République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

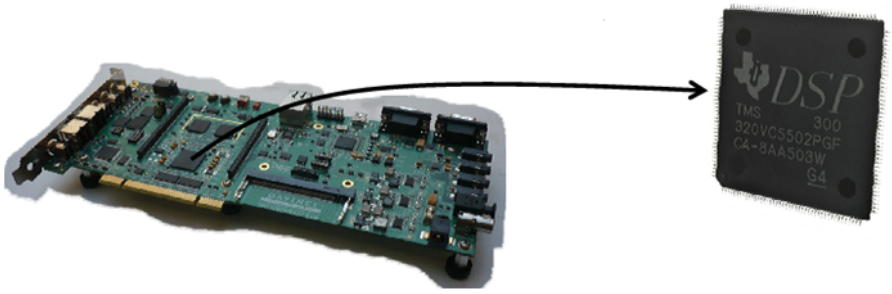
جامعة مصطفى اسطنبولي معسكر
كلية العلوم و التكنولوجيا
قسم الالكترونيك
Université de Mustapha Stambouli Mascara
Faculté des Sciences et Technologies
Département d'Electronique

Polycopié de Cours

Processeurs numériques du signal DSP(Digital Signal Processor)

Présenté par :

M. BOUGUENNA Ibrahim Farouk



2020/2021

Avant-propos

Ce cours est rédigé à l'intention des étudiants d'Electronique inscrits en première année master, option :Electronique des Systèmes Embarqués. Il peut aussi etre utilement utilisé par les étudiants d'autre paliers en génie électrique ou autre. Cet ouvrage traite l'architecture et les caractéristiques matérielles des processeurs de signaux numériques. Le cours n'enseigne pas en détail la théorie du traitement numérique du signal, ni la programmation informatique, car il suppose une certaine connaissance des deux. A travers ce fasicule les étudiants peuvent apprécier les concepts de DSP grâce à l'implémentation en temps réel d'expériences et de projets. J'espère que le cours pourra être utile en tant que : - Un cours de développement académique pour les étudiants - Un cours d'introduction pour les étudiants en thèse travaillant en DSP.

Ce polycopié est structuré en six chapitres comme suit :

Dans le premier chapitre, nous avons donné des généralités sur les processeurs DSP.

Le deuxième chapitre est dédié à l'Arithmétique à virgule fixe et à virgule flottante.

Le troisième chapitre décrit l'architecture des DSP TMS320C6x.

Le quatrième chapitre concerne la gestion de la mémoire.

L'Environnement de développement : 'Code Composer Studio' (CCS) est présenté dans le cinquième chapitre.

Enfin, dans le sixième chapitre nous décrivons les algorithmes de traitement du signal sur DSP.

Ce document représente le fruit de cinq années d'enseignement de la matière "DSP et FPGA" dispensé aux étudiants de première année Master Réseaux et Télécommunications et trois années d'enseignement de la matière "Processeurs numériques du signal " pour les étudiants de la première année master Electronique des Systèmes Embarqués à l'université Mustapha Stambouli de Mascara.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des sigles et acronymes

DSP	*Digital Signal Processor*
GPS	*Global positioning System*
HDTV	*High-Definition Television*
FPGA	*Field-Programmable Gate Arrays*
ASIC	*Application-Specific Integrated Circuits*
PLL	*Phase-Locked Loop*
HPI	*Host-Port Interface*
GPIO	*General-Purpose Input/Output*
SRAM	*Static Random Access Memory*
DDRAM	*Double Data Rate Random Access Memory*
EMIF	*External Memory InterFace*
EDMA	*Enhanced Direct Memory Access*
CCS	*Code Composer Studio*
DIP	*Dual In-line Package*
LED	*Light-Emitting Diode*
GEL	*General Extension Language*
RIF	*Réponse Impulsionnelle Finie*
RII	*Réponse Impulsionnelle Infinie*

FFT	*Fast Fourier Transform*
A/N	*Analogique-Numérique*
N/A	*Numérique-Analogique*
CI	*Circuit Intégré*
VLSI	*Very Large Scale Integration*
MAC	*Multiply and Accumulate*
E/S	*Entrée / Sortie*
CPU	*Central Processing Unit*
Te	*Temps d'échantillonnage*
KSPS	*kilo Sample Per Second*
MSPS	*Mega Sample Per Second*
RF	*Radio Frequency*
PCM	*Pulse Code Modulation*
HF	*Haute Fréquence*
IRM	*Imagerie à Résonance Magnétique*
FIR	*Finite Impulse Response*
DFT	*Discrete Fourier Transform*
FFT	*Fast Fourier Transform*
DWT	*Discrete Wavelet Transform*
DCT	*Discrete Cosine Transform*
JPEG	*Joint Picture Experts Group*
MPEG	*Moving Picture Experts Group*
MP3	*Moving Picture Experts Group-Audio Layer 3*

DMA *Direct Memory Access*

ADC *Analog Digital Converter*

DAC *Digital Analog Converter*

Introduction générale

Dans notre monde moderne, des signaux de toutes sortes émanent de différents types d'appareils - radios et téléviseurs, téléphones portables, systèmes de positionnement global (GPS), radars et sonars. Ces systèmes nous permettent de communiquer des messages, de contrôler des processus et de détecter ou mesurer des signaux. Au cours des 70 dernières années, avec l'avènement du transistor, de l'ordinateur et des fondements théoriques du traitement numérique du signal, la tendance a été à la représentation et au traitement numériques des données, qui dans de nombreuses applications se présente sous forme analogique. Une telle tendance souligne l'importance d'apprendre à représenter des signaux sous forme analogique et numérique et à modéliser et concevoir des systèmes capables de traiter différents types de signaux.

L'année 1948 est considérée comme l'année où sont nées les technologies et les théories responsables des progrès spectaculaires des communications, du contrôle et du génie biomédical. En effet, en juin de cette même année, Bell Telephone Laboratories a annoncé l'invention du transistor. Plus tard dans le mois, un ordinateur prototype construit à l'Université de Manchester au Royaume-Uni est devenu le premier ordinateur à programme stocké opérationnel. Cette année-là également, des résultats théoriques fondamentaux ont été publiés : la théorie mathématique des communications de Claude Shannon, la théorie de Richard W. Hamming sur les codes de correction d'erreurs et la cybernétique de Norbert Wiener comparant les systèmes biologiques aux systèmes de communication et de contrôle.

Les progrès du traitement numérique du signal sont allés cote-à-cote avec les progrès de l'électronique et des ordinateurs. En 1965, Gordon Moore, l'un des fondateurs d'Intel, prévoyait que le nombre de transistors sur une puce doublerait environ tous les deux ans. Ce sont ces avancées en électronique numérique et en génie informatique qui ont permis la prolifération des technologies numériques. Aujourd'hui, le matériel et les logiciels numériques traitent les signaux des téléphones portables, des récepteurs de télévision haute définition (HDTV), de la radio numérique, des radars et des sonars, pour n'en nommer que quelques-uns. L'utilisation de processeurs de signaux numériques (DSP) et les circuits programmables (FPGA) a remplacé l'utilisation de circuits intégrés spécifiques aux applications (ASIC) dans les applications industrielles, médicales et militaires.

Il est clair que les technologies numériques sont là pour rester. L'abondance d'algorithmes de traitement des signaux numériques et la présence omniprésente de DSP et de FPGA dans des milliers d'applications font de la théorie du traitement du signal numérique un outil nécessaire non seulement pour les ingénieurs, mais aussi pour quiconque

traiterait des données numériques - bientôt, ce sera tout le monde !.

Le traitement numérique du signal est effectué par des opérations mathématiques. En comparaison, le traitement de texte et les programmes similaires ne font que réorganiser les données stockées. Cela signifie que les ordinateurs conçus pour les entreprises et d'autres applications générales ne sont pas optimisés pour des algorithmes tels que le filtrage numérique et l'analyse de Fourier. Les processeurs de signaux numériques sont des microprocesseurs spécialement conçus pour gérer les tâches de traitement de signaux numériques. Ces composant ont connu une croissance considérable au cours de la dernière décennie, trouvant une utilisation dans tout. En fait, les ingénieurs en hardware utilisent « DSP » pour désigner un processeur de signal numérique, tout comme les développeurs d'algorithmes utilisent « DSP » pour désigner le traitement du signal numérique. Ce cours examine en quoi les DSP sont différents des autres types de microprocesseurs, comment décider si un DSP convient à votre application et comment se lancer dans ce nouveau domaine passionnant.

Pour rappel ce support est organisé comme suit :

Dans le premier chapitre on va présenté des généralités sur les processeurs DSP ,les différentes familles de DSP, classification des DSP, les domaines d'applications des DSP , les principaux algorithmes traités seront discuté, une comparaison des processeurs DSP avec d'autres approches, et à la fin l'historique et évolutions récentes.

La représentation des signaux numérique est abordé dans le deuxième chapitre. L'echantilonnage et la quantification uniforme (caractéristique, caractéristique de l'erreur, dynamique), quantification non-uniforme, quantification logarithmique (loi de compression expansion, approximations par segments des lois de compression A et µ), formats de représentations des nombres, et dans la dernière partie on discute le codage des nombres entiers (entiers positifs ou non signés, complément à 1, complément à 2),la représentation des nombres réels dans un calculateur (virgule fixe, virgule flottante).

Le chapire trois est consacré à l'architecture des DSP TMS320C6x , en commençons par l'architecture interne des processeur de la famille C6000,puis la cartographie de mémoire, les unités fonctionnelles, les paquets d'exécution et de fetch, l'architecture pipeline, les registres, les registres de contrôle, les périphériques (timers, PLL, interruptions, HPI, GPIO), la liaison série (multichannel buffered serial port), et on conclu par la présentation du jeu d'instructions.

Le chapitre quatre est dédié à la Gestion de la mémoire. Un détail sera discuté sur l'intérêt de l'architecture Harvard, les mémoires internes (niveaux L1 et L2),les mémoires externes (SRAM, Flash, DDRAM, ...) , le plan d'adressage des mémoires, fichier *.cmd (organisation des sections), la gestion de la mémoire externe par L'EMIF (External Memory InterFace). On va discuté aussi les modes d'adressage (indirect, circulaire)et les technique de transfert par blocs. On termine par une description sur l'organisation des données pour l'EDMA et les paramètres et options pour l'EDMA.

L'environnement de développement : 'Code Composer Studio' (CCS) est présenté dans le cinquième chapitre, la configuration de base 'Basic Setup', la création d'un nouveau projet sous CCS, l'exécution du programme (Break Point, Watch Window, Plots, Images,

enregistrement de données), scriptes GEL (General Extension Language) du CCS, utilisation des switches DIP et des LEDs seront illustrés dans ce chapitre.

Au chapitre finale nous nous intéréssons aux algorithmes de traitement du signal sur DSP et son adéquation avec l'architecture, puis le filtrage RIF et RII , ensuite les buffers à décalage et circulaire,les problèmes de quantification,les contraintes temps-réel, la gestion des entrées/sorties. A la fin de ce chapitre on présente l'Implémentation de la FFT sur DSP.

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 et en amplitude continus). Le traitement des signaux peut être effectué 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 à la fois en temps et en amplitude. Le processus de numérisation est réalisé via un convertisseur analogique-numérique (A / N).

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 demeure nécessaire de s'interfacer avec le monde analogique dans lequel nous vivons. Les convertisseurs de données analogique-numérique (A / N) et numérique-analogique (N / A) sont les dispositifs qui rendent cette interface possible. La Figure 1.1 illustre les principaux composants d'un système DSP, composé de périphériques A / N, DSP et N / A.

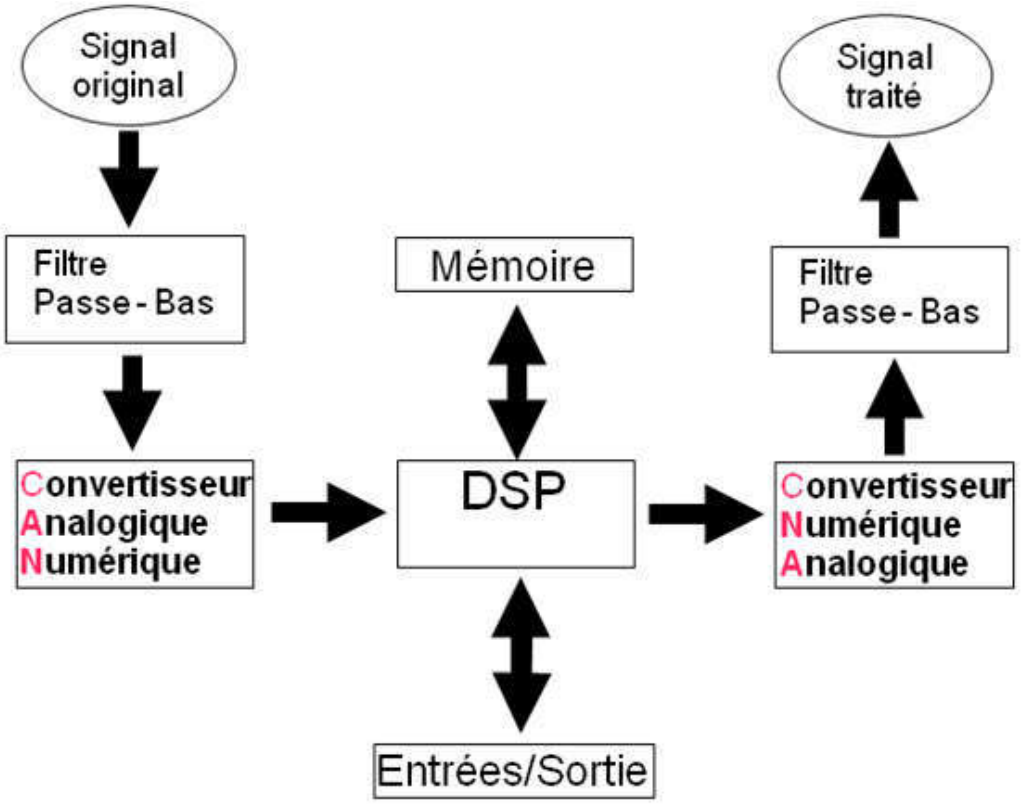


Fig. 1.1 : Principaux composants d'un système DSP.

1.2 Les avantages des systèmes DSP's

Le traitement du signal numérique bénéficie de plusieurs avantages par rapport au traitement du signal analogique. Le plus important d'entre eux est que les systèmes DSP sont capables d'accomplir à peu de frais des tâches qui seraient difficiles voire impossibles en utilisant l'électronique analogique. Des exemples de telles applications comprennent la synthèse vocale, la reconnaissance vocale et les modems à grande vitesse impliquant un codage de correction d'erreurs. Toutes ces tâches impliquent une combinaison de traitement et de contrôle du signal (par exemple, la prise de décisions concernant les bits reçus ou la parole reçue) qui est extrêmement difficile à mettre en œuvre en utilisant des techniques analogiques.

Les systèmes DSP bénéficient également de deux avantages supplémentaires par rapport aux systèmes analogiques :

- **Insensibilité à l'environnement.**Les systèmes numériques, de par leur nature même, sont considérablement moins sensibles aux conditions environnementales que les systèmes analogiques. Par exemple, le comportement d'un circuit analogique dépend de sa température. En revanche, sauf en cas de pannes catastrophiques, le fonctionnement d'un système DSP ne dépend pas de son environnement - que ce soit dans la neige ou dans le désert, un système DSP offre la même réponse.

- **Insensibilité aux tolérances des composants.** Les composants analogiques sont fabriqués selon des tolérances particulières - une résistance, par exemple, peut être garantie d'avoir une résistance à moins de 1% de sa valeur nominale. La réponse globale d'un système analogique dépend des valeurs réelles de tous les composants analogiques utilisés. En conséquence, deux systèmes analogiques exactement de la même conception auront des réponses légèrement différentes en raison de légères variations de leurs composants. En revanche, les composants numériques fonctionnant correctement produisent toujours les mêmes sorties avec les mêmes entrées.

Ces deux avantages se combinent en synergie pour donner aux systèmes DSP un avantage supplémentaire par rapport aux systèmes analogiques :

- **Comportement prévisible et reproductible.** Etant donné que la sortie d'un système DSP ne varie pas en raison de facteurs environnementaux ou de variations de composants, il est possible de concevoir des systèmes ayant des réponses exactes et connues qui ne varient pas.

Enfin, certains systèmes DSP peuvent également présenter deux autres avantages par rapport aux systèmes analogiques :

- **Reprogrammabilité.** Si un système DSP est basé sur des processeurs programmables, il peut être reprogrammé - même sur le terrain - pour effectuer d'autres tâches. En revanche, les systèmes analogiques nécessitent des composants physiquement différents pour effectuer différentes tâches.

- **Taille.** La taille des composants analogiques varie avec leurs valeurs ; par exemple, un condensateur de 100 µF utilisé dans un filtre analogique est physiquement plus grand qu'un condensateur 10 pF utilisé dans un filtre analogique différent. En revanche, les implémentations DSP des deux filtres pourraient bien être de la même taille - en fait, pourraient même utiliser le même matériel, ne différant que par leurs coefficients de filtre - et pourraient être plus petites que l'une ou l'autre des deux implémentations analogiques.

Ces avantages, associés au fait que le DSP peut tirer parti de la densité rapidement croissante des procédés de fabrication de circuits intégrés numériques (CI), font de plus en plus du DSP la solution de choix pour le traitement du signal.

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. Il existe une assez grande flexibilité d'application associée à l'implémentation du DSP, puisque 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 rentables car ils sont produits en masse et peuvent être utilisés pour de nombreuses applications. Une puce VLSI personnalisée est normalement construite 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 faire.

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

1.3 Les caractéristiques d'un processeur DSP

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

1. Ils sont optimisés pour faire face à la répétition ou au bouclage des 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 (MAC) à cycle unique.

MAC ou **"Multiply and Accumulate"** : Se sont des instructions DSP à double opérande, sont couramment utilisées pour effectuer l'opération principale somme de produit (sum of product) dans les filtres à réponse impulsionnelle finie ou infinie, ainsi que dans plusieurs autres algorithmes DSP. Le caractère unique des instructions de cette catégorie est qu'elles ont une vue « fractionnée » de l'espace de données, ce qui leur permet d'accéder simultanément aux données des espaces de données X et Y. L'opération MAC de base est $A = A + x * y$, où A est un accumulateur et x et y sont les opérandes source. Un microprocesseur classique va nécessiter plusieurs cycles d'horloge pour effectuer un tel calcul, par exemple, un 68000 à besoin de :

- 10 cycles d'horloge pour effectuer une addition,
- 70 cycles d'horloge pour effectuer une multiplication.

Soit 80 cycles pour seulement calculer A. Si ce temps est admissible dans des applications informatiques courantes, il n'est pas acceptable pour faire du traitement rapide du signal. Les DSP sont donc conçus pour optimiser ce temps de calcul, à cet effet, ils disposent de fonctions optimisées permettant de calculer A beaucoup plus rapidement.

Dans la pratique, la plupart des DSP ont un jeu d'instructions spécialisé permettant de lire en mémoire une donnée, d'effectuer une multiplication puis une addition, et enfin d'écrire en mémoire le résultat, le tout en un seul cycle d'horloge. Ce type d'opération est nommé MAC, de l'anglais Multiply and Accumulate. Comme le montre le diagramme ci-dessous, en exécutant à plusieurs reprises une instruction MAC, il est possible d'obtenir la somme des produits, de 2 tableaux, notés ici X et Y.

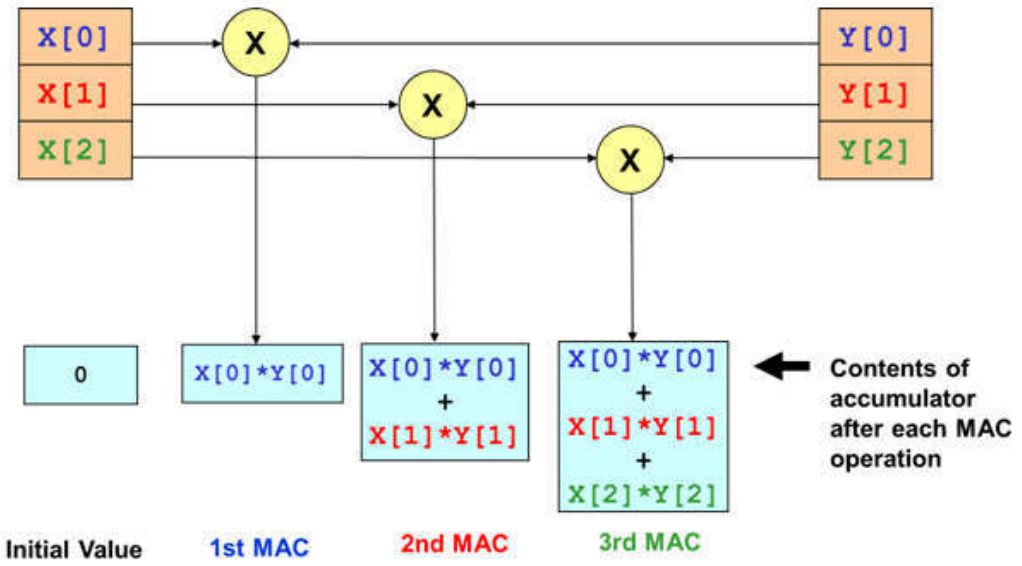


FIG. 1.2 : Opération MAC.

Comme le montre le diagramme ci-dessus, en exécutant à plusieurs reprises une

instruction MAC, il est possible d'obtenir la somme des produits, de 2 tableaux, notés ici X et Y.

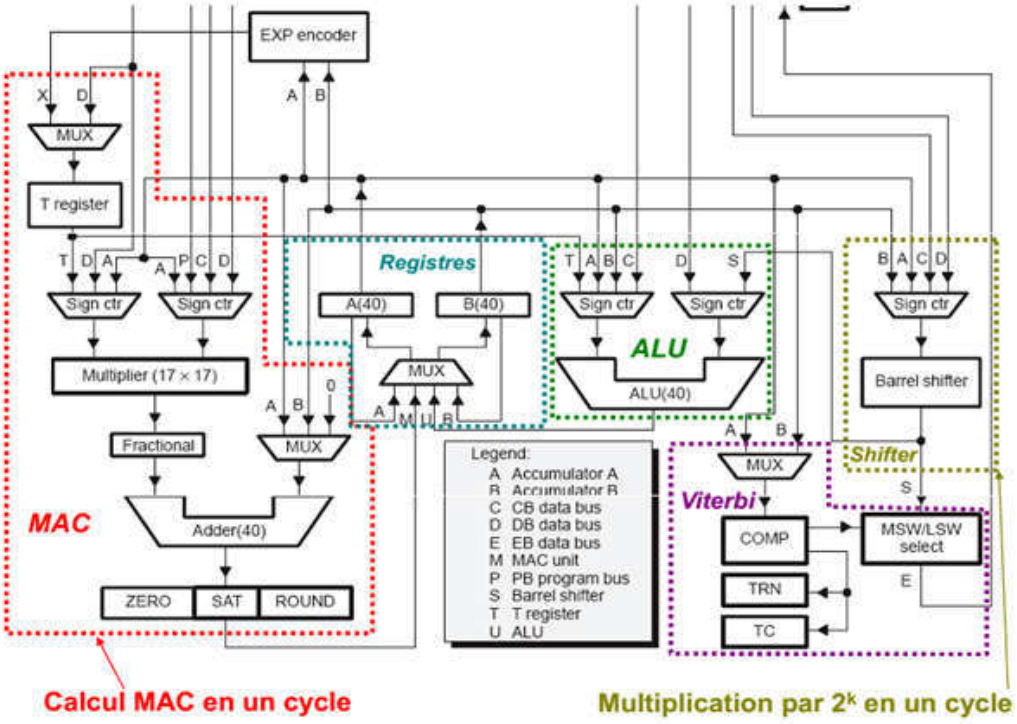


Fig. 1.3 : Structure interne du DSP TMS320C5416.

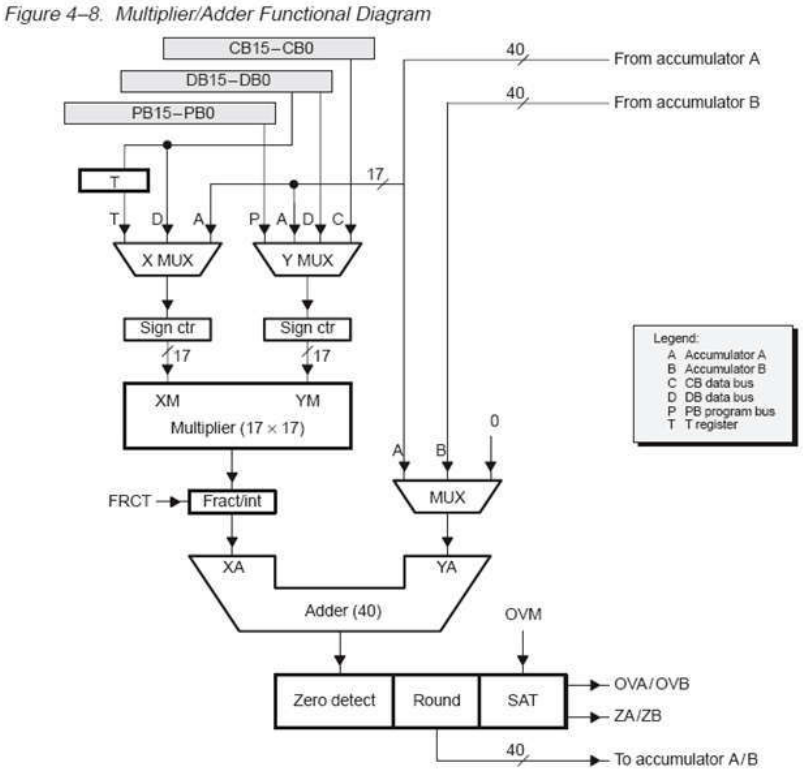


Fig. 1.4 : Unité MAC du DSP TMS320C5416.

2. Les DSP permettent des modes d'adressage spécialisés, comme l'adressage indirect et circulaire. Ce sont des mécanismes d'adressage efficaces pour mettre en œuvre 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.

5. Les processeurs DSPs doivent exécuter des taches en temps réel, alors qu'elle est la spécification d'exécution en temps réel ?
   Le temps réel c'est terminer le traitement dans le délai tolérable ou disponible entre les échantillons. Il faut que 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, ou à un intervalle d'échantillonnage d'environ 22,6 µs, le nombre d'instructions doit être inférieur que près de 4500, en supposant un temps de cycle d'instruction de 5 ns.
   Te/ temps de cycle d'instruction=nombre d'instructions possible entre deux échantillons.

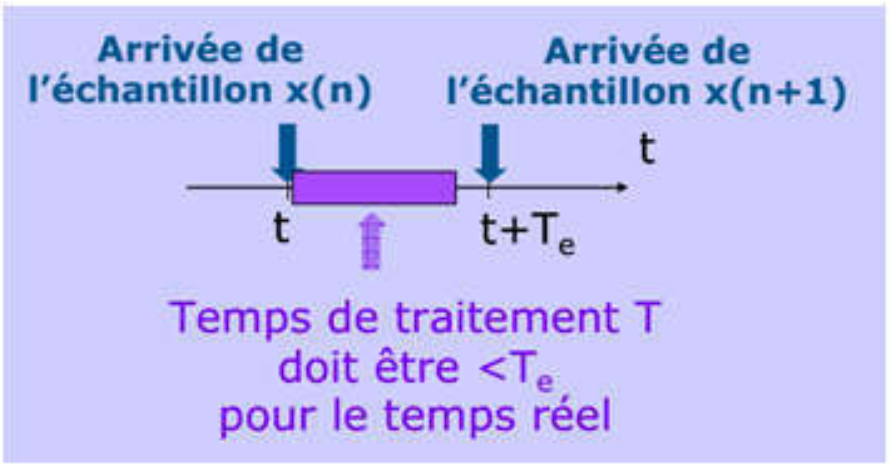


Fig. 1.5 : Contrainte temps réel.

1.4 Exemples des systemes DSP

Pour que le lecteur apprécie l'utilité des DSP, plusieurs exemples de systèmes DSP actuellement utilisés sont présentés ici.
Au cours des dernières années, le marché du sans fil a connu une croissance considérable. La Figure 1.6 illustre un système DSP de communication sans fil de téléphone cellulaire. Comme le montre cette figure, il existe deux ensembles de convertisseurs de données.

Du côté dc la bandc vocalc, un faiblc taux d'échantillonnagc (par cxcmplc, 8 kSPS [kilo sample per second]) et un Un convertisseur haute résolution (par exemple 13 bits) est utilisé, tandis que du côté modulation RF, un convertisseur relativement rapide (par exemple 20 MSPS) et basse résolution (par exemple 8 bits) est utilisé. Les concepteurs de systèmes préfèrent intégrer plus de fonctionnalités dans le DSP que dans les composants analogiques afin de réduire le nombre de composants et donc le coût global. Cette stratégie d'intégration accrue dans DSP dépend des spécifications réalisables pour une faible consommation d'énergie dans les appareils portables.

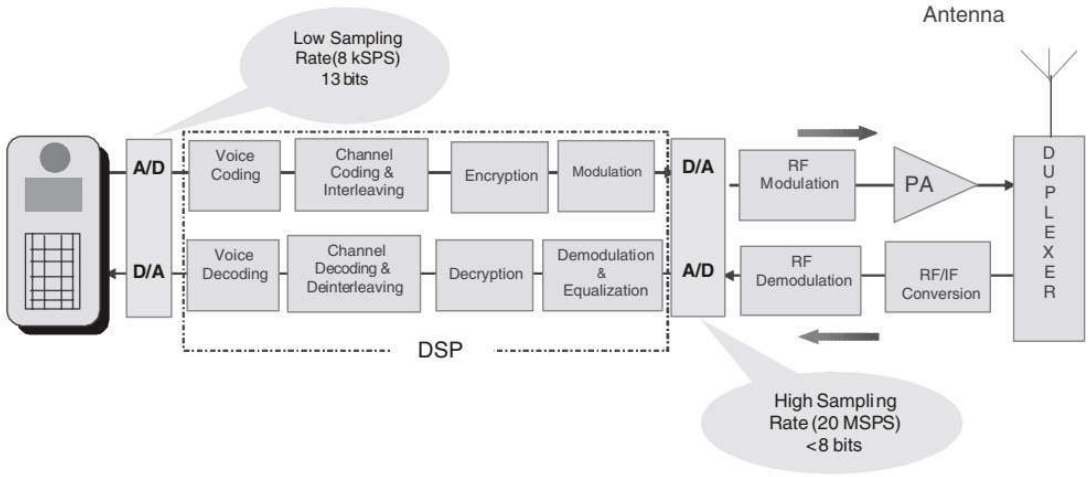


Fig. 1.6 : Système DSP de communication sans fil pour téléphone portable.

Étant donné que les réseaux de communication utilisés aujourd'hui sont numériques, un signal analogique atteignant le central de la compagnie de téléphone doit être conditionné et converti en un signal numérique pour être transmis à travers le réseau. La Figure 1.7 montre le codec de bande vocale à modulation par impulsions codées (PCM) utilisé dans les réseaux de communication. Comme on peut le voir, une bonne partie du traitement du signal est effectuée dans le domaine numérique par le composant DSP.

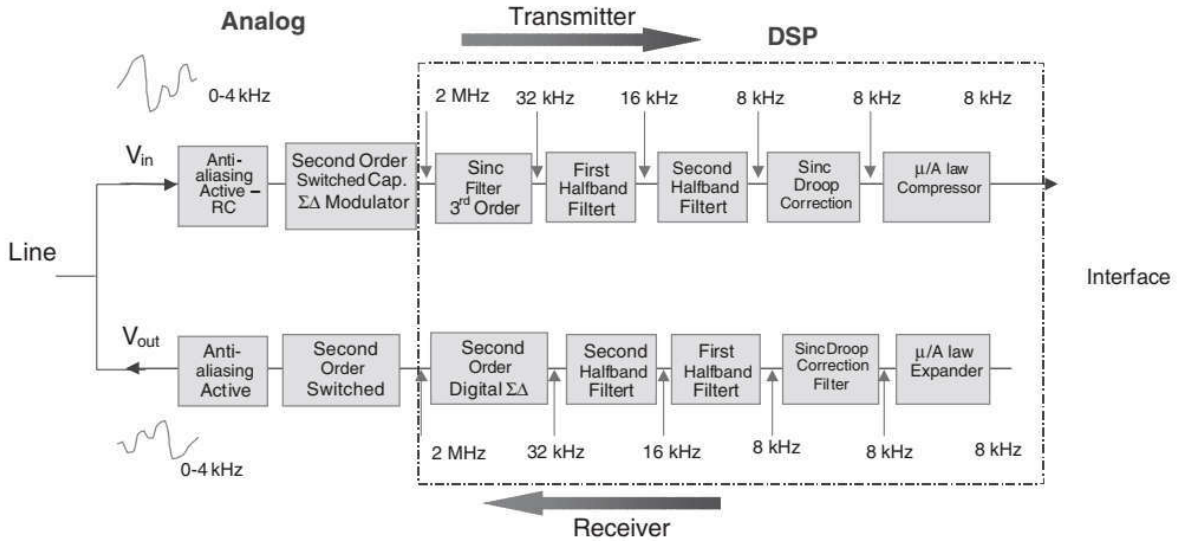


Fig. 1.7 : Système DSP de bande vocale PCM.

Le contrôle du moteur est un autre domaine dans lequel les DSP ont un impact. Par exemple, comme illustré sur la figure 1.8, les DSP sont utilisés pour contrôler les moteurs à induction via la surveillance des signaux de retour, y compris le courant, la tension et la position. Ces moteurs sont largement utilisés en raison de leur faible coût, de leur grande fiabilité et de leur rendement élevé.

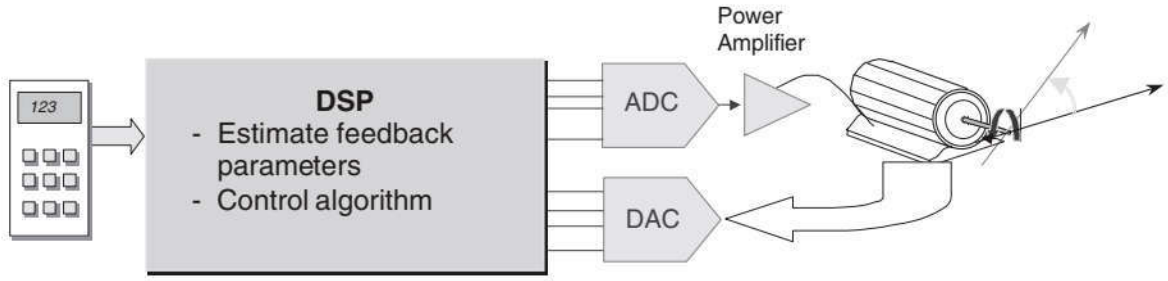


Fig. 1.8 : Contrôle d'un moteur par un Système DSP.

Les capteurs ou dispositifs intelligents sont un autre exemple de systèmes DSP. Ces capteurs sont capables à la fois d'acquérir et de traiter des données. Un exemple de tels capteurs est le système d'activation d'airbag dans les automobiles. L'accélération du véhicule est mesurée par un capteur de masse de suspension et convertie en un signal numérique par un convertisseur A / N. Ce signal est ensuite traité par un DSP pour détecter un accident en comparant les caractéristiques du signal avec celles de l'accident.

1.5 Les grandes familles des DSPs

Le marché est partagé entre quatre constructeurs principaux :

- Texas Instruments,
- Analog Devices,
- Freescale (Motorola) et
- Lucent.

1.6 Classification des DSP's

Les DSP se différencient par

- le format de calcul (fixe ou flottant),
- la taille du bus de données (16, 24 ou 32 bits),
- la puissance en millions d'instructions par secondes (MIPS) et

Site Web	Compagnie	Rang
www.ti.com	Texas Instrument	1
www.freescale.com	Freescale Semiconductor	2
www.analog.com	Analog Devices	3
www.nxp.com	Philips Semiconductors	4
www.Lsi.com	Agere Systems	5
www.toshiba.com	Toshiba	6
www.dspg.com	DSP Group	7
www.necel.com	NEC Electronics	8

TAB. 1.1 : Différentes familles des DSP

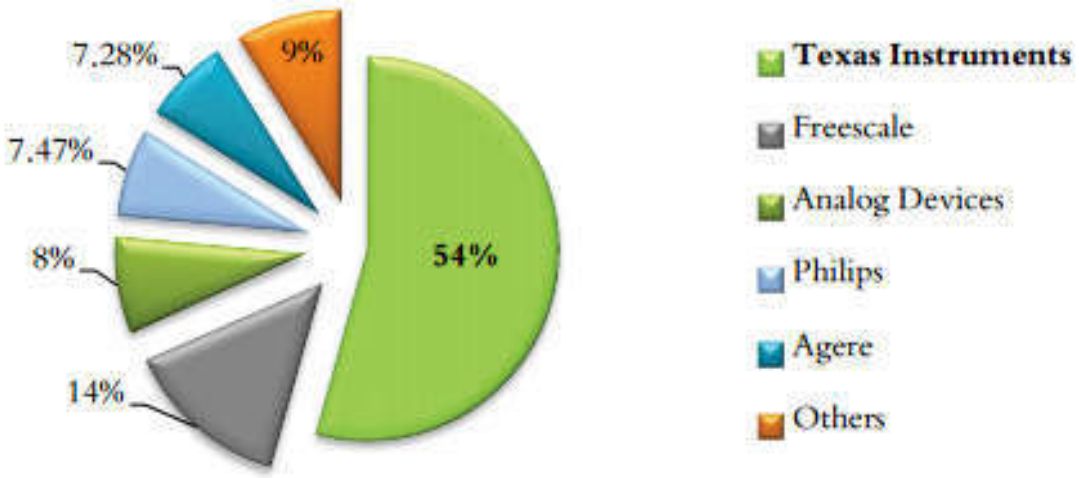


FIG. 1.9 : Marché de vente des DSPs par fabricant.

- les fonctionnalités spécifiques directement intégrées (traitement du son, de l'image, etc.)

On peut aussi classer les DSP en deux grandes catégories / au format des données :

1.6.0.1 A. DSP à virgule fixe

Dans ce cas, les données sont représentées comme des fractions d'entiers qui sont, par exemple, comprises entre -1.0 et 1.0, ou comme des entiers simples. L'avantage de la fraction est de permettre une addition binaire simple des nombres positifs et négatifs (via un complément à 2).
Contrairement aux apparences, un DSP à virgule fixe est plus complexe à programmer qu'un DSP à virgule flottante. Si on prend comme exemple le TMS230C25 de Texas Instrument, les nombres sont codés sur 16 bits mais les calculs sont effectués avec une précision de 32 bits, les 16 bits les moins significatifs étant perdus. Ainsi, le calcul étant sur 32 bits, on évite des erreurs cumulatives. Et même si les registres (32 bits) sont pleins, on peut toujours stocker les données en RAM (16 bits) via deux variables : 16 bits de poids faible + 16 bits de poids fort.

Il faut savoir que certains DSP 16 bits à virgule fixe ne possèdent pas de registre 32 bits. Dans ce cas, la précision est moindre. Et si on doit effectuer un calcul précis, il faudra séparer les parties du calcul en 16 bits et donc effectuer plus de calcul successifs, au détriment du temps. Ce qui nécessite un peu d'astuce et des concepteurs minutieux pour prévoir à l'avance la précision voulue et optimiser les calculs.

Cependant, les DSP à virgule fixe sont plus rapides, c'est donc aussi un critère de choix. Mais avec le temps, les DSP à virgule fixe sont moins développés et on été rattrapés par ceux à virgule flottante dont allons maintenant donner une description.

1.6.0.2 B. DSP à virgule flottante

Ces DSP sont bien plus commodes pour les développeurs. Les données sont représentées avec une mantisse et un exposant selon la formule : $n = mantisse * 2^{exposant}$. On utilise souvent une mantisse fractionnaire comprise entre -1.0 et 1.0. L'exposant indique la place de la virgule en base 2.

Prenons par exemple le TMS320C30 : la mantisse est codée sur 24 bits et l'exposant sur 8 bits, soit 32 bits en mémoire. Lors des calculs, les résultats intermédiaires sont mis en registre avec une mantisse de 32 bits et toujours un exposant sur 8 bits.

Mais le DSP sait aussi manipuler des entiers, donc avec une précision de 32 bits. Ainsi, cette grande plage permet de développer sans vraiment trop se préoccuper de la précision. C'est surtout le prix qui limitera le développement (nombre de broches plus important, surface de silicium du cœur doublée…).

Et pourtant, ils sont très utilisés car plus souples pour des utilisations particulières où :

- Les coefficients dépendent du temps (filtre adaptif) ;
- La structure mémoire est importante (traitement d'images) ;
- La précision est primordiale, sur un intervalle relativement large.

1.6.0.3 C. Avantages pour la virgule flottante

- Très grande plage dynamique.
- Applications avec calculs intensifs.
- Plus le nombre n'est petit, plus la précision et le bruit de quantification sont élevés.
- Calcul possible pour les grands nombres, au-delà de la capacité du point fixe.
- Développement plus simple et plus rapide sans se soucier des erreurs de débordement et de troncature.

1.6.0.4 D. Avantages pour la virgule fixe

- Meilleure résolution dans une plage étroite.
- Silicium DSP plus simple.

	DSP à virgule fixe	**DSP à virgule flottante**
Prix	Moin cher	Plus cher
Volume	Petit	Grand
Consommation d'énergie	Faible	Fort
Programmation	Complexe	Simple
Précision	Précis	Moins précis
Mémoire	Réduite	Large
Application	95% Application	Utilisée dans les applications avancées
Taille	16/24/36/40/56/64bits	32/40/44/96bits

Tab. 1.2 : Comparaison entre les deux catégories des DSPs

1.7 Domaines d'apllication des DSP's

Les DSP sont utilisés dans une grande variété d'applications offline et en temps réel. Certains domaines typiques et applications spécifiques sont :

Telecommunications : modems de ligne téléphonique, FAX, téléphones cellulaires,haut-parleur, transcodeurs, interpolation numérique de la parole, systèmes sans fil à large bande et répondeurs.

Voix / parole : numérisation et compression de la parole, messagerie vocale, vérification du locuteur et synthèse vocale.

Automobile : contrôle du moteur, freins antiblocage, suspension active, contrôle de l'airbag et diagnostic du système.

Systèmes de contrôle : systèmes d'asservissement de positionnement de la tête dans les lecteurs de disque, commande d'imprimante laser, commande de robot, commande de moteur et de moteur et commande numérique de machines-outils automatiques.

Militaire : traitement des signaux radar et sonar, systèmes de navigation, guidage de missiles, modems à radiofréquence HF, radios sécurisées à spectre étalé et voix sécurisée.

Médical : prothèses auditives, imagerie IRM, échographie et surveillance des patients.

Instrumentation : analyse spectrale, analyse transitoire, générateurs de signaux.

Traitement d'image : HDTV, reconnaissance de formes, amélioration de l'image, compression et transmission d'image, rotation 3D et animation.

1.8 Performances des DSP's

La plupart des DSP sont particulièrement destinés à des applications « temps réel » et spécialisées, c'est à dire des applications où le temps de traitement est bien sûr primordial, mais où la diversité des événements à traiter n'est pas notablement importante. De ce point de vue, l'approche DSP s'apparente plus à une étude « électronique » visant à réaliser une ou des fonctions de traitements de signal, que d'une approche informatique temps réel et/ou multitâche traditionnelle.

Dans tous les cas, les performances du DSP sont critiques. Le concepteur d'un système à base de DSP doit évaluer d'une part la « puissance » nécessaire pour réaliser les traitements numériques voulus, et d'autre part les performances des DSP disponibles pour réaliser son application.

1.8.1 Mesure de vitesse de calcul pure

La méthode classique pour évaluer les performances d'un DSP est de se baser sur sa vitesse d'exécution. Encore faut-il trouver une bonne définition de ce qu'est la vitesse d'exécution, ce qui n'est pas forcément simple.

Cette méthode de base consiste donc à compter le nombre d'instructions effectuées par seconde. Un obstacle apparaît alors, car une instruction ne signifie pas forcément la même chose d'une famille de DSP à l'autre. Le tableau suivant résume les principales définitions en usage. |p3cm|p3cm|

Une autre méthode consiste à définir une fois pour toute une opération de référence comme étant un MAC, puisqu'il s'agit d'une fonction commune à tous les DSP. Il ne reste plus qu'à compter le nombre de MAC par seconde.

Cependant cette définition n'apporte pas beaucoup d'informations sur les performances des DSP modernes. En effet, un MAC est exécuté en un seul cycle. Sachant que sur les DSP récents la plupart des instructions sont également exécutées en un cycle, cela revient donc à mesurer les MIPS du DSP. Il faut également tenir compte du fait que certains DSP en font plus dans un seul MAC (nombre, format et taille des opérandes traités) que d'autres.

1.8.2 Mesure du temps d'exécution

La vitesse de calcul pure d'un DSP n'est pas une indication universelle, les méthodes et les résultats diffèrent d'un DSP à l'autre. De plus elle ne rend pas compte d'un certain nombre de perfectionnement dont peuvent bénéficier tel ou tel DSP.

Acronyme Anglais	Définition
MFLOPS	Million Floating-Point Operations Per Second. Mesure le nombre d'opérations à virgule flottante (multiplications, additions, soustractions, etc.) que le DSP à virgule flottante peut réaliser en une seconde.
MOPS	Million Operations Per Second. Mesure le nombre total d'opérations que le DSP peut effectuer en une seconde. Par opérations, il faut comprendre non seulement le traitement des données, mais également les accès DMA, les transferts de données, les opérations d'E/S, etc. Cette définition mesure donc les performances globales d'un d'un DSP plutôt que ses seules capacités de calcul.
MIPS	Million Instructions Per Second. Mesure le nombre de codes machines (instructions) que le DSP peut effectuer en une seconde. Bien que cette mesure s'applique à tous les types de DSP, le MFLOPS est préféré dans le cas d'un DSP à virgule flottante.
MBPS	Mega-Bytes Per Second. Mesure la largeur de bande d'un bus particulier ou d'un dispositif d'E/S, c'est à dire son taux de transfert.

Tab. 1.3 : Définitions des unités les plus courantes de mesures des performances des DSP.

Certains DSP proposent en effet des modes d'adressages plus performants que d'autres. Ces modes sont spécialement adaptés à des algorithmes standards du traitement du signal (exemple : le mode d'adressage dit « bits reversing » pour accélérer les calculs des FFT). Les instructions itératives sont également importantes en terme de performance (rapidité des boucles logicielles) et ne devraient pas être ignorées.

Enfin, le temps d'accès à la mémoire est un autre paramètre incontournable. Certains DSP intègrent des blocs de mémoire vive rapide. Cette mémoire est placée dans l'espace d'adressage du DSP au même titre que de la mémoire vive externe, ce qui permet d'y ranger données et programmes sans avoir à effectuer des transfère permanents de ou vers l'extérieur. Les éventuels cycles d'attentes pouvant être nécessaires pour adresser une mémoire externe lente sont ainsi évités.

Pour toutes ces raisons, la mesure des performances par benchmark complète avantageusement la mesure de vitesse pure. Elle consiste à mesurer le temps que met le DSP pour exécuter des programmes « standards » de traitements du signal. Encore faut il définir ce qu'est un programme standard de traitement du signal.

Le point faible des benchmarks réside dans cette définition des d'algorithmes standards. Quel domaine d'applications faut il choisir ? Quels sont les algorithmes les plus représentatifs ? Il existe plusieurs systèmes de benchmarks se proposant de servir de référence. Un autre benchmark fréquemment utilisé est le « Standard Performance Evaluation Corporation », ou SPEC95, qui couvre plusieurs domaines tels que les vocodeurs, l'asservissement en position des têtes de lectures des disques dur, les modems, voire les applications multimédia sur PC.

Dans la pratique, un autre problème se pose : la qualité de l'implémentation des algorithmes peu varier d'un système de développement à l'autre. Ainsi par exemple, à

qualité égale, un filtre numérique peut demander plus ou moins de ressources processeur en fonction de telle ou telle implémentation. Cet aspect n'est pas pris en compte par les benchmarks.

La mesure des capacités d'un DSP par benchmark reste néanmoins intéressante, car elle tend à mesurer la performance globale du système de traitement numérique (y compris les capacités du programmeur !)

1.9 Les principaux algorithmes traités par les Processeurs DSP

L'Accumulation des Produits (SOP, Sum of Products) est l'élément clé dans la majorité des algorithmes du DSP, pour cela l'architecture des DSP est optimisée pour performer l'opération d'accumulation d'un produit. On peut citer quelques algorithmes typiques comme :

Algorithme du filtre à réponse impulsionnelle finie (RIF) : Un filtre numérique RIF est caractérisé par une réponse uniquement basée sur un nombre fini de valeurs du signal d'entrée. Par conséquent, quel que soit le filtre, sa réponse impulsionnelle sera stable et de durée finie, dépendante du nombre de coefficients du filtre.

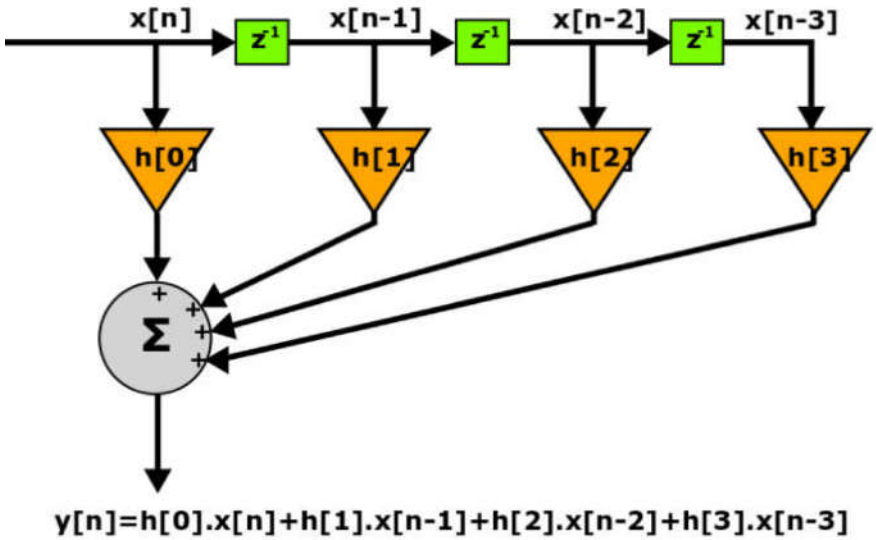


Fig. 1.10 : Filtre à réponse impulsionnelle finie

De façon générale le filtre à réponse impulsionnelle finie est décrit par l'équation sui-

vante :

$$y[n] = \sum_{i=0}^{N-1} h[i].x[n-i] \tag{1.1}$$

Algorithme du filtre à réponse impulsionnelle infinie : est un type de filtre électronique caractérisé par une réponse fondée sur les valeurs du signal d'entrée ainsi que les valeurs antérieures de cette même réponse.

Il est nommé ainsi parce que dans la majorité des cas, la réponse impulsionnelle de ce type de filtre est de durée théoriquement infinie. Il est aussi désigné par l'appellation de filtre récursif. Ce filtre est l'un des deux types de filtre numérique linéaire.

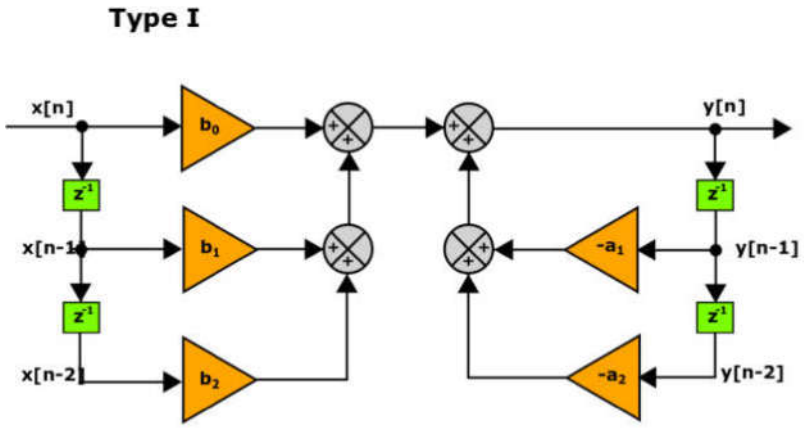


FIG. 1.11 : Filtre à réponse impulsionnelle infinie

$$y[n] = \sum_{i=0}^{N} b[i].x[n-i] - \sum_{i=1}^{N} a[i].y[n-i] \tag{1.2}$$

Algorithme du filtre adaptatif : L'aspect mathématique du filtrage adaptatif est basé sur les travaux du célèbre mathématicien Allemand du 19ème siècle GAUSS. Celui-ci a étudié la théorie de la minimisation des moindres carrés. Cette théorie est largement utilisée aujourd'hui dans tous les domaines de la science et de la technologie. Les DSP ont rendu possible l'utilisation de cette théorie en temps réel sur des signaux échantillonnés jusque environ 100kHz.

L'une des premières utilisations proposée en traitement adaptatif numérique du signal a été la publication de WIDROW et HOFF concernant des circuits de commutations adaptatifs en 1960. Ces travaux ont suscité beaucoup d'intérêt et on été complétés par d'autres publications dans les années 1970.

Le traitement adaptatif du signal a depuis trouvé de nombreuses applications. On peut les classer dans les catégories suivantes :

-détection de signaux (le signal x est-il présent ?),

-estimation de signaux (qu'est-ce que c'est ?),

-estimation d'un paramètre ou d'un état,

-compression de signaux,

-synthèse de signaux,

etc.

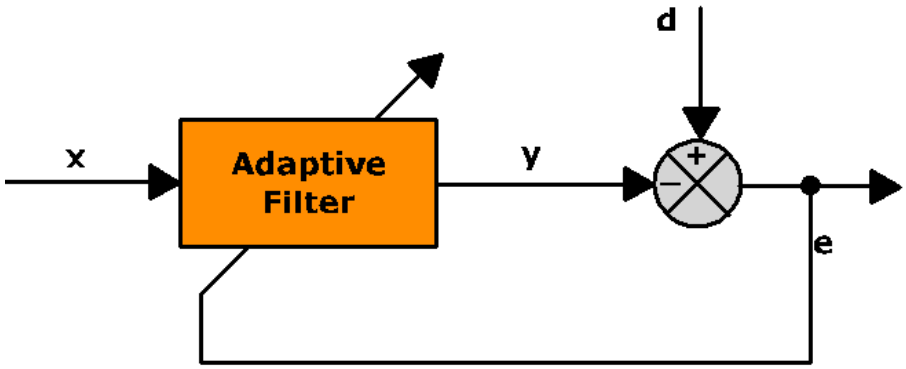


FIG. 1.12 : Filtre adaptatif

$$y[n] = \sum_{k=0}^{N-1} w[k].x[n-k] \tag{1.3}$$

$$e[n] = d[n] - y[n] \tag{1.4}$$

La sortie $y[n]$ du filtre adaptatif est égal au produit de convolution entre le signal d'entré $x[n]$ et le vecteur $w[n]$ représentant les coefficients du filtre adaptatif. Le vecteur $w[n]$ est mis à jour selon une fonction basée sur le signal d'erreur $e[n]$.

Produit de convolution : C'est à l'origine de toute opération de filtrage numérique.

$$x[n] * y[n] = \sum_{k=-\infty}^{+\infty} x[k].y[n-k] \tag{1.5}$$

Algorithme de la Transformée de fourier discrète (TFD) : La transformée de Fourier discrète (sigle anglais DFT Discrete Fourier Transform) est l'un des outils les plus importants du traitement numérique du signal qui calcule le spectre d'un signal de durée finie. Il est très courant d'encoder les informations dans les sinusoïdes qui forment un signal. Cependant, dans certaines applications, la forme d'une forme d'onde dans le domaine temporel ne s'applique pas aux signaux, auquel cas le contenu de fréquence de signal devient très utile autrement que sous forme de signaux numériques. La représentation d'un signal numérique en termes de sa composante fréquentielle dans un domaine fréquentiel est importante. L'algorithme qui transforme les signaux du domaine temporel en composantes du domaine fréquentiel est connu sous le nom de transformée de Fourier discrète ou DFT.

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N})nk} \tag{1.6}$$

Algorithme de la Transformée de fourier rapide (sigle anglais : FFT ou fast Fourier transform) : La Transformée de Fourier Rapide (FFT) est une implémentation de la DFT qui produit presque les mêmes résultats que la DFT, mais elle est incroyablement plus efficace et beaucoup plus rapide, ce qui réduit souvent considérablement le temps de calcul. C'est juste un algorithme de calcul utilisé pour un calcul rapide et efficace de la DFT. Diverses techniques de calcul DFT rapides connues collectivement sous le nom de transformée de Fourier rapide ou FFT.

Les algorithmes de transformée de Fourier rapide se divisent généralement en deux classes : la décimation en temps et la décimation en fréquence. L'algorithme FFT Cooley-Tukey réorganise d'abord les éléments d'entrée dans l'ordre inverse des bits, puis construit la transformée de sortie (décimation dans le temps). L'idée de base est de décomposer une transformée de longueur N en deux transformées de longueur N / 2 en utilisant l'identité

$$\begin{aligned} x[k] &= \sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N})nk} \\ &= \sum_{n=0}^{N/2-1} x[2n] e^{-j(\frac{2\pi k(2n)}{N})} + \sum_{n=0}^{N/2-1} x[2n+1] e^{-j(\frac{2\pi k(2n+1)}{N})} \\ &= \sum_{n=0}^{N/2-1} x[2n] e^{-j(\frac{2\pi k(n)}{N/2})} + e^{-j(\frac{2\pi k}{N})} \sum_{n=0}^{N/2-1} x[2n+1] e^{-j(\frac{2\pi k(n)}{N/2})} \\ &= x_{pair}[k] + e^{-j(\frac{2\pi k}{N})} x_{impair}[k] \end{aligned} \tag{1.7}$$

Algorithme de la transformée en ondelettes discrète (DWT) : Une transformée en ondelettes discrète (DWT) est une transformée qui décompose un signal donné en un certain nombre d'ensembles, où chaque ensemble est une série chronologique de coefficients décrivant l'évolution temporelle du signal dans la bande de fréquences correspondante.

Transformée en cosinus discrète (DCT) Une transformée en cosinus discrète (sigle anglais : DCT Discret cosine Transform) exprime une séquence finie de points de données en termes d'une somme de fonctions cosinus oscillant à différentes fréquences. Le DCT est une technique de transformation largement utilisée dans le traitement du signal et la compression de données. Il est utilisé dans la plupart des médias numériques, y compris les images numériques (comme JPEG), la vidéo numérique (comme MPEG et H.26x), l'audio numérique (comme MP3), la télévision numérique (comme la HDTV), la radio numérique et la parole codage.

