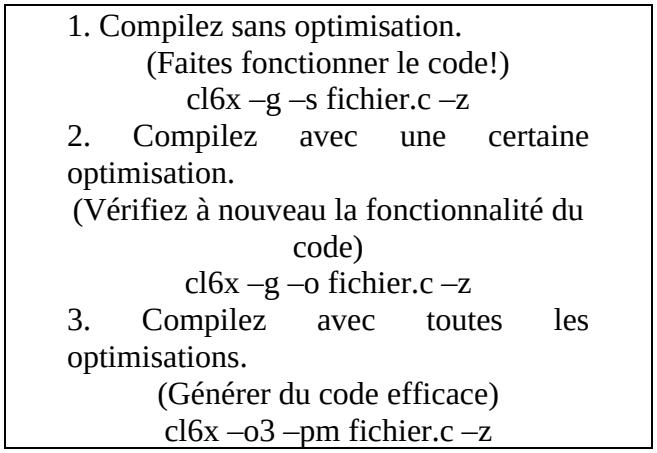


Fig. 5.7 : Principe de programmation

Afin d'optimiser davantage les codes C, il est recommandé d'utiliser les intrinsèques dans la mesure du possible. Les intrinsèques sont des fonctions similaires aux fonctions mathématiques faisant partie de la bibliothèque de support d'exécution. Les intrinsèques permettent au compilateur C d'accéder directement au matériel tout en préservant l'environnement C. Par exemple, au lieu d'utiliser l'opérateur multiply * en C, le _mpy () intrinsèque peut être utilisé pour dire au compilateur d'utiliser l'instruction C6x MPY. La figure 5.8 montre la version intrinsèque du code C du produit scalaire. Une liste des caractéristiques intrinsèques du C6x est fournie à l'annexe A de [1]122.

```
short DotP(int *m, int *n, short count)
{
      short i, productl, producth, suml = 0, sumh = 0;

     for(i=0; i<count; i++)
    {
            productl = _mpy(m[i],n[i]); // _mpy intrinsic
           producth = _mpyh(m[i],n[i]); // _mpyh intrinsic
          suml += productl;
         sumh += producth;
    }
    suml += sumh;
    return(suml);
}
```

Fig. 5.8 : Version intrinsèque du code C du produit scalaire (dot.product)

5.7 Initialisation du code

Tous les programmes commencent par un code d'initialisation réinitialisé. La figure 5.9 illustre à la fois la version C et assembleur d'un code d'initialisation de réinitialisation typique. Cette initialisation a pour but de commencer à un emplacement initial précédemment défini. À la mise sous tension, le système se rend toujours à l'emplacement de réinitialisation en mémoire, qui comprend normalement une instruction de branchement au début du code à exécuter. Le code de réinitialisation illustré à la figure 5.9 amène le compteur de programmes à un emplacement défini globalement dans la mémoire nommé init ou _c_int00.

« **ASM** »

```
vectors.asm

        .ref init
        .sect “vectors”
rst     MVK .s2 init,B0
        MVKH .s2 init,B0
        B .s2 B0
        NOP
        NOP
        NOP
        NOP
        NOP
```

« **C** »

```
cvectors.asm

        .global _c_int00
        .sect “vectors”
rst     B _c_int00
        NOP ;additional NOP’s
        NOP ;to create a
        NOP ;fetch packet
        NOP
        NOP
        NOP
        NOP
```

Fig. 5.9 : Code de réinitialisation

Comme indiqué dans la figure 5.10, lors de l'écriture en assembleur, un code d'initialisation est nécessaire pour créer des données et des variables initialisées et pour copier des données initialisées dans des variables correspondantes. Les valeurs initialisées sont spécifiées à l'aide des directives .byte, .short ou .int. Les variables non initialisées sont spécifiées à l'aide de la directive .usect.

Les premier, deuxième et troisième arguments de cette directive indiquent respectivement le nom de la section, la taille en octets et l'alignement des données en octets. Avant d'appeler la fonction principale ou le sous-programme, une autre partie du code d'initialisation est généralement nécessaire pour configurer les registres et les pointeurs et pour déplacer les données vers les emplacements appropriés en mémoire.

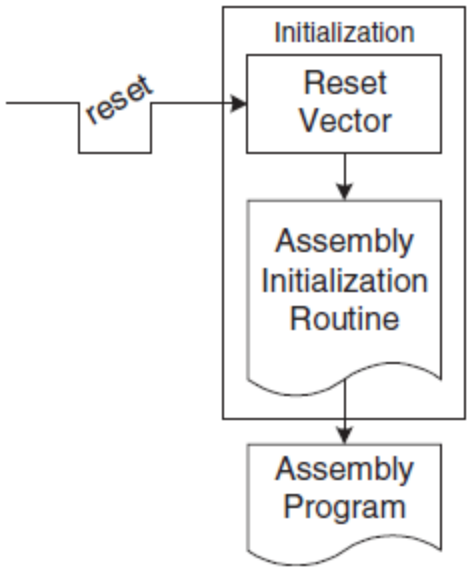


Fig. 5.10 : Initialisation de l'assembleur

La figure 5.11 représente le code d'initialisation de l'exemple de produit scalaire (dot-product) dans lequel des valeurs de données initialisées apparaissent pour trois tableaux de données initialisés intitulés table_a, table_x et table_y. De plus, trois sections de variables appelées a, x et y sont déclarées. La deuxième partie du code d'initialisation copie les données initialisées dans les variables correspondantes. Le code de configuration pour appeler la routine de produit scalaire est également illustré dans cette figure.

```
        .def    init
        .ref    dotp

;Data initialization
;Initialize tables

        .sect   "init_tables"

table_a .short  40,39,38,37,36,35,34,33,32,31,30,29,28,27   ;Initialize table_a array with values
        .short  26,25,24,23,22,21,20,19,18,17
        .short  16,15,14,13,12,14,10,9,8,7,6,5,4,3,2,1
table_x .short  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15         ;Initialize table_x array with values
        .short  16,17,18,19,20,21,22,23,24,25,26,27,28,29
        .short  30,31,32,33,34,35,36,37,38,39,40
table_y .short  0                                           ;table_y = 0

;Variable declaration
a       .usect  "var", 80, 2                                ;define variables
x       .usect  "var", 80, 2
y       .usect  "var", 2, 2

;Initialization to copy data into variables
        .sect   "init_code"
init    mvk     .s1     table_a, A0     ;move address of table_a to register A0
        mvkh    .s1     table_a, A0
        mvk     .s2     a,B0            ;move address of a to register B0
        mvkh    .s2     a,B0
        mvk     .s2     40,B1           ;create a counter in register B1, B1=40
loop_a  ldh     .d1     *A0++,A1        ;load an element from the address pointed by A0 into A1
        sub     .l2     B1,1,B1         ;decrement counter
        nop             3
        sth     .d2     A1,*B0++        ;store the element to address pointed by B0
   [B1] b       .s2     loop_a          ;branch back to loop_a
        nop             5               ;required latency
init_x  mvk     .s1     table_x, A0     ;move address of table_x into register A0
        mvkh    .s1     table_x, A0
        mvk     .s2     x, B0           ;move address of x into register A0
        mvkh    .s2     x, B0
        mvk     .s2     40, B1          ;create a counter
loop_x  ldh     .d1     *A0++,A1        ;load an element from the address pointed by A0 into A1
        sub     .l2     B1,1,B1         ;decrement counter
        nop             3
        sth     .d2     A1,*B0++        ;store element to address pointed by B0
   [B1] b       .s2     loop_x          ;branch back to loop_x
        nop             5
init_y  mvk     .s1     table_y, A0     ;repeat above procedure for table_y
        mvkh    .s1     table_y, A0
        mvk     .s2     y, B0
        mvkh    .s2     y, B0
        ldh     .d1     *A0, A1
        nop             4
        sth     .d2     A1, *B0
```

(a)

```
;Setup for calling dotp

start   mvk     .s1     a,A4            ;move a into register A4
        mvkh    .s1     a,A4
        mvk     .s2     x,B4            ;move x into register B4
        mvkh    .s2     x,B4
        mvk     .s1     40,A6           ;create a counter in A6, A6=40
        b       .s1     dotp            ;branch to routine dotp
        mvk     .s2     return, B3      ;store return address in B3
        mvkh    .s2     return, B3
        nop             3

;return from dotp here
return  mvk     .s1     y, A0           ;move y into register A0
        mvkh    .s1     y, A0
        sth     .d1     A4, *A0         ;store the result of dotp (returned in A4) to y

;infinite loop
end     b       .s1     end             ;infinite loop
        nop             5
```

(b)

```
;dotp
        .def    dotp

;A4 = &a, B4 = &x, A6 = 40 (iteration count) , B3 = return address

dotp    mv              A6,B0       ;move A6 to B0 (third argument passed from calling function)
        zero            A2          ;zero the sum register A2

loop    ldh     .d1     *A4++,A5    ;load an element from the location pointed by A4 into A5
        ldh     .d2     *B4++,B5    ;load an element from the location pointed by B4 into B5
        nop             4
        mpy     .m1x    A5,B5,A5    ;A5=B5*A5
        nop
        add     .l1     A2,A5,A2    ;A2 += A5
    [B0] sub    .l2     B0,1,B0     ;decrement counter B0
    [B0] b      .s1     loop        ;branch back to loop
        nop             5

        mv              A2,A4       ;move result in A2 to return register A4
        b       .s2     B3          ;branch back to calling address stored in B3
        nop             5
```

(c)

Fig. 5.11 : (a) Code d'initialisation pour l'exemple de produit scalaire, (b) code de configuration pour appeler la routine de produit scalaire, et (c) routine de produit scalaire.

En ce qui concerne le codage C, le compilateur C utilise boot.c dans la bibliothèque de support d'exécution pour effectuer l'initialisation avant d'appeler main (). L'option –c active boot.c pour initialiser automatiquement les variables. Ceci est illustré dans la figure 5.12.

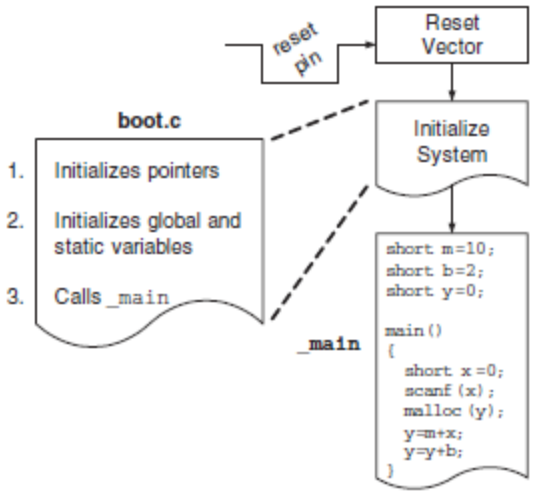


Fig. 5.12 : Initialisation du langage C

5.8 Code Composer Studio (Lab1)

Code Composer Studio ™ (CCStudio ou CCS) est un environnement de développement intégré (IDE) utile qui fournit un outil logiciel facile à utiliser pour créer et déboguer des programmes. De plus, il permet une analyse en temps réel des programmes d'application. La figure 5.13 montre les phases associées au processus de développement du code CCS. Lors de sa configuration, CCS peut être configuré pour différentes cartes DSP cibles (par exemple, C6711 DSK, C6416 DSK, C6701 EVM, C6xxx Simulator). La version utilisée tout au long de ce manuel est basée sur la version CCS 2.2.

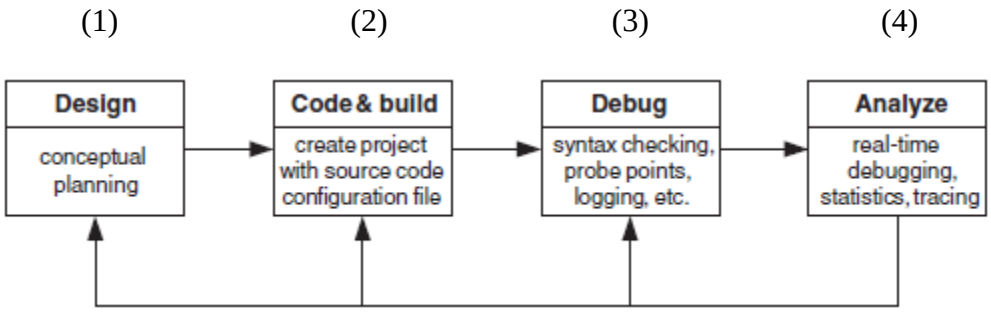


Fig. 5.13 : Processus de développement du code CCS.

(1) Conception : Planification conceptuelle

(2) Code & construction : créer un projet avec un fichier de configuration du code source

(3) Débogage : vérification de la syntaxe, points de sonde, journalisation, etc.

(4) Analyse : débogage en temps réel, statistiques, traçage

CCS fournit un environnement de gestion de fichiers pour la création de programmes d'application. Il comprend un éditeur intégré pour éditer les fichiers C et d'assemblage. À des fins de débogage, il fournit des points d'arrêt, des capacités de surveillance et de représentation graphique des données, un profileur pour l'analyse comparative et des points de sonde pour diffuser des données vers et depuis le DSP cible. Ce didacticiel présente les fonctionnalités de base de CCS qui sont nécessaires pour créer et déboguer un programme d'application.

Ce laboratoire montre comment un algorithme multifichier simple peut être compilé, assemblé et lié à l'aide de CCS. Tout d'abord, plusieurs valeurs de données sont écrites consécutivement dans la mémoire. Ensuite, un pointeur est affecté au début des données afin qu'elles puissent être traitées comme un tableau. Enfin, des fonctions simples sont ajoutées en C et en assembleur pour illustrer le fonctionnement de l'appel de fonction. Cette méthode de placement de données en mémoire est simple à utiliser et peut être utilisée dans des applications dans lesquelles les constantes doivent être en mémoire, comme les coefficients de filtre et les facteurs de torsion FFT. Les problèmes liés au débogage et à l'analyse comparative sont également traités dans ce laboratoire.

5.9 Création des projets (Lab 1.1)

Considérons tous les fichiers nécessaires pour créer un fichier exécutable ; c'est-à-dire, les fichiers source .c (c), .asm (assembleur), .sa (assembleur linéaire), un fichier de commande de l'éditeur de liens .cmd, un fichier d'en-tête .h et les fichiers de bibliothèque .lib appropriés. Le processus de développement du code CCS commence par la création d'un soi-disant projet pour intégrer et gérer facilement tous ces fichiers requis pour générer et exécuter un fichier exécutable. Le panneau **Project View** sur le côté gauche de la fenêtre CCS fournit un mécanisme simple pour le faire. Dans ce panneau, un fichier de projet (extension .prj) peut être créé ou ouvert pour contenir non seulement les fichiers source et de bibliothèque, mais également les options du compilateur, de l'assembleur et de l'éditeur de liens pour générer un fichier exécutable. Par conséquent, il n'est pas nécessaire de taper des lignes de commande pour la compilation, l'assemblage et la liaison, comme c'était le cas avec les outils de développement logiciel d'origine.

Pour créer un projet, choisissez l'élément de menu **Project → New** dans la barre de menus CCS. Cela fait apparaître la boîte de dialogue **Project Création**, comme le montre la figure 5.14. Dans la boîte de dialogue, accédez au dossier de travail, dans tout le livre supposé être C: \ ti \ myprojects, et tapez un nom de projet dans le champ **Project Name**. Cliquez ensuite sur le bouton **Finish** pour CCS pour créer un fichier de projet appelé lab1.prj. Tous les fichiers nécessaires à la création d'une application doivent être ajoutés au projet.

