DEROUARD Guillaume
RODRIGUEZ Antony

RAPPORT DSP

ISEN 5 2006-2007

INTRODUCTION

Les DSP (Digital Signal Processor) sont principalement utilisés en vu du traitement des signaux sonores. Ces séances de DSP nous ont ainsi permis de mettre en application la théorie reçue au cours de notre cursus ISEN lors des cours de traitement de signal.

Le principal avantage des DSP par rapport à un processeur ‘classique’ est qu’il est capable d’effectuer plusieurs actions en un même cycle d’horloge. C’est cette caractéristique qui permettra au processeur d’être plus performant pour la conception de filtre.

Ce rapport sera organisé de façon chronologique. Nous présenterons pour chaque TP, les objectifs de la séance. Nous détaillerons ensuite les différentes notions apprises et leurs applications dans les exercices. Chaque séance contiendra le code source annoté des exercices étudiés.

SEANCE 1

Objectifs :

Lors de cette séance, un premier **objectif** est de **prendre en main la carte DSP et son logiciel de développement.**

Nous étudierons ensuite les différents **modes d'adressages**, les **instructions parallèles** et l'instruction **repeat**.

- **Installation et test de configuration**

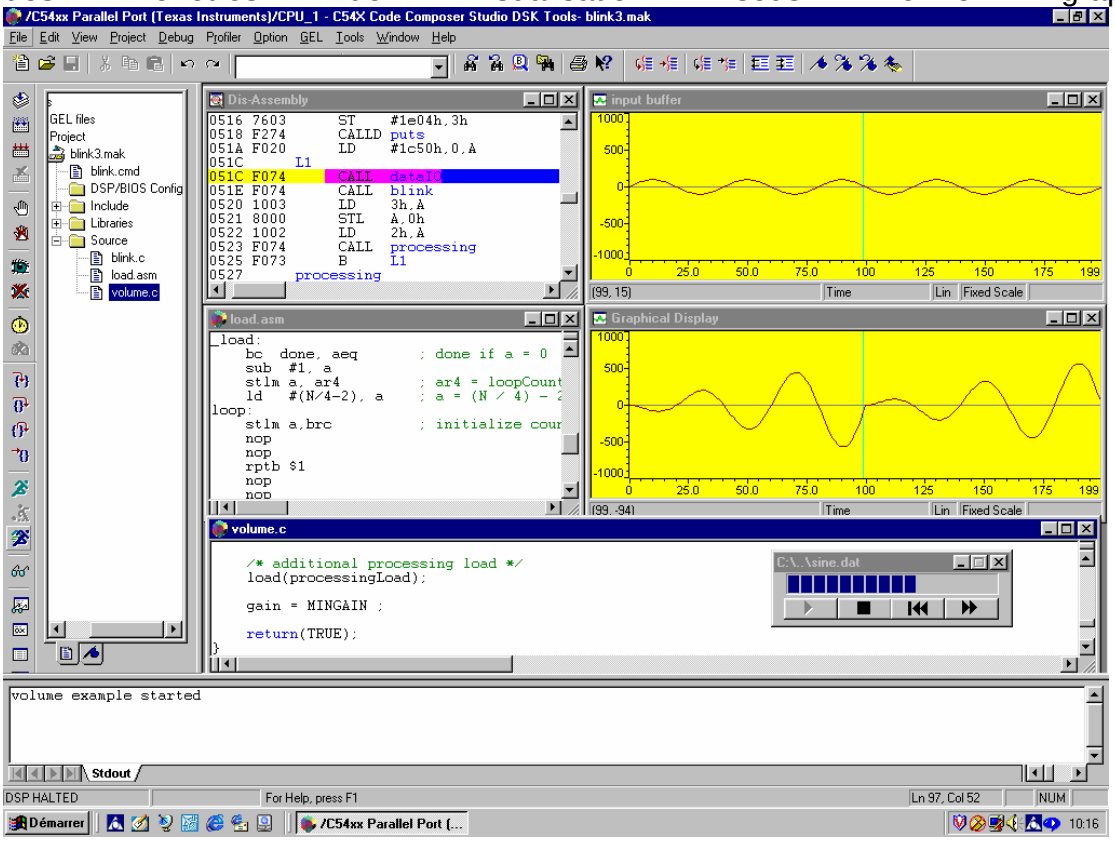
Notre outil de développement durant les séances de TP est le logiciel CODE COMPOSER STUDIO de Texas Instrument. Ce logiciel est couplé à la carte DSL TMS320C5510 qui comporte un DSP. Pour installer correctement ce logiciel il était nécessaire de suivre une procédure de connexion et de mise sous tension de la carte. Nous avons pu vérifier le bon fonctionnement de la carte DSP en utilisant le programme de diagnostic de CCS.

