M2P CCI : Corrigé Langage Machine, Novembre 2010

1 Variables et pointeurs (20mn)

Traduire en langage d'assemblage les déclarations de variables et les deux affectations suivantes :

```
          .bss
c1:       .skip 1           @ unsigned char c1;
          .balign 2
y:        .skip 2           @ unsigned short y;

          .data
c2:       .byte  'a'        @ unsigned char c2='a';
          .balign 4
ptr:      .word   x         @ unsigned short *ptr = &x;
x:        .hword  4         @ unsigned short  x = 4 ;

          .text
@ r0,r1 : temporaires adresse  r2: temporaire donnée
                               @ y = *ptr devient  *&y = **&ptr
          ldr  r0,=ptr     @ r0 = &ptr
          ldr  r1,[r0]     @ r1 = *r0 = *&ptr = ptr
          ldrh r2,[r1]     @ r2 *r1 = **&ptr = *ptr
          ldr  r0,= y      @ r0 = &y
          strh r2,[r1]     @ *r0 (*&y ou y) = *ptr

                               @   c1 = c2 devient   *&c1 = *&c2
          ldr  r0,= c2     @ r0 = &c2
          ldrb r2,[r1]     @ r2 = *r0 = *&c2 = c2
          ldr  r1,=c1      @ r1 = &c1
          strb r2,[r1]     @ *r1 = r2 : *&c1 = *&c2 : c1 = c2
```

Toutes les variables sont stockées en mémoire.

2 Procédures et comparaisons (30mn)

On considère le programme suivant :

```
              .data
voici_x:      .asciz "x = %d\012"
debut:        .asciz "debut !\012"
fin:          .asciz "fin !\012"
```

```
                .global main
                .text
                                    @ void main (void) {
main:           ldr  r0,=debut        @ printf ("debut !\n");
                bl   printf
                bl   proc             @ proc();
                ldr  r0,=fin
                bl   printf           @ printf ("fin !\n");
                mov  pc,lr          @ }

                                    @ void proc (void) {
proc:           mov  r4, #3           @ int x = 3
                                      @ do {
boucle:         ldr  r0,= voici_x        @ printf ("x = %d\n",x);
                mov  r1, r4
                bl   printf
                sub  r4, r4, #1          @ x = x - 1;
                cmp  r4,#0            @ } while (x != 0);
                bne  boucle
                mov  pc,lr          @ }
```

Si le programme était traduit correctement, son exécution devrait donner l'affichage ci-dessous à gauche. Mais cette traduction contient une erreur et on obtient la trace d'exécution ci-dessous à droite :

```
debut !                          debut !
x = 3                            x = 3
x = 2                            x = 2
x = 1                            x = 1
fin !                            x = -1
                                 x = -2
                                 x = -3
                                 ...       @ et ainsi de suite
```

Question : expliquer **pourquoi** l'exécution **ne s'arrête pas** après l'affichage "x = 1" et n'affichage pas ("x = 0").

On peut constater que le problème ne vient pas de la condition utilisée dans le branchement : le cas x=0 n'est pas traité, ce qui prouve que la sortie de boucle a été réalisée correctement sur x=0.

Mais il semble que l'exécution revienne malgré tout ensuite sur le corps de la boucle. Puisque bge joue son rôle correctement, le problème vient de l'autre branchement : mov pc,lr. L'explication est simple : le contenu de lr est l'adresse stockée par l'instruction bl printf, qui a remplacé dans lr l'adresse de retour dans main.

La cause est la suivante : le corps de la procédure proc modifie le contenu d'un registre (lr) sans le sauvegarder dans le prologue et le restaurer dans l'épilogue.

comment corriger le problème ?

Ajouter "empiler lr" (sub sp,sp,#4 ; str lr,[sp]) avant le corps de proc, et "depiler lr" (ldr lr,[sp] ; add sp,sp,#4) avant l'instruction de retour.

Un étudiant (mais est-il compétent ?) affirme que puisque la constante à laquelle r4 est comparé est nulle, il est possible d'optimiser le code en supprimant l'instruction cmp.