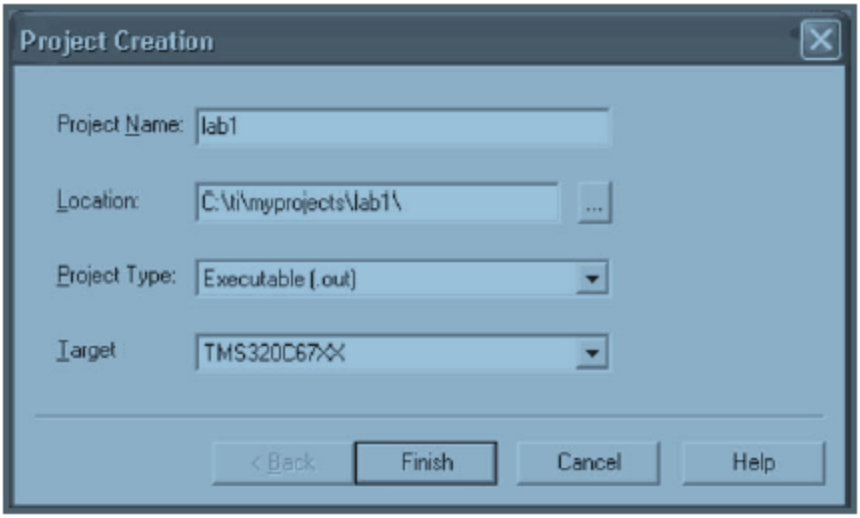


Fig. 5.14 : Création d'un nouveau projet.

CCS fournit un éditeur intégré qui permet la création de fichiers sources. Certaines des fonctionnalités de l'éditeur sont la mise en évidence de la syntaxe des couleurs, le marquage des blocs C entre parenthèses et accolades, la correspondance entre parenthèses / accolades, les indentations de contrôle et les capacités de recherche / remplacement / recherche. Il est également possible d'ajouter des fichiers au projet à partir de l'Explorateur Windows en utilisant l'approche glisser-déposer. Une fenêtre d'édition s'affiche en choisissant l'élément de menu **File → New → Source File**. Pour cet atelier, saisissons le code d'assembleur suivant dans la fenêtre de l'éditeur :

```
.sect       ".mydata"
.short      0
.short      7
.short      10
.short      7
.short      0
.short      -7
.short      -10
.short      -7
.short      0
.short      7
```

Ce code consiste en la déclaration de 10 valeurs à l'aide de directives .short. Notez que n'importe laquelle des directives d'allocation de données à mémoire peut être utilisée pour faire de même. Attribuez une section nommée .mydata aux valeurs à l'aide d'une directive .sect. Enregistrez le fichier source créé en choisissant l'élément de menu **File →Save**. Cela fait apparaître la boîte de dialogue **Save As**, comme le montre la figure 5.15. Dans la boîte de dialogue, accédez au champ **Save as type** et sélectionnez **Assembly Source Files** (*.asm) dans la liste déroulante. Ensuite, accédez au champ **File name** et tapez initmem.asm. Enfin, cliquez sur **Save** pour que le code puisse être enregistré dans un fichier source d'assembly nommé initmem.asm.

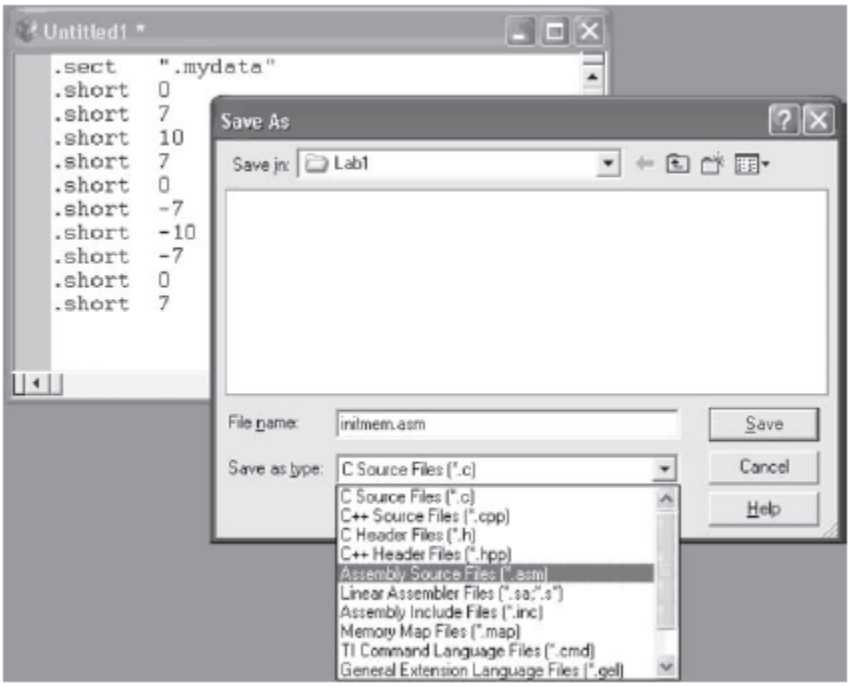


Fig. 5.15 : Création d'un fichier source

En plus des fichiers source, un fichier de commande de l'éditeur de liens doit être spécifié pour créer un fichier exécutable et pour se conformer aux spécificités de la mémoire du DSP cible sur lequel le fichier exécutable va s'exécuter. Un fichier de commandes de

l'éditeur de liens peut être créé en choisissant **File → New → Source File**. Pour cet atelier, saisissons le fichier de commandes illustré à la figure 5.16.

Ce fichier peut également être téléchargé à partir du CD-ROM fourni. Ce fichier de commandes de l'éditeur de liens est configuré en fonction de la carte mémoire DSK. Puisque notre intention est de placer le tableau de valeurs définies dans initmem.asm dans la mémoire, un espace qui ne sera pas écrasé par le compilateur devrait être sélectionné. L'espace mémoire externe CE0 peut être utilisé à cet effet. Assemblons les données à l'adresse mémoire 0x80000000 (0x désigne hex) située au début de CE0. Pour ce faire, affectez la section nommée .mydata à MYDATA en ajoutant .mydata> MYDATA à la partie SECTIONS du fichier de commandes de l'éditeur de liens, comme le montre la figure 5.16. Enregistrez la fenêtre de l'éditeur dans un fichier de commandes de l'éditeur de liens en choisissant **File → Save** ou en appuyant sur Ctrl + S. La boîte de dialogue **Save As** s'affiche. Accédez au champ Save as type et sélectionnez **TI Command Language Files** (* .cmd) dans la liste déroulante. Tapez ensuite lab1.cmd dans le champ Nom de fichier et cliquez sur **Save**.

Maintenant que le fichier source initmem.asm et le fichier de commandes de l'éditeur de liens lab1.cmd sont créés, ils doivent être ajoutés au projet pour l'assemblage et la liaison. Pour ce faire, choisissez l'élément de menu **Project → Add Files to Project**. Cela fait apparaître la boîte de dialogue **Add Files to Project**. Dans la boîte de dialogue, sélectionnez *initmem.asm* et cliquez sur le bouton **Open**. Cela ajoute *initmem.asm* au projet. Pour ajouter lab1.cmd, choisissez **Projet → Add Files to Project**. Ensuite, dans la boîte de dialogue **Add Files to Project**, définissez **Files of type** to **Linker Command File**

(*.cmd), afin que *lab1.cmd* apparaisse dans la boîte de dialogue. Maintenant, sélectionnez lab1.cmd et cliquez sur le bouton **Open**. En plus des fichiers initmem.asm et lab1.cmd, le fichier de bibliothèque de support d'exécution doit être ajouté au projet. Pour ce faire, choisissez **Project → Add Files to Project**, accédez au dossier de bibliothèque du compilateur, ici supposé être l'option par défaut C: \ ti \ c6000 \ cgtools \ lib, sélectionnez **Object and Library Files** (* .o *, *. l *) dans la case **Files of type**, puis sélectionnez rts6700.lib et cliquez sur **Open**. Si vous utilisez le point fixe DSP TMS320C6211, sélectionnez plutôt rts6200.lib. À des fins de débogage, utilisons le programme C shell vide suivant. Créez un fichier source C main.c, entrez les lignes suivantes et ajoutez main.c au projet de la même manière que celle qui vient d'être décrite.

```
 MEMORY
{
        I RA M:          o = 00000000h l = 00100000h
        MY DATA:         o = 80000000h l = 00000100h
        C E0 :           o = 80000100h l = 000FFF00h
}

 SECTIONS
{
       . cinit > IRAM
       .text > IRAM
      .stack > IRAM
      .bss > IRAM
     .const > IRAM
    .data > IRAM
    .far > IRAM
   .switch > IRAM
   .sysmem > IRAM
  .tables > IRAM
  . cio > I RAM
  . my data > MYDATA
}
```

Fig. 5.16 : Fichier de commandes de l'éditeur de liens pour le laboratoire 1.

À des fins de débogage, utilisons le programme vide shell C suivant. Créez un fichier source C main.c, entrez les lignes suivantes et ajoutez main.c au projet de la même manière que celle qui vient d'être décrite.

```
#include <stdio.h>
void main()
{
    printf("BEGIN\n");
    printf("END\n");
}
```

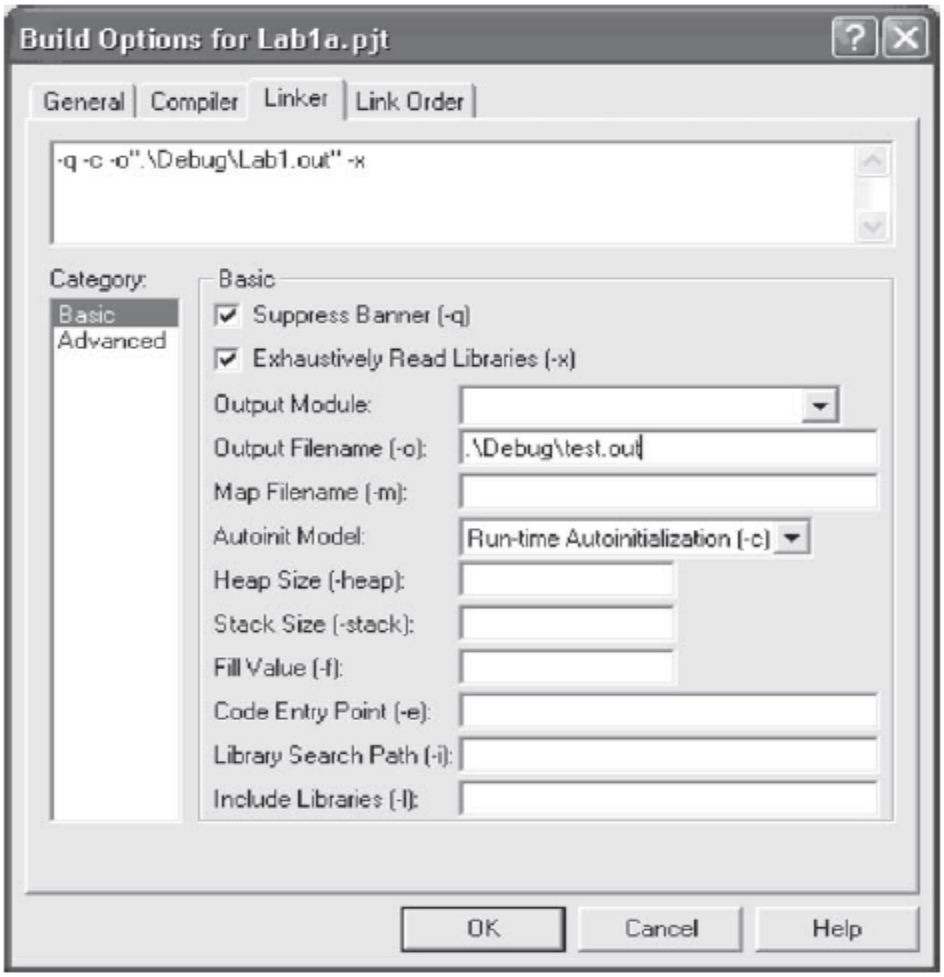
Après avoir ajouté tous les fichiers source, le fichier de commandes et le fichier de bibliothèque au projet, il est temps de créer le projet ou de créer un fichier exécutable pour le DSP cible. Pour ce faire, choisissez l'élément de menu **Project → Build**. CCS compile, assemble et relie tous les fichiers du projet. Les messages concernant ce processus sont affichés dans un panneau au bas de la fenêtre CCS. Une fois le processus de construction terminé sans erreur, le fichier exécutable lab1.out est généré. Il est également possible de faire des builds incrémentiels - c'est-à-dire de reconstruire uniquement les fichiers modifiés depuis la dernière build, en choisissant l'élément de menu **Project → Rebuild**. La fenêtre CCS fournit des boutons de raccourci pour les options de menu fréquemment utilisées, telles que **build et rebuild all**.

Bien que CCS propose des options de génération par défaut, ces options peuvent être modifiées en choisissant **Project → Build Options**. Par exemple, pour changer le nom du fichier exécutable en test.out, choisissez **Project → Build Options**, cliquez sur l'onglet **Linker** de liens de la fenêtre **Build Options** et tapez test.out dans le champ **Output Filename (-o)**, comme illustré dans la figure 5.17a.

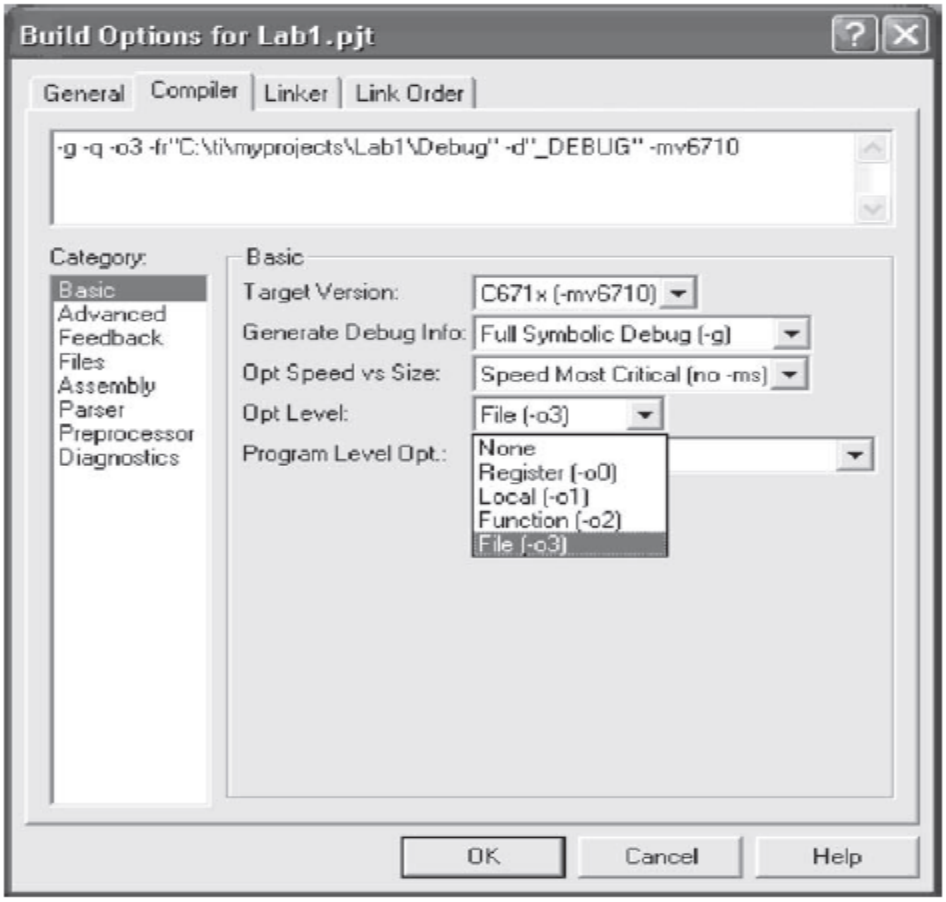
Notez que le fichier de commandes de l'éditeur de liens inclura test.out lorsque vous cliquez sur cette fenêtre.

