Corrigé du
Devoir surveillé de DSP (Processeur de Traitement de Signal)
I3 Electronique
Benoît Decoux – 13 janvier 2004
Durée : 2 heures - Tous documents et calculatrice autorisés

I) Exercices

Exercice 1 : Arithmétique des DSP (4 points)

1) Donner la valeur binaire et la valeur hexadécimale correspondant à la valeur décimale 23. (0,5 point)

On peut donner d'abord la valeur binaire correspondante : la méthode consiste à effectuer des divisions successives par 2, jusqu'à tomber sur un résultat nul. Les bits sont constitués par le reste des divisions successives.

On code d'abord le nombre en binaire :

$$23/2=11+1 \; ; \; 11/2=5+1 \; ; \; 5/2=2+1 \; ; \; 2/2=1+0 \; ; \; 1/2=0+1$$

d'où

$$23 = 10111$$

Puis en hexadécimal :

$$23 = \$17$$

2) Donner le code complément à 2 du nombre binaire 00010111. (1 point)

Dans ce mode de codage les poids des bits sont :

$$-2^{n-1}, 2^{n-2}, 2^{n-3}, \ldots 2^0$$

Pour obtenir le complément à 2 on inverse tous les bits :

$$00010111 \rightarrow 11101000$$

puis on ajoute 1

$$11101001$$

Enfin, on fait la conversion binaire → décimale :

$$11101001=-2^7\times1+2^6\times1+2^5\times1+2^4\times0+2^3\times1+2^2\times0+2^1\times0+2^0\times1$$
$$=-128+64+32+8+1=-128+105=-23$$

3) Donner la valeur décimale du nombre hexadécimal \$1111 (0,5 point)

Les poids des bits sont des puissances de 16. Pour n bits :

$$16^{n-1}, 16^{n-2}, 16^{n-3}, \ldots 16^0$$

Donc :

$$\$1111=1\times4096+1\times256+1\times16+1\times1=4369$$

4) Donner la valeur décimale correspondante à 6, codé en virgule fixe (sur 4 bits) (0,5 point)

Conversion en binaire :

$$6=0110$$

Dans ce mode de codage les poids des bits sont :

$$-2^0, 2^{-1}, 2^{-2}, 2^{-3}$$

On a donc :

$$6 \leftrightarrow 2^{-1}+2^{-2}=0,5+0,25=0,75$$

5) Donner la valeur hexadécimale opposée de ce nombre, en code complément à 2, et vérifier le résultat à l'aide de la représentation en virgule fixe. (0,5 point)

Code complément à 2 :

$$0110 \rightarrow 1001+1=1010=\$A$$

Vérification du résultat :

$$\$A=-2^{0}+2^{-2}=-1+0,25=-0,75$$

6) Quelle est la plus petite valeur positive que l'on puisse coder en virgule fixe sur 8 bits (donner la valeur décimale correspondante) (0,5 point)

Seul le bit de poids le plus faible est à 1 :

$$\$01=00000001$$

Valeur décimale correspondante :

$$2^{-7}=0,0078125$$

7) Donner la valeur hexadécimale 8 bits représentant l'arrondi de la valeur $9999. Préciser les valeurs décimales correspondantes. (0,5 point)

Le principe de l'arrondi est de ne garder que les bits de poids fort et d'ajouter le bit de poids le plus fort de la partie enlevée.

$$\$9999=1001100110011001$$

Dans le cas de notre valeur ce bit est à 1. La valeur arrondie sur 8 bits est donc :

$$10011001+1=10011010=\$9A$$

Exercice 2 : Programmation basique en assembleur DSP56303 (3 points)

1) Ecrire un programme qui copie en mémoire X les valeurs hexadécimales $0 à $F (à l'aide d'instructions assembleur). (1 point)

On peut s'inspirer du programme d'exemple recopiant le contenu d'un tableau dans un autre (en mode d'adressage en bits inversés, dans le cadre d'une TFD) disponible sur le site web. Un exemple de programme réalisant l'opération demandée est :

```
        org     X:$0            ; Choix d'un emplacement mémoire (arbitraire mais non utilisée ailleurs)
memx    dsm     16              ; Reservation de 16 emplacements mémoire

        org     p:$100
        move    #memx,R0
        clr     A
        do      #16,fin         ; Répéter 16 fois jusqu'au label "fin"
        move    A,X:(R0)+
        inc     A
fin
        end
```

2) Compléter le programme pour qu'il multiplie chacune de ces valeurs par 2 et stocke les résultats en mémoire Y. (1 point)