- **L'environnement de travail**

CODE COMPOSER STUDIO est un logiciel qui permet de développer en langage C et en assembleur. Lors des séances de TP, nous étudierons principalement le langage assembleur.
Le workspace regroupe les fichiers de développement. Il est composé entre autre d'un fichier .asm contenant le programme, d'un fichier .c dont le main permet d'appeler le programme et d'un dossier Debug regroupant les fichiers issus de la compilation (le .out est le fichier que l'on charge dans le DSP)

La capture d'écran ci-dessous représente l'interface du logiciel avec à gauche les fichiers du workspace et à droite la visualisation de ces fichiers (nous pouvons créer

des fenêtres de visualisation sous forme graphique)



1) Etude des différents modes d'adressage :

Afin de prendre en main CCS, nous avons eu tout d'abord à notre disposition l'exercice 1. Nous avons ainsi pu créer un environnement de travail et étudier le code source qui transforme un alphabet minuscule abcd……yz en une forme majuscule ABCD…..YZ

Le but est ici **d'analyser le mode d'adressage** et de **maîtriser le fonctionnement de ce programme.**

Le DSP comporte trois types d'adressage :

-**l'adressage absolu :** ou l'on désigne un emplacement de la mémoire comme une constante

-**l'adressage direct :** qui permet un adressage par saut

-**l'adressage indirect :** qui utilise les pointeur (*ARN AVEC N de 0 a 7)

Voici des exemples d'adressages utilisés dans l'exercice 1 :

```
mov    #26,AC1
mov    *ar1+, AC0
mov    #MAJUSCULE, ar2
```

2) Les instructions parallèles

Le parallélisme permet au DSP d'effectuer **plusieurs instructions dans le même cycle d'horloge**. Ceci est rendu possible grâce à l'architecture du processeur qui est composé de plusieurs BUS de données et de différents organes de calcul.

Le parallélisme peut s'effectuer entre des instructions identiques ou différentes

Voici 3 exemples de parallélisme réalisés dans l'exercice 2 :

```
mov          #26, BRC0     // parallélisme utilisant la même instruction
||mov        #0, AC0

mov    #26, AC1            // parallélisme utilisant des instructions différentes
||mpy *AR1,*AR2,AC2


mov    AC0, *ar2+
||sub   #1, AC1
```

L'utilisation des instructions parallèles permet donc d'exécuter plus rapidement un programme.

Toutefois, il n'est pas possible de paralléliser des instructions utilisant le même BUS ou le même organe de calcul.

Par conséquent, afin de paralléliser correctement des instructions, il est nécessaire de connaître parfaitement l'architecture interne du TMS320C5510.

.

3) L'instruction repeat

Cette commande nous permet de **répéter un nombre désiré de fois un groupe d'instruction.**
Ce nombre est à charger dans un des registres BRC de la CPU.

Il existe deux « Block-repeat counter » (BRC0 et BRC1) de 16 bits chacun

*Exemple de l'utilisation de l'instruction **repeat** dans l'exercice 2 :*

```
efface
        mov  #MAJUSCULE, ar2

                    mov         ST1,AC1
                    or          #0x8000,AC1
                    mov         AC1,ST1

                    mov         #26, BRC0  // le block d'instruction sera répété
26 fois
                    ||mov       #0, AC0
        rptb  finblock
```

4) Exercice 1 annoté

```
        .bss    MAJUSCULE, 26 // placé à l'adresse 003046h dans la data memory
        .data

minuscule   .byte   "zyxwvutsrqponmlkjihgfedcba"

.text
        .global _ex1
        _ex1

efface
  mov  #MAJUSCULE, ar2    //bloc qui efface les 26 majuscules
   mov          #26,AC1
 mov   #0, AC0
suivant     mov        AC0, *ar2+
sub             #1, AC1
bcc     suivant, AC1!=0

debut   mov  #minuscule, ar1      // L'alphabet en minuscule est chargé dans AR1
mov    #MAJUSCULE, ar2   // L'alphabet en majuscule (pour l'instant vide) est
chargé dans AR2

mov    #26,AC1        // compteur initialisé à 26
```

```
 loop
  mov          *ar1+, AC0 //le contenu de ar1 (une minuscule) est chargé dans AC0, puis incrémentation de *ar1 (pour la prochaine lettre)

sub     #20h, AC0   // on soustrait 20 pour la conversion en majuscule (code ASCI)
 mov   A, *ar2+       // on charge la majuscule en sortie

        sub              #1, AC1
        bcc      loop, AC1!=0          // on réitère la boucle 26 fois pour obtenir l'alaphabet complet
ret.end
```

Cet exercice nous aura permis de nous familiariser avec les instructions de base du langage assembleur et de maîtriser l'environnement de travail de CCS.

