UNIVERSITÉ SAVOIE MONT BLANC

DSP : Digital Signal Processors

Cours

Sylvain Montagny

sylvain.montagny@univ-smb.fr

04 79 75 86 86

1 Le traitement du signal numérique

1.1 La chaine de traitement

1.1.1 Les différents étages

La chaine de traitement de l'information peut être représentée par la Figure 1.

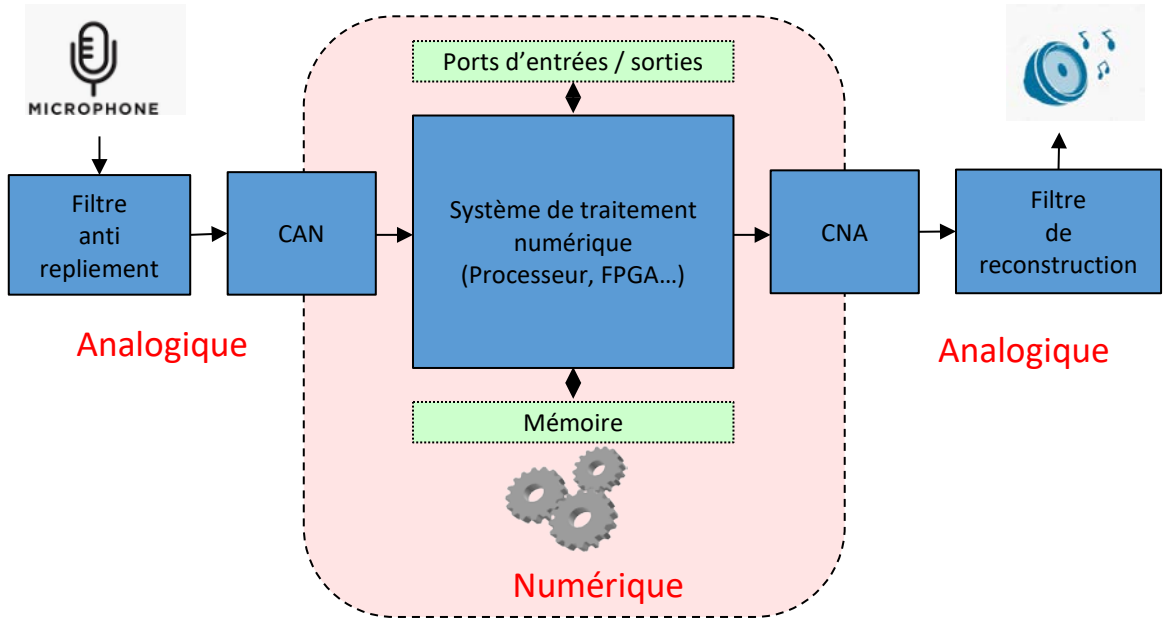


Figure 1 : Chaine de traitement numérique

Le filtre anti-repliement sert à supprimer les hautes fréquences du signal de façon à respecter le théorème de Shannon pour l'échantillonnage qui sera réalisé par le CAN.

Le filtre de reconstruction sert à supprimer les hautes fréquences du signal qui sont apparues lors de l'échantillonnage : le spectre est dupliqué à l'infini d'une période équivalente à la fréquence d'échantillonnage.

Exemple : Un signal sonore [0 – 20 kHz] doit être échantillonné à 8khz :

- Dessiner le spectre du signal **avant** l'échantillonnage
- Choisir la fréquence de coupure du filtre anti-repliement
- Dessiner le spectre du signal **après** le filtre anti-repliement
- Dessiner le spectre du signal **après** le CAN
- Choisir la fréquence de coupure du filtre de reconstruction
- Dessiner le spectre du signal **après** le filtre de reconstruction

Les filtres anti-repliement et les filtres de reconstruction sont (bien sûr) obligatoirement analogique.

1.1.2 Avantage du traitement numérique

Le traitement numérique possède de nombreux avantages :

- Il peut être fait en multitâche
- Il n'est pas contraint aux variations des valeurs de composants analogiques

1.2 Le filtrage numérique

1.2.1 Fonction de transfert et équation récurrente

La forme d'un filtre numérique FIR avec N coefficient est la suivante :

$$h(z) = b_0 + b_1.z^{-1} + \cdots + b_{N-1}.z^{-(N-1)}$$

La forme d'un filtre numérique IIR est la suivante (le nombre de coefficient du filtre est q+1+p) :

$$h(z) = \frac{b_0 + b_1.z^{-1} + \cdots + b_q.z^{-q}}{1 + a_1.z^{-1} + \cdots + a_p.z^{-p}}$$

Soit les deux filtres numériques suivants. Trouvez leurs fonctions récurrentes respectives, c'est-à-dire l'expression de l'échantillon de sortie en fonction des échantillons d'entrées précédents (FIR) ou des échantillons d'entrées et de sorties précédents (IIR).

$$H_{FIR}(z) = 0{,}21 + 0{,}47.z^{-1} + 0{,}47.z^{-2} + 0{,}21.z^{-3}$$

$$H_{IIR}(z) = \frac{0{,}98 + 0{,}29.z^{-1} + 0{,}29.z^{-2} + 0{,}98.z^{-3}}{1 - 0{,}57.z^{-1} + 0{,}42.z^{-2} + 0{,}05.z^{-3}}$$

La réponse en fréquence du filtre $H_{FIR}(z)$est donnée par le spectre de la Figure 2, et celui du filtre $H_{IIR}(z)$ par le spectre de la Figure 3.

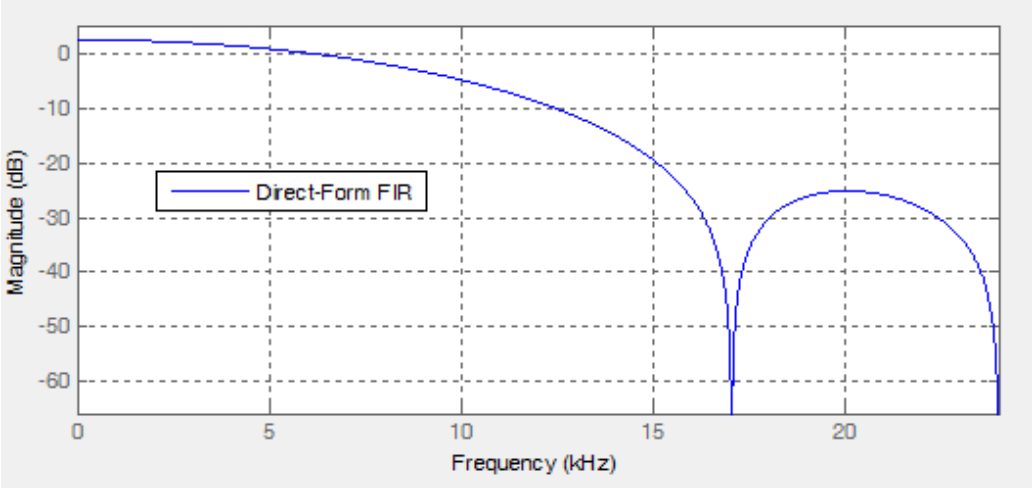


Figure 2 : Réponse en fréquence du filtre FIR

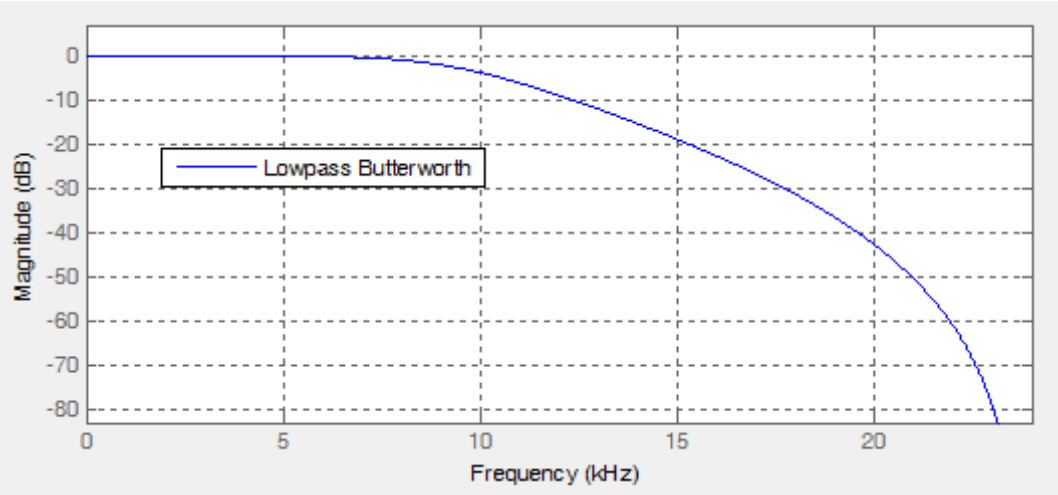


Figure 3 : Réponse en fréquence du filtre IIR

1.2.2 Schéma de fonctionnement d'un filtre numérique

La représentation graphique d'un filtre numérique FIR est représentée à la Figure 4. Les coefficients seront maintenant sont appelés "coeff".

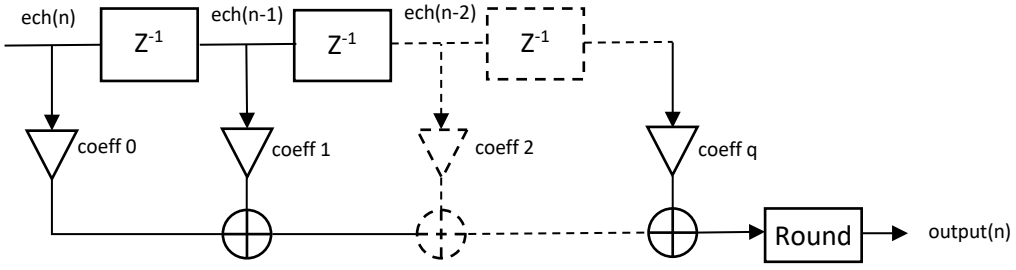


Figure 4 : Représentation graphique d'un filtre FIR

En vous aidant de la Figure 4 , donner le schéma de la représentation graphique du filtre $H_{FIR}(z)$ et $H_{IIR}(z)$ présentés au chapitre 1.2.1.

On caractérise la complexité d'un filtre numérique par le nombre de cellules multiplication / accumulation qu'il contient. Combien de cellules multiplication / accumulation contiennent les filtres FIR et IIR précédents ? Quel est le rapport entre le nombre de cellule et le nombre de coefficients ?

1.2.3 Actions à réaliser et objectifs

En regardant le schéma d'un filtre numérique, nous pouvons noter les actions à réaliser. Nous prenons l'exemple d'un filtre FIR avec N coefficients.

```
while ( 1 ) {
    ■ Récupération d'un nouvel échantillon (CAN) et stockage en mémoire

        for ( i = 0 ; i < N ; i ++ ) {          // Boucle N fois
            ■ Lectures des deux opérandes : échantillon et coefficient
            ■ Multiplication des deux opérandes : coefficient * échantillon
            ■ Accumulation
            ■ Gestion des pointeurs pour l'accès aux prochains échantillons et coefficients
            ■ Gestion de la boucle for : incrémentation de i, test de sa valeur et rebouclage
        }
    ■ Envoi du résultat du calcul sur le CNA
    ■ Vieillissement des échantillons
    ■ Gestion de la boucle infinie while(1)
}
```

Figure 5 : Algorithme de réalisation d'un filtre numérique FIR

Objectifs d'un processeur de traitement du signal :

- Récupérer le nouvel échantillon (CAN) en temps caché (DMA)
- Réaliser chaque itération de la boucle for en 1 cycle, soit N cycles pour un filtre à N coeff.
- Réaliser le vieillissement des échantillons rapidement : Buffer linéaire ou circulaire
- Envoyer le résultat (CNA) en temps caché (DMA)

1.3 Le vieillissement des échantillons pour le filtrage numérique

Soit le filtre numérique FIR suivant :

$$output(n) = coeff_0.ech(n) + coeff_1.ech(n-1) + \cdots + coeff_{N-1}.ech(N-1)$$

A chaque période d'échantillonnage, un nouvel échantillon arrive et doit être stocké dans le tableau. Il faut alors faire vieillir les autres échantillons.

On considère les tableaux d'échantillons et de coefficients comme le montre la Figure 6.