Question :

1. S'il a raison, **que faut-il modifier** dans le reste du programme (les instructions autres que cmp) ?
2. S'il a tort, expliquer **pourquoi** l'instruction cmp est **nécessaire**.

Il faut que l'indicateur Z soit mis à 1 si et seulement si le résultat de la soustraction donne 0. Il suffit pour celà d'utiliser une instruction subS au lieu de sub pour que Z soit positionné correctement : la comparaison à 0 devient inutile.

3 Tableaux, base 2, décalages et rotations (1h10)

3.1 Parcours de tableau (30mn)

Question c : Traduire en langage d'assemblage ARM la boucle for dans le corps de permuter (lignes commentées "à traduire"). Respectez l'allocation des variables aux registres indiquée en commentaire.

```
void permuter (unsigned short r)    // r a stocker dans r0
{
  register unsigned int i;          // a stocker dans r1
  register unsigned int j;          // a stocker dans r2
  register char tmp;                // a stocker dans r3

  for (i=0;i<r;i++)                 // a traduire
    {                               // a traduire
    tmp = tabcar[0];                // a traduire
    for (j=0; j<3; j++)             // a traduire
       tabcar[j] = tabcar[j+1];     // a traduire
    tabcar[3] = tmp;                // a traduire
    }                               // a traduire
}
```

```
                          @ r <-> r0, i <-> r1  j <-> r2 tmp <-> r3
                          @ tabcar : r4   temporaire : r5
          .global permuter
          .text
permuter: stmfd sp!,{r1-r5} @ prologue de permuter

          mov r1, #0        @ i=0;
          ldr r4,= tabcar   @
          b    testwi       @  while (i<r)                @ goto testwi
                            @    {
corpsi:   ldrsb r3,[r4,#0]  @    tmp = tabcar[0];          @ corpsi
          mov  r2, #0       @    j = 0;
          b    testwj       @    while (j <3)                 @ goto testwj
                            @      {
corpsj:   add  r5, r2, #1   @      tabcar[j] = tabcar[j+1];   @ corpsj
          ldrsb r5, [r4,r5]
          strb r5, [r4,r2]                                     @ fin corpsj
          add  r2, r2,#1    @      j++;                        @ maj j

testwj:   cmp  r2, #3       @      }                        @   if (j<3)
          blo  corpsj       @                               @      goto corpsj

          strb  r3, [r4,#3] @    tabcar[3] = tmp;          @ fin corpsi
          add  r1, r1, #1   @    i++;                      @ maj i

testwi:   cmp  r1, r0       @    }                         @ if (i<r)
          blo  corpsi                                      @      goto corpsi

          ldmfd sp!,{r1-r5} @ epilogue de permuter
          mov pc,lr
```

A noter : dans cette version avec test après le corps, on utilise la condition non inversée (saut si $<$). Avec le test devant le corps, il faudrait inverser la condition (saut à fin si $>=$).

Les variables i et j sont déclarées unsigned, donc utiliser les conditions pour entiers naturels : blo ($<$) ou bhs ($>=$).

On utiliserait blt ou bge si i et j vaient été déclarés comme des entiers relatifs (sans attribut unsigned).

```
          // squelette de code avec test en tete
          i = 0;
testi:    if (i >= r) goto fin_wi;
```

```
            tmp = tabcar [0];
            j = 0;
testj:      if (j >= 3) goto fin_wj;
               tabcar[j] = tabcar[j+1];
               j++;
            goto testj;
fin_wj:     tarcar[3] = tmp;
            i++;
          goto testi;
fin_wi:   // epilogue et branchement de retour
```

Voir aussi exemples branchements dans le placard.