5) Exercice 2 annoté

```
efface
          mov  #MAJUSCULE, ar2   //bloc qui efface les majuscules en utilisant l'instruction repeat

;                       mov          ST1,AC1                    ;
                           or          #0x8000,AC1
;                       mov          AC1,ST1

                        mov          #26, BRC0
                        ||mov        #0, AC0
          rptb  finblock
suivant     mov       AC0, *ar2+
finblock        nop


debut           nop

        mov    #minuscule, ar1
        mov    #MAJUSCULE, ar2

        mov    #26, AC1
        ||mpy *AR1,*AR2,AC2
  loop
        mov    *ar1+, AC0
        mov          AC0, *ar2+
        ||sub         #1, AC1
bcc     loop, AC1!=0
```

L’exercice 2 réalise la même fonction que l’exercice 1. La différence réside dans l’utilisation de l’instruction repeat.
Cet exercice nous aura permis de réaliser des instructions parallèles.

NB : Les instructions parallèles en gras ne sont que des tests. Elles n’ont aucun but de produire un résultat sur le programme mais simplement de vérifier quelles instructions peuvent s’exécuter parallèlement

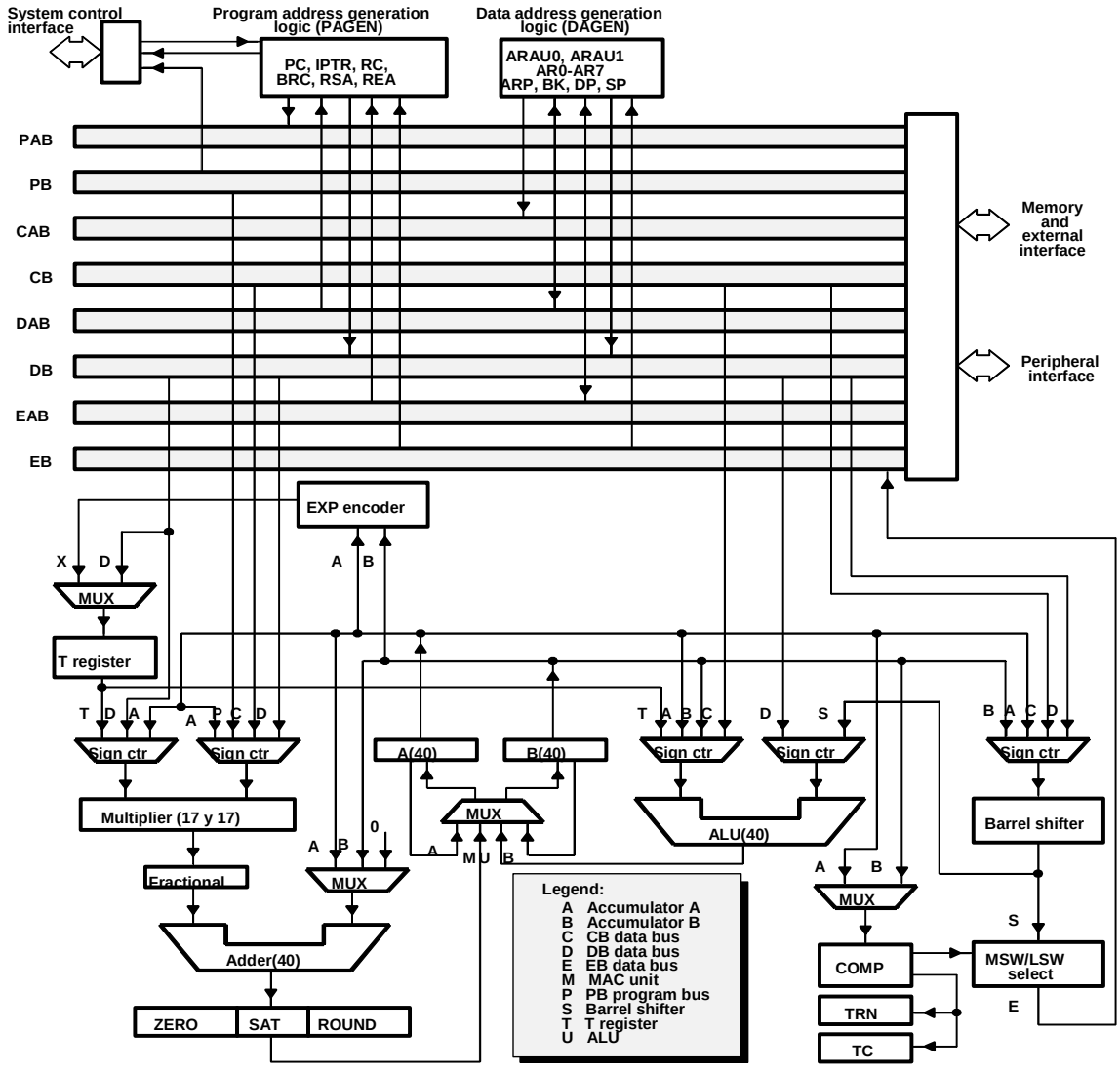
SEANCE 2

Objectifs :

Le premier objectif de cette séance est de **se familiariser avec l'adressage circulaire**. D'autres notions telles **que l'architecture du DSP** et **l'organisation de la mémoire du DSP** seront étudiés.