Toutes les options du compilateur, de l'assembleur et de l'éditeur de liens sont définies via l'élément de menu **Project → Build Options**. Parmi les nombreuses options du compilateur présentées dans la figure 5.17b, faites particulièrement attention aux options de niveau d'optimisation. Il existe quatre niveaux d'optimisation (0, 1, 2, 3), qui contrôlent le type et le degré d'optimisation. L'option d'optimisation de niveau 0 effectue une simplification du graphique de flux de contrôle, alloue des variables aux registres, élimine le code inutilisé et simplifie les expressions et les instructions. L'optimisation de niveau 1 effectue toutes les optimisations de niveau 0, supprime les affectations inutilisées et élimine les expressions communes locales. L'optimisation de niveau 2 effectue toutes les optimisations de niveau 1, plus le pipelining logiciel, les optimisations de boucle et le déroulement de boucle. Il élimine également les sous-expressions communes globales et les affectations inutilisées. Enfin, l'optimisation de niveau 3 effectue toutes les optimisations de niveau 2, supprime toutes les fonctions qui ne sont jamais appelées et simplifie les fonctions avec des valeurs de retour qui ne sont jamais utilisées. Il appelle en ligne également les petites fonctions et réordonne les déclarations de fonctions.

Notez que dans certains cas, le débogage n'est pas possible en raison de l'optimisation. Ainsi, il est recommandé de déboguer d'abord votre programme pour vous assurer qu'il est logiquement correct avant d'effectuer toute optimisation. Une autre option importante du compilateur est l'option **Target Version**. Lorsque l'application concerne le DSP cible à virgule flottante TMS320C6711 / 6713 DSK, accédez au champ **Target Version** et sélectionnez 671x (–mV 6710) dans la liste déroulante. Pour le DSK cible DSP TMS320C6416 à virgule fixe, sélectionnez C64xx (–mv 6400). Pour la cible EVM TMS320C6701, sélectionnez C670x.


Build Options for Lab1a.pjt
General
Compiler
Linker
Link Order
-q -c -o".\Debug\Lab1.out" -x
Category:
Basic
Advanced
Basic
Suppress Banner (-q)
Exhaustively Read Libraries (-x)
Output Module:
Output Filename (-o):
.\Debug\test.out
Map Filename (-m):
Autoinit Model:
Run-time Autoinitialization (-c)
Heap Size (-heap):
Stack Size (-stack):
Fill Value (-f):
Code Entry Point (-e):
Library Search Path (-i):
Include Libraries (-l):
OK
Cancel
Help

(a)



(b)

Fig. 5.17 : (a) Options de construction et (b) options du compilateur.

Une erreur courante en écrivant initmem.asm est de taper les directives .sect et .short dans la première colonne. Étant donné que seules les étiquettes peuvent démarrer dans la première colonne, cela entraînera une erreur d'assembleur. Lorsqu'un message indiquant une erreur de compilation apparaît, cliquez sur **Stop Build** et faites défiler vers le haut dans la zone **Build** pour voir le message d'erreur de syntaxe. Double-cliquez sur le texte rouge qui décrit l'emplacement de l'erreur de syntaxe. Notez que le fichier initmem.asm s'ouvre et que votre curseur apparaît sur la ligne qui a provoqué l'erreur, voir la figure 5.18. Après avoir corrigé l'erreur de syntaxe, le fichier doit être enregistré et le projet reconstruit.

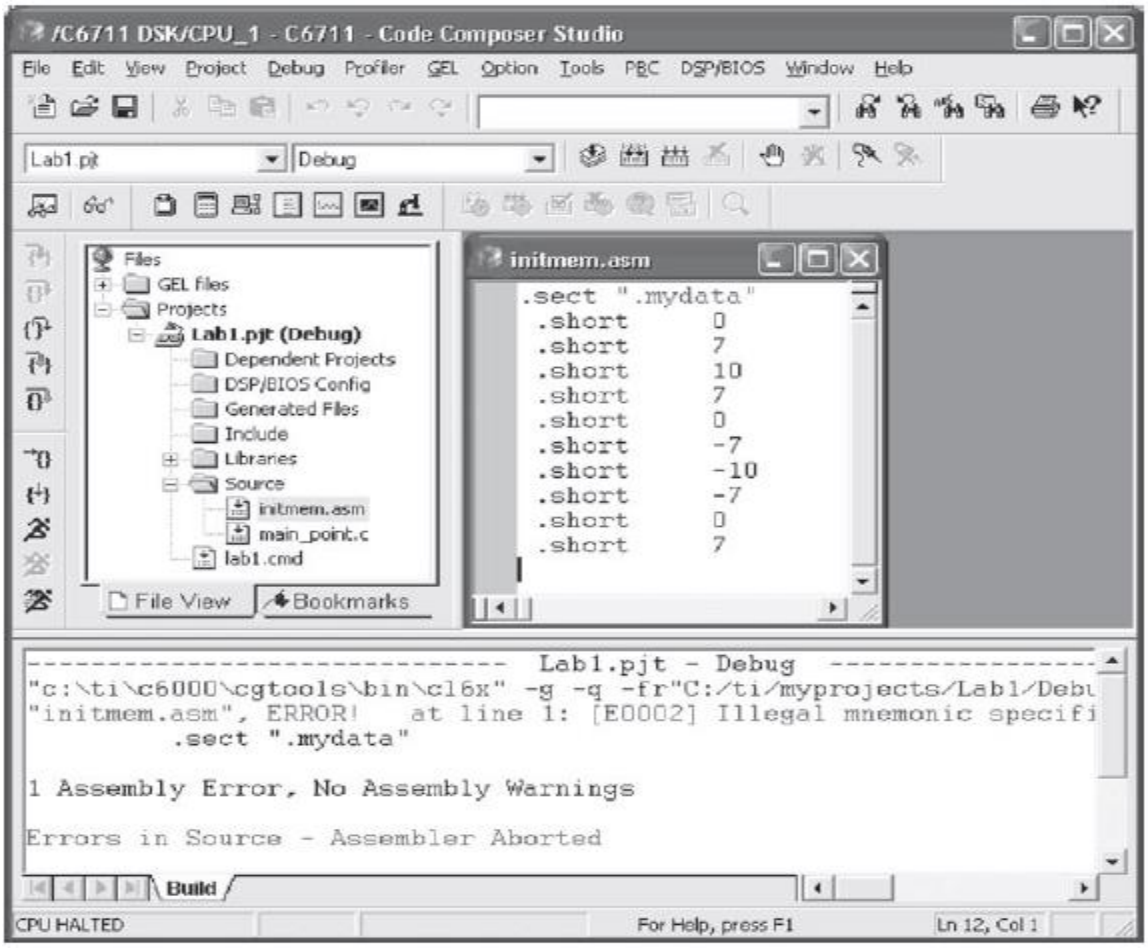


Fig. 5.18 : Erreur de construction

5.10 Outils de débogage (Lab 1.2)

Une fois le processus de construction terminé sans aucune erreur, le programme peut être chargé et exécuté sur le DSP cible. Pour charger le programme, choisissez **File → Load Program**, sélectionnez le programme lab1.out juste reconstruit et cliquez sur **Open**. Pour exécuter le programme, choisissez l'élément de menu **Debug → Run**. Vous devriez pouvoir voir BEGIN et END apparaître dans la fenêtre **Stdout**.

Maintenant, vérifions si le tableau de valeurs est assemblé dans l'emplacement mémoire spécifié. CCS permet de visualiser le contenu de la mémoire à un emplacement spécifique. Pour afficher le contenu de la mémoire à 0x80000000, sélectionnez **View → Memory** dans

le menu. La boîte de dialogue **Memory Window Options** apparaîtra. Cette boîte de dialogue permet de spécifier différents attributs de la fenêtre **Memory**. Accédez au champ **Address** et entrez 0x80000000. Ensuite, sélectionnez **16-bits Signed Int** dans la liste déroulante du champ **Format** et cliquez sur **OK**. Une fenêtre **Memory** apparaît comme illustré à la figure 5.19. Le contenu des registres CPU, périphérique, DMA et port série peut également être affiché en sélectionnant **View → Registers → Core Registers**, par exemple.

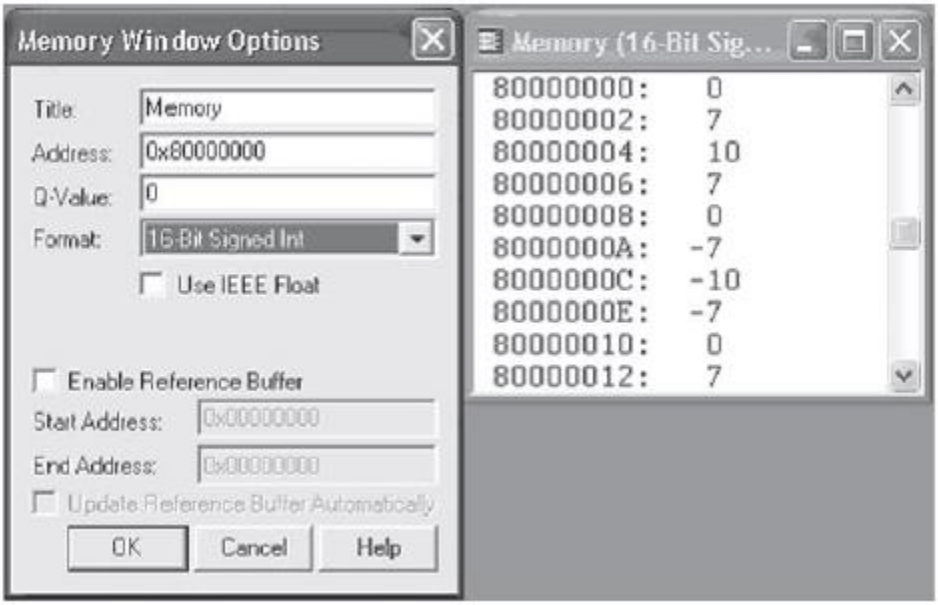


Fig. 5.19 : Boîte de dialogue Options de la fenêtre mémoire et fenêtre Mémoire.

Il existe une autre façon de charger des données d'un fichier PC vers la mémoire DSP. CCS fournit une capacité de point de sonde, de sorte qu'un flux de données peut être déplacé du fichier hôte du PC vers le DSP ou vice versa. Pour utiliser cette fonctionnalité, un **Probe Point** doit être défini dans le programme en plaçant le curseur de la souris sur la ligne où un flux de données doit être transféré et en cliquant sur le bouton **Probe Point**. Ensuite, choisissez **File→ File I / O** pour appeler la boîte de dialogue **File I / O**. Cliquez sur le

bouton **Add File** et sélectionnez le fichier de données à charger. Maintenant, le fichier doit être connecté au point de sonde en cliquant sur le bouton **Add Probe Point**. Dans le champ **Probe Point**, sélectionnez le point de sonde pour le rendre actif, puis connectez le point de sonde au fichier PC via **File In** :… dans le champ **Connect To**. Cliquez sur le bouton **Replace** puis sur le bouton **OK**. Enfin, entrez l'emplacement de la mémoire dans le champ **Address** et le nombre de données dans le champ **Length**. Notez qu'un nom de variable globale peut être utilisé dans le champ **Address**. La capacité de point de sonde est fréquemment utilisée pour simuler l'exécution d'un programme d'application en l'absence de signaux en direct. Un fichier PC valide doit avoir l'en-tête de fichier et l'extension corrects. L'en-tête du fichier doit être conforme au format suivant :

```
MagicNumber Format StartingAddress PageNum Length
```

MagicNumber est fixé à 1651. Le format indique le format des échantillons dans le fichier : 1 pour hexadécimal, 2 pour entier, 3 pour long et 4 pour float. StartingAddress et PageNum sont déterminés par CCS lorsqu'un flux de données est enregistré dans un fichier PC. La longueur indique le nombre d'échantillons dans la mémoire. Un fichier de données valide doit avoir l'extension .dat.

Un affichage graphique des données fournit souvent une meilleure rétroaction sur le comportement d'un programme. CCS fournit une interface d'analyse de signal pour surveiller un signal ou des données. Affichons le tableau de valeurs à 0x80000000 comme un signal ou un graphique temporel. Pour ce faire, sélectionnez **View → Graph → Time / Frequency** pour afficher la boite **Graph Property Dialog**. Les noms de champ apparaissent dans la colonne de gauche. Accédez au champ **Start Address**, cliquez dessus et tapez 0x80000000. Ensuite, accédez au champ **Acquisition Buffer Size**, cliquez dessus

et entrez 10. Enfin, cliquez sur DSP **Data Type**, sélectionnez un entier signé 16 bits dans la liste déroulante, puis cliquez sur **OK**. Une fenêtre graphique apparaît avec les propriétés sélectionnées. Ceci est illustré dans la figure 5.20. Vous pouvez modifier l'un de ces paramètres à partir de la fenêtre graphique en cliquant avec le bouton droit de la souris, en sélectionnant **Properties** et en ajustant les propriétés selon vos besoins. Les propriétés peuvent être mises à jour à tout moment pendant le processus de débogage.

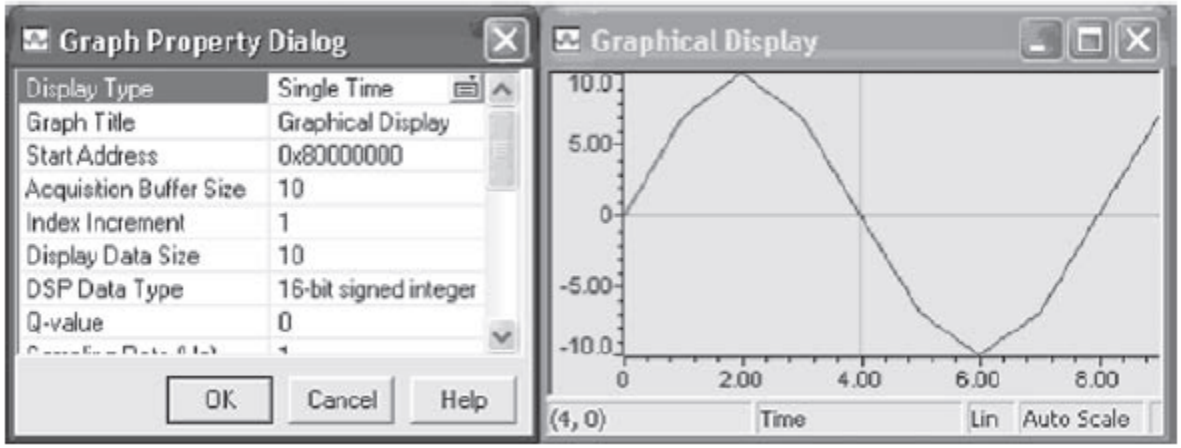


Fig. 5.20 : Boîte **Graph Property Dialog** et fenêtre **Graphical Display**.