3.2 Rotation (30mn)

Voici une autre traduction (optimisée par un programmeur humain) de la procédure permuter :

```
              .data
rotation:     .hword 3          @ unsigned short rotation = 3

              .balign 4
tabcar:       .byte       'w'   @ unsigned char tabcar [4] =
              .byte       'x'   @ {'w','x','y','z'};
              .byte       'y'
              .byte       'z'

              .text
permuter:     stmfd sp!, {r0-r2}
              ldr  r1,= tabcar
              ldr  r2,[r1]
              mov  r0, r0, LSL #3
              mov  r2, r2, ROR r0
              str  r2, [r1]
              ldmfd sp!,{r0-r2}
              mov  pc,lr
```

Question : expliquer le **principe de fonctionnement** de cette traduction sans boucle, puis **pourquoi** cette version ne fonctionne **pas** sur une machine **big endian**.

L'instruction **mov r0, r0, LSL #3** calcule 8*r. En supposant que le tableau soit stocké à une adresse muliple de 4, l'instruction **ldr r2,[r1]** considère le contenu du tableau comme la représentation d'un entier 32 bits, qui sera copié dans le registre r2. Une rotation de 8r bits à droite est appliquée à r2 et l'entier modifié dans r2 est réécrit

en mémoire.

Avec la convention little endian, tabcar[0] représente les 8 bits de poids faibles de l'entier et tabcar[3] les 8 bits de poids forts.

La rotation à droite de 8*r bits équivaut à une rotation à droite de r octets. Chaque octet déplacé de 8 bits vers la droite dans l'entier voit diminuer son poids et sera réécrit à un emplacement plus proche de tabcar[0] : ceci correspond à une recopie de type tabcar[i] ← tabcar[i+1]. De même, un octet déplacé vers la gauche de l'entier voit son poids augmenter et il sera réécrit à un emplacement plus proche de tabcar[3].

Si le contenu initial de tabcar est 0x77, 0x78, 0x79, 0x7a (codes ASCII de w,x,y et z), alors r2 contiendra l'entier 0x7a797877. Le tableau illustre le contenu du registre r2 après la rotation pour différentes valeur de r.

tab[3]	tab[2]	tab[1]	tab[0]	mémoire LE
0x7a	0x79	0x78	0x77	registre (r=0)
0x77	0x7a	0x79	0x78	registre (r=1)
0x78	0x77	0x7a	0x79	registre (r=2)
0x79	0x78	0x77	0x7a	registre (r=3)

Avec une convention big endian, le poids relatif des octets est inversé et la rotation change le sens des recopies : tabcar[i+1] ← tabcar[i].

tab[0]	tab[1]	tab[2]	tab[3]	mémoire BE
0x77	0x78	0x79	0x7a	registre (r=0)
0x7a	0x77	0x78	0x79	registre (r=1)
0x79	0x7a	0x77	0x78	registre (r=2)
0x78	0x79	0x7a	0x77	registre (r=3)

La **directive d'alignement** de tabcar est-elle nécessaire ou inutile (justifier pourquoi) ?

Puisque tabcar est accédé comme un entier de 32 bits avec une instruction ldr au lieu de 4 instructions ldrsb, il est impératif que l'adresse utilisée par ldr et str respecte la règle d'alignement.

Dans la variante non optimisée, tabcar est traité comme un ensemble d'octets accédés séparément chacun par une instruction ldrsb ou strb et la directive d'alignement devient inutile.

Proposer une traduction optimisée de permuter pour machine big endian.

L'ordre des octets étant inversé, on peut reprendre le code de l'optimisation pour little endian, mais avec une rotation à gauche (que le jeu d'instructions ARM ne fournit pas).

On peut remarquer qu'une rotation à gauche de b bits donne le même résultat qu'une rotation à droite que (32-b) bits (imaginer une rotation de 7 bits à gauche de l'entier auquel est appliqué ROR #25 et comparer). → Insérer une instruction rsb r0,r0, #32 avant la rotation

3.3 Décalages (question bonus s'il vous reste du temps) (10 mn)

En C, les booléens sont représentés sous forme d'entiers. Les opérateurs de comparaison retournent l'entier 1 si la condition est vraie et l'entier 0 si la condition est fausse. On veut traduire l'affectation dans le fragment de code suivant :

```
register int x;                 // a stocker dans r0
register unsigned int x_neg;    // a stocker dans r1
x_neg = (x<0);                  // x_neg = 1 pour x<0
                                // x_neg = 0 pour x>=0
```

Grçace aux opérateurs de décalage, il est possible de réaliser l'équivalent de cette affectation sans utiliser d'opérateur de comparaison. Ceci suppose de connaître la méthode de représentation en binaire des entiers relatifs.