Exemple d'un Filtre (RIF) : y[n]=$\sum_{i=0}^{N-1} h[i].x[n-i]$

Pour chaque cellule $h[i].x[n-i]$:

-Recharche de l'instruction
-Recherhe du coeifficient $h[i]$
-Recherche de la donnée $x[n-i]$

-Multiplication $h[i].x[n-i]$
-Accumulation $h[i-1].x[n-i-1]+h[i].x[n-i]$

-Décalage en mémoire x(n-i)→x(n-i-1)

Il y a : 04 accès en mémoire et 02 accès à l'unité de calcule.

Si on utilise un processeur des signaux numériques, on peut :

-Réduire les accès mémoire

-Augmenter les accès mémoire simultanés.

-Réduire le temps passé à faire des calculs (executer une multipliction et accumulation (MAC) en un seul cycle d'instruction).

-Si on veut réaliser un filtrage passe-bas par un filtre FIR de 50 coefficients

- Le calcul de la sortie nécessite 50 MACs

—-Dans un processeur à usage général :

-1 multiplication : 70 cycles horloge

-1 addition : 10 cycles horloge

Au totale > 4000 cycle d'horloge pour 50 coefficient

Question : Quelle sont les architectures utilisées du DSP pour réaliser un traitement le plus rapidement possible (>=1MAC par cycle) ?

Réponse :

1. Architecture matérielle parallèle
2. Mode d'adressage
3. DMA (Direct Memory Acces)
4. Pipeline

1.10 Architecture d'un processeur DSP

Un DSP est avant tout un circuit intégré qui va exécuter des instructions qui auront été définies à l'avance et généralement stockées dans la mémoire programme (EEPROM par exemple).

L'architecture va donc définir les liaisons internes au composant comme la largeur des bus de données, leur nombre et la rapidité de l'unité de calcul.Chaque bus est aussi relié à un registre temporaire permettant de stocker temporairement les instructions, les adresses, les données. D'autres registres sont aussi présents comme le registre d'état (ou de contrôle) dont les données sont soit en lecture seule soit en écriture seule par mesure de sécurité car ils servent à déterminer précisément le déroulement d'une instruction. D'autres encore permettent de configurer différents paramètres pour les ports (série, DMA...), les interfaces avec des mémoires de sorties, les Timers. Ces derniers ont une adresse en mémoire de données.

Le composant possède aussi une unité centrale, son cœur, dans laquelle s'exécutent les instructions. Ce cœur contient dans la plupart des cas une unique unité arithmétique de traitement des données. Chaque unité de calcul fonctionne sur des registres : un servant à placer les donnés à traiter et dont le contenu est géré par le programme, un autre servant à stocker de façon temporaire les résultats intermédiaires des calculs et un dernier servant au résultat final. Ce dernier est « géré par l'unité de calcul » : ce qui y est stocké sera remplacé lors du calcul suivant.

L'architecture est conçue pour permettrent aux instructions d'effectuer les opérations suivantes en un unique cycle d'horloge :

Prendre les données en mémoire vive pour les placer dans un registre temporaire. Ceci est possible via un bus d'adressage permettant de « cibler » la donnée à extraire de la mémoire et un bus de données qui va délivrer le contenu de la donnée voulue.

Faire le calcul voulu (logique ou arithmétique) sur une ou deux données.

Faire un test sur le résultat (dépassement, signe…).

Faire des modifications sur les registres.

Placer le résultat à l'adresse mémoire prévue.

L'architecture d'un DSP est donc caractérisée par un noyau DSP comportant deux unités de traitement des données contenant elles-mêmes des unités de calcul comme l'ALU (Arithmétic and Logic Unit), le MAC (Multiplier and ACcumulator), le décaleur à barillet (Barrel Shifter) et les multiplexeurs d'aiguillage des données. Un noyau DSP possède aussi un séquenceur qui envoie les adresses des instructions situées dans la mémoire programme et un ou plusieurs pointeurs ou générateur d'adresses qui est relié au bus d'adressage.

Après ces généralités, nous pouvons donc voir que finalement, il existe deux grandes familles d'architecture : l'architecture de Von Neumann et l'architecture de Harvard.

1.10.1 Architecture Von Neumann

Le processeur, via le compteur de programme, sait à tout instant où il se trouve dans le programme et sait donc ce qu'il doit exécuter ensuite. Ceci étant, il faut donc aller chercher les données en mémoire programme avec tout ce que cela implique (adressage, lecture…).

L'architecture de Von Neumann a la particularité de ne posséder qu'une seule mémoire dans laquelle sont stockées instructions et données. Il faut donc partager le bus de données venant de cette mémoire. Cette structure est souvent utilisée pour les microcontrôleurs et microprocesseurs car elle est très simple à manier pour le programmeur. Ainsi, c'est celle qui est utilisée dans la famille des microprocesseurs Motorola 68XXX et Intel 80X86.

1.10.2 Architecture Harvard

Ce nom est très bien porté par cette architecture puisqu'elle a été conçue outre-Atlantique vers 1930 dans l'Université de Harvard. On voit tout de suite la différence : les mémoires sont séparées. Cette séparation apporte deux avantages majeurs : les adressages

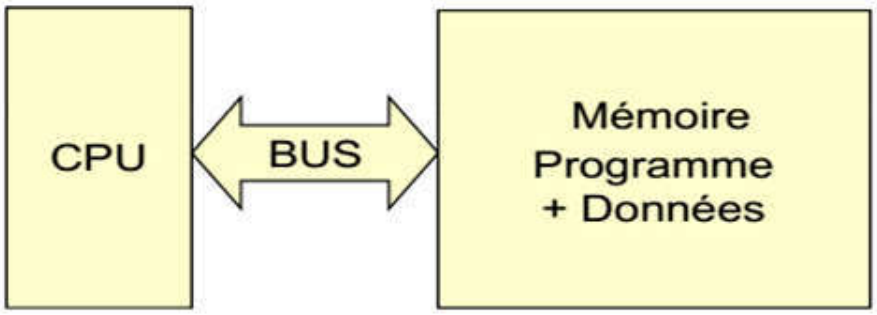


FIG. 1.13 : Architecture Von Neumann

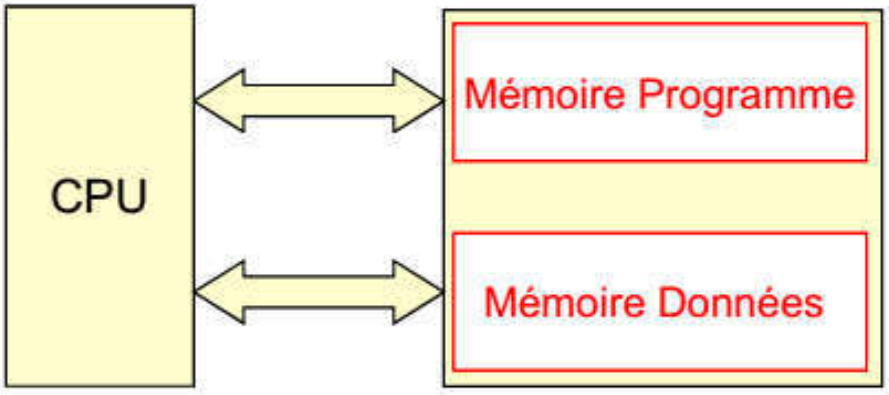


FIG. 1.14 : Architecture Harvard

et le transit des données sont plus simples et il est possible de placer des données dans la mémoire programme pour les algorithmes complexes.

Cette structure est utilisée pour les microprocesseurs spécialisés et pour des applications en temps réel. C'est donc le cas pour les DSP, hors DSP « low cost », sachant que l'architecture de Harvard est plus coûteuse à réaliser.

1.10.3 Architecture Harvard modifiée

Un processeur d'architecture Harvard modifiée ressemble beaucoup à un processeur d'architecture Harvard, mais il assouplit la séparation stricte entre l'instruction et les données tout en laissant le CPU accéder simultanément à deux (ou plus) bus mémoire. La modification la plus courante comprend des instructions séparées et des caches de données soutenus par un espace d'adressage commun. Alors que le processeur s'exécute à partir du cache, il agit comme une pure machine Harvard. Lors de l'accès à la mémoire de sauvegarde, il agit comme une machine von Neumann (où le code peut être déplacé comme des données, ce qui est une technique puissante). Cette modification est répandue dans les processeurs modernes, tels que les DSP et ARM, . On l'appelle parfois vaguement une architecture de Harvard, négligeant le fait qu'elle est en fait "modifiée".

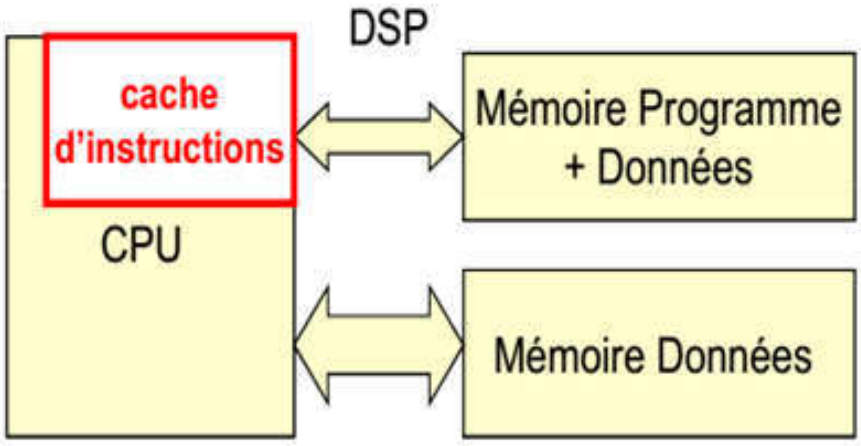


FIG. 1.15 : Architecture Harvard-modifiée

1.10.4 Accès mémoire multi-port

Plusieurs bus de données :

- Accès simultané à plusieurs données
- Nécessite une mémoire multi- accès ou multi- blocs

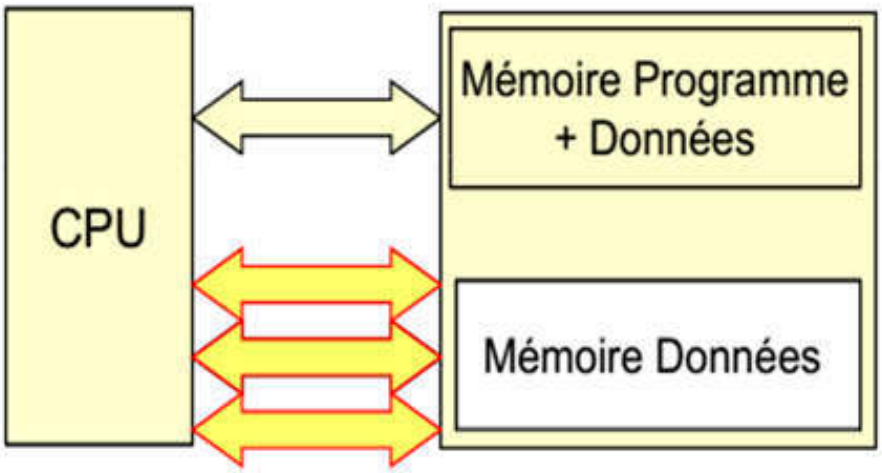


FIG. 1.16 : Accès mémoire multiport

Accès multiport dans TMS320C54xx :

- 1 bus programme (P)
- bus de lecture des données (C et D)
- bus d'écriture des données (E)

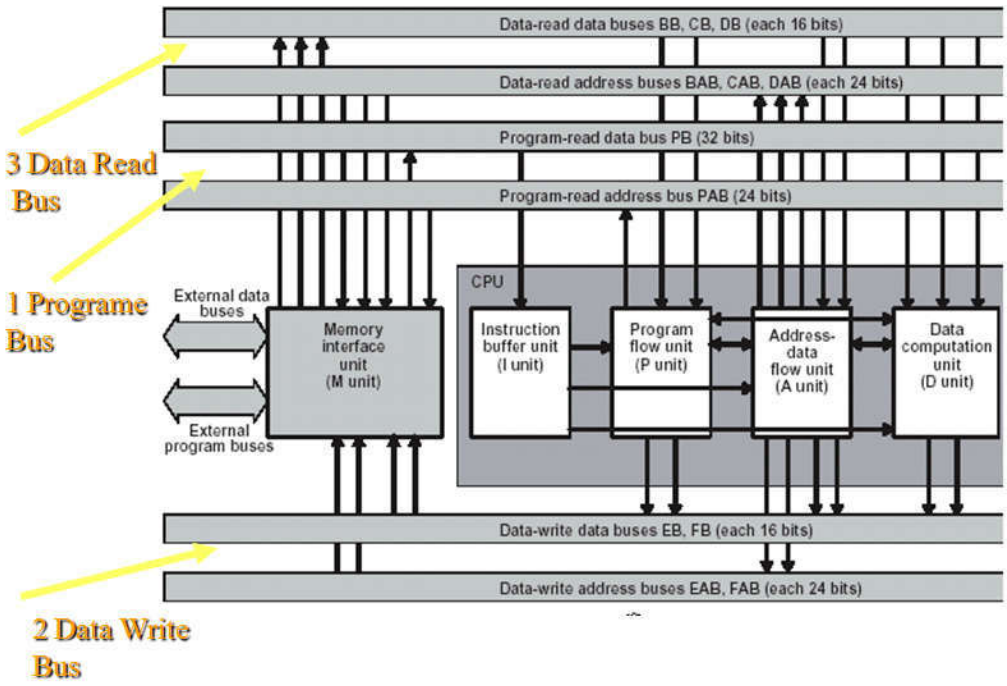


FIG. 1.17 : Accès mémoire multiport dans TMS320C54xx

Exemple accès mémoire multi-port :

Pour l'opération y[n] ← b[n] * x[n] On souhaite récupérer b[n] et x[n]. Chaque bus est capable de transmettre 1 donnée par cycle. La limite est imposée par les capacités de la mémoire :

- Mémoire à Simple Accès (SARAM) => 2 cycles
- Mémoire à Double Accès (DARAM) => 1 cycle

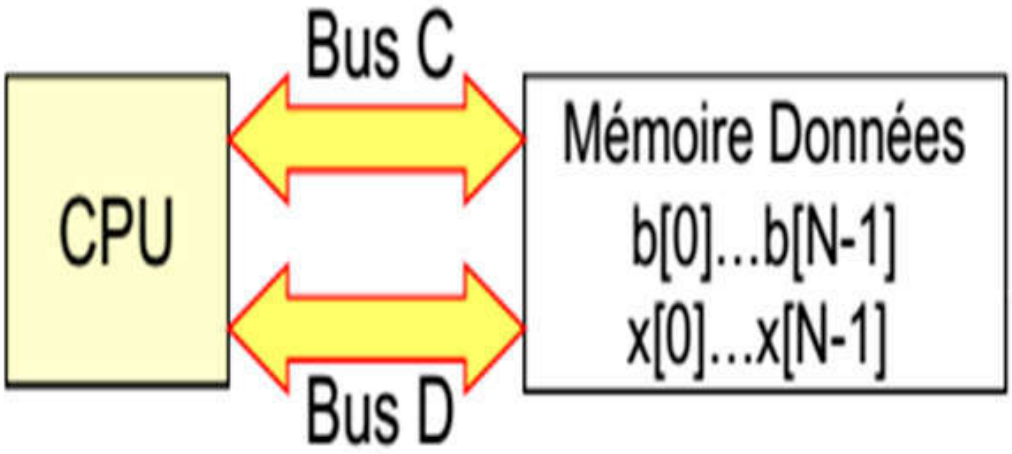


FIG. 1.18 : Exemple Accès mémoire multiport

1.10.5 Multiplexage (Mémoire multi-blocs)

La mémoire est Découpée en blocs indépendants ce qui donne un accès simultané à deux blocs distincts

Exemple – b[n] stockés dans le bloc 1, x[n] stockés dans le bloc 2

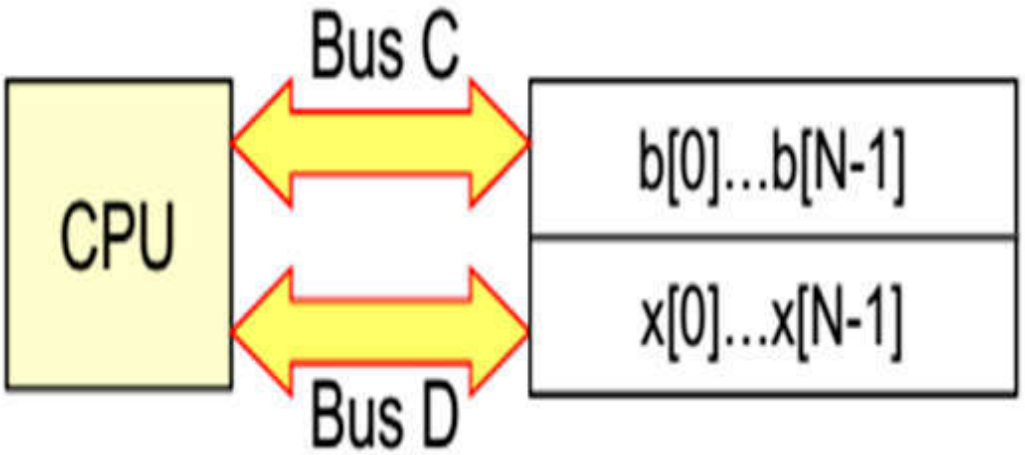


FIG. 1.19 : Exemple Accès mémoire multibloc

1.10.6 Cache d'instructions

La mémoire Cache est une mémoire associative rapide qui contient les dernières instructions exécutées

Utile en cas de boucle :

- Accès aux instructions sans accès à la mémoire programme
- Libère le bus pour des données, ou pour un gain en consommation

Cache pour les données :

- Peu utilisé sur DSP classiques
- De plus en plus utilisé sur les DSP hautes-performances
- Peut poser des problèmes pour la validation du temps réel

1.10.7 Multiplexage des bus dans une architecture Harvard

- Limitation du nombre de broches
- Réduction des coûts
- Diminution des performances lors des accès au bus externe

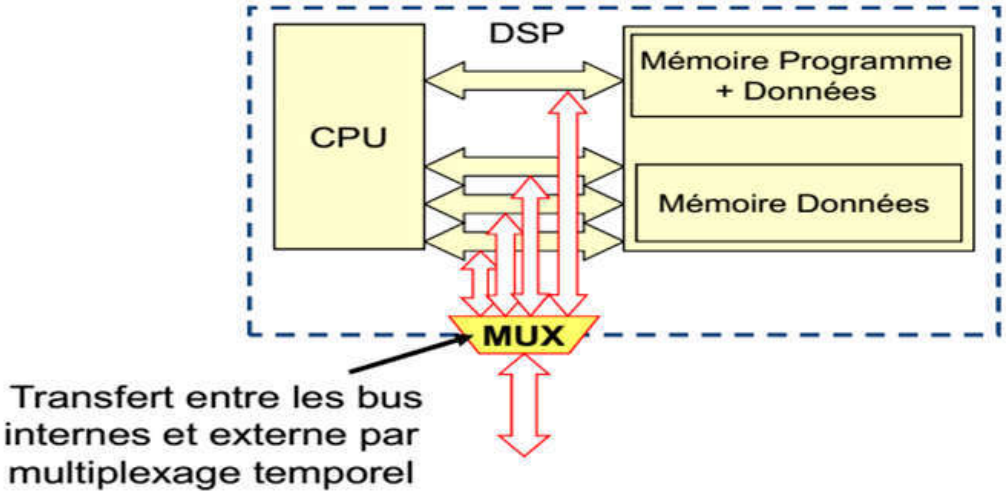


FIG. 1.20 : Multiplexage des bus

1.10.8 Accès Direct à la Mémoire (DMA : Direct Memory Access)

Il existe deux méthodes pour transférer des données d'une partie de la mémoire à une autre, celles-ci utilisent :

(1) CPU.

(2) DMA.

Si un DMA est utilisé, le CPU n'a besoin que de configurer le DMA. Pendant que le transfert a lieu, la CPU est alors libre d'effectuer d'autres opérations.

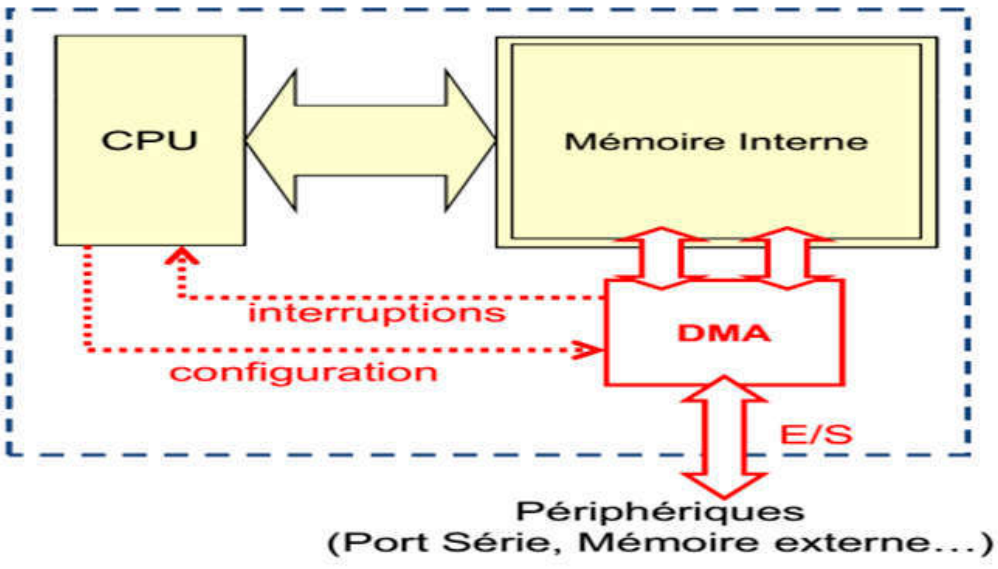


FIG. 1.21 : Accès Direct à la Mémoire (DMA)

1.10.9 Pipelining

Le pipeline est un concept s'inspirant du fonctionnement d'une ligne de montage. Considérons que l'assemblage d'un véhicule se compose de trois étapes (avec éventuellement des étapes intermédiaires) : 1. Installation du moteur. 2. Installation du capot. 3. Pose des pneus.

Un véhicule dans cette ligne de montage ne peut se trouver que dans une seule position à la fois. Une fois le moteur installé, le véhicule Y continue pour une installation du capot, laissant le poste « installation moteur » disponible pour un prochain véhicule X. Le véhicule Z se fait installer ses pneumatiques (roues) tandis que le second (Y) est à l'étape d'installation du capot. Dans le même temps un véhicule X commence l'étape d'installation du moteur.

Si l'installation du moteur, du capot et des roues prennent respectivement 20, 5 et 10 minutes, la réalisation de trois véhicules prendra, s'ils occupent un à un toute la chaîne de production, 105 minutes = $(20 + 5 + 10) \times 3$. Si on place un véhicule dans la chaîne de production dès que l'étage auquel le véhicule doit accéder est libre (principe du pipelining), le temps total pour réaliser les trois véhicules est de 75 minutes.

L'une des approches adoptées pour augmenter l'efficacité des microprocesseurs avancés ainsi que du DSP est le pipelining d'instructions. Un cycle d'instructions commençant par l'extraction d'une instruction et se terminant par l'exécution de l'instruction comprenant le stockage temporel des résultats peut être divisé en un certain nombre de micro-instructions. L'exécution de chacune des micro-instructions est également appelée une phase d'une instruction. Par exemple, un cycle d'instruction nécessitant quatre micro-instructions peut être considéré comme étant en quatre phases comme suit :

1. Phase d'extraction (**Fetch**) dans laquelle l'instruction est extraite de la mémoire du programme.

2. Phase de décodage (**Decode**) au cours de laquelle l'instruction est décodée.

3. Phase de lecture de la mémoire (**Read**) au cours de laquelle l'opérande nécessaire à l'exécution de l'instruction peut être lu à partir de la mémoire de données.

4. Phase d'exécution (**Execute**) au cours de laquelle l'exécution ainsi que le stockage des résultats dans l'un des registres ou dans la mémoire sont effectués.

Chacune des micro-instructions ci-dessus peut être effectuée séparément par quatre unités fonctionnelles. Supposons que chacune des quatre phases ci-dessus prennent le même temps pour se terminer. Dans ce cas dans un microprocesseur classique sans pipelining, chacune des unités fonctionnelles n'est occupée que 25% du temps. En effet, une seule instruction est traitée à la fois sur la CPU.

Exemple de pipeline dans TMS320C54x Le peipining dans cette famille de DSP est diviséé en 6 étages :

- Prefetch (P) : Incrémentaion du PC (Program counter).
- Fetch (F) : Lectrure de l'instruction en mémoire.

- Decode (D) : Décodage de l'instruction.
- Access (A) : Calcul des adresses des opérandes.
- Read (R) : Lecture des opérandes en mémoire.
  Calcul de l'adresse du résultat.
- Execute (X) : Execution du calcul et ecriture en mémoire.

1.10.9.1 Séquentiel vs pipeline :

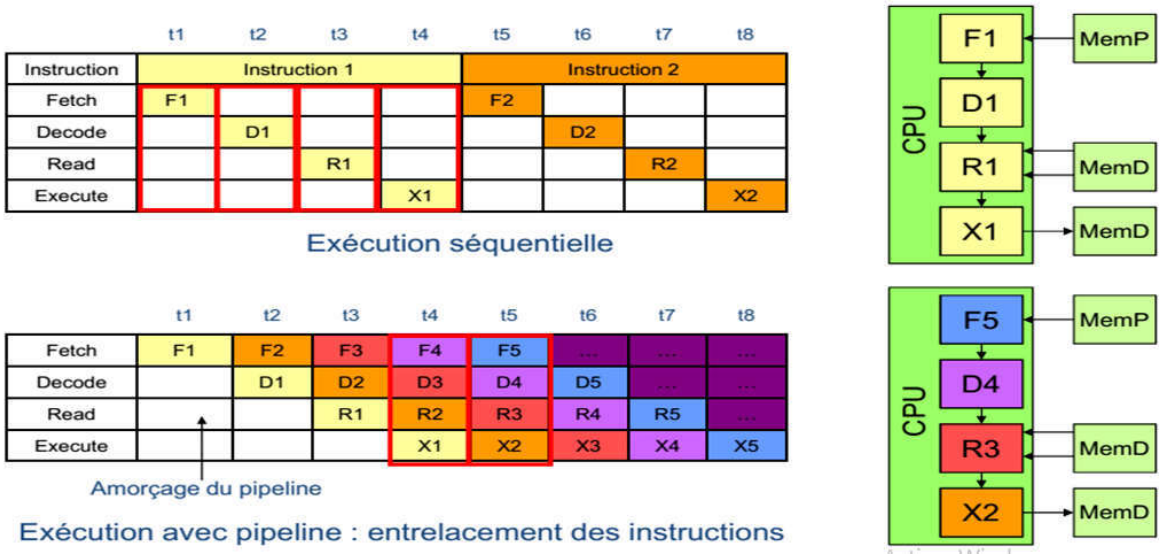


FIG. 1.22 : Séquentiel vs pipeline

1.11 Processeurs DSP et autres approches

a) Microprocesseur d'usage général : (General Purpose Processor, GPP)

Exemples : Intel Pentium, Motorola PowerPC, 68000, Digital Alpha Chip, Sun SPARC et PXA250 (RISC)

Caractéristiques :

- Formats de données varies : sur 32 ou 64 bits
- Adaptes aux langages évolues
- Bonnes performances en calcul numérique : instructions spécifiques (MMX, SSE...)
- Tendance aux évolutions multi-cœurs

- Nombreuses fonctionnalités pas toujours utiles : augmente le coût et la consommation électrique
- Gestion matérielle de la mémoire virtuelle : nécessaire pour l'utilisation de systèmes d'exploitation avances
- Fonctionnalités dynamiques qui compliquent la garantie temps réel : cache de données

b) Microcontrôleur :

Exemples : Motorola 68HC11, Intel 8751, MicroChip PIC16/17, Cypress PsoC.

Caractéristiques :

- Faible coût et faible consommation électrique
- Peu adapte aux signaux numériques : périphériques à faible débit, données sur 8 ou 16 bits
- Adaptes aux tâches de contrôle : Embarqué, temps réel, calculs logiques
- Mémoire limitée
- Utile pour calculs peu complexes

c) Processeur DSP :

Exemples : Texas Instrument TMS320Cxxxx, Freescale 56000, 96000 et Analog Devices ADSPxxxx, SHARC, Blackfin

Caractéristiques :

- Architecture spécialisée pour les traitements numériques : coût, encombrement et consommation au plus juste
- Processeur : programme en mémoire, exécution séquentielle, jeu d'instruction spécialisé
- Calculs numériques : addition, multiplication, unités de calcul spécialisées (filtrage, FFT…)
- Entrée-sortie de données : grand débit, périphériques intégrés

Un processeur d'usage général est préférable pour :

- Disposer d'une grande quantité de mémoire

- Développer sous un système d'exploitation avancé
- Nécessite de mixer calcul numérique et autres tâches

Un processeur DSP est préférable pour :

- Minimiser la taille
- Minimiser la consommation
- Traitement temps-réel à grand débit

d) GPU (Graphics Processing Unit)

Exemples : NVIDIA et ATI

Caractéristiques : Caractéristiques :

- Nombreuses unités de calcul indépendantes en parallèle : streaming Processor (SP), texture filtering Unit (TF)
- Programmation spécifique (CUDA, OpenCL…)
- Très efficace si GPU disponible (calcul scientifique, traitement d'image)
- Risque si contraintes de pérennité et de certification (aéronautique)

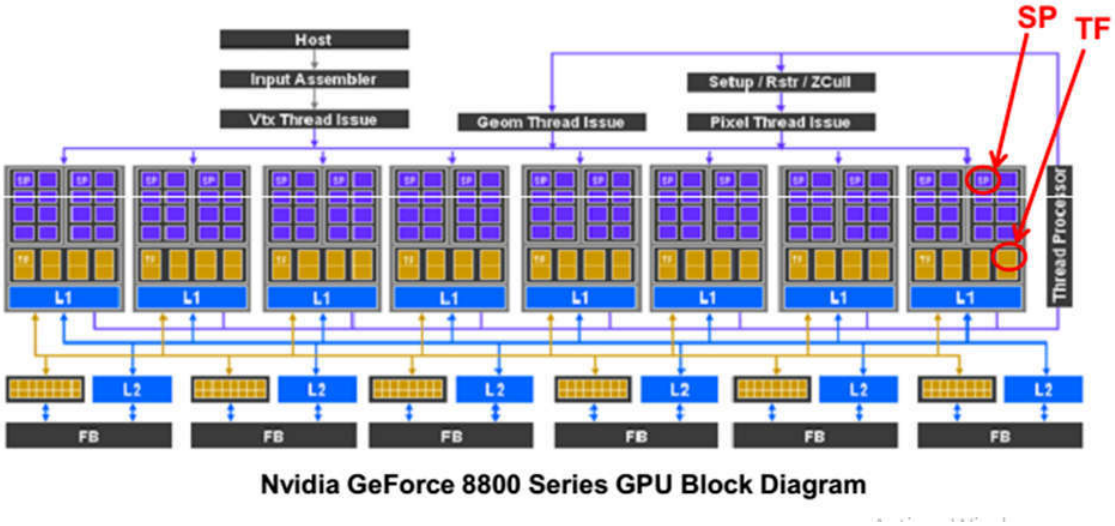


FIG. 1.23 : Exemple de GPU

e) FPGA (Field-Programmable Gate Array)

Exemples : Altera et Xilinx

Caractéristiques :

- Logique reconfigurable : VHDL compilé puis chargé sur la puce, facile à reconfigurer (mais pas en cours de fonctionnement)
- Produits récents intègrent des blocs « DSP » : unités arithmétiques simples
- Adapte si calculs fortement parallélisables

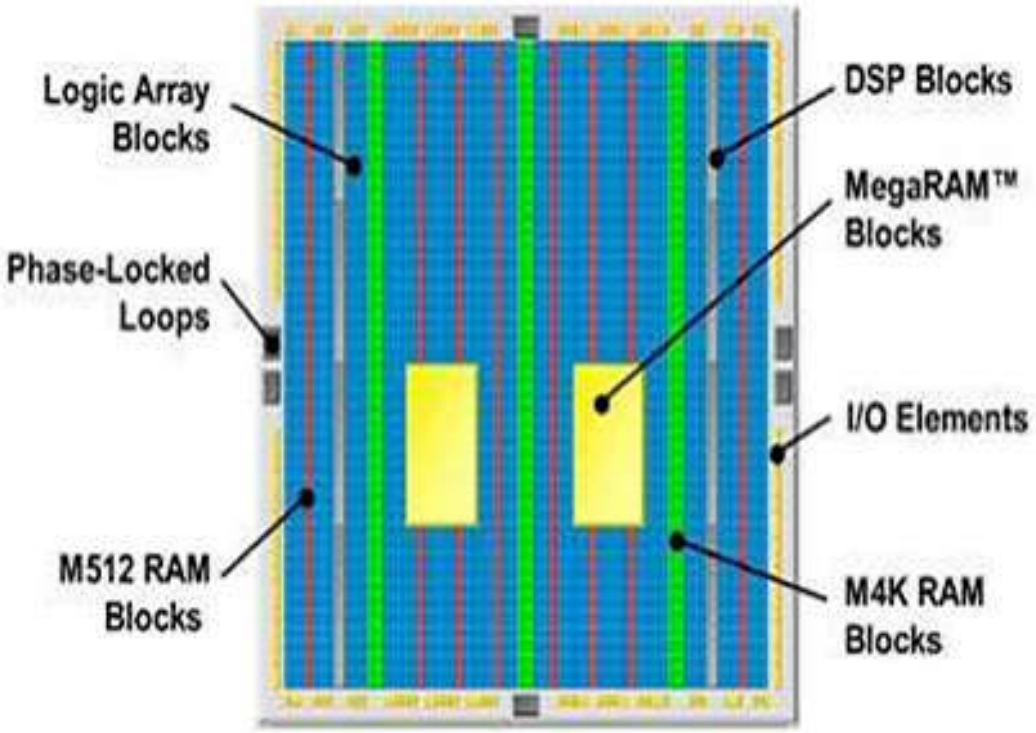


Fig. 1.24 : Exemple d'architecture : Altera Stratix

f) ASSP (Application Specific Standard Product) Puce spécialisée pour une application spécifique : peut-être programmable ou pas

Exemple : Micronas MDE9500 « Single-chip Digital TV mixed-signal decoder » Fonctionnalités nécessaires pour un poste de télé numérique : Décodage analogique, démultiplexage, décryptage, décodage MPEG2, interface vers sortie vidéo ou numérique, télétexte...

g) ASIC (Application-Specific Integrated Circuit)

- Puce conçue entièrement pour l'application visée : parfaitement adapté à l'application
- Meilleures performances : calcul et vitesse, consommation
- Passage par un fondeur : coûts de développement élevés, peu flexible
- Conception prend du temps : développement d'un circuit complet en VHDL
- Cœurs licenciables : intégration d'IP (cœurs DSP, modules d'E/S), permet la réutilisation de conceptions éprouvées, personnalisation possible de l'architecture

h) Conclusion Le processeur ou le matériel à utiliser dépend énormément de l'application considérée, en termes de performances, de contraintes de temps d'exécution, du coût, ... Ainsi, un compromis doit être fait pour un choix judicieux du processeur.

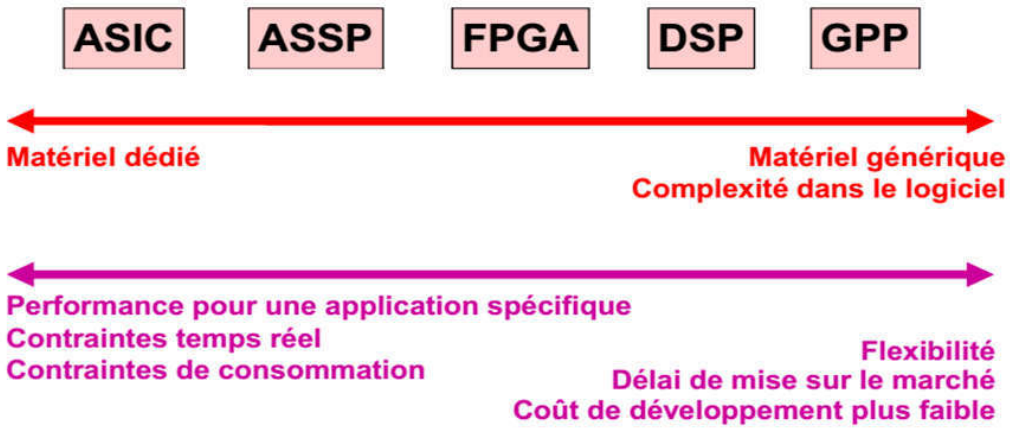


FIG. 1.25 : Compromis pour l'implantation d'algorithmes de traitement du signal

1.12 Historique et évolutions récentes

Le TMS32010 est le premier processeur introduit sur le marché en avril 1983. Le processeur est disponible selon différents modèles, avec arithmétique à virgule fixe ou à virgule flottante. Le processeur TMS32010, ainsi que ses variantes, est un CPU avec une architecture de Harvard modifiée ; avec une mémoire programme et une autre pour les données. Il est caractérisé par une très rapide opération MAC (multiply-and-accumulate). Un certain nombre de variantes de produits sont mis au point en se basant sur l''architecture TMS320. Les codes des produits utilisés par Texas Instruments sont donnés par « TMS320Cabcd » où a est la série principale, b la production et cd est un nombre personnalisé pour une sous-variante secondaire. Dans cette partie, nous donnerons quelques évolutions des DSP. On s'intéresse au DSP de Texas instrument TMS320. Nous parcourrons l'histoire des DSP de la première série à la série la plus récente. Les **TMS320** est une famille de processeurs de signal numérique (DSP) fabriqués par Texas Instruments.

Processeur hors-série :

- TMS320C1x, première génération « 16-bit fixed-point » DSPs. Tous les processeurs de cette série sont compatibles en termes de code avec le TMS32010.
  - o TMS32010, le tout premier processeur dans la première série introduit en 1983, et utilisant de la mémoire externe.
  - o TMS320M10, le même processeur mais avec une mémoire ROM interne de 3KB
  - o TMS320C10, TMS320C15 etc.

- TMS320C3x, « floating point »
  - o TMS320VC33

- TMS320C4x, « floating point »

- TMS320C8x, multiprocesseur chip
  - o TMS320C80 MVP (multimedia video processor) possède un "processeur maître" à 32 bit floating-point et 4 "processeur parallèles" à 32-bit fixed-point.

La série C2000 :
Cette série se base sur des microcontrôleurs 32-bits les performances des périphériques intégrés pour la conception des applications de contrôle en temps-réel. On trouve 5 sous-catégories : le nouveau C28x+ ARM Cortex M3, C28x Delfino Floating-point, C28x Piccolo, C28x Fixed-Point, et C240x.
La série C2000 se distingue par sa haute performance de l'ensemble des périphériques intégrés, comme les modules : PWM, ADC, encodeur en quadrature, et de capture. La série supporte en plus le I2C, SPI, série (SCI), CAN, chien de garde, McBSP, interface mémoire externe I et GPIO. Grâce à des fonctionnalités telles que la synchronisation signal PWM avec l'unité ADC, la série C2000 est bien adapté à de nombreuses applications de contrôle en temps réel. La famille C2000 est utilisée pour des applications comme le contrôle d'un moteur d'entraînement, l'automatisation industrielle, les énergies renouvelables solaire et autres, ...

La série C5000 :
Les TMS320C5X traitent des données au format virgule fixe. Leur conception les rend encore plus puissant, d'une part par leur mémoire interne et d'autre part par leurs jeus d'instruction toujours plus puissant. Leurs bus DMA sont aussi une raison à leur puissance. Le TMS320C54x est un 16-bit fixed point DSP, avec pipeline à 6 étages, opérations arithmétiques "parallel load/store", MAC opération et autres améliorations. C'était un choix populaire pour la 2G des téléphones portables ; le C54x était utilisait par Nokia et Ericsson pour la fabrication de ces téléphones vers la fin des années 90. À l'époque, le processeur ARM7, à usage général, était adopté afin d'améliorer l'interface utilisateur des téléphones portables et de contrôle, et décharger le DSP de cette fonction. Cela a finalement abouti à la création d'un dual core « ARM7 + C54x DSP », qui est devenue ensuite la ligne de produits OMAP. La génération TMS320C55x « fixed point », exécute le code C54x, mais ajoute plus

de parallélisme interne (autre ALU, double MAC, plus de la mémoire) et registres, tout en supportant de plus en plus des opérations en faible puissance. Aujourd'hui, la plupart des DSP C55x sont vendus comme des puces discrètes. Les puces OMAP1 combinent un ARM9 (ARMv5TEJ) avec une série DSP C55x. Les puces OMAP2420 combinent un ARM11 (ARMv6) avec une série DSP C55x.

La série C6000 :
La série TMS320 C6000, or « TMS320C6x : VLIW-based DSPs ». TMS320C62x fixed point /2000 MIPS/1.9 Watts.
TMS320C64x fixed point - code compatible avec TMS320C62x (connu aussi comme C64, C64x, et C64x+)
TMS320C67x floating point - code compatible avec TMS320C62x
TMS320C674x fixed et floating point - code compatible avec C64x and C67x
Autres parties de la série C6000 comporte :
La puce DaVinci comporte un ou les deux d'un ARM9 et un C64x+ ou un C674x DSP.
La puce OMAP-L13x comporte un ARM9 (ARMv5TEJ) et un C674x fixed et floating point DSP
La puce OMAP243x comporte un ARM11 (ARMv6) avec un DSP de la série C64x.
Les puces OMAP3 et OMAP4 comportent un ARM Cortex-A8 ou A9 (ARMv7) et fréquemment un DSP C64x+ à virgule fixe.

Les séries DaVinci :
Les series DaVinci ont débuté avec un 'system-on-a-chip' en utilisant les séries de DSP C6000 (C64x+), processeur d'application ARM9, et les périphériques multimédia numériques. Il existe des variantes sans ARMs, et sans DSPs. Leur commercialisation se base sur leurs capacités de traitement vidéo. les premières variantes supportent les standards NTSC et PAL, tandis que les plus récentes supportent la HDTV.

Les variantes OMAP :
Ces variantes incluent aussi un processeur ARM dans la même puce. Il existe d'autres types OMAP possédant des processeurs secondaires, qui ne sont pas nécessairement des DSPs.
Les variantes DA (cible "Digital audio")
Le DSP DA25x est un processeur ARM et un core C55x. il possède des périphériques sur la puce, comme le contrôleur esclave USB. Ces types de DSP sont exclusivement utilisés dans les gammes de lecteurs audio numériques Creative ZEN et Dell Digital Jukebox MP3 players, comme CPU principal et processeur de signal pour tous les traitements concernant le flux de données MP3.
Les puces TMS320DA7xx Aureus sont construites autour des DSP C67x+.

Le DA83x est un autre Aureus, identique au OMAP-L137 ... qui est basé sur les technologies « non-video DaVinci ». Il comporte un DSP C67x à virgule flottante et

un core ARM9 relativement rapide. Le TMS320DA828 est semlable aussi mais avec moins d'interfaces I/O.
En général les évolutions des DSP sont résumées comme suit :

- 1e génération 1979-1985 : architecture Harvard, multiplieur câblé.
- 2e génération 1985-1988 : parallélisme, bus multiples, mémoire sur la puce.
- 3e génération 1988-1992 : virgule flottante.
- 4e génération 1992-1997 : image et vidéo, processeurs faible consommation.
- 5e génération 1997-2008 : SIMD, VLIW.
- Plus récemment : SoC DSP (Homogènes (Multi-DSP), Hybrides (RISC+DSP)), Co-design hardware-software.

Chapitre 2

Arithmétique à virgule fixe et à virgule flottante

2.1 Introduction

Un signal physique est en général un signal analogique représentant une variation continue de tension en fonction du temps. Pour le traiter avec un DSP il doit être converti en un signal numérique, consistant en une variation discrète de tension en fonction du temps. Le processus de conversion analogique-numérique consiste à le convertir en une suite de nombres dont chacun représente l'amplitude instantanée du signal original à un instant significatif donné, puis à enregistrer ces nombres après un codage. L'échantillonnage et la quantification représentent les principales étapes

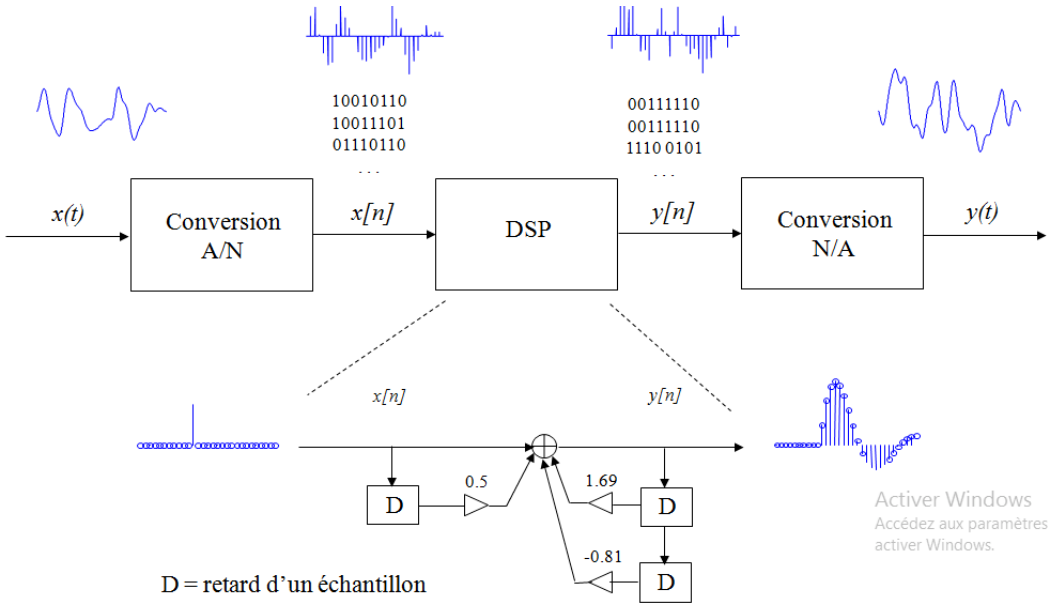


FIG. 2.1 : chaîne conversion analogique-numérique et numérique-analogique

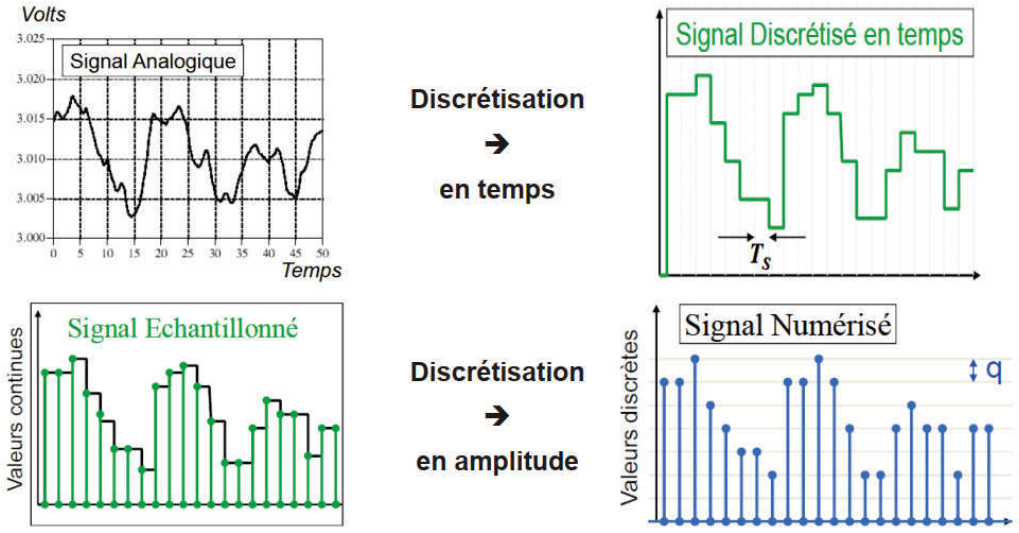


FIG. 2.2 : Processus de discrétisation en temps et en amplitude

de la conversion analogique/numérique. L'étude de ces deux étapes est nécessaire

pour comprendre les méthodes de minimisation de perte d'informations à l'issue d'une conversion analogique/numérique.

2.2 Echantillonnage d'un signal analogique

Pour échantillonner un signal analogique continu $x(t) = Acos(wt + \varphi)$ et le transformer en une suite discrète d'échantillons, on prélève périodiquement à des intervalles de temps $t = T_e$, la valeur du signal à l'aide d'un échantillonneur. L'échantillonneur est un commutateur analogique K qui se ferme durant un temps t_0 très bref toutes les T_e secondes. t_0 est le temps d'ouverture de la porte d'échantillonnage, T_e est la période d'échantillonnage et $f_e = 1/T_e$ est la fréquence d'échantillonnage. Le signal discret généré est donné par :

$$x[n] = Acos(wT_s + \varphi) \tag{2.1}$$

avec n=0,1,2,..., et $\theta = wT_e = \dfrac{2\pi f}{f_e}$

- le signal échantillonné est constitué par un train d'impulsions espacées de T_s, de largeur t_0 et d'amplitude $x(nT_e)$
- en pratique le temps d'ouverture t_0 est toujours petit devant la période d'échantillonnage T_e

Exemples de signaux échantillonnés :

Echantillonnage d'un signal sinusoïdal : Signal x(t) en bleu, signal échantillonné en rouge, fréquence du signal F = 1 kHz, fréquence d'échantillonnage f_e= 10 kHz ce qui produira 10 échantillons par période.

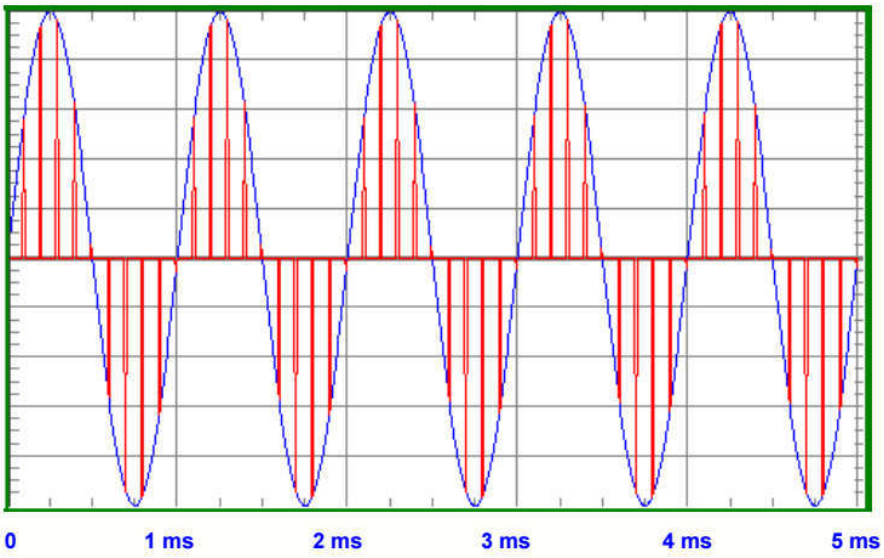


FIG. 2.3 : Echantillonnage d'un signal sinusoïdal

Echantillonnage d'un signal quelconque : signal x(t) en bleu, signal échantillonné en rouge et la fréquence d'échantillonnage f_e= 10 kHz.

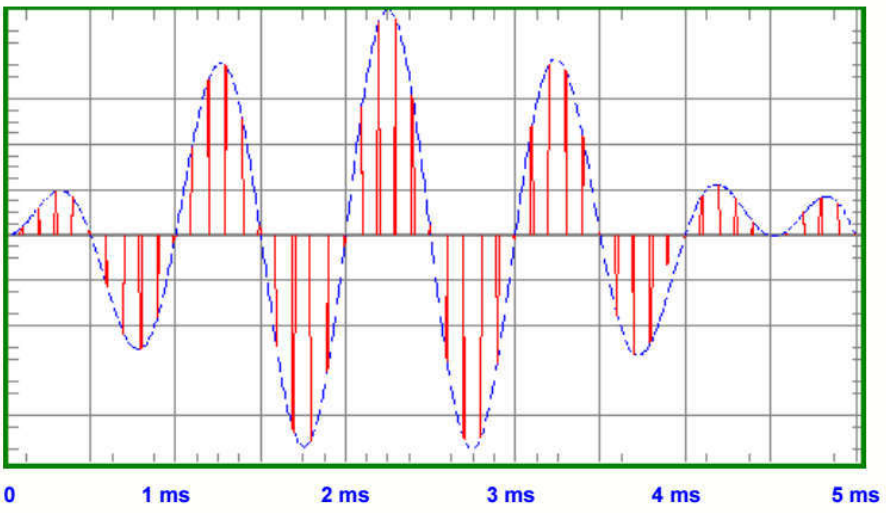


FIG. 2.4 : Echantillonnage d'un signal quelconque

Choix de la fréquence d'échantillonnage

L'opération d'échantillonnage ne doit pas amener une perte d'informations, c'est-à-dire que l'opération d'échantillonnage doit être réversible :

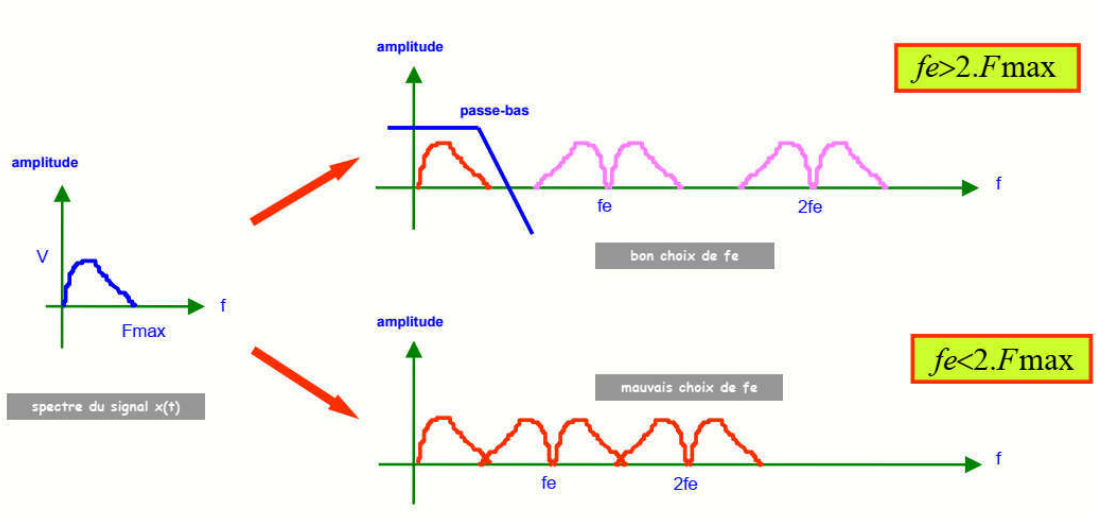


FIG. 2.5 : Règle de Shannon

Règle de Shannon : f_e doit être au moins égale au double de la fréquence maximale contenue dans le signal. Si la fréquence d'échantillonnage est supérieure à 2.Fmax, on peut revenir au signal analogique continu par simple filtrage passe-bas.

Exemples de fréquences d'échantillonnage
En téléphonie, on se contente d'une qualité moyenne, avec des fréquences vocales limitées à la bande de 300 Hz à Fmax = 3400 Hz.
Dans le cadre du réseau téléphonique numérique, la fréquence d'échantillonnage standard est de f_e = 8000 Hz.

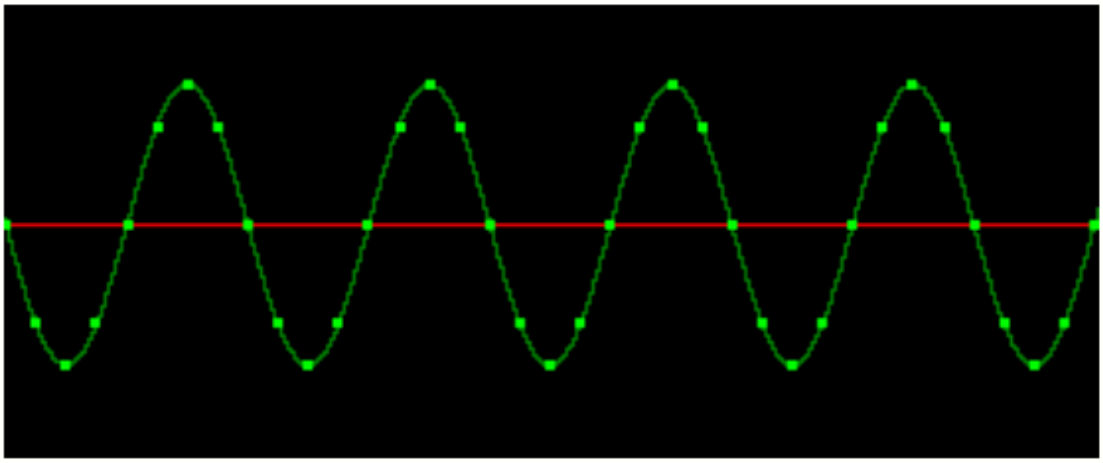


FIG. 2.6 : Signal de 1 KHz échantillonné à 8 KHz

- la musique nécessite une bande allant de 20 Hz jusqu'à Fmax = 20 kHz pour une qualité optimale
- dans le cas du disque CD (Compact Disc) la fréquence d'échantillonnage standard est de f_e = 44,1 kHz
- pour les magnétophones numériques DAT, la fréquence d'échantillonnage est de f_e = 48 kHz
- pour le matériel audio professionnel, la fréquence standard est de f_e = 96 kHz

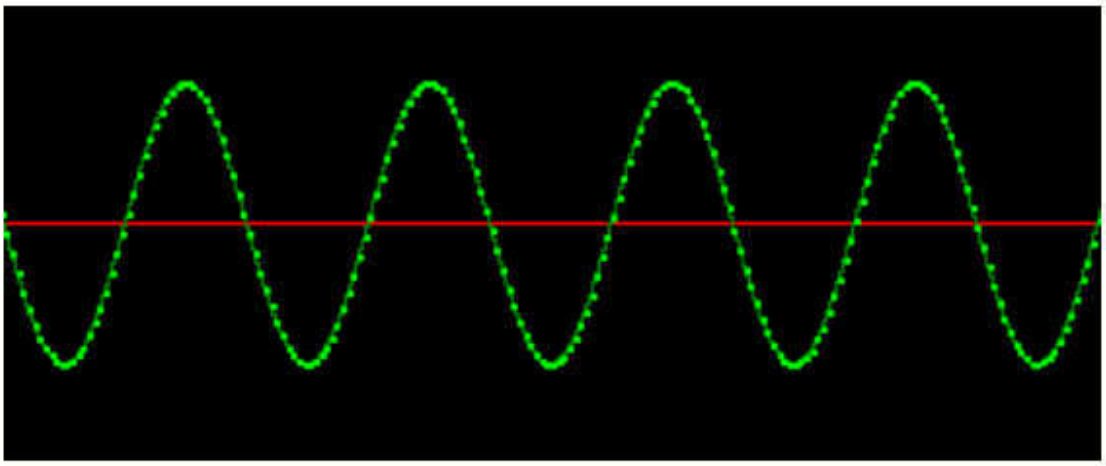


FIG. 2.7 : Signal de 1 KHz échantillonné à 44.1 KHz

- un signal vidéo de télévision a une spectre qui s'étend pratiquement du continu jusqu'à Fmax = 5 ou 6 MHz selon la qualité de l'image

- dans le cas de la vidéo numérique, la fréquence d'échantillonnage standard est de f_e = 13.5 kHz

Le phénomène de repliement de spectre
Le bon choix de f_e nécessite de bien connaître la valeur de la fréquence maximale Fmax contenue dans le signal à échantillonner.