On ajoute en début de programme le code suivant :

```
        org     Y:$0            ; Choix d'un emplacement mémoire (arbitraire mais non utilisée ailleurs)
memy    dsm     16              ; Réservation de 16 emplacements mémoire
```

et avant la fin du programme :

```
        move    #memy,R1
        move    #2,Y0
        do      #16,fin2
        move    X:(R0)+,X0
        mpy     X0,Y0,A
        move    A,Y:(R1)+
fin2
```

3) Modifier le programme pour qu'il multiplie toutes ces valeurs par elle-mêmes (et non plus par 2) et stocke les résultats en mémoire Y. (0,5 point)

On multiplie chaque valeur avec elle-même en chargeant 2 accumulateurs 24 bits (ici X0 et Y0) avec la même valeur et en utilisant l'instruction de multiplication MPY :

```
        move    #memy,R1
        do      #16,fin2
        move    X:(R0),X0
        move    X:(R0)+,Y0
        mpy     X0,Y0,A
        move    A,Y:(R1)+
fin2
```

4) Modifier le programme pour qu'il calcule $\sum_{x=0}^{15} x^2$.(0,5 point)

Il suffit de remplacer l'instruction MPY par l'instruction de multiplication et accumulation MAC, dans le programme précédent.

Exercice 3 : Filtrage en C (3 points)

1) Donner la méthode de détermination des coefficients d'un filtre numérique, pour qu'il possède les caractéristiques d'un filtre analogique passe-bande du 2nd ordre (sans détailler le calcul). (1 point)

On utilise les formules qui permettent le passage de la fonction de transfert analogique à la fonction de transfert échantillonnée :

$$T(j\omega) = \frac{2\xi j\frac{\omega}{\omega_0}}{1+2\xi j\frac{\omega}{\omega_0}+\left(j\frac{\omega}{\omega_0}\right)^2} \qquad \rightarrow \qquad H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

Pour passer de la fonction de transfert analogique à la fonction de transfert numérique, on remplace jω par p et on effectue le changement de variable directement issu de la transformée bilinéaire (simplifiée) :

$$\frac{p}{\omega_c} = \frac{f_e}{\pi f_c}\cdot\frac{z-1}{z+1}$$

On identifie la fonction de transfert en z obtenue avec la forme générale des fonctions de transfert en z, pour obtenir les coefficients du filtre analogique en fonction des paramètres du filtre analogique. On obtient :

$$a_0 = \frac{2}{k}\xi\frac{f_e}{\pi f_c}\ ;\ a_1 = 0\ ;\ a_2 = -a_0\ ;\ b_1 = \frac{2}{k}\left(1-\left(\frac{f_e}{\pi f_c}\right)^2\right)\ ;\ b_2 = \frac{1}{k}\left(1-2\xi\frac{f_e}{\pi f_c}+\left(\frac{f_e}{\pi f_c}\right)^2\right)$$

2) Ecrire une fonction C réalisant le calcul des coefficients du filtre numérique et le filtrage, permettant de générer un échantillon de sortie traité, en fonction d'un échantillon d'entrée. Cette fonction pourra faire appel à une autre fonction, dont il faudra préciser le prototype et les arguments. (1 point)

On peut s'inspirer de la fonction réalisant la même operation dans le cas d'un filtre passe-bas, donné en exemple sur le site (dans le programme "fltr_fich_c.c").

```
FLOAT
passebande_2deg_iir(FLOAT e, FLOAT fe, FLOAT fc, FLOAT xi, UBYTE *ia, UBYTE *ib, FLOAT *em, FLOAT *sm)
{
        FLOAT    k, k1, k1c, k2, a[3], b[2];

        k1=fe/M_PI/fc;
        k1c=k1*k1;
        k2=2*xi*k1;
        k=1+k2+k1c;
        a[0]=k2/k;
        a[1]=0.;
        a[2]=-a[0];
        b[0]=-2.*(1.-k1c)/k;
        b[1]=-(1-k2+k1c)/k;
        return iir(e, a, 3, b, 2, ia, ib, em, sm);
}
```

Cette fonction utilise la fonction de traitement d'un échantillon, dont le prototype pourrait être :

```
FLOAT   iir(FLOAT e, FLOAT *a, UWORD na, FLOAT *b, UWORD nb, UBYTE *ia, UBYTE *ib, FLOAT *em, FLOAT *sm) ;
          // e      échantillon d'entrée
          // a      tableau des coefficients a(i)
          // na     nombre de coefficients a(i)
          // b      tableau des coefficient b(i)
          // nb     nombre de coefficients b(i)
          // ia     pointeur dans le tableau des a(i)
          // ib     pointeur dans le tableau des b(i)
          // em     mémoire des échantillons d'entrée passés
          // sm     mémoire des échantillons de sortie passés
          // La fonction retourne l'échantillon traité
```