Expliquer brièvement le principe utilisé et écrire le code correspondant (en C et en assembleur).

Dans la représentation des entiers relatifs par la méthode du complément à deux, le bit de poids fort représente le signe de l'entier (0 : entier ≥ 0, 1 : *entier* < 0). Il suffit donc de décaler ce bit de signe de n-1 bits à droite en injectant des 0 en poids forts.

```
#define NB_BITS_MOINS_UN (8*sizeof(x_neg) -1)

// la conversion en unsigned int indique au compilateur de
// décaler x à droite comme s'il était déclaré unsigned
//   --> générer un décalage logique et non arithmétique
x_neg = ((unsigned int) x) >>  NB_BITS_MOINS_UN;
```

```
        @ en supposant que x_neg soit stocké dans r0
        .text
        mov  r0, r0, LSR #31
```

4 Commentaire sur les erreurs commises dans les copies

4.1 Variables et pointeurs

- Les variables sans initialisation vont normalement dans bss
- Un pointeur de quoi que ce soit contient une adresse (32 bits en ARM) : donc on réserve avec .word s'il est initialisé ou avec .skip 4
- Devant la réservation de place pour une variable de type short (.short valeur ou .skip 2) précédée d'une variable de taille 1 octet, il faut un .balign 2 (.balign 4 devant un .word).
- La syntaxe est soit .short valeur, .word valeur ou .byte valeur (uniquement dans data pour $valeur \neq 0$, soit .skip nombre_octets (2 ou 4 ou 1)
- Si y est en mémoire, y = v signifie *&y = v, donc mettre l'adresse y dans un registre reg_ad, v dans un registre reg_don (si elle n'y est pas déjà), puis ranger par strh reg_v, [reg_ad]. En particulier la séquence ldr r0,=y ; mov r0,r1 et l'instruction mov r0,r1 ont le même effet et ne font aucun accès mémoire à l'emplacement de stockage de y.
- Un unsigned char est de taille 1 octet : réservation .skip 1 ou .byte valeur, et accès par ldrb et strb.
- La séquence ldr r0,= ptr ; ldr r1,[r0] met dans r0 l'adresse de la variable répérée par ptr. Il faut ajouter ldrh r0,[r0] pour récupérer dans r0 le contenu de cette variable repérée par ptr.
- Pour faire y = y + 1, il faut copier la valeur de y de la mémoire dans un registre, ajouter 1 à ce registre et recopier le contenu de ce registre en mémoire. Mais pour faire seulement y = 3 ; il n'y a aucun raison de lire la valeur actuelle de y pour l'écraser ensuite : il suffit d'écrire la nouvelle valeur sans lire l'ancienne.

4.2 Tableaux

- Boucles avec test en tête : il faut inverser la condition pour brancher à la fin et à la fin du corps ne pas oublier de revenir au test.
- Boucles avec test à la fin : ne pas oublier le saut au test au début, on prend la condition non inversée dans le test.
- Variables indice de type unsigned : on utilise blo ou bhs et non bl ou bge.
- Les éléments sont des char : ldrsb et strb
- On utilise le nom du tableau pour l'accès : le nom du tableau est la constante adresse du premier élément donc affectation par
  ldr reg_ad,= tabcar ; ldrsb/strb reg_val, [reg_ad]. Idem pour un accès via un pointeur stocké dans un registre. Si on accède via un pointeur stocké en mémoire : ldr reg_ad,= ptr ; ldr reg_ad, [reg_ad] ; lsrsb/strb reg_val, [reg_ad]
- Voir exemple_branchements dans le placard pour la traduction des boucles imbriquées : le corps de la boucle externe inclut toute la traduction de la boucle interne.