Exemple : un microphone fournit un signal électrique composé :

- de la musique dans la bande 20 Hz-20 kHz

- de bruit électrique à densité spectrale constante dans la bande 0-40 kHz

- d'un signal parasite à 35 kHz

L'ingénieur du son qui choisit une fréquence d'échantillonnage f_e = 44,1 kHz respecte la règle de Shannon pour la musique, mais pas pour le bruit, ni pour le signal parasite.

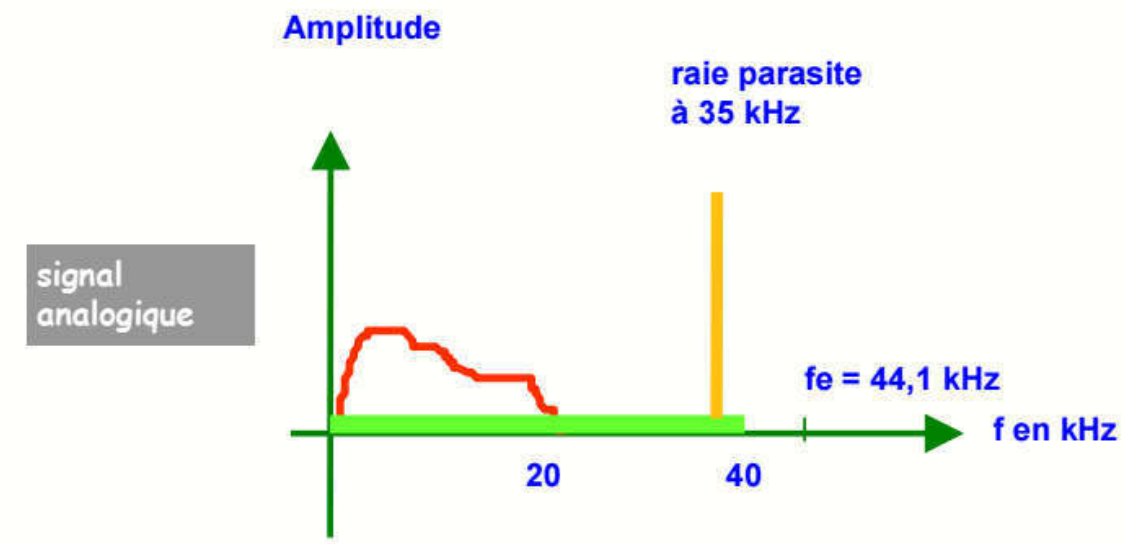


FIG. 2.8 : spectre du signal fourni par le microphone

On constate l'apparition dans la bande audio par repliement de spectre :

- d'un signal parasite à 44,1 -35 = 9,1 kHz qui se trouve dans la bande audio

- d'une augmentation du bruit provenant du bruit au-delà de 20 kHz replié vers les BF.

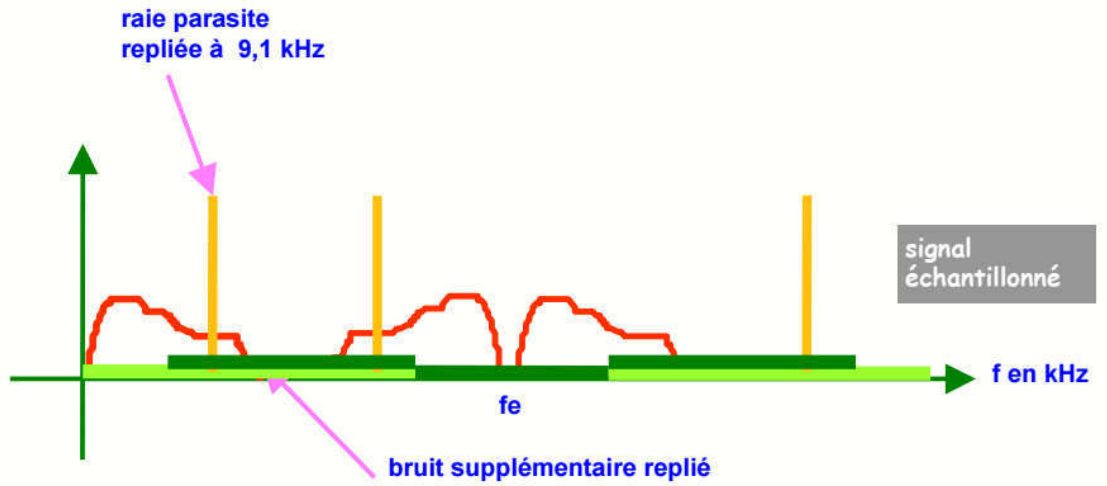


FIG. 2.9 : Repliement du spectre du signal fourni par le microphone

Remarques
- si la musique ainsi numérisée est gravée sur un CD audio, le 9,1 kHz ainsi que le bruit supplémentaire seront audibles.

- si on diminue fs sans filtre anti-repliement, la qualité se dégrade de plus en plus à cause du repliement de spectre.

Pour éviter les problèmes de repliement de spectre, il faut s'assurer que le spectre du signal analogique x(t) est vraiment limité à la valeur Fmax, toujours un peu inférieure à $f_e/2$. Pour cela, on place avant l'échantillonneur un filtre à coupure raide qui atténue très fortement tous les signaux parasites au-delà de la fréquence limite Fmax : c'est le filtre anti-repliement.

Ce filtre anti-repliement doit simplement :

- laisser passer le signal sans le déformer, donc avoir une courbe de réponse plate entre 0 et Fmax.

- atténuer au maximum toutes les composantes au-delà de $f_e/2$, qui seraient sinon repliées et apparaîtraient dans le signal échantillonné.

Remarque : le filtre anti-repliement ne supprime pas le phénomène, mais atténue le signal replié au point de le rendre négligeable.

Pour permettre la conversion analogique-numérique entre deux instants d'échantillonnage, il faut maintenir la valeur du signal x*(t) à l'entrée du convertisseur jusqu'à l'arrivée de l'échantillon suivant. Dans la pratique, les opérations d'échantillonnage et de maintien sont effectuées par la même fonction : l'échantillonneur-bloqueur dans lequel l'échantillon est conservé en mémoire par un condensateur C.

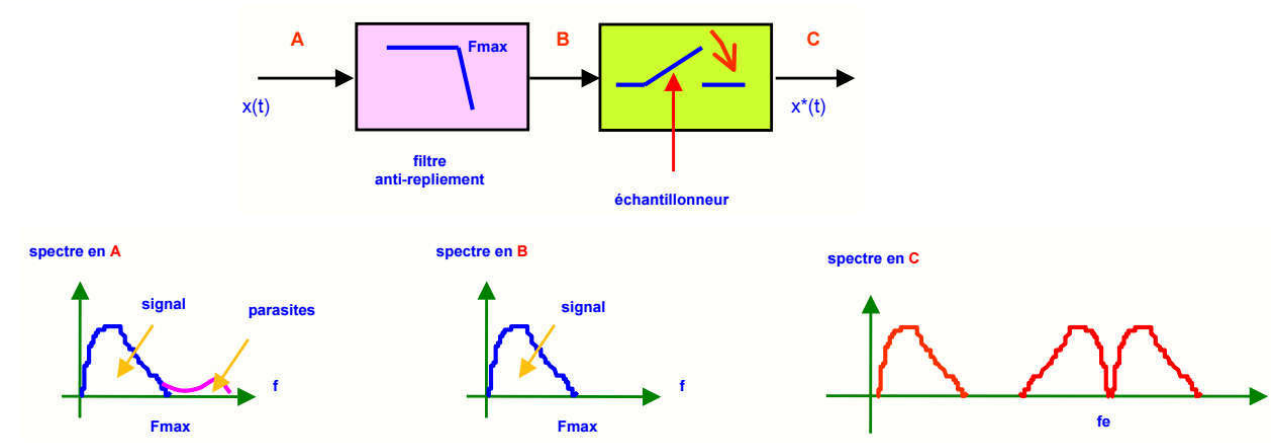


FIG. 2.10 : Amélioration du signal de sortie par le filtre anti-repliement de spectre

2.3 La quantification

On se limite ici à la quantification scalaire, c'est-à-dire à la quantification d'un échantillon isolé. Un convertisseur analogique-numérique (CAN) possède un nombre de bits fini (résolution). Comme résultats, les valeurs des amplitudes continues sont approximées par des niveaux d'amplitudes discrètes. Le processus de conversion d'amplitude continue en numérique se nomme la quantification. Cette approximation produit une erreur appelée 'erreur de quantification'. Dans la figure 2.11, on représente la caractéristique d'un CAN à 3 bits. L'intervalle de quantification dépend du nombre de bits utilisé. Pour éviter la saturation, l'amplitude su signal d'entrée doit rester dans la gamme $[V_{ref-}, V_{ref+}]$. La tension pleine échelle (Full Scale FS) est donnée par :

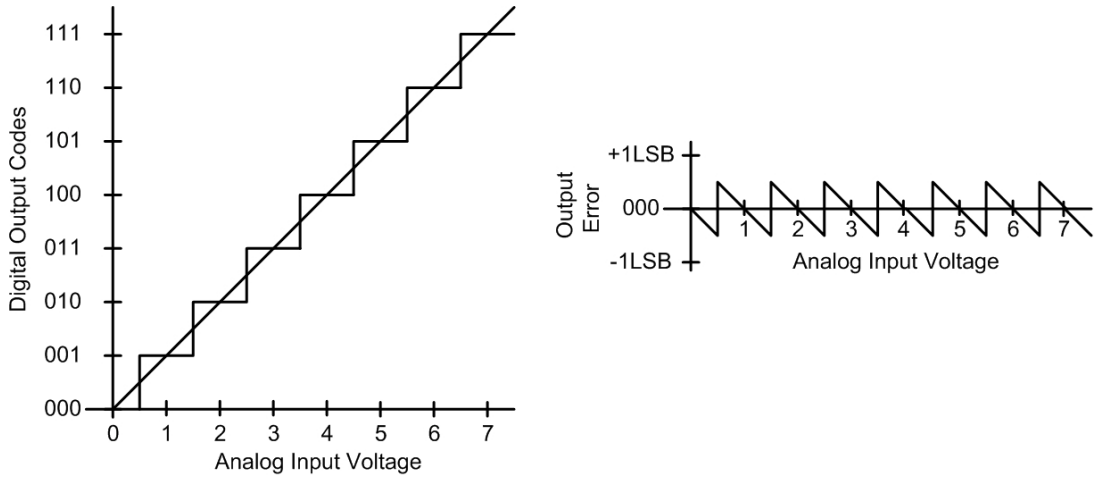


FIG. 2.11 : caractéristique d'un CAN à 3 bits : fonction de transfert,et bruit de quantification

$$V_{FS} = V_{ref} = V_{ref+} - V_{ref-} \tag{2.2}$$

V_{ref} \ N	8	10	12	14	16	20
5 V	19.5V	4.9 mV	1.2 mV	305 μV	76 μV	4.8 μV
3 V	11.7 mV	2.9 mV	732 μV	183 μV	45.8 μV	2.8 μV
1.8 V	7.0 mV	1.7 mV	439 μV	110 μV	27.5 μV	1.7 μV

Tab. 2.1 : LSB d'un CAN

Et le bit le moins significatif (LSB) est donné par :

$$LSB = \Delta = \frac{V_{ref}}{2^N} \tag{2.3}$$

Où N est le nombre de bits du CAN. Le tableau ci-dessous illustre les valeurs du LSB correspondantes à différents nombre de bits

Quantifier une valeur X réelle appartenant à un intervalle $[-X_{max}\ ,\ X_{max}]$, consiste à remplacer cette valeur X par la valeur $Q(X) = X_n$ la plus proche de X choisie dans un ensemble fini (ou dénombrable) de N valeurs réelles notées xn, (avec n entre 0 et N−1). La valeur quantifiée de X : $Q(X)$ est donnée par :

$\forall\ X \in [X_n, X_{n+1}]$
Si $\mid X - X_n \mid < \mid X - X_{n+1} \mid$ $\quad Q(X) = X_n$
Si $\mid X - X_n \mid > \mid X - X_{n+1} \mid$ $\quad Q(X) = X_{n+1}$
Si $(X_n + X_{n+1})/2$ $\quad Q(X) = X_n$ ou X_{n+1} selon les systèmes.

On distingue plusieurs types de quantification scalaire, en particulier

- la quantification uniforme,

- la quantification non uniforme, comme la conversion de type logarithmique.

2.3.1 Quantification Uniforme

Si tous les intervalles $[X_n, X_{n+1}[$ ont une même longueur, la quantification est dite uniforme et la constante q définie par $q = |X_n - X_{n+1}|$ est appelée pas de quantification ou quantum. Le pas de quantification q peut s'exprimer en fonction des valeurs extrêmes $\pm X_{max}$ par :

$$q = \frac{2X_{max}}{2^N} \tag{2.4}$$

où N est le nombre de valeurs de quantification.

2.3.1.1 Caractéristique de quantification

C'est la courbe donnant $Q(X)$ en fonction de X. Dans le cas d'une quantification uniforme, cette courbe a l'allure suivante :

La quantification uniforme est, de ce fait, parfois appelée quantification linéaire.

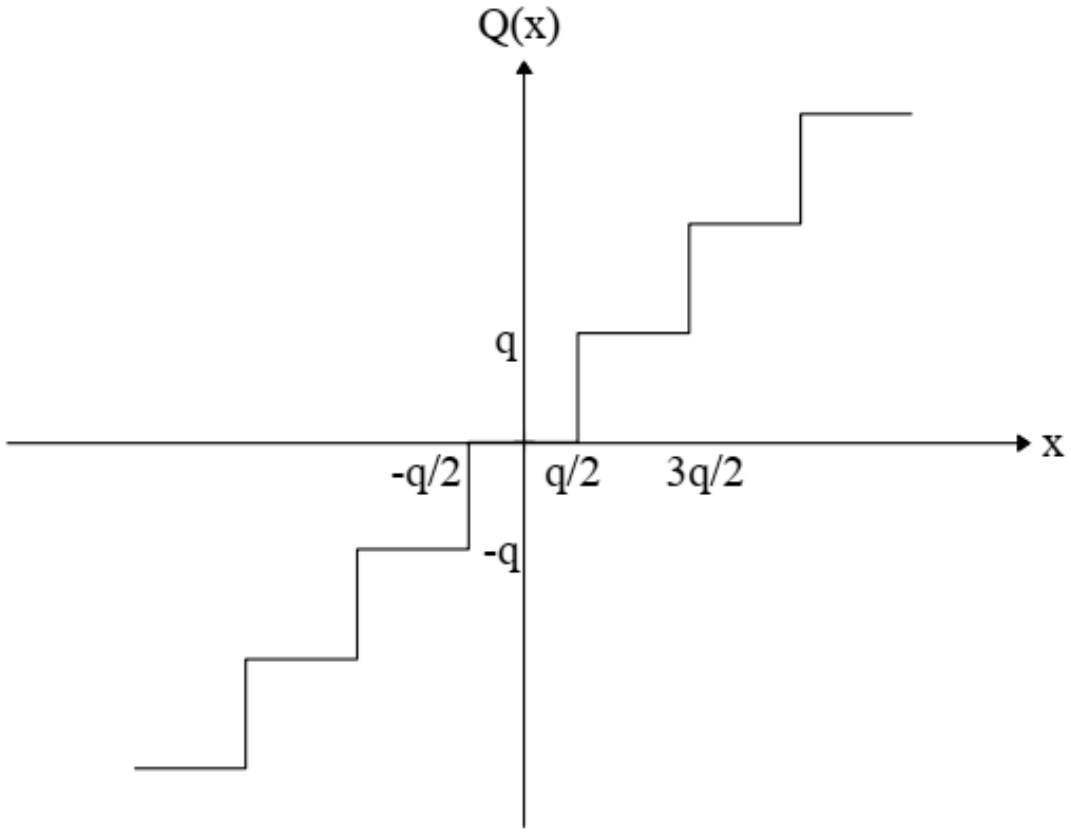


FIG. 2.12 : Quantification uniforme

Le signal échantillonné et bloqué en marches d'escalier variant entre 0 et E (pleine échelle) ou $-E/2$ à $+E/2$ peut maintenant être converti en une suite de valeurs binaires :

- le nombre de valeurs binaires n'est pas infini, il faut donc classer les échantillons analogiques en différents niveaux

- la fonction de quantification attribue le même niveau à tous les signaux situés dans une plage de tension donnée

Le nombre de niveaux de quantification est bien-sûr lié au nombre de bits N du CAN :

- un convertisseur 8 bits quantifie le signal analogique sur 256 niveaux, $q = 19,5mV$ si $E = 5V$ - un convertisseur 10 bits quantifie le signal analogique sur 1024 niveaux, $q = 4,9mV$ si $E = 5V$

- un convertisseur 16 bits quantifie le signal analogique sur 65536 niveaux, $q = 0,076mV$ si $E = 5V$

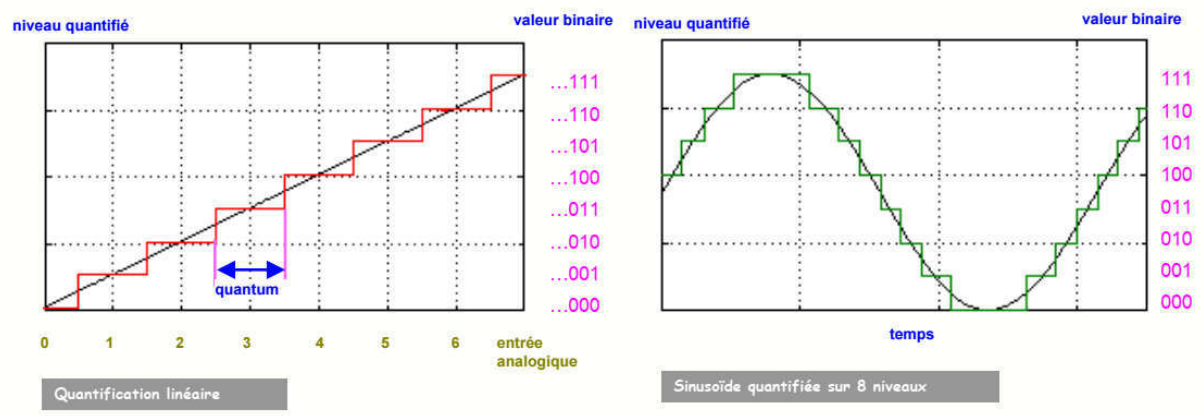


FIG. 2.13 : Quantification linéaire et Quantification d'une sinusoïde

2.3.1.2 Caractéristique de l'erreur de quantification, cas de la quantification uniforme

On s'intéresse dans ce paragraphe uniquement au cas de la quantification uniforme. On appelle erreur de quantification e la différence entre X et $Q(X)$.

$$e = X - Q(X) \tag{2.5}$$

L'opération de quantification dégrade le signal : - en remplaçant un échantillon par un autre de valeur voisine, on introduit une erreur d'arrondi quasiment aléatoire

- cette erreur d'arrondi est appelée bruit de quantification

- ce type de bruit qui se traduit en audio par une sorte de claquement n'apparaît qu'en présence du signal - on peut représenter le signal quantifié par la somme du signal initial et du bruit de quantification

- la valeur crête-crête du bruit est égale au quantum

- son amplitude est indépendante de l'amplitude du signal Le bruit de quantification se rapproche, par ses caractéristiques spectrales, du bruit blanc, mais il n'est pas gaussien puisque sa valeur crête-crête ne dépasse pas le quantum.

Le bruit de quantification diminue si la précision, c'est-à-dire le nombre de bits N, de la conversion augmente :

La quantification dégrade toujours la qualité du signal, mais cette dégradation peut être maîtrisée, de plus on peut toujours améliorer la qualité de la numérisation en augmentant le nombre de bits utilisés.

Lors de la quantification, deux sortes d'erreur peuvent être commises : l'erreur de granulation et l'erreur de saturation.

Une erreur de saturation se produit lorsque l'amplitude de l'échantillon à convertir est supérieure en valeur absolue à $Xmax$. Cette erreur est d'autant plus gênante qu'elle n'est pas bornée, on cherche donc à minimiser la probabilité de saturation.

Lorsque les échantillons sont d'amplitude inférieure à $Xmax$ en valeur absolue, l'erreur de quantification est appelée erreur de granulation. Cette erreur est bornée. Si

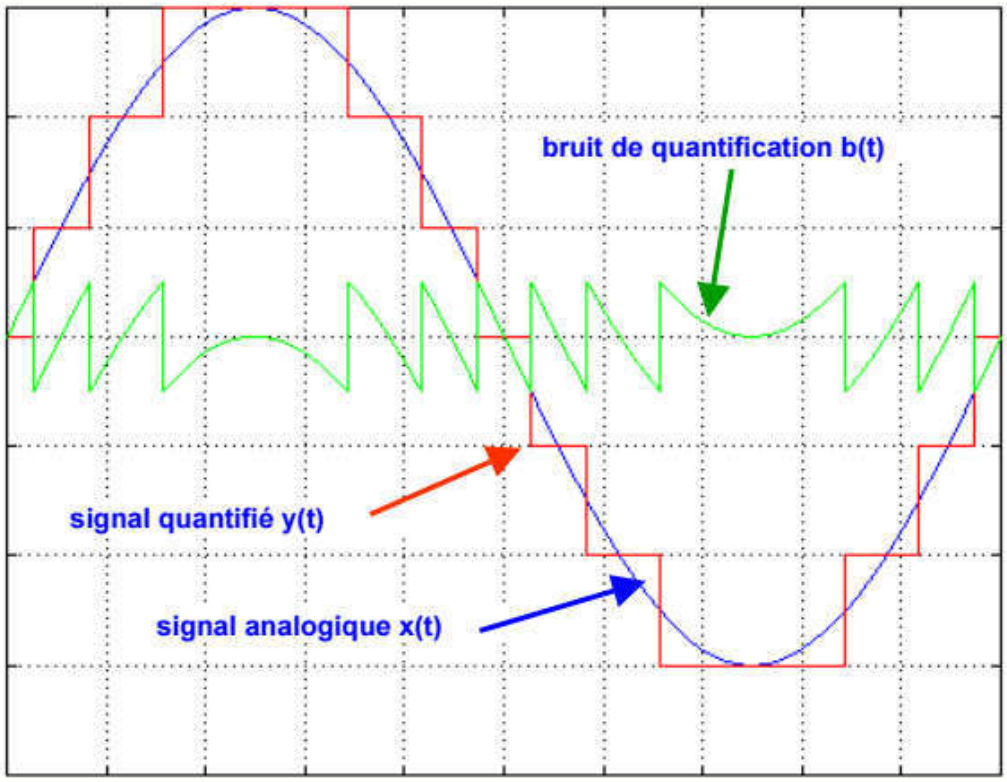


FIG. 2.14 : Bruit de la quantification uniforme

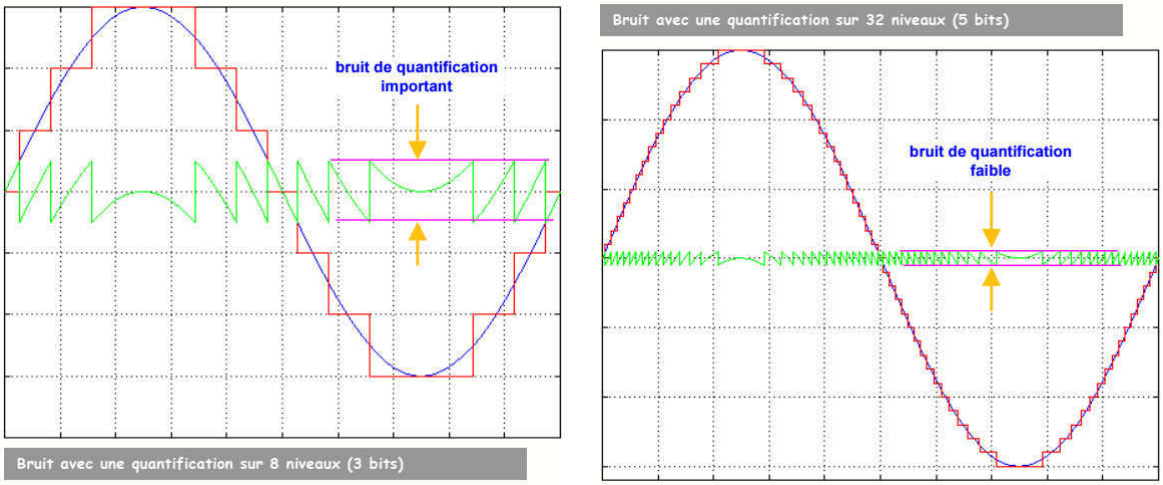


FIG. 2.15 : relation entre la précision et l'erreur de quantification

la quantification s'effectue par arrondi au plus proche voisin, l'erreur de granulation e_g en valeur absolue est inférieure à $q/2$.

$$e_g = X - Q(X) \qquad |e_g| \leq q/2 \tag{2.6}$$

Sous l'hypothèse, relativement générale, que eg est uniformément répartie entre $-q/2$ et $q/2$, on peut calculer la valeur moyenne et l'écart type de cette erreur.

La densité de probabilité de eg (notée $p(e_g)$) est dessinée ci-dessous :

Sous cette hypothèse :

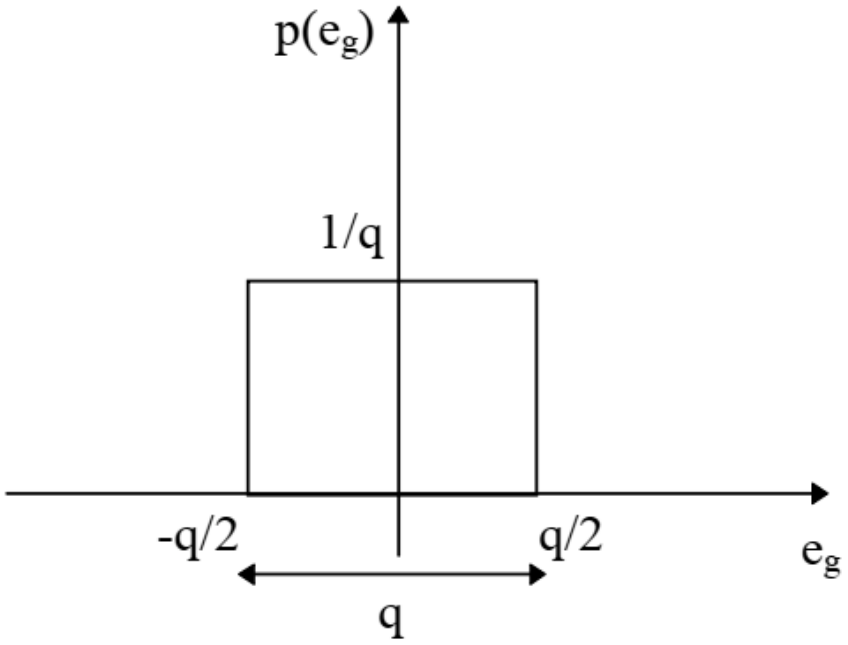


FIG. 2.16 : Densité de probabilité

$E(e_g) = \int_{\frac{-q}{2}}^{\frac{+q}{2}} e_g p(e_g) de_g = \frac{1}{q} \int_{\frac{-q}{2}}^{\frac{+q}{2}} e_g de_g = 0$

$E(e_g^2) = \sigma_g^2 = \int_{\frac{-q}{2}}^{\frac{+q}{2}} e_g^2 p(e_g) de_g = \frac{1}{q} \int_{\frac{-q}{2}}^{\frac{+q}{2}} e_g^2 de_g = \frac{q^2}{12}$

Et enfin $\sigma_g^2 = \frac{q^2}{12} = \frac{1}{3} \frac{X_{max}^2}{2^{2N}}$

Avec les mêmes hypothèses, Le rapport (noté RSB_{db}) entre la puissance du signal σ_x^2 et la puissance de l'erreur de granulation σ_g^2, peut s'exprimer en décibels par la relation suivante où N représente le nombre de bits du convertisseur :

$RSB_{db} = 10log_{10}(\frac{\sigma_x^2}{\sigma_g^2})$ et en remplalçant σ_g^2 par sa valeur (fonction de X_{max} et N)

$RSB_{db} \approx 10log_{10}(\sigma_x^2) + 6N - 10log_{10}(X_{max}^2) + 10log_{10}$

$RSB_{db} \approx 6N + 10log_{10}(3) - 20log_{10}(\Gamma)$ Où $\Gamma = X_{max}/\sigma_x$, facteur de surcharge, est le rapport entre la valeur maximale du convertisseur X_{max} et l'écart type des échantillons à convertir.

Le rapport signal sur bruit en dB dépend donc de façon linéaire de la puissance du signal en dB. Il est d'autant plus grand que la puissance du signal est grande.

2.3.1.3 Dynamique d'un quantificateur uniforme N bits

Appelons codeur un système effectuant une quantification uniforme (arrondi au plus proche voisin) puis une numérisation sur N bits.

La dynamique D du codeur est définie par :

$$D_{en \quad dB} = 10log_{10}(\frac{S}{B}) \tag{2.7}$$

– S est la puissance de crête du codeur, c'est-à-dire puissance de la sinusoïde d'amplitude maximale codable sans écrêtage,

– B est la puissance du bruit de quantification : $B = q^2/12$

Calcul de S la puissance crête du codeur :
Un codeur N bits peut convertir sans saturation des valeurs X comprises entre $-X_{max}$ et X_{max} avec $2X_{max} = (2N-1)q$. De ce fait

$S = \frac{1}{2}(\frac{2^N-1}{2}q)^2 \approx 2^{2N-3}q^2$

$D_{dB} \approx 10log_{10}(\frac{S}{B}) \approx 10log_{10}(\frac{2^{2N-3}q^2}{\frac{q^2}{12}})$

$D_{dB} \approx 10log_{10}(2^{2N}) + 10log_{10}(\frac{3}{2})$

$RSB_{db} \approx (6N + 1.76)dB$

Rajouter 1 bit au convertisseur revient à rajouter 6 dB à la dynamique. Il s'agit là d'une formule bien connue des concepteurs de systèmes d'acquisition.

On choisira le pas de quantification q en fonction de la précision désirée lors de la conversion et le nombre de bits N en fonction de la dynamique du signal à coder.

2.3.2 Quantification non-uniforme, quantification logarithmique

Avec une quantification uniforme, la précision absolue est la même pour les petites et les grandes valeurs. Il est parfois plus intéressant de travailler avec une précision relative à peu près constante. C'est le cas pour les signaux de parole, l'oreille ayant une sensibilité logarithmique.

La quantification de type logarithmique permet d'obtenir un rapport signal sur bruit de quantification à peu près constant quel que soit la puissance du signal. L'écart entre les valeurs de quantification n'est pas constant. Il croit logarithmiquement en fonction de l'amplitude du signal à quantifier. Une quantification logarithmique peut se réaliser par une compression des amplitudes suivie d'une quantification uniforme, puis d'une expansion des amplitudes.

2.3.2.1 Loi de compression expansion

La loi de compression est notée $C(x)$. La loi d'expansion est l'opération inverse.
$-X_{max} \leq x \leq X_{max} \quad y = C(x) \quad x = C^{-1}(y)$

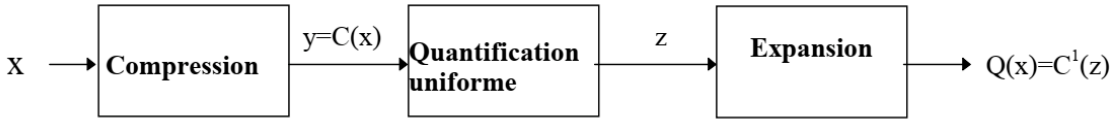


Fig. 2.17 : Quantification logarithmique

La loi de compression doit approcher une fonction logarithme. Deux lois sont utilisées en pratique : la loi A et la loi μ. Ces deux lois sont appliquées dans les codecs : circuits de conversion analogique numérique pour la téléphonie. La loi A est appliquée en Europe, la loi μ aux USA et au Japon.

La figure suivante représente les deux fonctions de compression ainsi obtenues. On s'aperçoit qu'elles sont pratiquement superposées. Sur la figure, on a normalisé X_{max} à 1.

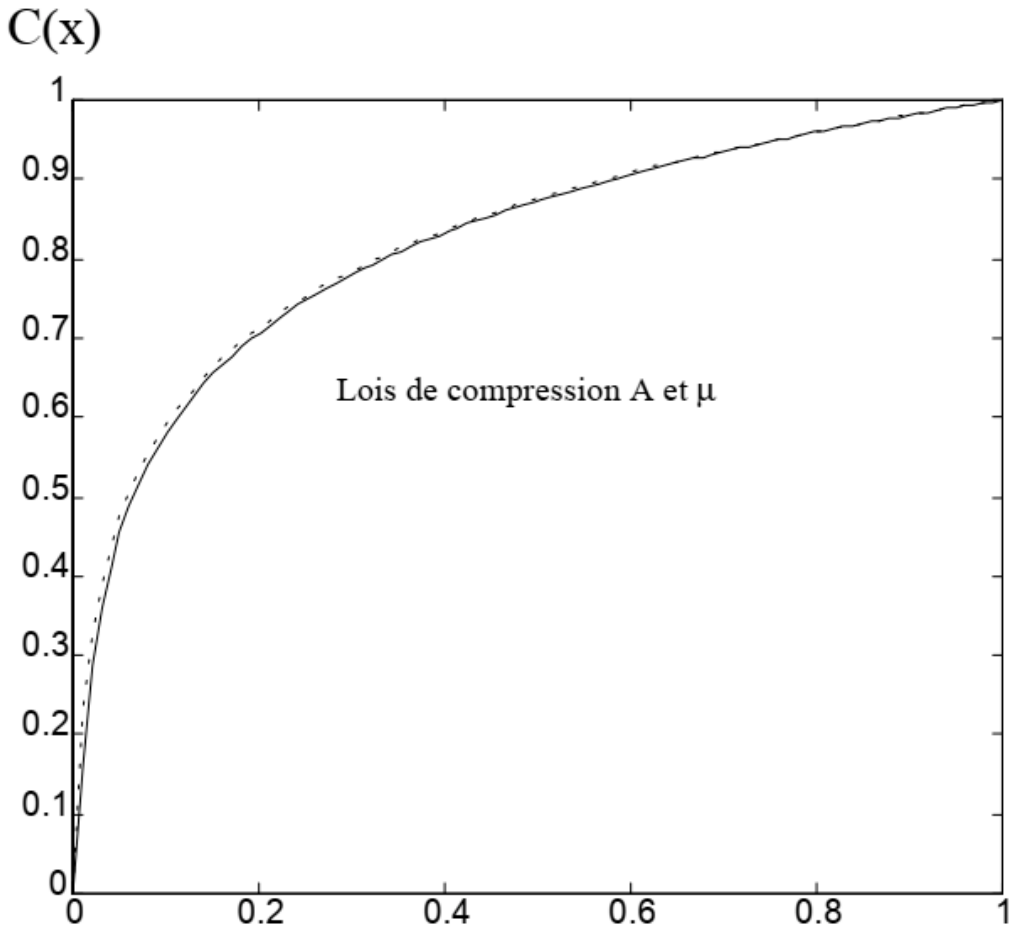


FIG. 2.18 : Lois de compression A et μ

La loi A est définie par la relation suivante :

pour $|x| < \frac{1}{A}$ $\quad C(x) = \frac{Ax}{1+log(A)}$

pour $|x| \geq 1_{\overline{A}}$ $\quad C(x) = \frac{1+log(Ax)}{1+log(A)}$

Avec A=87.6

2.3.2.2 Approximations par segments des lois de compression A et μ

Pour leur réalisation matérielle les lois A et μ sont approchées par des segments de droites. La loi A est approchée par une courbe à 13 éléments, la loi μ par contre à 15 segments. Elles sont appliquées dans ce cas avec une numérisation sur 8 bits.

En ce qui concerne la loi A, la pente du premier segment passant par l'origine, est 16. Puis les pentes des segments successifs sont obtenues par divisions successives par deux. La pente du dernier segment vaut donc 1/4. La courbe suivante représente la loi A à 13 segments. Là encore X_{max} est normalisé à 1. La numérisation est faite sur

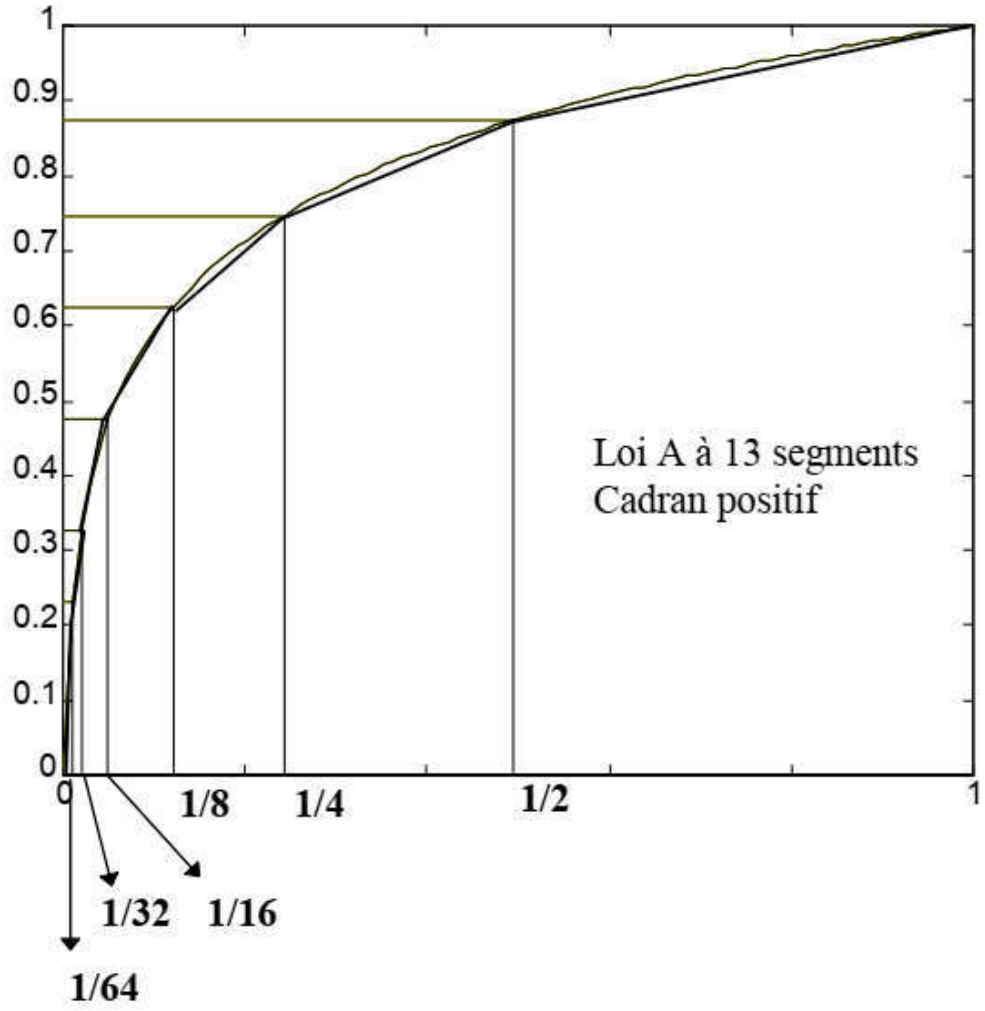
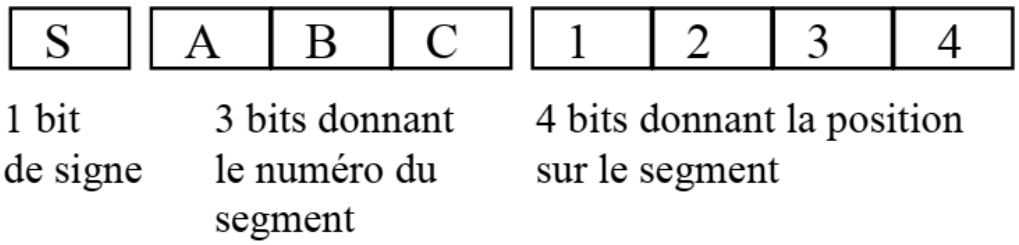


FIG. 2.19 : Approximations par segments des lois de compression A et μ

8 bits : Le rapport signal sur bruit de quantification obtenu avec la loi A est constant



sur une plage de signal. Dans le cas de la conversion sur 8 bits, on peut remarquer que les petits signaux sont amplifiés par un facteur 16 avant d'être convertis, ce qui revient à diviser par 16 le pas de quantification, c'est-à-dire à utiliser 12 bits de quantification (gain de 4 bits). Par contre pour les grands signaux, le pas de

quantification est multiplié par 4 par rapport à un convertisseur 8 bits uniforme, on perd donc 2 bits.

Cette méthode donne des résultats qualitatifs (notion subjective) comparable à une quantification linéaire sur 12 bits.

2.4 Les différents types de formats de données

Le choix d'une représentation binaire relève d'une problématique qui concerne toute plateforme numérique actuelle. Faire un choix d'une représentation ou d'une arithmétique représente une étape primordiale dans le développement d'une application numérique. Les arithmétiques virgule fixe et virgule flottante sont généralement utilisées dans un but de stockage ou de calcul. En général, les algorithmes de traitement numérique de signal demandent des capacités de calcul intensif. Par conséquent, le choix du bon format de représentation des nombres joue un rôle crucial pour une implémentation efficace de n'importe quel algorithme de traitement numérique de signal.

Dans le calcul numérique, le système de numération spécifie la méthode selon laquelle les nombres sont représentés comme étant une séquence de chiffres binaires et il spécifie également les règles pour effectuer les opérations arithmétiques (ex. addition, multiplication etc.) entre ces nombres. Dans la plupart du temps, les calculs scientifiques donnent seulement une approximation de la valeur exacte qui serait obtenue avec une précision infinie. Ceci est une conséquence du nombre limité de bits utilisés dans le système de numération. Quel que soit l'arithmétique virgule fixe ou l'arithmétique virgule flottante utilisée, seulement un nombre fini de bits est utilisé pour la représentation des nombres réels.

La précision limitée des standards de codage peut être évaluée selon deux perspectives diffé- rentes. La première concerne la précision des calculs déterminée par l'étape de quantification du système de numération (la distance entre deux nombres). Le second aspect est relatif à la variation de la dynamique maximale permise par la représentation. Cette variation de la dynamique d'un système de numération est donnée par l'ensemble des valeurs possibles qui peuvent être représentées. Elle est évaluée à travers le logarithme du quotient entre les amplitudes maximale et minimale du signal comme indiqué à l'équation (2.8).

Par conséquent, la comparaison entre le codage en virgule flottante et en virgule fixe est fondée en particulier sur l'analyse de la précision numérique et la variation de la dynamique selon la relation :

$$D_{dB} = 20log_{10}(\frac{X_{MAX}}{X_{MIN}}) \tag{2.8}$$

Généralement, les algorithmes pour les systèmes embarqués sont développés en utilisant l'arithmétique virgule flottante afin d'éviter les problèmes liés à la longueur finie d'un mot de code. Ce processus valide l'intégrité de l'algorithme et vérifie si l'algorithme proposé répond bien au cahier des charges. Même si l'erreur inhérente

à l'exactitude de calcul existe toujours, elle demeure faible par rapport à celle obtenue par l'arithmétique en virgule fixe. Par conséquent, le calcul en virgule flottante garantit une précision et une variation de dynamique suffisantes dans la plupart des cas.

Néanmoins, la plupart des implémentations VLSI utilisent l'arithmétique en virgule fixe pour réduire la surface et la consommation d'énergie et obtenir un matériel rentable. En contrepartie, une dégradation de la précision des calculs est produite en raison du nombre limité de bits utilisés dans la représentation des données. En effet, l'utilisation du codage en virgule fixe introduit des bruits de quantification lors de l'élimination de bits à travers les opérations de saturation (arrondi) et troncature. En outre, cela conduit à l'apparition de débordements à chaque fois que la longueur du mot de code de la partie entière est insuffisante pour représenter l'ensemble de la variation de la dynamique.

Dans cette partie, les différents types d'arithmétiques pour les systèmes à base de DSP sont abordés. En premier lieu, l'arithmétique virgule flottante est introduite, ensuite, l'arithmétique fixe est présentée en la comparant avec celle virgule flottante afin de montrer les avantages et les inconvénients de chacune de ces deux représentations.

2.4.1 Représentation des nombres entiers

2.4.1.1 Numération simple de position

Un entier p (compris entre 0 et B^N-1) est décomposé dans la base B d'une façon unique selon la formule suivante :

$$p = \sum_{i=0}^{N-1} a_i B^i \tag{2.9}$$

Les a_i sont des chiffres compris entre 0 et $B-1$. L'entier p est noté dans la base B de la façon suivante : $(a_{N-1}...a_2a_1a_0)_B$ ou tout simplement $(a_{N-1}...a_2a_1a_0)$ si la base B a été prédéfinie auparavant. Ainsi la chaîne $a_{N-1}...a_2a_1a_0$ est nommée l'entier simple associé à p en base B (ESA_B) selon :

$$ESA_B(a_{N-1}...a_2a_1a_0) = \sum_{i=0}^{N-1} a_i B^i \tag{2.10}$$

2.4.1.2 Notation Signe - Valeur Absolue (SVA)

La notation en numération simple d'un nombre signé correspond à la représentation de la valeur absolue de ce nombre en numération simple de position en ajoutant un symbole spécifiant le signe du nombre à représenter. L'inconvénient de cette représentation se manifeste par les opérations mathématiques effectuées sur des nombres respectant cette notation. En effet, à titre d'exemple, l'addition de deux nombres nécessite l'utilisation de deux algorithmes : un algorithme d'addition si les

deux nombres possèdent le même signe et un algorithme de soustraction dans le cas contraire.

2.4.1.3 Présentation en complément à la base

Afin d'éviter le problème de prise en compte du signe des opérandes présent pour la notation SVA, le système de numération des entiers, en complément à la base B a été créé. Par conséquent dans le cas où B est une base paire, l'entier $p = (a_{N-1}...a_2a_1a_0)_{\bar{B}}$ est représenté de la façon suivante :

-Si $0 \leq p \leq B^N/2 - 1$, la représentation est celle de l'équation (2.9) :

$$ESA_B(a_{N-1}...a_2a_1a_0) = p$$

-Si $-B^N/2 \leq p \leq -1$, le codage est réalisé selon l'expression suivante :

$$ESA_B(a_{N-1}...a_2a_1a_0) = B^N + p$$

Par conséquent, grâce à cette notation, on utilise un seul algorithme d'addition modulo B^N pour réaliser l'addition de tous les entiers compris entre $-B^N/2$ et $B^N/2$. Si le signe du résultat de l'addition est différent de celui des opérandes, il en résulte un dépassement de capacité. Dans le cas où $B = 2$, cette notation est connue sous le nom de notation à complément à deux ($CA2$).

2.4.2 Le format virgule fixe

Soit x un réel écrit sous la forme $x = pK$ avec p un entier codé selon l'une des représentations décrites précédemment où K est appelé le facteur d'échelle. Ce facteur d'échelle est une puissance de la base B préalablement fixée : $K = B^n$. La position de la virgule peut alors être référencée par un entier n relativement au bit de poids faible ($LeastSignificantBit$ ou LSB) ou par un entier m relativement au bit de poids fort ($MostSignificantBit$ ou MSB). Dans ce cas, le réel x peut se représenter selon la figure 2.20 où les deux entiers n et m représentent les distances entre la virgule et le LSB et le MSB ($MostSignifiantBit$) respectivement.

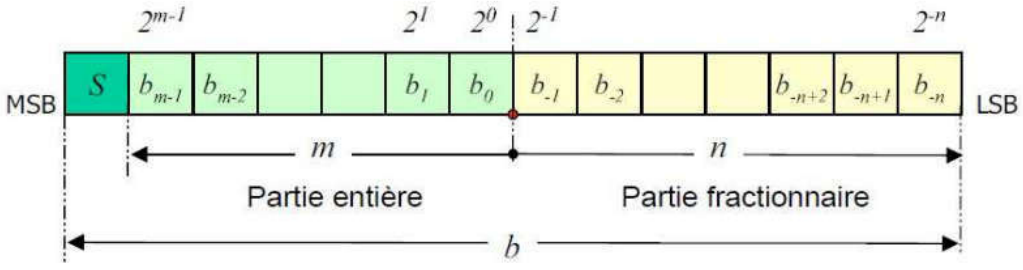


FIG. 2.20 : Représentation des données en virgule fixe

La figure 2.20 représente un codage en virgule fixe d'un nombre avec un bit de signe S, m bits pour la partie entière et n bits pour la partie fractionnaire. Le nombre de bits utilisés dans cette représentation est $b = m + n + 1$.

Le format d'une donnée en virgule fixe est entièrement défini par la représentation choisie et la largeur de sa partie entière et de sa partie fractionnaire. Généralement, les nombres en arithmétique virgule fixe utilisent une représentation en complément à deux. La valeur d'un nombre est donnée par la relation suivante :

$$x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i \tag{2.11}$$

Cette représentation possède des propriétés arithmétiques intéressantes pour les opérations d'addition et soustraction. En effet, même si les résultats intermédiaires d'une série d'additions sont en dehors du domaine de définition du codage, le résultat final sera correct si celui-ci appartient au domaine de définition du codage. De plus, l'implantation dans les processeurs numériques des opérateurs traditionnels utilisant ce code est simplifiée lorsque l'on utilise cette représentation.

$$x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i \tag{2.12}$$

Par ailleurs, elle possède l'avantage de n'autoriser qu'une seule représentation possible pour 0. Par conséquent, le domaine des valeurs possibles n'est pas symétrique par rapport à l'origine, et il possède 2^{m+n} valeurs négatives et $2^{m+n}-1$ valeurs positives, soit :

$$D = [-2^m; 2^m - 2^{-n}] \tag{2.13}$$

La notation en format virgule fixe est simple et permet d'effectuer les calculs rapidement. En contrepartie, comme le facteur d'échelle K est fixé, la notation en virgule fixe ne permet pas de représenter des nombres d'ordre de grandeur très différents. De plus, il n'est pas toujours facile de fixer a priori le facteur d'échelle si l'ordre de grandeur des valeurs prises par la donnée n'est pas connu par avance. La notation en virgule flottante permet de résoudre ces problèmes.

2.4.3 Le format virgule flottante

Le système des nombres en virgule flottante est le standard de codage le plus utilisé lorsqu'une grande précision est requise. Il représente un nombre réel selon une notation scientifique avec une partie fractionnaire appelée *mantisse* et un facteur d'échelle appelé *exposant*, comme représenté dans la figure 2.21. L'exposant est défini comme une puissance de la base (typiquement 2 et 10) et il est utilisé comme un facteur d'échelle explicite qui change durant les calculs permettant ainsi une dynamique large pour les valeurs représentées. La mantisse détermine la précision de nombre représenté. Soit x un réel signé écrit en format flottant dans la base B avec un signe S, une mantisse M et un exposant E. La valeur associée à ce réel est donnée par l'équation suivante :

$$x = (-1)^S M B^E \tag{2.14}$$

Comme le nombre de bits accordés à la mantisse et à l'exposant peut prendre un grand nombre de valeurs possibles, un format virgule flottante normalisé a été introduit. La norme IEEE pour l'arithmétique virgule fixe binaire (IEEE 745-2008)

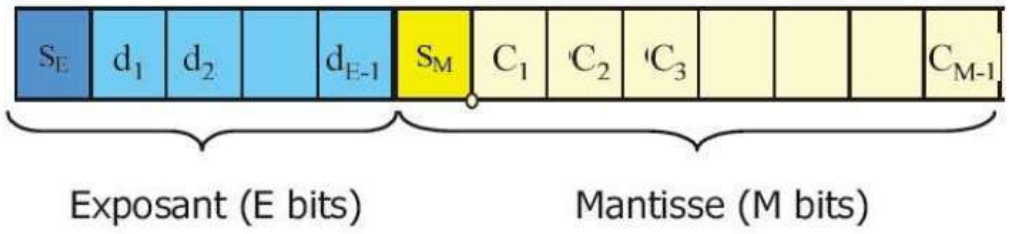


FIG. 2.21 : Représentation des données en virgule flottante

est utilisée dans la plupart des CPUs d'aujourd'hui. Elle spécifie le format virgule flottante et également les modes d'arrondis. Elle décrit non seulement comment les opérations arithmétiques doivent être effectuées, mais aussi le traitement des exceptions (division par zéro, les débordements,...). La valeur d'un nombre représenté selon le format virgule flottante de la norme IEEE 745 est calculée :

$$x = (-1)^S 1M 2^{E-biais} \tag{2.15}$$

La mantisse est normalisée pour représenter une valeur dans l'intervalle [1, 2]. Par conséquent, la valeur de son premier bit est fixée à 1 et devient implicite. La valeur de l'exposant est codée comme un nombre non signé. Alors, afin de représenter les nombres inférieurs à 1, un biais est introduit. Ce biais dépend du nombre de bits alloués à l'exposant : $biais = 2^{N_e-1}-1$. Dans le cas d'un format de précision simple, le biais est 127 et la gamme de l'exposant est $[-126, 127]$. Pour la double précision, le biais est 1023 et l'exposant prend ses valeurs dans $[-1022, 1023]$.

Il est à noter que la représentation d'un nombre n'est pas unique et les zéros à gauche de la mantisse dégradent la précision. Par exemple, considérons la base 10 avec 4 chiffres de mantisse, peut être représenté par 0.03110^2 ou 3.14210^0. Ces deux représentations ne possèdent pas la même précision. Par conséquent, pour éviter cette ambiguïté, une représentation sans zéro à gauche est proposée et correspond à la représentation normalisée en virgule flottante. Dans ce cas, le premier bit de la mantisse représente le coefficient 1/2 et est fixé à 1. La valeur de ce bit est inchangée au cours du traitement et ne figure pas dans dans la représentation.

Dans le cas où l'exposant est une puissance de deux (cas binaire), la mantisse et l'exposant sont codés avec une représentation en signe valeur absolue, la valeur de la donnée x est la suivante :

$$x = 2^u(-1)^{S_M}(\frac{1}{2} + \sum_{i=1}^{M-1} C_i 2^{-i-1}), u = (-1)^{S_E} \sum_{i=1}^{E-1} d_i 2^i \tag{2.16}$$

La valeur 0 n'est pas représentable selon l'équation (2.16). Par conséquent, le domaine de définition des valeurs représentables avec ce codage est l'union de deux sous-domaines donnés par l'expression suivante :

$$D_R = [-2^K; -2^{-K-1}] \cup [-2^{-K-1}; 2^K], K = 2^{E-1} - 1 \tag{2.17}$$

L'écart minimal entre deux valeurs représentables par le code considéré détermine le pas de quantification dont la valeur varie en fonction de la valeur représentée.

Pour les valeurs de x comprises dans l'intervalle $[-2^u; -2^{u-1}] \cup [2^u; 2^{u-1}]$, le pas de quantification est égal à :

$$q = 2^u 2^{-(M+1)} \tag{2.18}$$

L'étape de quantification de la norme de codage en virgule flottante dépend de la valeur à représenter. En effet, lorsque la valeur de l'exposant augmente (respectivement diminue), la distance entre deux nombres successifs représentables devient plus importante (respectivement plus faible), i.e. le pas de quantification augmente (respectivement diminue) conformément à la relation d'ordre :

$$2^{-(M+1)} < \frac{q}{|x|} < 2^{-M} \tag{2.19}$$

Ainsi, la présence d'un facteur d'échelle explicite permet d'adapter le pas de quantification à la valeur de la donnée à coder.

2.4.4 Comparaison arithmétiques virgule flottante et virgule fixe

Nous présentons dans cette parite une comparaison entre le format virgule flottante et le format virgule fixe afin de mettre en évidence leurs avantages et inconvénients. Cette comparaison est fondée sur l'analyse de deux aspects. Le premier est l'aspect arithmétique et le deuxième est le coté implantation matérielle puisque la recherche du format optimal de représentation conduisant à une implémentation matérielle efficace en termes de consommation d'énergie et de ressources.

2.4.4.1 Comparaison d'un point de vue arithmétique

La qualité du codage est analysée, d'un point de vue arithmétique, en évoquant la dynamique de chacun des deux codages et également le rapport signal à Bruit de quantification.

Comparaison de la dynamique Une représentation efficace des nombres doit à la fois supporter une dynamique large mais également disposer d'une grande précision. En d'autres termes, la représentation d'un nombre devient plus polyvalente avec un rapport plus élevé de la dynamique et la précision. Typiquement, les deux formats virgule fixe et virgule flottante sont symétriques par rapport à leur espace des valeurs représentables. Donc, la dynamique dont l'expression est donnée par l'équation (2.8) est utilisée pour mesurer la fidélité du format. Plus la dynamique est importante, plus le débordement est faible. Dans le cas de la représentation en virgule flottante la dynamique s'exprime par l'équation (2.20) :

$$D_{dB} \approx 20log_{10}(2^{2K+1}), K = 2^{N_e-1} - 1 \tag{2.20}$$

N_e représente le nombre de bits accordés à l'exposant. Pour un format en simple précision où l'exposant est représenté par 8 bits, la dynamique devient :

$$D_{dB} = 20log_{10}(2^{(2^8-1)}) = 20log_{10}(2^{255}) \approx 1535dB \tag{2.21}$$

Pour le format virgule fixe, la dynamique varie linéairement en fonction du nombre de bits, b, utilisé pour la représentation :

$$D_{dB} = 20log_{10}(\frac{X_{MAX}}{X_{MIN}}) = 20log_{10}(2^{b-1})dB \tag{2.22}$$

En appliquant les propriétés logarithmiques et des simplifications, la relation précédente devient :

$$D_{dB} = 20(b-1)log_{10}(2) \approx 6.02(b-1) \tag{2.23}$$

La variation de la dynamique en fonction de la longueur du mot de code est plus importante pour les nombres en virgule flottante que pour les nombres en arithmétique virgule fixe. A titre d'exemple, dans le tableau suivant le format en précision simple est comparé avec différents types de représentation des données en format virgule fixe. Une différence importante entre les différents formats peut être observée. En effet, le format virgule fixe sur 128 bits possède significativement une petite dynamique par rapport à la représentation virgule flottante sur 32 bits.