@ du tableau	Echantillons		Coefficients
0	ech_0		$coeff_0$
1	ech_1	**X**	$coeff_1$
2	ech_2		$coeff_2$
…	…		…
…	…	**Multiplication coeff x ech**	…
N-2	ech_{N-2}		$coeff_{N-2}$
N-1	ech_{N-1}		$coeff_{N-1}$

Figure 6 : Tableau d'échantillons et de coefficients

1.3.1 Buffer linéaire

Dans le cas de l'utilisation d'un buffer linéaire, tous les échantillons du tableau "Echantillons" sont décalés vers le bas d'une position, puis le nouvel échantillon est positionné dans la case qui s'est libérée en haut du tableau. La Figure 7 représente le tableau d'échantillon juste après le décalage et l'arrivée du nouvel échantillon.

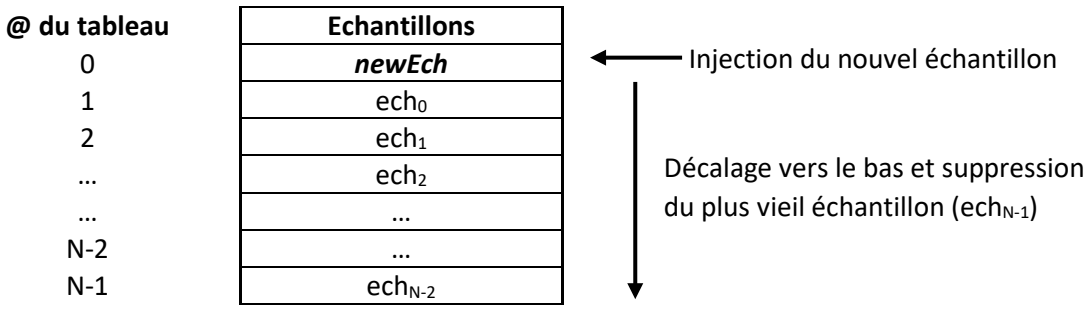


Figure 7 : Réalisation d'un buffer linéaire

Compléter le code suivant pour réaliser le calcul complet du filtre numérique en langage C.

```
float coeff[N] = {  ,  ,  ... ,  ,  };
uint16_t ech[N];
uint16_t newEch;
uint16_t output = 0;


void main(void){
    while(1){
        // Reception d'un echantillon depuis le CAN
        HAL_SAI_Receive(&newEch);

        // Stockage dans le tableau d'échantillon
        // Réinitialiser output a 0.
        // A COMPLETER

        // Calcul du filtre numérique (cas du Buffer Linéaire)
        // A COMPLETER

        //Envoi du résultat sur le CNA
        HAL_SAI_Transmit(&output);

        // Vieillissement du Buffer linéaire : décalage
        // A COMPLETER
}
```

1.3.2 Buffer circulaire

Dans le cas d'un buffer circulaire, le nouvel échantillon remplace uniquement le plus ancien. Les figures suivantes représentent le tableau d'échantillon au temps T = t (Figure 8), c'est-à-dire avant l'apparition du nouvel échantillon et T = t + 1 (Figure 9), c'est-à-dire lorsque le nouvel échantillon a été injecté dans le tableau en remplacement du plus ancien.

@ du tableau	Echantillons		Coefficients
0	…		$coeff_0$
1	ech_{N-2}		$coeff_1$
2	ech_{N-1}		$coeff_2$
…	ech_0		…
	ech_1		…
N-2	…		$coeff_{N-2}$
N-1	…		$coeff_{N-1}$

Figure 8 : Tableau d'échantillons et de coefficients à T = t

Dans la Figure 8, l'échantillon le plus ancien est ech_{N-1}. A t = t + 1, il sera remplacé par le nouvel échantillon nommé newEch.

@ du tableau	Echantillons		Coefficients
0	…		$coeff_0$
1	ech_{N-2}		$coeff_1$
2	***newEch***		$coeff_2$
…	ech_0		…
…	ech_1		…
N-2	…		$coeff_{N-2}$
N-1	…		$coeff_{N-1}$

Figure 9 : Tableau d'échantillons et de coefficients à T = t + 1

Au prochain cycle, le nouvel échantillon remplacera ech_{N-2}, etc…

Vous devez particulièrement faire attention dans quel sens votre tableau d'échantillon fonctionne. Dans notre cas, lorsque le tableau était vide, nous avons injecté le premier échantillon à la case N-1. Le second à la case N-2, etc… Vous pouvez tout à fait commencer par le haut : premier échantillon à l'adresse 0, le second à l'adresse 1, etc… Il faut simplement penser à ajuster votre calcul du filtre numérique en conséquence lorsque vous multiplier les coefficients et les échantillons.

Compléter le code suivant pour réaliser le calcul complet du filtre numérique en langage C.

```
float coeff[N] = {  ,  ,  ... ,  ,  };
uint16_t ech[N];
uint16_t newEch;
uint16_t output = 0;
uint16_t index = N-1;

void main(void){
    while(1){
        // Reception d'un echantillon depuis le CAN
        HAL_SAI_Receive(&newEch);

        // Stockage dans le tableau d'échantillon
        // Réinitialiser output a 0.
        // A COMPLETER

        // Calcul du filtre numérique (cas du Buffer Circulaire)
        // A COMPLETER
```

```
            // Vieillissement du Buffer circulaire : MAJ de l'index
            // A COMPLETER

            //Envoi du résultat sur le CNA
            HAL_SAI_Transmit(&output);
        }
    }
```

1.3.3 Avantage et inconvénient des buffer linéaires et circulaires

D'après les codes que vous avez écrit dans le cas d'un buffer linéaire et dans le cas d'un buffer circulaire, nous pouvons faire le résumé suivant :

	Vieillissement des échantillons	**Calcul du filtre**
Buffer Linéaire	*Complexe*	*Simple*
Buffer circulaire	*Simple*	*Complexe*

2 Représentation numérique du signal

Afin de bien coder les algorithmes de traitement du signal, il est indispensable de bien comprendre:

- Le codage des nombres
- Les calculs réalisés à l'intérieur du processeur

2.1 Codage des nombres entiers

2.1.1 Le codage des entiers non signés

A chaque chiffre est affecté un poids exprimé en puissance de 2 : $Nombre = \sum_{i=0}^{N-1} b_i 2^i$, b étant la valeur binaire (0 ou 1) du poids concerné et N le nombre de bits du Nombre.

Exemple : $(101)_2 = 1.2^2 + 0.2^1 + 1.2^0 = (5)_{10}$

Donner la plage de codage des nombres entiers non signés en fonction du nombre de bit N

2.1.2 Le codage des entiers signés

Les nombres signés utilisent une représentation en Complément A2 qui possède des propriétés beaucoup plus intéressantes qu'une représentation signe/valeur absolue.

$$Nombre = -b_{N-1}.2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

Exemple : $(101)_{CPLA2} = -1.2^2 + 0.2^1 + 1.2^0 = (-3)_{10}$

Donner la plage de codage des nombres entiers signés en fonction du nombre de bit N. Donner la représentation binaire des nombres de -4 à 3 du tableau suivant. Quelle opération binaire nous fait passer de 3 à -4.

Nombre	Codage
3	*011*
2	*010*
1	*001*
0	*000*
-1	*111*
-2	*110*
-3	*101*
-4	*100*

Cette propriété est intéressante mais devra être contrôlée. Dans certains cas, nous souhaitons une saturation du signal au valeur max et min lorsqu'il y a un dépassement, dans d'autres cas, nous préférons un rebouclage.

2.2 Codage des nombres réels

La représentation des nombres réels doit répondre à deux exigences contradictoires :

- Précision : L'intervalle entre deux valeurs codées doit être le plus petit possible
- Dynamique : La plage des valeurs codées doit être la plus grande possible

2.2.1 Le codage en virgule fixe

Le codage en virgule fixe représente un nombre réel en partant d'un nombre entier signé en complément A2. On rajoute simplement dans la représentation une virgule **virtuelle** permettant de séparer la partie entière et la partie décimale.

- Soit **N** le nombre de bit total de la représentation
- Soit **k** le nombre de bit de la partie décimal
- Soit **m** le nombre de bit de la partie entière

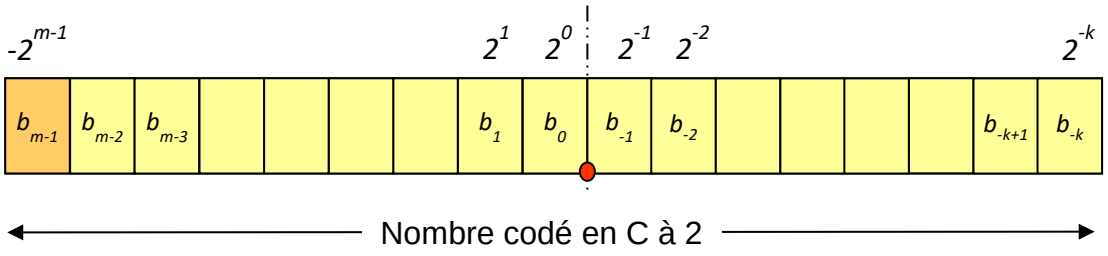


Figure 10 : Représentation d'un nombre réel en virgule fixe

$$Nombre = -b_{m-1}.2^{m-1} + \sum_{i=-k}^{m-2} b_i 2^i$$

La valeur de k doit obligatoirement être connue. On appelle une représentation en Q_k, la représentation en virgule fixe avec k bits réservés pour la partie décimale. Parfois, pour être plus explicite, la représentation est donnée en $Q_{m.k}$.

Donner la plage de codage des nombres réels codés en virgule fixe en fonction de N et k.

Soit la représentation binaire suivante : 0101 1101. Quelle est sa valeur si on considère ce nombre en représentation Q_2, Q_4, Q_7?

- Q_2 : 23.25
- Q_4 : 5.8125
- Q_7 : 0.7265625

Soit une représentation d'un nombre réel en Q_5 sur 8 bits. Donner la plus petite valeur, la plus grande valeur, ainsi que l'erreur maximale réalisée sur le codage.

Sur 8 bits, quel format Q_k faut-il choisir pour optimiser au mieux le codage d'un nombre en virgule fixe pour les plages de nombres suivants. Donner la précision (+/-) pour chacun des codages choisis.

- -1 ≤ Nombre <1
- -6 ≤ Nombre ≤ -4
- 200 ≤ Nombre ≤ 200
- -0.1 ≤ Nombre ≤ 0

Donner la représentation en virgule fixe sur 8 bits, des nombres suivants dans **un format Q_k commun**.

- 98.895
- 0.01298

2.2.2 Le codage en virgule flottante

La notation en virgule flottante est très similaire à la notation scientifique en base 10. En base 10, nous fixons un nombre de bits significatif, puis nous représentons le nombre sous la forme :

$$Nombre = mantisse.10^{\exp osant}$$

Note : Dans la notation scientifique en base 10 : $1 \le |mantisse| < 10$.

Donner la représentation scientifique **en base 10** des nombres suivants si on garde 3 digits significatifs pour la mantisse.

- 98.895
- 10.1
- 0.01298
- -128000

En binaire, la représentation est similaire mais en base 2 :

$$Nombre = mantisse.2^{\exp osant}$$

Note : Dans la notation scientifique en base 2 : $1 \leq |mantisse| < 2$.

- **La mantisse** est un nombre réel codé sur M bits en virgule fixe
- **L'exposant** est un nombre entier signé codé sur E bits en complément A2.

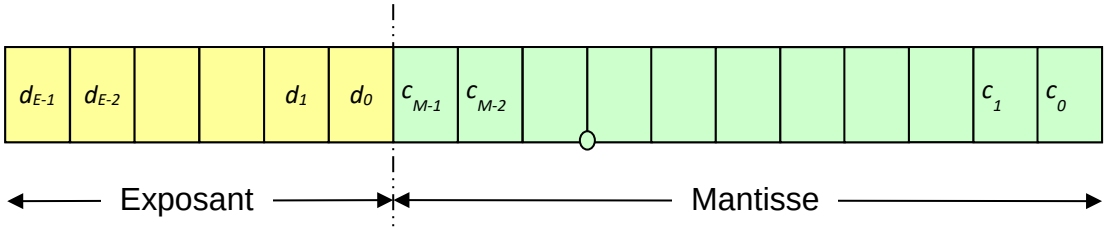


Figure 11 : Représentation binaire d'un nombre en virgule flottante

Dans le cas d'une représentation en virgule flottante avec 4 bits pour la mantisse et 4 bits pour l'exposant, donner la représentation binaire des nombres suivants :

- 98.895
- 0.01298

La norme IEEE définissant la représentation en virgule flottante est très proche de celle que nous venons de réaliser et les propriétés restent similaires.