Remarque : pour améliorer la compacité de l'écriture, on a utilisé des types de variables re-definis :

```
#define UBYTE     unsigned char
#define UWORD     unsigned short
#define FLOAT     float
```

3) On suppose que l'on dispose d'une fonction réalisant la lecture d'un bloc d'échantillons de signal à partir d'une entrée audio et d'une fonction d'envoi d'un bloc d'échantillon à la sortie audio, dont les prototypes sont respectivement :

```
float       *entree_bloc_audio();
void        sortie_bloc_audio(float *buf);
```

Ecrire le programme principal utilisant la fonction précédemment écrite, comprenant les initialisations nécessaires. (1 point)

L'initialisation consiste en une allocation de mémoire pour les coefficients a_i et b_i et les échantillons passés de e et de s, et le bloc d'échantillons (ou déclaration de tableaux statiques, respectivement a[3], b[2], em[2], sm[2], et par exemple buf[NB_ECH]):

```
float     a[3], b[3], em[2], sm[2];
float     buf[NB_ECH];                    // NB_ECH préalablement défini
```

Le traitement proprement dit consiste en une boucle infinie :

```
while(1)          // boucle infinie
{
          buf=lect_bloc_audio() ;
          for(i=0 ; i<NB_ECH ; i++)
                    *(buf+i)=passehaut_2deg_iir(*(buf+i), fe, fc, xi, ia, ib, em, sm) ;

          sortie_bloc_audio(buf) ;
}
```

Exercice 4 : Filtrage en assembleur DSP56303 (6 points)

1) Ecrire une fonction en assembleur DSP56303, appliquant un filtrage passe-bas du 1er ordre à un échantillon présent dans l'accumulateur B (sans utiliser les traitements parallèles du DSP). Utiliser l'implémentation directe. Commenter le programme. (2 points)

$$H(j\omega) = \frac{1}{1 + j\frac{\omega}{\omega_c}} \qquad \rightarrow \qquad H(z) = \frac{a_0 + a_1 z^{-1}}{1 + b z^{-1}}$$

avec

$$a_0 = \frac{1}{k} \; ; \; a_1 = 1 \; ; \; b = \frac{1}{k}\left(1 - \frac{f_e}{\pi f_c}\right) \text{ avec } k = 1 + \frac{f_e}{\pi f_c}$$

Dans la forme directe, on mémorise des valeurs passées des échantillons d'entrée et de sortie. La mémoire nécessaire pour cette implémentation est :

- 3 mots pour les coefficients a_i et b_i
- 1 mot pour les échantillons d'entrée
- 1 mot pour les échantillons de sortie

```
a0        equ       0.0001/2
a1        equ       …..
b         equ       …..

          org       X:$0              ;
coef      dsm       3                 ; réservation de 3 mots mémoire

          org       Y:$0
en        dsm       1
sn        dsm       1

          org       X :coef           ;
          dc        a0,a1,b           ; copie des coefficients en mémoire

          move      #coef,R0          ; R0 pointe sur la zone mémoire 'coef'
          move      #2,M0             ; zone de mémoire 'coef' modulo 3
          move      #en,R4            ; R4 pointe sur la zone mémoire 'en'
          move      #sn,R5            ; R5 pointe sur la zone mémoire 'sn'

          clr       A                 ; A=0
          move      A,Y :(R4)         ; en=0
          move      A,Y :(R5)         ; sn=0

          move      X:(R0)+,X0        ; X0=a0
```

```
filtrage
        move    A,Y1                ; Y1=e(n)
        mpy     X0,Y1,A             ; A=a0.e(n)
        move    X:(R0)+,X0          ; X0=a1
        move    Y:(R4),Y0           ; Y0=e(n-1)
        mac     X0,Y0,A             ; A=A+a1.e(n-1)
        move    X:(R0)+,X0          ; X0=b
        move    Y:(R5),Y0;Y0=s(n-1)
        macr    -X0,Y0,A ; A=A-b.s(n-1)=s(n)/2
        move    Y1,Y:(R4); e(n) dans e(n-1)
        asl     A                   ; A=2.A=s(n)
        move    X:(R0)+,X0          ; X0=a0
        move    A,Y:(R5)            ; s(n) dans s(n-1)
rts
```

