Devoir surveillé de DSP (Processeur de Traitement de Signal)
I3 Electronique

Benoît Decoux – vendredi 7 janvier 2005

Durée : 2 heures - Tous documents et calculatrice autorisés

Le corrigé de cet examen sera disponible en début de semaine prochaine sur le site www-dsp.efrei.fr

Exercice 1 : Arithmétique des DSP (4 points)

1) Quelle est la plage de valeurs du format virgule fixe [1,24] ?

Petite erratum : il s'agit plutôt du format [1,23], c'est à dire à 24 bits. Ca ne change pas grand chose au résultat.

La plage de valeurs est $[-1, +1-2^{-23}]$; 23 est le nombre de bits à droite de la virgule, 2^{-23} est la précision du codage :

$$2^{-23}=0,0000001192$$

donc

$$1-2^{-23}=0,99999988$$

La réponse était dans le document « Arithmétique des DSP ».

2) Donner la valeur décimale correspondant à la valeur hexadécimale $111111 exprimée au format virgule fixe [1,24].

En binaire, cette valeur est égale à

$$000100010001000100010001$$

En virgule fixe, on rajoute la virgule :

$$0,000100010001000100010001$$

On n'a plus qu'a multiplier les bits à 1 par leur poids :

$$2^{-3}+2^{-7}+2^{-11}+2^{-15}+2^{-19}+2^{-23}$$
$$=0,125+0,007125+0,0004883+\ldots$$
$$=0,132613\ldots$$

Les autres termes de l'addition ne faisaient qu'augmenter la précision, pas les chiffres les plus significatifs (on n'était pas obligé de tout calculer).

3) Donner la valeur hexadécimale exprimée dans le format virgule fixe [1,24], correspondant à la valeur décimale 0,125.

On pouvait trouver la réponse directement en se souvenant que le poids du 1er bit à droite de la virgule est 0,5, le 2e 0,25, le 3e 0,125, etc. Donc le résultat est 0,001.

Avec la méthode de conversion par calcul, on a :

- Nombre positif donc bit de gauche (=bit de signe) =0 ;
- Partie fractionnaire :
  0,125 : 0,125×2=0,25 → 0 ; 0,25×2=0,5 → 0 ; 0,5×2=1,0 → 1 ; 0,0×2=0 → que des 0 ;
  Donc résultat =0,001.
  Voir paragraphe « Conversion entre formats » dans document « Arithmétique des DSP ».

4) Quelle est la plus grande valeur positive que l'on peut coder dans le format mixte [9, 47] ? (Donner la valeur hexadécimale et la valeur décimale)

La partie entière comporte 9 bits, dont un bit de signe.

Le principe est le même qu'au 1) :

$$2^{9-1}-2^{-47}=2^{8}-2^{-47}=256-2^{-47}$$
$$=256-7{,}105\times10^{-15}$$
$$\approx255{,}999999999999993$$

Exercice 2 : Programmation de base en assembleur DSP56303 (4 points)

1) Ecrire un programme qui recopie en mémoire Y le contenu d'une zone de 16 valeurs de la mémoire X. Le commenter.

On pouvait s'inspirer du programme « bit_rev.asm » qui a pour but de tester l'adressage avec renversement des bits des adresses, sauf qu'ici on ne fait que recopier un tableau sans modifier l'ordre des adresses, c'est à dire des éléments du tableau.

Un exemple de programme est donc :

```
points  equ     16              ; nombre de valeurs
data1   equ     $10             ; adresse du tableau dans la mémoire source
data2   equ     $30             ; adresse du tableau dans la mémoire destination

        org     x:data1         ; réservation de l'espace mémoire de départ (mémoire X)
        ds      points

        org     y:data2         ; idem dans la mémoire d'arrivée (mémoire Y)
        ds      points

        org     p:$400          ; programme principal
        move    #data1,r2       ; r2 pointe sur la zone 'data1'
        move    #data2,r3       ; r3 pointe sur la zone 'data2'
        move    #-1,m2          ; mode d'adressage linéaire, facultatif ici (mode par défaut,
        move    #-1,m3          ; de plus on n'a pas besoin de revenir en début de tableau)
        do      #points,fin_cop ; boucle de copie sur 'points' itérations
        move    x:(r2)+,x0      ; lecture de la valeur dans la mémoire source
        move    x0,y:(r3)+      ; écriture de la valeur dans la mémoire destination
fin_cop                         ; fin de la boucle
        jmp     *               ; tourne en rond et ne fait rien
        end
```

2) Compléter le programme pour qu'il multiplie chacune de ces valeurs par 2 lors du transfert.