2.3 L'erreur de quantification

L'erreur de quantification est l'erreur entre la valeur réelle du nombre et la valeur codée en binaire. On considère l'erreur de quantification comme un bruit qui est rajouté au signal. Les principales conséquences de l'erreur de quantification sur les filtres numériques sont :

- Au mieux, un non-respect du gabarit du filtre numérique
- Au pire, le filtre peut même devenir instable

L'erreur relative permet d'exprimer l'erreur en %. C'est le rapport entre l'erreur de quantification et la valeur exact du nombre à coder :

- $\varepsilon_{relative}(\%) = \dfrac{\varepsilon_{quantification}}{Nombre} x100$

Nous travaillerons sur les nombres vus précédemment (98,895 et 0,01298) codés soit en virgule fixe (en Q_0), soit en virgule flottante (4 bits pour l'exposant et 4 bits pour la mantisse). Nous allons étudier l'erreur relative maximale lorsque nous travaillons avec des petits nombres (autour de 0,01298) et des grands nombres (autour de 98,895). Remplir les colonnes **Codage**, $\varepsilon_{absolue}$ et $\varepsilon_{relative}$ du tableau suivant :

Codage Virgule Fixe					
Nombre	**Codage**	**$\varepsilon_{absolue}$**	**$\varepsilon_{relative}$**	**$\varepsilon_{absolue\ max}$**	**$\varepsilon_{relative\ max}$**
98.895	99	0.105	0.11 %	+/- 0.5	0.5 %
0.01298	0	0.01298	100 %	+/- 0.5	3852 %
Codage Virgule Flottante					
Nombre	**Codage**	**$\varepsilon_{absolue}$**	**$\varepsilon_{relative}$**	**$\varepsilon_{absolue\ max}$**	**$\varepsilon_{relative\ max}$**
98.895	96	2.895	2.93 %	+/- 8	8.09 %
0.01298	0.01367	0.00069	5.31 %	+/- 0.00098	7.52 %

Tableau 1 : Erreur absolue et relative suivant les méthodes de codage

2.3.1 Cas de la représentation en virgule fixe

Donner l'erreur relative maximale pour des petits nombres (autour de 0,01298) et des grands nombres (autour de 98,895) dans le cas d'une représentation en virgule fixe Q_0 sur 8 bits. Remplir les colonnes $\varepsilon_{absolue\ max}$, et $\varepsilon_{relative\ max}$ du Tableau 1 dans le cas de la représentation en virgule fixe.

Sur la Figure 12, donner l'évolution de cette erreur relative en fonction du nombre à coder.

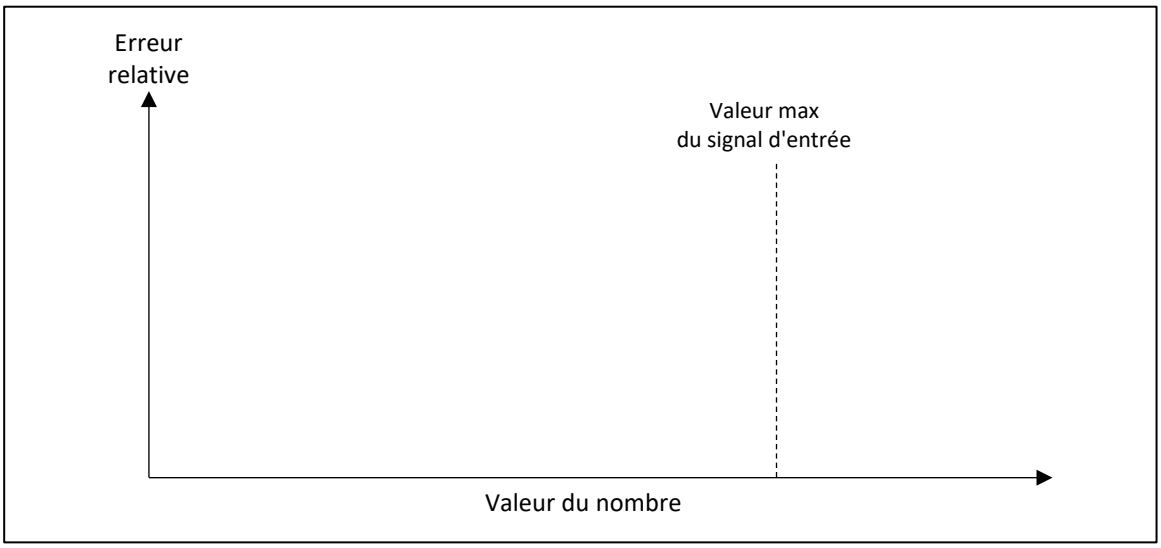


Figure 12 : Evolution de l'erreur relative en fonction du nombre (en virgule fixe)

2.3.2 Cas de la représentation en virgule flottante

Donner l'erreur relative maximale pour des petits nombres (autour de 0,01298) et des grands nombres (autour de 98,895) dans le cas d'une représentation en virgule fixe flottante 4 bits pour l'exposant et 4 bits pour la mantisse). Remplir les colonnes $\varepsilon_{absolue\ max}$, et $\varepsilon_{relative\ max}$ du Tableau 1 dans le cas de la représentation en virgule fixe.

Sur la Figure 12, donner l'évolution de cette erreur relative en fonction du nombre à coder.

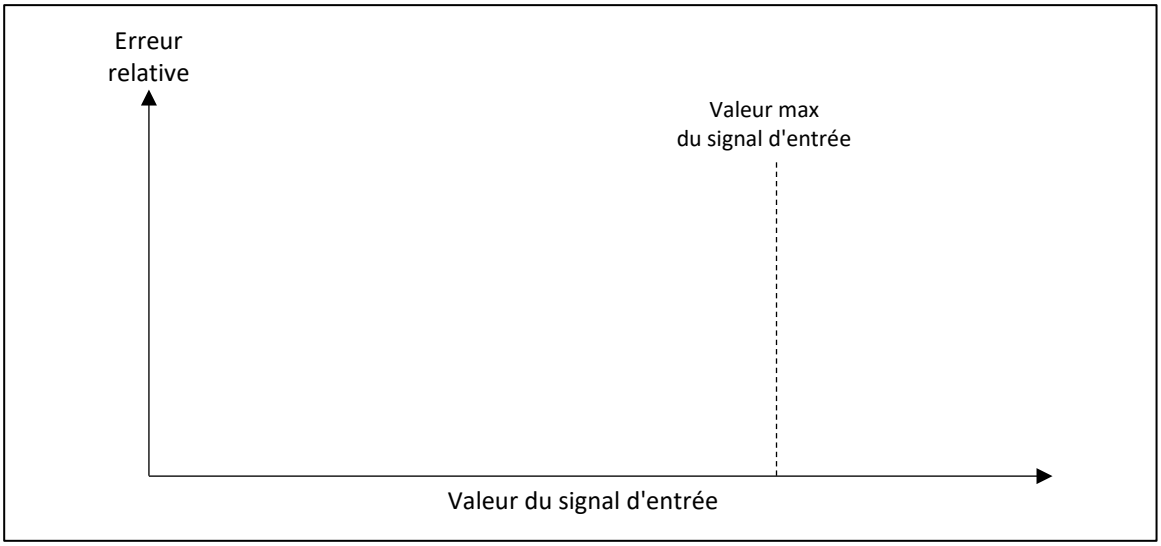


Figure 13 : Evolution de l'erreur relative en fonction du nombre (en virgule flottante)

2.4 Relation entre le codage des nombres et le type de variable

Il y a une relation directe entre le codage des nombres et le type de variable utilisé en langage C.

Dans le tableau suivant, donner les plages de valeurs de chacune des variables.

Nom du type	Signification	Codage	Plage de valeur
char	Entier signé	8 bits	*-128 à 127*
unsigned int	Entier non signé	32 bits	*0 à 2^{32}-1*
int	Entier signé	32 bits	*-2^{31} à 2^{31}-1*
float	Réel signé	32 bits 24 bits de mantisse 8 bits d'exposant	*-3,4 x 10^{38} à 3,4 x 10^{38}*
double	Réel signé	64 bits 53 bits de mantisse 11 bits d'exposant	*-1,7 x 10^{308} à 1,7 x 10^{308}*

Figure 14 : Les différents types de variables utilisés en langage C

3 Les processeurs de traitement du signal

3.1 Les deux types de processeur

Nous avons vu au chapitre 2.2 deux façons de coder les nombres réels : le codage en **virgule fixe** et le codage en **virgule flottante**. Ces deux méthodes sont à l'image des deux technologies de processeurs.

3.1.1 DSP à virgule fixe

Ces processeurs ne possèdent pas d'unité arithmétique pour les nombres flottants. Cela ne signifie pas qu'ils ne peuvent pas réaliser des calculs en flottant, mais ces calculs font appel à des librairies logicielles de gestion des nombres flottants. Le temps de calcul est très ralenti et l'occupation mémoire du programme est augmentée.

Avantages :

- L'architecture du processeur est plus simple
- Le processeur consomme moins
- Le processeur est moins cher

Inconvénients :

- L'arithmétique a virgule fixe est beaucoup plus complexe, notamment pour la gestion des débordements et du recadrage des données. Nous étudierons ces différents aspects au chapitre 4.2 et 4.4.

> **Dans un DSP à virgule fixe (qui ne possède pas d'unité d'arithmétique flottant), nous ne devrions pas utiliser de type float car cela réduit considérablement les performances de l'algorithme.**

Il est important de savoir qu'énormément de **microcontrôleurs** ne possèdent pas de gestion de l'arithmétique flottante appelée FPU (Floating Point Unit). Cependant la majorité des processeurs en possède une.

3.1.2 DSP à virgule flottante

Ces processeurs possèdent une unité spécifique pour le calcul flottant (FPU) :

- Dans un DSP à virgule flottante (FPU single precision), les calculs en float s'exécutent à la même vitesse que les calculs avec les nombres entiers.
- Dans un DSP à virgule flottante (FPU double precision), les calculs en double s'exécutent à la même vitesse que les calculs avec les nombres entiers ou float.

> **Dans un DSP à virgule flottante Single Precision, nous ne devrions pas utiliser de type double car cela réduit considérablement les performances de l'algorithme.**

Avantages :

La programmation de l'algorithme est très simplifiée car nous n'avons pas besoin de gérer la dynamique des données sur lesquels nous travaillons.

Inconvénients :

- L'architecture du processeur est plus complexe
- Le processeur consomme plus
- Le processeur est plus cher

3.2 L'unité arithmétique et Logique (UAL)

3.2.1 Multiplieur câblé

Nous avons vu que l'opération la plus importante pour la réalisation d'un filtre numérique était la multiplication (associé à une accumulation). Celle-ci doit être réalisée de façon matérielle. Les processeurs de traitement du signal possèdent donc des multiplieurs câblés.

3.2.2 Les bits de garde de l'accumulateur

- Les multiplications de deux nombres codés sur n bits donne un résultat sur 2n bits.
- L'addition de deux nombres de 2n bits donne un résultat sur 2n + 1 bits.

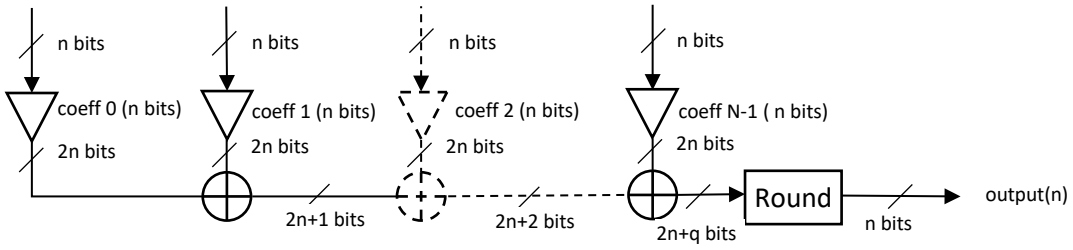


Figure 15 : Format des nombres dans la chaine de traitement

Exemple : Les multiplications de 2 nombres de 16 bits donnent un résultat sur 32 bits. Les additions successives donnent des résultats sur 33, 34, 35… bits. En conséquence, les DSP (par exemple le TMS320C54 de chez Texas Instruments) prévoient des accumulateurs de 40 bits pour garantir la précision.

3.2.3 Le registre à décalage

A la fin du calcul, le résultat doit être tronqué sur le format initial. Il ne faut donc garder que les bits les plus représentatifs. Cela est fait par un registre à décalage appelé registre à barillet dans les DSP.

3.3 Le pipeline

3.3.1 Objectif et fonctionnement

Contrairement à ce que nous pourrions imaginer, l'objectif n'est pas d'augmenter la vitesse de traitement d'une instruction mais de faire en sorte que plus d'instructions soit traitées en un temps donné.