Pour affecter un pointeur au début de l'espace mémoire assemblé, l'adresse mémoire peut être saisie directement dans un pointeur. Il est nécessaire de transtyper le pointeur en raccourci car les valeurs sont de ce type. Le code suivant peut être utilisé pour affecter un pointeur au début des valeurs et les parcourir pour les imprimer dans la fenêtre **Stdout** :

```
#include <stdio.h>
void main()
{
   int i;
   short *point;
   point = (short *) 0x80000000;
   printf("BEGIN\n");
   for(i=0;i<10;i++)
   { printf("[%d] %d\n",i, point[i]); }
   printf("END\n");

}
```

Au lieu de créer un nouveau fichier source, nous pouvons modifier le fichier main.c existant en double-cliquant sur le fichier main.c dans le panneau **Project View**, comme le montre la figure 5.21. Cette action fera apparaître le fichier source main.c dans la partie droite de la fenêtre CCS. Ensuite, entrez le code et reconstruisez-le. Avant d'exécuter le fichier exécutable, assurez-vous de recharger le fichier lab1.out. En exécutant ce fichier, vous devriez pouvoir voir les valeurs dans la fenêtre Stdout. Un double-clic dans le panneau **Project View** fournit un moyen simple d'afficher n'importe quel fichier source ou de commande à des fins de révision ou de modification.

Lors du développement et du test de programmes, il faut souvent vérifier la valeur d'une variable pendant l'exécution du programme. Ceci peut être réalisé en utilisant des points d'arrêt et des fenêtres de surveillance. Pour afficher les valeurs du pointeur dans main.c avant et après l'affectation du pointeur, choisissez **File → Reload Program** pour recharger le programme. Ensuite, double-cliquez sur main.c dans le panneau **Project View**. Vous souhaiterez peut-être agrandir la fenêtre afin de voir plus de fichiers en un seul endroit. Ensuite, placez votre curseur sur la ligne qui dit point = (short *) 0x80000000 et appuyez sur F9 pour définir un point d'arrêt. Pour ouvrir une fenêtre de surveillance, choisissez

View → Watch Window dans la barre de menus. Cela fera apparaître une fenêtre de surveillance (**Watch Window**) avec des variables locales répertoriées dans l'onglet **Watch Locals**. Pour ajouter une nouvelle expression à la fenêtre de surveillance, sélectionnez l'onglet **Watch 1**, puis tapez le point (ou toute expression que vous souhaitez examiner) dans la colonne Nom. Ensuite, choisissez **Debug → Run** ou appuyez sur F5. Le programme s'arrête au point d'arrêt et la fenêtre de surveillance affiche la valeur du pointeur. Il s'agit de la valeur avant que le pointeur ne soit défini sur 0x80000000. En appuyant sur F10 pour passer la ligne ou le bouton de raccourci, vous devriez pouvoir voir la valeur 0x80000000 dans le **Watch Window**.

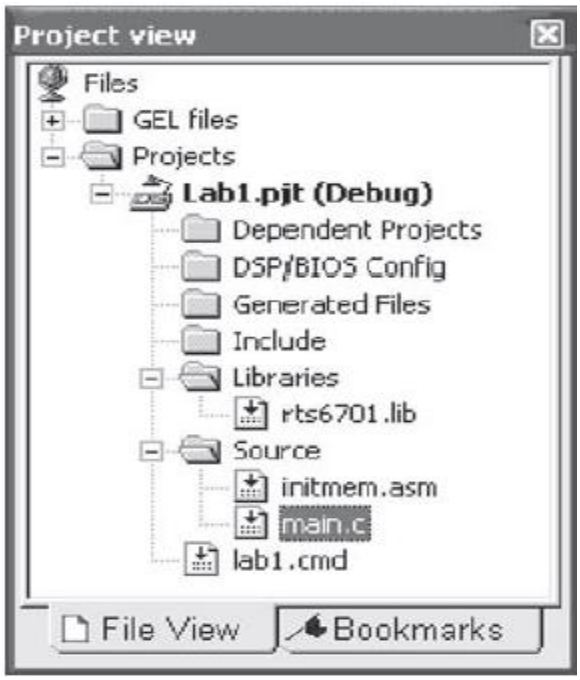


Fig. 5.21 : Panneau **Project View**

Pour ajouter une simple fonction C qui additionne les valeurs, nous pouvons simplement passer le pointeur au tableau et avoir un type de retour entier. Pour l'instant, ce qui est préoccupant, ce n'est pas comment les variables sont transmises, mais plutôt combien de temps il faut pour effectuer l'opération.

La fonction simple suivante peut être utilisée pour additionner les valeurs et renvoyer le résultat:

```
#include <stdio.h>
void main()
{
      int i,ret;
      short *point;
      point = (short *) 0x80000000;
      printf("BEGIN\n");
      for(i=0;i<10;i++)
      {  printf("[%d] %d\n",i, point[i]); }
      ret = ret_sum(point,10);
      printf("Sum = %d\n",ret);
      printf("END\n");
}

int ret_sum(const short* array,int N)


{
   int count,sum;
   sum = 0;
   for(count=0 ; count < N ; count++)
   sum += array[count];
   return(sum);

}
```

Dans le cadre du processus de débogage, il est normalement nécessaire de comparer ou de chronométrer le programme d'application. Dans ce laboratoire, déterminons le temps nécessaire à l'exécution de la fonction ret_sum (). Pour réaliser cette analyse comparative, rechargez le programme et choisissez **Profiler → Start New Session**. Cela fera apparaître **Profile Session Name**. Tapez un nom de session, MySession par défaut, puis cliquez sur **OK**. La fenêtre **Profile** affichant la taille du code et les statistiques sur le nombre de cycles sera ancrée au bas de CCS. Redimensionnez cette fenêtre en faisant glisser ses bords ou en la désancrant afin que vous puissiez voir toutes les colonnes. Maintenant, faites un clic

droit sur le code à l'intérieur de la fonction à comparer, puis choisissez Profile **Function → in… Session**. Le nom de la fonction sera ajouté à la liste dans la fenêtre **Profile**. Enfin, appuyez sur F5 pour exécuter le programme. Examinez le nombre de cycles indiqué dans la figure 5.22 pour ret_sum (). Il devrait être d'environ 732 cycles (le nombre exact peut légèrement varier). Il s'agit du nombre de cycles nécessaires pour exécuter la fonction ret_sum ().

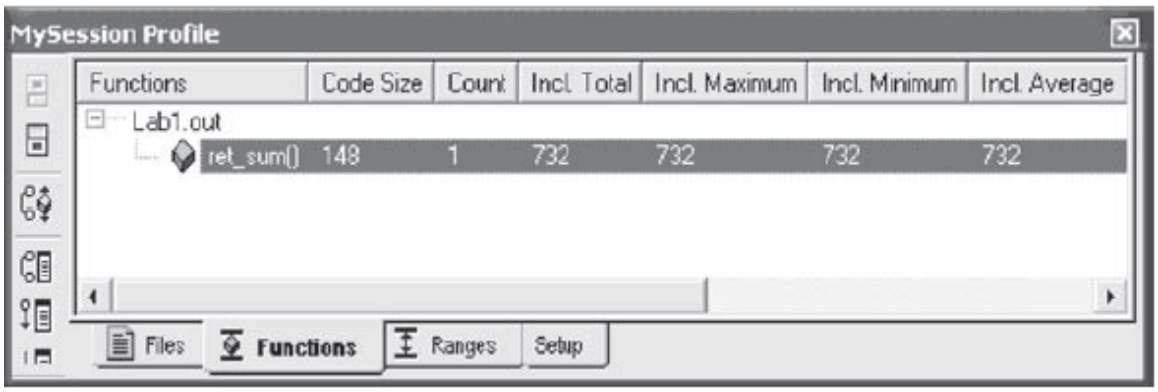


Fig. 5.22 : Fenêtre de profil.

Il existe une autre façon de comparer les codes à l'aide de points d'arrêt. Double-cliquez sur le fichier main.c dans le panneau **Project View** et choisissez **View → Mixed Source / ASM** pour répertorier les instructions assemblées correspondant aux lignes de code C. Définissez un point d'arrêt sur la ligne appelante en plaçant le curseur sur la ligne qui lit ret = ret_sum (point, 10), puis appuyez sur F9 ou double-cliquez sur **Selection Margin** situé sur le côté gauche de l'éditeur. Définissez un autre point d'arrêt sur la ligne suivante, comme indiqué dans la figure 5.23. Une fois les points d'arrêt définis, choisissez **Profiler → Enable Clock** pour activer l'horloge de profil. Ensuite, choisissez **Profiler → View Clock** pour faire apparaître une fenêtre affichant **Profile Clock**. Maintenant, appuyez sur F5 pour

exécuter le programme. Lorsque le programme est arrêté au premier point d'arrêt, réinitialisez l'horloge en double-cliquant sur la zone intérieure de la fenêtre **Profile Clock**. Enfin, cliquez sur **Step Out** ou **Run** dans le menu **Debug** pour exécuter et arrêter au deuxième point d'arrêt. Examinez le nombre d'horloges dans la fenêtre **Profile Clock**. Il doit lire 752. L'écart entre le point d'arrêt et les approches de profil provient des procédures supplémentaires pour appeler des fonctions, par exemple, passer des arguments à la fonction, stocker l'adresse de retour, se ramifier à partir de la fonction, etc.

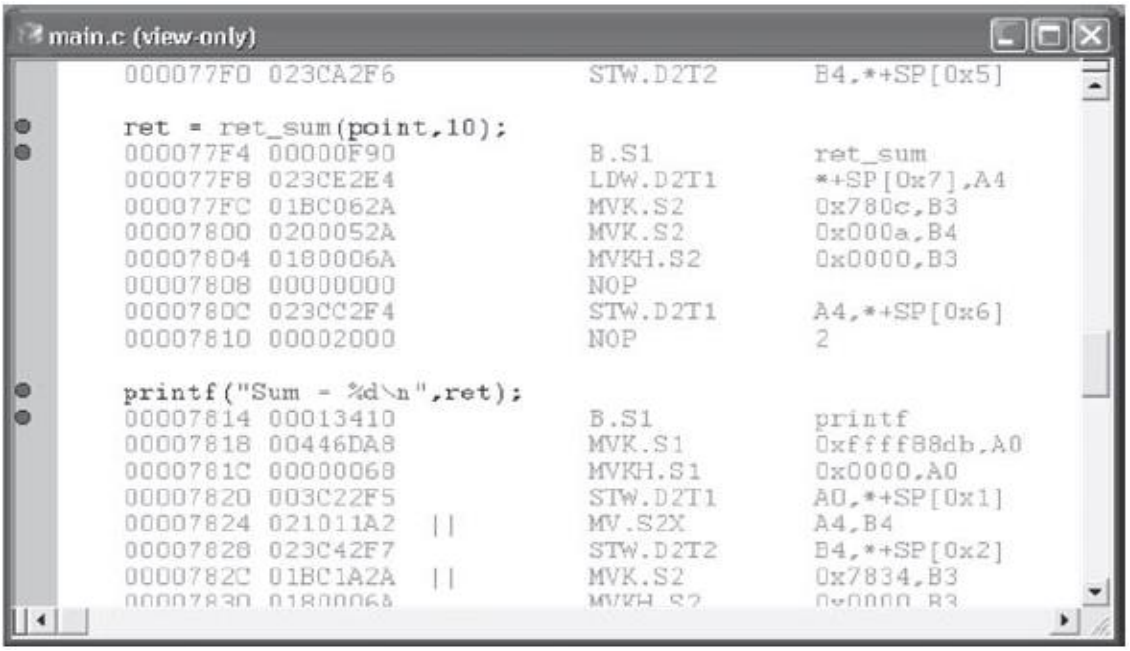


Fig. 5.23 : Profilage du temps d'exécution du code avec point d'arrêt.

Un espace de travail contenant des points d'arrêt, des points de sonde, des graphiques et des fenêtres de surveillance peut être stocké à des fins de rappel. Pour ce faire, choisissez **File → Workspace → Save Workspace As**. Cela affichera la fenêtre **Save Work**. Saisissez le nom de l'espace de travail dans le champ **File Name**, puis cliquez sur **Save**.

Le fichier ci-dessous est un programme d'assemblage pour calculer la somme des valeurs. Ici, les deux arguments de la fonction somme sont passés dans les registres A4 et B4. La valeur de retour est stockée dans A4 et l'adresse de retour dans B3. L'ordre dans lequel les registres sont choisis est régi par la convention d'argument de passage discutée plus loin.

Le nom de la fonction doit être précédé d'un trait de soulignement comme .global _sum.

Créez un nouveau fichier source sum.asm, comme illustré ci-dessous, et ajoutez-le au projet afin que main () puisse appeler la fonction sum ().

```
        .global         _sum
_sum:
        ZERO    .L1     A9              ; registre de somme
        MV      .L1     B4,A2           ; initialiser le compteur avec un argument passé
loop:   LDH     .D1     *A4++,  A7      ; valeur de charge pointée par A4 dans reg. A7
        NOP             4
        ADD     .L1     A7,A9,A9        ;A9 += A7
[A2]    SUB     .L1     A2,1,A2         ; compteur de décrémentation
[A2]    B       .S1     loop            ;revenir à la boucle
NOP                 5
MV              .L1     A9,A4           ; déplacer le résultat dans le registre de retour
A4
B               .S2     B3              ; rebrancher à l'adresse stockée dans B3

NOP                 5
```

Pour enregistrer le fichier, accédez au champ **Save as type** et sélectionnez **Assembly Source Files (* .asm)** dans la liste déroulante.

Le programme main () doit également être modifié en ajoutant un appel de fonction à la fonction d'assembleur sum (). Ce programme est illustré dans la figure 5.24. Générez le programme et exécutez-le. Vous devriez pouvoir voir la même valeur de retour.

```
#include <stdio.h>

void main()
{
    int i, ret;
    short *point;

    point = (short *) 0x80000000;

    printf ("BEGIN\n");

    for (i=0 ; i<10 ; i++)
    { printf (" [%d]  %d\n",i, point[i]); }

    ret = ret_sum (point,10);
    print f ("C program Sum = %d\n", ret);

    ret = sum (point, 10);
    print f ("Assembly program Sum = %d\n", ret);
    print f ("END\n");
}

int ret_sum (const short* array, int N)
{
    int count, sum;
        sum = 0;

    for (count=0 ; count < N ; count++)
        sum += array[count];

    return(sum);

}
```

Fig. 5.24: Programme complet de Lab1.