Il suffit d'utiliser un accumulateur A ou B plutôt que X0, ce qui permet d'utiliser l'instruction de décalage à gauche (=multiplication par 2) : « asl ». La boucle de copie devient :

```
        do      #points,fin_cop ; boucle de copie sur 'points' itérations
        move    x:(r2)+,a       ; lecture de la valeur dans la mémoire source
        asl     a               ; multiplication du contenu de A par 2
        move    a,x:(r3)+       ; écriture de la valeur dans la mémoire destination
fin_cop                         ; fin de la boucle
```

3) En utilisant l'adressage modulo, copier cette même zone de la mémoire X 10 fois dans la mémoire Y (ces copies étant concaténées).

Cette fois ci on peut utiliser l'adressage modulo pour la lecture, puisqu'on veut lire plusieurs fois le même tableau.

De plus, on imbrique la boucle existante dans une nouvelle boucle de 10 itérations.

Les modifications à apporter au programme sont donc :

```
        move    #15,m2          ; mode d'adressage modulo (taille du tableau -1)
        …
        do      #10,fin_tot
        do      #points,fin_cop ; boucle de copie sur 'points' itérations
        …                       ; pas de changements
fin_cop                         ; fin de la boucle
fin_tot
```

4) Ecrire un programme qui génère un tableau de 256 valeurs de la fonction sinus sur une période, à l'adresse $100 dans la mémoire Y.

Le programme d'exemple de la FFT « test_fft.asm » comporte une telle fonction, sous la forme d'une macro utilisant des directives d'assemblage (et donc exécutée avant le lancement du programme, lors de l'assemblage).

```
pi      equ     3.14159
freq    equ     2.0*pi/@cvf(points)   ; 1 période de sinus pour « points » valeurs
….
sincosr  macro points,coef            ; paramètres de la macro : « points » (nombre de points) et « coef » (adresse tableau)
sincosr  ident 1,1
        org     y:coef
count   set     0
        dup     points
        dc      @sin(@cvf(count)*freq)
count   set     count+1
        endm
        endm            ; fin macro sincos
```

Exemple d'appel de la macro :

```
        fft   64,$100
```

Exercice 3 : Filtrage en C++ (4 points)

1) Ecrire une classe "Filtre1o" permettant de programmer des filtres numériques du 1er ordre. Cette classe devra posséder comme membres : 1) des fonctions (= méthodes) : constructeur, destructeur, filtrage d'un échantillon, filtrage d'un tableau d'échantillons, et 2) des variables adéquates.

L'implémentation d'un filtre du 1er ordre est beaucoup plus simple que celle du 2e ordre étudiée en cours, puisqu'on n'a pas à gérer de tableaux d'échantillons avec pointage circulaire dans ce tableau : on a seulement besoin de e(n-1) et s(n-1), soit 2 variables. Pour les coefficients, on a seulement besoin d'un tableau de 2 éléments pour les a_i (coefficients des e(n-i)), et d'une variable pour b, coefficient de s(n-1) :

$$s(n)=a_0.e(n)+a_1.e(n-1)+b.s(n-1)$$

```
class Filtre1o {
  public:
        Filtre1o ();                                    // constructeur
        ~Filtre1o ();                                   // destructeur
        float   iir(float e);                           // filtrage d'un échantillon
        float*  filtrage_buf(float *buf, int nb_ech);   // filtrage d'un tableau d'échantillons

  //variables publiques, pour pouvoir etre utilisees en dehors des fonctions ci-dessus
        float   fc;             // fréquence de coupure
        float   fe;             // fréquence d'échantillonnage
                                // (pas vraiment une caractéristiques du filtre mais nécessaire…)
  //variables privees (accessibles uniquement dans les fonctions ci-dessus)
  private:
        int     type;           // par exemple 1 pour passe-bas, 2 pour passe-bande, etc
        float   a[2], b;        // tableau et variable pour coefficients du filtre (resp. numerateur et denominateur)
        float   em, sm;         // variables pour memorisation des echantillons passes d'entree et de sortie, resp.
};
```

2) Ecrire la fonction de filtrage d'un échantillon citée ci-dessus, dont le seul argument est l'échantillon d'entrée de type float nommé "e" et la valeur retournée l'échantillon de sortie.

Cette fonction implémente l'équation de récurrence nécessitant la mémorisation d'un seul échantillon d'entrée et un seul échantillon de sortie :

```
float
Filtre2d::iir(float e)
{
        float s;

        s=a[0]*e+a[1]*em+b*sm;      // em et sm doivent être initialisés à 0 en début de programme
        em=e;                       // mémorisation de l'échantillon d'entrée courant e(n) (futur e(n-1))
        sm=s;                       // mémorisation de l'échantillon de sortie courant s(n) (futur s(n-1))
        return s;
}
```

3) Ecrire la fonction de filtrage d'un tableau d'échantillons de type float nommé "tab_e", dont les arguments sont un pointeur sur ce tableau et sa taille, et la valeur retournée un pointeur sur un tableau des échantillons résultant du traitement.