Pour comprendre le principe, il faut analyser l'ensemble des étapes permettant de parvenir au traitement d'une instruction. Nous prendrons l'exemple d'un processeur DSP Texas Instrument TMS320C54.

Stage	Description
P - Prefetch	Incrémentation du compteur ordinal
F - Fetch	Lecture du code de l'instruction en mémoire
D - Decode	Décodage de l'instruction
A - Access	Calcul des adresses des opérandes et du résultat
R –Read	Lecture des opérandes
X – Execute	Exécution de l'instruction et écriture du résultat

Figure 16 : Les 6 étages du pipeline d'un DSP TMS320C54

La Figure 17 représente le déroulement de ces étapes temporellement.

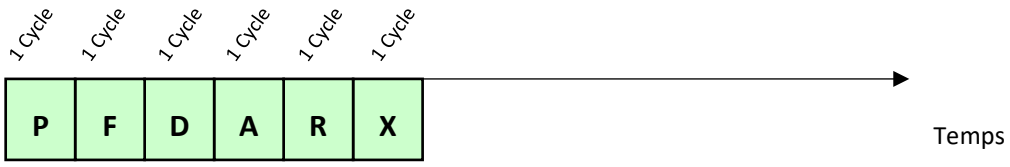


Figure 17 : Représentation temporelle des étapes d'une instruction sans pipeline

Représenter sur la Figure 17 le déroulement d'une deuxième instruction si nous considérons pour l'instant que le processeur n'utilise pas de pipeline. En combien de cycle est réalisé chacune des instructions ?

Représenter sur la Figure 18 le déroulement de 5 autres instructions si nous considérons que le processeur utilise maintenant un pipeline. Combien de temps mets chacune des instructions à s'exécuter. Combien d'instructions sont exécutées à chaque cycle à partir du 6ème cycle ?

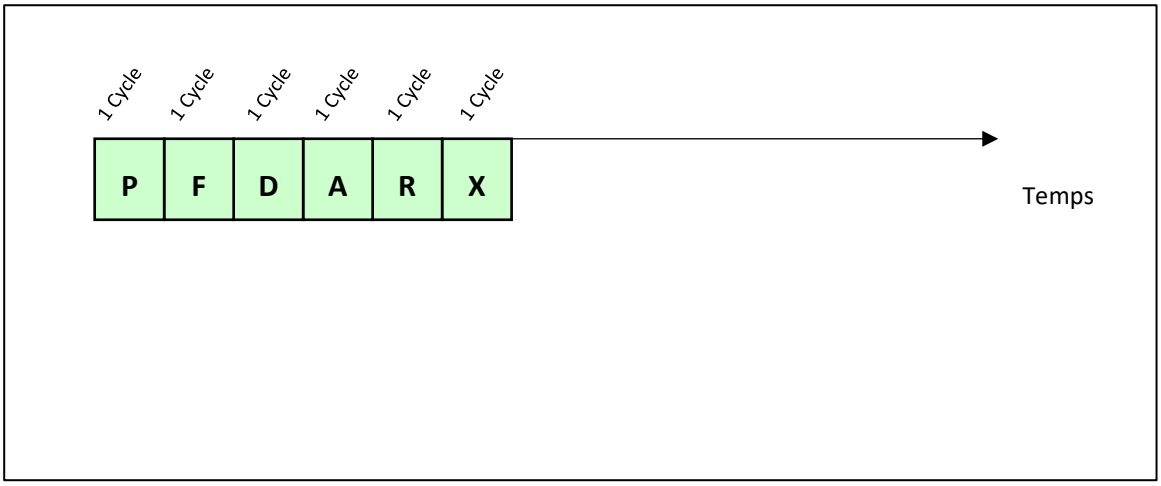


Figure 18 : Représentation temporelle des étapes d'une instruction avec pipeline

3.3.2 Vidange du pipeline

Dans le cas général, la prochaine instruction qui est récupérée en mémoire est celle située à l'adresse + 1 dans la mémoire flash. C'est donc cette instruction qui sera insérée dans le pipeline.

On considère maintenant un cas pratique représentant l'appelle d'une fonction (Instruction CALL) qui constitue un saut en mémoire.

Soit le programme suivant :

```
PROGRAM : Instruction 1
          Instruction 2 // CALL FUNCTION
          Instruction 3
          Instruction 4
          Instruction 5
          Instruction 6
          Instruction 7
          Instruction 8
          Instruction 9


FUNCTION  Instruction 10
          Instruction 11
          Instruction 12 // RETURN
```

Représenter sur la Figure 19, le chargement des instructions dans le pipeline dans le cas du programme précédent. Mettez en évidence la vidange du pipeline.

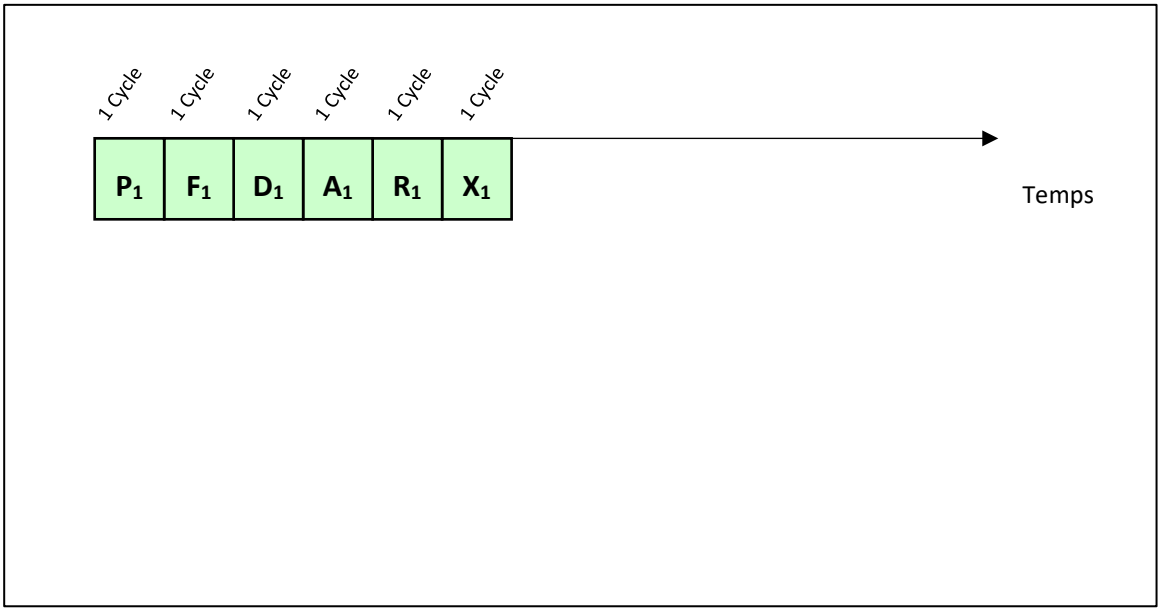


Figure 19 : Exemple d'une vidange de pipeline

Les vidanges du pipeline dues à des sauts en mémoire sont anticipées (et **évitées** !) avec le plus de réussite possible si le processeur possède des prédicteurs de branchement. Un ARM cortex M3 ne possède pas de prédicteur de branchement. Un ARM cortex M7 en possède un.

3.3.3 Les aléas dans le pipeline

Chaque étage du pipeline utilise un certain nombre de circuit du processeur pour réaliser l'action à laquelle il est préposé. La Figure 20 résume les circuits utilisés par chaque étage.

Stage	Description	Circuit utilisé
P - Prefetch	Incrémentation du compteur ordinal	Compteur ordinal
F - Fetch	Lecture du code de l'instruction en mémoire	Mémoire Instruction
D - Decode	Décodage de l'instruction	Décodeur
A - Access	Calcul des adresses des opérandes et du résultat	UAL
R –Read	Lecture des opérandes	Mémoire donnée
X – Execute	Exécution de l'instruction et écriture du résultat	UAL et mémoire donnée

Figure 20 : Utilisation des ressources du processeur à chaque étage du pipeline.

On imagine que le programmeur d'une l'application a utilisé la même mémoire pour les instructions et pour les données. Quels étages du pipeline ne pourront plus se dérouler simultanément ? Représenter sur la Figure 21 le pipeline et mettre en évidence les retards engendrés.

```
PROGRAM : Instruction 1
          Instruction 2
          Instruction 3
          Instruction 4
          Instruction 5
          Instruction 6
          Instruction 7
```

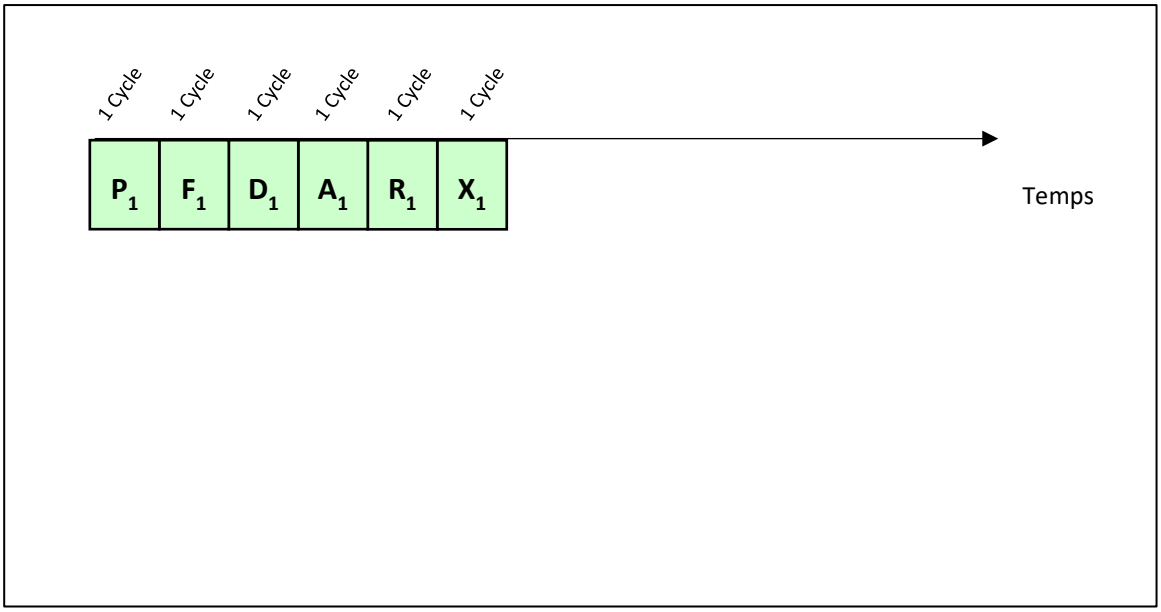


Figure 21 : Exemple d'aléas dans le pipeline

Les aléas du pipeline sont très dommageables pour la rapidité de l'exécution de l'algorithme. Il convient donc au programmeur de vérifier que l'emplacement de l'ensemble de son code soit pertinent vis-à-vis de l'architecture utilisée.

3.4 La gestion des boucles

On considère deux tableaux représentant les échantillons ech[ ] et les coefficients coeff[ ] d'un filtre FIR.

@ du tableau	Coefficients	Echantillons
0	$coeff_0$	ech_0
1	$coeff_1$	ech_1
2	$coeff_2$	ech_2
…	…	…
N-1	$coeff_{N-1}$	ech_{N-1}

Redonner l'expression du calcul du filtre numérique en langage C en utilisant une boucle for.

Cette boucle logicielle intègre une instruction essentielle en traitement du signal, il s'agit d'une instruction de multiplication et d'accumulation : MAC [**M**ultiply **A**nd **A**ccumulate] :

```
A = A + coeff * ech
```

```
MAC coeff_ptr, ech_ptr, A
```

L'instruction MAC n'est pas suffisante pour réaliser l'ensemble de l'algorithme. Il faut aussi :

- Incrémenter les index des opérandes dans les tableaux (coefficients et échantillons)
- Vérifier que la boucle est réaliser N fois

Ces actions alourdissent la boucle de traitement. Un DSP doit trouver un moyen d'optimiser la réalisation de ces deux opérations sans pénaliser le temps d'exécution.

3.4.1 Auto incrémentation des pointeurs d'adresses

Dans l'exemple simple que nous avons réalisé, l'incrémentation des index est de 1 à chaque itération. Cette opération (aussi simple soit-elle) doit être réalisée avec une UAL. Elle prendra un cycle supplémentaire dans le calcul.