Le tableau 5.2 indique le nombre de cycles nécessaires pour exécuter la fonction somme à l'aide de plusieurs versions différentes. Lorsqu'un programme est trop volumineux pour tenir dans la mémoire interne, il doit être placé dans la mémoire externe. Bien que le

programme de ce laboratoire soit suffisamment petit pour tenir dans la mémoire interne, il est placé dans la mémoire externe pour étudier le changement du nombre de cycles. Pour déplacer le programme dans la mémoire externe, ouvrez le fichier lab1.cmd et remplacez la ligne .text> IRAM par .text> CE0. Comme le montre le tableau 5.2, cette génération ralentit l'exécution à 2535 cycles. Dans la deuxième version, le programme réside dans la mémoire interne et le nombre de cycles est donc réduit à 732. En augmentant le niveau d'optimisation, le nombre de cycles peut être encore réduit à 281. La version d'assemblage du programme donne 472 cycles. C'est plus lent que le programme C entièrement optimisé car il n'est pas encore optimisé. L'optimisation des codes d'assemblage sera discutée plus loin. À ce stade, il convient de souligner que, dans tous les laboratoires, le nombre de cycles indiqué indique les C6711 DSK avec CCS version 2.2. Le nombre de cycles variera légèrement en fonction de la cible DSK et de la version CCS utilisée.

Type de génération	Taille du code	Nombre de cycles	
		Données en mémoire externe	Données en mémoire interne
Programme C en mémoire externe	148	2535	2075
Programme C en mémoire interne	148	732	382
–o0	64	464	113
–o1	64	463	111
–o2	100	404	57
–o3	84	281	33
Programme assembleur	64	472	150

Tableau 5.2 : Nombre de cycles pour différentes générations (sur le DSK C6711 avec CCS2.2).

5.11 Cible EVM (Lab 1.3)

Comme décrit dans la section 4.1 et illustré dans la figure 4.1, la carte mémoire d'EVM possède plus d'espace mémoire interne/externe que celle de DSK, permettant ainsi plus de

flexibilité dans le chargement de code dans la section programme / données. La figure 5.25 montre un exemple utilisant la cible EVM où les données situées à EXT3, disons 0x03000000, sont placées au début de EXT3.

```
MEMORY
{

  PMEM        : origin = 0x00000000, length = 0x00010000
  E XT2       : origin = 0x02000000, length = 0x01000000
  E XT3       : origin = 0x03000000, length = 0x01000000
  DMEM        : origin = 0x80000000, length = 0x00010000
}

SECTIONS
{

     .vectors   > PMEM
     .text      > PMEM
     .bss       > DMEM
     .cinit     > DMEM
     .const     > DMEM
     .stack     > DMEM
     .cio       > DMEM
     .sysmem    > DMEM
     .f a r     > EXT2
     .mydata    > EXT3

   }
```

Fig. 5.25: Fichier de commande de Lab1.

Afin de placer le code de programme dans la mémoire externe, remplacez la ligne .text> PMEM par .text> EXT2. Le nombre de cycles sur la cible EVM est indiqué dans le tableau 5.3. Pour générer le projet, la bibliothèque rts6701.lib pour C6701 EVM ou rts6201.lib pour C6201 EVM doit être ajoutée au projet. Le champ **Target Version** de **Build option** doit être sélectionné en tant que **C670x (–mv 6700)** pour le DSP TMS320C6701 cible à

virgule flottante ou C620x (–mv 6200) pour le DSP TMS320C6201 EVM cible à virgule fixe.

Enfin, il convient de mentionner une remarque sur la réinitialisation de la carte EVM. Parfois, vous remarquerez que votre programme ne peut pas être chargé dans le DSP même s'il n'y a rien de mal à cela. Dans de telles circonstances, vous devez réinitialiser la carte EVM pour résoudre le problème. Cependant, vous devez fermer CCS avant de réinitialiser la carte. Sinon, le problème ne sera pas résolu.

légèrement en fonction de la cible DSK et de la version CCS utilisée.

Type de génération	Taille du code	Nombre de cycles	
		Données en mémoire externe	Données en mémoire interne
Programme C en mémoire externe	148	1185	955
Programme C en mémoire interne	148	541	355
–o0	64	282	97
–o1	64	281	95
–o2	100	213	36
–o3	84	129	21
Programme assembleur	64	139	133

Tableau 5.3 : Nombre de cycles pour différentes générations (sur C6701 EVM avec CCS2.2).

5.12 Simulateur (Lab 1.4)

Lorsqu'aucune carte DSP n'est disponible, le simulateur CCS peut être utilisé pour exécuter les programmes du laboratoire. Pour configurer CCS en tant que simulateur, sélectionnez simplement **simulateur** dans le champ **Platform** pendant le processus d'installation de CCS, comme illustré dans la figure 5.26. Dans la fenêtre **Imoprt Configurations**, sélectionnez l'option du simulateur pour l'une des cartes DSP spécifiées. En cliquant sur le

bouton **Import**, puis sur le bouton **Save and Quit**, le simulateur se configure et devient prêt à l'emploi. Notez que bien que le simulateur prenne en charge les opérations DMA et EMIF, les opérations liées à McBSP, HPI et Timer ne sont pas prises en charge. Les fichiers pour exécuter les travaux pratiques via le simulateur sont fournis dans le dossier du simulateur.

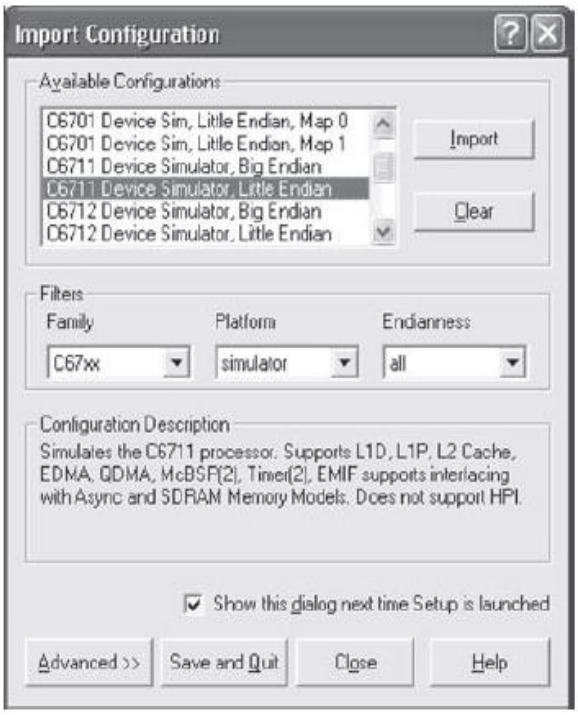


Fig. 5.26 : Installation du simulateur.

6.1 Adéquation algorithme-architecture

Le choix d'un processeur DSP pour implémenter un algorithme en temps réel dépend de l'application. De nombreux facteurs influencent ce choix. Ces facteurs comprennent le coût, les performances, la consommation d'énergie, la facilité d'utilisation, le délai de commercialisation et les capacités d'intégration / d'interfaçage.

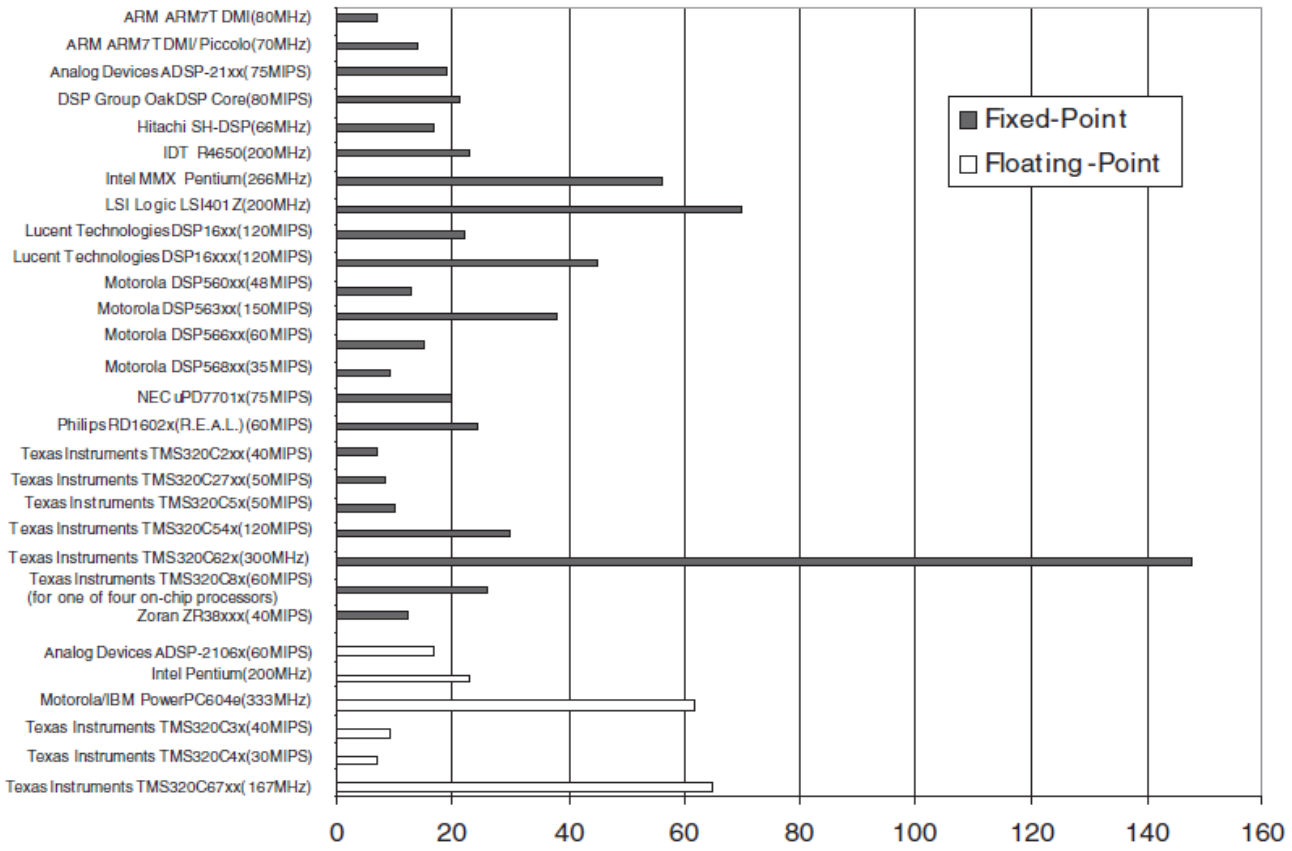
Le tableau 6.1 fournit une liste des processeurs C6x à virgule fixe et à virgule flottante actuellement disponibles. Comme on peut le voir dans ce tableau, la durée du cycle d'instruction, la vitesse, la consommation d'énergie, la mémoire, les périphériques, le boitier et les spécifications de coût varient selon les produits de cette famille.

La figure 6.1 illustre la puissance de traitement du C6x en montrant une comparaison de l'analyse comparative de vitesse avec d'autres processeurs DSP courants.

Dispositif	RAM(Bytes) Data/Prog	McBSP	(E) DMA	COM	Timers	MHz	Cycles (ns)	MIPS	Activité standard - Puissance interne totale (W) (pleine vitesse du dispositif)	Tension(V) Core, E/S	Boitier
TMS320C6711-100	4K/4K/64K	2	16	HPI/16	2	100	10	600	1.1	1.8, 3.3	256 BGA, 27mm
TMS320C6711-150	4K/4K/64K	2	16	HPI/16	2	150	6.7	900	1.1	1.8, 3.3	256 BGA, 27mm
TMS320C6713-200	4K/4K/256K	2	16	HPI/16	2	200	5	1200	1.0	1.2, 3.3	208 TKFP, 28mm
TMS320C6713-225	4K/4K/256	2	16	HPI/16	2	225	4.4	1350	1.2	1.26, 3.3	272 BGA, 27mm
TMS320C6701-150	64K/64K	2	4	HPI/16	2	150	6.7	900	1.3	1.8, 3.3	352 BGA, 35mm
TMS320C6701-167	64k/64k	2	4	HPI/16	2	167	6	1000	1.4	1.9, 3.3	352 BGA, 35mm
TMS320C6416-500	16k/16k/1M	2+UTOPIA*	64	PCI /HPI 32/16	3	500	2	4000	0.64	1.2, 3.3	532 BGA, 23mm
TMS320C6416-600	16k/16k/1M	2+UTOPIA*	64	PCI /HPI 32/16	3	600	1.67	4800	1.06	1.4, 3.3	532 BGA, 23mm

* Broches UTOPIA mélangées avec un troisième McBSP.

Tableau 6.1 : Exemples de spécifications de produit C6x DSP



*Le BDTImark est une mesure récapitulative de la vitesse DSP, distillée à partir d'une série de repères DSP développés et vérifiés indépendamment par Berkeley Design Technology, Inc. Un score BDTImark plus élevé indique un processeur plus rapide. Pour une description complète du BDTImark et de la méthodologie d'analyse comparative sous-jacente, ainsi que des scores BDTImark supplémentaires, reportez-vous à http://www.bdti.com. © 2000 Berkeley Design Technology, Inc.

Fig. 6.1 : BDTImark™ DSP Speed Metric (Mesure de vitesse)
par Berkeley Design Technology, Inc.1

6.2 Filtrage en temps réel (Lab 2)

Le but de ce laboratoire est de concevoir et de mettre en œuvre un filtre à réponse impulsionnelle finie sur le C6x. La conception du filtre se fait en utilisant MATLAB ™. Une fois la conception terminée, le code de filtrage est inséré dans le programme de shell d'échantillonnage en tant qu'ISR pour traiter les signaux en direct en temps réel.

Lab 2.1 Conception du filtre FIR (Réponse Impulsionnelle Finie)

MATLAB ou des packages de conception de filtre peuvent être utilisés pour obtenir les coefficients d'un filtre FIR souhaité. Pour avoir une simulation plus réaliste, un signal composite peut être créé et filtré dans

MATLAB. Un signal composite composé de trois sinusoïdes, comme le montre la figure 6.2, peut être créé par le code MATLAB suivant :

```
Fs=8e3;
Ts=1/Fs;
Ns=512;

t=[0:Ts:Ts*(Ns-1)];

f1=750;
f2=2500;
f3=3000;

x1=sin(2*pi*f1*t);
x2=sin(2*pi*f2*t);
x3=sin(2*pi*f3*t);

x=x1+x2+x3;
```

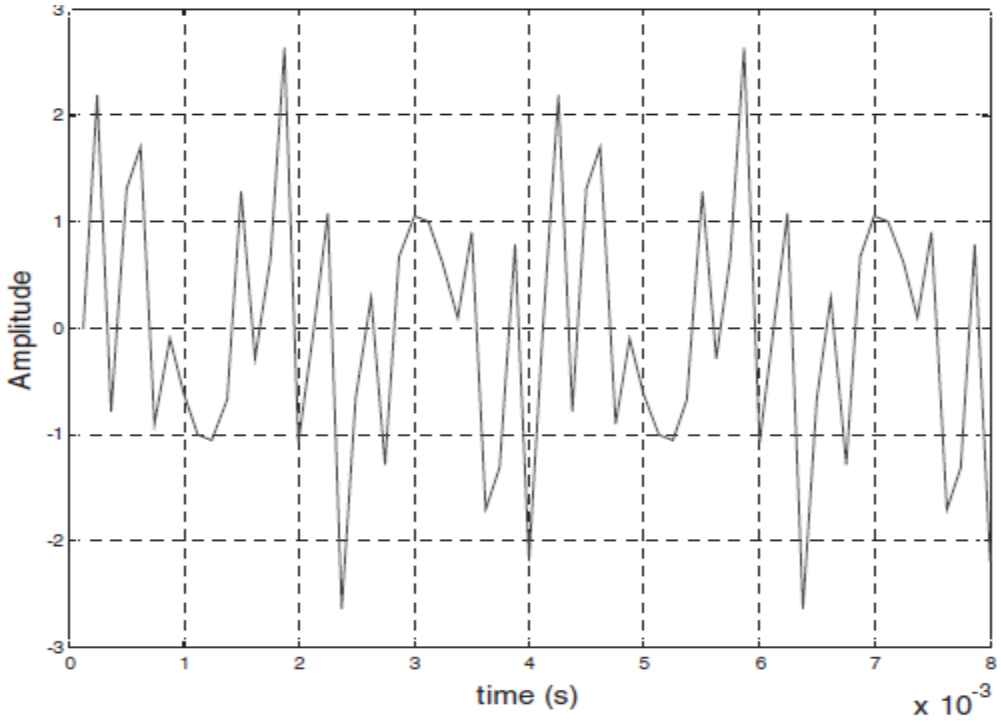


Fig. 6.2 : Deux cycles de signal composite.