Tel que l'énoncé est posé, on est obligé d'allouer la mémoire pour le tableau des échantillons de sortie dans cette fonction (ce qui n'est pas la meilleure méthode).

```
float*
Filtre2d::filtrage_tab(float *tab_e, int nb_ech)
{
        int i;
        float *tab_s;

        tab_s=(float*)malloc(nb_ech*sizeof(float));
        for(i=0 ; i<nb_ech ; i++)
                *(tab_e+i)=iir(*(tab_e+i));
        return tab_s;
}
```

4) Donner le programme principal permettant d'utiliser la fonction précédente et d'appliquer à un tableau d'échantillons "tab_e" deux filtrages en parallèle, la sortie de chaque filtre étant passée par un gain fixe paramétrable et les sorties des deux filtres étant additionnées pour générer un tableau "tab_s". On supposera 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 *tab_s);
```

```
Filtre f1, f2;
float *tab_1, *tab_2, *tab_s;
..
// ici doivent être lancées des fonctions d'initialisation des filtres f1 et f2 (calcul des coefficients a(i) et b(i),
// initialisation des variables em et sm…
…
tab_s=(float*)malloc(nb_ech*sizeof(float));
…
while(1)                        //boucle infinie sur les blocs
{
        tab_e=entree_bloc_audio();                          //lecture bloc
        tab_1=f1::filtrage_tab(float *tab_e, int nb_ech)    // 1er filtrage
        tab_2=f2::filtrage_tab(float *tab_e, int nb_ech)    // 2e filtrage (appliqué au même tableau d'échantillons)
        for(i=0 ; i<nb_ech ; i++)
                *(tab_s+i)=gain1*(tab_1+i)+ gain2*(tab_2+i);    // on suppose gain1 et gain2 initialisés
        sortie_bloc_audio(tab_s);                           // sortie bloc
}
```

Exercice 4 : Filtrage en assembleur DSP56303 (4 points)

1) Ecrire une routine en assembleur, appliquant l'opération de dérivation suivante à un échantillon d'entrée :

$$s(n)= e(n)-e(n-1)$$

s(n) est l'échantillon de sortie et e(n) l'échantillon d'entrée, n est le pas d'échantillonnage $k \times T_e$ avec k entier et T_e période d'échantillonnage.

On supposera cet échantillon présent dans l'accumulateur A à l'entrée dans la routine, et il faudra mettre l'échantillon résultat avant la sortie dans ce même accumulateur.

Préciser les autres parties à ajouter éventuellement au programme de départ (programme réalisant l'opération entrée-sortie directe "pass.asm", par exemple).

On peut s'inspirer du programme de filtrage étudié en TP « filtrage.asm », mais celui-ci est beaucoup plus simple puisqu'on n'a pas besoin de mémoriser plusieurs échantillons : plutôt que de les stocker en mémoire, on peut les sauvegarder dans un registre disponible (X0, X1, Y0, Y1, etc). Un exemple de programme est :

```
deriv
        move        A,Y:en          ; mémorisation de e(n) car on va écraser A
        sub         X0,A            ; A=A-X0  <=> s(n)=e(n)-e(n-1)
        move        Y:en,X0         ; e(n) -> e(n-1)
        rts
```

On suppose que Y:en a été alloué par exemple par :

```
        org Y:0
en      ds  1
```

2) Même question pour l'opération d'intégration suivante :

$$s(n)= e(n)+s(n-1)$$

L'allocation mémoir est similaire à celle du 1). Pour le reste, un exemple de programme est :

```
integr
        move        Y:sn,X0         ; récupération de s(n-1) dans X0 (péalablement mémorisé)
        add         X0,A            ; A=A+X0 <=>s(n)=e(n)+s(n-1)
        move        A,Y:sn          ; mémorisation de s(n) (futur s(n-1))
        rts
```

Cette fois-ci il faut initialiser à 0 la variable stockée dans la mémoire (Y:sn), puisque dans la boucle on commence par la lire ; par exemple en faisant :

```
        move        #0,X1
        move        X1,Y:sn
```

3) Même question pour l'effet audio appliquant un écho répétitif à un signal, décrit par l'équation de récurrence suivante :

$$s(n)= e(n)+a \times s(n-4800), \text{ avec } a<1$$

Le principe est le même que pour l'intégration, mais avec les différences suivantes :

- dans la mémoire il faut sauvegarder 4800 échantillons : impose d'utiliser un registre d'adresse (par exemple R0) ;

- s(n) est multiplié par une constante (qu'on appellera "gain") : on peut utiliser l'instruction "mpyi", qui permet de multiplier le contenu d'un registre 24 bits avec une valeur immédiate (24 bits également).

D'où l'exemple de programme :