D'un point de vue de programmation, il sera nécessaire de connaître la procédure d'adressage circulaire afin de comprendre l'exercice 3. Nous verrons enfin **l'utilisation de l'instruction MAC** dans l'exercice 4.

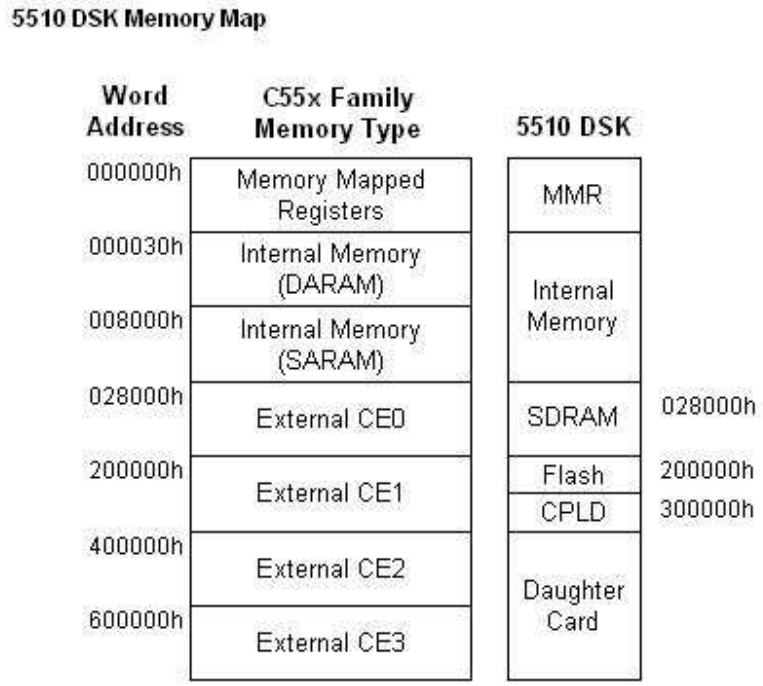
1) Architecture du DSP

Lors de la dernière séance, nous avons compris l'intérêt de connaître l'architecture du DSP pour savoir ce que l'on pouvait paralléliser. En préparant cette séance nous avons donc étudier le schéma de la documentation afin de finaliser le parallélisme de l'exercice 2.



On peut observer sur ce schéma les différents BUS et organes de calculs. L'analyse de l'architecture nous a permis de comprendre pourquoi certains parallélismes avaient crées des erreurs de compilation à la dernière séance.

2) L'organisation de la mémoire du DSP



3) L'adressage circulaire et son analyse dans l'exercice 3

En préparation de ce TP, nous avons analysé un autre mode d'adressage : l'adressage circulaire. L'intérêt est ici de ne pas utiliser de variable pour modifier le registre.

En effet, les registres AR0-AR7 et le CDP peuvent être configurés pour être modifiés circulairement.

Voici la procédure à suivre pour implémenter un buffer circulaire :

1ère étape :

On initialise la taille du buffer circulaire en utilisant une des 3 registres BK03, BK47 ou BKC .
Cette taille correspond au nombre de mots pour un buffer de mots.