2) Optimiser la durée d'exécution de la fonction de filtrage écrite ci-dessus, en utilisant les traitements parallèles du DSP. (1 point)

```
filtrage
        move    A,Y1                                        ; Y1=e(n)
        mpy     X0,Y1,A     X:(R0)+,X0      Y:(R4),Y0       ; A=a0.e(n)     X0=a1       Y0=e(n-1)
        mac     X0,Y0,A     X:(R0)+,X0      Y:(R5),Y0       ; X0=b          Y0=s(n-1)   A=A+a1.e(n-1)
        macr    -X0,Y0,A    Y1,Y:(R4)                       ; e(n) dans e(n-1)      A=A-b.s(n-1)=s(n)/2
        asl     A           X:(R0)+,X0      A,Y:(R5)        ; X0=a0         s(n) dans s(n-1)    A=2.A=s(n)
rts
```

3) Ecrire un programme de l'implémentation canonique correspondante au programme écrit ci-dessus. (1,5 point)

On peut utiliser le programme proposé dans le corrigé du TP 1, implémentant un filtre du 2nd ordre. Il suffit alors de supprimer des parties car le 1er ordre comporte moins de termes que le 2nd ordre.

$$s(n) = a_0 w(n) + a_1 w(n-1) \text{ avec } w(n) = e(n) - b_1 w(n-1)$$

$$\frac{s(n)}{2} = \frac{a_0}{2} w(n) + \frac{a_1}{2} w(n-1) \text{ et } \frac{w(n)}{2} = \frac{e(n)}{2} - \frac{b_1}{2} w(n-1)$$

```
filtrage                                                ;il faut A=e(n) ici
        asr     A                                       ;e(n)=e(n)/2
        move    X:(R0)+,X0      Y:(R4)+,Y0              ;X0=w(n-1), Y0=b1
        mac     -X0,Y0,A                                ;A=A-b1*w(n-1)=w(n)
        asl     A                                       ;w(n)=w(n)*2
        move    A,X1                                    ;X1=w(n) (sauvegarde temporaire pour memorisation)
        move    Y:(R4)+,Y0                              ;Y0=a0
        mpy     X1,Y0,A                                 ;A=a0*w(n)
        move    X:(R0)+,X0      Y:(R4)+,Y0              ;X0=w(n-1), Y0=a1
        mac     X0,Y0,A                                 ;A=A+a1*w(n-1)
        move    X1,X:(R0)                               ;w(n)->w(n-1)
        asl     A                                       ;A=2.A=s(n) (car les coefficients ont ete divises par 2)
        rts
```

4) Ecrire un programme réalisant un filtre passe-bas d'ordre 4 par la mise en cascade de 2 filtres passe-bas du 2nd ordre identiques à ceux décrits ci-dessus dans l'implémentation directe. Indiquer les limitations d'un tel filtre, par rapport à un filtre passe-bas du 2nd ordre. (1,5 point)

Il suffit de copier une fois la liste des instructions de la fonction de filtrage, en enlevant toutefois le remplacement de s(n-1) par s(n) dans la première partie.

```
filtrage
        move    A,Y1                                        ; Y1=e(n)
        mpy     X0,Y1,A     X:(R0)+,X0      Y:(R4),Y0       ; A=a0.e(n)     X0=a1           Y0=e(n-1)
        mac     X0,Y0,A     X:(R0)+,X0      Y:(R5),Y0       ; X0=b          Y0=s(n-1)       A=A+a1.e(n-1)
        macr    -X0,Y0,A    Y1,Y:(R4)                       ; e(n) dans e(n-1)          A=A-b.s(n-1)=s(n)/2
        asl     A           X:(R0)+,X0                      ; A=2.A=s(n)    X0=a0

        move    A,Y1                                        ; Y1=e(n)
        mpy     X0,Y1,A     X:(R0)+,X0      Y:(R4),Y0       ; A=a0.e(n)     X0=a1           Y0=e(n-1)
        mac     X0,Y0,A     X:(R0)+,X0      Y:(R5),Y0       ; X0=b          Y0=s(n-1)       A=A+a1.e(n-1)
        macr    -X0,Y0,A    Y1,Y:(R4)                       ; e(n) dans e(n-1)A=A-b.s(n-1)=s(n)/2
        asl     A           X:(R0)+,X0      A,Y:(R5)        ; A=2.A=s(n)    X0=a0           s(n) dans s(n-1)
rts
```

Les limitations de ce filtre par rapport à un filtre passe-bas du 2nd ordre est que l'on ne pourrait pas avoir de phénomène de résonance, c'est à dire que cela reviendrait à avoir un filtra passe-bas du 2nd ordre avec ξ>0,7.