L'idée des architecture DSP est de dédier une nouvelle UAL très simple qui réalisera ces calculs automatiquement sans utiliser l'UAL principale du DSP. Avec cette méthode, à chaque instruction MAC réalisée, les pointeurs ech_ptr et coeff_ptr seront automatiquement incrémenté et prêt pour la prochaine instruction MAC.

```
MAC coeff_ptr+, ech_ptr+, A
```

3.4.2 Gestion des boucles par blocage du PC

On considère que l'auto incrémentation des pointeurs est active. Nous rappelons que le calcul à effectuer est N fois une instruction MAC (filtre à N coefficients) :

@ du tableau	Echantillons
0	ech_0
1	ech_1
2	ech_2
…	…
N-2	ech_{N-2}
N-1	ech_{N-1}

X

Coefficients
$coeff_0$
$coeff_1$
$coeff_2$
…
$coeff_{N-2}$
$coef_{N-1}$

Au lieu de faire une boucle de N itération qui demande d'incrémenter l'index de la boucle for et de vérifier sa valeur, il est possible de bloquer le PC (Program Counter) afin de réaliser la même opération N fois. Dans les DSP, cette astuce est rendue possible par la présence d'instruction REPEAT.

```
REPEAT N
MAC coeff_ptr+, ech_ptr+, A
```

3.4.3 Adressage circulaire (modulo)

L'utilisation de l'auto incrémentation et du blocage du PC tel que nous l'avons vu fonctionne bien avec le buffer linéaire. En revanche, dans le cas des buffers circulaires, la répétition de N fois de l'instruction MAC nous ferait sortir du tableau des échantillons. Il faudrait alors pouvoir faire en sorte que le pointeur sur les échantillons réalise une opération modulo N pour revenir automatiquement à zéro après l'index N-1.

```
REPEAT N
MAC coeff_ptr+, ech_ptr+%N, A
```

Cela nécessite la présence de buffer circulaire matériel au sein du processeur. Seul les vrais DSP possède ce type de circuit. Les DSC (**D**igital **S**ignal **C**ontroller) n'en possèdent pas. Pour les DSC, l'utilisation des buffers circulaires est donc beaucoup moins intéressante que pour les DSP. D'autres techniques sont donc utilisées pour essayer de s'en approcher, sans pour autant y parvenir complètement.

3.5 Les bus d'accès aux mémoires

Les DSP utilisent une architecture de Harvard avec une duplication des bus d'accès aux mémoires programme et donnée.

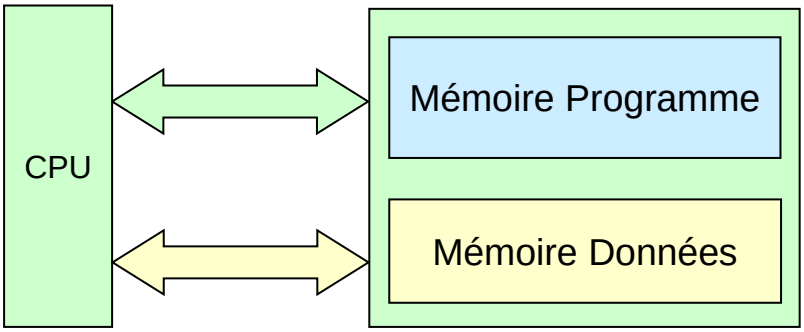


Figure 22 : Architecture de Harvard

D'après les étages du pipeline présenté Figure 16 dans le processeur TMS320C54, combien d'accès mémoire simultanées avons-nous besoin? Compléter le schéma de l'architecture de Harvard modifiée du DSP Figure 23.

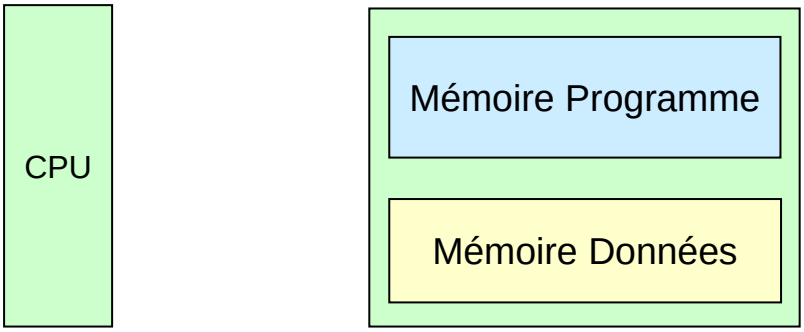


Figure 23 : Architecture de Harvard modifiée dans un TMS320C54

3.6 Résumé

while (1) {

- Récupération d'un nouvel échantillon (CAN) et stockage en mémoire

for (i = 0 ; i < N + 1 ; i ++) { // Boucle N +1 fois

- Lectures des deux opérandes : échantillon et coefficient
- Multiplication des deux opérandes : coefficient * échantillon
- Accumulation
- Gestion des pointeurs pour l'accès aux prochains échantillons et coefficients
- Gestion de la boucle for : incrémentation de i, test de sa valeur et rebouclage

}

- Envoi du résultat du calcul sur le CNA
- Vieillissement des échantillons
- Gestion de la boucle infinie while(1)

}

Figure 24 : Algorithme de réalisation d'un filtre numérique FIR

4 Réalisation des calculs dans le processeur

Pour la suite nous adopterons les dénominations suivantes :

- a : 1er opérande
- b : 2ème opérande
- sign( ) : Sign de l'opérante
- r : Résultat
- n : Nombre de bits
- k : Nombre de bits après la virgule

4.1 L'addition des nombres entiers

4.1.1 Addition en entier non signés

Exemple sur 4 bits. Réaliser le calcul de 12 + 5 = 17 en binaire

- nr = max(na,nb) + 1 (on a une éventuelle retenue)

> **En traitement du signal, on utilise très rarement le codage en unsigned. Nous ne traiterons plus ce cas par la suite.**

4.1.2 Addition de 2 entiers signés de signes opposés

Si sign(a) != sign(b) alors il n'y a **pas de débordement possible**.

- Faire le test avec -8 + x $x \in [0,7]$
- Faire le test avec 7 + x $x \in [-8,-1]$

Dans ces cas, on peut s'affranchir de s'occuper du bit de retenu.

> **Le nombre maximum de bits du résultat est donc : nr = max(na , nb)**

4.1.3 Addition de 2 entiers signés de même signe

Tout d'abord un calcul simple sans débordement : 4 + 3 = 7

Mais il faut savoir que si sign(a) = sign(b) alors il y a **un débordement possible**. En effet, le résultat du calcul de 7 + 3 suivant est faux :

$$\begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{1} & \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 \end{array}$$

On trouve -6, il faut donc bien penser au bit de retenue.

Il faut une représentation commune des nombres : ici on travaille sur 4 bits (nombre de -8 à 7). Il faut donc anticiper un résultat sur 5 bits.

Réaliser le calcul de 7 + 3 = 10 en binaire en retrouvant cette fois le bon résultat.

> **Le nombre maximum de bits du résultat est donc : nr = max(na , nb) + 1**
> **On utilisera toujours le cas général en anticipant le bit de retenu dans le calcul, comme c'est le cas dans le processeur.**

4.2 Addition en virgule fixe

Le nombre de bits des opérandes a et b doit être identique (na = nb). Il faut aussi choisir un format Q_k commun pour le calcul : on positionne la virgule de façon à ce qu'elle soit alignée :

- Aligner la virgule
- Réaliser l'extension de signe à gauche pour la partie entière
- Etendre le nombre de bit à droite pour la partie décimale

Pour les débordements, on se retrouve dans la même configuration que pour les nombres entiers signé.

4.2.1 Deux opérandes de signes opposés

Si sign(a) != sign(b) alors il n'y a **pas de débordement possible**.

4.2.2 Deux opérandes de même signe

Si sign(a) = sign(b) alors il y a un débordement possible.

- **Le nombre maximum de bits du résultat est donc : nr = max(na,nb) + 1**
- **Si a est en Q_k et b en $Q_{k'}$ alors le résultat est en $Q_{max(k\,;\,k')}$**

Exemple : Soit l'addition de **a** au format $Q_{1.4}$ et de **b** au format $Q_{2.2}$. Le format Qk commun est donc $Q_{2.4}$. On prend en compte la retenue possible du résultat, donc le résultat sera codé en $Q_{3.4}$. Pour la réalisation des calculs :

- On étend l'opérande a de 2 bits vers la gauche (extension de signe)
- On étend l'opérande b de 1 bit vers la gauche (extension de signe) et 2 bits vers la droite.

$$\begin{array}{cccccccc} \boldsymbol{a_0} & \boldsymbol{a_0} & a_0 & . & a_{-1} & a_{-2} & a_{-3} & a_{-4} \\ \boldsymbol{b_1} & b_1 & b_0 & . & b_{-1} & b_{-2} & \boldsymbol{0} & \boldsymbol{0} \\ \hline r_2 & r_1 & r_0 & . & r_{-1} & r_{-2} & r_{-3} & r_{-4} \end{array}$$

Figure 25 : Exemple d'une addition de deux nombres codés en virgule fixe

Réaliser le calcul de 1,25 + (-8) = -5,75 (Pas de débordement)

Réaliser le calcul de (-7,25) + (-7.25) = -14,50 (Débordement)

4.3 Multiplication en entiers signés

- **Le nombre maximum de bit du résultat est nr = na + nb**

- En vert (gras) on a les bits de l'extension de signe des calculs intermédiaires
- En rouge (gras), on a les retenues

Exemple : -4 x 6 = -24 (les opérandes -4 et 6 sont codés sur 4 bits

$$\begin{array}{cccccccc} \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{1} & 1 & 1 & 0 & 0 \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & 0 & 1 & 1 & 0 \\ \hline \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{1} & & & & & \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & . \\ 1 & 1 & 1 & 1 & 0 & 0 & . & . \\ \hline 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

On a bien -24 codé sur 8 bits.

Exemple, réaliser le calcul de 7 x -8 = -56 en binaire (7 et 8 seront codés sur 4 bits)

Exemple, réaliser le calcul de -8 x -8 = 64 en binaire (-8 et -8 seront codés sur 4 bits)

4.4 Multiplication en virgule fixe

La valeur du nombre de bit du résultat est la même que pour les multiplications des nombres signés. En effet nr = na + nb. Mais ce qui nous intéresse c'est l'emplacement de la virgule après le calcul.

4.4.1 Multiplication sans perte de précision

On considère dans ce cas que tous les chiffres après la virgules seront conservés. Il n'y aura donc pas de recadrage du résultat.

Le format Q_k peut être différent. La méthode consiste juste à réaliser l'extension de signe à gauche pour la partie entière. Le reste du calcul se fait de façon classique pour une multiplication.

Exemple : Soit la multiplication de **a** au format $Q_{1.4}$ et de **b** au format $Q_{2.2}$. Le résultat sera codé en $Q_{3.6}$. Pour la réalisation des calculs :

- On étend l'opérande a de 4 bits vers la gauche (extension de signe)
- On étend l'opérande b de 5 bits vers la gauche (extension de signe)

$$\begin{array}{cccccccccc} a_0 & a_0 & a_0 & a_0 & a_0 \bullet & a_{-1} & a_{-2} & a_{-3} & a_{-4} \\ b_1 & b_1 & b_1 & b_1 & b_1 & b_1 & b_0 \bullet & b_{-1} & b_{-2} \\ \hline r_3 & r_2 & r_1 \bullet & r_{-1} & r_{-2} & r_{-3} & r_{-4} & r_{-5} & r_{-6} \end{array}$$

Réaliser le calcul de :

- a : -3,5 en Q1 sur 4 bits
- b : 1,25 en Q2 sur 4 bits

Le nombre de bits après la virgule du résultat est égale à $k_r = k_a + k_b$

4.4.2 Multiplication avec perte de précision : cas général

Dans le calcul d'un filtre numérique, le format de donnée doit être recadré dans le type d'origine. Certes, ce recadrage se produit le plus tard possible, mais il doit de toute façon intervenir.

Si on travaille sur deux opérandes (coefficients et échantillons sur 16 bits), le résultat du calcul est sur 32 bits, comme nous l'avons déjà prouvé [nr = na + nb = 32 bits]. A la fin du calcul, le résultat doit être recadré sur 16 bits pour pouvoir être à nouveau utilisé. Le problème est de savoir quels bits nous devons conserver. Nous avons alors un compromis à faire :

- Plus nous gardons des bits à droite, plus nous gardons de la précision dans le résultat, mais plus nous avons de chance de perdre des bits significatifs.
- Plus nous gardons des bits à gauche, plus nous gardons des bits significatifs, mais plus nous réalisons un arrondi du résultat.