2ième étape :

En configurant le bit ST2_55, on définit la circularité des pointeurs.

3ième étape :

Le buffer de mot doit être placé dans l'une des 128 pages principales.
Pour cela on initialise le registre étendu approprié (XARy ou XCDP)

4ième étape :

Il s'agit d'initialiser l'adresse de départ du buffer.

5ième étape :

On charge notre pointeur avec une valeur comprise entre 0 et la taille du buffer-1

Dans l'exercice 3, nous avons pu voir un exemple de buffer circulaire. Pour bien comprendre la configuration, il est nécessaire de se référer à la table suivante :

Pointer	Linear/Circular Configuration Bit	Supplier of Main Data Page	Buffer Start Address Register	Buffer Size Register
AR0	ST2_55(0) = AR0LC	AR0H	BSA01	BK03
AR1	ST2_55(1) = AR1LC	AR1H	BSA01	BK03
AR2	ST2_55(2) = AR2LC	AR2H	BSA23	BK03
AR3	ST2_55(3) = AR3LC	AR3H	BSA23	BK03
AR4	ST2_55(4) = AR4LC	AR4H	BSA45	BK47
AR5	ST2_55(5) = AR5LC	AR5H	BSA45	BK47
AR6	ST2_55(6) = AR6LC	AR6H	BSA67	BK47
AR7	ST2_55(7) = AR7LC	AR7H	BSA67	BK47
CDP	ST2_55(8) = CDPLC	CDPH	BSAC	BKC

```
mov   #6, BK03   // la taille du buffer circulaire est de 6
bset  AR2LC      //  On choisi de configurer AR2 en pointeur circulaire (il est possible de choisir AR2 car on a choisi BK03)
```

amov #000000h, xar2 // le buffer de mot est placé dans la page principale 0.

mov #MAJUSCULE, BSA23 //L'adresse de départ du buffer circulaire sera celle de majuscule (d'après le tableau ci-dessus, il faut utiliser BSA23)
mov #0000h, ar2 // On charge notre pointeur à la valeur 0.

4) L'instruction MAC et son intérêt dans l'exercice 4

Lors de cette séance, nous avons utilisé l'instruction MAC (multiply and accumulate). La syntaxe utilisée est la suivante :

MAC[R] ACx, Tx, ACy[, ACy]
Cette instruction réalise une multiplication et une accumulation dans le D-unit MAC. On obtiendra : ACy = ACy + (ACx * Tx)

Cette opération se révèle particulièrement utile dans l'exercice 4. En effet, il s'agit d'implémenter un filtre passe bas du premier ordre dont l'équation discrétisée est la suivante:

yn= a1*Yn-1 + b1*Un-1 + b0*Un

On utilisant 3 fois l'instruction MAC, on pourra obtenir cette équation récursive.

5) Exercice 3 annoté

Debut

{bloc qui efface les majuscules}

```
debut     nop

mov       #minuscule, ar1
 ld       #26,AC1

 mov      #6, BK03        // bloc qui permet d'implémenter le buffer circulaire (détaillé précédemment)
bset      AR2LC
amov      #000000h, xar2
mov       #MAJUSCULE, BSA23
mov             #0000h, ar2
```

```
loop        mov    *ar1+, AC0
                    sub    #20h, AC0
            mov    A, *ar2+
            sub     #1, AC1
            bcc     loop, AC1!=0


                   ret

            .end
```

Cet exercice est une variante des deux exercices précédents. Il s'agit toujours d'obtenir des majuscules en sortie mais cette fois en utilisant un buffer circulaire. Les majuscules s'inscriront circulairement dans AR2. La taille du buffer circulaire étant de 6, les 6 premières majuscules s'inscriront puis elles seront « écrasées » par les 6 suivantes et ainsi de suite.

6) Exercice 4 annoté

Dans cet exercice, le but est de mettre au point un filtre à réponse impulsionnelle infinie
Son équation est la suivante : yn= a1*Yn-1 + b1*Un-1 + b0*Un
Il nous donc été nécessaire de calculer les coefficients du filtre à l'aide des équations :

$a1 = \tau/(Te+\tau)$, $b1 = Te/(Te+\tau)$ et $b0=0$

Il fallait ensuite convertir le résultat en code Q8 selon le principe suivant :