Format	**Dynamique (dB)**
Précision simple	1535
virgule fixe 16 bits	90
virgule fixe 32 bits	186
virgule fixe 64 bits	379
virgule fixe 128 bits	764

Tab. 2.2 : Comparaison de la dynamique

En comparant les équations (2.21) et (2.22), il est clair que la dynamique augmente exponentiellement pour le format virgule flottante et linéairement pour le format virgule fixe en fonction du nombre de bits accordés pour la représentation. Dans la figure 2.22, l'évolution de la dynamique en fonction du nombre de bits utilisés est présentée pour les deux formats virgule fixe et virgule flottante. Dans cet exemple, la taille de l'exposant représente le quart de la longueur totale du mot de code de la représentation. Nous pouvons observer que, lorsque la longueur d'un mot de code dépasse 16 bits, la dynamique du format virgule flottante devient plus large par rapport au format virgule fixe. Par conséquent, la représentation virgule flottante sur 32 bits peut être utilisée dans la plupart des applications sans aucun risque de débordement.

Le processus de codage d'un nombre en arithmétique virgule fixe correspond à une représentation d'une valeur réelle x par une autre valeur $\widehat{x}$ appartenant au domaine de codage. A chaque fois que le nombre réel à coder dépasse l'intervalle des valeurs permises par la norme de codage (ce qui veut dire que $x \notin [\widehat{X_{min}}, \widehat{X_{max}}]$), un débordement se produit et une erreur est introduite. Il est alors nécessaire de définir une procédure de gestion de débordement afin d'être en mesure d'attribuer un mot de code dans cette situation. Deux techniques sont alors classiquement utilisées pour

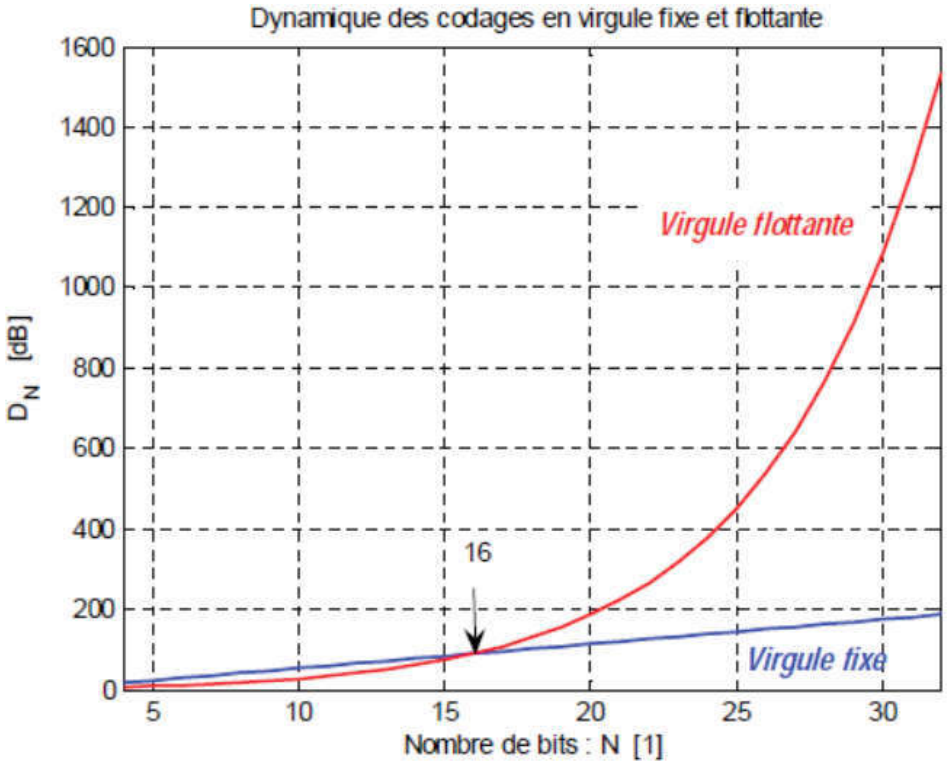


FIG. 2.22 : Comparaison de l'évolution du niveau de la dynamique pour les représentations virgule fixe et virgule flottante

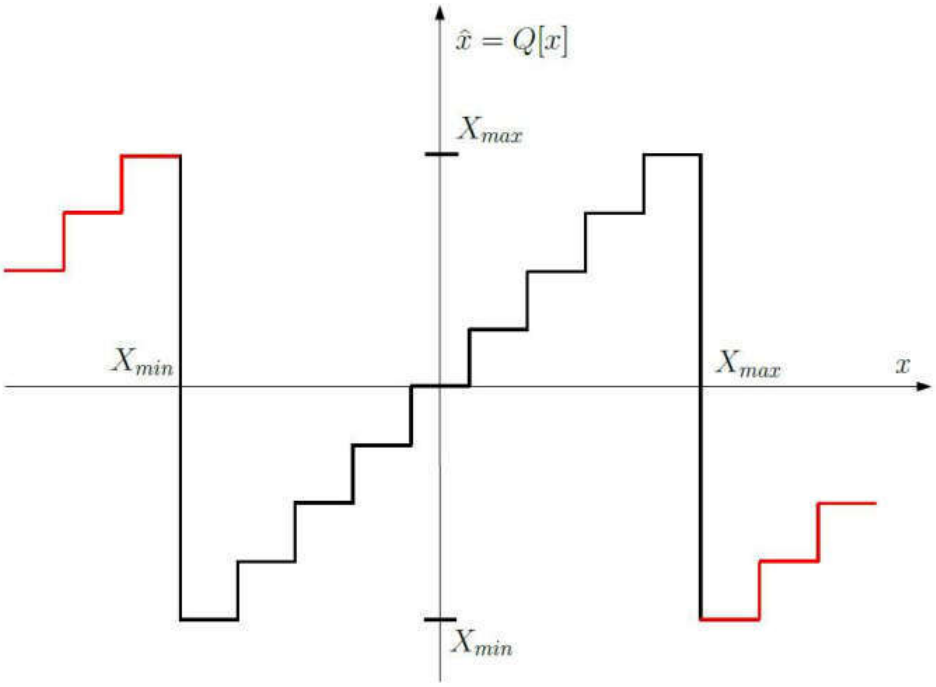


FIG. 2.23 : Effet du débordement utilisant la technique de l'enveloppe autour de la valeur

effectuer cette gestion du débordement. La première technique, décrite à la figure 2.23,consiste à établir une enveloppe autour de la valeur du nombre réel à coder. Cette technique est équivalente à une arithmétique modulaire puisque toute valeur

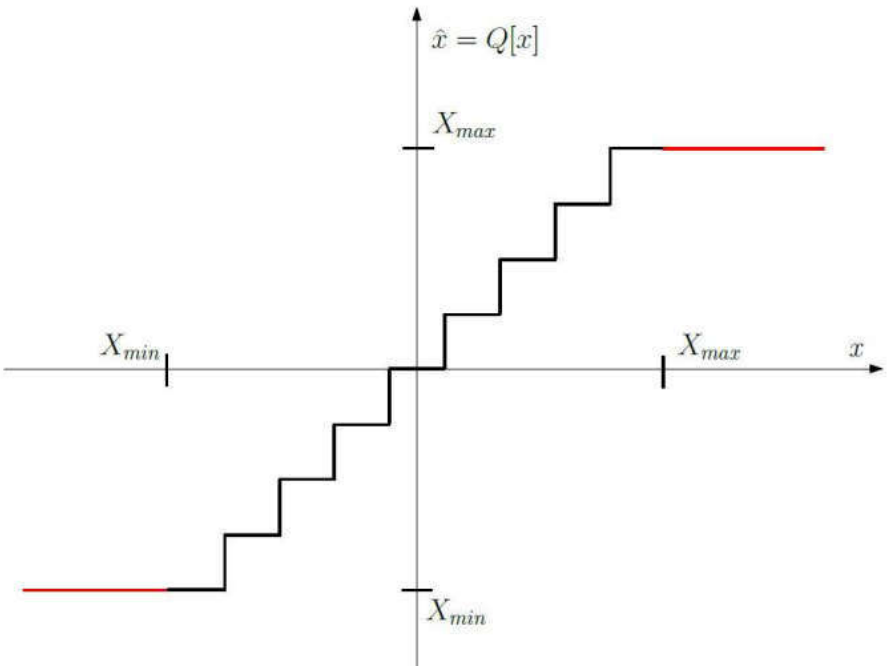


FIG. 2.24 : Effet du débordement utilisant la technique de saturation

dépassant les bornes est remplacée par sa valeur modulo 2^b.

La deuxième méthode qui peut être appliquée est l'arithmétique de saturation. Dans ce cas, n'importe quelle valeur dépassant le domaine de codage est remplacée par le nombre représentable le plus proche (la borne maximale ou minimale). Cette technique est présentée par la figure 2.24. L'erreur introduite est plus petite que dans le cas de l'arithmétique modulaire. Cependant, à l'inverse de la technique de l'enveloppe autour de la valeur, l'implémentation de l'arithmétique de saturation nécessite une complexité matérielle plus importante et, par conséquent, son utilisation est limitée dans la pratique.

Comparaison du rapport Signal à Bruit de Quantification Pour les applications de traitement numérique de signal (DSP), la précision des calculs est mesurée le plus souvent à travers le critère de Rapport Signal à Bruit de Quantification (RSBQ). Le RSBQ est défini par le rapport entre la puissance du signal (P_x) et la puissance de l'erreur (Pe) due au bruit de quantification et il est généralement exprimé selon une échelle logarithmique :

$$RSBQ_{dB} = 10log(\frac{P_x}{P_e}) = 10log\frac{E[x^2]}{[e^2]} \tag{2.24}$$

Pour un signal x, sa puissance P_x peut s'exprimer en fonction de sa dynamique linéaire D_x et du rapport Kx entre la racine carrée de la puissance P_x et la dynamique D_x :

$$P_x = (K_x D_x)^2 \tag{2.25}$$

Par conséquent, dans le cas de l'arithmétique virgule fixe, le RSBQ suit une relation linéaire en fonction de la dynamique, selon :

$$\begin{aligned} RSBQ_{dB} =& 20log(D_x) + 20log(K_x) - 10log(P_e) \\ =& D_{dB} + 20log(K_x) - 10log(P_e) \end{aligned}$$

Comme le format virgule fixe possède une quantification uniforme, le RSBQ dépend linéairement de l'amplitude du signal. Lorsque l'amplitude du signal augmente, le rapport bruit de quantification devient plus large et le RSBQ s'améliore.

Considérons le cas d'un signal sinusoïdal à grande échelle avec une amplitude $A = 2^n$. La variance du signal est : $\sigma_x^2 = \frac{A^2}{2}$. Le RSBQ devient :

$$RSBQ = 20log(2^n\sqrt{\frac{3}{2}}) \approx 1.76 + 6.02n \qquad dB \tag{2.27}$$

Pour conclure, dans le cas de la représentation virgule fixe avec un signal utilisant la totalité de la dynamique disponible, chaque bit additionnel conduit à l'augmentation du RSBQ de 6 dB approximativement.

Contrairement au cas de l'arithmétique virgule fixe, l'arithmétique virgule flottante possède l'avantage d'une étape de quantification proportionnelle à l'amplitude du signal. La valeur du RSBQ utilisant une échelle logarithmique est donnée par l'expression (2.28) et dépend du nombre de bits m utilisé pour la mantisse. En contre partie, le RSBQ de la représentation virgule flottante ne varie pas en fonction de l'amplitude du signal, il peut être considéré constant pour toutes les valeurs de x.

$$RSBQ = 10log(\frac{E[x^2]}{e_{vf}}) = 10log(5.55 * 2^{2m}) \approx 7.44 + 6.02m \tag{2.28}$$

Dans le tableau (2.3), une comparaison entre le RSBQ des deux formats virgule flottante et virgule fixe est présentée. Nous pouvons remarquer que, pour un même nombre de bits, l'arithmétique virgule fixe garantit un RSBQ plus large que celui de l'arithmétique virgule flottante si le nombre est correctement mis à l'échelle.

Format	**RSBQ (dB)**
Précision simple	151
Précision double	326
virgule fixe 32 bits	194
virgule fixe 64 bits	387

Tab. 2.3 : Comparaison du RSBQ

Comparaison pour les opérations de calcul

D'après ce qui précède, il est clair que le format virgule flottante est plus performant que la représentation virgule fixe car il permet d'obtenir une dynamique suffisante et également une meilleure précision. Cependant, ce résultat est obtenu au prix d'une complexité des opérateurs arithmétiques. L'addition et la multiplication étant les deux opérations les plus génériques, nous étudions ci-après les coûts de réalisation de ces opérations.

L'addition Le nombre en format virgule fixe est nativement dans le format binaire. Sous cette hypothèse, il est possible d'additionner deux nombres représentés en virgule fixe en utilisant un circuit d'entraînement de report d'addition simple après l'alignement des points binaires de deux nombres. L'effort total pour l'addition en arithmétique virgule fixe ϵ^a_{fix} est égal à l'effort (le coût de réalisation) de l'addition binaire ϵ_a

$$\epsilon^a_{fix} = \epsilon_a \tag{2.29}$$

Cependant, le format virgule flottante est avant tout un format compressé. Il est donc nécessaire tout d'abord de décompresser le nombre en virgule flottante afin d'avoir accès au signe, à l'exposant et à la mantisse. Ensuite, l'exposant doit être normalisé en comparant les bits dans la partie exposant des deux nombres et en décalant les bits de la mantisse d'une manière telle que les exposants soient égaux. Si le bit de signe est réglé, les bits de la mantisse sont convertis à leurs formes en complément à deux avant l'addition. Après l'addition, le résultat doit être converti au format complément à deux s'il est négatif. Comparativement, l'addition de deux nombres en arithmétique virgule fixe est relativement plus simple que l'addition en format virgule flottante. Soit ϵ_{cmp} l'effort nécessaire pour la comparaison des exposants et soit ϵ_{cmp} l'effort demandé pour le calcul des compléments à deux des deux nombres, l'effort total pour réaliser une addition en virgule flottante ϵ^a_{flt} :

$$\epsilon^a_{flt} = \epsilon_a + \epsilon_{cmp} + 2\epsilon_{cal} \tag{2.30}$$

Les efforts ϵ_a, ϵ_{cmp} et ϵ_{cal} sont tous de l'ordre de $O(N)$ lorsque N est le nombre de bits accordés à la sortie de l'additionneur. Le temps pour la décompression du format virgule flottante est négligeable puisque cette opération est triviale du point de vue matériel.

La multiplication La multiplication de deux nombres binaires signés au format virgule fixe nécessite l'utilisation d'un multiplicateur matériel. Le résultat de la multiplication possède davantage de bits dans son niveau de dynamique et sa précision. La position du point binaire du nombre résultant en arithmétique virgule fixe est déterminée en prenant en compte les positions du point binaire des deux nombres impliqués dans la multiplication. Habituellement, les bits les moins significatifs du résultat sont rejetés par divers modes d'arrondis ou de troncature pour obtenir le résultat final. Par la suite, l'effort total de la multiplication en format virgule fixe ϵ^m_{fix} est ϵ_m c'est à dire le coût correspondant à une multiplication binaire, soit :

$$\epsilon^m_{fix} = \epsilon_m \tag{2.31}$$

Dans le cas d'une multiplication entre des nombres représentés en virgule flottante, leurs exposants sont simplement additionnés, leurs mantisses et leurs bits de signe sont multipliés entre eux. L'effort total de la multiplication en virgule flottante ϵ^m_{flt} est :

$$\epsilon^m_{flt} = \epsilon_m + \epsilon_a \tag{2.32}$$

La complexité algorithmique d'une multiplication (cas virgule flottante) peut être supérieure à $O(N^2)$ et le temps pour l'addition est $O(N)$ avec N est le nombre de bits accordés à la longueur du mot de code à la sortie du multiplicateur.

2.4.4.2 Comparaison de point de vue implémentation matérielle et logicielle

Chaque implémentation logicielle ou matérielle possède ses propres contraintes qui déterminent le choix de l'arithmétique la plus adaptée. Dans le cas d'une implémentation logicielle, les longueurs du mot de code sont fixées ou bien alors elles possèdent un nombre limité de représentations dans le format virgule flottante ou virgule fixe. Cependant, le format de la représentation en virgule fixe est flexible dans la mesure où le point binaire n'est pas fixe. Dans les plateformes logicielles typiques, les tailles d'un mot de code en arithmétique virgule fixe sont généralement d'un octet (8bits), moitié d'un mot (16 bits), un mot (32 bits) et aussi longue que mot double (64 bits). Les nombres en virgule flottante sont habituellement selon deux formats : soit la précision simple ou la précision double. De plus, il convient de préciser que les plateformes processeurs supportent les opérations des vecteurs de données SIMD (Single Instruction Multiple Data). De telles instructions fonctionnent sur un ensemble de données de même taille et de même type rassemblées en un bloc de données, d'une taille fixe, appelé vecteur. Il est alors possible de configurer le chemin de données selon une taille normalisée (le nombre de bits est généralement multiple de quatre ou huit) afin de contrôler plus précisément les longueurs des mots de code en arithmétique virgule fixe.

L'implémentation matérielle ouvre également des horizons pour des unités en virgule flottante personnalisées. Dans certain litérature , les spécifications des opérateurs en virgule flottante sont décrites en utilisant une bibliothèque C++ paramétrée d'opérateurs en virgule flottante. Il en résulte une génération automatique des implémentations optimales des opérateurs en virgule flottante dans le matériel de telle sorte que le temps de calcul est suffisamment petit pour atteindre la fréquence désirée de l'opération. L'impact du nombre de bits accordés à l'opérateur en virgule flottante sur le niveau de la dynamique et la précision de l'opération n'est pas aussi simple que dans le cas de système des nombres en virgule fixe. En définitive, il s'avère difficile de faire un choix entre le format virgule fixe et le format virgule flottante sans explorer explicitement toutes les opérations.

Certaines stratégies fournissent une bibliothèque paramétrable permettant de faire un choix entre les deux formats à partir de l'étude de différents compromis obtenus à partir de simulations. Certain spécialiste disent que le niveau de la dynamique est considéré pour optimiser le format d'un nombre, i.e. minimiser la dynamique des calculs tout en contrôlant la précision à un niveau permettant de ne pas dépasser les coûts de la virgule flottante. Nous pouvons également remarquer que de nombreuses applications de traitement du signal sont linéaires et ces systèmes peuvent faire usage du facteur d'échelle et donc travailler avec une gamme dynamique normalisée. En outre, les applications de traitement du signal et en particulier les algorithmes de communications numériques sont conçus pour fonctionner en présence de bruit de canal dont la valeur est beaucoup plus grande par rapport au bruit de quantification introduit, même lorsqu'une quantification 16 bits est utilisée. Par conséquent, la précision requise pour la mise en œuvre de ces algorithmes n'est pas très stricte. Dans de telles circonstances, il est possible de profiter pleinement de la représentation à virgule fixe. Par conséquent, les applications de traitement numérique de signal et de communications numériques sont le domaine d'intérêt de ce cours.

Au lieu d'utiliser un arbre d'addition à virgule fixe, d'autres travaux utilisent un sys-

tème de représentation hybride où les nombres à additionner sont normalisés relativement au plus grand exposant et où l'addition à virgule fixe est utilisée pour ajouter la mantisse. Cela réduit l'effort de calcul de l'exposant à chaque fois.

Une autre possibilité consiste à utiliser le format virgule fixe double. Dans ce cas, un seul bit est utilisé pour représenter l'exposant et la mantisse qui correspond à un nombre signé en virgule fixe. Une telle technique permet de réaliser un compromis entre dynamique et précision en étendant le concept de l'arithmétique virgule flottante. Une troisième alternative utilise un système de nombres en virgule flottante en bloc . Dans ce cas, lorsqu'un groupe de nombres partagent un même exposant, la mantisse de ces nombres peut être différente. Cette technique a été implémentée pour des nombres en virgule flottante dans plusieurs algorithmes de traitement de signal dont notamment la bibliothèque DirectX pour les applications graphiques à haut niveau de dynamique. L'idée commune à toutes ces approches hybrides vise à tirer avantage de la grande plage de dynamique et la haute précision de la représentation en virgule flottante tout en conservant la flexibilité de la représentation en virgule fixe.

Conclusion L'arithmétique virgule flottante présente l'avantage d'une dynamique et d'une précision supé- rieures à l'arithmétique virgule fixe. A l'opposé, l'arithmétique virgule fixe, plus flexible, possède des opérateurs moins complexes et conduit à des implémentations moins coûteuses en termes de consommation d'énergie et de ressources matérielles. Cependant, l'implantation en virgule fixe nécessite la maîtrise de la précision des calculs qui est évaluée par des méthodes de simulation ou par des méthodes analytiques .

Chapitre 3

Architecture des DSP TMS320C6x

3.1 Introduction

En 1982, Texas Instruments (TI) a introduit le TMS32010 - le premier DSP à point fixe de la famille TMS320. Avant la fin de l'année, le magazine Electronic Products a décerné au TMS32010 le titre de « Produit de l'année ». Aujourd'hui, la famille TMS320 se compose de plusieurs générations : DSP à virgule fixe (C1x, C2x, C2xx, C5x , C54x), DSP à virgule flottante (C3x et C4x), et DSP multiprocesseurs C8x

La plate-forme de processeur de signal numérique (DSP) TMS320C6000™ fait partie de la famille TMS320. Selon la Figure 3.1 La génération TMS320C62x et la génération TMS320C64x comprennent des périphériques à virgule fixe dans la plate-forme C6000, et la génération TMS320C67x comprend des périphériques à virgule flottante dans la plate-forme C6000. Les DSP TMS320C62x et TMS320C64x sont codecompatibles. Les DSP TMS320C62x et TMS320C67x sont compatibles avec le code. Les trois DSP utilisent l'architecture VelociTI, une architecture VLIW (Very Long Instruction Word) haute performance et avancée, faisant de ces DSP d'excellents choix pour les applications multicanaux et multifonctions.

Il existe maintenant une nouvelle génération de DSP, la génération TMS320C6x, avec des performances et des fonctionnalités qui reflètent l'engagement de Texas Instruments à être le leader mondial des solutions DSP.

3.2 Caractéristiques générales du C6000

Avec une performance allant jusqu'à 6000 millions d'instructions par seconde (MIPS) et un compilateur C efficace, les DSP TMS320C6x offrent aux architectes système des possibilités illimitées pour différencier leurs produits. Haute performance, facilité d'utilisation et prix abordable font de la génération TMS320C6x la solution idéale pour les applications multicanaux et multifonctions, telles que :

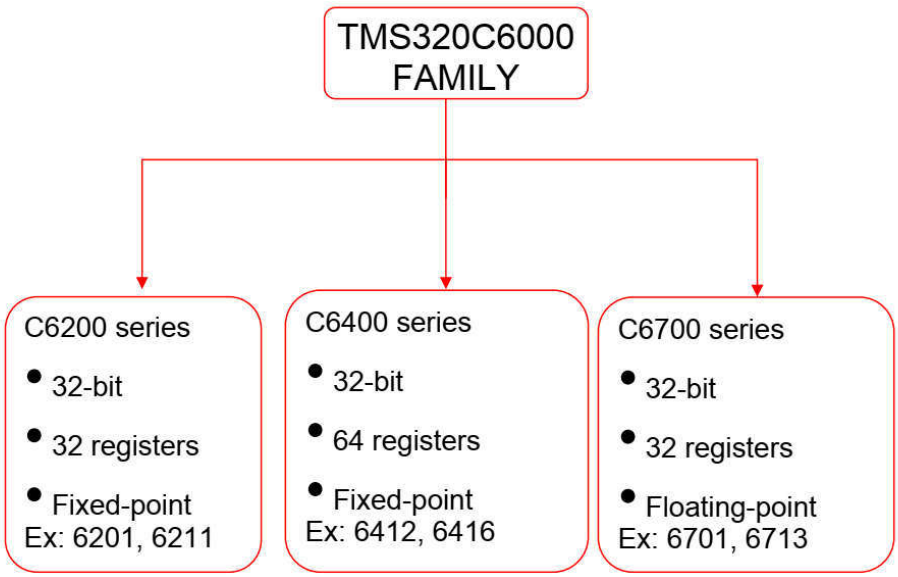


Fig. 3.1 : La famille TMS320C6000

o Modems groupés

o Stations de base en boucle locale sans fil

o Serveurs d'accès à distance

o Systèmes de boucle d'abonné numérique

o Modems câblés

o Systèmes de téléphonie multicanaux

Les appareils C6000 exécutent jusqu'à huit instructions 32 bits par cycle. Le processeur central du périphérique C62x / C67x se compose de 32 registres à usage général d'une longueur de mot de 32 bits et de huit unités fonctionnelles. Le processeur central C64x se compose de 64 registres 32 bits à usage général et de huit unités fonctionnelles. Ces huit unités fonctionnelles contiennent : • Deux multiplicateurs • Six ALU

Les caractéristiques des appareils C6000 comprennent : o CPU VLIW avancé avec huit unités fonctionnelles, y compris deux multiplicateurs et six unités arithmétiques

o Exécute jusqu'à huit instructions par cycle jusqu'à dix fois performances des DSP typiques

o Permet aux concepteurs de développer un code de type RISC très efficace pour un temps de développement rapide

o Emballage d'instructions

o Donne l'équivalence de taille de code pour huit instructions exécutées en série ou en parallèle

o Réduit la taille du code, les récupérations de programmes et la consommation d'énergie

o Exécution conditionnelle de toutes les instructions

o Réduit les ramifications coûteuses

o Augmente le parallélisme pour des performances plus durables

o Exécution efficace du code sur des unités fonctionnelles indépendantes

o Compilateur C le plus efficace du secteur sur la suite de référence DSP

o Premier optimiseur d'assemblage de l'industrie pour un développement rapide et une parallélisation améliorée

o Prise en charge des données 8/16/32 bits, offrant une prise en charge efficace de la mémoire pour une variété d'applications

o Les options arithmétiques 40 bits ajoutent une précision supplémentaire pour les vocodeurs et autres applications exigeantes en calcul

o La saturation et la normalisation prennent en charge les opérations arithmétiques clés

o La manipulation de champ et l'extraction d'instructions, la définition, l'effacement et le comptage de bits prennent en charge les opérations courantes trouvées dans les applications de contrôle et de manipulation de données.

Caractéristiques uniques du C6700: Le C67x possède ces fonctionnalités supplémentaires : o Prise en charge matérielle des opérations en virgule flottante IEEE simple précision (32 bits) et double précision (64 bits)

o Les entiers 32 x 32 bits se multiplient avec un résultat 32 ou 64 bits.

o Le DSP à virgule flottante TMS320C67x utilise toutes les instructions disponibles pour le TMS320C62x, mais il utilise également d'autres instructions spécifiques au C67x.

o Ces instructions spécifiques concernent les opérations de multiplication d'entiers 32 bits, de chargement de mots doubles et de virgule flottante, y compris l'addition, la soustraction et la multiplication.

Caractéristiques uniques du C6400: Le C64x possède ces fonctionnalités supplémentaires : o Chaque multiplicateur peut effectuer deux multiplications 16 x 16 bits ou quatre multiplications 8 x 8 bits à chaque cycle d'horloge.

o Extensions de jeu d'instructions quadruple 8 bits et double 16 bits avec prise en charge du flux de données

o Prise en charge des accès mémoire 32 bits (mot) et 64 bits (mot double) non alignés

o Des instructions spécifiques à la communication ont été ajoutées pour traiter les opérations courantes dans les codes de correction d'erreurs.

o Le comptage de bits et le matériel de rotation étendent la prise en charge des algorithmes au niveau des bits.

3.3 Architectures de processeur

L'un des plus gros goulots d'étranglement dans l'exécution des algorithmes DSP est le transfert d'informations vers et depuis la mémoire. Cela comprend des données, telles que des échantillons du signal d'entrée et des coefficients de filtre, ainsi que des instructions de programme, les codes binaires qui entrent dans le séquenceur de programme. Par exemple, supposons que nous devions multiplier deux nombres qui résident quelque part en mémoire. Pour ce faire, nous devons récupérer trois valeurs binaires de la mémoire, les nombres à multiplier, plus l'instruction du programme décrivant ce qu'il faut faire.

Architecture Von Neumann La figure 3.2 montre comment cette tâche apparemment simple est effectuée dans un microprocesseur traditionnel. C'est ce qu'on appelle souvent l'architecture Von Neumann, du nom du brillant mathématicien américain John Von Neumann (1903-1957). Von Neumann a guidé les mathématiques de nombreuses découvertes importantes du début du XXe siècle. Ses nombreuses réalisations comprennent : le développement du concept d'un ordinateur à programme stocké, la formalisation des mathématiques de la mécanique quantique et le travail sur la bombe atomique. Si c'était nouveau et excitant, Von Neumann était là !

Comme le montre la figure 3.2, l'architecture Von Neumann contient une seule mémoire et un seul bus pour transférer des données vers et depuis l'unité centrale (CPU). La

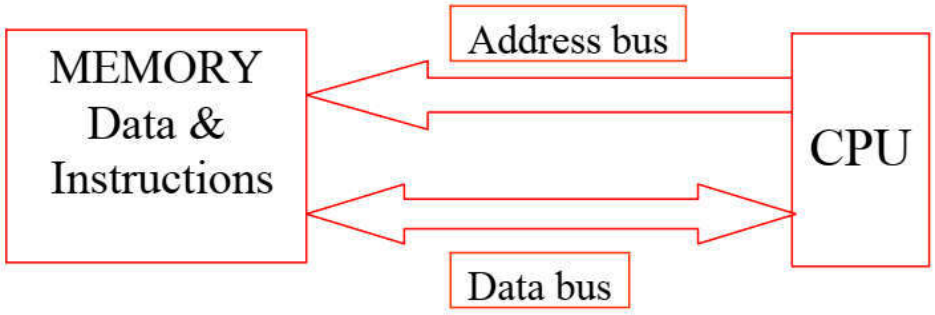


FIG. 3.2 : Architecture Von Neumann

multiplication de deux nombres nécessite au moins trois cycles d'horloge, un pour transférer chacun des trois nombres sur le bus de la mémoire vers le CPU. Nous ne comptons pas le temps de transfert du résultat vers la mémoire, car nous supposons qu'il reste dans le CPU pour une manipulation supplémentaire (comme la somme des produits dans un filtre FIR).

La conception de Von Neumann : est tout à fait satisfaisante lorsque vous vous contentez d'exécuter toutes les tâches requises en série. En fait, la plupart des ordinateurs actuels sont de conception Von Neumann. Nous n'avons besoin d'autres architectures que lorsqu'un traitement très rapide est nécessaire et nous sommes prêts à payer le prix d'une complexité accrue.

Architecture Harvard : Cela nous amène à l'architecture de Harvard, illustrée à la figure 3.3. Il porte le nom du travail effectué à l'Université Harvard dans les années 1940 sous la direction de Howard Aiken (1900-1973). Comme le montre cette illustration, Aiken a insisté sur des mémoires séparées pour les données et les instructions de programme, avec des bus séparés pour chacune. Puisque les bus fonctionnent indépendamment, les instructions de programme et les données peuvent être récupérées en même temps, améliorant la vitesse par rapport à la conception de bus unique. La plupart des DSP actuels utilisent cette architecture à quadruple bus.

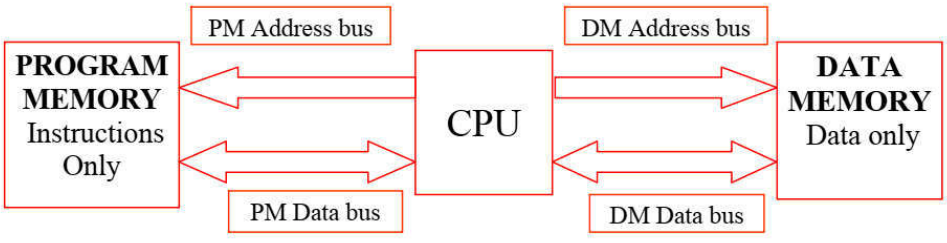


FIG. 3.3 : Architecture Harvard

PM : Program Memory

DM : Data Memory

Un handicap de la conception de base de Harvard est que le bus de mémoire de données est plus occupé que le bus de mémoire de programme. Lorsque deux nombres

sont multipliés, deux valeurs binaires (les nombres) doivent être passées sur le bus de mémoire de données, tandis qu'une seule valeur binaire (l'instruction de programme) est transmise sur le bus de mémoire de programme.

Architecture VLIW : VLIW signifie Very Long Instruction Word. Cette architecture a été introduite par Texas Instruments. Comme son nom l'indique, nous récupérons (fetching) une « instruction longue ». Cela signifie que dans le C6000, huit instructions (c'est certainement long !) Sont toujours récupérées à chaque cycle d'horloge. Ceci constitue un paquet d'extraction fetch packet.

Une architecture VLIW traditionnelle se compose de plusieurs unités d'exécution fonctionnant en parallèle, exécutant plusieurs instructions au cours d'un seul cycle d'horloge. Selon la figure 3.4. Ici, il y a trois ALU (c'est-à-dire des unités d'exécution) qui partagent le même programme et la même mémoire de données. Chaque ALU a sa propre adresse et son propre bus de données qui est indépendant de tous les autres bus.

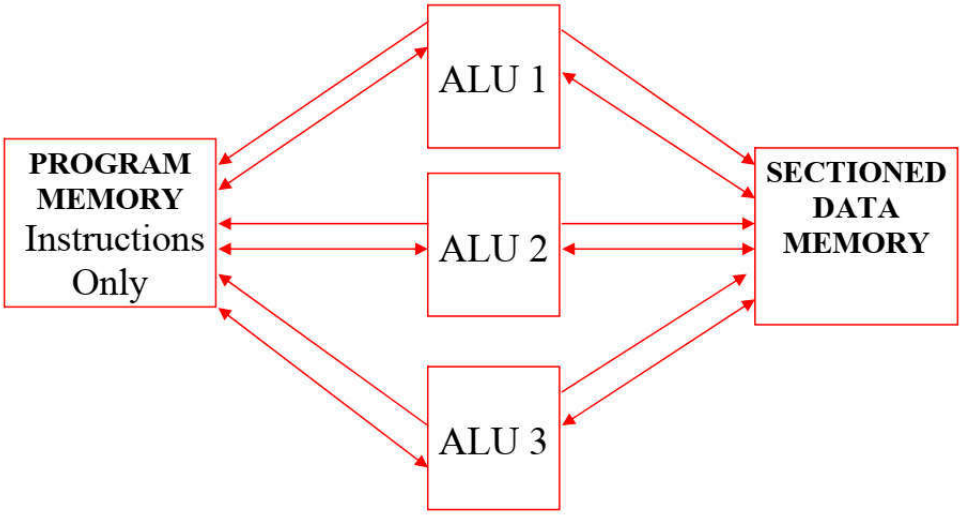


Fig. 3.4 : Architecture VILW

Une différence majeure entre VLIW et les architectures précédentes est la présence d'une mémoire de données sectionnée. La mémoire est divisée en différentes sections telles que stack, heap, const, text, var (variables), args (arguments), cio (I / O en langage C), etc. L'utilisateur peut explicitement décider du début et de la fin de chaque section mémoire, ce qui n'est pas possible dans les architectures précédentes.

Architecture VelociTI™ : L'architecture VelociTI de la plate-forme d'appareils C6000 en fait les premiers DSP prêts à l'emploi à utiliser le VLIW avancé pour atteindre des performances élevées grâce à un parallélisme accru au niveau des instructions.

VelociTI est une architecture hautement déterministe, ayant peu de restrictions sur la manière et le moment où les instructions sont extraites, exécutées ou stockées. Les fonctionnalités avancées de VelociTI comprennent :

o Emballage d'instructions : taille de code réduite

o Toutes les instructions peuvent fonctionner de manière conditionnelle : flexibilité du code

o Instructions à largeur variable : flexibilité des types de données

o Branches entièrement pipelinées

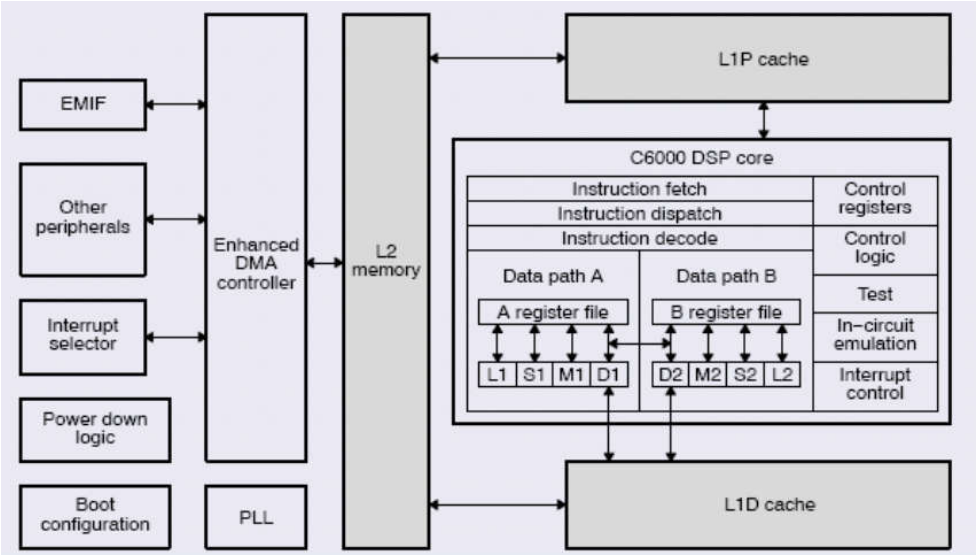


Fig. 3.5 : Schéma fonctionnel du C6000

Les autres périphériques embarqués incluent :

(a) Contrôleur EDMA (Enhanced Direct Memory Access)

(b) Interface de mémoire externe (EMIF)

(c) Sélecteur d'interruption

(d) Logique de mise hors tension

(e) Contrôleur de boucle à verrouillage de phase (PLL)

3.4 Processeur de signal numérique TMS320C6713

Le DSP TMS320C6713 est un processeur à virgule flottante basé sur l'architecture à rendement élevé et avancé du VLIW 'très-long-instruction-mot' (Very Long Instruction Word) développée par Texas Instruments (TI). Ce DSP est un excellent choix pour les applications multicanaux et multifonctionnelle.

Grace à une fréquence de fonctionnement de 225MHz, le C6713 peut exécuter deux opérations MAC (multiply and accumulate) en un cycle, totalisant ainsi 450 millions de MACs par seconde. Par l'utilisation de six parmi les huit unités fonctionnelles, il est possible d'atteindre 1350 MFLOPS (million floating-point operations per second). Pour une fréquence de 225MHz, ceci est équivalent à 1800 MIPS (million instructions per second) avec un cycle d'instruction de 4.44 ns.

La famille TMS 320c67xx a un grand espace d'adressage. Le programme et les données peuvent être placés n'importe où dans l'espace d'adressage unifié. La mémoire interne

comprend une architecture de cache à deux niveaux avec 8 KB de niveau 1 : 4KB pour le programme (L1P) et 4 KB pour les données (L1D), et 256 KB de mémoire de niveau 2 partagé entre le programme et les données. Le DSP dispose d'une interface directe pour les mémoires synchrones (SDRAM et SBSRAM) et les mémoires asynchrones (SRAM et EPROM). La mémoire synchrone nécessite une horloge, mais offre un compromis entre la SRAM statique et la DRAM dynamique, ou la SRAM est plus rapide mais coûte plus cher que la DRAM.

Les bus internes comprennent un bus d'adresses de programme à 32 bits, un bus de données programme 256 bits pour supporter huit instructions de 32 bits chacune, deux bus de données d'adresses 32 bits, deux bus de données 64 bits, et deux bus de 64 bits pour le stockage de données. Un bus d'adresse de 32 bits permet l'adressage d'un espace mémoire total est de $2^{32} = 4$ Go, dont quatre pages extérieures de mémoire : CE0, CE1, CE2 et CE3. La figure 3.6 montre un schéma fonctionnel du DSP C6713 intégrant la CPU, les périphériques et les interfaces externes.

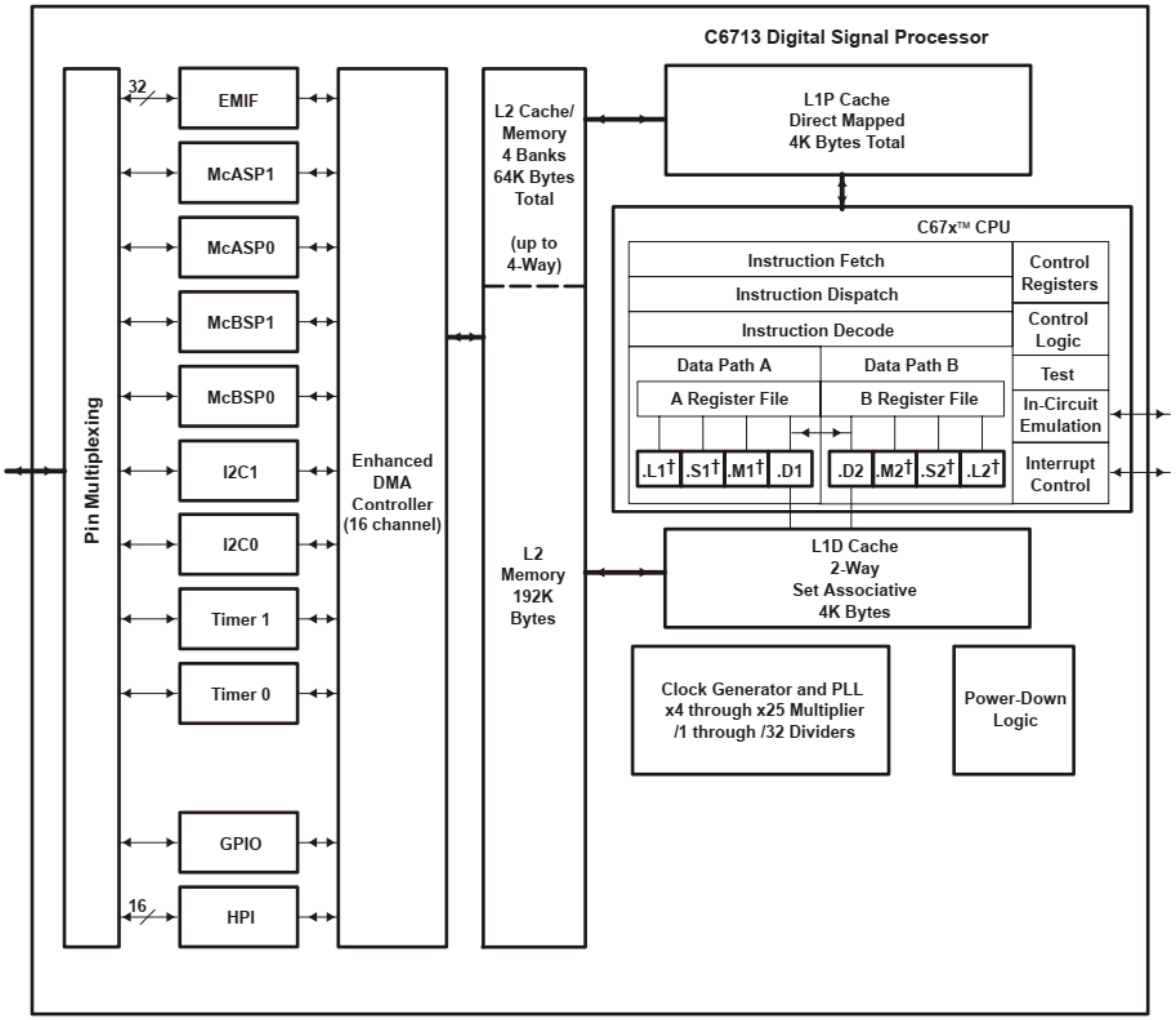


FIG. 3.6 : diagramme bloc fonctionnel du TMS320C6713

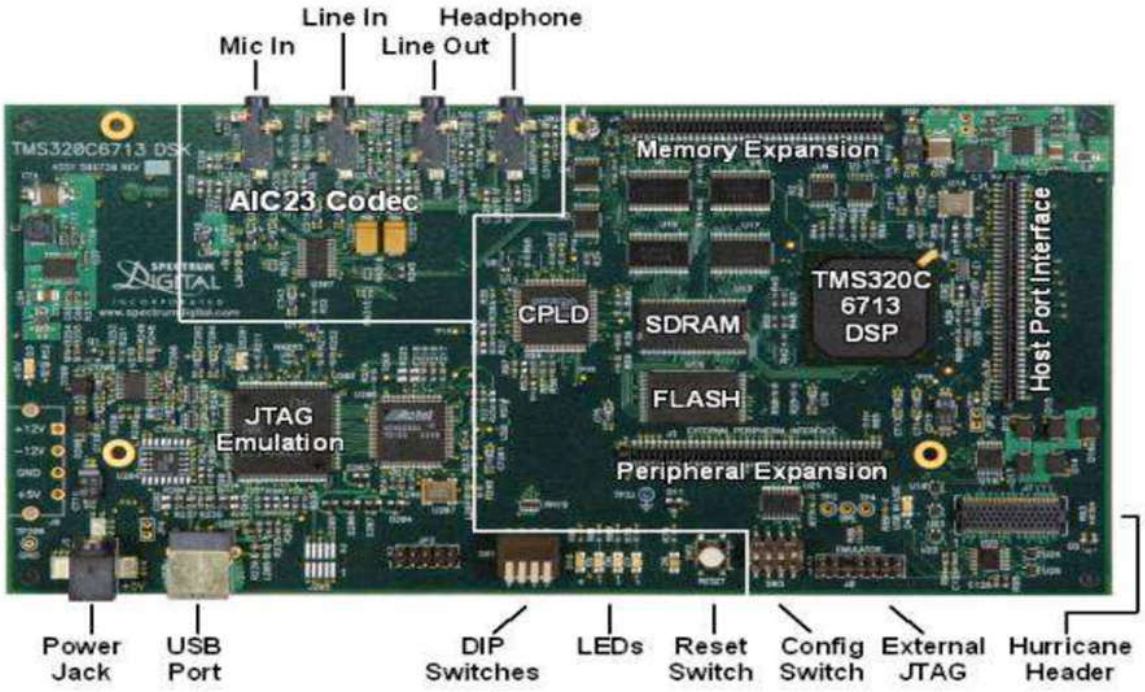


FIG. 3.7 : DSP Starter Kit TMS320C6713

3.5 Cartographie de mémoire

Les banques de mémoire indépendantes sur la C6x permettent deux accès de mémoire dans un cycle d'instruction. Deux banques de mémoire indépendantes sont accessibles via deux bus indépendants. Puisque la mémoire interne est organisée en banques de mémoire, deux chargements ou deux stockages d'instructions peuvent être exécutées en parallèle. Aucun conflit ne se produira si les données accédées sont dans différentes banques de mémoire. Les bus séparés pour le programme, les données et l'accès direct à la mémoire (DMA) permettent au processeur C6x d'effectuer en parallèle, les opérations « program fetch », lecture et écriture de données, et DMA. Avec les données et les instructions se trouvant dans des espaces mémoire séparés, l'accès aux mémoires simultanément est possible.

Le TMS320C6713 basé sur l'architecture de Havard modifiée utilise une mémoire externe et une mémoire interne .

- La mémoire externe occupe les espaces CEO, CE1, CE2, CE3.

- La mémoire interne a une taille de 260 KB qui est décomposée en deux niveaux :

Le niveau (L1) est constitué de deux mémoires caches de 4 KB chacune, (L1P) qui est utilisée pour les programmes et (L1D) qui est utilisée pour les données.

Le Niveau (L2) est composé de 256 KB de mémoire partagée entre mémoire des données et mémoire de programmes.

Le C6713 sur le DSP Starter Kit comprend 264KB de mémoire interne, qui commence à 0x00000000, et 16 Mo de SDRAM externe, repérés à travers CE0 à partir de 0x80000000. Le DSK comprend également 512 Ko de mémoire flash (256 kB facilement disponibles

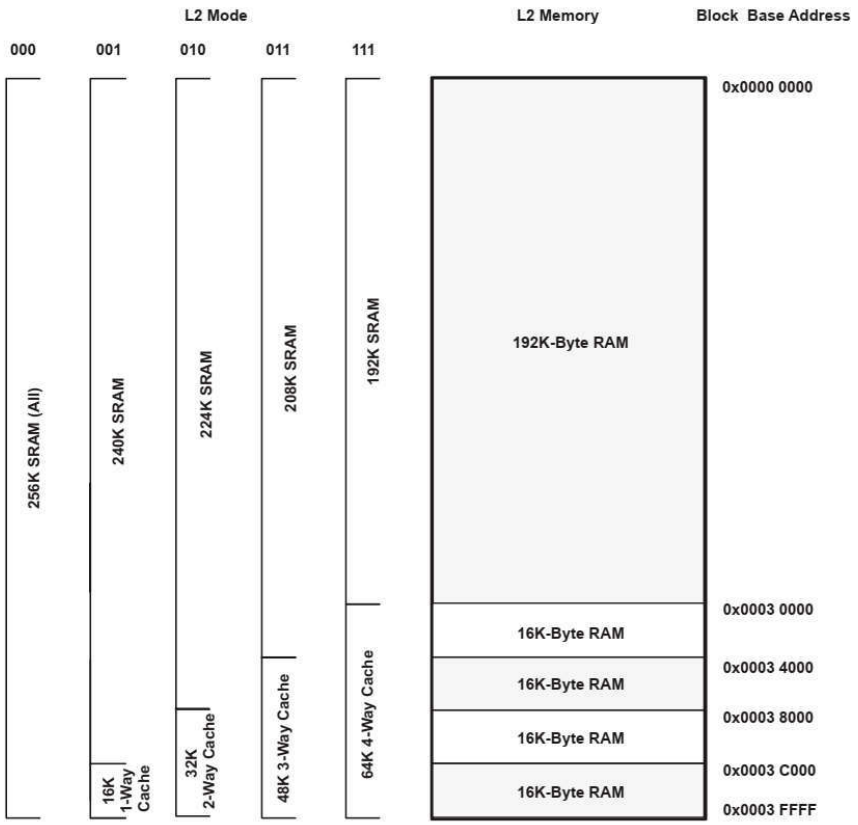


FIG. 3.8 : Configuration de la mémoire interne du niveau 2 (L2)

pour l'utilisateur), tracée à travers CE1 à partir de 0x90000000. La figure 3.8 montre la configuration de la mémoire interne du niveau 2. Le Tableau 3.1 illustre l'organisation de la mémoire.

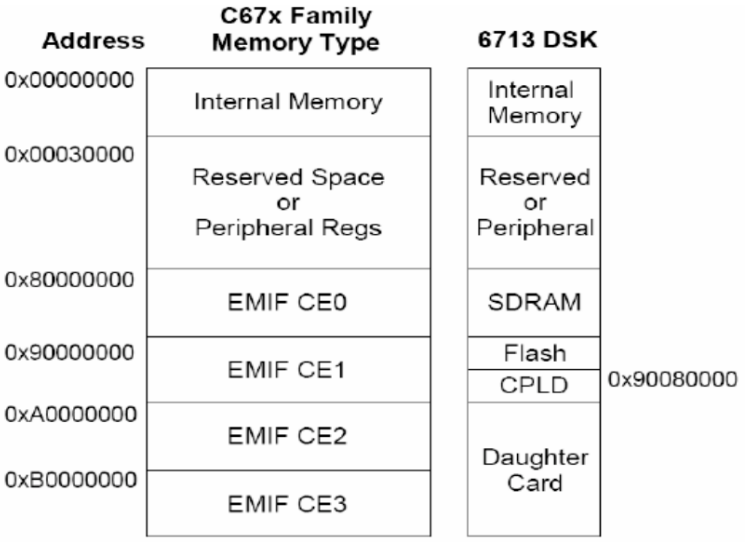


FIG. 3.9 : Cartographie de la mémoire du DSK C6713

3.6 L'unité centrale de traitement CPU

Elle est commune pour C62x/C64x/C67x . Elle est constituée des éléments suivants :

A. Unité de contrôle de programme :

1- Unité "fetch" programme : Elle a pour rôle récupérer les programmes (Unité d'acquisition du programme). Cette opération se déroule en quatre phases :

- Phase PG : l'adresse du code est générée.
- Phase PS : l'adresse est envoyée à la mémoire.
- Phase PW : l'attente de lecture du code de la mémoire.
- Phases PR : la lecture du code.

2 - Unité "dispatch" de l'instruction : le code récupéré de la mémoire est affecté à 1'unité fonctionnelle associée.

3 - Unité de "décodage de l'instruction" : elle a pour rôle de décoder l'instruction.

B. Unités fonctionnelles :

Le CPU comporte huit unités fonctionnelles indépendantes subdivisées en deux : A et B, comme montré sur la figure 3.6.

- Chaque chemin possède une unité pour la multiplication (**.M**),
- Une unité pour les opérations arithmétiques et logiques (**.L**),
- Une unité pour les sauts, la manipulation des bits et les opérations arithmétiques (**.S**) ;
- Enfin l'unité de chargement et stockage et des opérations arithmétiques (**.D**).

Les unités (.S) et (.L) sont utilisées pour les opérations arithmétiques et logiques et pour les instructions de branchement.

Tous les transferts de données passent par l'unité (.D). les opérations arithmétiques telle que l'addition (ADD) et la soustraction (SUB) peuvent être exécutées sur les unités (.S, .L et .D).

Quand le CPU exécute MPY.M1 A0, A1, A3

l'unité fonctionnelle .M1 prend les valeurs stockées dans A0 et A1, multiplier ensemble et stoker le résultat dans A3. .M1 dans MPY .M1 A0, A1, A3 indique que l'opération se déroule dans l'unité .M1. .L'unité M1 à un multiplieur 16 bits.

Les huit unités fonctionnelles comportent quatre ALUs pour les opérations à virgule fixe et à virgule flottante (.L et .S), deux ALUs pour les opérations à virgule fixe (.D) ; et

deux ALUs pour la multiplication à virgule fixe ou virgule flottante (.M).

Chaque unité peut accéder directement aux registres de son chemin. Dans chaque chemin, se trouve un ensemble de 16 registres à 32-Bits : A0-A15 et B0-B15. Les unités (.L1, .S1, .M1 et .D1) utilisent les registres A ; et les autres unités utilisent les registre B.

Deux chemins croisés (1x et 2x) (figure 3.10) permettent aux unités d'un chemin donné d'accéder aux registres du chemin opposé. On ne peut pas avoir plus de deux chemins croisés par cycle. Le DSP comporte 32 registres à utilisation générale ; cependant quelques registre sont réservés pour un adressage spécifique et d'autres sont utilisés pour les instructions conditionnelles.

C. Chemin de données ou Data Path

La figure 3.10 montre les deux chemins de données « data paths » du c67 :

- Les deux ensembles de registres (A et B).
- Les huit unités (L1, L2, M1, M2, S1, S2, D1, et D2).
- Les deux voies de chargement depuis la mémoire LD1 et LD2 (LD=load).
- Les deux voies de stockage vers la mémoire ST1 et ST2 (ST=store).
- Deux croisements (registre file cross paths) entre les ensembles des registre 1X et 2X.
- Deux chemins d'adresse de données (data adresse paths) DA1 etDA2.

3.7 Paquets d'exécution et de fetch

L'architecture VELOCITI, introduite par Texas Instruments, est dérivée de l'architecture VLIW.

Un **paquet d'exécution (EP)** comporte un ensemble d'instructions qui peuvent être exécutées en parallèle pendant le même cycle. Le nombre des EPs pendant un **paquet fetch (FP)** peut varier de 1 (avec 8 instructions en parallèle) jusqu'à 8 (pas d'instructions en parallèle). L'architecture VLIW est modifiée pour permettre un plus grand nombre de EP dans un FP.

Le bit le moins signifiant de chaque instruction à 32-Bits est utilisé pour déterminer si l'instruction suivante fait partie du même EP.

Supposons un FP avec trois EPs : EP1, contenant deux instructions en parallèle ; et EP2 et EP3, chacun contenant trois instructions en parallèle, comme suit :

Instruction A

|| Instruction B

Instruction C

|| Instruction D

|| Instruction E

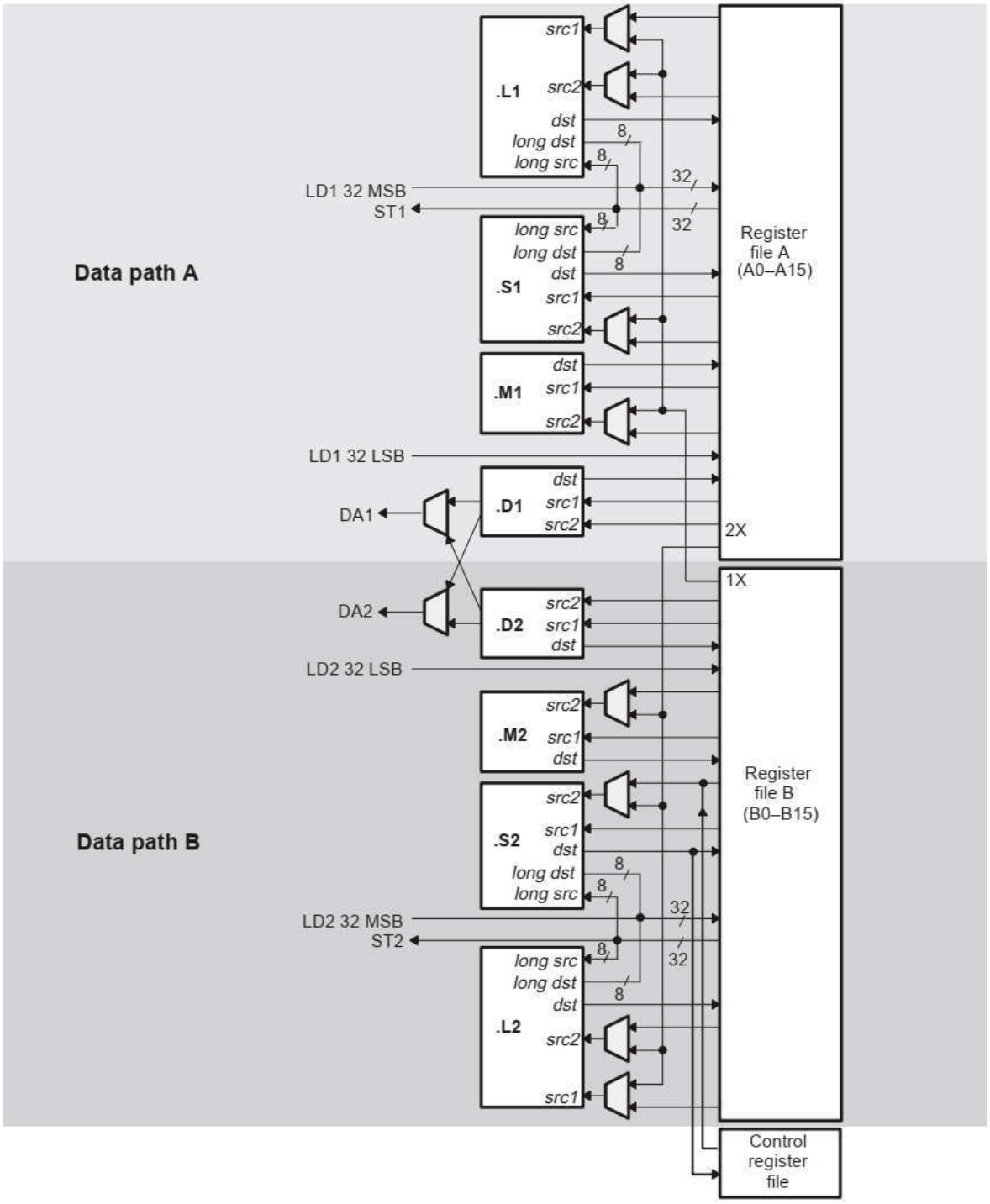


FIG. 3.10 : chemins de données CPU du TMS320C6713

Instruction F

|| Instruction G

|| Instruction H

La figure 3.12 montre un exemple. Le Bit 0 (LSB) de chaque instruction signale si cette instruction doit être exécutée en parallèle avec l'instruction suivante. Le bit 0 de l'instruction B est à zero, ce qui signifie que l'instruction B n'est pas exécutée pendant

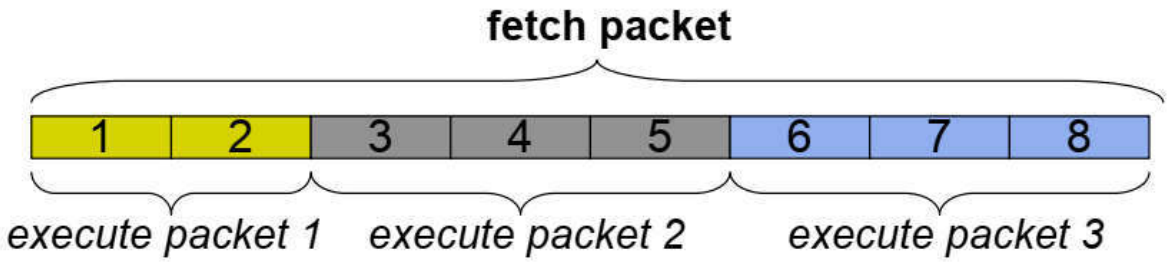


FIG. 3.11 : Paquets d'exécution et de fetch

le même EP de l'instruction adjacente C. similairement, l'instruction E n'est pas dans le même EP que l'instruction F.