4.4.3 Multiplication avec perte de précision, cas du traitement du signal

En traitement numérique du signal, les deux opérandes sont :

- Coefficients au format Q_k
- Echantillons au format $Q_{k'}$

Le résultat est donc au format $Q_{k+k'}$. Lors de l'arrondi du nombre, on cherche toujours à revenir sur le format des échantillons, soit le format $Q_{k'}$. D'une part parce que le résultat lui-même servira d'échantillon dans les calculs suivants (cas d'un filtre IIR), d'autre part parce que le format des échantillons qui a été récupéré par le CAN, doit être le même que celui qui sera fourni au CNA.

Reprenons une étude de cas complète autour d'un exemple :

- Soit un ADC travaillant sur 16 bits [-32768 ; 32767]. Codage en Q_0
- Soit des coefficients sur 16 bits en Q_{15}

La multiplication des deux nombres donne une représentation en Q_{15} sur 32 bits.

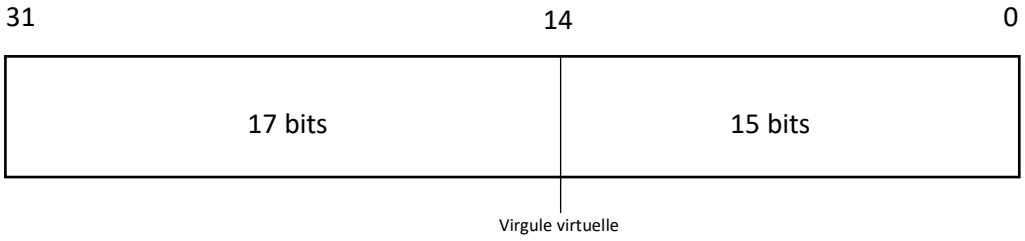


Figure 26 : Résultat d'une opération de deux opérandes Q16.0 x Q16.15 en Q32.15

1er cas : Nous gardons les 16 bits de poids faible de 15 à 0 en représentation Q_{15}. Le résultat gardera toute sa précision sans aucun arrondi, en revanche il sera complètement erroné dès lors que le résultat sera en dehors de la plage [-1; 1[.

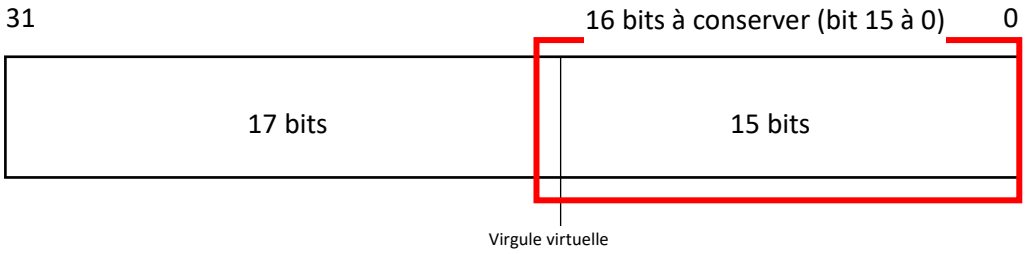


Figure 27 : Conservation des 16 bits de poids faibles

2ème cas : Nous gardons les 16 bits de poids fort de 31 à 16. Dans ce cas, nous faisons un très gros arrondi du résultat car nous supprimons toutes les valeurs situées après la virgule ainsi que le premier bit à gauche de la virgule. Il y a 2 inconvénients supplémentaires à cela :

- Le résultat stocké est divisé par deux (dans notre exemple) par rapport à la valeur réelle, donc il faudra à un moment donné le multiplier par 2 (dans notre exemple) avec à nouveau un risque de débordement.
- Le résultat n'est plus dans un format directement réutilisable car nous avons travaillé avec des nombres en Q0 (échantillons) et Q15 (coefficients). Nous ne pouvons donc pas l'accumuler directement comme nous souhaiterions le faire dans le cas des filtres numériques.

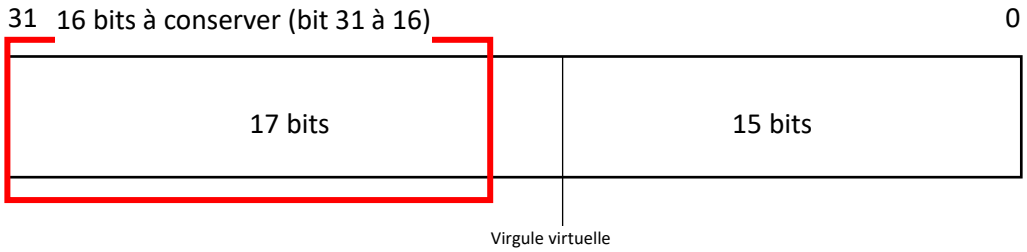


Figure 28 : Conservation des 16 bits de poids forts

3ème cas : Dans ce cas, nous allons garder les bits nous permettant d'avoir un résultat en Q_0 sur 16 bits car il sera à nouveau multiplié par un Q15, etc… Il y a donc un bit de poids fort qui n'est pas pris en compte. Il y a un risque (faible) d'un débordement puisque le bit significatif 31 n'est pas pris en compte. Nous verrons plus tard comment il sera possible de s'affranchir de ce problème.

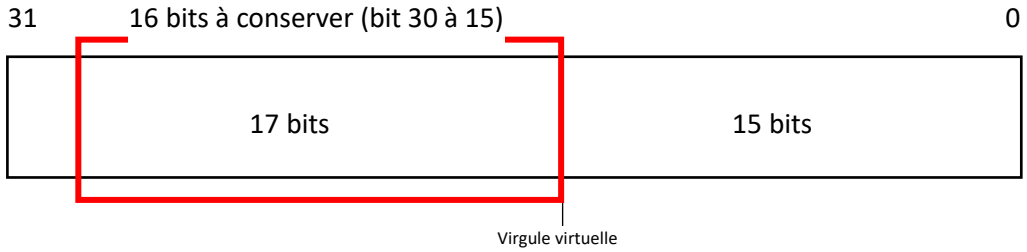


Figure 29 : Solution retenue

Exemple :

- Soit un ADC travaillant sur 8 bits. Codage en Q_7.
- Soit des coefficients sur 8 bits Codage en Q_7.

➡ Donner le format du résultat et l'opération pour y parvenir.

4.4.4 Cas particulier de la multiplication en Q7, Q15 et Q31.

Par exemple, le codage Q_{15} sur 16 bits est très courant en traitement du signal. Il ne s'agit que d'un cas particulier de ce que nous avons vu précédemment. Les mêmes propriétés existent pour le format en Q_7 sur 8 bits et Q_{31} sur 32 bits

La multiplication de deux opérandes en Q_{15} sur 16 bits donne un résultat en Q_{30} sur 32 bits. Afin de remettre le résultat en Q_{15}, on ne conserve que les bits de 31 à 15. Nous réalisons donc un décalage de 15 à droite du résultat.

Comme nous l'avons vu précédemment, nous laissons de côté un bit de poids fort de la représentation. Si celui-ci est significatif, le résultat sera alors erroné. Nous allons voir dans quel cas ce bit est significatif.

En Q_{15}, le résultat est un nombre résultant de la multiplication de deux opérandes comprises entre [-1 ; 1 [. Et le problème est que le résultat de cette opération est un nombre compris entre [-1 ; 1] (avec le 1 compris cette fois !!!). En effet, toutes les opérandes entre [-1 ; 1[peuvent être codées avec un seul bit pour la partie entière, mais lorsque le résultat vaut 1, il doit nécessairement être codé avec 2 bits pour la partie entière.

Codage binaire en Q_{15} sur 16 bits :

Nombre en décimal	Nombre en binaire
-1	1000 0000 0000 0000
0	0000 0000 0000 0000
0,999969482421875	0111 1111 1111 1111

Figure 30 : Codage des nombres en $Q_{16.15}$.

Nous devrions donc en théorie rajouter un bit dans notre représentation juste pour coder une valeur qui est le résultat d'une seule opération possible **1 = -1 x -1**. Ceci est très dommageable pour la réalisation des calculs dans un DSP. La solution retenu est souvent de conserver la représentation en Q_{15} et de détecter le débordement. Si le résultat est 1, il sera alors saturé à la valeur la plus proche représentable en Q_{15}, soit 0,999969482421875.

4.5 Etude d'un cas concret : Filtre FIR

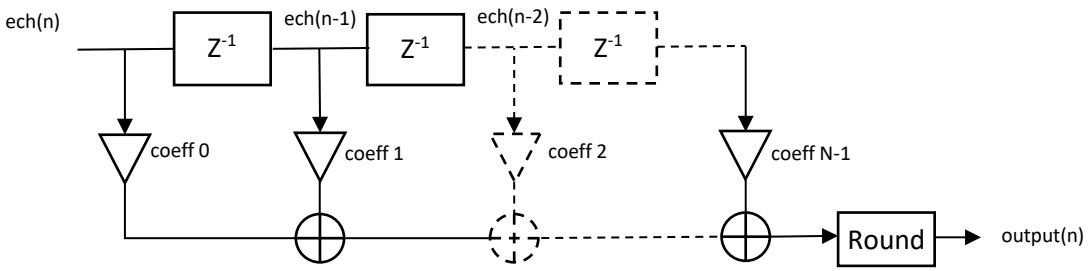


Figure 31 : Représentation graphique d'un filtre FIR

Dans un filtre numérique, nous avons une série de multiplication (coefficients x échantillons), puis une série d'accumulations successives. Les coefficients sont codés en Q_{15} sur 16 bits et les échantillons sont codés en Q_0.

Le filtre que nous étudierons aura le gabarit suivant :

```
% sampling frequency: 2000 Hz
% 0 Hz - 400 Hz /   gain = 1
% 800 Hz - 1000 Hz /  gain = 0
nbrCoeff = 7;
coeff = [ -2097 -1622 9736 18369 9736 -1622 -2097 ];
```

- Donnez les valeurs réelles des coefficients de ce filtre numérique

En entrée du filtre on considérera un signal sinusoïdal de 150 Hz d'amplitude 19660 par rapport à la pleine échelle (16 bits / 32768). Donner les 7 premières valeurs du signal d'entrée.

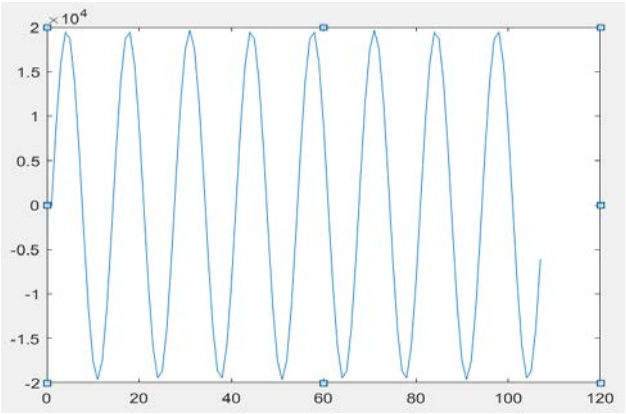
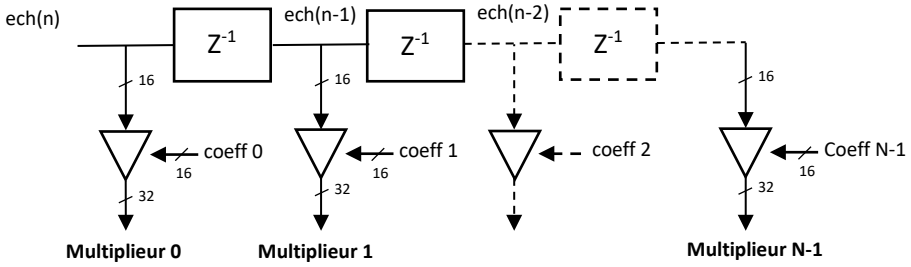


Figure 32 : Signal d'entrée du filtre numérique

Nous allons essayer d'étudier les différentes valeurs en différents points du filtre numérique en commençant par la sortie des multiplieurs.

4.5.1 Etude de la sortie des multiplieurs

La multiplication de deux nombres sur 16 bits donne une représentation en 32 bits.