Le contenu de la fréquence du signal peut être tracé en utilisant la fonction MATLAB ‘fft’. Trois pics doivent être observés, à 750 Hz, 2500 Hz et 3000 Hz. La fuite de fréquence observée sur la parcelle est due au fenêtrage causé par la période d'observation finie. Un filtre passe-bas est conçu ici pour filtrer les

fréquences supérieures à 750 Hz et conserver les composants inférieurs. La fréquence d'échantillonnage est choisie pour être de 8 kHz, ce qui est courant dans le traitement de la voix. Le code suivant est utilisé pour obtenir le tracé de fréquence illustré à la figure 6.3:

```
X=(abs(fft(x,Ns)));
y=X(1:length(X)/2);
f=[1:1:length(y)];
plot(f*Fs/Ns,y);
grid on;
```

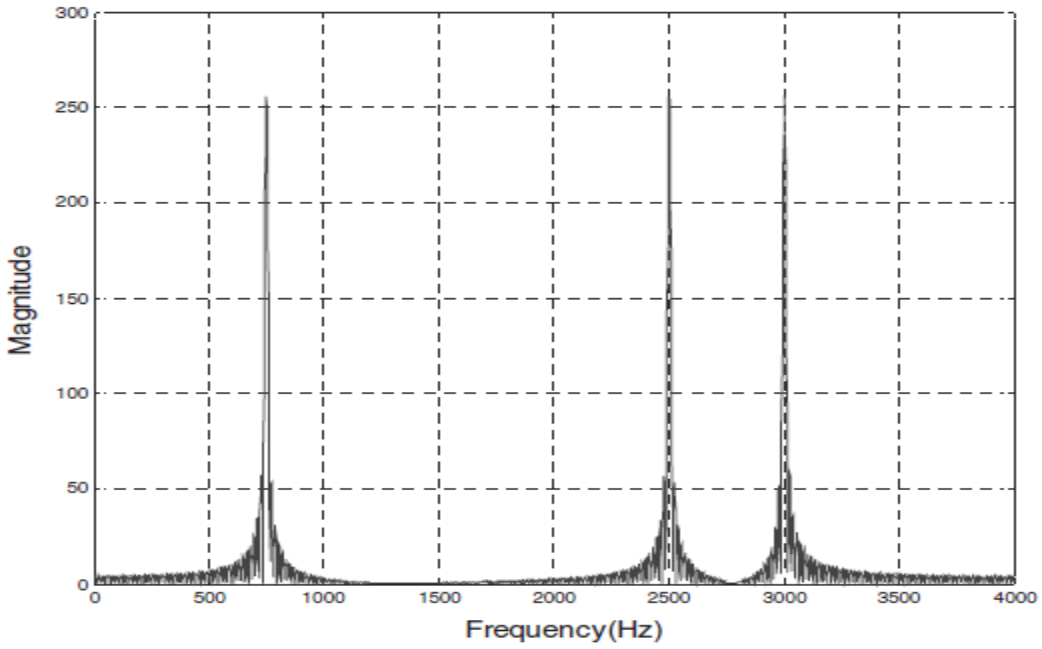


Fig. 6.3 : Composantes fréquentielles du signal composite.

Pour concevoir un filtre FIR avec une fréquence de bande passante = 1600 Hz, une fréquence de bande d'arrêt = 2400 Hz, un gain de bande passante = 0,1 dB, une atténuation de la bande passante = 20 dB, un taux d'échantillonnage = 8000 Hz, la méthode Parks-McClellan est utilisée via la fonction `` remez " de MATLAB [14]. La magnitude et la réponse de phase sont illustrées à la figure 6.4 et les coefficients sont indiqués au tableau 6.2. Le code MATLAB est le suivant:

```
rp = 0.1;           % Passband ripple
rs = 20;            % Stopband ripple
fs = 8000;          % Sampling frequency
f = [1600 2400];    % Cutoff frequencies
a = [1 0];          % Desired amplitudes
% Compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)];
[n,fo,ao,w] = remezord(f,a,dev,fs);
B = remez(n,fo,ao,w);
A=1;
freqz(B,A);
```

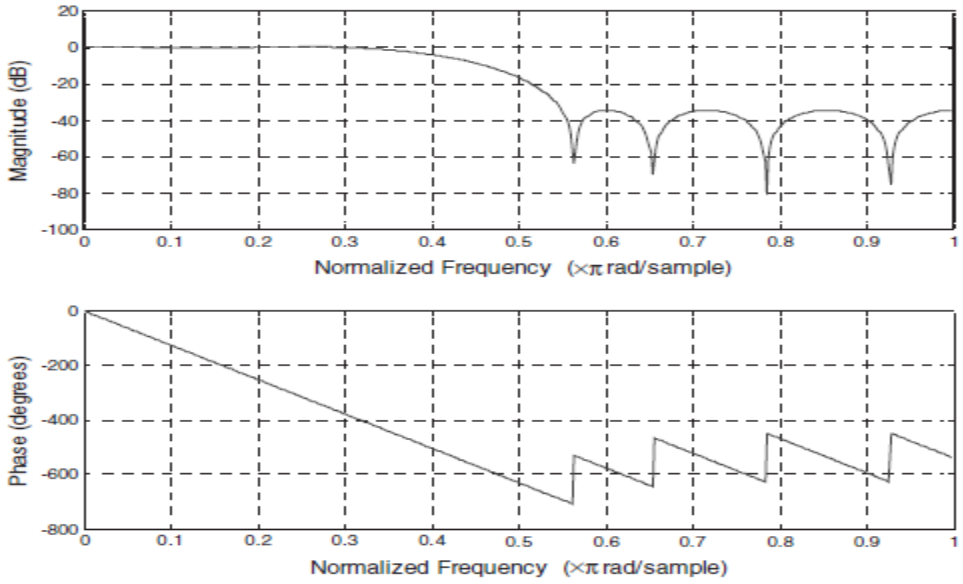


Fig. 6.4 : Amplitude du filtre et réponse de phase.

Coefficient	*Valeurs*	*Representation Q-15*
B0	0.0537	0x06DF
B1	0.0000	0x0000
B2	–0.0916	0xF447
B3	–0.0001	0xFFFD
B4	0.3131	0x2813
B5	0.4999	0x3FFC
B6	0.3131	0x2813
B7	–0.0001	0xFFFD
B8	–0.0916	0xF447
B9	0.0000	0x0000
B10	0.0537	0x06DF

Tableau 6.2: Coefficients du filtre FIR (RIF).

Avec ces coefficients, la fonction «filter» de MATLAB est utilisée pour vérifier que le filtre FIR est réellement capable de filtrer les signaux de 2,5 kHz et 3 kHz. Le code MATLAB suivant permet d'inspecter visuellement l'opération de filtrage :

```
% Figure 6-5
subplot(3,1,1);
va_fft(x,1024,8000);
subplot(3,1,2);
[h,w]=freqz(B,A,512);
plot(w/(2*pi),10*log(abs(h)));
grid on;

subplot(3,1,3);
y = filter(B,A,x);
va_fft(y,1024,8000);

function va_fft(x,N,Fs)
X=fft(x,N);
XX=(abs(X));
XXX=XX(1:length(XX)/2);
y=XXX;
f=[1:1:length(y)];
plot(f*Fs/N,y);
grid on;

% Figure 6-6
n=128
subplot(2,1,1);
plot(t(1:n),x(1:n));
grid on;
xlabel('Time(s)');
ylabel('Amplitude');
title('Original and Filtered Signals');
subplot(2,1,2);
plot(t(1:n),y(1:n));
grid on;
xlabel('Time(s)');
ylabel('Amplitude');
```

En observant les tracés apparaissant sur les figures 6.5 et 6.6, nous voyons que le filtre est capable de supprimer les composantes de fréquence souhaitées du signal composite. Notez que la réponse temporelle a un temps de configuration initial, ce qui rend les premiers échantillons de données inexacts. Maintenant que la conception du filtre est terminée, considérons la mise en œuvre du filtre.

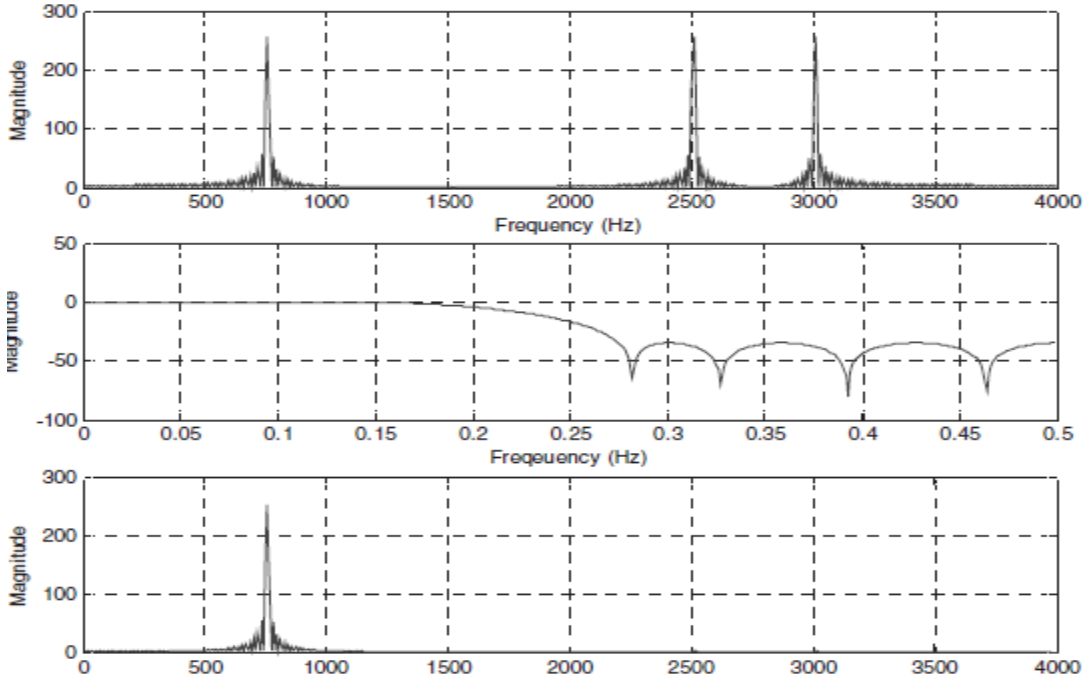


Fig. 6.5 : Représentation en fréquence de l'opération de filtrage.

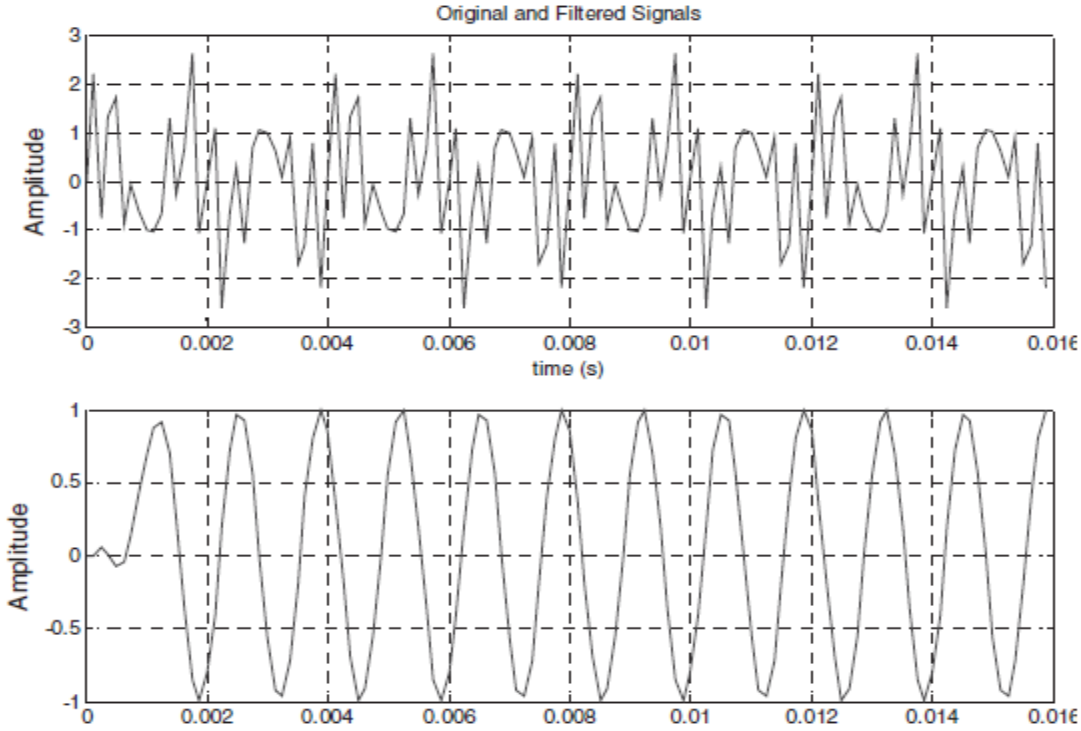


Fig. 6.6 : Représentation dans le domaine temporel de l'opération de filtrage.

Lab 2.2 Implémentation du filtre FIR

Un filtre FIR peut être implémenté en C ou en assembleur. Le but de l'implémentation est d'avoir un algorithme de temps de cycle minimum. Cela signifie que pour effectuer le filtrage aussi rapidement que possible afin d'atteindre la fréquence d'échantillonnage la plus élevée (le plus petit intervalle de temps d'échantillonnage). Initialement, le filtre est implémenté en C, car cela demande un moindre effort de codage. Une fois un algorithme de travail en C obtenu, les niveaux d'optimisation du compilateur (c'est-à-dire –o2, –o3) sont activés pour réduire le nombre de cycles. Une mise en œuvre du filtre est ensuite effectuée dans un assembleur codé à la main, qui peut être canalisé par logiciel pour des performances optimales. Une mise en œuvre finale du filtre est effectuée en assembleur linéaire et les résultats de synchronisation sont comparés.