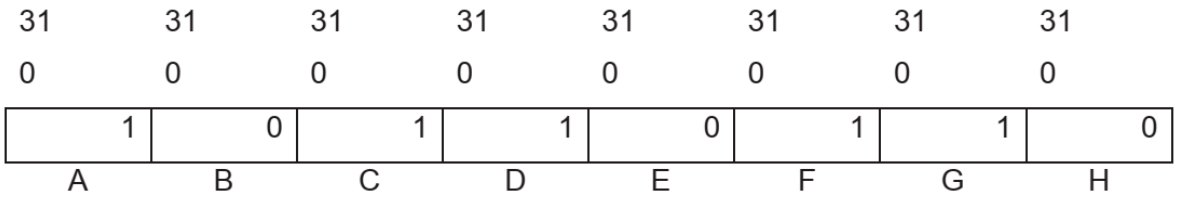


FIG. 3.12 : Un FP avec trois EPs montrant le bit LSB de chaque instruction

3.8 Architecture pipeline

Les phases du pipeline sont divisées en trois étapes :

- Fetch (Récuperer)
- Decode(Décoder)
- Execute (Exécuter)

Toutes les instructions du jeu d'instructions C67x passent par les étapes de fetch, de décodage et d'exécution du pipeline. L'étape fetch du pipeline a quatre phases pour toutes les instructions, et l'étape de décodage a deux phases pour toutes les instructions. L'étape d'exécution du pipeline nécessite un nombre variable de phases, en fonction du type d'instruction. Les étapes du pipeline C67x sont illustrées à la Figure 3.13.

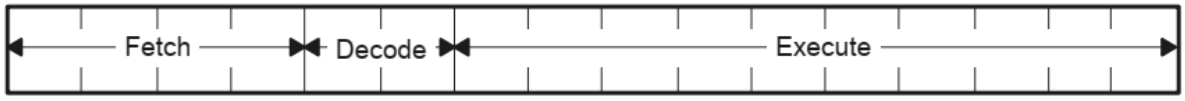


FIG. 3.13 : Étapes de pipeline à virgule flottante

3.8.1 Fetch

Les phases de Fetch du pipeline sont : • PG : génération d'adresse de programme (Program address generate) • PS : envoi d'adresse de programme (Program address

scnd) • PW : Attcntc dc prêt d'accès au programmc (Program acccss rcady wait) • PR : réception du paquet de récupération du programme (Program fetch packet receive)

Le C67x utilise un paquet fetch (FP) de huit instructions. Les huit instructions passent par un traitement fetch ensemble, par les phases PG, PS, PW et PR. La Figure 3.14 (a) montre les phases de fetch dans un ordre séquentiel de gauche à droite. La Figure 3.14 (b) montre un diagramme fonctionnel du flux d'instructions à travers les phases fetch. Pendant la phase PG, l'adresse du programme est générée dans la CPU. Dans la phase PS, l'adresse du programme est envoyée en mémoire. Dans la phase PW, une lecture de la mémoire se produit.

Enfin, dans la phase PR, le paquet fetch est reçu au niveau de la CPU. La Figure 3.14 (c) montre des paquets fetch traversant les phases de l'étape de fetch du pipeline. Dans la Figure 3.14 (c), le premier paquet fetch (en PR) est composé de quatre paquets d'exécution, et les deuxième et troisième paquets fetch (en PW et PS) contiennent chacun deux paquets d'exécution. Le dernier paquet fetch (dans PG) contient un seul paquet d'exécution de huit instructions à cycle unique.

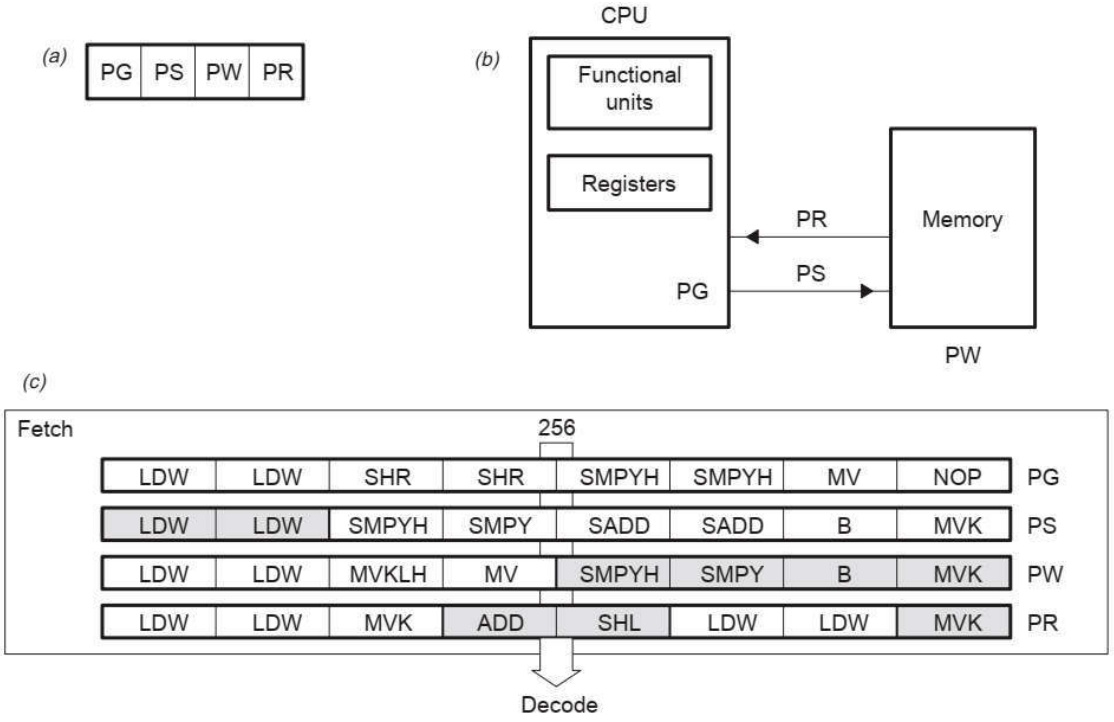


Fig. 3.14 : les phases Fetch du pipeline

3.8.2 Decode

Les phases de décodage du pipeline sont :

- DP : Instruction dispatch (envoi d'instructions)
- DC : Instruction decode (décodage d'instructions)

Dans la phase DP du pipeline, les paquets fetch sont divisés en paquets d'exécution. Les paquets d'exécution se composent d'une instruction ou de deux à huit instructions

parallèles. Pendant la phase DP, les instructions d'un paquet fetch sont affectées aux unités fonctionnelles appropriées. Dans la phase DC, les registres source, les registres de destination et les chemins associés sont décodés pour l'exécution des instructions dans les unités fonctionnelles.

La Figure 3.15 (a) montre les phases de décodage dans un ordre séquentiel de gauche à droite. La Figure 3.15 (b) montre un paquet fetch contenant deux paquets d'exécution au fur et à mesure de leur traitement par l'étape de décodage du pipeline. Les six dernières instructions du paquet fetch (FP) sont parallèles et forment un paquet d'exécution (EP). Cet EP est en phase d'expédition (DP) de l'étape de décodage. Les flèches indiquent l'unité fonctionnelle attribuée à chaque instruction pour exécution au cours du même cycle. L'instruction NOP dans le huitième créneau du FP n'est pas envoyée à une unité fonctionnelle car aucune exécution ne lui est associée.

Les deux premiers créneaux du paquet d'extraction (grisés ci-dessous) représentent un paquet d'exécution de deux instructions parallèles qui ont été distribuées lors du cycle précédent. Ce paquet d'exécution contient deux instructions MPY qui sont maintenant en décodage (DC) un cycle avant l'exécution. Il n'y a pas d'instructions décodées pour les unités fonctionnelles .L, .S et .D pour la situation illustrée.

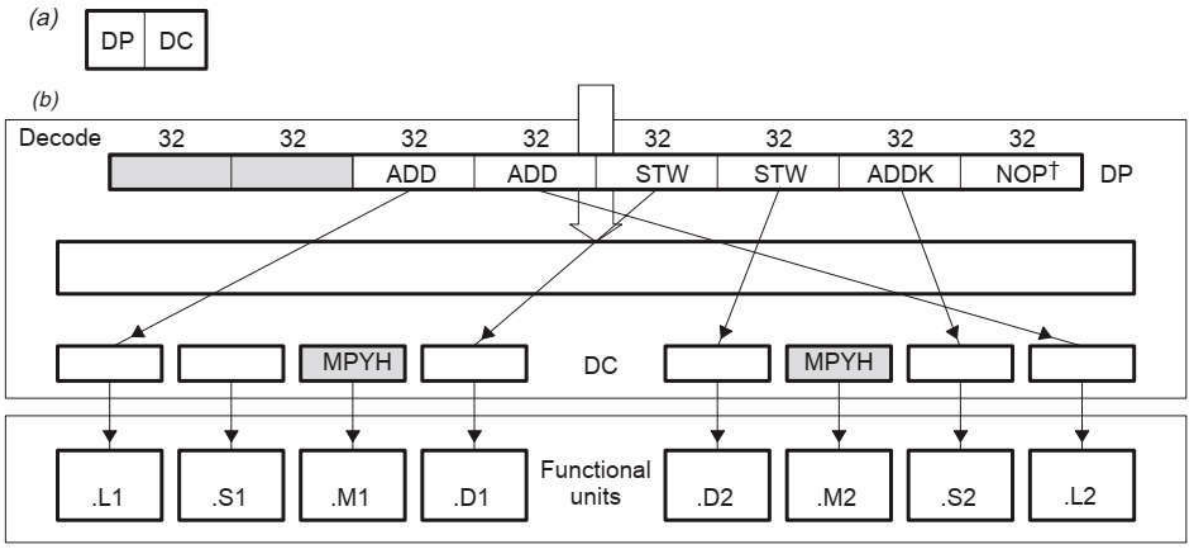


Fig. 3.15 : les phases de décodage du pipeline

3.8.3 Execute

La partie d'exécution du pipeline à virgule flottante est subdivisée en dix phases (E1-E10), par rapport aux cinq phases du pipeline à virgule fixe. Différents types d'instructions nécessitent différents nombres de ces phases pour terminer leur exécution. Ces phases du pipeline jouent un rôle important dans votre compréhension de l'état de l'appareil aux limites du cycle du processeur. La Figure 3.16 (a) montre les phases d'exécution du pipeline dans un ordre séquentiel de gauche à droite. La Figure 3.16 (b) montre la partie du diagramme fonctionnel dans laquelle l'exécution a lieu.

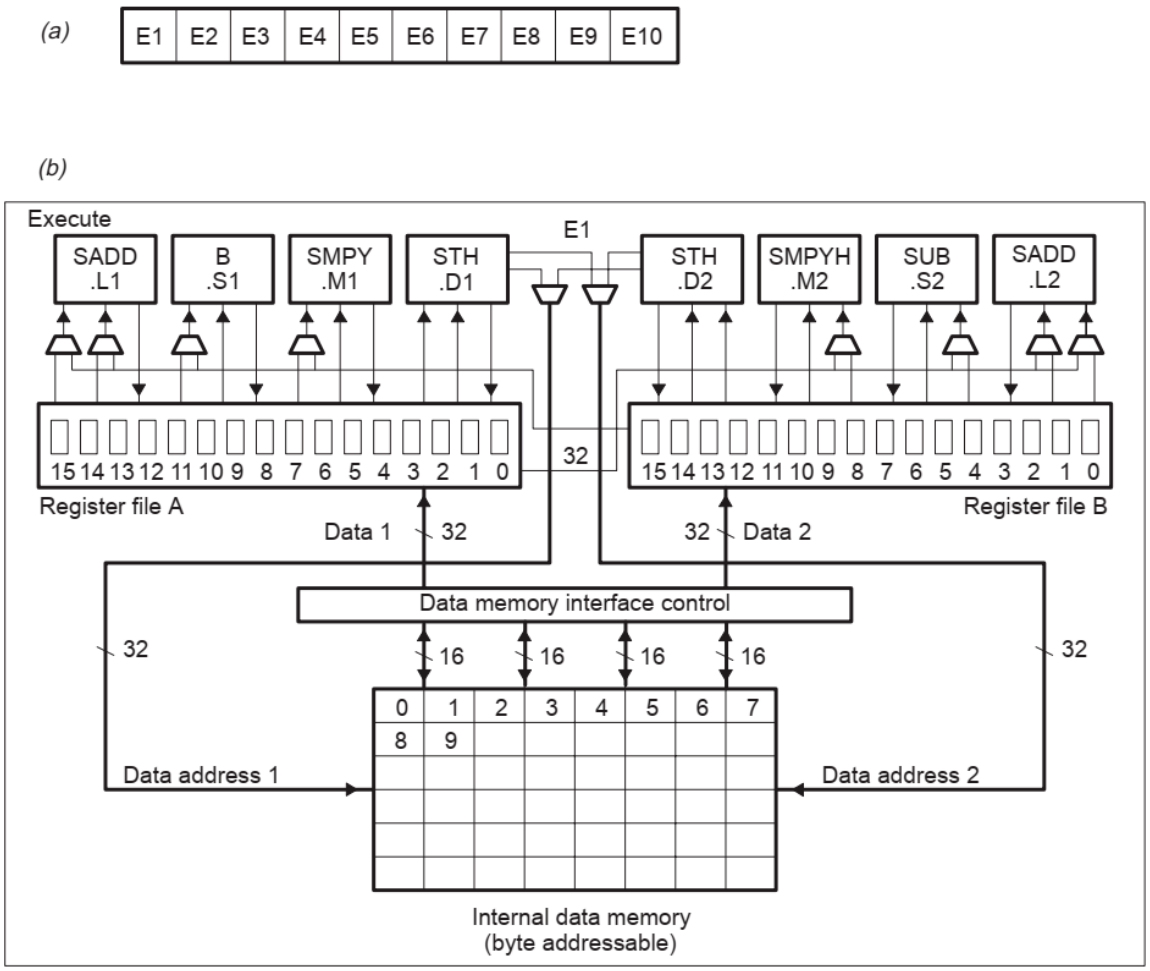


FIG. 3.16 : les phases d'exécution du pipeline et le diagramme fonctionnel du TMS320C67x

3.8.4 Résumé de l'exploitation du pipeline

La Figure 3.17 montre toutes les phases de chaque étage du pipeline C67x dans un ordre séquentiel, de gauche à droite.

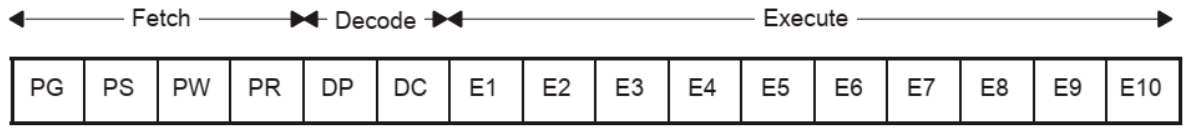


FIG. 3.17 : Phases de pipeline à virgule flottante

La Figure 3.18 montre un exemple du flux de pipeline de paquets d'extraction consécutifs contenant huit instructions parallèles. Dans ce cas, où le pipeline est plein, toutes les instructions d'un paquet d'extraction sont en parallèle et divisées en un seul paquet d'exécution par paquet d'extraction. Les paquets d'extraction circulent de manière séquentielle à travers chaque phase du pipeline.

Par exemple, examinez le cycle 7 dans la figure 3.18. Lorsque les instructions de FP n atteignent E1, les instructions du paquet d'exécution de FP n +1 sont en cours de

décodage. FP n + 2 est en répartition tandis que les FP n + 3, n + 4, n + 5 et n + 6 sont chacun dans l'une des quatre phases de recherche de programme.

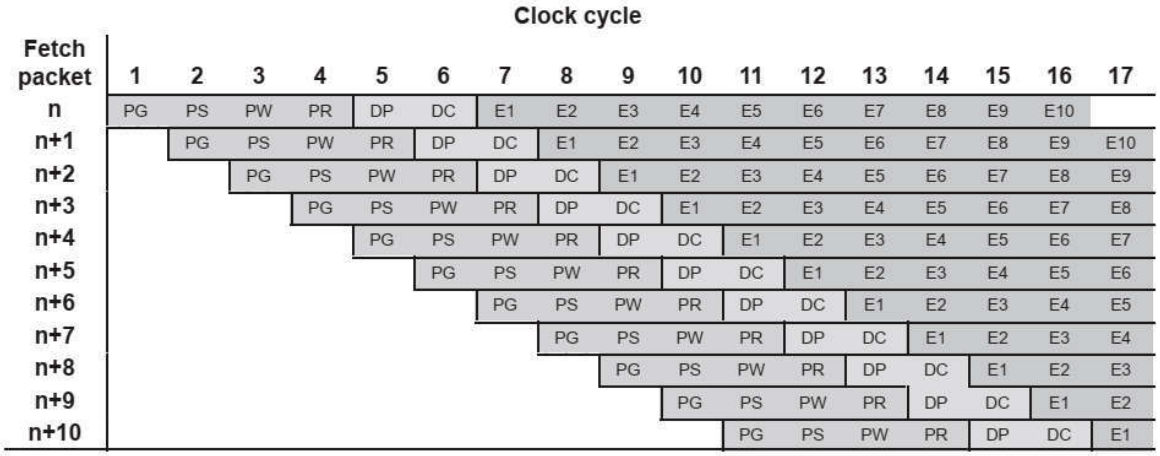


FIG. 3.18 : Opération de pipeline : un paquet d'exécution par paquet fetch

3.8.5 Considérations relatives aux performances

Le pipeline C67x est plus efficace lorsqu'il est maintenu aussi plein que les algorithmes du programme le permettent. Il est utile de considérer certaines situations qui peuvent affecter les performances du pipeline.

Un paquet fetch (FP) est un groupement de huit instructions. Chaque FP peut être divisé en un à huit paquets d'exécution (EP). Chaque EP contient des instructions qui s'exécutent en parallèle. Chaque instruction s'exécute dans une unité fonctionnelle indépendante. L'effet sur le pipeline de combinaisons d'EP qui incluent un nombre variable d'instructions parallèles, ou juste une seule instruction qui s'exécute en série avec un autre code, est considéré ici.

En général, le nombre de paquets d'exécution dans un seul FP définit le flux d'instructions à travers le pipeline. Un autre facteur déterminant est les types d'instructions dans l'EP. Chaque type d'instruction a un nombre fixe de cycles d'exécution qui détermine le moment où les opérations de cette instruction sont terminées.

Enfin, l'effet du système de mémoire sur le fonctionnement du pipeline est considéré. L'accès au programme et à la mémoire de données est discuté, ainsi que les blocages de mémoire (stall memory).

3.8.5.1 Fonctionnement de pipeline avec plusieurs paquets d'exécution dans un paquet fetch

En se référant à nouveau à la figure 3.18, le fonctionnement du pipeline est représenté avec huit instructions dans chaque paquet d'extraction. La figure 3.19, cependant, montre l'opération de pipeline avec un paquet d'extraction qui contient plusieurs paquets d'exécution. Le code de la figure 3.19 peut avoir cette disposition :

instruction A ; EP k FP n

|| instruction B ;

instruction C ; EP k + 1 FP n

|| instruction D

|| instruction E

instruction F ; EP k + 2 FP n

|| instruction G

|| instruction H

instruction I ; EP k + 3 FP n + 1

|| instruction J

|| instruction K

|| instruction L

|| instruction M

|| instruction N

|| instruction O

|| instruction P

... en continuant avec les EP k + 4 à k + 8, qui ont huit instructions en parallèle, comme k + 3.

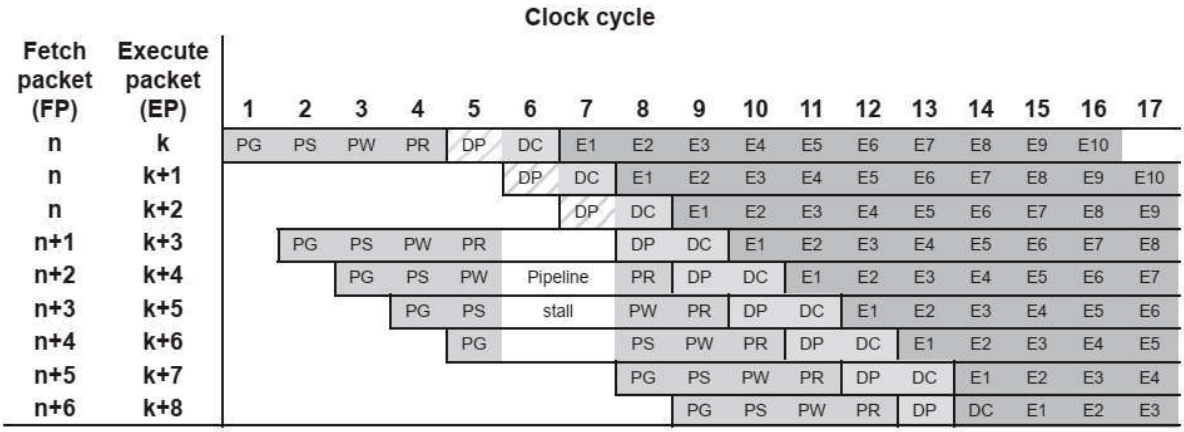


FIG. 3.19 : Fonctionnement du pipeline : Fetch paquets avec différents nombres de paquets d'exécution

Sur la figure 3.19, le paquet fetch n, qui contient trois paquets d'exécution, est montré suivi de six paquets fetch (n + 1 à n + 6), chacun avec un paquet d'exécution (contenant huit instructions parallèles). Le premier paquet fetch (n) passe par les phases de fetching du programme pendant les cycles 1 à 4. Au cours de ces cycles, une phase fetch de programme est lancée pour chacun des paquets fetch qui suivent.

Au cycle 5, la phase de dispatching de programme (DP), l'unité centrale scanne les p bits et détecte qu'il y a trois paquets d'exécution (k à k + 2) dans le paquet fetch n. Cela

force le pipeline à caler (stall), ce qui permet à la phase DP de démarrer pour exécuter les paquets k + 1 et k + 2 dans les cycles 6 et 7. Une fois que le paquet d'exécution k + 2 est prêt à passer à la phase DC (cycle 8), le décrochage du pipeline (stall pipeline) est libéré.

Les paquets fetch n + 1 à n + 4 étaient tous bloqués afin que l'unité centrale puisse avoir le temps d'exécuter la phase DP pour chacun des trois paquets d'exécution (k à k + 2) dans le paquet fetch n. Le paquet d'extraction n + 5 a également été bloqué dans les cycles 6 et 7: il n'était pas autorisé à entrer dans la phase PG jusqu'à ce que le blocage du pipeline ait été libéré au cycle 8. Le pipeline continue de fonctionner comme indiqué avec les paquets fetch n + 5 et n + 6 jusqu'à ce qu'un autre paquet fetch contenant plusieurs paquets d'exécution entre dans la phase DP, ou qu'une interruption se produise.

3.8.5.2 NOP multicycle

L'instruction NOP a un opérande optionnel, count, qui vous permet d'émettre une seule instruction pour les NOP multicycles. Un NOP 2, par exemple, remplit des intervalles de retard supplémentaires pour les instructions dans son paquet d'exécution et pour tous les paquets d'exécution précédents. Si un NOP 2 est en parallèle avec une instruction MPY, les résultats du MPY seront disponibles pour une utilisation par les instructions dans le prochain paquet d'exécution.

La figure 3.20 montre comment un NOP multicycle peut piloter l'exécution d'autres instructions dans le même paquet d'exécution. La figure 3.20 (a) montre un NOP dans un paquet d'exécution (en parallèle) avec un autre code. Les résultats des LD, ADD et MPY seront tous disponibles pendant le cycle approprié pour chaque instruction. Par conséquent, NOP n'a aucun effet sur le paquet d'exécution.

La figure 3.20 (b) montre le remplacement d'un NOP à cycle unique par un NOP à plusieurs cycles (NOP 5) dans le même paquet d'exécution. Le NOP 5 n'effectuera aucune opération autre que les opérations à partir des instructions à l'intérieur de son paquet d'exécution. Les résultats des LD, ADD et MPY ne peuvent être utilisés par aucune autre instruction tant que la période NOP 5 n'est pas terminée.

3.9 Les registres

Deux ensemble de registres, de 16 registres chacun, sont disponibles : les registre A (A0 à A15) et les registres B (B0 à B15). Les Registres A0, A1, B0, B1, et B2 sont utilisés comme registres conditionnels. Les registres A4 à A7 et B4 à B7 sont utilisés pour l'adressage circulaire. Les registres A0 à A9 et B0 à B9 (à l'exception de B3) sont des registres temporaires. N'importe quel registre des registres (A10 à A15) et (B10 par B15) utilisé est sauvegardé puis restauré avant le retour d'un sousprogramme.

Une donnée à 40 bits peut être représentée par deux registres de 32 bits. Les 32 bits de poids faible (LSB) sont stockés dans le registre pair (par exemple A2), et les 8 bits restants sont stockés dans les 8 bits de poids faible du registre impair supérieur adjacent (A3). Le même système est utilisé pour stocker une valeur à double précision de 64 bits.

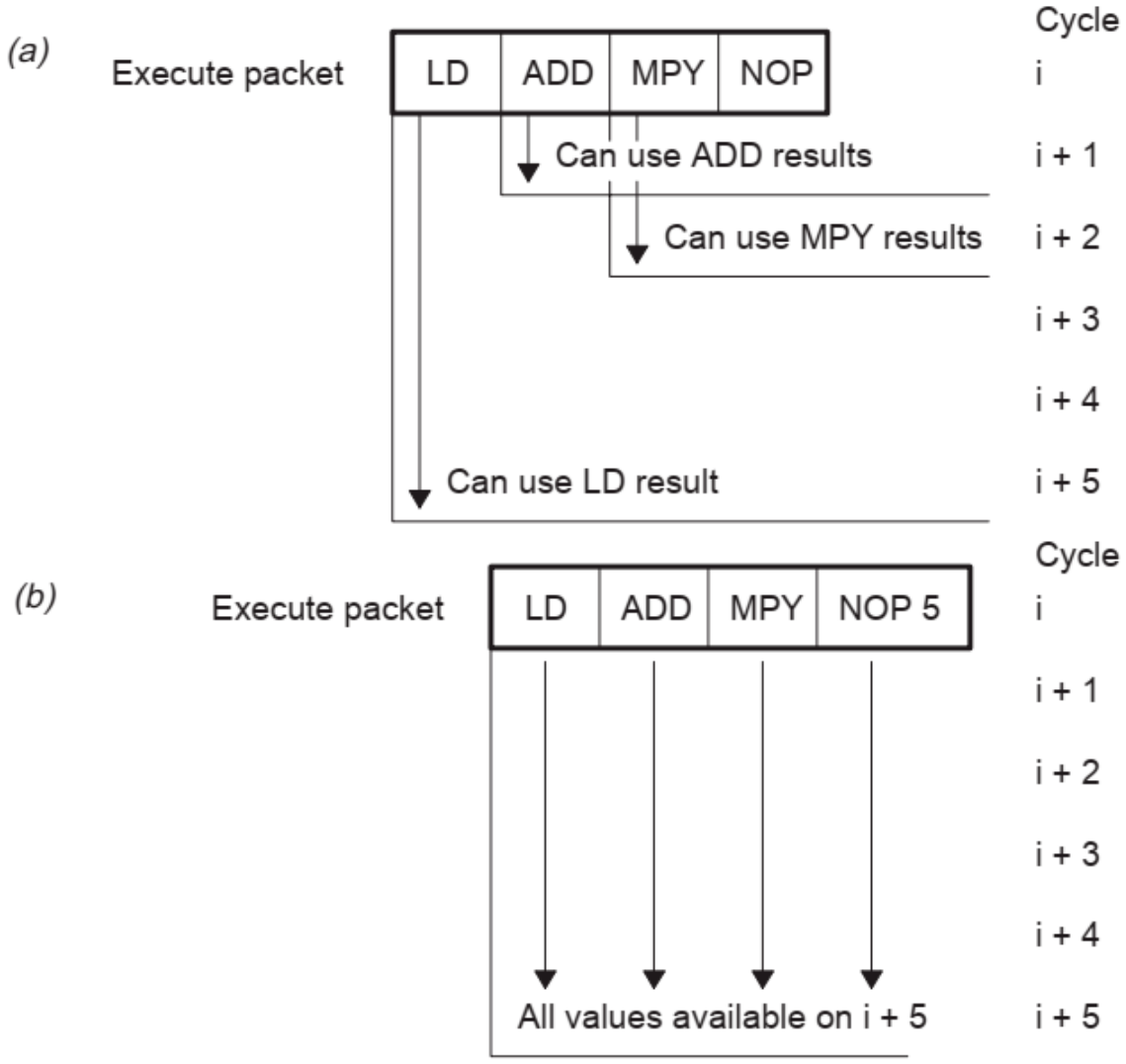


FIG. 3.20 : Multicycle NOP dans un Execute Packet

Ces 32 registres sont considérés comme des registres à usage général. Plusieurs registres à usage spécial sont également disponibles pour le contrôle et les interruptions : par exemple, le registre de mode d'adressage (AMR) qui est utilisé pour l'adressage circulaire et le contrôle des interruptions.

3.10 Les registres de contrôle

La figure ci-dessous représente les registres de contrôle de la famille TMS320c6000 et leur rôle.

Les registres additionnels suivants sont ceux de la sous familleTMS320c67xx

Register		
Abbreviation	Name	Description
AMR	Addressing mode register	Specifies whether to use linear or circular addressing for each of eight registers; also contains sizes for circular addressing
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits
IFR	Interrupt flag register	Displays status of interrupts
ISR	Interrupt set register	Allows you to set pending interrupts manually
ICR	Interrupt clear register	Allows you to clear pending interrupts manually
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt
PCE1	Program counter, E1 phase	Contains the address of the fetch packet that contains the execute packet in the E1 pipeline stage

FIG. 3.21 : Registres de contrôles des TMS320c6000

Register		
Abbreviation	Name	Description
FADCR	Floating-point adder configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .L unit.
FAUCR	Floating-point auxiliary configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .S unit.
FMCR	Floating-point multiplier configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .M unit.

FIG. 3.22 : Registres de contrôles spécifiques aux TMS320c67xx

3.11 Modes d'adressage

Les modes d'adressages déterminent comment on accède à la mémoire. Ils précisent comment les données sont accessibles, telles que la récupération d'un opérande indirectement d'un espace de mémoire. Les deux modes d'adressage linéaire et circulaire sont supportés. Le mode le plus utilisé est l'adressage indirect de la mémoire.

3.11.1 Le mode d'adressage indirect

L'adressage indirect peut être utilisé avec ou sans déplacement (offset). Le registre R, représentant l'un des 32 registres A0 à A15 et B0 à B15, peut indiquer une adresse

mémoire. Ainsi, ces registres sont des pointeurs. Le mode d'adressage indirect utilise un "*" avec l'un des 32 registres. Pour illustrer ceci, considérant R comme un registre d'adresse.

1. *R. le registre R contient l'adresse d'un emplacement de mémoire où une valeur de données est stockée.

2. *R++(d). le registre R contient l'adresse mémoire (emplacement). Après utilisation de l'adresse de mémoire, R est post- incrémenté (modifié) de telle sorte que la nouvelle adresse est l'adresse actuelle plus le déplacement d. Si d = 1 (par défaut), la nouvelle adresse est (R + 1). Un double signe moins (-) postdécrément l'adresse (R-d).

3. *++R(d). L'adresse est pré-incrémentée par d, de telle sorte que l'adresse courante est (R+d). Un double moins est utilisé pour pré-décrémenter l'adresse mémoire de sorte que l'adresse est (R-d).

4. *+R(d). L'adresse est pré-incrémentée de d, de telle sorte que l'adresse courante est (R+d) (comme dans le cas précédent). Toutefois, dans ce cas, R pré-incrémentée sans modification.

3.11.2 Le mode d'adressage circulaire

L'adressage circulaire est utilisé pour créer une mémoire tampon circulaire (buffer). Ce tampon est créé dans le matériel et est primordial dans plusieurs algorithmes de traitement du signal, comme le filtrage ou la corrélation. La mise en œuvre d'un tampon circulaire est plus efficace en assembleur qu'en programmation C.

L'adressage circulaire est matérialisé dans le C6x. Ce mode d'adressage peut être utilisé en conjonction avec un tampon circulaire pour mettre à jour les échantillons en décalant les données sans surcharge, créé en décalant les données directement. Une fois le pointeur atteint une des deux extrémités du tampon circulaire, le pointeur est automatiquement déplacer vers l'autre extrémité.

Deux tampons circulaires indépendants sont disponibles en utilisant BK0 et BK1 avec le registre AMR. Les huit registres A4 à A7 et B4 à B7, en conjonction avec les deux unités (.D), peuvent être utilisés comme pointeurs (tous les registres peuvent être utilisés pour l'adressage linéaire). Le segment de programme suivant, en assembleur, illustre l'utilisation d'un tampon circulaire en utilisant le registre B2 (uniquement côté B peut être utilisé) afin de paramétrer le registre AMR :

MVKL .S2 0x0004,B2 ; les 16 bits poids faible vers B2. Sélectionner A5 comme pointeur.

MVKH .S2 0x0005,B2 ; les 16 bits poids forts vers B2. Sélectionner BK0, et mettre N = 5

MVC .S2 B2, AMR ; copier les 32 bits de B2 vers AMR

Les deux instructions de déplacement MVKL et MVKH (utilisant l'unité .S) fixe le registre B2 à la valeur de 32 bits de 0x00040005. L'instruction MVC (qui sert à déplacer une constante) est la seule instruction qui peut accéder au registre AMR. Elle met AMR à la même constante contenue dans B2.

Mode	Description
00	For linear addressing (default on reset)
01	For circular addressing using BK0
10	For circular addressing using BK1
11	Reserved

Tab. 3.1 : Mode AMR et description

Cette valeur définit le mode 01 et sélectionne le registre A5 comme le pointeur du buffer en utilisant le bloc BK0.

Le tableau dans la Figure 3.21 présente les modes associés aux registres A4 à A7 et B4 à B7. La valeur $0x0005 = (0101)_b$ dans la partie poids fort du registre AMR met les bits 16 et 18 à 1. Cela correspond à la valeur N, qui permet de fixer la longueur du buffer comme suit : $2^{N+1} = 2^{5+1} = 2^6 = 64$ octets en utilisant BK0. Si on avoir une taille de 128, les 16 bits de poids fort du AMR doit être égale à $(0110)_b = 0x0006(log2(128) - 1 = 6)$.

Dans le cas où un programme en assembleur est utilisé pour l'adressage circulaire, il faut réinitialiser AMR en mode linéaire lors d'un retour vers une fonction en C. Ainsi l'adresse du pointeur doit être enregistrée.

3.12 Les périphériques

3.12.1 Les Timers

Les timers permettent de mesurer la durée des événements, de compter des événements, de générer des impulsions, de générer des interruptions CPU et de synchroniser les échanges lors des accès directs à la mémoire.

Le C6713 contient deux timers à 32 bits chacun peuvent être utilisés comme horloge ou compteur d'événements ou source d'interruption. Le timer utilise un registre pour définir la période (fréquence de l'horloge), un registre de comptage, qui contient la valeur du compteur d'incrémentation, et un registre de commande de l'horloge, qui surveille l'état de timer. Chaque timer peut être piloté par une horloge interne ou une horloge externe. Il possède chacun une TINPx et une TOUTx.

Les registres des timers Chaque timer possède 3 registres permettant de le configurer, de fixer la période et un registre contenant du comptage.

Les TIMERx CTL, TIMERx PRD et TIMERx CNL sont des registres 32bits.

Le registre de TIMERx PRD contient le nombre de cycle d'horloge de ''timer input '' à compter. Ce nombre contrôle la fréquence de TSTAT.

Le registre de TIMERx CNT incrémente quand il lui est permis de compter. Il réinitialise à 0 sur la prochaine CPU chronométrée après que la valeur dans le registre de TIMERx est atteinte.

Hex Byte Address			Name and Abbreviation	Description
Timer 0	Timer 1	Timer 2		
01940000	01980000	01AC0000	Timer Control (CTL)	Determines the operating mode of the timer, monitors the timer status, and controls the function of the TOUT pin.
01940004	01980004	01AC0004	Timer Period (PRD)	Contains the number of timer input clock cycles to count. This number controls the TSTAT signal frequency.
01940008	01980008	01AC0008	Timer Counter (CNT)	Current value of the incrementing counter

FIG. 3.23 : Les registres des timers :

3.12.2 Les Interruptions

Une interruption peut être générée à l'intérieur ou l'extérieur du DSP. Une interruption interrompe le programme en cours d'exécution par l'unité centrale afin d'effectuer la tâche prioritaire définie dans le sous-programme d'interruption. Les sources des interruptions sont le CAN, les timers, ... Lors d'une interruption, les données du programme en cours doivent être sauvegardées de façon à pouvoir les restaurées après que la tâche d'interruption soit exécutée. Lorsqu'une interruption surgisse, les contenus des registres sont sauvegardés et le traitement de la tâche d'interruption est entamé. Ensuite, les contenus des registres sont restaurés.

Dans le C6713, il y a 16 sources d'interruption :

- 2 interruptions des deux timers,

- 4 interruptions externes,

- 4 interruptions McBSP,

- et 4 interruptions CPU DMA interrupts.

12 interruptions sont disponibles pour le CPU ; où un sélecteur d'interruption est utilisé pour choisir entre les interruptions (INT4-INT11).

Registres de contrôle des interruptions :

Les registres de contrôle des interruptions sont :

1. CSR (control status register) : qui contient le bit d'activation d'interruption globale et d'autres bits de contrôle et d'indication d'états

2. IER (interrupt enable register) : pour activer/désactiver individuellement les interruptions

3. IFR (interrupt flag register) : affiche les états des interruptions

4. ISR (interrupt set register) : gère les interruptions en attente

5. ICR (interrupt clear register) : annule les interruptions en attente

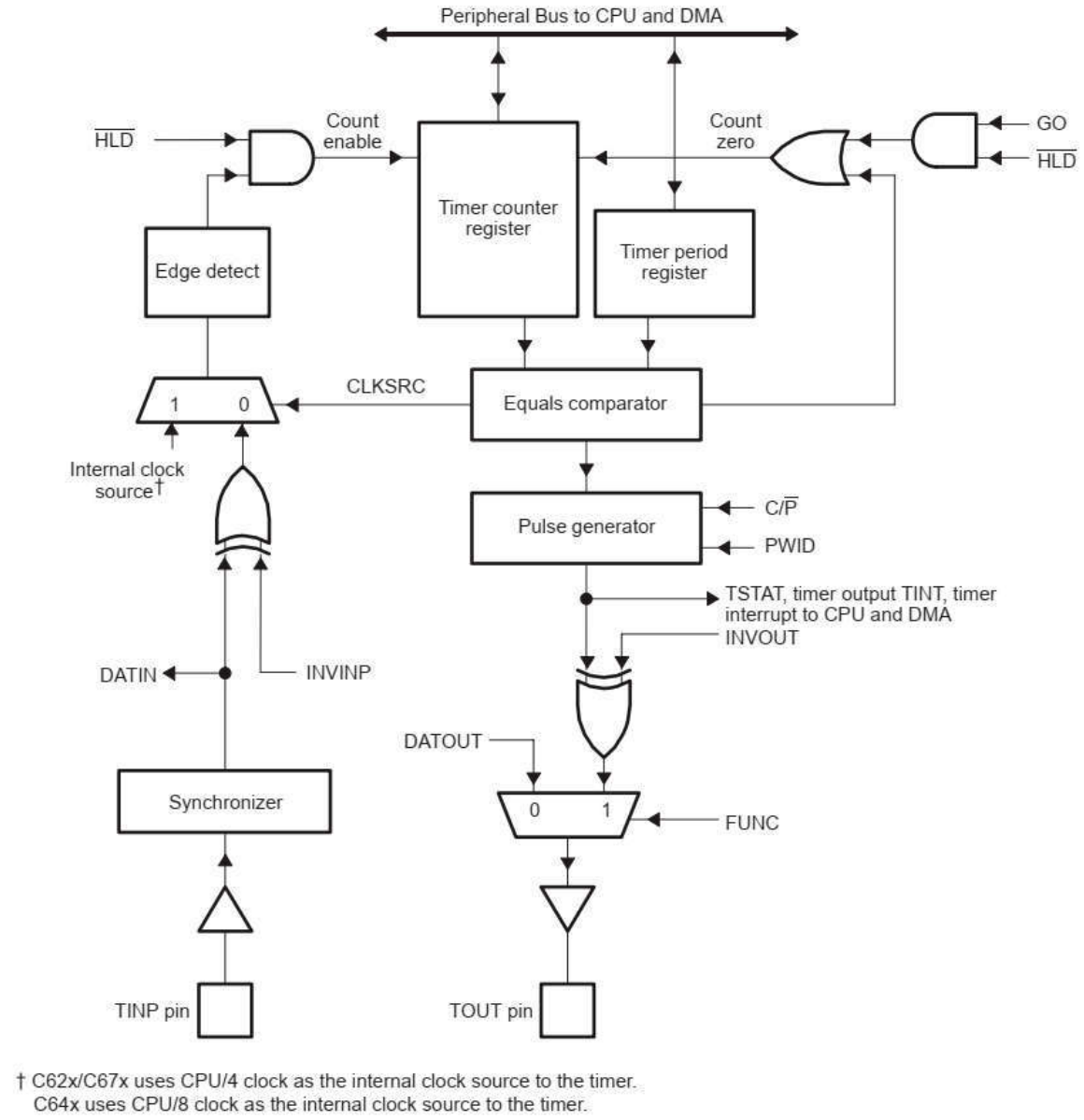


FIG. 3.24 : Schéma bloc du timer

31				12	11	10	9	8
Rsvd					TSTAT	INVINP	CLKSRC	C/P
R, +0					R, +0	RW, +0	RW, +0	RW, +0
7	6	5	4		3	2	1	0
HLD	GO	Rsvd	PWID		DATIN	DATOUT	INVOUT	FUNC
RW, +0	RW, +0	R, +0	RW, +0		R, +X	RW, +0	RW, +0	RW, +0

FIG. 3.25 : Timer Control Register (CTL)

6. ISTP (interrupt service table pointer) : localise les sous-programmes des interruptions

7. IRP (interrupt return pointer) pour le retour d'un sous-programme d'interruption

8. NRP (nonmaskable interrupt return pointer) pour les interruptions nonmasquables

No.	Bitfield	Description
31–12	Rsvd	Reserved.
11	TSTAT	Timer status. Value of timer output.
10	INVINP	TINP inverter control. Only affects operation if CLKSRC = 0. INVINP = 0: Uninverted TINP drives timer. INVINP = 1: Inverted TINP drives timer.
9	CLKSRC	Timer input clock source CLKSRC = 0: External clock source drives the TINP pin. CLKSRC = 1: Internal clock source. For C62x/C67x: CPU clock/4 For C64x: CPU clock/8
8	C/$\overline{P}$	Clock/pulse mode C/$\overline{P}$ = 0: Pulse mode. TSTAT is active one CPU clock after the timer reaches the timer period. PWID determines when it goes inactive. C/$\overline{P}$ = 1: Clock mode. TSTAT has a 50% duty cycle with each high and low period one countdown period wide.
7	$\overline{HLD}$	Hold. Counter may be read or written regardless of $\overline{HLD}$ value. $\overline{HLD}$ = 0: Counter is disabled and held in the current state. $\overline{HLD}$ = 1: Counter is allowed to count.
6	GO	GO bit. Resets and starts the timer counter. GO = 0: No effect on the timers. GO = 1: If $\overline{HLD}$ = 1, the counter register is zeroed and begins counting on the next clock.
5	Rsvd	Reserved.
4	PWID	Pulse width. Only used in pulse mode (C/$\overline{P}$ = 0). PWID = 0: TSTAT goes inactive one timer input clock cycle after the timer counter value equals the timer period value. PWID = 1: TSTAT goes inactive two timer input clock cycles after the timer counter value equals the timer period value.
3	DATIN	Data in: Value on TINP pin
2	DATOUT	Data output When FUNC = 0: The DATOUT is driven on TOUT. When FUNC = 1: The TSTAT is driven on TOUT after inversion by INVOUT.
1	INVOUT	TOUT inverter control. Used only if FUNC = 1. INVOUT = 0: Uninverted TSTAT drives TOUT. INVOUT = 1: Inverted TSTAT drives TOUT.
0	FUNC	Function of TOUT pin FUNC = 0: TOUT is a general-purpose output pin. FUNC = 1: TOUT is a timer output pin.

FIG. 3.26 : Bits du registre de TIMERx CTL

Les interruptions possèdent des priorités, où l'interruption de remise à zéro (RESET) a la plus haute priorité. L'interruption RESET et l'interruption non-masquable (NMI)

(deuxième haute priorité) sont activées à travers des pins externes. Le registre d'activation des interruptions (IER) est utilisé pour activer une interruption spécifique et permet de vérifier si et quelle interruption s'est manifestée à travers le registre des drapeaux d'indications des interruptions (IFR). NMI (nonmaskable interrupt) peut être masquée (désactivée) en mettant le bit d'activation d'interruption non masquable (NMIE) à zéro dans le registre CSR.

Le signal RESET est un signal actif bas utilisé pour redémarrage du processeur, et le signal NMI informe le processeur de la survenue d'un problème matériel grave. 12 interruptions CPU avec une priorité inférieure sont disponibles, correspondant aux signaux masquables INT4 par INT15.

Les priorités de ces interruptions sont : INT4, INT5. . . , INT15, avec INT4 ayant la priorité la plus élevée et l'INT15 de priorité la plus basse. Pour traiter une interruption masquable, le bit GIE du registre RSE et le bit NMIE du registre IER sont mis à 1.

Le bit d'activation d'interruption (IE) correspondant à l'interruption masquable souhaitée est également mis à 1. Lorsque l'interruption se produit, le bit IFR correspondant est mis à 1 pour indiquer l'état de l'interruption. Le traitement d'une interruption masquable nécessite les étapes suivantes :

1. Le bit GIE est mis à 1.
2. Le bit NMIE est mis à 1.
3. Le bit IE approprié est mis à 1.
4. Le bit IFR correspondant est mis à 1.

Pour que l'interruption se produise, la CPU ne doit pas être en cours d'exécution une boucle d'attente (delay slot) associée à une instruction de branchement.

Le tableau de service d'interruption (IST), illustré par le tableau 3.3, est utilisé au début d'une interruption.

Chaque emplacement lui est associé un FP à chaque interruption. Le tableau contient 16 FPs, chacun avec huit instructions. Les adresses sur le côté droit correspondent à un décalage associé à chaque interruption spécifique. Par exemple,le FP pour interruption INT11 est à une adresse de base plus un décalage de 160 h.

Comme chaque FP contient huit instructions de 32 bits (256 bits) ou 32 octets, chaque adresse est incrémentée de 32 (20h) dans le tableau.

Le FP RESET doit être mis à l'adresse 0. Toutefois, les FPs associés aux autres interruptions peuvent être déplacés. La nouvelle adresse est spécifiée dans le registre ISTB (interrupt service table base). Initialement le registre ISTB est à 0.

Acquittement des interruptions :

Les signaux IACK and INUMx (INUM0 à INUM3) sont des pins hardware sur le processeur C6x pour informer qu'une interruption s'est manifestée et qu'elle est en cours de traitement. Les quatre signaux INUMx indiquent le numéro de l'interruption en cours d'exécution. Par exemple,

Interrupt	Offset
RESET	000h
NMI	020h
Reserved	040h
Reserved	060h
INT4	080h
INT5	0A0h
INT6	0C0h
INT7	0E0h
INT8	100h
INT9	120h
INT10	140h
INT11	160h
INT12	180h
INT13	1A0h
INT14	1C0h
INT15	1E0h

TAB. 3.2 : service d'interruptions

INUM3 = 1 (MSB), INUM2 = 0, INUM1 = 1, INUM0 = 1 (LSB)

Corresponds à $(1011)_b = 11$, indique que l'interruption INT11 est en cours d'exécution.

Le bit IE11 est mis à 1 pour activer l'interruption INT11. Le registre IFR est lu pour vérifier que le bit IF11 est mis à 1 (INT11 active).

Mettre à 1 un bit dans le registre ISR (interrupt set register) produit une mise à 1 du drapeau correspondant dans le registre IFR ; cependant mettre à 0 un bit dans le registre ICR (interrupt clear register) causera l'effacement de l'interruption correspondante.

Toutes les interruptions restent en attente lorsque le CPU a une instruction de branchement en attente. Puisqu'une instruction de branchement a cinq périodes de retard, une boucle avec un nombre de cycles inférieur à 6 est non-interruptible. Toute interruption en attente sera traitée tant qu'il n'y a pas de branchements en attente.

3.12.3 Le port HPI

Le port HPI **''Host Port Interface ''** est un port 16 bits pouvant être adressé directement à la mémoire. L'hôte et le CPU peuvent échanger des informations via une mémoire interne ou externe. Le HPI permet notamment à un processeur « hôte » externe d'effectuer des accès mémoire directs, que celle-ci soit interne ou dans l'espace mémoire du 6713 propriétaire.

La figure représente une interface entre un hôte et un 6713. On pourra remarquer que le format des données soit sur 32 bits, le port, lui n'en contient que 16 bits. Le transfert se fera donc en 2 étapes, l'ordre du demi-mot transféré étant repéré par le HWOB du registre HPIC.

Abréviation	Nom de registre	'accès autorisé à l'hôte	accès CPU	adresse
HPID	HPI data	R/W	-	-
HPIA	HPI adresse	R/W	-	-
HPIC	HPI controle	R/W	RW	0x01880000

Tab. 3.3 : Registres de HPI

Les broches HDS1, HDS2, HR/W et HAS permettent un interfaçage avec un nombre de composant sans ajouts de logique d'interface.

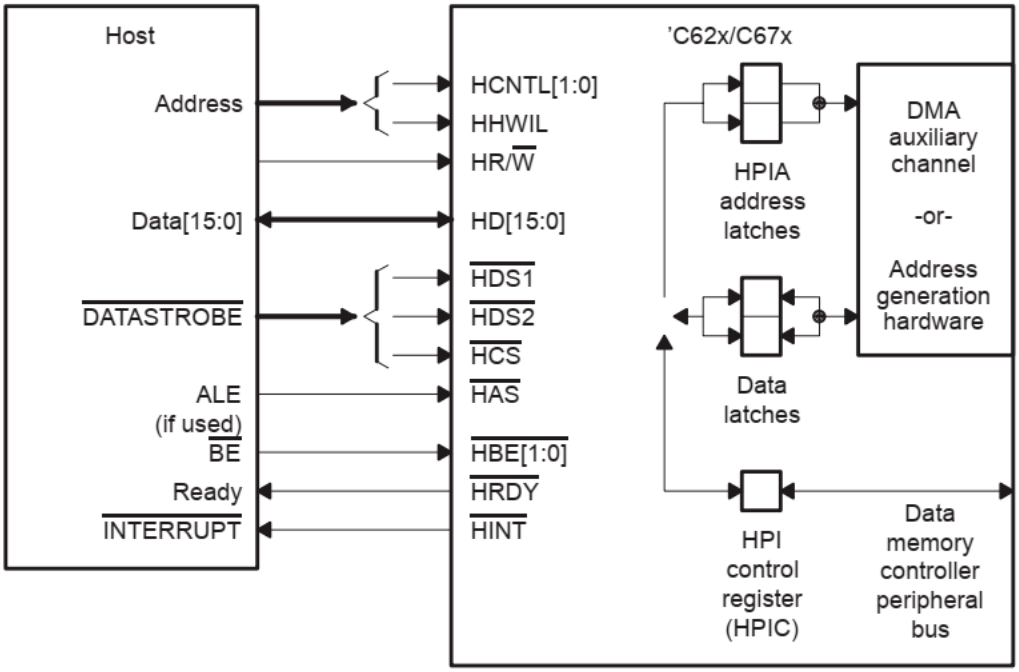


Fig. 3.27 : Description du module HPI

Les registres de HPI :
Les registres de HPI sont utilisés pour la communication entre l'appareil hôte et le CPU.

- HPID contient la donnée à échange sur 32 bits.
- HPIA contient l'adresse sur 30 bits donc les deux LSBs (les deux bits de poids faible) ne sont pas utilisés par les HPIA writes et sont toujours lus comme 0.
- HPIC est normalement le premier registre à accéder à la configuration des bits et à initialiser l'interface.

Le HPIC est organisé en deux mots de 16 bits dont l'organisation du point de vue des champs est identique. Lorsque l'hôte écrit, les deux mots doivent être identiques. La CPU seule écrite dans le mot de poids faible.

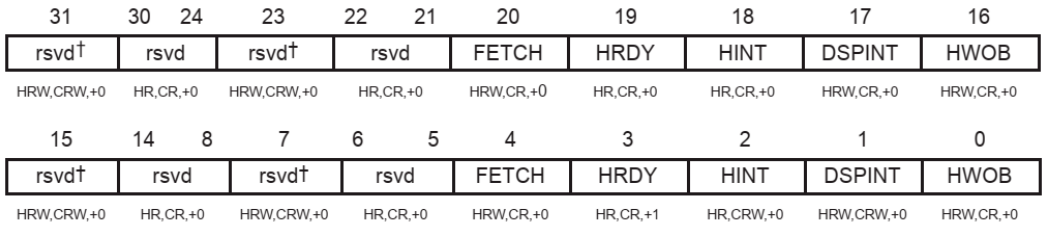


FIG. 3.28 : Registre HPI

Bit	Description
HWOB	Halfword ordering bit If HWOB = 1, the first halfword is least significant. If HWOB = 0, the first halfword is most significant. HWOB affects both data and address transfers. Only the host can modify this bit. HWOB must be initialized before the first data or address register access. For HPI32, HWOB is not used and the value of HWOB is irrelevant.
DSPINT	The host processor-to-CPU/DMA interrupt
HINT	DSP-to-host interrupt. The inverted value of this bit determines the state of the CPU $\overline{\text{HINT}}$ output.
HRDY	Ready signal to host. Not masked by $\overline{\text{HCS}}$ (as the $\overline{\text{HRDY}}$ pin is). If HRDY = 0, the internal bus is waiting for an HPI data access request to finish.
FETCH	Host fetch request The value read by the host or CPU from this register field is always 0. The host writes a 1 to this bit to request a fetch into HPID of the word at the address pointed to by HPIA. The 1 is never actually written to this bit, however.

FIG. 3.29 : Description du registre HPI

3.12.4 La liaison série McBSP (multichannel buffered serial port)

Le 6713 contient deux modules de liaison série (McBSP0 et McBSP1). Ces liaisons séries permettent des communications en full-duplex, un flot continu des données, une indépendance dans le timing, le format des données à l'émission et à la réception, une interface directe avec les Codec, CNA/CAN, et une synchronisation par horloge interne ou externe. Ces liaisons transmettent des données et des informations de contrôle. La communication avec des composants externes se fait en émission et en réception sur des broches différentes.

Le contrôle de la communication se fait par 4 autres broches comme indiquées dans la figure 3.30. Le format des données est fixé dans des registres de contrôle accessible par le CPU.