```
        org         Y:0
gain    ds          1
sn      dsm         4800
…
; à mettre avant la boucle infinie de lecture-écriture des échantillons
        move        sn,R0           ; R0 pointe sur le début du tableau (=s(n-d))
...
echo
        move        A,Y1            ; e(n) -> Y1 (en vue du "mpy")
        mpyi        #gain,Y1,A      ; gain*s(n-d) -> A
        move        Y:(R0),X0       ; récupération de s(n-d) dans X0
        add         X0,A            ; A=A+X0 <=> s(n)=e(n)+s(n-1)
        move        A,Y:(R0)+       ; mémorisation de s(n) : écrase l'échantillons le plus ancien (futur s(n-d))
        rts
```

4) Dans le cas du traitement précédent, indiquer la durée du décalage temporel entre les répétitions successives du son à la fréquence maximale d'échantillonnage de la carte à DSP utilisée en travaux pratiques, et préciser si ce traitement peut être implanté sur cette même carte (en justifiant la réponse).

La fréquence d'échantillonnage maximal de la carte est 48000 Hz. Donc 4800 échantillons représentent 0,1 seconde. Ce traitement peut être implémenté sur les cartes utilisées en TP car elles comportent 2 blocs mémoire X et Y de 4096 mots-mémoire (de 24 bits), mais il faut alors gérer ces 2 blocs dans le programme, ce qui le complique.

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

1) Décrire en quelques phrases ce que fait le programme suivant :

```
nb_val  equ     8
tab1    equ     $20
tab2    equ     $40
genetab macro   points
        org     x:tab1
cmpt    set     0
        dup     nb_val
        dc      cmpt
cmpt    set     cmpt+1
        endm
        endm
        org     x:tab1
        ds      nb_val
        org     x:tab2
        ds      nb_val
        genetab nb_val
        org     p:$40
        jsr     rout
        jmp     *
rout
        move    #tab1,r0
        move    #tab2,r1
        move    #0,m0
        move    #-1,m1
        move    #nb_val/2,n0
        do      #nb_val,fin
        move    x:(r0)+n0,x0
        move    x0,x:(r1)+
fin     rts
        end
```

Ce programme comporte une macro (« genetab »), une routine (« rout ») et un programme principal.

La macro, qui s'arrête au 2e « endm » copie en mémoire X, à l'adresse $20, 8 valeurs variant de 0 à 7.

La routine comporte une boucle de 8 itérations allant lire le contenu du tableau « tab1 », dans la mémoire X, en mode binaire inversé, c'est à dire avec renversement des bits des adresses. Ce mode d'adressage est obtenu en initialisant à 0 le registre m0, associé au registre d'adresse r0, et le registre n0 à la moitié de la taille du tableau, soit « nb_val/2 » (=4). Ce contenu est copié dans le tableau « tab2 », également en mémoire X mais à l'adresse 40, adressé lui en mode linéaire.

Le programme principal (qui commence à « org p :$40 ») se limite à un appel de la routine « rout ». Il est précédé des allocations mémoires nécessaires aux tableaux « tab1 » et « tab2 » (en fait, pour « tab1 » elle n'est pas nécessaire puisque la macro s'en charge avec la directive « dc »).

2) Donner la valeur des indices des éléments d'un tableau en contenant 16, après renversement des bits des valeurs initiales (0, 1,… 15) de ces indices.

On part des indices du tableau :

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

On écrit ces indices en binaire, puis on leur applique une symétrie centrale.

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0000 1000 0100 1100 0010 1010 0110 1110 0001 1001 0101 1101 0011 1011 0111 1111

Ce qui donne :

0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15

3) Donner l'état des sorties de la TFR (Transformée de Fourier Rapide, ou FFT) appliquées à 256 échantillons d'un signal sinusoïdal composé de 2 périodes, d'amplitude égale à 1.

Cet exercice est très proche d'un autre donné dans l'examen de l'année dernière.

Seules 2 sorties seront différentes de 0 : la 2e et la (256-2)e=254e, dans l'ordre naturel. Or le résultat brut de la TFR est dans l'ordre binaire inversé. Il faut donc appliquer le renversement des bits des adresses de ces 2 éléments, exprimées sur 8 bits puisqu'il y a 256 échantillons :

adresse(2)=00000010	donne	01000000=64
adresse(254)=11111110	donne	01111111=127-1=126

Donc après remise dans l'ordre binaire inversé ce seront les 64e et 126e qui deviendront différentes de 0.

La valeur de ces sorties sera $1/\sqrt{2}$, car elles correspondront à la partie réelle d'un nombre complexe dont l'amplitude est égale à 1 (l'amplitude du sinus analysé) :

$$\sqrt{\left(\frac{1}{\sqrt{2}}\right)^2+\left(\frac{1}{\sqrt{2}}\right)^2}=1$$