L'équation de différence $y[n] = \sum_{k=0}^{N-1} B_k * x[n-k]$ est implémentée pour réaliser le filtre. Étant donné que le filtre est implémenté sur le DSK, le codage se fait en modifiant le programme d'échantillonnage dans le laboratoire 2, qui utilise une ISR (routine d'interruption) capable de recevoir un échantillon du port série et de le renvoyer sans aucune modification.

Lorsque vous utilisez le C6711 DSK avec la carte fille audio, les données reçues de McBSP1 ont une largeur de 32 bits, les 16 bits les plus significatifs provenant du canal droit et les 16 bits les moins significatifs provenant du canal gauche.

En considérant la représentation Q-15 ici, l'instruction MPY est utilisée pour multiplier la partie inférieure d'un échantillon de 32 bits (canal gauche) par un coefficient de 16 bits. Afin de stocker le produit sur 32 bits, il doit être décalé d'un gauche pour se débarrasser du bit de signe étendu. Maintenant, pour exporter le produit vers la sortie du codec, il doit être décalé vers la droite de 16 pour le placer dans les 16 bits inférieurs. Alternativement, le produit peut être décalé vers la droite de 15 sans retirer le bit de signe.

Pour implémenter l'algorithme en C, les opérateurs _mpy () intrinsèque et shift «<<» et «>>» doivent être utilisés comme suit:

```
result = ( _mpy(sample,coefficient) ) << 1;
result = result >> 16;
```

ou

```
result = ( _mpy(sample,coefficient) ) >> 15;
```

Ici, le résultat et l'échantillon ont une largeur de 32 bits, tandis que le coefficient est de 16 bits. Le _mpy () intrinsèque multiplie les 16 bits inférieurs du premier argument par les 16 bits inférieurs du deuxième argument. Par conséquent, les 16 bits d'échantillon inférieurs sont utilisés dans la multiplication.

Pour le bon fonctionnement du filtre FIR, il est nécessaire que l'échantillon actuel et N-1 échantillons précédents soient traités en même temps, où N est le nombre de coefficients. Par conséquent, les N échantillons les plus récents doivent être stockés et mis à jour avec chaque échantillon entrant. Cela peut être fait facilement via le code suivant :

```
void interrupt serialPortRcvISR()
{
      int i, temp, result= 0;
      temp = MCBSP_read(hMcbsp);

      // Update array samples
      for(i=N-1;i>=0;i--)
           samples[i+1] = samples[i];

      samples[0] = temp;

      // Filtering
      for( i = 0 ; i <= N ; i++ )
            result += ( _mpy(samples[i], coefficients[i]) ) << 1;
       result = result >> 16;
       MCBSP_write(hMcbsp, result);

  }
```

Pour terminer la mise en œuvre du filtre FIR, nous devons incorporer les coefficients du filtre :

```
#define N 10

// FIR filter coefficients
short coefficients[N+1] = { 0x6DF, 0x0, 0xF447, 0xFFFD, 0x2813, 0x3FFC,
0x2813, 0xFFFD, 0xF447, 0x0, 0x6DF};

int samples[N];
```

```
        .
        .
int main()
{

      .
      .

      for(i = 0; i <= N; i++ )
            samples[i]=0;
      .
      .

   }
```

6.3 Filtrage adaptative (Lab 3)

Le filtrage adaptatif est utilisé dans de nombreuses applications allant de l'annulation du bruit à l'identification du système. Dans la plupart des cas, les coefficients d'un filtre RIF sont modifiés en fonction d'un signal d'erreur afin de s'adapter à un signal souhaité. Dans ce laboratoire, un exemple d'identification de système est mis en œuvre dans lequel un filtre RIF adaptatif est utilisé pour s'adapter à la sortie d'un filtre passe-bande IIR du septième ordre. Le filtre RII (Réponse Impulsionnelle Infinie) est conçu dans MATLAB et implémenté en C. Le RIF adaptatif est d'abord implémenté en C puis assemblé en utilisant un tampon circulaire.

Dans l'identification de système, le comportement d'un système inconnu est modélisé en accédant à ses entrées et sorties. Un filtre RIF adaptatif peut être utilisé pour s'adapter à la sortie du système sur la base de la même entrée. La différence entre la sortie du système, d [n] et la sortie du filtre adaptatif, y [n], constitue le terme d'erreur e [n], qui est utilisé pour mettre à jour les coefficients du filtre RIF. La figure 6.7 illustre ce processus.

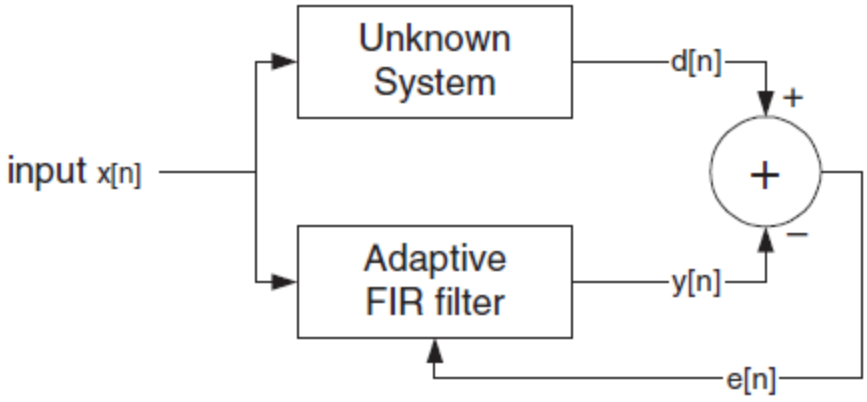


Fig. 6.7 : Filtrage adaptif

Le terme d'erreur calculé à partir de la différence des sorties des deux systèmes est utilisé pour mettre à jour chaque coefficient du filtre FIR selon la formule (algorithme du moindre carré moyen (LMS) [15]) :

$$h_n[n] = h_{n-1}[k] + \delta * \mathrm{e}[n] * x[n-k] \qquad (6.1)$$

où les h désignent la réponse unitaire de l'échantillon ou les coefficients du filtre RIF. La sortie y [n] est nécessaire pour approcher d [n]. Le terme δ indique la taille du pas. Une petite taille de pas assurera la convergence, mais entraîne un taux d'adaptation lent. Une grande taille de pas, bien que plus rapide, peut conduire à ignorer la solution.

Lab 3.1 Conception du filtre RII

Un filtre RII passe-bande du septième ordre est utilisé pour agir comme le système inconnu. Le RIF adaptatif est conçu pour s'adapter à la réponse du système RII. Compte tenu d'une fréquence d'échantillonnage de 8 kHz, laissez le filtre IIR avoir une bande passante de π / 3 à 2π / 3 (radians), avec une atténuation de bande d'arrêt de 20 dB. La conception du filtre peut être facilement réalisée avec la fonction MATLAB 'yulewalk' [14]. Le code MATLAB suivant peut être utilisé pour obtenir les coefficients du filtre :

```
Nc=7;
f=[0 0.32 0.33 0.66 0.67 1];
```

```
m=[0 0 1 1 0 0];
[B,A]=yulewalk(Nc,f,m);
freqz(B,A);
%Create A sample signal
Fs=8000;
Ts=1/Fs;
Ns=128;
t=[0:Ts:Ts*(Ns-1)];
f1=750;
f2=2000;%The one to keep
f3=3000;
x1=sin(2*pi*f1*t);
x2=sin(2*pi*f2*t);
x3=sin(2*pi*f3*t);
x=x1+x2+x3;
%Filter it
y=filter(B,A,x);
```

On peut vérifier que le filtre fonctionne en déployant un signal composite simple. À l'aide de la fonction MATLAB «filtre», vérifiez la conception en observant que les composantes de fréquence du signal composite tombant dans la bande d'arrêt sont supprimées. (Voir tableau 6.3 et figure 6.8.)

A	B
1.0000	0.1191
0.0179	0.0123
0.9409	−0.1813
0.0104	−0.0251
0.6601	0.1815
0.0342	0.0307
0.1129	−0.1194
0.0058	−0.0178

Remarque : Ne pas confondre les coefficients A&B avec les registres CPU A&B!

Tableau 6.3: Coefficients du filtre RII

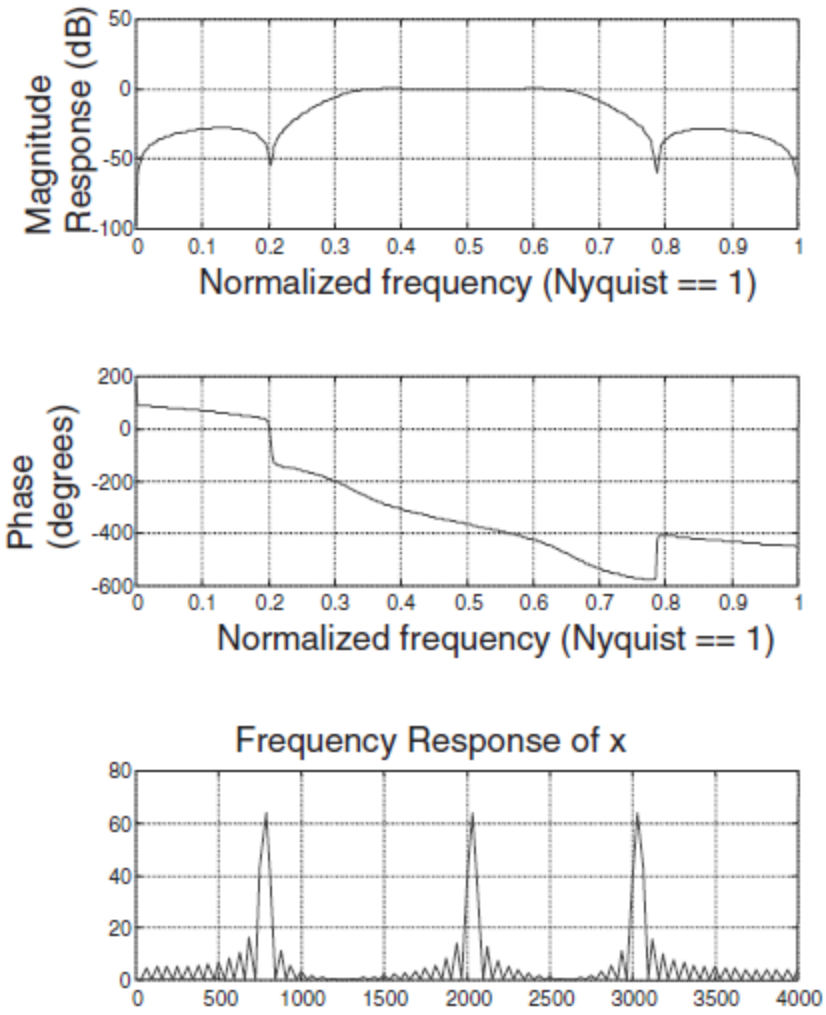


Fig. 6.8: Réponse du filtre RII.

Lab 3.2 Implémentation du filtre RII

L'implémentation du filtre IIR s'effectue d'abord en C, en utilisant l'équation de différence suivante

$$y[n] = -\sum_{k=1}^{N} a_k * y[n-k] + \sum_{K=0}^{N} b_k * x[\ddot{n}-k], \qquad (6.2)$$

où les a_k et b_k désignent les coefficients du filtre RII. Deux tableaux sont nécessaires, l'un pour les échantillons d'entrée x [n] et l'autre pour les échantillons de sortie y [n]. Etant donné que le filtre est de l'ordre 7, un tableau d'entrée de taille 8 et un tableau de sortie de taille 7 sont considérés. Les tableaux sont utilisés pour simuler un tampon circulaire, car en C cette propriété du CPU n'est pas accessible. Lorsqu'un nouvel échantillon arrive, tous les éléments du tableau d'entrée sont décalés d'une unité vers le bas. De

cette manière, le dernier élément est perdu et les huit derniers échantillons sont toujours conservés. Le tableau d'entrée est utilisé pour calculer la sortie résultante, puis la sortie est utilisée pour modifier le tableau de sortie. Une implémentation simple de ce schéma est présentée dans le code suivant :

```
interrupt void serialPortRcvISR (void)
{

      int temp,n,ASUM,BSUM;
      short input,IIR_OUT;

      temp = MCBSP_read(hMcbsp);
      input = temp >> S;                        // Facteur d'échelle

      for(n=7;n>0;n--)                          // Tampon d'entrée
            IIRwindow[n] = IIRwindow[n-1];

IIRwindow[0] = input;

BSUM = 0;

for(n=0;n<=7;n++)
{                                               // Multiplication de Q-15 avec Q-15
 BSUM += ((BS[n]*IIRwindow[n]) << 1);           // Résultats en Q-30. Décalage par un pour
}                                               // Eliminer le bit d'extension de signe

ASUM = 0;
for(n=0;n<=6;n++)
{

      ASUM += ((AS[n] * y_prev[n]) << 1);

}

IIR_OUT = (BSUM - ASUM) >> 16;

for(n=6;n>0;n--)                                 // Tampon de sortie
      y_prev[n] = y_prev[n-1];

y_prev[0] = IIR_OUT;

MCBSP_write(hMcbsp, IIR_OUT << S);               // Facteur d'échelle S

}
```

En exécutant ce programme tout en connectant un générateur de fonctions et un oscilloscope à l'entrée et à la sortie ligne de la carte fille audio, la fonctionnalité du filtre RII peut être vérifiée. Chaque fois que le DRR (référence Matlab d'un algorithme pour estimer le rapport d'énergie directe à diffuser) reçoit un nouvel échantillon entrant du générateur de fonctions, l'ISR est appelé. Ensuite, le nouvel échantillon est décalé vers la droite du facteur d'échelle S. Ce facteur est inclus pour la correction de tout débordement possible. Dans ce laboratoire, il n'est pas nécessaire de déplacer. Une fois le nouvel échantillon mis à l'échelle, les huit derniers échantillons sont conservés en éliminant l'échantillon le plus ancien et en

ajoutant le nouvel échantillon au tampon d'entrée IIRwindow. Cette opération se fait en décalant les données dans le tableau d'entrée. Notez que ce tableau est global et est initialisé à zéro dans la fonction principale.

6.4 Tampon circulaire

Dans de nombreux algorithmes DSP, tels que le filtrage, le filtrage adaptatif ou l'analyse spectrale, nous devons déplacer des données ou mettre à jour des échantillons (c'est-à-dire que nous devons traiter avec une fenêtre mobile). La méthode directe de déplacement des données est inefficace et utilise de nombreux cycles. La mise en mémoire tampon circulaire est un mode d'adressage par lequel un effet de fenêtre mobile peut être créé sans la surcharge associée au décalage des données. Dans un tampon circulaire, si un pointeur pointant vers le dernier élément du tampon est incrémenté, il est automatiquement enroulé et pointé vers le premier élément du tampon. Cela fournit un mécanisme simple pour exclure l'échantillon le plus ancien tout en incluant l'échantillon le plus récent, créant un effet de fenêtre mobile comme illustré à la figure 6.9.