Registres de configuration :

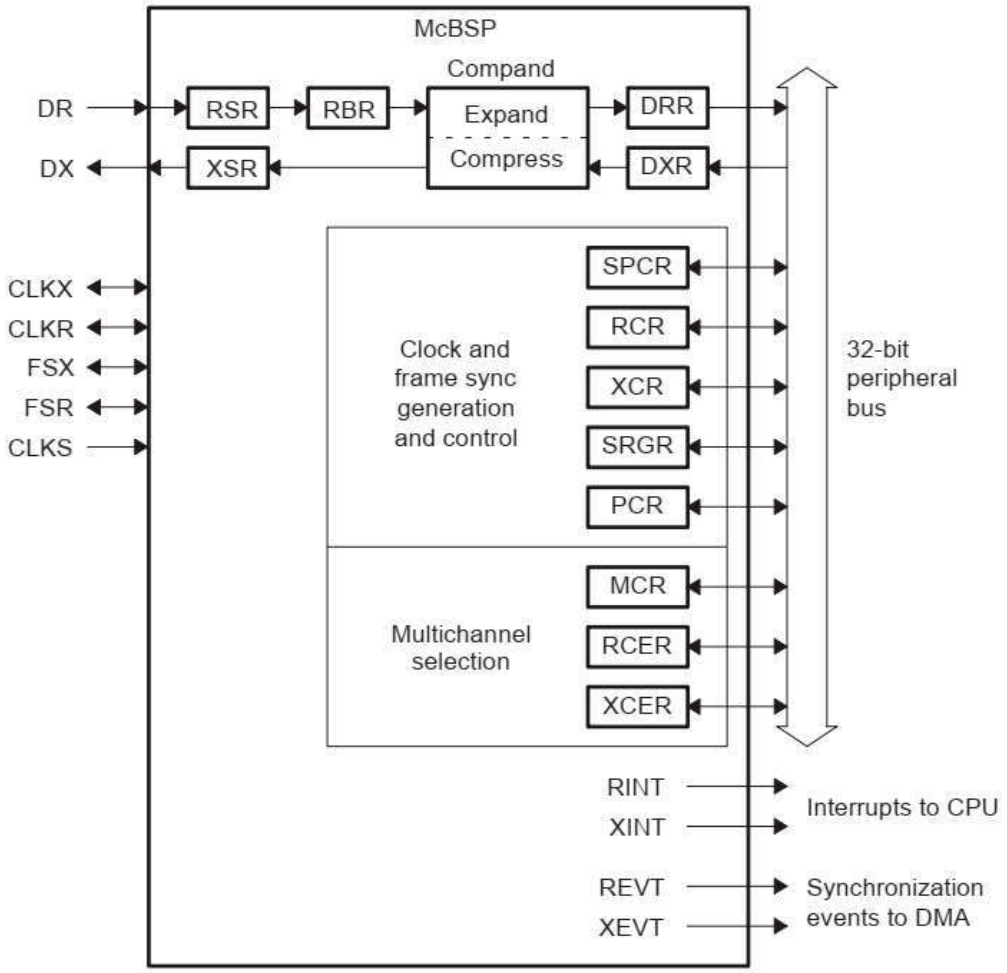


FIG. 3.30 : Schéma bloc du module « liaison série » du TMS320C6713

Interruption :

- RINT est l'interruption générée à la réception.
- XINT est l'interruption générée à l'émission.

3.13 Contrôleur EDMA (Enhanced Direct Memory Access)

Le contrôleur EDMA (Enhanced Direct Memory Access) gère tous les transferts de données entre le contrôleur de cache / mémoire de niveau deux (L2) et les périphériques de l'appareil sur le TMS320C621x / C671x / C64x. Ces transferts de données incluent la maintenance de la mémoire cache, les accès à la mémoire sans mise en cache, les transferts de données programmés par l'utilisateur et les accès aux hôtes.

L'architecture du contrôleur EDMA du C621x / C671x / C64x est différente de celle du contrôleur DMA précédent des appareils C620x / C670x. L'EDMA inclut plusieurs améliorations du DMA en ce qu'il fournit 64 canaux (C64x) ou 16 canaux C621x / C671x,

Hex Byte Address		Abbrevi-ation	McBSP Register Name
McBSP 0	McBSP 1		
–	–	RBR	Receive buffer register
–	–	RSR	Receive shift register
–	–	XSR	Transmit shift register
018C 0000	0190 0000	DRR	Data receive register
018C 0004	0190 0004	DXR	Data transmit register
018C 0008	0190 0008	SPCR	Serial port control register
018C 000C	0190 000C	RCR	Receive control register
018C 0010	0190 0010	XCR	Transmit control register
018C 0014	0190 0014	SRGR	Sample rate generator register
018C 0018	0190 0018	MCR	Multichannel control register
018C 001C	0190 001C	RCER RCERE0	Receive channel enable register Enhanced receive channel enable register
018C 0020	0190 0020	XCER XCERE0	Transmit channel enable register Enhanced transmit channel enable register
018C 0024	0190 0024	PCR	Pin control register

FIG. 3.31 : Registres de configuration du McBSP

Pin	I/O/Z	Description
CLKR	I/O/Z	Receive clock
CLKX	I/O/Z	Transmit clock
CLKS	I	External clock
DR	I	Received serial data
DX	O/Z	Transmitted serial data
FSR	I/O/Z	Receive frame synchronization
FSX	I/O/Z	Transmit frame synchronization

Note: I = Input, O = Output, Z = High Impedance

FIG. 3.32 : Signaux d'interface McBSP

avec une priorité programmable, et la possibilité de lier et de chaîner les transferts de données. L'EDMA permet le déplacement des données vers / depuis n'importe quel espace mémoire adressable, y compris la mémoire interne (L2 SRAM), les périphériques et la mémoire externe.

Le contrôleur EDMA comprend : • Registres de traitement des événements et des interruptions

- Encodeur d'événement
- Paramètre RAM, et
- Matériel de génération d'adresses

Un schéma de principe du contrôleur EDMA est illustré à la Figure 3.34.

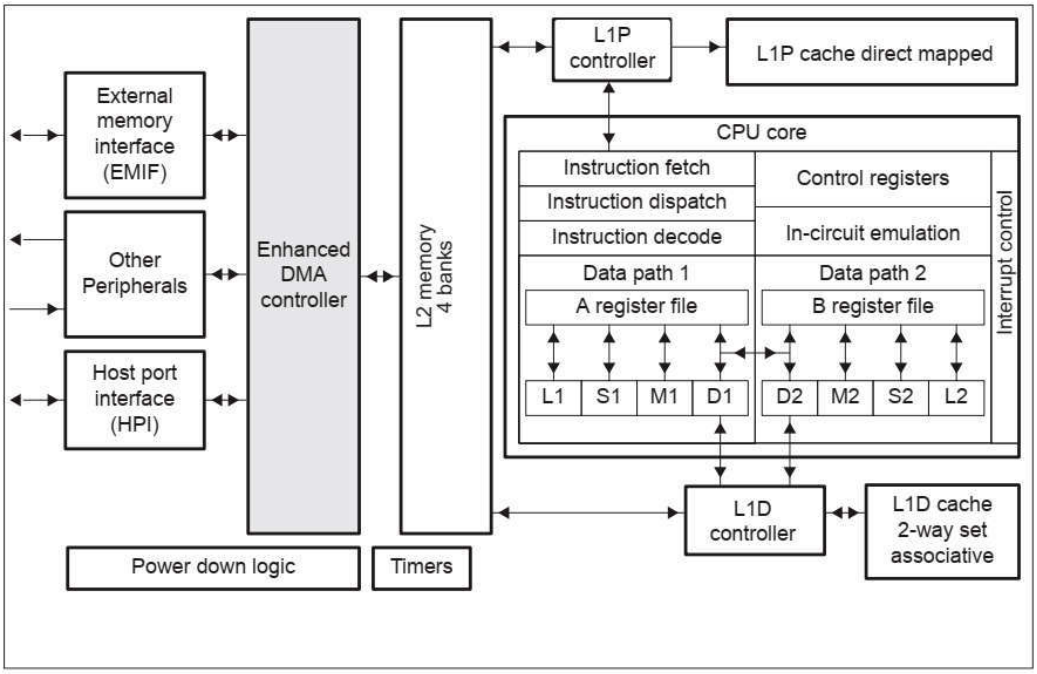


FIG. 3.33 : Schéma fonctionnel TMS320C621x / C671x / C64x

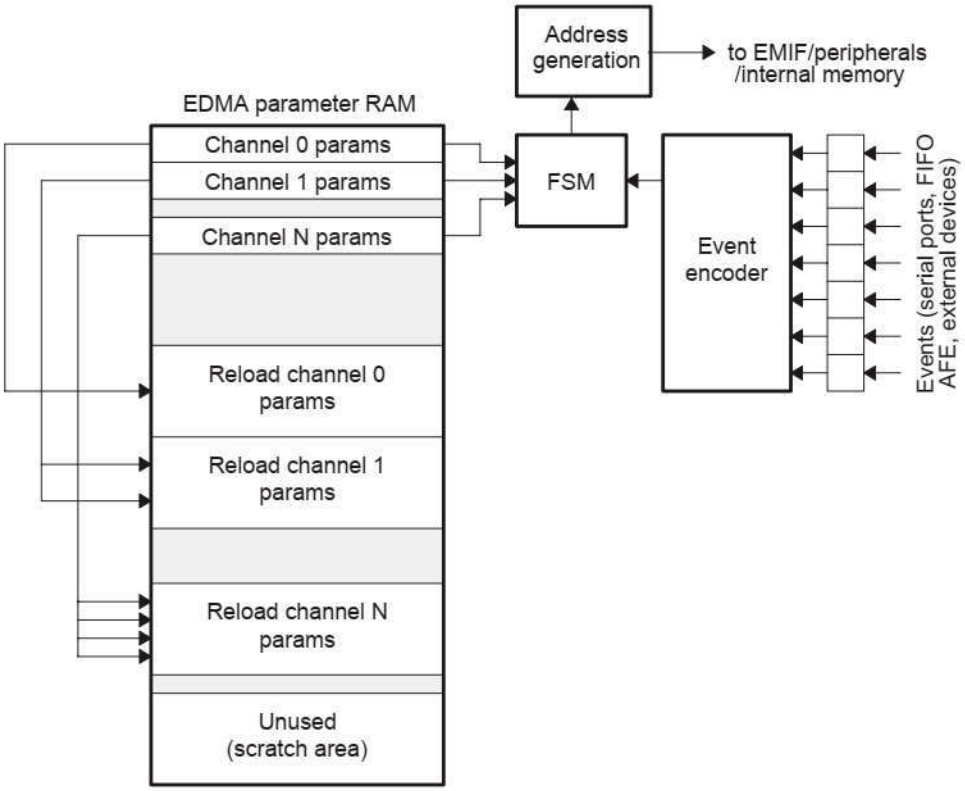


FIG. 3.34 : Contrôleur EDMA

Les événements EDMA sont capturés dans le registre des événements. Un événement est un signal de synchronisation qui déclenche un canal EDMA pour démarrer un transfert. Si des événements se produisent simultanément, ils sont résolus au moyen du codeur

d'événements. Les paramètres de transfert correspondant à cet événement sont stockés dans la RAM de paramètres EDMA, et sont transmis au matériel de génération d'adresse, qui adresse l'EMIF et / ou les périphériques pour effectuer les transactions de lecture et d'écriture nécessaires.

L'EDMA a la capacité d'effectuer des transferts rapides et efficaces en acceptant une requête DMA rapide (QDMA) de la CPU. Un transfert QDMA est le mieux adapté pour les applications qui nécessitent des transferts de données rapides, telles que les demandes de données dans un algorithme en boucle serrée.

3.14 L'interface de mémoire externe (External Memory Interface EMIF)

L'interface de mémoire externe du TMS320C6713 permet l'interfaçage avec différents types de composants : SBSRAM, SRAM, RAM, FIFO, EDMA ..... L'EMIF du TMS320C6713 nécessite une horloge externe (ECLKIN) fournie par le système. L'horloge ECLKOUT est élaborée par l'EMIF et toutes les mémoires externes doivent travailler avec cette horloge.

Les signaux de l'EMIF :
Le schéma suivant représente les signaux nécessaires au fonctionnement de l'EMIF.

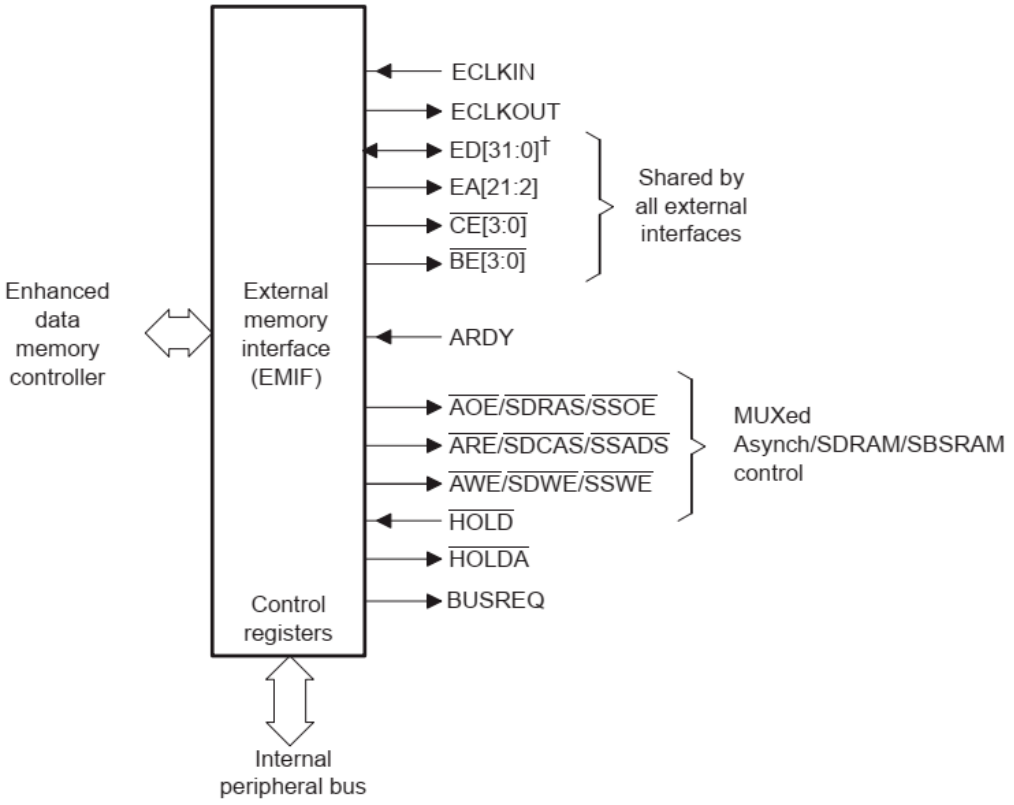


Fig. 3.35 : les signaux de l'EMIF

Pin	(I/O/Z)	Description
CLKOUT1	O	Clock output. Runs at the CPU clock rate.
CLKOUT2	O	Clock output. Runs at 1/2 the CPU clock rate. Used for synchronous memory interface on Group 2 devices.
ECLKOUT	O	EMIF clock output. All EMIF I/O are clocked relative to ECLKOUT.
ED[31:0]§	I/O/Z	EMIF 32-bit data bus I/O§
EA[21:2]	O/Z	External address output. Drives bits 21–2 of the byte address. (Effectively a word address.)
$\overline{CE0}$	O/Z	Active-low chip select for memory space CE0
$\overline{CE1}$	O/Z	Active-low chip select for memory space CE1
$\overline{CE2}$	O/Z	Active-low chip select for memory space CE2
$\overline{CE3}$	O/Z	Active-low chip select for memory space CE3
$\overline{BE[7:0]}$	O/Z	Active-low byte enables.On C64x, byte-enables go active for only the appropriate byte lane for both writes and reads.
ARDY	I	Ready. Active-high asynchronous ready input used to insert wait states for slow memories and peripherals.
$\overline{AOE}$	O/Z	Active-low output enable for asynchronous memory interface
$\overline{AWE}$	O/Z	Active-low write strobe for asynchronous memory interface
$\overline{ARE}$	O/Z	Active-low read strobe for asynchronous memory interface
$\overline{SSADS}$	O/Z	Active-low address strobe/enable for SBSRAM interface
$\overline{SSOE}$	O/Z	Active low output buffer enable for SBSRAM interface
$\overline{SSWE}$	O/Z	Active-low write enable for SBSRAM interface
$\overline{SDRAS}$	O/Z	Active-low row address strobe for SDRAM memory interface
$\overline{SDCAS}$	O/Z	Active-low column address strobe for SDRAM memory interface
$\overline{SDWE}$	O/Z	Active-low write enable for SDRAM memory interface

FIG. 3.36 : Description des signaux de l'EMIF

Byte Address		Abbreviation	EMIF Register Name
EMIF/EMIFA	EMIFB‡		
0180 0000h	01A8 0000h	GBLCTL	EMIF global control
0180 0004h	01A8 0004h	CE1CTL	EMIF CE1 space control
0180 0008h	01A8 0008h	CE0CTL	EMIF CE0 space control
0180 000Ch	01A8 000Ch		Reserved
0180 0010h	01A8 0010h	CE2CTL	EMIF CE2 space control
0180 0014h	01A8 0014h	CE3CTL	EMIF CE3 space control
0180 0018h	01A8 0018h	SDCTL	EMIF SDRAM control
0180 001Ch	01A8 001Ch	SDTIM	EMIF SDRAM refresh control

FIG. 3.37 : Les registres de l'EMIF

3.15 Jeu d'instructions TMS320C6000

Les DSP TMS320C6000 ont un jeu d'instructions complet qui est adapté aux algorithmes utilisés dans le traitement du signal numérique. Il existe des instructions pour l'addition, la soustraction et la multiplication à virgule fixe et flottante, ainsi que pour les opérations logiques, la mise en mémoire tampon circulaire et le chargement et le stockage des données. Toutes les instructions peuvent être exécutées de manière conditionnelle.

les instructions avec le préfixe ADD font l'addition, MPY est une multiplication, LDW charge un mot de la mémoire dans un registre, STB stocke un octet d'un registre dans la mémoire et MPYSP effectue une multiplication à virgule flottante simple précision.

3.15.1 Format du code assembleur

Un format de code assembleur est représenté par le champ

Label || [ ] Instruction Unit Operands ;comments

Une étiquette (Label), si elle est présente, représente une adresse ou un emplacement mémoire spécifique contenant une instruction ou des données. L'étiquette doit être dans la première colonne. Les barres parallèles (||) sont là si l'instruction est exécutée en parallèle avec l'instruction précédente. Le champ suivant est facultatif pour rendre l'instruction associée conditionnelle. Cinq des registres - A1, A2, B0, B1 et B2 - peuvent être utilisés comme registres conditionnels. Par exemple, [A2] spécifie que l'instruction associée s'exécute si A2 n'est pas zéro. Par contre, avec [! A2], l'instruction associée s'exécute si A2 est nul. Toutes les instructions C6x peuvent être rendues conditionnelles avec les registres A1, A2, B0, B1 et B2 en déterminant quand le registre conditionnel est nul. Le champ d'instruction peut être une directive assembleur ou un mnémonique. Une directive assembleur est une commande pour l'assembleur. Par example :

.word value

réserve 32 bits en mémoire et se remplit avec la valeur spécifiée. Un mnémonique est une instruction réelle qui s'exécute au moment de l'exécution. L'instruction (mnémonique ou directive assembleur) ne peut pas démarrer dans la colonne 1. Le champ Unit, qui peut être l'une des huit unités fonctionnelles de l'UC, est facultatif. Les commentaires commençant dans la colonne 1 peuvent commencer par un astérisque ou un point-virgule, tandis que les commentaires commençant dans toutes les autres colonnes doivent commencer par un point-virgule.

3.15.1.1 Types d'instructions

1. Add/Subtract/Multiply

(a) Instruction

ADD .L1 A3,A7,A7 ;add A3 + A7 → A7 (accum in A7)

ajoute les valeurs dans les registres A3 et A7 et place le résultat dans le registre A7.

L'unité .L1 est facultative. Si la destination ou le résultat est en B7, l'unité sera .L2.

(b) Instruction

SUB .S1 A1,1,A1 ;subtract 1 from A1

soustrait 1 de A1 pour le décrémenter en utilisant l'unité .S.

(c) Instruction parallèle

MPY .M2 A7,B7,B6 ;multiply 16 LSBs of A7, B7 → B6

|| MPYH .M1 A7,B7,A6 ;multiply 16 MSBs of A7, B7 → A6

multiplie les 16 bits inférieurs ou les moins significatifs (LSB) de A7 et B7 et place le produit dans B6, en parallèle (simultanément dans le même paquet d'exécution) avec une seconde instruction qui multiplie les 16 bits supérieurs ou les plus significatifs (MSB) de A7 et B7 et place le résultat dans A6. De cette manière, deux opérations MAC peuvent être exécutées dans un seul cycle d'instructions. Cela peut être utilisé pour décomposer une somme de produits en deux ensembles de somme de produits : un ensemble utilisant les 16 bits inférieurs pour fonctionner sur le premier, le troisième, le cinquième,. . . et un autre ensemble utilisant les 16 bits supérieurs pour fonctionner sur les deuxième, quatrième, sixième,. . . Notez que le symbole parallèle n'est pas dans la colonne 1.

2. Load/Store

(a) Instruction

LDH .D2 *B2++,B7;load (B2) → B7, increment B2

|| LDH .D1 *A2++,A7;load (A2) → A7, increment A2

charge dans B7 le demi-mot (16 bits) dont l'adresse en mémoire est spécifiée / pointée par B2. Ensuite, le registre B2 est incrémenté (post-incrémenté) pour pointer vers l'adresse mémoire immédiatement supérieure. En parallèle se trouve une autre instruction de mode d'adressage indirect pour charger dans A7 le contenu en mémoire dont l'adresse est spécifiée par A2. Ensuite, A2 est incrémenté pour pointer vers la prochaine adresse de mémoire supérieure.

L'instruction LDW charge un mot de 32 bits. Deux chemins utilisant .D1 et .D2 permettent le chargement des données de la mémoire vers les registres A et B à l'aide de l'instruction LDW. L'instruction à virgule flottante LDDW de chargement de deux mots sur le C6713 peut charger simultanément deux registres 32 bits du côté A et deux registres 32 bits du côté B.

(b) Instruction

STW .D2 A1,*+A4[20] ;store A1→(A4) offset by 20

stocke en mémoire le mot de 32 bits A1 dont l'adresse est spécifiée par un décalage A4 de 20 mots (32 bits) ou 80 octets. Le registre d'adresse A4 est pré-incrémenté avec offset, mais il n'est pas modifié (deux signes plus sont utilisés si A4 doit être modifié).

3. Branch/Move

Le segment de code suivant illustre le branchement et le transfert de données :

```
Loop MVKL .S1 x,A4  ; move 16 LSB of x address →A4
     MVKH .S1 x,A4  ; move 16 MSBs of x address →A4
     .
     .
     .
     SUB .S1 A1,1,A1  ; decrement A1
[A1] B .S2 Loop  ; branch to Loop if A1 != 0
     NOP 5  ; five no - operation instructions
     STW .D1 A3,*A7  ; store A3 into (A7)
```

La première instruction déplace les 16 bits inférieurs (LSB) de l'adresse x dans le registre A4. La deuxième instruction déplace les 16 bits supérieurs (MSB) de l'adresse x dans A4, qui contient maintenant l'adresse 32 bits complète de x. Il faut utiliser les instructions MVKL / MVKH pour obtenir une constante 32 bits dans un registre.

Le registre A1 est utilisé comme compteur de boucle. Après avoir été décrémenté avec l'instruction SUB, il est testé pour une branche conditionnelle. L'exécution se branche sur l'étiquette ou la boucle d'adresse si A1 n'est pas égal à zéro. Si A1 = 0, l'exécution se poursuit et les données du registre A3 sont stockées en mémoire dont l'adresse est spécifiée (pointée) par A7.

.L unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SSHL	LDB	SUBAB
CMPGT	MPYHU	B IRP	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPHLU	EXT	ZERO	LDB (15-bit offset)	
MV	MPYHULS	EXTU		LDBU (15-bit offset)	
NEG	MPYHSLU	MV		LDH (15-bit offset)	
NORM	MPYLH	MVC		LDHU (15-bit offset)	
NOT	MPYLHU	MVK		LDW (15-bit offset)	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

FIG. 3.38 : Instructions communes aux 'C62x et' C67x

Chapitre 4

Gestion de la mémoire

4.1 Introduction

L'organisation de la mémoire et la méthode de sa gestion dépendent des ressources matérielles du système conçu et du degré de complexité des tâches que le système exécute. La méthode d'allocation et la manière d'utiliser une mémoire dépendent de la structure et de la capacité de mémoire interne du processeur de signal. Le temps d'accès à la mémoire a une influence cruciale sur la dynamique d'un système.

Il existe deux options possibles de gestion des ressources mémoire :

– l'utilisation d'un modèle de mémoire par défaut résultant de l'architecture du processeur de signal ;

– mise en place d'un modèle de mémoire spécifié par l'utilisateur lorsque les ressources mémoire du processeur sont étendues.

Dans le second cas, les ressources mémoire d'un système doivent être déclarées dans le compilateur, en utilisant les directives d'un éditeur de liens dans un fichier de commandes. Ces directives, outre une déclaration de la taille de la mémoire et de l'adresse de son emplacement, contiennent également un nom de bloc mémoire unique et les attributs : lecture/écriture, initialisation et contenu, par ex. le code d'exécution du programme.

Lors d'une compilation du fichier source, les ressources mémoire peuvent être allouées de manière statique ou dynamique. De plus, la zone mémoire pour les variables peut être allouée dynamiquement ainsi que pendant l'exécution du programme. Les logiciels fournis par les fabricants de matériel simplifient la gestion des ressources mémoire.

Les sections dites sont des objets de base utilisés dans la gestion des ressources mémoire. Ils sont affectés à des constantes, des variables, des tableaux, à une pile, un code du programme exécuté ainsi qu'à la gestion dynamique de la mémoire.

Les sections sont créées par défaut dans le processus de compilation (en tenant compte des ressources de mémoire physique du système conçu) ou elles peuvent être définies par l'utilisateur par des directives de l'éditeur de liens. Les sections définies n'ont pas besoin d'être initialisées.

La possibilité de définir les sections par un utilisateur simplifie le processus de gestion de la mémoire. Les sections créées sont affectées aux ressources mémoire du processeur déclarées dans le fichier de commandes de l'éditeur de liens.

La gestion de la mémoire dans les processeurs de signaux produits par la plupart des principaux fabricants est similaire. Certaines différences incluent les noms de fichiers, l'organisation des sections et la méthode pour y faire référence.

Les fabricants de processeurs de signal, outre des instructions d'allocation dynamique et de libération de ressources mémoire issues du langage C standard (malloc, free), proposent également des instructions supplémentaires. Ce sont des extensions du langage C standard, dans lesquelles il est possible à la fois de déclarer la taille d'une variable et d'allouer de la mémoire. Cependant, ces instructions ont certaines restrictions, par ex. ils ne peuvent pas être utilisés dans la routine de service d'interruption.

4.2 Mémoire interne du C6713

Le C6713 a une architecture de mémoire à deux niveaux. Les caches de niveau 1 font 4 Ko chacun (programme et données). La mémoire de niveau 2 est de 256 Ko, et jusqu'à ¼ de celle-ci peut être mise en cache. Vous pouvez en fait ajouter des moyens de cache de 16 Ko pour un cache associatif à 4 voies maximum.

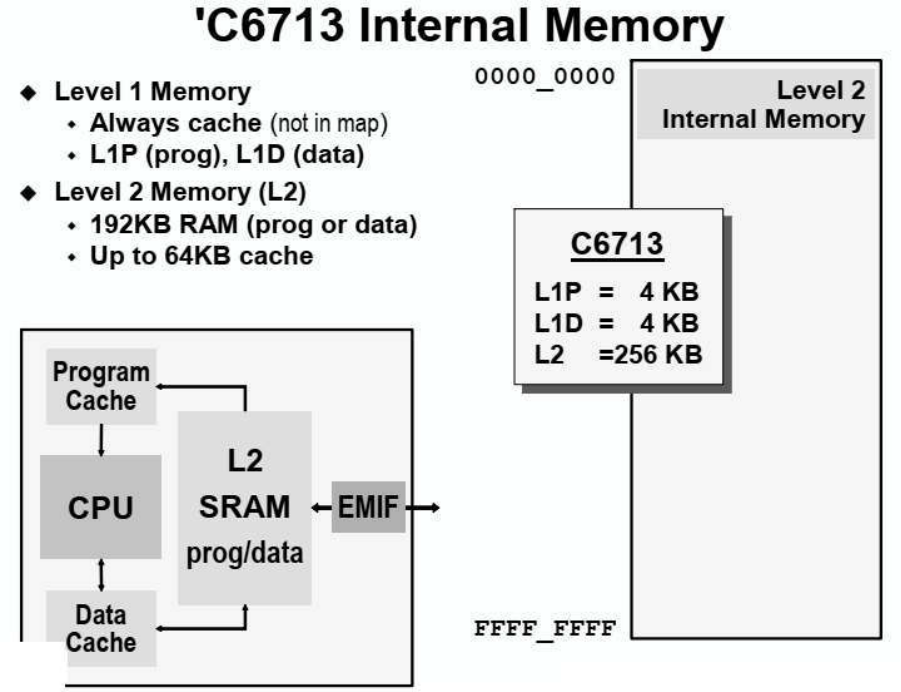


FIG. 4.1 : Mémoire interne

4.3 Mémoire externe du C6713

Le C6713 a un EMIF avec quatre plages externes. Chaque gamme a un CE dédié. Les adresses mémoire qui se trouvent en dehors des plages ne sont pas utilisées.

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 la SDRAM (Synchronous DRAM). La mémoire synchrone nécessite une horloge, contrairement à la mémoire asynchrone, qui n'en nécessite pas.

Les plages de mémoire externe CE0, CE1, CE2 et CE3 prennent en charge la mémoire synchrone (SBSRAM, SDRAM) ou asynchrone (SRAM, ROM, etc.), accessible sous forme d'octets (8 bits), de demi-mots (16 bits) ou de mots (32 morceaux). 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 accédées à partir de la même banque dans une instruction, un conflit de mémoire se produit et le CPU est bloqué d'un cycle.

Si un programme tient dans on-chip (sur puce) ou la mémoire interne, il doit être exécuté à partir de là pour éviter les retards associés à l'accès à la mémoire externe ou off-chip. Si un programme est trop volumineux pour être inséré dans la mémoire interne, la plupart de ses parties chronophages doivent être placées dans la mémoire interne pour une exécution efficace. Pour les codes répétitifs, il est recommandé que la mémoire interne soit configurée comme mémoire cache. Cela permet d'accéder le moins possible à la mémoire externe et donc d'éviter les délais associés à de tels accès.

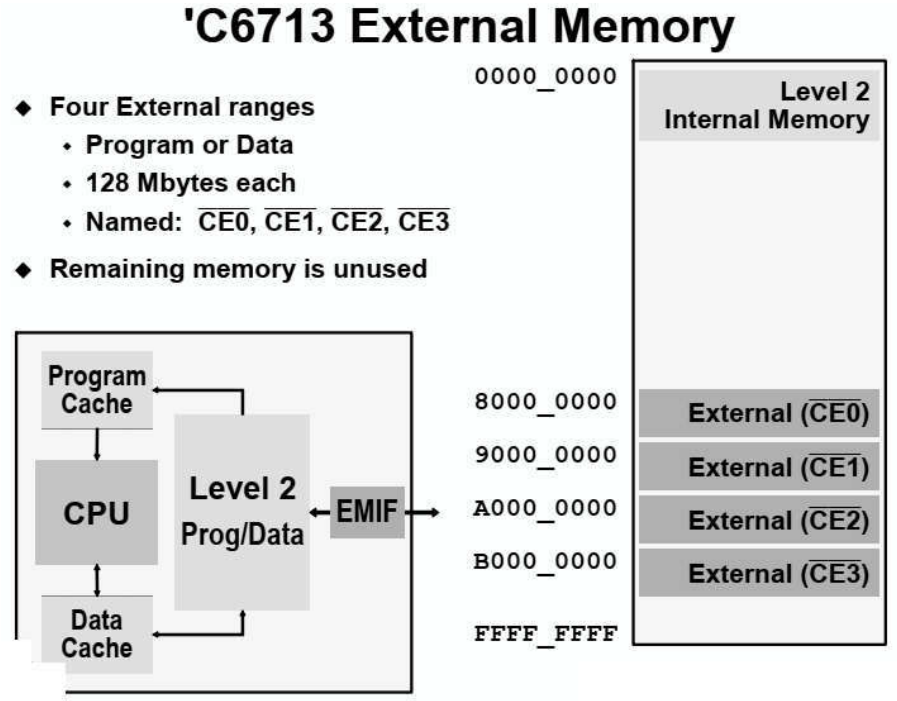


Fig. 4.2 : Mémoire externe

4.4 Mémoire du DSK C6713

Voici un schéma fonctionnel de la mémoire (interne et externe) disponible sur le C6713 DSK.(figure 4.3).

Il existe les ressources mémoire suivantes dans TMS320C6713 DSK : — mémoires internes du processeur (4 Ko de mémoire cache de programme de niveau I, 4 Ko de mémoire cache de données de niveau I, 64/192 Ko de mémoire cache de programme/de mémoire de données/SRAM de niveau II) ; — mémoires externes (512 Ko FLASH, 16 Mo SDRAM).

Dans la figure 4.4 la carte mémoire pour le C6713 DSK. Cela montre la mémoire totale disponible d'un C6713 et comment cette mémoire a été utilisée sur le DSK.

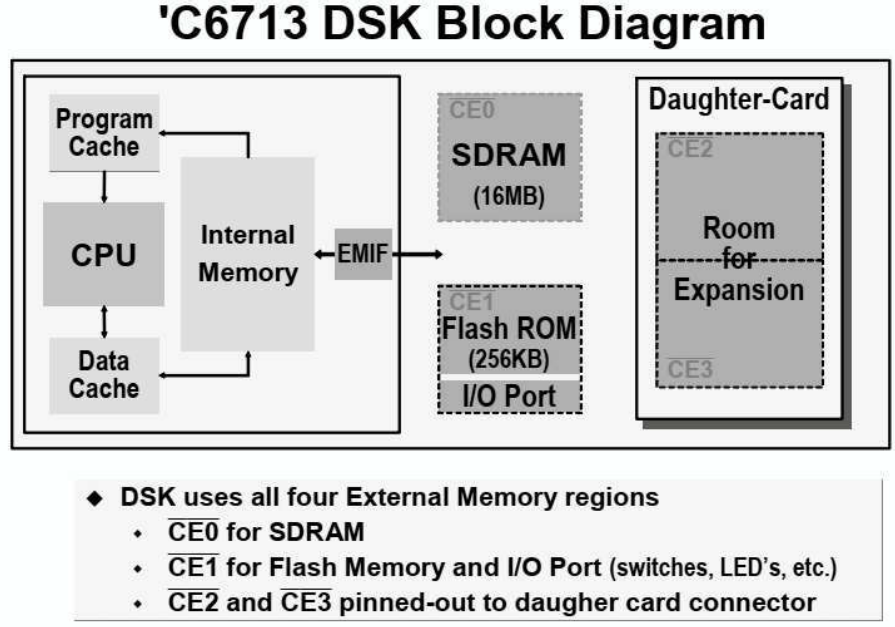


Fig. 4.3 : Mémoire du TMS320C6713 DSK

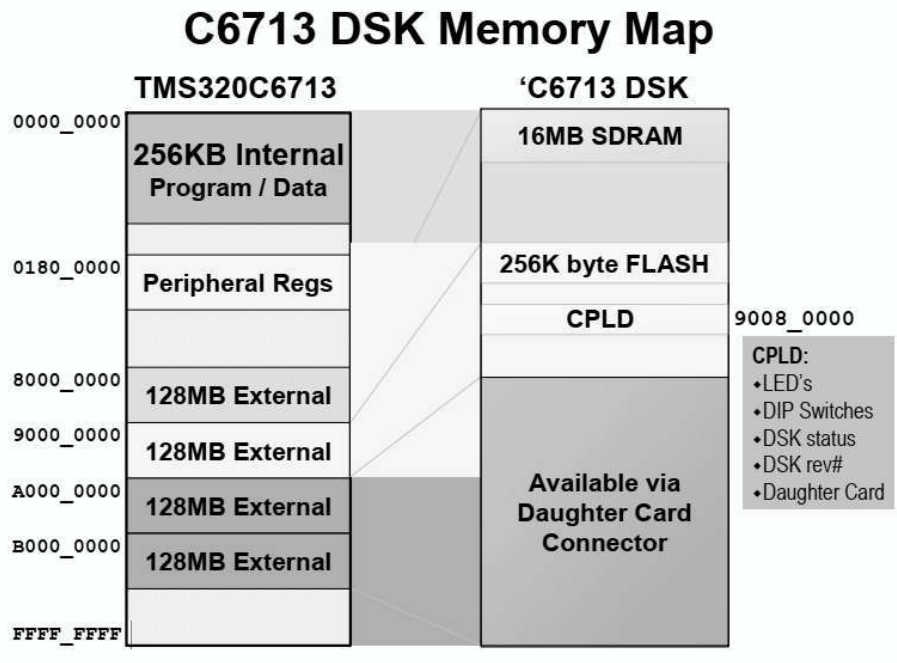
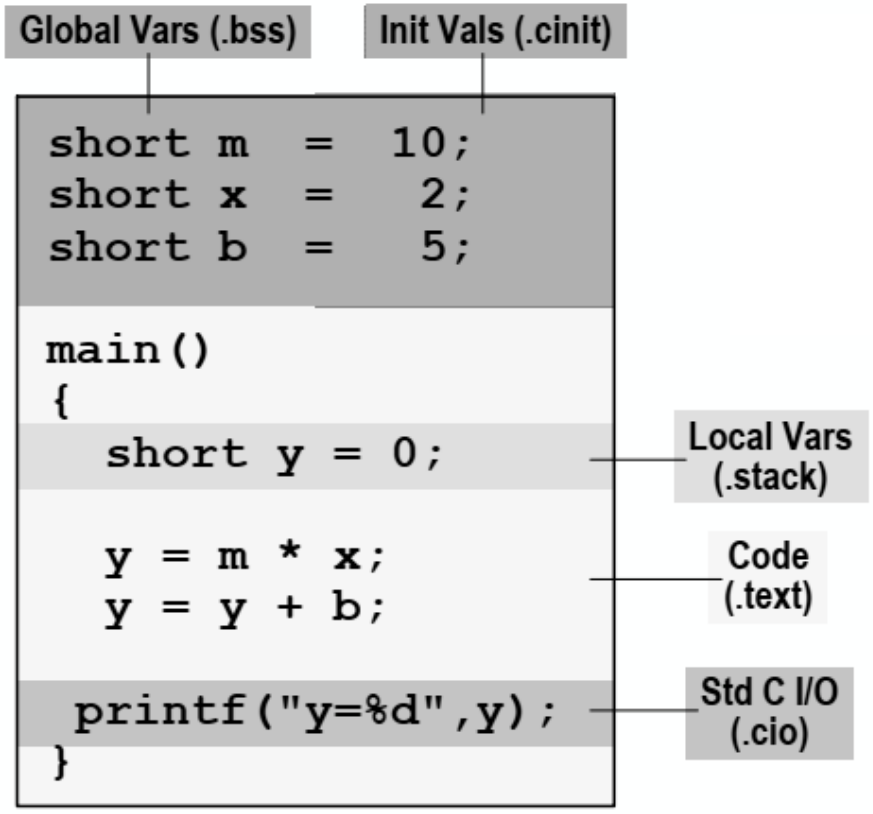


Fig. 4.4 : Cartographie mémoire du TMS320C6713 DSK

4.5 Section → Placement de la mémoire

Qu'est-ce qu'une section ?

En regardant un programme C, vous remarquerez qu'il contient à la fois du code et différents types de données (globales, locales, etc.).



- Chaque programme C se compose de différentes parties appelées sections
- Tous les noms de section par défaut commencent par "."

Dans les outils de génération de code TI (comme pour tout ensemble d'outils basé sur le COFF - Common Object File Format), ces différentes parties d'un programme sont appelées Sections. La division du code du programme et des données en différentes sections offre une flexibilité car elle vous permet de placer des sections de code dans la ROM et des variables dans la RAM. Le schéma précédent illustre cinq sections :

- Variables globales
- Valeurs initiales pour les variables globales
- Variables locales (c'est-à-dire la pile)
- Code (les instructions réelles)
- Fonctions d'E/S standard

Cependant, ce ne sont pas toutes les sections ventilées par le compilateur du C6000...

Passons en revue les noms des sections du compilateur

Voici une liste des sections créées par le compilateur. En plus de leur description, nous fournissons le nom de section défini par le compilateur.

Compiler's Section Names

Section Name	Description	Memory Type
.text	Code	initialized
.switch	Tables for switch instructions	initialized
.const	Global and static string literals	initialized
.cinit	Initial values for global/static vars	initialized
.pinit	Initial values for C++ constructors	initialized
.bss	Global and static variables	uninitialized
.far	Global and static variables	uninitialized
.stack	Stack (local variables)	uninitialized
.sysmem	Memory for malloc fcns (heap)	uninitialized
.cio	Buffers for stdio functions	uninitialized

Si vous pensez que certains de ces noms sont un peu ésotériques, nous sommes d'accord avec vous. (.code aurait peut-être eu plus de sens que .text, mais nous devons vivre avec les noms qu'ils ont choisis.)

Vous devez lier (placer) ces sections aux zones de mémoire appropriées comme indiqué ci-dessus. En termes plus simples, initialisé peut être considéré comme une mémoire de type ROM et non initialisé comme une mémoire de type RAM.

4.5.1 Coff fichier cible

coff est un format de fichier exécutable binaire populaire, avec l'extension .out dans CCS v5, vous pouvez directement télécharger sur la puce. Les fichiers exécutables incluent les en-têtes de section, le code exécutable, les données d'initialisation et les informations réadressables, ainsi que la table des chaînes de la table des symboles et d'autres informations.

Le processus de la section de traitement du compilateur est :

(1) Compilez chaque fichier source dans un fichier objet indépendant (.obj), chaque fichier objet a son propre segment

(2) L'éditeur de liens connecte les parties avec le même nom de segment dans le fichier cible pour générer le fichier exécutable final Coff Fonction de complétion des fichiers de compilation (1) et Fonction de complétion de construction (2) dans CCS v5.

4.5.2 Segment de code personnalisé et segment de données

// Attribuer un symbole au segment de données indiqué par le nom de la section

pragma DATA_SECTION(symbole, "nom de la section") ;

// Attribuer un symbole au segment de code indiqué par le nom de la section

pragma CODE_SECTION(symbolc, ”nom dc la scction”) ;

Le symbole est souvent défini en combinaison avec la structure, comme suit, struct volatile Symbol symbol ; // Structure prédéfinie de symboles décrivant des périphériques spécifiques

Par exemple, pour le périphérique Timer0 en C6713, faites la définition suivante,

```
struct Timer0 {
    ...
}
#pragma DATA_SECTION(C6713_Timer0, "C6713_Timer0_cmd");
volatile struct Timer0 C6713_Timer0;
```

”C6713_Timer0_cmd” allouera de l’espace dans le fichier cmd.

4.5.3 Fichier cmd

Le fichier cmd est principalement utilisé pour compléter la fonction de lien, vous pouvez donc utiliser la commande de lien dans le fichier cmd, par exemple :

-stack 0x200 définit la taille de la pile à 0x200 octets

-heap 0x200 définit la taille du tas sur 0x200 octets

-l rst67xx.lib lien bibliothèque rst67xx.lib

En plus de la commande link, le fichier cmd comprend également deux parties, MEMORY et SECTOINS, qui sont respectivement utilisées pour la division de la zone de stockage et l’allocation des segments.

Le format de la division MEMORY est :

```
L2SRAM    : o = 00000000h l = 00030000h  /* L2 SRAM 192K */
```

o indique l’adresse de départ, l indique la longueur de la zone de stockage (en octets)

Un exemple simple (TMS320C6713 par exemple, différentes puces sont différentes), les périphériques ajoutent uniquement Timer0:

```
MEMORY
{
        L2SRAM    : o = 00000000h l = 00030000h  /* L2 SRAM 192K */
        L2CACHE   : o = 00030000h l = 00010000h  /* L2 Cache 64 K */


        /* Peripheral */
        CPU_TIMER0 : o = 01940000h l = 00040000 /* Timer0 */


        EXTERNAL  : o = 80000000h l = 80010000h
}


SECTIONS
{
        /* Allocate program areas */
        .text     > L2SRAM        /* code segment */
        .cinit    > L2SRAM        /* init segment */
```

```
{
        /* Allocate program areas */
        .text     > L2SRAM        /* code segment */
        .cinit    > L2SRAM        /* init segment */


        /* Allocate data areas */
        .stack    > L2SRAM
        .far      > L2SRAM
        .switch   > L2SRAM        /* C switch table */
        .tables   > L2SRAM
        .data     > L2SRAM        /* data segment */
        .bss      > L2SRAM        /* data that haven't init */
        .sysmem   > L2SRAM
        .const    > L2SRAM        /* string, const ... */
        .cio      > L2SRAM


        .buffers  > EXTERNAL


        C6713_Timer0_cmd > CPU_TIMER0  /* Timer 0 */
}
```

Le fichier cmd comprend 2 parties-MEMORY et SECTIONS MEMORY termine la division de l'espace d'adressage ; SECTIONS complète l'allocation de l'espace d'adressage à des usages spécifiques (en plus des sections générales du programme, il peut également y avoir des sections personnalisées).

Remarques : En développement normal, le programme est téléchargé dans l'espace RAM, et lorsqu'il est publié, il doit être téléchargé dans l'espace Flash. C'est le fichier cmd de SRAM, et le fichier cmd de Flash est différent.

Ce qui suit est le cmd du DSK de TI, vous pouvez vous y référer directement,

Chapitre 5

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

5.1 Introduction

L'environnement de programmation des DSPs de Texas Instruments est 'the Code Composer Studio (CCS)'.

Ce logiciel contient des outils de développement, y compris un compilateur de haute optimisation de C/C++, émulation basée sur le JTAG, débogueur à temps réel (realtime debugging).

Le Studio Code Composer est un compilateur conçu exclusivement pour les C6x et C5x, développé par Texas Instrument.

Il s'agit d'un environnement complet avec un éditeur, une aide à la création d'un projet, un compilateur croisé (''cross-compiler'', un compilateur produisant un exécutable pour une autre architecture que celle sur laquelle la compilation se fait), un émulateur qui exécute le programme cible en simulant le fonctionnement du DSP, des outils de débogage et de traçage, et un loader (qui charge l'application en mémoire RAM ou FLASH du DSP) etc....

L'application CCS fournit tous les outils Software nécessaires pour le développement du DSP.

Au cœur de CCS, il existe l'original compositeur IDE. L'IDE fournit une seule fenêtre d'application dans laquelle est exécuté le code de développement ; d'entrée et d'édition du code de programme, la compilation et construction d'un fichier exécutable ''building '', et le débogage du code de programme. Dans ce chapitre on s'intéresse à l'environnement CCS V3.3.

5.2 Configuration de base 'Basic Setup'

Le logiciel Code Composer Studio (CCS) permet toutes les opérations concernant la programmation, la construction (building) et le débogage des DSPs de Texas Instruments. Il communique avec la carte DSKC6713 via l'interface JTAG intégré et peut échanger avec cette carte des données en temps réel.

Avant de lancer CCS, on doit sélectionner la plate-forme matérielle à utiliser à partir de l'interface de configuration. Cette interface est accessible après avoir cliqué sur l'icône du programme d'installation CCS (semblable à la figure 5.1) sur le bureau de l'ordinateur.

Fig. 5.1 : Icone raccourci du CCS Setup

5.2.1 Mode de simulation

Pour utiliser le CCS V3.3 en mode simulateur C6713, sélectionnez 'C6713 Device Cycle Accurate Simulator' de la liste des cartes. Il doit être ajouté et montré sur le côté gauche de l'écran de configuration comme illustré dans la Figure 5.2.

Pour modifier les propriétés de la mémoire, on clique sur l'icône "TMS320C6713" comme le montre la figure 5.2 avec la flèche rouge et sélectionnez " Properties ". La propriété "Detect Reserved Memory Access" doit être sélectionnée sur "No" comme montré dans la figure 5.3.

Appuyez sur "OK" pour revenir à l'écran de configuration. Appuyez sur "Save Quit". CCS est prêt à fonctionner en mode simulateur C6713.

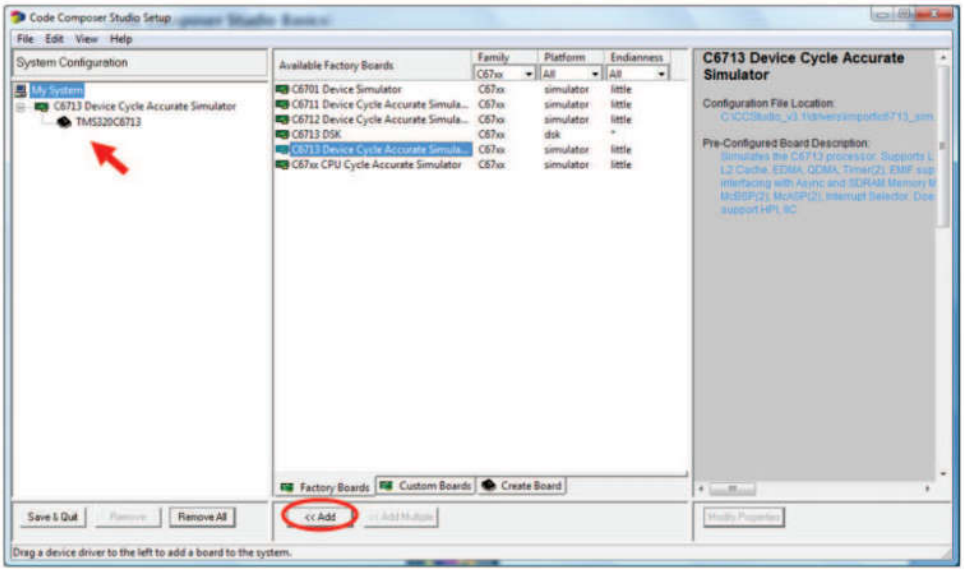


FIG. 5.2 : Ecran setup du CCS

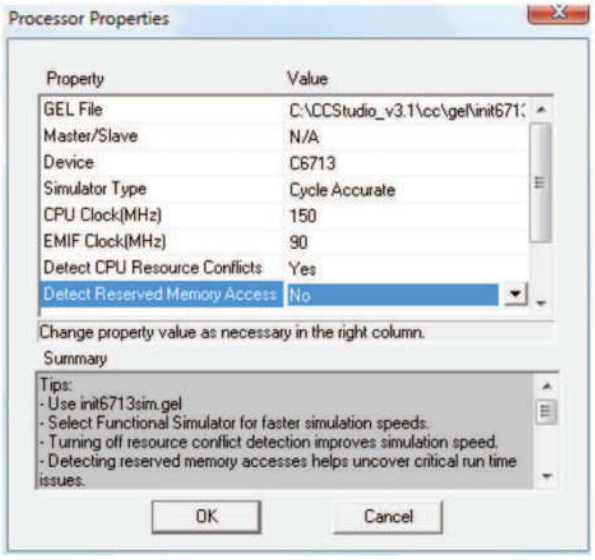


FIG. 5.3 : Propriétés du processeur

5.2.2 La carte C6713 DSK

Si on veut utiliser le logiciel CCS pour programmer le kit DSK C6713, alors on doit choisir le "C6713 DSK" de la liste des cartes de l'écran de configuration. On appuie ensuite sur le bouton "Add". On doit voir "C6713 DSK" sur le côté gauche de l'écran de configuration comme indiqué dans la Figure 5.4. On appuie sur "Save Quit". CCS est maintenant prêt à programmer et exécuter les programmes sur le kit DSK C6713.
Pour démarrer CCS, cliquez sur l'icône CCS (semblable à la figure 5.5).

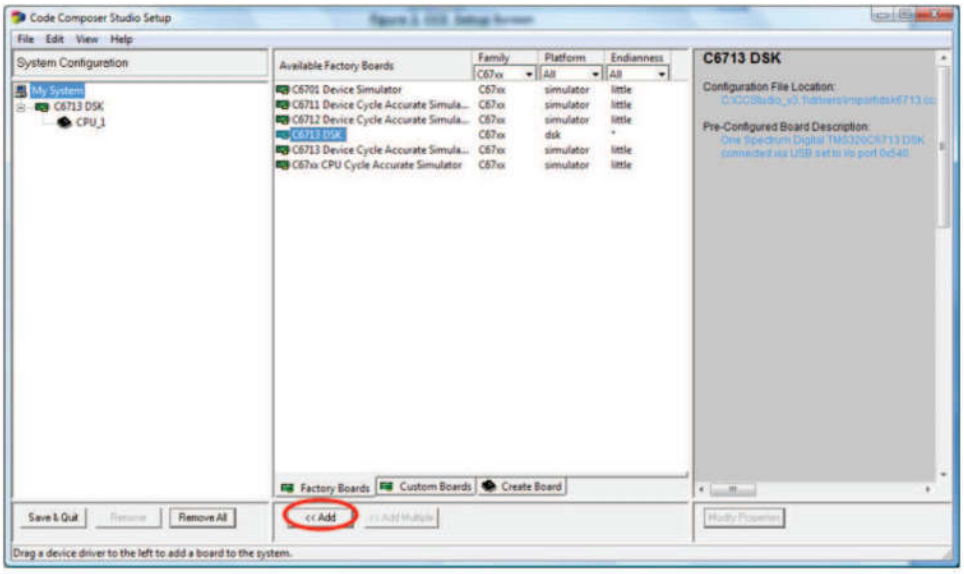


Fig. 5.4 : Sélection du DSK C6713

Fig. 5.5 : Icone raccourci du CCS

5.3 Création d'un nouveau projet sous CCS

On doit appuyer sur "project" et "New" pour créer un nouveau projet dans CCS. Pour nommer le projet, entrez l'onglet "Project Name", comme dans la Figure 5.6. Lorsqu'on clique sur "Finish", un projet vide sera créé.

5.3.1 Configuration bios

Pour exécuter des programmes sur le kit DSK C6713, on doit régler les paramètres de base dans l'ordre correct. Cette configuration concerne la mémoire, la gestion des interruptions, le contrôle des entrées/sorties, etc... Pour gérer cette configuration, un système

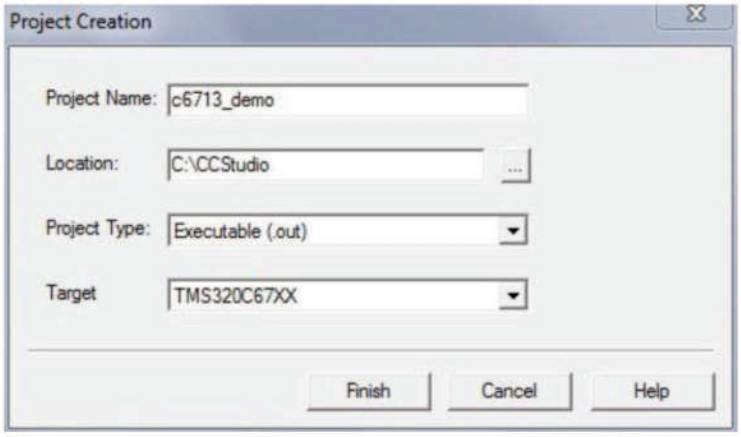


FIG. 5.6 : Exemple de création de projet sous CCS

d'exploitation en temps réel appelé DSP/BIOS est utilisé. Dans le projet, le système d'exploitation et le programme (à exécuter) sont intégrés ensemble au DSP. DSP/BIOS gère les opérations nécessaires à l'exécution du programme en cours. DSP/BIOS est configuré en utilisant le fichier de configuration "xxx.cdb". Pour ouvrir un nouveau fichier de configuration DSP/BIOS, on appuie sur "File→New→DSP/BIOS Configuration..." et on sélectionne le "dsk6713.cdb" comme montré dans la Figure 5.7.

Si CCS est utilisé en mode simulateur, on clique à droite sur les "RTDX-Real-Time Data

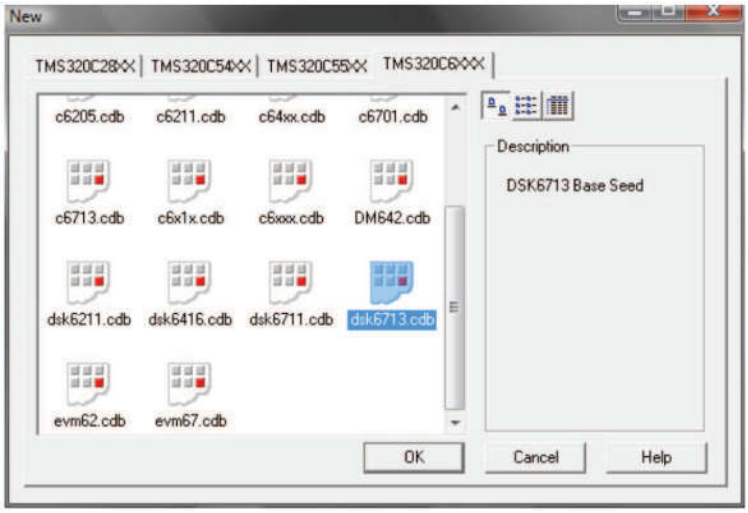


FIG. 5.7 : Configuration DSP/BIOS

Exchange Settings", comme montré dans la figure 5.8 et on sélectionne ses propriétés. Il faut aussi changer le mode RTDX à "Simulator", comme illustré dans la figure 5.9 pour fonctionner en mode simulateur.

Le fichier de configuration doit être enregistré par "File → Save : c6713_demo.cdb". Maintenant, un nouveau fichier de configuration est créé. Cependant, il n'est pas encore ajouté au projet. Il peut être ajouté au projet par "Project → Add Files to Project..." et en sélectionnant le fichier "c6713_demo.cdb" comme montré dans la Figure 5.10.

Dans l'arborescence du projet, les fichiers de configuration qui sont créés automatiquement par DSP/BIOS doivent être visibles comme montré dans la figure 5.11.