1 correspond à tous les bits à 1 soit 255, puis l'on fait une règle de proportionnalité sur les résultats calculés précédemment.

```
input           .word
        0,0,0,0,0,0,0,0,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,un,
un,un,un,un          // génération d'un échelon en entrée du filtre
coeff           .word  a1,b1,b0  //  nos coefficient regroupés dans le mot coeff
.bss    OUTPUT, 32


un      .set    0x0100
```

```
a1      .set    0x00D1          //coefficient du filtre calculé en Q8     ;
b1      .set    0x0017
b0      .set    0x0017

exposant      .set    -8          // nombre de décalage à réaliser après multiplication
.text
      .global _ex4
_ex4

efface
        mov  #OUTPUT, ar2
        mov   #32,AC2
        mov   #0,AC0

   suite_eff        mov   AC0, *ar2+
                    sub   #1, AC2
                    bcc   suite_eff, AC2!=0  // bloc qui permet d'effacer la sortie du filtre


debut_fil      mov   #input, ar1    // entrée dans AR1
                mov  #OUTPUT, ar2  // sortie dans AR2
               mov   #31,AC2



loop                mov  #coeff,AR3
                    mov   #0,AC0 //  il faut initialiser AC0 à 0 en vue de l'utilisation prochaine de MAC qui utilise cette valeur

                    mac   *ar2+,*ar3+,AC0   //on charge dans AC0: 0+a1*(la première sortie) (val= a1*y(0))
                     mac  *ar1+,*ar3+,AC0    // val=a1*y(0) + b1*u(0)
                    mac   *ar1,*ar3+,AC0     // val=a1*y(0) + b1*u(0) + b0*u(1)

              // les commentaires du MAC sont dans le cas de la première boucle
                    sftl          AC0,exposant  // on décale car on est en mode Q8
                          mov          AC0, *ar2          // rangement de l'échantillon de sortie le pointeur est en position pour l'échantillon suivant

                    sub   #1, AC2
                    bcc   loop, AC2!=0
```

SEANCE 3

Objectif : cette séance nous a permis de découvrir l'acquisition de données en tant réel grâce au **codec du DSP**. De plus, il nous permettra de **découvrir le filtrage d'un sinus bruité** au travers de l'exo6.

1) Acquisition d'un signal

Présentation du CODEC

Durant ce TP nous avons voulu recevoir un signal à l'aide du Codec DSP. Ce Codec est stéréo (sur deux voies) , il va acquérir les signaux analogiques reçus sur l'entrée de la carte(line in) afin de les convertir en données numériques utilisables par le DSP. A noter que l'opération inverse peut être également effectué c'est-à-dire retranscrire les données numérique en signaux analogiques afin de pouvoir par exemple les visualiser a l'oscilloscope.

Principe de DMA

L'exercice 7 qui nous permet de recevoir un signal extérieur fait intervenir le principe de DMA (direct memory acces). Le DMA est un module qui permet d'effectuer une opération en hardware afin de soulager le processeur ce module sera notamment utilisé lors d'un échange de données ave un périphérique externe.

Le principe du PING-PONG

Ensuite ces donnés une fois numérisée et envoyée par le codec doivent être stockée. C'est a ce moment qu'arrive alors le principe du ping et du pong . Pour cela on utilisera deux registres étant donné que l'on travaille en stéréo.

Le principe du ping et du pong est en fait relativement simple les données reçus seront d'abord stockées sur le ping sur lequel les opérations désirées peuvent être réalisées une fois celui-ci rempli et Pendant ce temps le pong se remplit.

Manipulation de l'exercice 7

Une fois le principe de Codec assimilé nous avons pu recevoir une sinusoïde a l'aide de l'exercice7. La seule modification que nous avons apporté à ce programme consisté a modifier la fréquence d'échantillonnage afin de pouvoir observer le phénomène GIBBS qui correspond à un non respect de la fréquence de Shannon ce qui provoque une absence de signal en sortie.

Présentation de l'exercice 7 ,

Nous avons inclus a ce niveau uniquement le code source qui permet la configuration du Codec, étant donné que c'est la seul chose que nous avons modifié

Le codec se configure en utilisant le DSK5510_AIC23:

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	CLKOUT	CLKIN	SR3	SR2	SR1	SR0	BOSR	USB/Normal
Default	0	0	0	1	0	0	0	0	0

```
DSK5510_AIC23_Config config =
{ \
   0x001B,  /* 0 DSK5510_AIC23_LEFTINVOL  Left line input channel volume 17
0dB 1B +6dB*/ \
   0x001B,  /* 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume 17
0dB 1B +6dB*/\
   0x01f9,  /* 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone volume */  \
   0x01f9,  /* 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume */
\
   0x0010,  /* 4 DSK5510_AIC23_ANAPATH    Analog audio path control */      \
   0x0000,  /* 5 DSK5510_AIC23_DIGPATH    Digital audio path control */     \
   0x0000,  /* 6 DSK5510_AIC23_POWERDOWN  Power down control */             \
   0x0043,  /* 7 DSK5510_AIC23_DIGIF      Digital audio interface format */ \
   0x008D,  /* 8 DSK5510_AIC23_SAMPLERATE permet de modifier la fréquence
d'échantillonnage ex 81=48 kHz 8D=8kHz
   0x0001   /* 9 DSK5510_AIC23_DIGACT     Digital interface activation */   \
```

2) Filtrage d'une sinusoïde bruitée

Pour le filtrage de notre sinusoïde nous n'utiliserons pas un signal venant de l'extérieur mais une sinusoïde qui sera déclarée dans le software. Cependant nous la déclarerons de manière à simuler une provenance extérieure, c'est-à-dire en utilisant les registres d'entrée ping et pong.

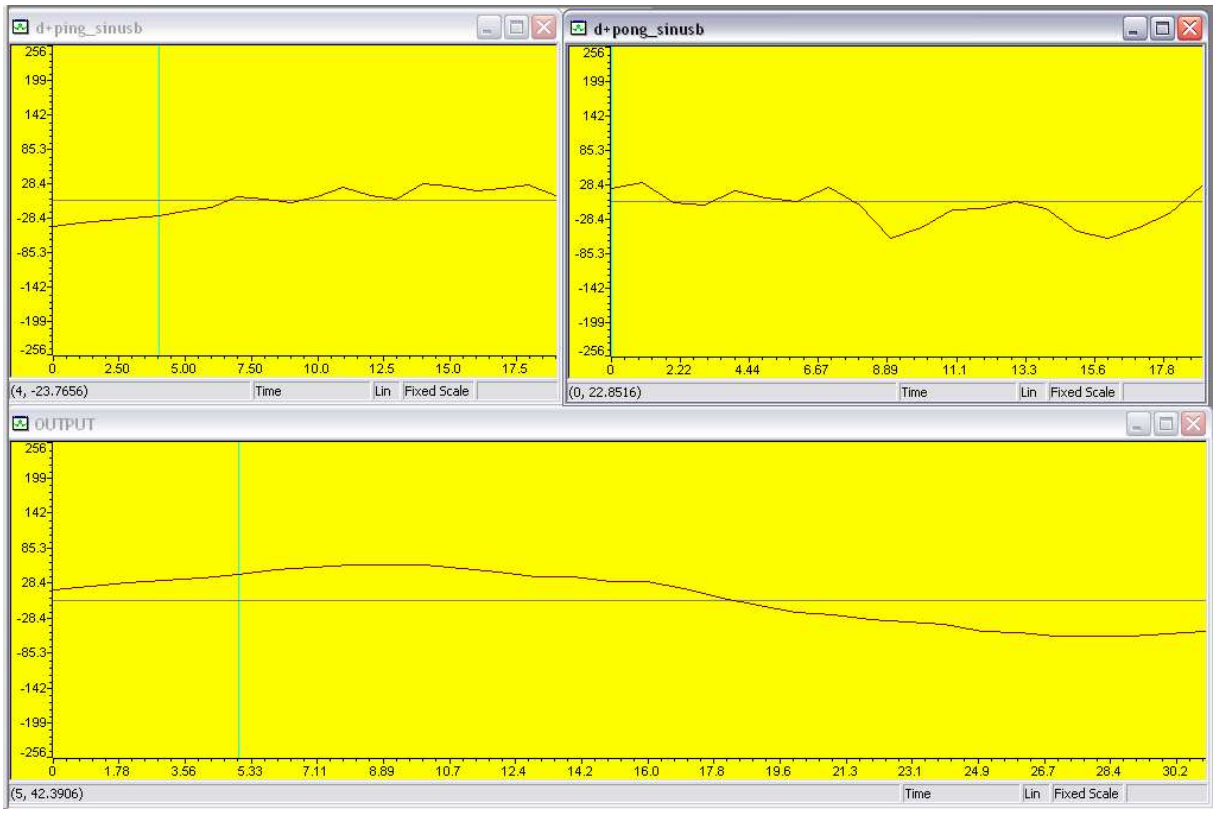
Cependant la réalisation de notre filtre RII nous impose de connaître les 4 échantillons précédent l'échantillon que nous désirons calculer. C'est pourquoi nous avons introduit un delay_buffer ping et un delay_buffer pong avant d'y stocker ces échantillons.

