DESS CCI : corrigé examen Langage Machine, Décembre 2004

1 Entiers, décalages et boucles

Question a : Donner une séquence C équivalente à la boucle de nb_uns du programme ci-dessous, sans instruction for (en utilisant if et goto).

```
init_for: i=0;
test_for: if (! (i<16)) goto fin_for;   /* ou if (i>= 16) goto ... */
corps_for:    moins_un_si_bit = ...
   /* corps du for inchange */
                i++;
                goto test_for;
fin_for: return (-res);
```

Ou variante avec 1 seul branchement conditionnel :

```
init_for: i=0;
                goto test_for; /* on peut se passer de ce goto */
                                    /* parce que i=0 implique <16 */
                                    /* et au moins un passage dans la */
                                    /* boucle */
corps_for:    moins_un_si_bit = ...
   /* corps du for inchange */
                i++;
test_for: if (<16) goto corps_for;
fin_for: return (-res);
```

Question b : Traduire en langage d'assemblage ARM la boucle for de nb_uns. On ne vous demand pas le prologue et ni l'épilogue (sauvegarde, restauration de registres, allocation de mémoire dans la pile) de la fonction.

La convention d'appel standard dit que le premier paramètre (ici x) est stocké dans le registre r0. Au moment où l'appelante copie x de la mémoire vers r0 (par un ldrsh) avant appel, la représentation de x est étendue à 32 bits par recopie du bit de signe (16 fois) en poids forts.

```
              .text
init_for:     mov     r2,#0           @ i = 0
                                      @ test de i<16
test_for:     cmp     r2,#16          @ ou   rsbS     r0,r2,#16
              bge     fin_for         @      blt     fin_for
corps_for:    add     r4, r2, #16     @ r4 = 16 + i
              mov     r3, r0, LSL r4  @ moins_un_si_bit = x << r4
              mov     r3, r3, LSR #31 @ moins_un_si_bit >>= 31
              add     r1, r1, r3      @ res += moins_un_si_bit
              add     r2,r2,#1        @ i++
              b       test_for
fin_for:      rsb     r0,r1,#0        @ return (-res) : resultat dans r0
              /* epilogue */          @ par convention d'appel
              mov     pc,lr
```

Question b : Comment représente-t-on -1 en binaire sur 32 bits ?
32 bits à 1 : $-2^{31} + \sum_1^{30} x_i 2^i) = -2^{31} + (2^{31} - 1) = -1$

Question c : Expliquer en quelques lignes le principe de la méthode de comptage du nombre de uns utilisée ici.

On décale à gauche chaque bit à tester, pour l'amener en poids fort. Il jouera donc le rôle de bit de signe de l'entier décalé. Le décalage arithmétique à droite recopie en poids fort le bit de signe. Si le bit de départ était à 0, on obtient que des bits à zéros, donc 0. Si le bit de départ était à 1, on obtient que des uns, donc -1.
Au signe près, le cumul donne le nombre de bits à 1.

Question d : Traduire en langage d'assemblage ARM les deux appels à la procédure nb_uns présents dans le corps de main.

L'extension et la réduction de format entre 16 et 32 bits se fait automatiquement lors des ldr et str.

```
@ passage du paramètre dans r0
@ r0 = *&a
ldr r1,= a
ldrh r0, [r1]
@ appel de la procédure
                bl nb_uns
@ résultat stocké dans r0 par convention
ldr r1,= b
strh r0, [r1]

ldr r1,= b
ldrh r0, [r1]
@ appel de la procédure
                bl nb_uns
@ résultat stocké dans r0 par convention
ldr r1,= c
strh r0, [r1]
```

Question e : Quel est l'effet de la séquence de code ARM suivante ? Ecrire une instruction C correspond à cette séquence.

```
mov r0, #5
mov lr, pc
ldr pc, =nb_uns
mov r5,r0
```