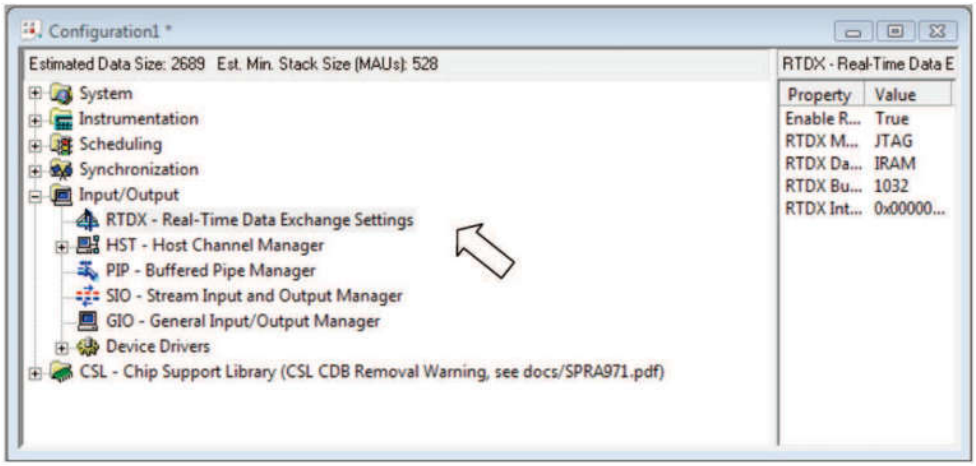


FIG. 5.8 : RTDX-Real-Time Data Exchange settings

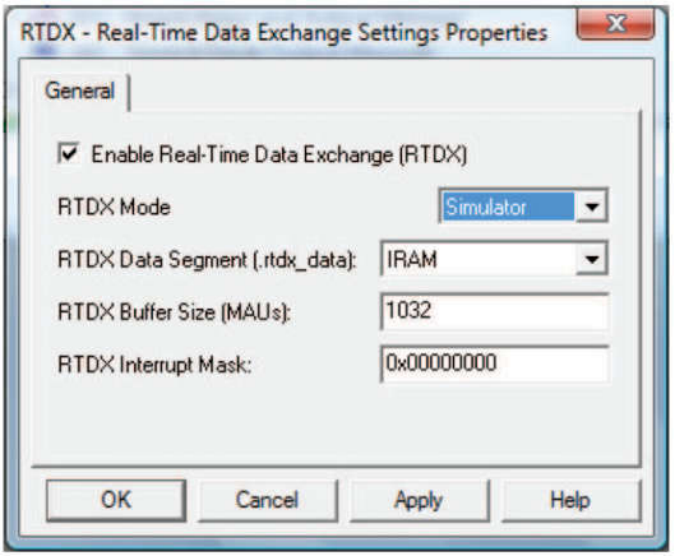


FIG. 5.9 : Sélection du mode simulateur

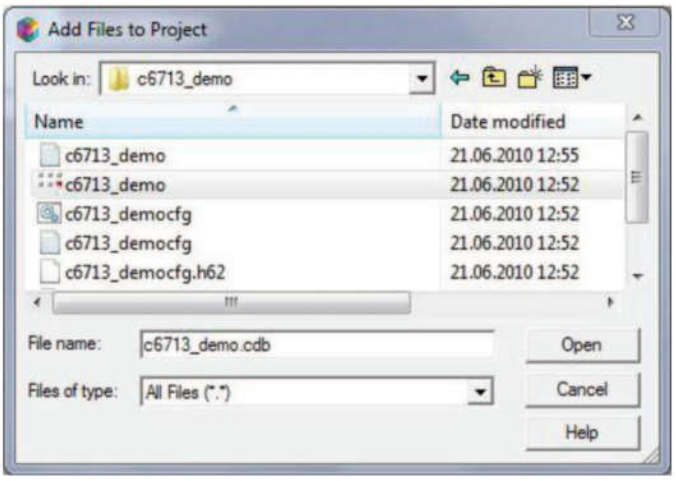


FIG. 5.10 : Ajout de fichier cbd au projet

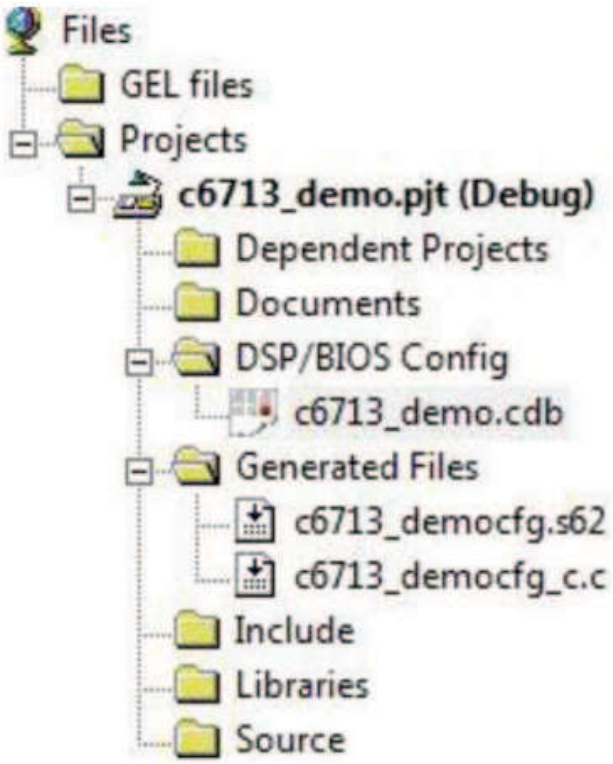


FIG. 5.11 : L'arborescence du projet

5.3.2 Création de programme source

Pour créer un code source, on doit suivre les étapes du menu "File → New → Source File". La figure 5.12 montre un exemple d'un programme source simple. On doit enregistrer ce code en utilisant "File → Save c6713_demo.c".

```
void main (void)
{
    int a[5]={1,2,3,4,5};
    int b[5]={1,2,3,4,5};
    int c[5];
    int i;

    for(i=0;i<5;i++)
    {
        c[i]=a[i]+b[i];
    }

}
```

FIG. 5.12 : Premier programme

Encore une fois, on doit ajouter ce fichier au projet en utilisant "Project → Add Files to Project...". Ce fichier doit être visible dans l'arborescence du projet comme montré dans

la Figure 5.13.

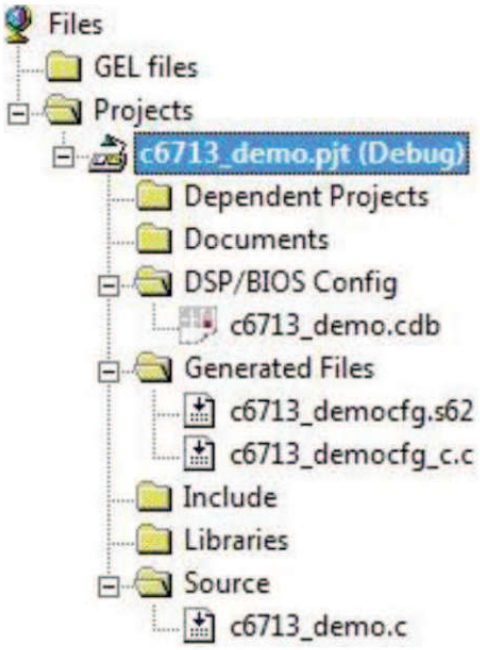


FIG. 5.13 : Ajout de code source au projet

5.4 Exécution du programme

Ce premier code est prêt à être compilé et exécuté. Pour compiler le code, suivre les étapes "Project → Build". S'il n'y a pas d'erreurs dans le code, CCS génère un message semblable à celui de la figure 5.14.

```
Build Complete,
  0 Errors, 2 Warnings, 0 Remarks.
```

FIG. 5.14 : Compilation réussie (Successful code building)

Pour exécuter un programme, on doit suivre les étapes suivantes :

File → Load Program ...: Sélectionner "experiment2_demo.out" sous le répertoire debug.

Debug → Reset CPU

Debug → Restart

Debug → Go Main

Debug → Run

5.4.1 Insertion de point d'arrêt (Break Point)

Pour insérer un point d'arrêt dans le programme, vous devez déplacer le curseur à l'endroit désiré. Cliquez sur l'icône en haut de l'écran comme illustré dans la Figure 5.15.

FIG. 5.15 : Insertion de point d'arrêt (breakpoint)

5.4.2 Ajout d'une fenêtre de surveillance (Watch Window)

Pour surveiller ou modifier les valeurs des variables, on peut utiliser la fenêtre de surveillance. Pour ajouter une variable à la fenêtre de surveillance, on fait un clic droit sur la variable et on sélectionne "Add to watch window" comme indiqué dans la Figure 5.16.

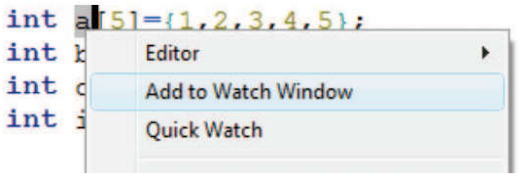
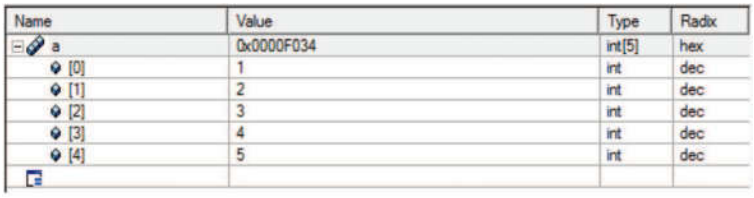


FIG. 5.16 : Ajout de fenêtre de surveillance

On peut visualiser les valeurs et les changements dans la fenêtre de surveillance en bas à droite de l'écran du logiciel CCS, voir la Figure 5.17.



Name	Value	Type	Radix
a	0x0000F034	int[5]	hex
[0]	1	int	dec
[1]	2	int	dec
[2]	3	int	dec
[3]	4	int	dec
[4]	5	int	dec

FIG. 5.17 : Visualisation et changement des valeurs dans une fenêtre de surveillance

5.5 Plots

Afin de visualiser un vecteur sous forme de graphe, on utilise "View → Graph → Time/Frequency...". Sur la fenêtre pop-up (voir Figure 5.18), remplir :

Start Address : nom du vecteur à tracer.

Acquisition Buffer Size : taille du vecteur.

Display Data Size : taille des données à afficher sur le graphe.

DSP Data Type : type du vecteur (integer ou float).

Sampling Rate : fréquence d'échantillonnage du signal dans le vecteur.

Par exemple, pour visualiser le vecteur 'c' dans le programme précédant, on règle les paramètres comme montré dans la figure 5.18. Le graphe à obtenir est semblable au graphe de la figure 5.19.

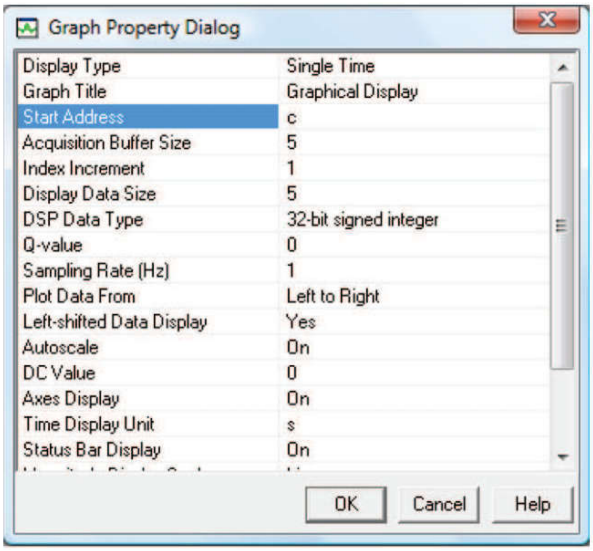


FIG. 5.18 : Réglage des propriétés du graphe

5.6 Images

Pour visualiser une image sous CCS on doit aller à "View→ Graph → image". On remplit les paramètres de la fenêtre pop-up comme indiqué dans la figure 5.20.

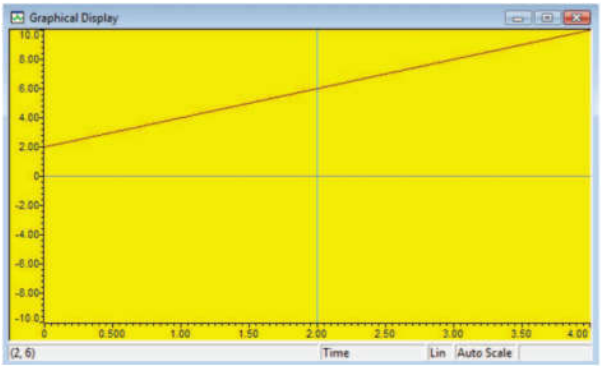


FIG. 5.19 : Graphe

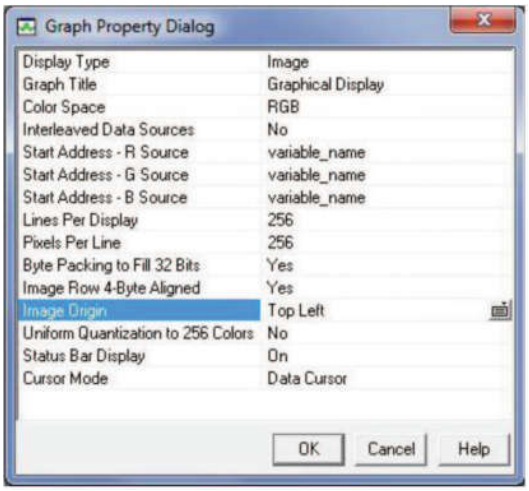


FIG. 5.20 : Paramètres de visualisation d'images

5.6.1 Enregistrement de données

Pour sauvegarder les matrices dans un fichier (data file), on doit utiliser "File → Data → Save". Dans la figure 5.21, on choisit le format d'enregistrements des données.

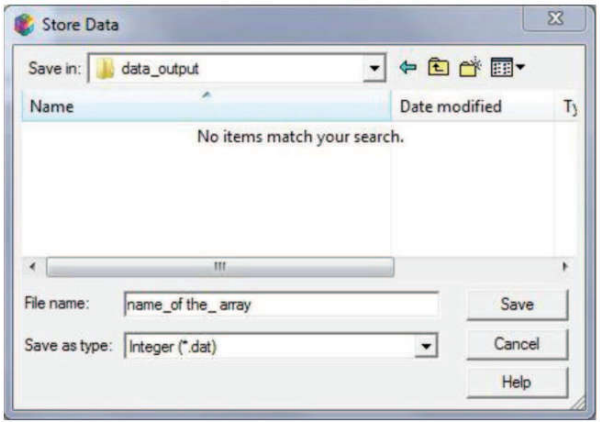


FIG. 5.21 : Enregistrement de data-file

Une fois le bouton "save" est actionner, une nouvelle fenêtre apprêtera (figure 5.22). Il faut mettre le nom de la matrice dans le champ "Address" et sa taille dans le champ "length". CCS enregistrera les données dans un fichier ".dat".

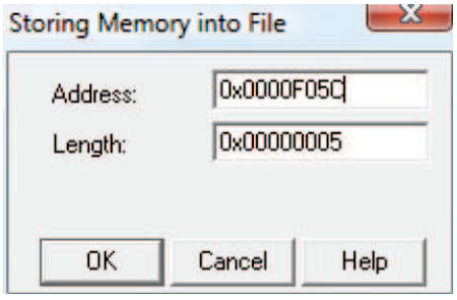


FIG. 5.22 : Enregistrement de variables

5.7 Scriptes GEL (General Extension Language) du CCS

CCS peut être personnalisé en utilisant les scriptes GEL. GEL est un langage interprété qui permet l'écriture de fonctions pour configurer l'environnement de développement intégré et d'accéder au processeur cible.

Ci-dessous est un fichier exemple de script GEL :

```
menuitem "Gain Slider";
slider FilterGain1(1, 4, 1, 1, volume1)
{
/* initialize the target variable with the Parameters passed by the slider object. */
gain1 = volume1;
}
slider FilterGain2(1, 4, 1, 1, volume2) {
/* initialize the target variable with the Parameters passed by the slider object. */
gain2 = volume2;
}
slider FilterGain3(1, 4, 1, 1, volume3) {
/* initialize the target variable with the Parameters passed by the slider object. */
gain3 = volume3;
}
```

Pour charger un fichier GEL, on sélectionne "File > Load GEL..." comme montré dans la figure 5.23. Charger le fichier "volume_slider.gel".

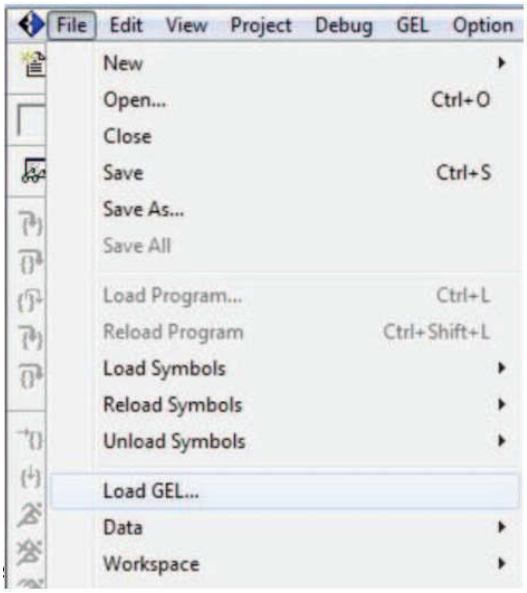


FIG. 5.23 : Chargement de fichier Gel

Pour ouvrir les curseurs des volumes, il faut sélectionner "GEL> Gain Slider > Filter-

Gain1". La figure 5.24 montre un chargement de trois curseurs

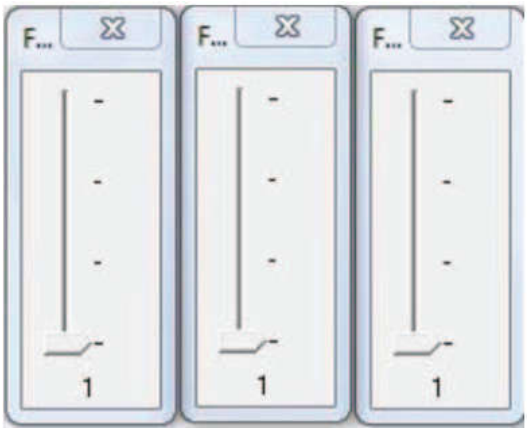


FIG. 5.24 : Chargement de curseurs par un fichier Gel

5.8 Utilisation des switches DIP et des LEDs

Le DSK C6713 possède quatre switches et quatre LEDs.

Afin de pouvoir les manipuler, les fichiers "dsk6713_dip.h" et "dsk6713_led.h" doivent être inclus dans le programme. On sélectionne "Project → Build options → compiler", puis "processor". Sous l'onglet "compiler", champ "include search path", on spécifie le chemin des deux fichiers "C : composer studio setup folder/C6000/dsk6713/include" , comme montré sur la figure 5.25.

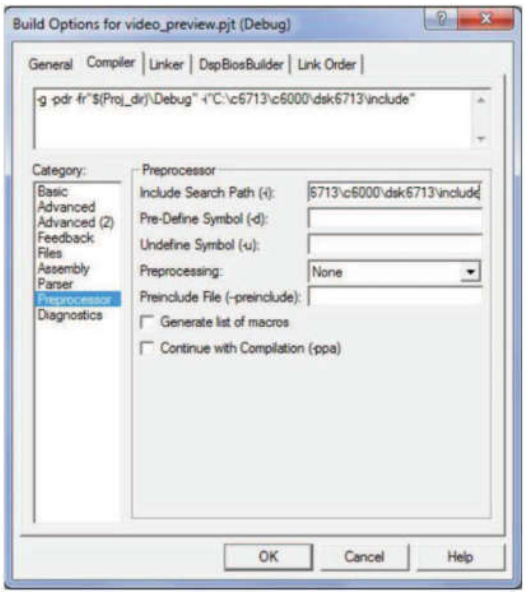


FIG. 5.25 : Spécification du chemin pour chercher des fichiers nécessaires au programme

On doit aussi ajouter le fichier "dsk6713bsl.lib" qui se trouve sous « C :/code composer studio setup folder /C6000/dsk6713/lib » en utilisant l'option "Add Files to Project", comme montré sur la figure 5.26. Maintenant, les DIP switches et LEDs du DSK C6713

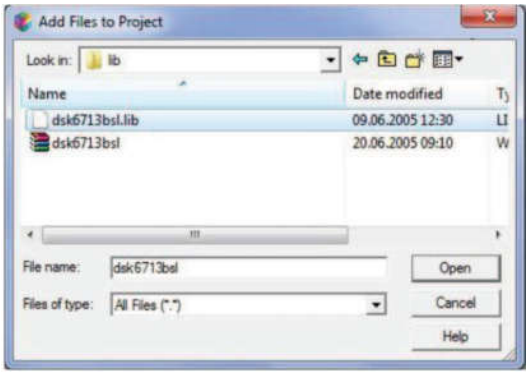


FIG. 5.26 : Ajout du fichier dsk6713bsl.lib au projet

peuvent être utilisés. Le programme suivant est un exemple qui illustre l'utilisation des DIP switches et des LEDs.

```
/* Read the status of the DIP switches and control LEDs*/
/* All header files starting with dsk6713 belong to the board support
library */
include<dsk6713.h> /* General functions */
include<dsk6713_dip.h> /* Functions for DIP */
include<dsk6713_led.h> /* Functions for LED */
void main()
{
DSK6713_init();
DSK6713_DIP_init(); /*Initialize DIP switches*/
DSK6713_LED_init(); /*Initialize LEDs*/
while(1){
/*DSK6713_DIP_get('switch number'); Retrieve the DIP switch value*
/
if(DSK6713_DIP_get(1) == 1){
/*DSK6713_LED_on('LED number'); SettheLED*/
DSK6713_LED_on(1);
}else{
/*DSK6713_LED_off('LED number'); Clear the LED*/
DSK6713_LED_off(1);
}
}
}
```

Chapitre 6

Algorithmes de traitement du signal sur DSP

6.1 Introduction

Les algorithmes de traitement numérique du signal sont généralement construits à partir de trois fonctions de base : **Add**, **Multiply** et **Delay**. Les fonctions sont appliquées en combinaison pour construire des algorithmes complexes dans les systèmes à temps discret . Les fonctions Multiply et Add sont appelées opérations. Les besoins de performances d'un algorithme DSP particulier sont généralement indiqués en termes de nombre d'opérations que l'algorithme nécessite pour s'exécuter à la fréquence d'échantillonnage requise. Le nombre total d'opérations correspond simplement aux opérations requises pour un seul échantillon multipliées par la fréquence d'échantillonnage fe.

La première fonction est un additionneur ; il ajoute des échantillons de deux ou plusieurs signaux temporels discrets. Des échantillons sont ajoutés pour toutes les valeurs de n, comme le montre la figure 6.1.

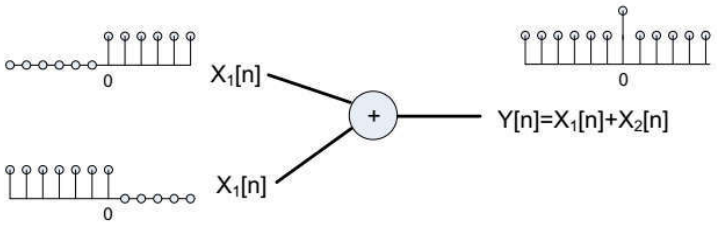


Fig. 6.1 : Fonction Add

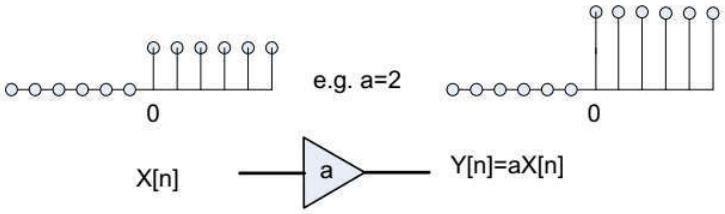


Fig. 6.2 : Fonction Multiply

La deuxième fonction est le multiplicateur ; le bloc de multiplication met à l'échelle tous les échantillons du signal de temps discret selon le même facteur d'échelle, comme illustré à la figure 6.2.

Le troisième élément clé pour un algorithme DSP est une fonction de retard. La fonction de retard décale les échantillons vers la droite. Le retard est symboliquement défini comme Z^{-1} pour un retard de 1, Z^{-2} pour un retard de 2, et ainsi de suite. La figure 6.3 montre la fonction de retard.

Un système de temps discret avec des combinaisons de multiplication, d'addition et de retard peut être décrit comme une équation de différence, telle que :

y[n] = a.x[n] + b.x[n-1] + c.x[n-2]

Cette équation est symboliquement représentée comme le montre la figure 6.4.

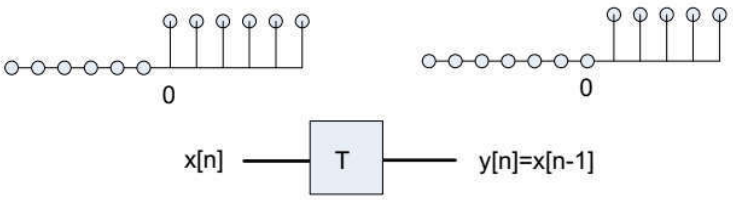


FIG. 6.3 : Fonction Delay

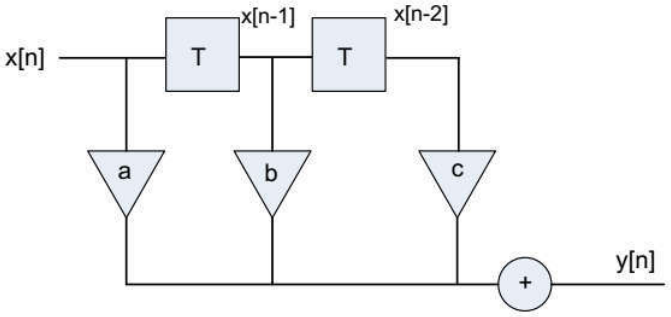


FIG. 6.4 : Equation de différence

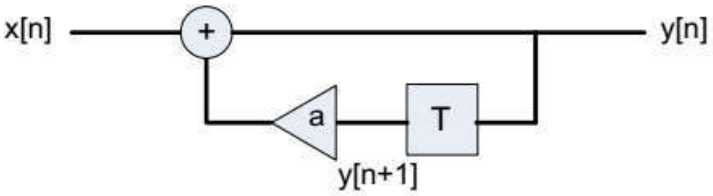


FIG. 6.5 : le système de rétroaction

Ce système illustré à la figure 6.4 est anticipé. Comme le système n'utilise que des échantillons d'entrée, il est défini comme un système statique ou sans mémoire. Une alternative, le système de rétroaction, est illustrée à la Figure 6.5.

La combinaison de ces deux constructions de système peut être utilisée selon les besoins pour développer toute équation de différence discrète.

Pour un système temporel discret, le signal d'entrée peut être représenté par une série de signaux d'impulsion pondérés décalés dans le temps. La réponse d'un système (sortie) au signal d'impulsion donne une caractérisation complète du système. La réponse à un signal d'entrée arbitraire x [n] est calculée comme une convolution du signal d'entrée (représentée comme une série de signaux impulsionnels retardés mis à l'échelle) et la réponse impulsionnelle du système.

6.2 L'adéquation algorithme-architecture

La communauté du traitement du signal et des images s'est toujours intéressée à l'implantation des algorithmes qu'elle concevait . L'évolution rapide des architectures des processeurs, des circuits intégrés spécifiques et des machines construites avec ces com-

posants, ainsi que l'évolution des outils logiciels d'aide à l'implantation, ont permis de réaliser à des coûts raisonnables certaines applications complexes que l'on n'osait même pas envisager il y a quelques années . On est peu à peu passé d'une étude séparée des algorithmes et des architectures, à une approche globale plus formalisée, prenant en compte les deux aspects simultanément . Ceci permet de réaliser de meilleures implantations, tenant compte pleinement de l'accroissement rapide d'une part des progrès technologiques tels que la vitesse des circuits électroniques et leur densité d'intégration, et d'autre part des nouvelles fonctionnalités introduites dans ces circuits par les fondeurs de silicium.

L'Adéquation Algorithme Architecture consiste à étudier en même temps les aspects algorithmiques et architecturaux en prenant en compte leurs interactions, en vue d'effectuer une implantation optimisée de l'algorithme (minimisation des composants logiciels et matériels) tout en réduisant les temps de développement et les coûts finaux de l'application étudiée . L'adéquation est un processus réciproque de mise en correspondance de l'algorithme et de l'architecture . Elle doit être basée sur une formalisation au niveau système qui peut être unifiée, des algorithmes, des architectures et des implantations, prenant en compte les contraintes (temps-réel, embarquabilité . . .), le besoin en puissance de calcul et la nature distribuée des informations à traiter (capteurs et actionneurs multiples, données distribuées . . .) ; ces deux derniers points conduisent à étudier et maîtriser le problème délicat du parallélisme . Enfin les aspects portabilité et réutilisation, aussi bien du matériel que du logiciel applicatif, doivent aussi être pris en considération . Elle permet aussi, d'une part d'effectuer des vérifications formelles le plus tôt possible dans le cycle de développement de l'application afin d'assurer la sécurité et la continuité de la conception et d'autre part, de poser des problèmes d'optimisation permettant de dimensionner au mieux les architectures . On peut ainsi améliorer les techniques de « prototypage rapide » et aborder de manière plus claire « la conception conjointe logicielmatériel » (co-design). Ces deux points sont des enjeux majeurs pour le futur. Afin d'en faire bénéficier les architectures utilisées lors de l'adéquation, il est indispensable de suivre l'évolution très rapide de la technologie de l'électronique numérique et analogique. Les nouvelles architectures, conçues aussi bien dans les laboratoires universitaires qu'industriels, doivent être soigneusement évaluées puis caractérisées, afin d'en tirer le maximum de performances . Enfin, les études méthodologiques et les nouvelles architectures sont validées sur des applications test tenant compte des préoccupations industrielles .

6.3 filtres numériques

Le filtrage est l'une des opérations de traitement du signal les plus utiles. Des processeurs de signaux numériques sont désormais disponibles pour implémenter des filtres numériques en temps réel. Le jeu d'instructions et l'architecture du TMS320C6x le rendent bien adapté à de telles opérations de filtrage. Un filtre analogique fonctionne sur des signaux continus et est généralement réalisé avec des composants discrets tels que des amplificateurs opérationnels, des résistances et des condensateurs. Cependant, un filtre numérique, tel qu'un filtre à réponse impulsionnelle finie (FIR), fonctionne sur des signaux à temps discret et peut être mis en œuvre avec un processeur de signal numérique tel que le TMS320C6x. Cela implique l'utilisation d'un ADC pour capturer un signal d'entrée

externe, le traitement des échantillons d'entrée et l'envoi de la sortie résultante via un DAC.

6.3.1 Convolution en temps discret et réponses en fréquence

La sortie y [n] d'un système à temps discret (LTI) linéaire, invariant dans le temps, peut être calculée en convoluant son entrée x [n] avec sa réponse impulsionnelle unitaire h [n]. L'équation de cette convolution en temps discret est :

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] = \sum_{k=-\infty}^{\infty} h[k]x[n-k] \tag{6.1}$$

La transformée en z de la convolution en temps discret de deux signaux est le produit des deux transformées, c'est-à-dire

$$Y(z) = \sum_{n=-\infty}^{\infty} y[n]z^{-n} = X(z)H(z) \tag{6.2}$$

$$H(z) = \sum_{n=-\infty}^{\infty} h[n]z^{-n} \quad and \quad X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{6.3}$$

La réponse d'un système LTI à une sinusoïde après que les transitoires sont devenus négligeables est appelée sa réponse sinusoïdale en régime permanent. Pour déterminer cette réponse, laissez l'entrée être la sinusoïde complexe échantillonnée : $x[n] = Ce^{jwnT}$

D'après (6.1), la sortie est

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]Ce^{jw(n-k)T} = Ce^{jwnT} \sum_{k=-\infty}^{\infty} h[k]e^{jwkT} = x[n]H(z)|_{z=e^{jwT}} \tag{6.4}$$

Ainsi, la sortie est une sinusoïde à la même fréquence que l'entrée mais avec son amplitude mise à l'échelle par le nombre complexe

$$H^*(w) = A(w)e^{j\theta(w)} \tag{6.5}$$

ainsi, selon (6.4), la sortie peut être exprimée comme

$$y[n] = CA(w)e^{j[wnT+\theta(w)]} \tag{6.6}$$

Lorsque l'entrée est la vraie sinusoïde

$x[n] = C\cos(wnT + \phi) = Re\{Ce^{j\phi}e^{jwnT}\}$

La sortie est

$y[n] = Re\{H^*(w)Ce^{j\phi}e^{jwnT}\} = CA(w)\cos[wnT + \theta(w) + \phi]$

En d'autres termes, le système met à l'échelle l'amplitude de l'entrée sinusoïdale par la réponse d'amplitude et décale sa phase par la réponse de phase. C'est la base du filtrage numérique.

6.3.2 Filtres à réponse impulsionnelle de durée finie (FIR)

6.3.2.1 Schéma fonctionnel pour la réalisation la plus courante

Si la réponse impulsionnelle unitaire est identique à zéro en dehors de l'ensemble des entiers 0, 1, · · ·, N - 1, la convolution (6.1) devient

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] = \sum_{k=n-N+1}^{n} x[k]h[n-k] \tag{6.7}$$

Un filtre de ce type est appelé filtre à réponse impulsionnelle à durée finie (FIR) à N prises, filtre non récursif, filtre transversal ou filtre à moyenne mobile. Un schéma fonctionnel de la méthode la plus courante de mise en œuvre des filtres FIR est illustré à la Fig. 3.6. Il se compose d'une ligne à retard représentée par la chaîne de blocs étiquetés z^{-1} et d'un ensemble de prises dans la ligne à retard avec des poids égaux aux échantillons de réponse impulsionnelle unitaire.

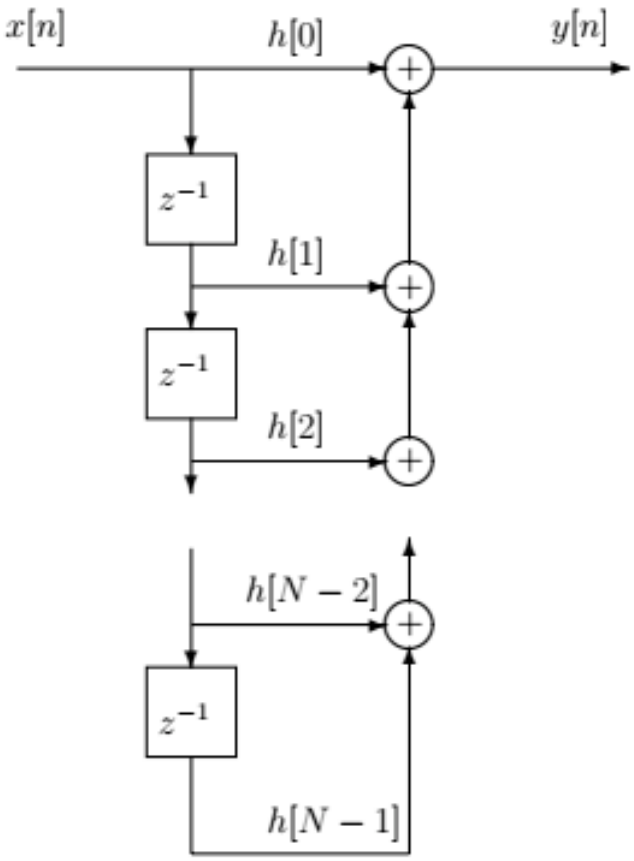


Fig. 6.6 : Réalisation directe de forme de type 1

Le schéma de principe d'un filtre FIR représenté sur la figure 6.6 pourrait représenter la disposition physique d'une implémentation matérielle ou simplement la structure d'un algorithme logiciel. Dans une implémentation logicielle, la ligne à retard serait juste un tableau en mémoire. Entrer un nouvel échantillon dans un tableau en décalant le tableau entier est inefficace.

6.3.2.2 Implémenation du filtre FIR à l'aide de la série de fourier

La conception d'un filtre FIR utilisant une méthode en série de Fourier est telle que la réponse en amplitude de sa fonction de transfert H (z) se rapproche d'une réponse en amplitude désirée. La fonction de transfert désirée est

$$H_d(w) = \sum_{n=-\infty}^{\infty} C_n e^{jnwT} \qquad |n| < \infty \tag{6.8}$$

où C_n sont les coefficients de la série de Fourier. En utilisant une variable de fréquence normalisée ν telle que $\nu = f/F_N$, où F_N est la fréquence de Nyquist, ou $F_N = F_e/2$, la fonction de transfert souhaitée dans (6.8) peut être écrite comme

$$H_d(\nu) = \sum_{n=-\infty}^{\infty} C_n e^{jn\pi\nu} \tag{6.9}$$

où $wT = 2\pi f/F_s = p$ et $|\nu| < 1$. Les coefficients C_n sont définis comme

$$\begin{aligned} C_n &= \frac{1}{2}\int_{-1}^{1} H_d(\nu) e^{-jn\pi\nu} d\nu \\ &= \frac{1}{2}\int_{-1}^{1} H_d(\nu)(\cos n\pi\nu - j\sin n\pi\nu) d\nu \end{aligned} \tag{6.10}$$

Supposons que $H_d(\nu)$ est une fonction paire (filtre sélectif en fréquence) ; puis (6.10) se réduit à

$$C_n = \int_{0}^{1} H_d(\nu)\cos n\pi\nu d\nu \qquad n \geq 0 \tag{6.11}$$

puisque $H_d(\nu)\sin n\pi\nu$sinnp est une fonction impaire et

$$\int_{-1}^{1} H_d(\nu)\sin n\pi\nu d\nu = 0$$

avec $C_n = C_{-n}$. La fonction de transfert désirée $H_d(\nu)$ dans (6.9) est exprimée en termes d'un nombre infini de coefficients, et pour obtenir un filtre réalisable, il faut tronquer (6.9), ce qui donne la fonction de transfert approximée

$$H_a(\nu) = \sum_{n=-Q}^{Q} C_n e^{jn\pi\nu} \tag{6.12}$$

où Q est positif et fini et détermine l'ordre du filtre. Plus la valeur de Q est élevée, plus l'ordre du filtre FIR est élevé et meilleure est l'approximation en (6.12) de la fonction de transfert souhaitée. La troncature de la série infinie avec un nombre fini de termes conduit à ignorer la contribution des termes en dehors d'une fonction de fenêtre rectangulaire entre -Q et + Q.

soit $z = e^{j\pi\nu}$;ensuite (6.12) devient

$$H_a(z) = \sum_{n=-Q}^{Q} C_n z^n \tag{6.13}$$

avec les coefficients de réponse impulsionnelle C_{-Q}, C_{-Q+1},. . . , C_{-1}, C_0, C_1,. . . , C_{Q-1}, C_Q. La fonction de transfert approximée dans (6.13), avec des puissances positives de z, implique un filtre non causal ou non réalisable qui produirait une sortie avant qu'une entrée ne soit appliquée.

Pour remédier à cette situation, nous introduisons un retard de Q échantillons dans (6.13) pour donner

$$H(z) = z^{-Q} H_a(z) = \sum_{n=-Q}^{Q} C_n z^{n-Q} \tag{6.14}$$

soit $n - Q = -i$; ensuite $H(z)$ dans (6.14) devient

$$H(z) = \sum_{i=0}^{2Q} C_{Q-i} z^{-i} \tag{6.15}$$

soit $h_i = C_{Q-i}$ et $N - 1 = 2Q$; ensuite $H(z)$ devient

$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i} \tag{6.16}$$

où $H(z)$ est exprimé en termes de coefficients de réponse impulsionnelle h_i, et $h_0 = C_Q$, $h_1 = C_{Q-1}$,. . . , $h_Q = C_0$, $h_{Q+1} = C_{-1} = C_1$,. . . , $h_{2Q} = C{-}Q$. Les coefficients de réponse impulsionnelle sont symétriques par rapport à h_Q, avec $C_n = C_{-n}$.

L'ordre du filtre est $N = 2Q + 1$. Par exemple, si $Q = 5$, le filtre aura 11 coefficients $h_0, h_1, ..., h_10$ ou

$$\begin{aligned} \mathrm{h}_0 &= h_{10} = C_5 \\ \mathrm{h}_1 &= h_9 = C_4 \\ \mathrm{h}_2 &= h_8 = C_3 \\ \mathrm{h}_3 &= h_7 = C_2 \\ \mathrm{h}_4 &= h_6 = C_1 \\ \mathrm{h}_5 &= C_0 \end{aligned}$$

La figure 6.7 montre les fonctions de transfert souhaitées $H_d(\nu)$ idéalement représentées pour les filtres sélectifs en fréquence : passe-bas, passe-haut, passe-bande et coupe-bande pour lesquels les coefficients $C_n = C_{-n}$ peuvent être trouvés.

1. Passe-bas : $C_0 = \nu_1$

$$C_n = \int_0^{\nu_1} H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_1}{n\pi} \tag{6.17}$$

2. Passe-haut :$C_0 = 1 - \nu_1$

$$C_n = \sum_{\nu_1}^{1} H_d(\nu) \cos n\pi\nu d\nu = -\frac{\sin n\pi\nu_1}{n\pi} \tag{6.18}$$

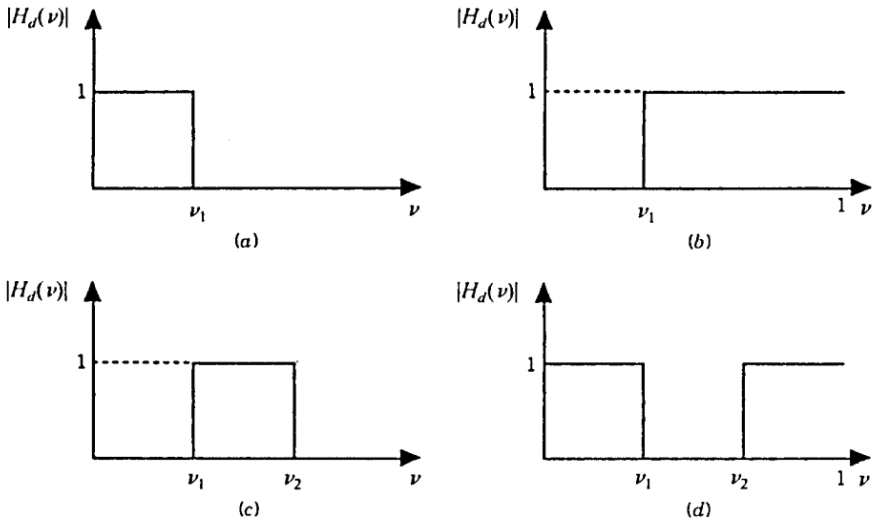


FIG. 6.7 : Fonction de transfert désirée : (a) passe-bas ; (b) passe-haut ; (c) passe-bande ; (d) coupe-bande

3. Passe-bande :$C_0 = \nu_2 - \nu_1$

$$C_n = \int_{\nu_1}^{\nu_2} H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_2 - \sin n\pi\nu_1}{n\pi} \tag{6.19}$$

4. Coupe-bande :$C_0 = 1 - (\nu_2 - \nu_1)$

$$C_n = \int_0^{\nu_1} H_d(\nu) \cos n\pi\nu d\nu + \int_{\nu_2}^{1} H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_1 - \sin n\pi\nu_2}{n\pi} \tag{6.20}$$

où ν_1 et ν_2 sont les fréquences de coupure normalisées illustrées à la figure 6.7. Plusieurs packages de conception de filtres sont actuellement disponibles pour la conception de filtres FIR, comme discuté plus loin. Lorsque nous implémentons un filtre FIR, nous développons un programme générique tel que les coefficients spécifiques détermineront le type de filtre (par exemple, passe-bas ou passe-bande).

6.3.2.3 Fonctions de la fenêtre

Nous avons tronqué la série infinie dans l'équation de la fonction de transfert (6.9) pour arriver à (6.12). Nous avons essentiellement mis une fonction de fenêtre rectangulaire avec une amplitude de 1 entre -Q et + Q et avons ignoré les coefficients en dehors de cette fenêtre. Plus cette fenêtre rectangulaire est large, plus Q est grand et plus nous utilisons de termes dans (6.12) pour obtenir une meilleure approximation de (6.9). La fonction de fenêtre rectangulaire peut donc être définie comme

$$w_R(n) = \begin{cases} 1 & \text{pour} |n| \leq Q \\ 0 & \text{sinon} \end{cases} \tag{6.21}$$

La transformée de la fonction de fenêtre rectangulaire $w_R(n)$ donne une fonction sinc dans le domaine fréquentiel. On peut montrer que

$$W_R(\nu) = \sum_{n=-Q}^{Q} e^{jn\pi\nu} = e^{-jQ\pi\nu}\left(\sum_{n=0}^{2Q} e^{jn\pi\nu}\right) = \frac{sin\left[\left(\frac{2Q+1}{2}\right)\pi\nu\right]}{sin(\pi\nu/2)} \tag{6.22}$$

qui est une fonction sinc qui présente des lobes latéraux élevés ou des oscillations provoquées par la troncature abrupte, en particulier, à proximité de discontinuités.

Un certain nombre de fonctions de fenêtre sont actuellement disponibles pour réduire ces oscillations de grande amplitude ; ils fournissent une troncature plus graduelle à l'expansion de série infinie. Cependant, alors que ces fonctions de fenêtre alternatives réduisent l'amplitude des lobes latéraux, elles ont également un lobe principal plus large, ce qui se traduit par un filtre avec une sélectivité plus faible. Une mesure de la performance d'un filtre est un facteur d'ondulation qui compare le pic du premier lobe latéral au pic du lobe principal (leur rapport) .Un compromis est de sélectionner une fonction de fenêtre qui peut réduire les lobes secondaires tout en approchant la sélectivité qui peut être réalisé avec la fonction de fenêtre rectangulaire. La largeur du lobe principal peut être réduite en augmentant la largeur de la fenêtre (ordre du filtre). Nous tracerons plus tard la réponse en amplitude d'un filtre FIR qui montre les lobes secondaires indésirables.

En général, les coefficients de la série de Fourier peuvent s'écrire

$$C_n^{'} = C_n w(n) \tag{6.23}$$

où $w(n)$ est la fonction de fenêtre. Dans le cas de la fonction fenêtre rectangulaire, $C_n^{'} = C_n$. La fonction de transfert dans (6.16) peut alors être écrite comme

$$H^{'}(z) = \sum_{i=0}^{N-1} h_i^{'} z^{-i} \tag{6.24}$$

où

$$h_i^{'} = C_{Q-1}^{'} \qquad 0 \leq i \leq 2Q \tag{6.25}$$

La fenêtre rectangulaire a son niveau de lobe latéral le plus élevé, en baisse de seulement -13 dB par rapport au sommet de son lobe principal, ce qui entraîne des oscillations d'une amplitude de taille considérable. D'autre part, il a le lobe principal le plus étroit qui peut fournir une sélectivité élevée. Les fonctions de fenêtre suivantes sont couramment utilisées dans la conception des filtres FIR

Fenêtre de Hamming
La fonction de fenêtre de Hamming est

$$w_R(n) = \begin{cases} 0.54 + 0.46\cos(n\pi/Q) & \text{for} |n| \leq Q \\ 0 & \text{sinon} \end{cases} \tag{6.26}$$

qui a le niveau de lobe latéral le plus élevé ou le premier à environ -43 dB du pic du lobe principal.

Fenêtre de Hanning
La fonction de fenêtre Hanning ou cosinus surélevé est

$$w_R(n) = \begin{cases} 0.5 + 0.5\cos(n\pi/Q) & \text{for}|n| \leq Q \\ 0 & \text{sinon} \end{cases} \tag{6.27}$$

qui a le niveau de lobe latéral le plus élevé ou le premier à environ -31 dB du pic du lobe principal.

Fenêtre de Blackman
La fonction de fenêtre Blackman est

$$w_R(n) = \begin{cases} 0.42 + 0.5\cos(n\pi/Q) + 0.08\cos(2n\pi/Q) & \text{for}|n| \leq Q \\ 0 & \text{sinon} \end{cases} \tag{6.28}$$

qui a le niveau de lobe latéral le plus élevé jusqu'à environ -58 dB à partir du pic du lobe principal. Alors que la fenêtre Blackman produit la plus grande réduction du lobe latéral par rapport aux fonctions de fenêtre précédentes, elle possède le lobe principal le plus large. Comme pour les fenêtres précédentes, la largeur du lobe principal peut être diminuée en augmentant la largeur de la fenêtre.

Fenêtre de Kaiser
La conception des filtres FIR avec la fenêtre Kaiser est devenue très populaire ces dernières années. Il a un paramètre variable pour contrôler la taille du lobe latéral par rapport au lobe principal. La fonction de fenêtre Kaiser est

$$w_R(n) = \begin{cases} I_0(b)/I_0(a) & \text{for}|n| \leq Q \\ 0 & \text{sinon} \end{cases} \tag{6.29}$$

où a est une variable déterminée empiriquement, et $b = a[1 - (n/Q)^2]^{1/2}.I_0(x)$ est la fonction de Bessel modifiée du premier type défini par

$$I_0(x) = 1 + \frac{0.25x^2}{(1!)^2} + \frac{(0.25x^2)^2}{(2!)^2} + ... = 1 + \sum_{n=1}^{\infty}\left[\frac{(x/2)^n}{n!}\right]^2 \tag{6.30}$$

which converges rapidly. A trade-off between the size of the sidelobe and the width of the mainlobe can be achieved by changing the length of the window and the parameter a.

Approximation assistée par ordinateur

Une technique efficace est la conception itérative assistée par ordinateur basée sur l'algorithme d'échange Remez, qui produit une approximation équiripple des filtres FIR. L'ordre du filtre et les bords des deux bandes passantes et des bandes d'arrêt sont fixes, et les coefficients sont modifiés pour fournir cette approximation en équiripple. Cela minimise l'ondulation dans les bandes passantes et les bandes d'arrêt. Les régions de transition ne sont pas soumises à des contraintes et sont considérées comme des régions « indifférentes », où la solution peut échouer. Plusieurs packages de conception de filtres commerciaux incluent l'algorithme Parks – McClellan pour la conception d'un filtre FIR.

6.3.3 Utilisation de buffers circulaires pour implémenter des filtres FIR en C

Pour un filtre FIR N-tap avec des coefficients différents de zéro uniquement pour les indices de l'ensemble $\{0, ..., N-1\}$, la somme de convolution (6.1) devient

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] = h[0]x[n] + h[0]x[n-1] + ... + h[N-1]x[n-N+1] \quad (6.31)$$

Notez que l'échantillon d'entrée le plus ancien $x[n-(N-1)]$ est multiplié par l'échantillon de réponse impulsionnelle $h[N-1]$ avec le plus grand indice et l'échantillon le plus récent x [n] est multiplié par l'échantillon de réponse impulsionnelle $h[0]$ avec le plus petit index. Cette équation est représentée schématiquement sur la figure 6.6 où les N échantillons de signal requis sont représentés stockés dans une ligne à retard. Dans une implémentation logicielle, la ligne à retard représente un tableau en mémoire. Entrer l'échantillon le plus récent dans la « ligne à retard » en décalant les éléments dans l'ensemble du tableau est inefficace pour une implémentation logicielle et une meilleure approche consiste à utiliser des buffers circulaires. Les tampons circulaires peuvent être implémentés en C. En outre, les 'C6000 DSP ont un support matériel auquel les instructions d'assemblage peuvent accéder pour implémenter encore plus efficacement les tampons circulaires.

Array Index	**Filter Coefficient Array `h[]`**	**Circular Buffer Array `xcirc[]`**
0	$h[0]$	$x[n-newest]$
1	$h[1]$	$x[n-newest+1]$
⋮	⋮	⋮
		$x[n-1]$
newest		$x[n]$
oldest		$x[n-N+1]$
		$x[n-N+2]$
⋮	⋮	⋮
$N-2$	$h[N-2]$	$x[n-newest-2]$
$N-1$	$h[N-1]$	$x[n-newest-1]$

FIG. 6.8 : Contenu du tableau de coefficients et du buffer circulaire

Le concept de tampon circulaire est illustré à la figure 6.8. Les coefficients de filtre sont stockés dans le tableau à N éléments $h[]$. Une variable, la plus récente, pointe vers l'emplacement dans le tableau de tampons circulaire qui contient l'échantillon le plus récemment entré. Lorsqu'un nouvel échantillon est reçu au temps n, il est écrit sur l'échantillon à l'emplacement $oldest = newest + 1$ modulo N et $newest$ est incrémenté modulo N.Dans une implémentation physique utilisant un registre à décalage, l'échantillon écrasé serait

décalé hors du fin du registre à décalage lorsque le nouveau registre est décalé. Notez que lorsque le *newest* a initialement la valeur $N-1$, il devient 0 lorsqu'il est incrémenté modulo N. Ainsi, les échantillons de données sont écrits dans le tableau de manière circulaire en descendant le tableau un élément à la fois vers le bas à l'emplacement $N-1$, puis remonter vers le haut du tableau à l'emplacement 0. Enfin, la sortie du filtre peut être calculée comme

$$y[n] = \sum_{k=0}^{N-1} h[k]xcirc[mod(newest - k, N)] \tag{6.32}$$

où $mod(newest - k, N)$ est l'entier de l'ensemble $\{0, ..., N-1\}$ formé en ajoutant des multiples de N au plus récent $-k$ jusqu'à ce qu'il tombe dans cet ensemble.

Un segment d'un programme C pour implémenter le filtre FIR avec un buffer circulaire est affiché ci-dessous

```
main()
{
   int x_index = 0;
   float y, xcirc[N];
       ...

/*-------------------------------------------*/
/* circularly increment newest               */
    ++newest;
    if(newest == N) newest = 0;
/*-------------------------------------------*/
/* Put new sample in delay line.             */
    xcirc[newest] = newsample;
/*-------------------------------------------*/
/* Do convolution sum                        */
    y = 0;
    x_index = newest
    for (k = 0; k < N; k++)
      {
       y += h[k]*xcirc[x_index];
       /*-----------------------------------*/
       /* circularly decrement x_index      */
       --x_index;
       if(x_index == -1) x_index = N-1;
       /*-----------------------------------*/
      }
    ...
}
```

Fig. 6.9 : Segment de programme C pour un filtre FIR avec buffer circulaire

Attention : $DSK6713\_AIC23\_read()$ et $MCBSP\_read()$ renvoient chacun un entier non signé de 32 bits. Convertissez la valeur renvoyée en un entier avant de décaler les 16 bits vers la droite pour supprimer le canal droit et obtenir le canal gauche avec l'extension de signe. Le décalage d'un int non signé vers la droite remplit les MSB de 0 afin que le signe ne soit pas étendu.

Remarque : C a l'opérateur mod,%, mais son implémentation par le compilateur est très inefficace car le compilateur doit tenir compte de tous les cas généraux. Par conséquent, vous devez implémenter l'opération de mod comme indiqué dans le segment de code ci-dessus.

6.3.4 Buffers circulaires utilisant le matériel 'C6000

La famille de DSP TMS320C6000 possède une capacité matérielle intégrée pour les tampons circulaires. Les huit registres, A4 – A7 et B4 – B7, peuvent être utilisés pour l'adressage indirect linéaire ou circulaire. Le registre de mode d'adresse (AMR) contient des champs de 2 bits, comme indiqué dans figure 6.10, pour chaque registre qui déterminent les modes d'adresse comme indiqué dans la figure 6.11. Le nombre de mots dans le tampon est appelé la taille de bloc. La taille de bloc est déterminée par les champs de 5 bits BK0 ou BK1 dans l'AMR. Le choix entre eux est déterminé par les champs de mode 2 bits. Soit Nblock la valeur du champ BK0 ou BK1. Ensuite, le tampon circulaire a la taille $BUF\_LEN = 2^{Nblock+1}$ octets. Ainsi, la taille du tampon circulaire ne peut être qu'une puissance de 2 octets.

31 – 26	25 – 21	20 – 16	15 – 14	13 – 12	11 – 10
Resvd	BK1	BK0	B7 mode	B6 mode	B5 mode

9 – 8	7 – 6	5 – 4	3 – 2	1 – 0
B4 mode	A7 mode	A6 mode	A5 mode	A4 mode

FIG. 6.10 : Champs du registre de mode d'adresse (AMR)

Mode	Addressing Option
00	Linear Mode
01	Circular Mode Using BK0 Size
10	Circular Mode Using BK1 Size
11	Reserved

FIG. 6.11 : Encodage de champ en mode AMR

Le tampon doit être aligné sur une limite d'octet qui est un multiple de la taille de bloc BUF_LEN. Par conséquent, les Nblock + 1 lsb de l'adresse de base du tampon doivent tous être 0. Cela peut être fait dans un programme C en utilisant le pragma

DATA_ALIGN. Supposons que le tampon soit un tableau $x[]$. La commande d'alignement est :

pragma DATA_ALIGN (x, BUF_LEN)

Le tableau $x[]$ doit être un tableau global. L'alignement peut également être effectué en créant une section nommée dans le programme d'assemblage et en utilisant l'éditeur de liens pour aligner correctement la section.

6.3.4.1 Comment le buffer circulaire est implémenté

L'adressage circulaire est mis en œuvre en inhibant les acheminements ou emprunts entre les bits Nblock et Nblock + 1 dans les calculs d'adresse. Par conséquent, les bits Nbloc + 1 à 31 ne changent pas lorsque l'adresse est incrémentée ou décrémentée d'une quantité inférieure à la taille de la mémoire tampon.

6.3.4.2 Adressage indirect via des registres

L'adressage circulaire matériel ne peut pas être effectué en C. Il doit être réalisé selon les instructions d'assembleur. L'adressage circulaire est réalisé par adressage indirect via l'un des huit registres autorisés en utilisant les modes d'auto-incrémentation / décrémentation et indexé. Une instruction de mise en mémoire tampon circulaire typique est

LDW *A5- -, A8

où le champ A5 de l'AMR a été défini pour l'adressage circulaire. LDW est le mnémonique pour « charger un mot ». Le mot est chargé dans le registre de destination A8 à partir de l'adresse pointée par A5 et l'adresse est décrémentée de 4 octets selon le mode dans l'AMR après avoir été utilisée (post décrémentée).

6.3.5 Interfaçage des fonctions C et assembleur

En raison des énormes progrès des capacités matérielles DSP et des outils de génération de code logiciel, il est de plus en plus courant d'implémenter des applications presque entièrement dans un langage de niveau supérieur tel que C. Certains avantages sont :

- Développement rapide de logiciels utilisant un langage de haut niveau.
- Peut utiliser de puissants compilateurs d'optimisation.
- L'application peut être facilement portée sur différents DSP.
- Les outils de profilage peuvent trouver des segments de code chronophages qui peuvent ensuite être écrits dans un code d'assembleur optimisé.

Il est très difficile de générer manuellement un code d'assembleur efficace pour la famille 'C6000 car il existe plusieurs unités d'exécution, un pipeline à plusieurs niveaux et différentes instructions prennent des temps différents pour s'exécuter.

Une autre raison d'utiliser des routines d'assembleur est que certaines capacités matérielles du DSP, telles que la mise en mémoire tampon circulaire matérielle, ne sont pas directement accessibles par C.

Le compilateur C a un ensemble spécifique de conventions pour l'utilisation des registres et le passage d'arguments. Les conventions d'utilisation des registres sont présentées dans les figures 6.12 et 6.13. En particulier, notez que le registre B15 est utilisé comme pointeur de pile, B3 contient l'adresse de retour, le registre A4 est utilisé pour renvoyer des résultats 32 bits et la paire A5: A4 est utilisée pour renvoyer des résultats 64 bits comme des entiers longs et des doubles.

6.3.5.1 Responsabilités de la fonction appelante et de la fonction appelée

Les étapes suivantes doivent être effectuées par la fonction appelante (parent) qui peut être une routine C ou d'assembleur. La fonction appelée (child) peut également être une fonction C ou d'assembleur.

1. Les arguments passés sont placés dans des registres ou sur la pile. Par convention,

Register	Preserved By	Special Uses
A0	Parent	
A1	Parent	
A2	Parent	
A3	Parent	Structure register
A4	Parent	Argument 1 or return value
A5	Parent	Argument 1 or return value with A4 for doubles and longs
A6	Parent	Argument 3
A7	Parent	Argument 3 with A6 for doubles and longs
A8	Parent	Argument 5
A9	Parent	Argument 5 with A8 for doubles and longs
A10	Child	Argument 7
A11	Child	Argument 7 with A10 for doubles and longs
A12	Child	Argument 9
A13	Child	Argument 9 with A12 for doubles and longs
A14	Child	
A15	Child	Frame pointer (FP)

Fig. 6.12 : Utilisation du registre côté « A »

l'argument 1 est l'argument le plus à gauche.