Exercice 5 : FFT (tranformée de Fourier rapide) en C et en assembleur (4 points)

1) A partir des propriétés de la Transformée de Fourier Discrète (TFD), donner l'état des 64 sorties obtenues par l'application de cette transformée à 8 échantillons d'un signal sinusoïdal composé de 4 périodes, d'amplitude 0,5. (1,5 point)

Les propriétés de la TFD (vue comme un bloc fonctionnel dont on ne considère que les entrées/sorties) utiles ici sont :

- Les sorties de la TFD sont cryptées, c'est à dire se trouvent dans l'ordre correspondant à l'adressage binaire inversé ;
- Soient 0 à N-1 les indices des sorties de la TFD ; les sorties sont symétriques par rapport à la sortie d'indice N/2-1 ;
- Les sorties de la TFD sont des complexes ;
- La sortie d'indice i correspond à une composante fréquentielle de fréquence

$$f_i = i \times \frac{f_e}{N} \qquad i=0,\ldots,N-1 \qquad (1)$$

où f_e est la fréquence d'échantillonnage.

On peut déduire de ces propriétés que seules 2 sorties seront différentes de 0 : la 5e et la (8-5)e=3e, dans l'ordre naturel. Or le résultat brut de la TFD est dans l'ordre binaire inversé, donc après remise dans l'ordre naturel ce seront les 5e et 6e qui deviendront différentes de 0, car 5=101 ne change pas par renversement des bits, par contre 3=011 donne 110=6. L'amplitude de ces composantes sera 0,7, la racine carrée de 0,5, car le module de la sortie complexe doit être égale à 0,5.

2) Ecrire une fonction C permettant de réaliser l'inversion des bits d'un nombre, et l'appliquer à l'une des sorties non-nulles de la TFD définie ci-dessus, en pas-à-pas. (1 point)

On peut utiliser la fonction vue en cours, qui retourne le nombre en bits inversés d'un nombre n (b est le nombre de bits du nombre à traiter) :

```
int
bitrev(int n, int b)
{
        int m, r ;

        r=0 ;
```

```
        for(m=b-1 ; m>=0 ; m--)
          if((n>>m)==1)
          {
                r+=two(b-1-m) ;
                n-=two(m) ;
          }
        return r ;
}
```

avec

```
#define two(x)      (1<<(x))   //puissance de 2 de x
```

qui implémente l'algorithme :

```
r=0
pour m=b-1 à 0 par pas de –1
        si n>>m=1 (*)
                r=r+2^(b-1-m)
                n=n-2^m
```

L'exécution de cette fonction avec une TFD de 8 échantillons va donner :

n=5 (=101) ; b=3
b-1=3-1=2
r=0

m=2	n>>2=1	$r=0+1\times2^0=1$	n=1
m=1	n>>1=0		
m=0	n>>0=1	$r=1+1\times2^2=5$	n=0

3) Ecrire une fonction assembleur DSP56303 permettant de remettre dans l'ordre naturel les sorties de la TFD définie ci-dessus (avec les initialisations adéquates du programme principal). (1,5 point)

```
points  equ     8
…
        org     X:0             ; zone source pour les parties réelles
data1   dsm     points

        org     X:points        ; zone destination pour les parties réelles
data2   dsm     points

        org     Y:0             ; zone source pour les parties imaginaires
data1   dsm     points

        org     Y:points        ; zone destination pour les parties imaginaires
data2   dsm     points
…

bitrev
        move    #data1,R1       ; R1 pointe sur la zone source 'data1' (partie réelle)
        move    R1,R2           ; R2 : idem pour la partie imaginaire
        move    #data2,R3       ; R3 pointe sur la zone destination 'data2' (partie réelle)
        move    R3,R4           ; R4 : idem pour les parties imaginaires
        move    #-1,M1          ; programmation de l'adressage linéaire pour M2
        move    M1,M2           ; idem pour les parties imaginaires
        move    #0,M3           ; programmation de l'adressage 'bit reverse' pour M3
        move    M3,M4           ; idem pour les parties imaginaires
        move    #points/2,N3    ; nombre d'adresses qu'il faut inverser -> N3
                                ; (= la moitié du tableau seulement)
        move    N3,N4           ; idem pour les parties imaginaires
        do      #points,fin_cop ; boucle de copie sur 'points' itérations
        move    X:(R1)+,X0      ; lecture de la valeur réelle dans la mémoire source X
        move    Y:(R2)+,Y0      ; idem pour la partie imaginaire
        move    X0,X:(R3)+N3    ; écriture de la valeur dans la mémoire destination (avec adressage en 'bit reverse')
        move    Y0,Y:(R4)+N4    ; idem pour la partie réelle
fin_cop rts
```