Sorties multiplieurs

Figure 33 : Etude de la sortie des multiplieurs du filtre

A chaque échantillon, nous notons le résultat de chaque multiplieur dans un tableau. Le tableau ci-dessous représente l'évolution pour une durée d'un peu plus d'une demie période de la sinusoïde d'entrée.

outputTaps =

Multiplieur 0	Multiplieur 1	Multiplieur 2	Multiplieur 3	Multiplieur 4	Multiplieur 5	Multiplieur 6
0	0	0	0	0	0	0
-18717822	0	0	0	0	0	0
-33354882	-14477972	0	0	0	0	0
-40721643	-25799532	86903536	0	0	0	0
-39211803	-31497618	154860816	163961694	0	0	0
-29152494	-30329778	189063384	292177314	86903536	0	0
-12741372	-22549044	182053464	356707611	154860816	-14477972	0
6450372	-9855272	135349872	343481931	189063384	-25799532	-18717822
24232932	4989272	59155936	255365838	182053464	-31497618	-33354882
36735246	18743832	-29947936	111610044	135349872	-30329778	-40721643
41229117	28414196	-112509216	-56503044	59155936	-22549044	-39211803
36735246	31890142	-170555248	-212272164	-29947936	-9855272	-29152494
24232932	28414196	-191419496	-321788142	-112509216	4989272	-12741372
6450372	18743832	-170555248	-361152909	-170555248	18743832	6450372
-12741372	4989272	-112509216	-321788142	-191419496	28414196	24232932
-29152494	-9855272	-29947936	-212272164	-170555248	31890142	36735246
-39211803	-22549044	59155936	-56503044	-112509216	28414196	41229117
-40721643	-30329778	135349872	111610044	-29947936	18743832	36735246
-33354882	-31497618	182053464	255365838	59155936	4989272	24232932
-18717822	-25799532	189063384	343481931	135349872	-9855272	6450372
0	-14477972	154860816	356707611	182053464	-22549044	-12741372
18717822	0	86903536	292177314	189063384	-30329778	-29152494
33354882	14477972	0	163961694	154860816	-31497618	-39211803
40721643	25799532	-86903536	0	86903536	-25799532	-40721643
39211803	31497618	-154860816	-163961694	0	-14477972	-33354882
29152494	30329778	-189063384	-292177314	-86903536	0	-18717822
12741372	22549044	-182053464	-356707611	-154860816	14477972	0

Figure 34 : Valeurs des sorties des multiplieurs du filtre numérique

Nous pouvons remarquer que toutes les valeurs sont bien comprises sur 32 bits $[-2^{31};2^{31}-1]$. Il n'y a eu aucun débordement.

4.5.2 Etude de la sortie des additionneurs

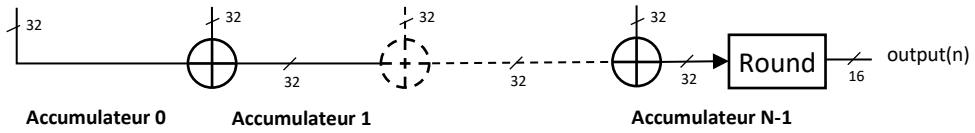


Figure 35 : Etude de la sortie des accumulateurs

Chaque sortie du multiplieur est accumulée jusqu'à produire la valeur de la sortie. L'addition de deux opérandes de 32 bits nous donne un résultat sur potentiellement 33 bits. Cela nécessiterait donc de manipuler des variables plus grande (64 bits par exemple) pour les additions. Dans la pratique, cela n'est pas toujours fait :

- Car les coefficients du filtre sont faibles et donc le résultat de la multiplication est souvent faible.
- Car les résultats de la multiplication étant tantôt **positif** tantôt **négatif**, les accumulations "s'annulent" souvent.

A chaque échantillon, nous allons noter les valeurs de chaque accumulation. La dernière valeur du tableau correspond donc à la sortie du filtre. Le tableau ci-dessous rassemble les valeurs pour une durée d'un peu plus d'une demie période de sinusoïde.

outputFilter =

Accumulateur 0	Accumulateur 1	Accumulateur 2	Accumulateur 3	Accumulateur 4	Accumulateur 5	Accumulateur 6
0	0	0	0	0	0	0
-18717822	-18717822	-18717822	-18717822	-18717822	-18717822	-18717822
-33354882	-47832854	-47832854	-47832854	-47832854	-47832854	-47832854
-40721643	-66521175	20382361	20382361	20382361	20382361	20382361
-39211803	-70709421	84151395	248113089	248113089	248113089	248113089
-29152494	-59482272	129581112	421758426	508661962	508661962	508661962
-12741372	-35290416	146763048	503470659	658331475	643853503	643853503
6450372	-3404900	131944972	475426903	664490287	638690755	619972933
24232932	29222204	88378140	343743978	525797442	494299824	460944942
36735246	55479078	25531142	137141186	272491058	242161280	201439637
41229117	69643313	-42865903	-99368947	-40213011	-62762055	-101973858
36735246	68625388	-101929860	-314202024	-344149960	-354005232	-383157726
24232932	52647128	-138772368	-460560510	-573069726	-568080454	-580821826
6450372	25194204	-145361044	-506513953	-677069201	-658325369	-651874997
-12741372	-7752100	-120261316	-442049458	-633468954	-605054758	-580821826
-29152494	-39007766	-68955702	-281227866	-451783114	-419892972	-383157726
-39211803	-61760847	-2604911	-59107955	-171617171	-143202975	-101973858
-40721643	-71051421	64298451	175908495	145960559	164704391	201439637
-33354882	-64852500	117200964	372566802	431722738	436712010	460944942
-18717822	-44517354	144546030	488027961	623377833	613522561	619972933
0	-14477972	140382844	497090455	679143919	656594875	643853503
18717822	18717822	105621358	397798672	586862056	556532278	527379784
33354882	47832854	47832854	211794548	366655364	335157746	295945943
40721643	66521175	-20382361	-20382361	66521175	40721643	0
39211803	70709421	-84151395	-248113089	-248113089	-262591061	-295945943
29152494	59482272	-129581112	-421758426	-508661962	-508661962	-527379784
12741372	35290416	-146763048	-503470659	-658331475	-643853503	-643853503

Figure 36 : Valeur des résultats d'accumulation successives du filtre

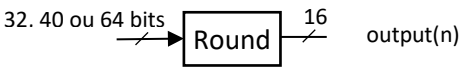
- Donner la valeur maximal atteinte par le filtre et sa marge par rapport au débordement.

Cette méthode empirique n'est bien sûr pas valable pour s'assurer qu'aucun débordement n'aura lieu.

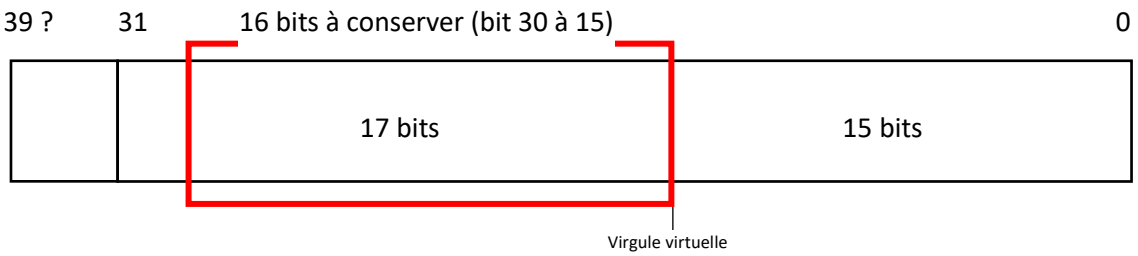
Comme nous l'avons déjà précisé au chapitre 3.2.2, dans les DSP, les accumulations successives se font sur des nombres de bits supérieurs à 32 bits. Par exemple dans le DSP virgule fixe TMS320C54, l'accumulateur est de 40 bits. Les 8 bits supplémentaires sont appelés "bits de garde".

4.5.3 Arrondi du résultat

Quel que soit le format du résultat des accumulations (32, 64, 40 bits…) la dernière étape consiste toujours à stocker le résultat dans le format d'origine (ici 16 bits). Cet arrondi amènera nécessairement à une erreur supplémentaire sur le résultat.



Dans notre cas, nous récupérerons toujours les bits 30 à 15 du résultat :



On remarque donc que les bits de garde sont intéressants seulement si le résultat a eu un débordement temporaire et qu'il revient dans des plages correctes avant le processus d'arrondi. Car dans tous les cas, si le résultat ne peut pas être représenté sur 16 bits (dans notre exemple), l'arrondi donnera lieu à un résultat erroné.

4.5.4 Arrondi Versus Troncature

Lors du codage en virgule fixe, il est toujours intéressant d'arrondir le nombre plutôt que de le tronquer.

Exemple : Codage de 2,95 en Q1

On code donc la valeur entière de 2,95 x 2 =5,9

- Si on tronque > 5 > 0101 soit la valeur 2,5 en Q1
- Si on arrondi > 6 > 0110 soit la valeur 3 en Q1

Il est donc intéressant **d'arrondir** le nombre **avant** codage :

- On ajoute la moitié du digit de poids faible (soit 0,5 pour un nombre entier en décimal)
- Donc on code la valeur (2,95 x 2) + 0,5 =6,4 tronquée soit 6, soit la valeur 3 en Q1

4.5.5 Arrondi d'un nombre binaire

Dans le paragraphe précédent, nous avons compris qu'il était nécessaire de ne conserver que les bits 30 à 15 pour le résultat (cas des coefficients codés en Q_{15}). Cela implique de réaliser un décalage du résultat de 15 bits vers la droite. En réalité, lorsque nous réalisons ce type d'opération, nous ne réalisons pas un **arrondi** mais une **troncature**, alors que nous venons de démontrer que c'est justement l'arrondi qui est le plus intéressant.

Imaginons un résultat sur 32 bits en Q_{15} : 0x 0000 0000 0101 0101 0.**111 1111 1111 1111**

- Le résultat par **troncature** serait de : 0x0000 0000 1010 1010
- Le résultat par **arrondi** serait de : 0x0000 0000 1010 1011, ce qui est plus proche de la vraie valeur.

En filtrage numérique, nous utilisons souvent la multiplication de 2 nombres en Q_{15}, le résultat est en Q_{30}. Pour récupérer un résultat en Q_{15}, on doit :

- Soit réaliser un décalage de 15 bits à droite et récupérer les 16 bits de poids faibles.
- Soit réaliser un décalage de 1 à gauche et récupérer les 16 bits de poids forts.

Prendre 16 bits sur les 32 bits du registre correspond à faire une troncature. Comment pourrait-on faire pour réaliser un arrondi ?

Exemple sur 4 bits en Q3 : 0,75 x 0,625 = 0,46875

Le résultat sur 8 bit en binaire est : 00.01 1110,

- si on fait une troncature, on a 0.011 soit 0,375 (erreur de x %)
- si on fait un arrondi, on a 0.100 soit 0,5 (erreur de x %)

L'arrondie se fait encore en additionnant la moitié du bit de poids faible et en tronquant. Donc on ajoute ici 1<<(k-1)

5 Programmation d'un filtre numérique

5.1 Traitement en temps réel / traitement par bloc

5.1.1 Traitement en temps réel

Le traitement en temps réel nécessite qu'une exécution de l'algorithme soit lancée à chaque période d'échantillonnage. L'avantage de ce type de traitement est que le délai de génération de l'échantillon de sortie est faible (1 période d'échantillonnage). Si le délai est une forte contrainte alors cette solution est la bonne. Néanmoins, comme nous le verrons plus loin cela a une forte tendance à augmenter la charge de calcul du processeur pour les DSC. Les vrais DSP ne sont pas (ou peu) affectés.

5.1.2 Traitement par bloc

Comme nous l'avons vu au chapitre 1.3.3, le réalisation de buffer circulaire augmente la complexité de la boucle de calcul du filtre numérique. En effet, nous avons à chaque itération un test vérifiant le dépassement du pointeur sur le tableau des échantillons. L'intérêt du buffer circulaire est donc très réduit, voire nul dans ce cas. Nous allons voir comment le traitement par bloc permet d'améliorer ce phénomène.

5.2 Optimisation du calcul

5.2.1 Amélioration du vieillissement des échantillons

Nous prendrons comme exemple un filtre FIR de 6 coefficients (N = 6).

@ du tableau	Echantillons		Coefficients
0	ech_0		$coeff_0$
1	ech_1		$coeff_1$
2	ech_2	X	$coeff_2$
3	ech_3		$coeff_3$
4	ech_4		$coeff_4$
5	ech_5		$coeff_5$

Figure 37 : Calcul d'un filtre numérique avec 6 coefficients

Le calcul du filtre numérique dans ce cas est très simple comme nous l'avons vu au chapitre 1.3.1. Afin de garder l'avantage de cette simplicité, nous allons modifier la façon dont nous réalisons le vieillissement des échantillons. Dans cette méthode l'espace réservé aux échantillons est bien plus grand que le tableau précédemment réservé comme cela est représenté à la Figure 38.

@ du tableau	Echantillons
0	
1	
2	
3	
4	
5	
6	
7	ech_0
8	ech_1
9	ech_2
10	ech_3
11	ech_4
12	ech_5

X

Coefficients
$coeff_0$
$coeff_1$
$coeff_2$
$coeff_3$
$coeff_4$
$coeff_5$

Figure 38 : Utilisation d'un traitement par bloc

A partir de là, chaque nouvel échantillon se positionnera à l'adresse 6, puis 7, puis 8, etc… du tableau d'échantillons. Le positionnement des nouveaux échantillons est représenté Figure 39. Le nombre total d'échantillons mis en fin de tableau est appelé un bloc.