- Les dix premiers arguments sont passés dans les registres A et B comme indiqué dans les figures 6.12 et 6.13
- Des arguments supplémentaires sont passés sur la pile.

2. La fonction appelante (parent) doit enregistrer A0 à A9 et B0 à B9 si nécessaire après l'appel, en les poussant sur la pile.

3. L'appelant passe à la fonction (child).

4. Au retour, l'appelant récupère l'espace de pile utilisé pour les arguments.

Register	Preserved By	Special Uses
B0	Parent	
B1	Parent	
B2	Parent	
B3	Parent	Return address
B4	Parent	Argument 2
B5	Parent	Argument 2 with B4 for doubles and longs
B6	Parent	Argument 4
B7	Parent	Argument 4 with B6 for doubles and longs
B8	Parent	Argument 6
B9	Parent	Argument 6 with B8 for doubles and longs
B10	Child	Argument 8
B11	Child	Argument 8 with B10 for doubles and longs
B12	Child	Argument 10
B13	Child	Argument 10 with B12 for doubles and longs
B14	Child	Data page pointer (DP)
B15	Child	Stack pointer (SP)

Fig. 6.13 : Utilisation du registre côté « B »

La fonction child appelé doit effectuer les opérations suivantes :

1. La fonction appelée alloue de l'espace sur la pile pour les variables locales, le stockage temporaire et les arguments aux fonctions que cette fonction peut appeler. Le pointeur de trame (FP) est utilisé pour accéder aux arguments sur la pile.

2. Si la fonction appelée en appelle une autre, l'adresse de retour doit être enregistrée sur la pile. Sinon, il est laissé en B3.

3. Si la fonction appelée modifie A10 à A15 ou B10 à B15, elle doit les sauvegarder dans d'autres registres ou sur la pile.

4. Le code de fonction appelé est exécuté.

5. La fonction appelée renvoie un entier, un flottant ou un pointeur en A4. Double ou long double sont retournés dans la paire A5: A4.

6. A10 – A15 et B10 – B15 sont restaurés s'ils sont utilisés.

7. Le frame et les pointeurs de pile sont restaurés.

8. La fonction retourne en passant à la valeur de B3.

6.3.5.2 Utilisation des fonctions d'assembleur avec C

Pour écrire des fonctions d'assembleur qui peuvent être appelées à partir de C, les éléments suivants doivent être gardés à l'esprit :

• Les noms de variables C sont précédés d'un trait de soulignement par le compilateur lors de la génération du code d'assembleur. Par exemple, une variable C nommée x est appelée _x dans le code d'assembleur.

• L'appelant doit placer les arguments dans les registres appropriés ou sur la pile pour les arguments au-delà du numéro 10.

• A10 – A15 et B10 – B15, B3 et éventuellement A3 doivent être conservés par la fonction appelée.

Il peut utiliser tous les autres registres librement.

• La fonction appelée doit afficher tout ce qu'elle a poussé sur la pile avant de retourner à l'appelant.

• Tout objet ou fonction déclaré dans la fonction d'assembleur qui est accédé ou appelé à partir de C doit être déclaré avec une directive .def ou .global dans le code assembleur. Cela permet à l'éditeur de liens de résoudre les références à celui-ci.

6.3.5.3 Implémentation FIR à l'aide de la fonction ASM appelée d'un programme C

Le programme C **FIRcasm.c** (Figure 6.14) appelle la fonction ASM FIRcasmfunc.asm (Figure 6.16), qui implémente un filtre FIR.

Générer et exécuter ce projet en tant que FIRcasm. Vérifier que la sortie est un filtre passe-bande FIR 1 kHz. Deux buffers sont créés : *dly* pour les échantillons de données et h pour les coefficients du filtre. A chaque interruption, un nouvel échantillon de données est acquis et stocké à la fin (adresse mémoire supérieure) du buffer *dly*. Les échantillons de retard et les coefficients de filtre sont disposés en mémoire comme le montre la figure 6.15. Les échantillons de retard sont stockés en mémoire en commençant par l'échantillon le plus ancien avec l'échantillon le plus récent à la fin du buffer. Les coefficients sont disposés en mémoire avec $h(0)$ au début du buffer de coefficients et $h(N-1)$ à la fin.

Les adresses du tampon d'échantillonnage de retard, du buffers de coefficient de filtre et de la taille de chaque buffers sont transmises à la fonction ASM via les registres A4, B4 et A6, respectivement. La taille de chaque tampon à travers le registre A6 est doublée puisque les données dans chaque emplacement de mémoire sont stockées sous forme d'octet. Les pointeurs A4 et B4 sont incrémentés ou décrémenté tous les deux octets (deux emplacements de mémoire). L'adresse de fin du buffer des coefficients est en B4, qui est à $2N-1$.

Les deux instructions LDH chargent le contenu en mémoire pointé sur (dont l'adresse est spécifiée par) A4 et le contenu en mémoire à l'adresse spécifiée en B4. Cela charge les échantillons les plus anciens, x (n - (N - 1)) et h (N - 1), respectivement. A4 est

```
//FIRCASM.c FIR C program calling ASM function fircasmfunc.asm

#include "bp41.cof"                //BP @ Fs/8 coefficient file
int yn = 0;                        //initialize filter's output
short dly[N];                      //delay samples

interrupt void c_int11()           //ISR
{
      dly[N-1] = input_sample();   //newest sample @bottom buffer
      yn = fircasmfunc(dly,h,N);   //to ASM func through A4,B4,A6
      output_sample(yn >> 15);     //filter's output
      return;                      //return from ISR
}

void main()
{
      short i;

      for (i = 0; i<N; i++)
      dly[i] = 0;                  //init buffer for delays
      comm_intr();                 //init DSK, codec, McBSP
      while(1);                    //infinite loop
}
```

FIG. 6.14 : Programme C appelant une fonction ASM pour l'implémentation FIR (FIR-casm.c).

	Samples		
Coefficients	Time n		Time n + 1
h(0)	A4 → x(n - (N - 1))		A4 → x(n - (N - 2))
h(1)	x(n - (N - 2))		x(n - (N - 3))
h(2)	x(n - (N - 3))		x(n - (N - 4))
.	.		.
.	.		.
.	.		.
h(N - 2)	x(n - 1)		x(n)
B4 → h(N - 1)	x(n)	← newest →	x(n + 1)

FIG. 6.15 : Organisation de la mémoire des coefficients et des échantillons pour FIRcasm

ensuite post-incrémenté pour pointer en x (n - (N - 2)), et B4 est post-décrémenté pour pointer en h (N - 2). Après la première accumulation, l'échantillon le plus ancien est mis à jour. Le contenu en mémoire à l'adresse spécifiée par A4 est chargé dans A7, puis stocké à l'emplacement mémoire précédent. En effet, A4 est post-décrémenté sans modification pour pointer vers l'emplacement mémoire contenant l'échantillon le plus ancien. En conséquence, l'échantillon le plus ancien, x (n - (N - 1)), est remplacé (mis à jour) par x (n - (N - 2)). La mise à jour des échantillons de retard est pour l'unité de temps suivante. Au fur et à mesure que la sortie au temps n est calculée, les échantillons

sont mis à jour ou « amorcés » pour le temps (n + 1). Au temps n, la sortie du filtre est

$$y[n]=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+...+ h(1)x(n-1)+h(0)x(0)$$

```
;FIRCASMfunc.asm ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1]

            .def    _fircasmfunc
_fircasmfunc:                   ;ASM function called from C
            MV      A6,A1       ;setup loop count
            MPY     A6,2,A6     ;since dly buffer data as byte
            ZERO    A8          ;init A8 for accumulation
            ADD     A6,B4,B4    ;since coeff buffer data as byte
            SUB     B4,1,B4     ;B4=bottom coeff array h[N-1]
loop:                           ;start of FIR loop
            LDH     *A4++,A2    ;A2=x[n-(N-1)+i] i=0,1,...,N-1
            LDH     *B4--,B2    ;B2=h[N-1-i] i=0,1,...,N-1
            NOP     4
            MPY     A2,B2,A6    ;A6=x[n-(N-1)+i]*h[N-1-i]
            NOP
            ADD     A6,A8,A8    ;accumlate in A8

            LDH     *A4,A7      ;A7=x[(n-(N-1)+i+1]update delays
            NOP     4           ;using data move "up"
            STH     A7,*-A4[1]  ;-->x[(n-(N-1)+i] update sample
            SUB     A1,1,A1     ;decrement loop count
    [A1]    B       loop        ;branch to loop if count # 0
            NOP     5

            MV      A8,A4       ;result returned in A4
            B       B3          ;return addr to calling routine
            NOP     5
```

Fig. 6.16 : Fonction FIR ASM appelée depuis C (FIRcasmfunc.asm)

La boucle est traitée 41 fois. Pour chaque fois n, n + 1 et n + 2, une valeur de sortie est calculée, chaque échantillon étant mis à jour pour l'unité de temps suivante. L'échantillon le plus récent est également mis à jour dans ce processus, avec une valeur de données non valide résidant à l'emplacement de mémoire au-delà de la fin de la mémoire tampon. Mais cela est corrigé puisque pour chaque unité de temps, le dernier échantillon, acquis via l'ADC du codec, l'écrase.

L'accumulation est en A8 et le résultat, pour chaque unité de temps, est déplacé vers A4 pour être renvoyé à la fonction appelante. L'adresse de la fonction appelante est en B3.

Exemple : Implémentation FIR avec un programme C appelant la fonction ASM à l'aide d'un buffer circulaire (FIRcirc)
Le programme C FIRcirc.c (Figure 6.17) appelle la fonction ASM FIRcircfunc.asm (Figure 6.18), qui implémente un filtre FIR en utilisant un buffer circulaire. Au lieu de déplacer

```
//FIRcirc.c C program calling ASM function using circular buffer

#include "bp1750.cof"                //BP at 1750 Hz coeff file
int yn = 0;                          //init filter's output

interrupt void c_int11()             //ISR
{
 short sample_data;

 sample_data = input_sample();       //newest input sample data
 yn = fircircfunc(sample_data,h,N);  //ASM func passing to A4,B4,A6
 output_sample(yn >> 15);            //filter's output
 return;                             //return to calling function
}

void main()
{
 comm_intr();                        //init DSK, codec, McBSP
 while(1);                           //infinite loop
}
```

Fig. 6.17 : Programme C appelant une fonction ASM à l'aide d'un buffer circulaire (FIRcirc.c)

les données pour mettre à jour les échantillons de retard, un pointeur est utilisé. Les 16 LSB du registre de mode d'adresse (AMR) sont définis avec une valeur de

0x0040 = 0000 0000 0100 0000

Ceci sélectionne le mode A7 comme registre de pointeur de tampon circulaire. Les 16 MSB d'AMR sont définis avec N = 0x0007 pour sélectionner le bloc BK0 comme tampon circulaire. La taille du tampon est $2^{N+1} = 256$. Un tampon circulaire est utilisé dans cet exemple uniquement pour les échantillons de retard.

Il est également possible d'utiliser un deuxième tampon circulaire pour les coefficients. Par exemple, en utilisant

0x0140 = 0000 0001 0100 0000

sélectionnerait deux pointeurs, B4 et A7. Dans un programme C, un code d'assemblage en ligne peut être utilisé avec l'instruction asm. Par example,

asm ("MVK 0x0040, B6")

Notez l'espace vide après le premier guillemet pour que l'instruction ne commence pas dans la colonne 1. Le mode d'adressage circulaire élimine le déplacement des données pour mettre à jour les échantillons de retard, car le pointeur peut être déplacé pour obtenir le même résultat plus rapidement.

```
;FIRcircfunc.asm ASM function called from C using circular addressing
;A4=newest sample, B4=coefficient address, A6=filter order
;Delay samples organized: x[n-(N-1)]...x[n]; coeff as h(0)...h[N-1]

            .def   _fircircfunc
            .def   last_addr
            .def   delays
            .sect  "circdata"     ;circular data section
            .align 256            ;align delay buffer 256-byte boundary
delays      .space 256            ;init 256-byte buffer with 0's
last_addr   .int   last_addr-1    ;point to bottom of delays buffer
            .text                 ;code section
_fircircfunc:                      ;FIR function using circ addr
            MV     A6,A1          ;setup loop count
            MPY    A6,2,A6        ;since dly buffer data as byte
            ZERO   A8             ;init A8 for accumulation

            ADD    A6,B4,B4       ;since coeff buffer data as bytes
            SUB    B4,1,B4        ;B4=bottom coeff array h[N-1]

            MVKL   0x00070040,B6  ;select A7 as pointer and BK0
            MVKH   0x00070040,B6  ;BK0 for 256 bytes (128 shorts)

            MVC    B6,AMR         ;set address mode register AMR

            MVK    last_addr,A9   ;A9=last circ addr(lower 16 bits)
            MVKH   last_addr,A9   ;last circ addr (higher 16 bits)
            LDW    *A9,A7         ;A7=last circ addr
            NOP    4
            STH    A4,*A7++       ;newest sample-->last address

loop:                             ;begin FIR loop
            LDH    *A7++,A2       ;A2=x[n-(N-1)+i] i=0,1,...,N-1
    ||      LDH    *B4--,B2       ;B2=h[N-1-i] i=0,1,...,N-1
            SUB    A1,1,A1        ;decrement count

    [A1]    B      loop           ;branch to loop if count # 0
            NOP    2
            MPY    A2,B2,A6       ;A6=x[n-(N-1)+i]*h[N-1+i]
            NOP
            ADD    A6,A8,A8       ;accumulate in A8

            STW    A7,*A9         ;store last circ addr to last_addr
            B      B3             ;return addr to calling routine
            MV     A8,A4          ;result returned in A4
            NOP    4
```

FIG. 6.18 : Programme C appelant une fonction ASM à l'aide d'un buffer circulaire (FIRcirc.c)

6.3.6 Filtres à réponse impulsionnelle à durée infinie (IIR)

Un filtre avec une réponse impulsionnelle, $h(n)$, qui a une durée infinie est appelé filtre IIR. Lorsque $h(n)$ est la somme des exponentielles amorties, sa transformée en z, $H(z)$, qui est également appelée sa fonction de transfert, est une fonction rationnelle de z. Autrement dit, il s'agit du rapport de deux polynômes de degré fini. Nous utiliserons une fonction rationnelle de la forme

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_M z^{-M}} = \frac{B(z)}{A(z)} \tag{6.33}$$

6.3.6.1 Réalisations pour les filtres IIR

Les fonctions de transfert rationnel peuvent être réalisées de plusieurs manières. Trois réalisations communes seront décrites ci-dessous. La première réalisation sera appelée forme directe de type 0. Le rapport des transformées en z de la sortie et de l'entrée du filtre est

$$\frac{Y(z)}{X(z)} = H(z) = \frac{B(z)}{A(z)} \tag{6.34}$$

La multiplication croisée donne

$$Y(z)A(z) = X(z)B(z) \quad ou \quad Y(z)\left(1 + \sum_{k=1}^{M} a_k z^{-k}\right) = X(z)\sum_{k=0}^{N} b_k z^{-k} \tag{6.35}$$

Prendre tout sauf le terme $Y(z)$ sur le côté droit donne

$$Y(z) = \sum_{k=0}^{N} b_k X(z) z^{-k} - \sum_{k=1}^{M} a_k Y(z) z^{-k} \tag{6.36}$$

L'équation du domaine temporel est l'équation de différence

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{M} a_k y[n-k] \tag{6.37}$$

Cette équation montre comment calculer la sortie du filtre actuel à partir du courant et des N entrées passées et des M sorties passées. Un filtre mis en œuvre de cette manière est également appelé filtre récursif car les sorties passées sont utilisées pour calculer la sortie de courant. C'est ce qu'on appelle une forme directe parce que les coefficients de la fonction de transfert apparaissent directement dans l'équation de différence.

Une autre réalisation que nous appellerons une forme directe de type 1 est basée sur l'observation que (6.34) peut être réarrangé dans la forme en cascade

$$Y(z) = \frac{X(z)}{A(z)} B(z) = V(z)B(z) \tag{6.38}$$

ou

$$V(z) = X(z)\frac{1}{A(z)} \tag{6.39}$$

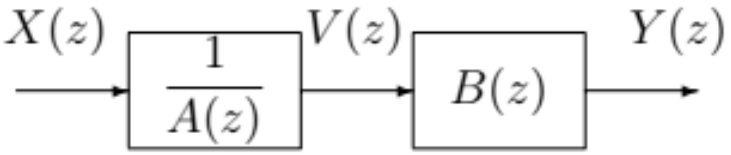


FIG. 6.19 : Première étape de la recherche de la réalisation directe de forme type 1

Ceci est illustré à la figure 6.19. Le signal intermédiaire v [n] peut être calculé en utilisant la réalisation directe de la forme 0

$$v[n] = x[n] - \sum_{k=1}^{M} a_k v[n-k] \tag{6.40}$$

Ensuite, la sortie peut être calculée comme

$$y[n] = \sum_{k=0}^{N} b_k v[n-k] \tag{6.41}$$

Un schéma fonctionnel de ces équations est illustré à la figure 6.20 où l'on suppose que M = N. Cette forme nécessite moins de stockage que la forme directe de type 0.

Le contenu des éléments de retard, $s_1[n], ..., s_N[n]$, sont des variables d'état pour le filtre. La sortie actuelle et l'état suivant peuvent être calculés à partir de l'entrée et de l'état actuels. La séquence d'étapes suivante peut être utilisée pour calculer les sorties et les états du filtre :

Etape 1: Calculer $v[n]$

$$v[n] = x[n] - \textstyle\sum_{k=1}^{M} a_k s_k[n]$$

Etape 2: Calculer la sortie $y[n]$

$$y[n] = b_0 v[n] \textstyle\sum_{k=1}^{N} b_k s_k[n]$$

Étape 3: Mettre à jour les variables d'état

$$s_N[n+1] = s_{N-1}[n]$$
$$s_{N-1}[n+1] = s_{N-2}[n]$$
$$.$$
$$.$$
$$.$$
$$s_2[n+1] = s_1[n]$$
$$s_1[n+1] = v[n]$$

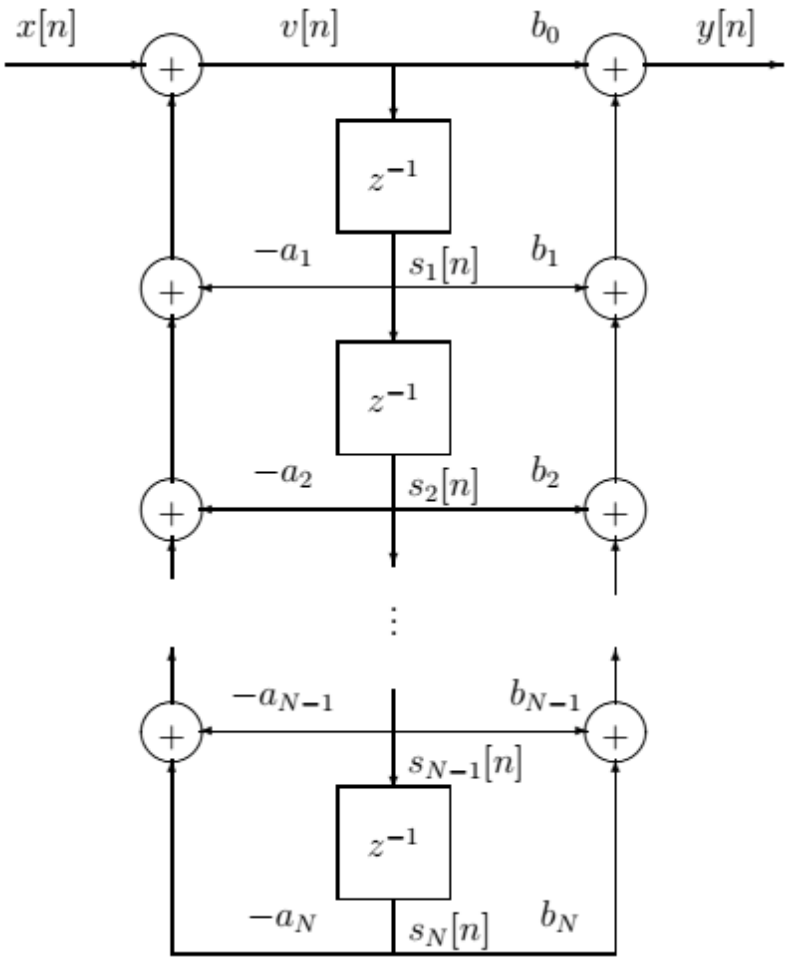


FIG. 6.20 : Réalisation de forme directe type 1

Une autre réalisation appelée forme directe de type 2 peut être trouvée en réarrangeant (6.35). Pour simplifier, soit M = N. Alors

$$Y(z) = b_0 X(z) + \sum_{k=1}^{M} [b_k X(z) - a_k Y(z)] z^{-k} \tag{6.42}$$

Un schéma fonctionnel de cette réalisation est illustré à la figure 6.21. Il nécessite essentiellement le même stockage et la même arithmétique qu'une forme direct de type 1.

La séquence d'étapes pour calculer la sortie de la forme direct de type 2 et mettre à jour son état est :

Etape 1: Calculer la sortie $y[n]$

$$y[n] = b_0 x[n] + s_1[n]$$

Etape 2: Mettre à jour les variables d'état

$$s_1[n+1] = b_1 x[n] - a_1 y[n] + s_2[n]$$

$$s_2[n+1] = b_2 x[n] - a_2 y[n] + s_3[n]$$

.

.

.

$$s_{N-1}[n+1] = b_{N-1} x[n] - a_{N-1} y[n] + s_N[n]$$

$$s_n[n+1] = b_N x[n] - a_N y[n]$$

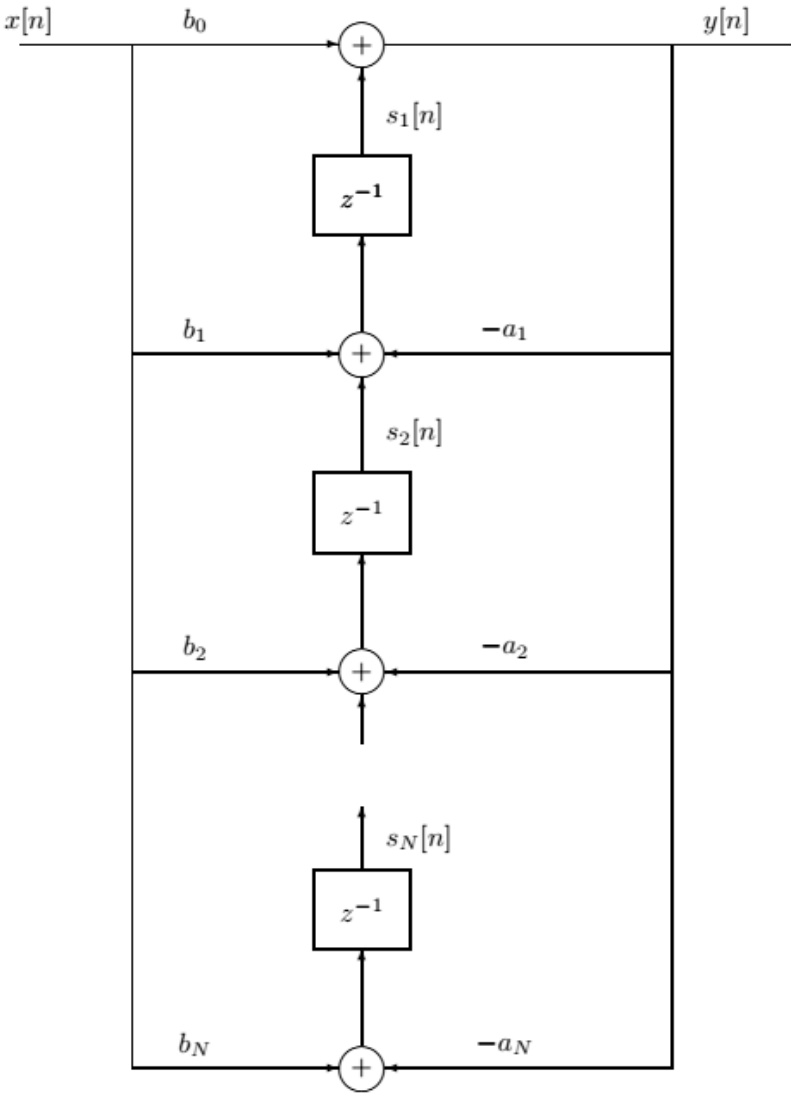


FIG. 6.21 : Réalisation de forme directe type 2

6.3.6.2 Implémentation de la forme directe I sur le DSK C6713 *

Lors de l'implémentation d'une réalisation directe de forme I sur la puce C6713 (avec l'hypothèse M = N), l'équation (6.37) serait réécrite comme

$$y[n] = b[0]x[n] + \sum_{i=1}^{N}(b[i]x[n-i] - a[i]y[n-i]) \tag{6.43}$$

Cela peut être codé en C comme illustré à la figure 6.22.

```
// IIR_directI.c  IIR filter using Direct Form I
// assume M=N
/*1 */ #include "DSK6713_aic23.h"
/*2 */ #include "IIR_LPF1800.cof"     // LPF @ 1800 Hz coefficient file
/*3 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;// sampling frequency of codec
/*4 */ short x[N+1] = {0};             // input samples
/*5 */ short y[N+1] = {0};             // output samples
/*6 */ int yn;
/*7 */
/*8 */ interrupt void c_int11()        // ISR
/*9 */ {
/*10*/   int i;
/*11*/   yn=0;
/*12*/   x[0] = input_left_sample();   // input sample
/*13*/   yn += (b[0]*x[0]>>15);        // b[0]*x[n]
/*14*/
/*15*/   for (i = 1; i <= N; i++)
/*16*/     yn += ((b[i]*x[i]>>15) - (a[i]*y[i]>>15));
/*17*/   for (i = N; i > 0; i--)
/*18*/     {
/*19*/       x[i]=x[i-1];             // update buffers
/*20*/       y[i]=y[i-1];
/*21*/     }
/*22*/   y[1]=yn;                     // store current output as y[n-1]
/*23*/   // for next output
/*24*/   output_left_sample((short)yn);// output final result for time n
/*25*/   return;                      // return from ISR
/*26*/ }
/*27*/
/*28*/ void main()
/*29*/ {
/*30*/   comm_intr();                 // init DSK, codec, SPO
/*31*/   while(1);                    // infinite loop
/*32 */ }
```

Fig. 6.22 : Réalisation de forme directe type 1

6.4 Transformée de Fourier Rapide (Fast Fourier Transform FFT)

Les opérations telles que DFT ou FFT nécessitent qu'une trame de données soit présente au moment du traitement. Contrairement au filtrage, où les opérations sont effec-

tuées sur chaque échantillon entrant, dans le traitement de trame, N échantillons sont d'abord capturés, puis les opérations sont effectuées sur tous les N échantillons.

Pour effectuer un traitement de trame, une méthode appropriée de collecte de données et de traitement et d'envoi de données est requise. Le traitement d'une trame de données n'est généralement pas terminé dans l'intervalle de temps d'échantillonnage, il est plutôt étalé sur la durée d'une trame avant que la prochaine trame de données ne soit collectée. Par conséquent, les échantillons entrants doivent être stockés dans une mémoire tampon distincte autre que celle en cours de traitement. En outre, un autre tampon est nécessaire pour envoyer une trame de données précédemment traitée. Comme expliqué précédemment, cela peut être réalisé par une triple mise en mémoire tampon impliquant trois tampons : entrée, intermédiaire et sortie.

Pour faire du triple buffering sur le C6x, nous incorporons une boucle sans fin tournant autour de la rotation de trois buffers. Les tampons tournent à chaque fois que le tampon d'entrée est plein de sorte qu'une nouvelle trame de N données échantillonnées est transmise au tampon intermédiaire pour traitement et qu'une trame précédemment traitée est transmise au tampon de sortie pour transmission.

```
short *output;                /* POINTER TO DATA ARRAY FOR OUTPUT       */
short *input;                 /* POINTER TO DATA ARRAY FOR INPUT        */
short *intermediate;          /* POINTER TO DATA ARRAY FOR DMA ACCESS   */
static short index=0;

main()
{
      CSL_init();             // Initialize the library

      hMcbsp = MCBSP_open(MCBSP_DEV1,MCBSP_OPEN_RESET);
      MCBSP_config(hMcbsp,&MyConfig);

      hTimer = TIMER_open(TIMER_DEV0, TIMER_OPEN_RESET);
      TIMER_config(hTimer, &timerCfg);

      init_arrays();

      IRQ_globalDisable();
      IRQ_nmiEnable();
      IRQ_map(IRQ_EVT_RINT1,15);
      IRQ_enable(IRQ_EVT_RINT1);
      IRQ_globalEnable();

      /* Main Loop, wait for Interrupt */

      for( ; ; )
      {
            wait_buffer();    /* WAIT FOR A NEW BUFFER OF DATA */
      }
}
```

Ici, la plupart des initialisations pour le codec et McBSP n'ont pas été montrées pour rendre le code plus facile à suivre. Une fois le port série initialisé, les trois tableaux sont alloués en mémoire et initialisés à zéro. Le programme entre alors dans une boucle sans fin où la fonction wait_buffer(), illustrée ensuite, est exécutée sans fin :

```
void wait_buffer(void)
{
      short *p;
      /* WAIT FOR ARRAY INDEX TO BE RESET TO ZERO BY ISR  */
      while(index);

      /* ROTATE DATA ARRAYS */
      p = input;
      input = output;
      output = intermediate;

      //Function call here...

      intermediate = p;
      HostTargetComm();
      while(!index);
}
```

Cette fonction vérifie l'index de la variable globale pour faire la rotation des tableaux et lancer le traitement. Lorsque le tableau d'entrée est plein (indiqué par l'index), les tableaux sont pivotés et le tableau intermédiaire est défini pour le traitement. Le commentaire //Function call here… indique où la fonction de traitement telle que FFT doit être placée.

L'ISR est également modifié comme indiqué dans le bloc de code suivant. Dans l'ISR, EDMA est utilisé pour transférer un échantillon de DRR vers DXR sans utiliser l'API MCBSP$_{w}rite()$.

Notez que l'index est incrémenté dans l'ISR.

```
EDMA_Handle hEdma;

EDMA_Config myConfig = {
      0x28000000, // opt
      0x01900000, // src: DRR 1
      0x00000001, // cnt
      0x01900004, // dst: DXR 1
      0x00000000, // idx
      0x00000000  // rld
};

interrupt void serialPortRcvISR (void)
{
      int temp;
      temp =  MCBSP_read(hMcbsp);
      input[index] = temp;

      myConfig.src = EDMA_SRC_RMK(&temp); // Update EDMA source

      hEdma = EDMA_open(EDMA_CHA_XEVT1, EDMA_OPEN_RESET);
      EDMA_config(hEdma,&myConfig);
      EDMA_enableChannel(hEdma);
      EDMA_setChannel(hEdma);
      EDMA_disableChannel(hEdma);
      EDMA_close(hEdma);


      if (++index == BUFFLENGTH)
            index = 0;
}
```

Pour configurer EDMA pour la lecture à partir de DRR et la réécriture dans DXR, les adresses source et de destination doivent se référer respectivement à DRR1 et DXR1. Dans le registre des paramètres d'options, le niveau de priorité est défini sur élevé, la taille de l'élément sur 32 bits, avec une source et une destination unidimensionnelles.

Aucune incrémentation d'adresse pour la source ou la destination n'est nécessaire puisque la source et la destination sont des registres mappés en mémoire à adresse fixe. Semblable aux autres périphériques CSL, EDMA est ouvert et configuré avec les API EDMA_open() et EDMA_config(). L'API EDMA_ setChannel() déclenche le canal EDMA. Après avoir transmis les données, le canal ouvert doit être fermé avec EDMA_close().

À des fins de traitement FFT, les données lues à partir de DRR sont décalées vers la gauche de 16 bits pour se débarrasser de la partie de données correspondant au canal droit. Pour stocker au format Q-15, il est décalé vers la droite de 16 bits. Les échantillons d'entrée sont ensuite placés dans le tableau d'entrée pour construire une trame de longueur BUFFLENGTH. L'index de la variable est incrémenté à chaque fois qu'un nouvel échantillon d'entrée est lu. Cette variable est remise à zéro lorsque le tampon d'entrée est plein, c'est-à-dire que l'index atteint BUFFLENGTH. Cette réinitialisation fait sortir le programme de la boucle while(index) dans la fonction wait_buffer(). Ensuite, le reste du programme dans wait_buffer() est exécuté.

Revenons maintenant à wait_buffer(). Une fois l'index remis à zéro, le tampon d'entrée est réaffecté à un pointeur nommé p. Cette réaffectation est nécessaire pour la partie //Function call here... afin d'éviter tout conflit avec l'ISR. Notez que l'ISR utilise le tampon d'entrée pour recevoir de nouveaux échantillons. Si, par exemple, la fonction FFT traite les données dans le tampon d'entrée, des résultats erronés peuvent être produits, car l'ISR peut modifier le contenu du tampon d'entrée à tout moment. Ce dysfonctionnement peut se produire parce que l'ISR s'exécute sur une base de priorité plus élevée, tandis que la fonction FFT s'exécute sur une base de priorité plus faible. Après l'instruction p=input, le tampon de sortie est réaffecté au tampon d'entrée. Cette réaffectation permet à l'ISR d'utiliser le tampon de sortie pour recevoir et stocker de nouveaux échantillons entrants. Notez que les données dans le tampon de sortie sont envoyées par le DMA. La prochaine sortie de réaffectation = intermédiaire est nécessaire pour envoyer les données traitées dans le tampon intermédiaire au DXR.

Une fois les données traitées dans //Function call here..., le pointeur p est réaffecté au tampon intermédiaire pour qu'il pointe vers les échantillons traités. Les données du tampon intermédiaire sont envoyées par la fonction HostTargetComm() dans le cadre de la fonction wait_buffer(). La boucle while(!index) à la fin de wait_buffer() garantit qu'une trame n'est traitée qu'une seule fois. En l'absence d'un nouvel échantillon dans le DRR, index reste à zéro et le programme attend à while(!index) car !index est VRAI. Lorsqu'un nouvel échantillon arrive dans le DRR, l'index est incrémenté et le programme sort de wait_buffer() et tombe dans la boucle dans main(). Là, il attend un nouveau cadre.

Pour la communication avec un hôte PC, deux méthodes différentes sont mentionnées ici en fonction de la plate-forme cible (DSK ou EVM) utilisée. Lors de l'utilisation d'EVM, la communication s'effectue via l'adresse d'E/S mappée en mémoire, PCI FIFO, correspondant au canal PCI du PC. Il existe deux registres FIFO (premier entré, premier sorti) 32 bits, appelés FIFO-read et FIFO-write, dans le contrôleur PCI de la carte EVM à travers lesquels l'hôte PC et le C6x peuvent communiquer. La lecture FIFO est utilisée pour le transfert de données du PC hôte vers le DSP et l'écriture FIFO pour le transfert de données du DSP vers le PC hôte. L'initialisation de la FIFO se fait via le registre d'état FIFO mappé en mémoire. La communication est réalisée à l'aide de la fonction HostTargetComm(). Cette fonction utilise la capacité DMA du C6x pour envoyer la matrice

intermédiaire à l'hôte via les registres FIFO. Ce qui suit fournit le code pour le faire :

```
void HostTargetComm(void)
{
      dma_reset();
      dma_init(2,              //Channel
            0x0A000110u,      //Primary Control Register   (Peripherals pp4-9)
            0x0000000Au,      //Secondary Control Register
(unsigned int) intermediate,  //Source Address
            0x01710000u,      //Destination Address
            0x00010080u);     //Transfer Counter Register
      DMA_START(DMA_CH2);
}
```

Les deux fonctions API DMA dma_reset() et dma_init() sont utilisées pour initialiser le DMA pour le transfert de données entre le tableau intermédiaire et la FIFO. Une seule trame d'échantillons de longueur 128 est transférée comme indiqué par le contenu du registre de compteur de transfert (TCR). L'API dma_reset() réinitialise tous les registres DMA à leurs états de réinitialisation à la mise sous tension. L'API dma_init() initialise un canal DMA sélectionné en attribuant des valeurs appropriées à ses registres de contrôle. Ici, le premier argument de l'API dma_init() est utilisé pour sélectionner le canal DMA 2. Le deuxième argument définit le registre de contrôle primaire sur 0x0A000110u, comme le montre la figure 6.23. Cette valeur est spécifiée sur la base des éléments suivants : Le bit EMOD est défini sur 1 pour mettre en pause le canal DMA pendant un arrêt d'émulation. Le bit TCINT est mis à 1 pour activer l'interruption du contrôleur de transfert. La taille de l'élément est de 16 bits, les bits ESIZE sont donc définis sur 01. Les bits SRC DIR sont définis sur 01 pour incrémenter l'adresse source de la taille de l'élément en octets. Le troisième argument de l'API dma_init() définit le registre de contrôle secondaire. Les bits SX IE et FRAME IE de ce registre sont mis à 1 pour activer l'interruption du canal DMA. Le quatrième argument de l'API dma_init() affecte le tableau intermédiaire comme adresse source. Le cinquième argument est défini sur 0x01710000u, qui est l'adresse de l'interface PCI EVM, à travers laquelle le programme hôte lit la sortie. Le dernier argument définit le registre du compteur de transfert sur 0x00010080u. Les 16 bits supérieurs spécifient le nombre de trames. Ici, ces bits sont mis à 0x0001u afin d'envoyer une trame à la fois. Les 16 bits inférieurs sont définis sur 0x0080u pour avoir 128 éléments dans une trame. Enfin, la fonction HostTargetComm() appelle la macro DMA_START() pour activer le canal DMA 2. Cette macro définit le champ START du Primary Control Register sur 01.

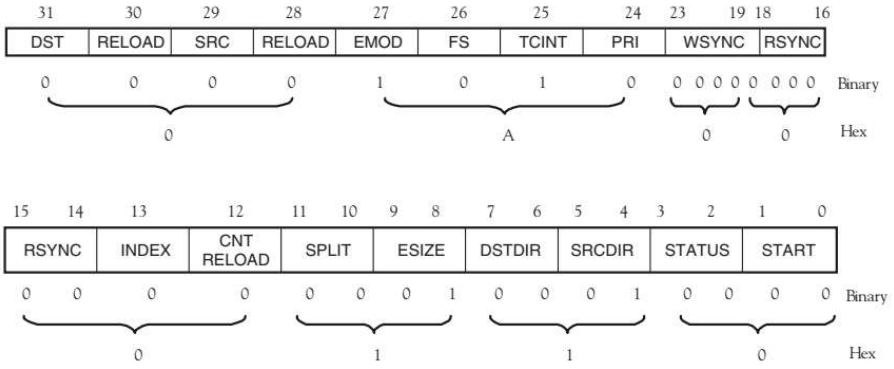


Fig. 6.23 : Registre de controle primaire

Lors de l'utilisation du C6711 DSK, la communication s'effectue via le port parallèle.

Afin d'implémenter une fonction équivalente à HostTargetComm, l'interface hôte-port (HPI) est utilisée au lieu de DMA. HPI est un port parallèle de 16 bits à travers lequel un hôte PC peut accéder directement à la mémoire et aux périphériques du C6x. Aucune configuration spécifique n'est requise pour la communication HPI côté DSP, puisque le programme hôte fait office de maître. Cependant, le pointeur intermédiaire doit être transmis à l'hôte pour avoir un accès mémoire approprié, car le schéma de triple mise en mémoire tampon continue de changer l'adresse de la mémoire physique où le tableau de valeurs intermédiaires est stocké. En outre, une variable doit être utilisée comme indicateur pour la synchronisation des données. Une adresse mémoire physique fixe, par exemple 0x00000200, est allouée en tant que section distincte, .hpi, pour permettre l'accès depuis l'hôte. De cette façon, les données traitées peuvent toujours être lues en se référant à son adresse stockée dans cette section. Ainsi, le fichier de commande de l'éditeur de liens et la fonction HostTargetComm doivent être modifiés comme suit :

```
MEMORY
{
      VECS: o = 00000000h     l = 00000200h
      HPI:  o = 00000200h     l = 00000008h
      IRAM: o = 00000208h     l = 0000FDF8h
      CE0:  o = 80000000h     l = 01000000h
}

SECTIONS
{
      .hpi        >       HPI
}
#pragma DATA_SECTION(hpi_access, ".hpi")
int hpi_access[2];    // pointer of intermediate data array

...
void HostTargetComm(void)
{
      hpi_access[0] = (int) intermediate;
      hpi_access[1] = 0xffffffff;         // Data is ready to be read

 // Wait until the flag is reset to 0 by Host after data transmission
      while(hpi_access[1] != 0x00000000);
}
```

6.4.1 Implementation DFT

DFT peut être simplement calculé à partir de l'équation

$$X[k] = \sum_{n=0}^{N-1} x[n] * W_N^{nk}, k = 0, 1, ..., N-1 \tag{6.44}$$

où $W_N = e^{-j2\pi/N}$. Cette équation nécessite N multiplications complexes et N–1 additions complexes pour chaque terme. Pour tous les N termes, N^2 multiplications complexes et N^2–N additions complexes sont nécessaires. Comme cela est bien connu, cette méthode n'est pas efficace car les propriétés de symétrie de la transformée ne sont pas exploitées. Cependant, il est utile d'implémenter l'équation sur le C6x en comparaison avec l'implémentation FFT. La capacité graphique de CCS est utilisée ici à cette fin. Ceci est effectué de manière hors ligne car le temps nécessaire pour effectuer la DFT dépasse la durée d'une capture de trame.

Tout d'abord, un simple signal composite est généré dans MATLAB avec les composantes de fréquence à 750 Hz, 2500 Hz et 3000 Hz. La sauvegarde de deux périodes de ce signal échantillonné à 8000 Hz donne un signal de 64 points. La figure 6.24 montre le signal lu dans CCS et tracé en utilisant sa capacité graphique. Le contenu fréquentiel du signal est également tracé sur la base d'une option FFT intégrée.

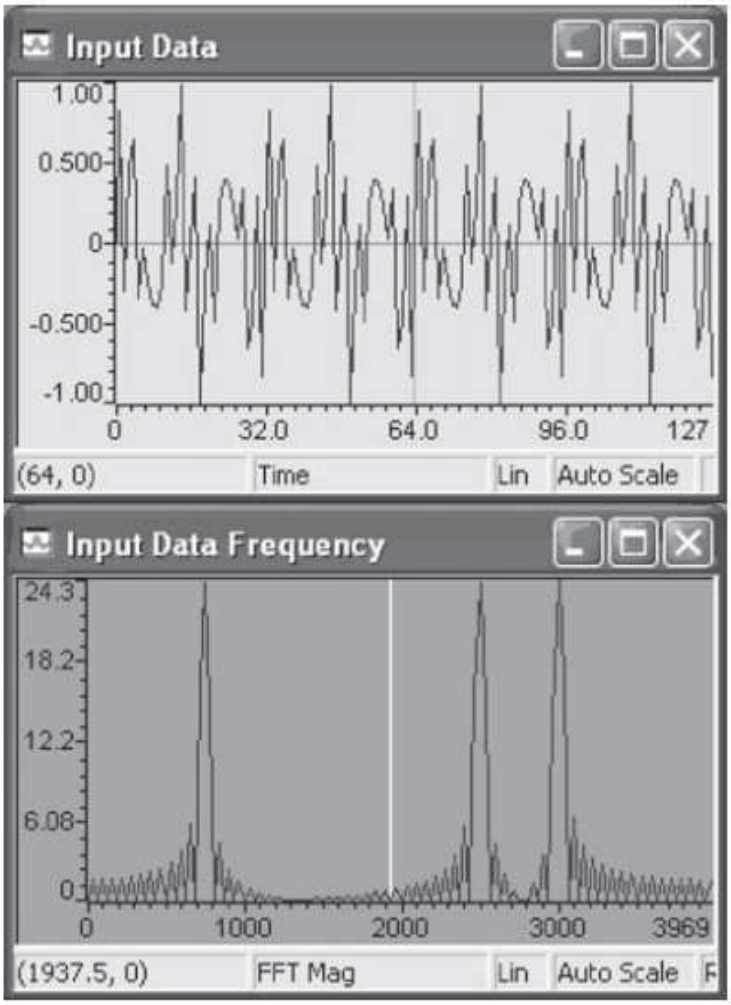


FIG. 6.24 : Signal d'entrée dans les domaines temporel et fréquentiel

Le code DFT utilisé est celui figurant dans le TI Application Report SPRA291 . Voici le code :

```
#include <math.h>
#include <math.h>
#include "params.h"

void dft(int N, COMPLEX *X){
      int n, k;
      double arg;
      int Xr[1024];
      int Xi[1024];
      short Wr, Wi;
      for(k=0; k<N; k++){
            Xr[k] = 0;
            Xi[k] = 0;
            for(n=0; n<N; n++){
                  arg =(2*PI*k*n)/N;
                  Wr = (short)((double)32767.0 * cos(arg));
                  Wi = (short)((double)32767.0 * sin(arg));
                  Xr[k] = Xr[k] + X[n].real * Wr + X[n].imag * Wi;
                  Xi[k] = Xi[k] + X[n].imag * Wr - X[n].real * Wi;
            }
      }

      for (k=0;k<N;k++){
            X[k].real = (short)(Xr[k]>>15);
            X[k].imag = (short)(Xi[k]>>15);
      }
}
```

Pour utiliser ce code, l'entrée doit être représentée sous forme de nombres complexes. Ceci est fait en utilisant une définition de structure pour créer une variable complexe avec des composants réels et imag. Le programme principal utilisé pour effectuer la DFT est le suivant :

```
main()
{
      int i,j;
      COMPLEX x[128];
      int mag[128];

      /*Change input to Q-15*/
      for(i=0;i<128;i++)
      {
            x[i].real=0x7FFF * input_data[i];
            x[i].imag=0;
      }
      dft(128, x);

            for(i=0;i<128;i++)
                  mag[i]=(x[i].real*x[i].real + x[i].imag*x[i].imag) << 1;
      return(0);
}
```

Dans ce programme, l'entrée est convertie au format Q-15 et stockée dans la structure complexe, qui est ensuite utilisée pour appeler la fonction DFT. L'ampleur du résultat de la DFT est illustrée à la figure 6.25. Comme prévu, il y a trois pics, à 750 Hz, 2500 Hz et 3000 Hz. Notez que ce code est assez inefficace, car il calcule chaque facteur de twiddle à l'aide de la bibliothèque mathématique à chaque itération. L'exécution de ce code à partir de la SDRAM externe entraîne un temps d'exécution d'environ 1,6 × 109 cycles pour une trame de 128 points. Par conséquent, le code DFT précédent ne peut pas être exécuté en temps réel sur le DSK, car seuls 18 750 × 128 = 2,4 × 10^6 cycles sont disponibles pour effectuer une transformation à 128 points.

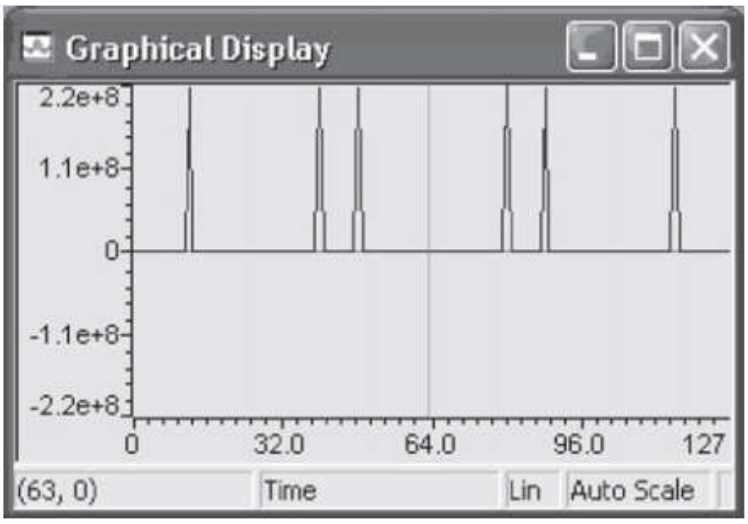


FIG. 6.25 : Réponse en amplitude de la DFT

6.4.2 Implementation de la FFT

Pour permettre un traitement en temps réel, FFT utilise les propriétés de symétrie de DFT. L'approche de calcul d'une FFT à 2N points telle que mentionnée dans le rapport d'application TI SPRA291 est adoptée ici. Cette approche consiste à former deux nouveaux signaux à N points $x_1[n]$ et $x2[n]$ à partir du signal à 2N points d'origine $g[n]$ en le divisant en parties paires et impaires comme suit :

$$\begin{aligned} x_1[n] &= g[2n] \qquad 0 \leq n \leq N-1; \\ x_2[n] &= g[2n+1] \end{aligned} \tag{6.45}$$

A partir des deux séquences $x_1[n]$ et $x_2[n]$, une nouvelle séquence complexe est définie comme

$$x_1[n] = x_1[n] + jx_2[n] \qquad 0 \leq n \leq N-1 \tag{6.46}$$

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

$$\begin{aligned} G[k] &= X[k]A[k] + X^*[N-k]B[k], \\ & k = 0, 1, ..., N-1, with X[N] = X[0], \end{aligned} \tag{6.47}$$

est utilisé, où

$$A[k] = \frac{1}{2}(1 - jW_{2N}^k \tag{6.48}$$

et

$$A[k] = \frac{1}{2}(1 + jW_{2N}^k \tag{6.49}$$

Seuls N points de $G[k]$ sont calculés à partir de l'équation. (6.47). Les points restants sont trouvés en utilisant la propriété conjuguée complexe de $G[k]$, $G[2N–k] = G*[k]$. En conséquence, une transformée à 2N points est calculée sur la base d'une transformée à N points, conduisant à une réduction du nombre de cycles. Les codes des fonctions (split1 ,R4DigitRevIndexTableGen, digit$_r everseetradix4)implmentantcetteapprochesontfournisdansleTIAp$

La figure 6.26 montre le résultat FFT où le signal a été réduit de 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. Comme c'est moins que le temps de capture disponible pour une trame de données de 128 points à une fréquence d'échantillonnage de 8 kHz, on s'attend à ce que cet algorithme s'exécute en temps réel sur le DSK.

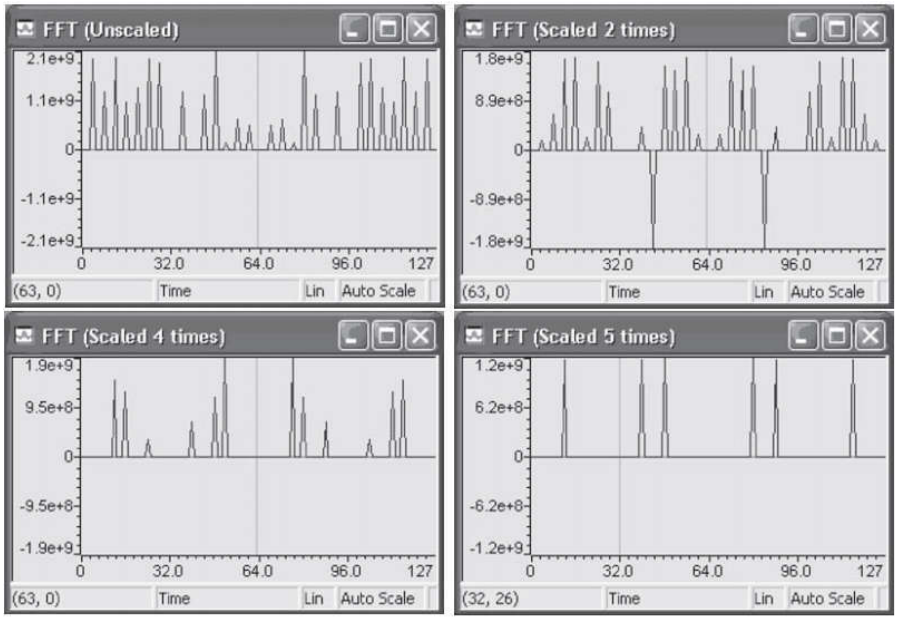


FIG. 6.26 : Mise à l'échelle pour obtenir une réponse d'amplitude FFT correcte

6.4.3 FFT en temps réel

Pour effectuer la FFT en temps réel, le programme de triple mise en mémoire tampon est utilisé. Une longueur de trame de 128 est considérée ici. La sortie est observée en arrêtant le processeur via CCS. La fonction d'animation de CCS ne peut pas être utilisée ici car elle ralentit le traitement et provoque le chevauchement des images.

Les modifications suivantes sont apportées au programme de triple mise en mémoire tampon pour exécuter l'algorithme FFT susmentionné en temps réel :

```
void wait_buffer(void)
{
      int n,k;
      short *p;

      while(index);

      p = input;
      input = output;
      output = intermediate;

      for (n=0; n<NUMPOINTS; n++)
      {
            x[n].imag = p[2*n + 1];       // x2(n) = g(2n + 1)
            x[n].real = p[2*n];           // x1(n) = g(2n)
      }

      radix4(NUMPOINTS, (short *)x, (short *)W4);
      digit_reverse((int *)x, IIndex, JIndex, count);
      x[NUMPOINTS].real = x[0].real;
      x[NUMPOINTS].imag = x[0].imag;

      split1(NUMPOINTS, x, A, B, G);
      G[NUMPOINTS].real = x[0].real - x[0].imag;
      G[NUMPOINTS].imag = 0;
      for (k=1; k<NUMPOINTS; k++){
            G[2*NUMPOINTS-k].real = G[k].real;
            G[2*NUMPOINTS-k].imag = -G[k].imag;
      }
      for (k=0; k<NUMDATA; k++){
            mag1[k] = (G[k].real*G[k].real) << 1;
            mag2[k] = (G[k].imag*G[k].imag) << 1;
            mag[k] = mag1[k] + mag2[k];
      }
      intermediate = p;
      HostTargetComm();
      while(!index);
}
```

La fonction wait_buffer() est modifiée avec les appels de fonction appropriés de sorte que lorsque le tampon d'entrée est plein, la transformation est calculée et envoyée à l'hôte via la FIFO.

La fonctionnalité du programme peut être vérifiée en connectant un générateur de fonctions à l'entrée de ligne. La capacité graphique de CCS peut être utilisée pour tracer le résultat de la FFT. En changeant la fréquence de l'entrée, les pointes de la réponse en fréquence se déplaceraient vers la gauche ou la droite en conséquence. La figure 6.27 illustre la sortie pour un signal sinusoïdal de 1 kHz et 2 kHz. Ces clichés instantanés sont capturés en arrêtant le processeur. L'entrée est ici mise à l'échelle en la décalant vers la droite de 4 bits.

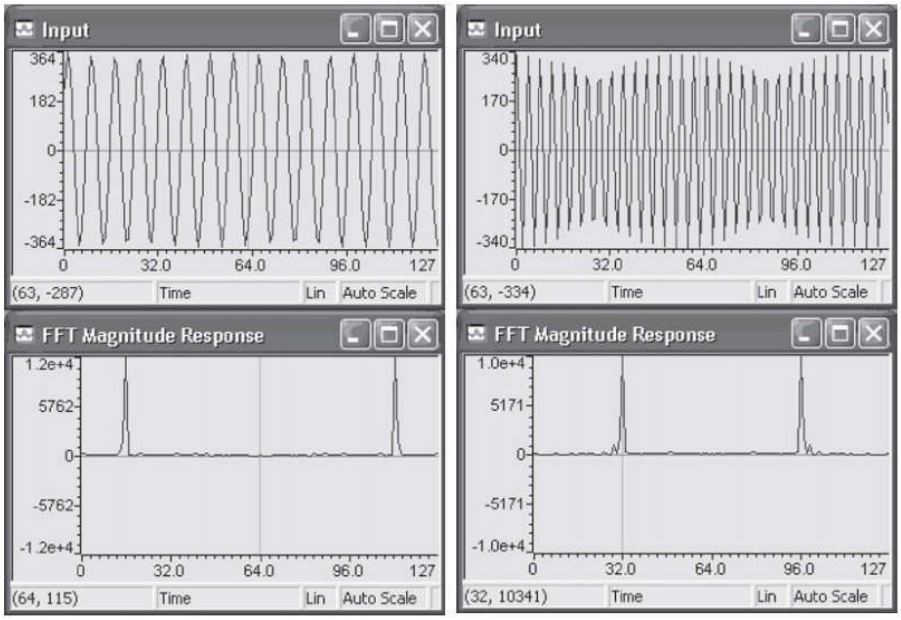


FIG. 6.27 : Réponse en amplitude FFT en temps réel (a) f = 1 kHz (b) f = 2 kHz

Références

1. R. Chassaing, D. Reay, Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK, John Wiley Sons, 2008.

2. D. Reay, Digital Signal Processing and Applications with the OMAP-L138 eXperimenter, John Wiley Sons, 2012.

3. T.B. Welch, C.H.G. Wright and M.G. Morrow, Real-Time Digital Signal Processing from MATLAB to C with TMS320C6x DSPs, CRC Press, 2012.

4. Steven A Tretter, Communication System Design Using DSP Algorithms, Springer 2008.

5. N. Dahnoun, Digital Signal Processing Implementation using the TMS320 C6000 DSP platform, Prentice Hall, 2000.

6. N. Kehtarnaz, N. Kim, Real Time Digital Signal Processing Based on TMS320C6000, Newnes, 2004.

7. N. Kehtarnaz, M. Keramat, DSP System Design using TMS320C6000, Prentice Hall, 2006.

8. S. W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing.

9. G. Baudoin et F. Virolleau, Les DSP : famille TMS320C54x. Développement d'applications.

10. L. Correvon, DSP et Temps réel : Application Industrielle, Haute Ecole d'Ingénierie du Canton de Neuchâtel.

11. P. Laspsley , J. Bier , A. Shoham, E. A. Lee, DSP Fundamentals : Architecture and Features, Berkley Design Technology, Inc, 1994.

12. Oktay Alkin, Digital Signal Processing : A Laboratory Approach using. PC-DSP, Prentice Hall.

13. Digital Control Applications with the TMS320 Family : Selected Application notes, Texas Instruments, 1991.

14. M. Pinard, Les DSP, famille ADSP 218X : Principes et applications, Dunod, 2000.

15. B. Bouchez, Applications audio-numériques des DSP : théorie et pratique du traitement numérique du son, Publitronic, 2003.

16. Texas Instruments, TMS320C6000 Code Composer Studio Tutorial (Rev. C).

17. Texas Instruments, Code Composer Studio Development Tools v3.3 Getting Started Guide (Rev. H).

18. Texas Instruments, TMS320C6000 Programmer's Guide (Rev. K),

19. Texas Instruments, TMS320C6000 CPU and Instruction Set Reference Guide (Rev. G).

20. Texas Instruments, TMS320C6000 Chip Support Library API Reference Guide (Rev. J).

21. Texas Instruments, TMS320C1X User's Guide. Juillet 1991.