Ainsi une fois ces déclarations faites nous avons pu réaliser notre filtre RII grâce à l'instruction FIRSADD qui permet de réaliser simultanément une multiplication, une addition et une incrémentation. Cette opération étant réalisé 4 fois consécutive nous avons pu utiliser la fonction repeat qui permet d'optimiser le mode pipeline du DSP et ainsi réaliser toutes ces opérations en un seul cycle d'horloge.

3)Exercice 6 annoté :

```
;Sous programme appelé par le main.c du dossier ex6_b_dsk55
exposant      .set    -8              ;exposant nombre de décalage à droite
.text
      .global _ex6fir

_ex6fir
debutfir      ; réalisation d'une boucle pour couvrir l'étendu du buffer (blksize)

mov           T0,T0         ; blksize est transféré par T0

; Positionnement des pointeurs, AR0 sera incrémenté pour balayage

boucle        mov           ar0, ar3      ; AR3 sert à pointer les échantillons du filtre
              mov           ar3, ar4      ; le pointeur AR4 est positionné à AR2 +4
              add           #4,ar4

              mov           ar1, CDP      ; le pointeur de coefficient est sur h0




              mov           #0,AC0 // initialisé à 0 pour le premier firsadd
              mov           #0,AC1

              rpt           #3  // répète 3 fois la ligne en dessous. Celle-ci est
donc exécutée 4 fois. Il n'ya alors besoin que d'un seul firsadd


              firsadd *ar4-,*ar3+,*CDP+,AC0,AC1 ; nouveau AC0=x(n-1)+x(n-5),
AC1=(AC1 d'avant)+h0*(AC0 d'avant)
              ;firsadd *ar4-,*ar3+,*CDP+,AC0,AC1 ; AC1=h1*[x(n-1)+x(n-5)],
AC0=x(n-2)+x(n-4)
              ;firsadd *ar4-,*ar3+,*CDP+,AC0,AC1 ; AC1=h1*[x(n-1)+x(n-5)] +
h2*[x(n-2)+x(n-4)], AC0=x(n-3)+x(n-3)
              ;firsadd *ar4-,*ar3+,*CDP+,AC0,AC1 ; AC1=h1*[x(n-1)+x(n-5)] +
h2*[x(n-2)+x(n-4)] +h3*[x(n-3)+x(n-3)], pas besoin de diviser par deux=>car
notre cœfficient h3 a déjà été divisé par deux lors de sa déclaration.

              sfts    AC1,exposant   ; mode Q8
mov    AC1, *ar2+     ; rangement de AC1 dans le Buffer de sortie pointer par
AR2
              amar  *ar0+         ; pour décaler d'un échantillon
              sub   #1, T0
              bcc   boucle, T0 !=0

;
```

Courbes obtenues sur Code Composer Studio



SEANCE 4

Objectif : Cette séance va nous permettre de réaliser la fusion de l'exercice 6 et de l'exercice 7. En effet nous allons ici acquérir une sinusoïde bruitée en temps en réel dans le but de la filtrer, à l'aide du filtre réalisé à l'exercice 6.

Manipulation :

Il y a ici **deux principales méthodes** pour réaliser le filtrage de la sinusoïde.

La première consiste à **programmer le filtre directement en langage C** dans le fichier ex7_dsk55.c en remplaçant le filtre « transparent » initial par celui de l'exercice 6.

La deuxième consiste à **« appeler » le filtre de l'exercice 6** programmé en assembleur dans le fichier ex7_dsk55.c

Pendant cette séance nous n'avons pu expérimenter que la deuxième méthode.

Voici les différentes modifications apportées à l'exercice 7 :

Tout d'abord, il est nécessaire de déclarer notre fonction de l'exercice 6 comme étant **une fonction externe** :

```
extern exo6fir(Int16 *data,Int16 *coeff, Int16 *results, Int16 blksize);
```

Il faut ensuite **préciser les coefficients du filtre :**

```
#define        NBCOEFF              4
int coeff[NBCOEFF]={0,0x0009,0x0033,0x0029}
```

Enfin **on appelle la fonction** en désignant correctement les paramètres :

```
{

   ex6fir(data,coeff,results,blksize);

}
```

L'intérêt d'avoir deux méthodes différentes est de comparer la rapidité de l'une est de l'autre. Cependant nous n'avons pu essayer la programmation directe du filtte en langage C .