@ du tableau	Echantillons
0	
1	
2	
3	
4	newEch2
5	newEch1
6	newEch0
7	ech_0
8	ech_1
9	ech_2
10	~~ech_3~~
11	~~ech_4~~
12	~~ech_5~~

X

Coefficients
$coeff_0$
$coeff_1$
$coeff_2$
$coeff_3$
$coeff_4$
$coeff_5$

Figure 39 : Positionnement des nouveaux échantillons aux adresse 6, 5, 4…

Lorsque la fin du bloc est arrivée, nous réalisons le décalage comme nous le faisions dans le cas du buffer linéaire afin de remettre les échantillons au début du tableau. Ce décalage est couteux en temps mais il est réalisé qu'une seule fois toutes les X itérations (X étant la taille du bloc). Le résultat est décrit à la Figure 40.

@ du tableau	Echantillons
0	
1	
2	
3	
4	
5	
6	
7	newEch6
8	newEch5
9	newEch4
10	newEch3
11	newEch2
12	newEch1

X

Coefficients
$coeff_0$
$coeff_1$
$coeff_2$
$coeff_3$
$coeff_4$
$coeff_5$

Figure 40 : Réinitialisation du buffer d'échantillon

Cette méthode est donc avantageuse sous certains aspects. L'inconvénient étant qu'elle nécessite plus d'espace mémoire pour s'exécuter.

Le résumé des différents mode de vieillissement est donné sur la Figure 41.

	Vieillissement des échantillons	**Calcul du filtre**	**Occupation mémoire**
Buffer Linéaire	*Complexe*	*Simple*	*Taille du filtre*
Buffer circulaire	*Simple*	*Complexe*	*Taille du filtre*
Buffer par bloc	*Simple sauf 1 fois tous les X échantillons. X étant la taille du bloc*	*Simple*	*Taille du filtre + Taille du bloc*

Figure 41 : Résumé des méthodes de vieillissement des échantillons

5.2.2 Loop unrooling

La traduction signifierait "développement des boucles". On cherche toujours à éviter le test des fins de boucles réalisé à chaque itération. Si nous reprenons le cas de la Figure 37. Le code est le suivant :

```
for ( i = 0 ; i < N ; i++) {           // 6 itérations
       output += coeff[ i ] * ech[ i ];
}
```

Dans ce code, le nombre i est testé 6 fois. A chaque fois que la condition de fin n'est pas respectée, on a un saut (rupture du pipeline) et une incrémentation de i.

Ceci pourrait être transformé en :

```
for ( i = 0 ; i < N/3 ; i++) {         // 2 itérations
       output += coeff[ i ] * ech[ i ];
       output += coeff[ i + 1] * ech[ i + 1 ];
       output += coeff[ i + 2] * ech[ i + 2 ];
}
```

Dans ce code, le nombre i est testé 2 fois. La condition de fin n'est pas respectée une fois : on a un seul saut (rupture du pipeline) et une incrémentation de i. Néanmoins, il y a une incrémentation supplémentaire de i pour chaque ligne de la boucle.

5.3 Utilisation de CMSIS DSP

CMSIS est une HAL (Hardware Abstraction Layer) produit par ARM. Il s'agit d'une librairie logicielle disponible sur tous les produits ARM indépendamment du constructeur. Parmi les librairies CMSIS disponible, il en existe une spécifique pour le traitement du signal : CMSIS-DSP.

```
void arm_fir_q15 ( const arm_fir_instance_q15 *  S,
                   const q15_t *                 pSrc,
                   q15_t *                       pDst,
                   uint32_t                      blockSize
                 )
```

Parameters

[in]	**S**	points to an instance of the Q15 FIR filter structure
[in]	**pSrc**	points to the block of input data
[out]	**pDst**	points to the block of output data
[in]	**blockSize**	number of samples to process

Returns

none

Scaling and Overflow Behavior

The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Remarks

Refer to **arm_fir_fast_q15()** for a faster but less precise implementation of this function.

	Cortex-M3	Cortex-M4	Cortex-M7	Traditional DSP
Single cycle MAC		Fixed-point only	Fixed and floating-point	Y
Floating-point		Y	Y	Y
Fractional and saturating math		Y	Y	Y
SIMD operations		Y	Y	Y
Load and store in parallel with math			Y	Y
Zero overhead loops			Y	Y
Accumulator with guard bits				Y
Circular and bit-reversed addressing				Y

6 Solutions des exercises

1.2.1.Fonction de transfert et équation récurrente

$y(n) = 0.21.x(n) + 0.47.x(n-1) + 0.47.x(n-2) + 0.21.x(n-3)$

$y(n) = 0.98.x(n) + 0.29.x(n-1) + 0.29.x(n-2) + 0.98.x(n-3) + 0.57y(n-1) - 0.42y(n-2) - 0.05y(n-3)$

1.3.1.Buffer linéaire

```
float coeff[N] = {  ,  ,  ... ,  ,  };
uint16_t ech[N];
uint16_t newEch;
uint16_t output = 0;


void main(void){
    while(1){
        // Reception d'un echantillon depuis le CAN
        HAL_SAI_Receive(&newEch);

        // Stockage dans le tableau d'échantillon
        // Réinitialiser output a 0.
        ech[0] = newEch;
        output = 0;

        // Calcul du filtre numérique (cas du Buffer Linéaire)
        for (uint32_t i = 0 ; i < N ; i++){
            output = output + ech[i] * coeff[i];
        }

        //Envoi du résultat sur le CNA
        HAL_SAI_Transmit(&output);

        // Vieillissement en Buffer linéaire
        for (uint32_t i = N-1 ; i > 0; i--){
            ech[i] = ech[i-1];
        }
    }
}
```

1.3.2.Buffer circulaire

```
float coeff[N] = {  ,  ,  ... ,  ,  };
uint16_t ech[N];
uint16_t newEch;
uint16_t output = 0;
uint16_t index = N-1;

void main(void){
    while(1){
        // Reception d'un echantillon depuis le CAN
        HAL_SAI_Receive(&newEch);
```

```
            // Stockage dans le tableau d'échantillon
            // Réinitialiser output a 0.
            ech[index] = newEch;
            output = 0;

            // Calcul du filtre numérique (cas du Buffer Circulaire)
            for (uint32_t i = index ; i < (index + N) ; i++){
                output = output + ech[i%N] * coeff[i-index];
            }

            // Vieillissement du Buffer circulaire : MAJ de l'index
            if(index == 0
                index = N-1;
            else
                index--;

            //Envoi du résultat sur le CNA
            HAL_SAI_Transmit(&output);
        }
    }
```

2.1.1.Le codage des entiers non signés

$$0 \leq Nombre < 2^N$$

2.1.2.Le codage des entiers signés

$$-2^{N-1} \leq Nombre < 2^{N-1}$$

Nombre	Codage
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100

2.2.1.Le codage en virgule fixe

$$(-2^{N-1}).2^{-k} \leq Nombre \leq (2^{N-1}-1).2^{-k}$$

- Q_2 : 23.25
- Q_4 : 5.8125
- Q_7 : 0.7265625

Q5 sur 8 bits : $-4 \leq Nombre \leq 3.96875$ avec une précision de +/- 0.015625

- -1 ≤ Nombre <1 Q_7 avec une erreur de +/- 2^{-8}
- -6 ≤ Nombre ≤ -4 Q_4 avec une erreur de +/- 2^{-5}
- 200 ≤ Nombre ≤ 200 Q_{-1} avec une erreur de +/- 2^0

- -0.1 ≤ Nombre ≤ 0 Q_{10} avec une erreur de +/- 2^{-11}

Codage de nombre en Q0 sur 8 bits :

- 98,895 en Q_0 : 1100 0011, 99 codé
- 0,01298 en Q_0 : 0000 0000, 0 codé

2.2.2.Le codage en virgule flottante

- 98.895 = 9.89×10^1
- 10.1 = 1.01×10^1
- 0.01298 = 1.29×10^2
- -128000 = -1.28×10^5

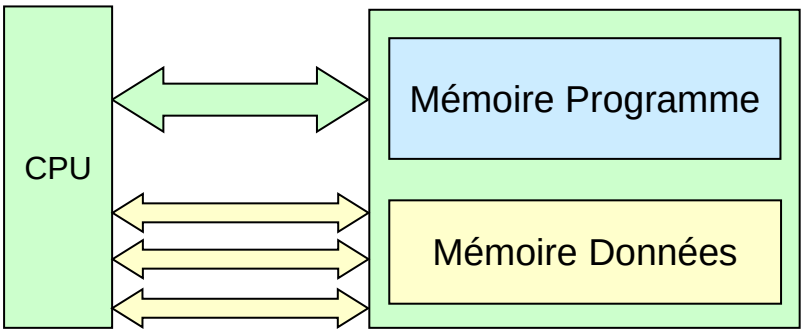
- 98.895 : Exposant = 6 soit 0110, mantisse = 1.5452 soit 0110 en Q_2, 96 codé
- 0.01298 : Exposant = -7 soit 1001, mantisse = 1.6614 soit 0111 en Q_2, 0.01367 codé

2.3.1.Cas de la représentation en virgule fixe

2.4.Relation entre le codage des nombres et le type de variable

Nom du type	Signification	Codage	Plage de valeur
char	Entier signé	8 bits	-128 à 127
unsigned int	Entier non signé	32 bits	0 à 2^{32}-1
int	Entier signé	32 bits	-2^{31} à 2^{31}-1
float	Réel signé	32 bits 24 bits de mantisse 8 bits d'exposant	$-3,4 \times 10^{38}$ à $3,4 \times 10^{38}$
double	Réel signé	64 bits 53 bits de mantisse 11 bits d'exposant	$-1,7 \times 10^{308}$ à $1,7 \times 10^{308}$

3.5.Les bus d'accès aux mémoires



4.1.1. Addition en entier non signés

1	**1**			
	1	1	0	0
	0	1	0	1
1	0	0	0	1

Nous avons donc un bit de retenue

4.1.2. Addition de 2 entiers signés de signes opposés

4.1.3. Addition de 2 entiers signés de même signe

Tout d'abord un calcul simple sans débordement : 4 + 3 = 7

0	1	0	0
0	0	1	1
0	1	1	1

Calcul de 7 + 3 = 10

	1	**1**	**1**	
0	0	1	1	1
0	0	0	1	1
0	1	0	1	0

4.2.1. Deux opérandes de signes opposés

4.2.2.Deux opérandes de même signe

Calcul de 1,25 + (-8) = -5,75 (Pas de débordement)

0	0	0	0	1	.	0	1
1	1	0	0	0	.	0	0
1	1	0	0	1	.	0	1

Calcul de (-7,25) + (-7.25) = -14,50 (Débordement)

1				1		1	
1	1	0	0	0	.	1	1
1	1	0	0	0	.	1	1
1	0	0	0	1	.	1	0

Le résultat est donc bien -14,5 sur 7 bits (6 + 1 bit)

4.3. Multiplication en entiers signés

Réaliser le calcul de 7 x -8 = -56 en binaire

0	0	0	0	0	1	1	1
1	1	1	1	1	0	0	0
2	2	1					
0	0	1	1	1	.	.	.
0	1	1	1	.	.	.	.
1	1	1	.	.	.	.	.
1	1	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	1	0	0	1	0	0	0

On a bien -56 codé sur 8 bits

Réaliser le calcul de -8 x -8 = 64

1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	0	0	0	.	.	.
1	0	0	0	.	.	.	.
0	1	0	0	0	0	0	0

4.4.1.Multiplication sans perte de précision

1	1	1	1	1	0	0	1
0	0	0	0	0	1	0	1
1	1						
1	1	1	1	1	0	0	1
1	1	1	0	0	1	.	.
1	1	0	1	1	1	0	1

On a donc bien -4,375 représenté en Q3 sur 8 bits.

4.5. Etude d'un cas concret : Filtre FIR

```
      // Valeurs entières

coeff = [   -2097
            -1622
            9736
            18369
            9736
            -1622
            -2097
 ];
```

```
// Valeurs réelles

coeff = [   0.06402587891
            -0.0494995172
            0.297119406
            0.5605773926
            0.297119406
            -0.0494995172
            0.06402587891
        ];
```

Les 7 premières valeurs du signal d'entrée sont :

0 / 8926 / 15906 / 19419 / 18699 / 13902 / 6076 / -3076