Toute affection au compteur ordinal est un branchement. L'instruction ldr pc,= nb_uns fonctionne l'instruction de branchement b, à ceci près que l'adresse est donnée de manière absolue dans un registre, au lieu d'êtrev calculée par addition d'un déplacement relatif au compteur ordinal.
L'instruction move qui précède permet de sauvegarder dans lr l'adresse de l'instruction move, ce qui reconstitue l'équivalent d'une instruction bl (à la méthode de codage de l'adresse de destination près).
La séquence réalise donc un appel à la procédure nb_uns, avec 5 comme paramètre, et stockage du résultat de l'appel dans r5 : r5=nb_uns(5) ;.

Questione e : Expliquer en quelques lignes comment traduire en langage d'assemblage ARM la dernière instruction de main. Cette dernière instruction rend-elle le programme récursif ?.

Il suffit d'appeler nb_uns avec le contenu de c comme paramètre. Il suffit d'utiliser le résultat de ce premier appel comme paramètre du deuxième appel et le résultat du

deuxième appel comme paramètre du troisième appel.
Coup de chance pour les fainéants : la convention d'appel fait que le résultat est déposé dans r0 à la place du premier argument. Il se trouve donc déjà à la bonne place pour l'appel suivant.

```
ldr r1,=c
ldrh r0, [r1]
bl nb_uns
bl nb_uns
bl nb_uns
ldr r1,=b
strh r0,[r1]
```

Le appels à nb_uns ne sont pas emboîtés les uns dans les autres et le programme n'est pas récursif. On aurait pu écrire :

```
void main()
{
register unsigned int r5, t;
b = nb_uns(a);
c = nb_uns(b);
t = nb_uns(c);
t = nb_uns(t);
t = nb_uns(t);
```

2 Pointeurs et tableaux

2.1 Déclarations

Question : traduire en langage d'assemblage ARM la procédure xplusplus (inutile de détailler le prologue et l'épilogue de la procédure) ainsi que la déclaration des deux tableaux.

```
                .bss
y:              .skip   4
z:              .skip   4

.data
x:              .short  4
u:              .short  32

tab_ptrshort    .word x
                .word   y
                .word   z
                .word   u
```

```
tab_proc:       .word xpluspplus
                .word yplusplus
                .word zplusplus
                .word uplusplus

                .text
                .global uplusplus
uplusplus:      ...                 @ prologue
                ldr   r0, = u       @ r0 = &u
                ldrsh r1, [r0]
                add   r1, r1, #1
                strh  r0, [r0]
                ...                 @ epilogue
                mov   pc, lr
                .ltorg
```

2.2 Tableau de fonctions

Question : traduire en langage d'assemblage ARM l'instruction **ptr_proc = tab_proc[i] ;**.

```
                                    @ stockage de ptr_proc dans r4
                                    @ stockage de i dans r5
                ldr  r0,= tab_proc
                ldrsh  r4, [r0, r5]
                                    @ ajouter un .ltorg plus loin
```

2.3 Pointeur de fonction

Question : traduire en langage d'assemblage ARM l'instruction **ptr_proc = &xplusplus ;**.

```
                                    @ stockage de ptr_proc dans r4
                                    @ stockage de i dans r5
                ldr  r4, = xplusplus
                                    @ ajouter un .ltorg plus loin
```

2.4 Appel de fonction pointée

Question : traduire en langage d'assemblage ARM l'instruction **(*ptr_proc) () ;**.
La relecture de la question 1e peut vous être utile.

```
                                    @ stockage de ptr_proc dans r4
                mov  lr,pc
                mov  pc, r4
```

2.5 Indice versus pointeur

Question : écrire une boucle C while équivalente à la première boucle, n'utilisant que ps (et pas i) comme variable de boucle.

```
ps = tab_ptrshort;
while (ps < (tab_ptrshort + 4))
  {
  *ps = *ps + 1;
  ps ++;
  }
```