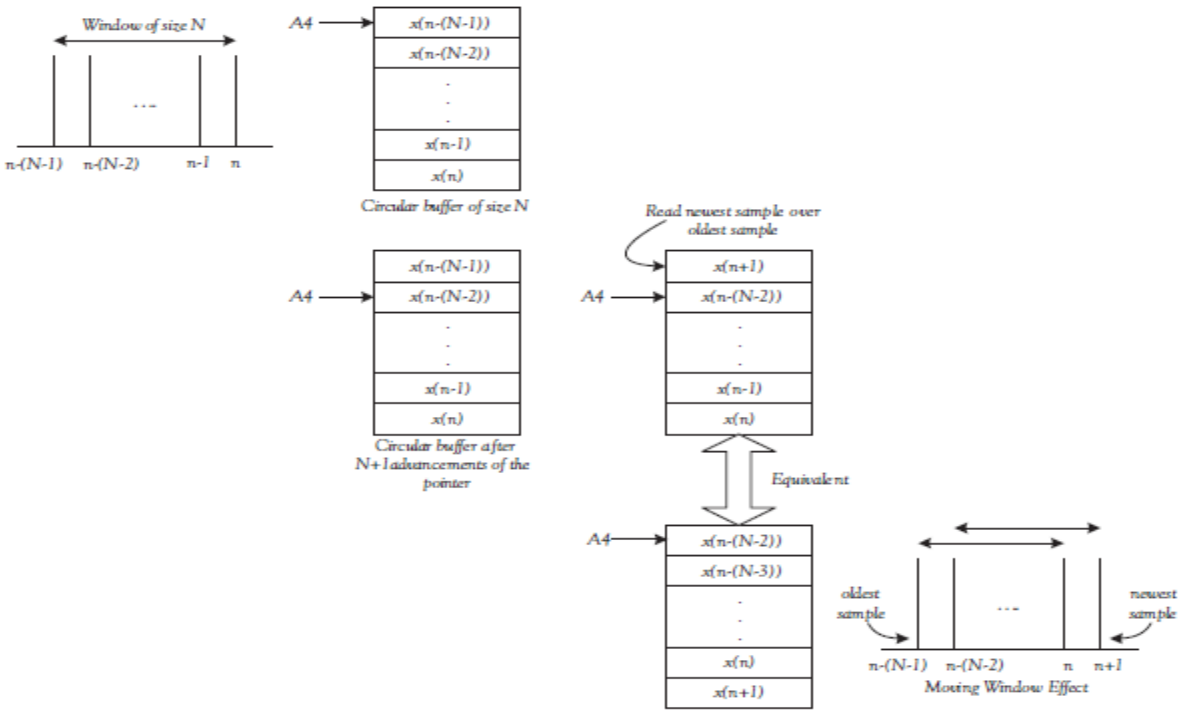


Fig. 6.9 : Effet de fenêtre-mobile.

Certains DSP disposent d'un matériel dédié pour effectuer ce type d'adressage. Sur le processeur C6x, l'unité logique et arithmétique a la capacité du mode d'adressage circulaire intégrée. Pour utiliser la mise en mémoire tampon circulaire, les tailles de mémoire tampon circulaire doivent d'abord être écrites dans les champs de taille de bloc BK0 et BK1 du registre de mode d'adresse (AMR), comme illustré dans la figure 6.10. Le C6x permet deux tampons circulaires indépendants de puissances de 2. La taille de la mémoire tampon est spécifiée comme $2^{(N+1)}$ octets, où N indique la valeur écrite dans les champs de taille de bloc BK0 et BK1.

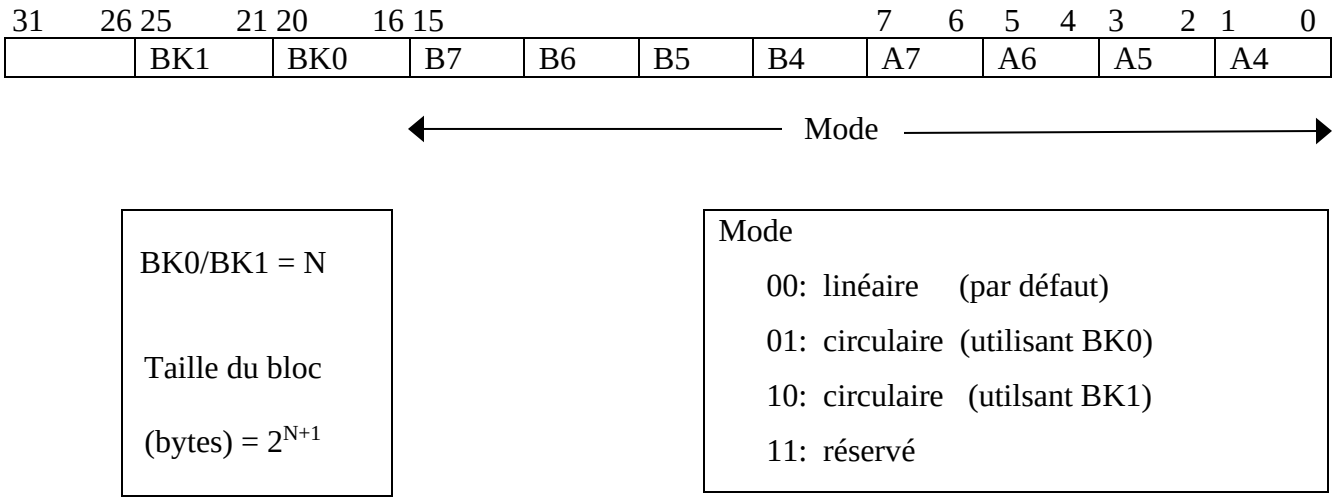


Fig. 6.10 : AMR (registre du mode d'adresse).

Ensuite, le registre à utiliser comme pointeur de tampon circulaire doit être spécifié en définissant les bits appropriés de AMR à 1. Par exemple, comme le montre la figure 8-2, pour utiliser A4 comme pointeur de tampon circulaire, le bit 0 ou 1 est réglé sur 1. Sur les 32 registres du C6x, 8 peuvent être utilisés comme pointeurs tampons circulaires : A4 à A7 et B4 à B7. Il faut noter que l'adressage linéaire est le mode d'adressage par défaut pour ces registres.

La figure 6.11 montre le code pour configurer le registre AMR pour un tampon circulaire de taille 8, avec un exemple de chargement. Pour configurer un tel tampon circulaire en C, il faut utiliser ce qu'on appelle un assembleur en ligne comme suit:

```
asm ("MVK.S2     0001h,B2");
asm ("MVKLH.S2   0002h,B2");
asm ("MVC.S2     B2,AMR");
```

```
; Blk size = 8, use   A4/BK0

      MVK.S2          0001H , B2
      MVKLH.S2        0002H , B2
      MVC.S2          B2, AMR

      LDH.D1          * A4++[2],  A1 ; A1 = 0, A4=&s[2]
      LDH.D1          * A4++[3],  A1 ; A1 = 2, A4=&s[1]
```

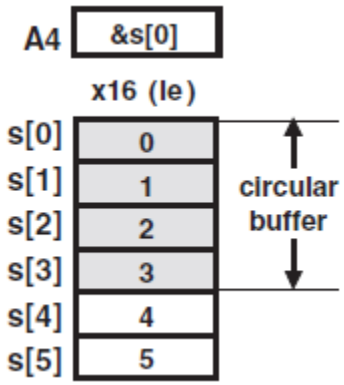


Fig. 6.11: Exemple de configuration AMR.

Lors de l'utilisation de tampons circulaires, il faut veiller à aligner les données sur la limite de taille de tampon. En C, cela peut être réalisé en utilisant des directives pragma. Les directives Pragma indiquent quels types de prétraitement sont effectués par le compilateur. Le Pragma DATA_ALIGN peut être utilisé pour aligner le symbole sur une constante de limite d'alignement de puissance 2 (en octets) comme suit :

#pragma DATA_ALIGN (symbol,constant)

6.5 Implémentation de la FFT (Fast Fourier Transform)

Pour permettre le traitement en temps réel, la FFT est utilisée qui utilise les propriétés de symétrie de la DFT. L'approche de calcul d'une FFT à 2N points telle que mentionnée dans le rapport d'application TI SPRA291 [16] est adoptée ici. Cette approche consiste à former deux nouveaux signaux à N points x_1 [n]

et x_2 [n] à partir du signal d'origine à 2N points g [n] en le divisant en parties paires et impaires comme suit :

$x_1[n] = g[2n], \quad 0 \leq n \leq N-1$;

$x_2[n] = g[2n+1]$ (6.3)

À partir des deux séquences x1 [n] et x2 [n], une nouvelle séquence complexe est définie comme

$x[n] = x1[n] + jx2[n], \quad 0 \leq n \leq N-1$ (6.4)

Pour obtenir G [k], DFT de g [n], l'équation

$G[k] = X[k]A[k] + X^*[N-k]B[k],$ (6.5)

K = 0, 1, …., N-1, avec X[N] = X[0]

est utilisée, où

$A[k] = ½(1-j W_{2N}^k)$ (6.6)

et $B[k] = ½(1+j W_{2N}^k)$ (6.7)

Seuls N points de G [k] sont calculés à partir de l'équation. (6.5). Les points restants sont trouvés en utilisant la propriété conjuguée complexe de G [k], G [2N - k] = G * [k]. En conséquence, une transformation à 2N points est calculée sur la base d'une transformation à N points, ce qui conduit à une réduction du nombre de cycles. Les codes des fonctions (*split1*, *R4DigitRevIndexTableGen*, *digit_reverse* et *radix4*) implémentant cette approche sont fournis dans le *TI Application Report* [16].

La figure 6-12 montre le résultat de la FFT où le signal a été réduit à 0, 2, 4 et 5 fois, respectivement. La mise à l'échelle est effectuée pour éliminer les débordements, qui sont présents pour les facteurs d'échelle 0, 2 et 4. Comme le révèlent ces figures, le signal d'entrée doit être réduit cinq fois pour éliminer les débordements. Lorsque le signal est réduit cinq fois, les pics attendus apparaissent. Le nombre total de cycles pour cette FFT est de 56 383. Étant donné que cela est inférieur au temps de capture disponible

pour une trame de données de 128 points à une fréquence d'échantillonnage de 8 kHz, il est prévu que cet algorithme s'exécute en temps réel sur le DSK.

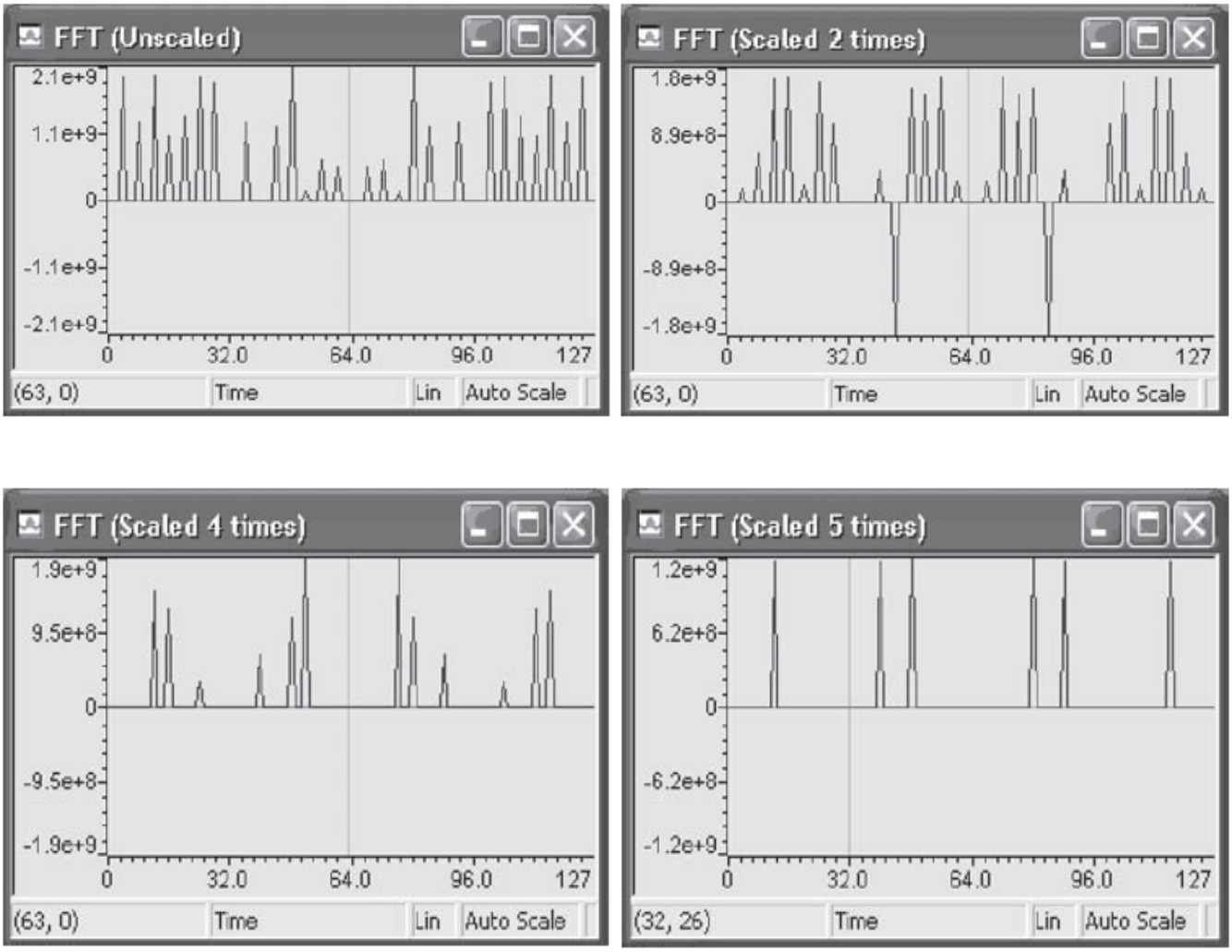


Fig. 6.12 : Mise à l'échelle pour obtenir une réponse d'amplitude FFT correcte.

Références bibliographiques

1. Nasser Kehtarnavaz, *Real-Time Digital Signal Processing*., 2005, Elsevier Inc. ISBN 0-7506-7830-5.
2. Texas Instruments, *TMS320C6201/6701 Evaluation Module Reference Guide*, Literature ID# SPRU 269F, 2002.
3. J. Proakis and D. Manolakis, *Digital Signal Processing:Principles, Algorithms, and Applications*, Prentice-Hall, 1996.
4. S. Mitra, *Digital Signal Processing: A Computer-Based Approach*, McGraw Hill, 1998.
5. B. Razavi, *Principles of Data Conversion System Design*, IEEE Press, 1995.
6. Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, Literature ID# SPRU 189F, 2000.
7. Texas Instruments, *Technical Training Notes on TMS320C6x*, TI DSP Fest, Houston, 1998.
8. Texas Instruments, *TMS320C6000 Assembly Language Tools User's Guide*, Literature ID# SPRU 186M, 2003.
9. Texas Instruments, *TMS320C6000 Optimizing Compiler User's Guide*, Literature ID# SPRU 187K, 2002.
10. Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, Literature ID# SPRU 189F, 2000.
11. S. Tretter, *Communication System Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711,* Kluwer Academic Publishers, 2003.
12. Sen M Kuo, Bob H Lee, *Real-Time Digital Signal Processing*, 2001 John Wiley. ISBN 0-470-84534-1.
13. B. A. Shenoi, *Introduction to Digital Signal Processing*, 2006 by John Wiley & Sons, Inc. ISBN-13 978-0-471-46482.
14. The Mathworks, *MATLAB Reference Guide*, 1999.
15. S. Haykin, *Adaptive Filter Theory*, Prentice-Hall, 1996.1
16. Texas Instruments, *Application Report SPRA 291*, 1997.