M2P CCI : examen Langage Machine, Novembre 2010

Deux heures, tous documents et calculatrices autorisés. Ordinateurs (PC) interdits.

Table des matières

1 Variables et pointeurs (20mn)

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

```
unsigned char c1;
unsigned short y;

unsigned char c2='a';
unsigned short *ptr = &x;
unsigned short  x = 4 ;

   y = *ptr;
   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"). **Comment corriger** le problème ?

Un étudiant (mais est-il bon ?) 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**.

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

Les optimisations abordées dans cette question reposent sur les notions de décalage, de rotation et de convention little/big endian, rappelées dans les annexes en fin de sujet.

On suppose que l'on travaille sur une machine ARM little endian qui représente les entiers relatifs selon la méthdoe du complément à deux. Les entiers de type int sont codés sur 32 bits. On considère un tableau de caractères sur lequel effectuer des permutations d'éléments.

```
unsigned short rotation = 3;
char tabcar[4] = {'w','x','y','z'};

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
}

void main (int argc, char *argv[] )
{
  sscanf (argv[1],"%hu",&rotation);
  afficher ();
  permuter (rotation);
  afficher ();
}
```

```
hopper> permuter 1      hopper> permuter 2      hopper> permuter 3
tab[0] = w              tab[0] = w              tab[0] = w
tab[1] = x              tab[1] = x              tab[1] = x
tab[2] = y              tab[2] = y              tab[2] = y
tab[3] = z              tab[3] = z              tab[3] = z

tab[0] = x              tab[0] = y              tab[0] = z
tab[1] = y              tab[1] = z              tab[1] = w
tab[2] = z              tab[2] = w              tab[2] = x
tab[3] = w              tab[3] = x              tab[3] = y
```

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.

3.2 Rotation (25mn)

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**.

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

Proposer une traduction optimisée de permuter pour machine big endian.

3.3 Décalages (15 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).

4 Annexes

4.1 Big et Little Endian

Soit un entier 32 bits stocké en mémoire à l'adresse 4X : chaque octet contient 8 bits de l'entier. Il existe deux méthodes de rangement de ces octets en mémoire :

- big endian [1] : les octets d'adresses croissantes contiennent les paquets de 8 bits de poids croissant.
- little endian [2] : les octets d'adresses croissantes contiennent les paquets de 8 bits de poids décroissant.

Exemple : stockage de l'entier 0xabcdef12 à l'adresse 1000.

Adresse d'octet	Contenu Big endian	Contenu Little Endian
1000	0xab	0x12
1001	0xcd	0xef
1002	0xef	0xcd
1003	0x12	0xab

1. parfois traduit "gros boutiste" en français
2. "petit boutiste"

4.2 Rappels sur les décalages et rotations

En C, les opérateurs de décalage sont notés $\ll$ et $\gg$ (la nature du décalage à droite dépend du type de la variable décalée). En langage d'assemblage, les notations sont LSL, LSR et ASR et il existe aussi une rotation à droite notée ROR (mais il n'existe pas de rotation à gauche).

C	Lang. assemblage	Opération	Vers	Type d'entier r0
r2=r0»r1	mov r2, r0, LSR r1	décalage logique	droite	naturel (unsigned int)
	mov r2, r0, ASR r1	décalage arithmétique	droite	relatif (int)
r2=r0«r1	mov r2, r0, LSL r1	décalage logique	gauche	naturel ou relatif
	mov r2, r0, ROR r1	rotation	droite	

Un décalage à gauche de b bits supprime les b bits de poids forts et ajoute (à droite) b bits à 0 en poids faibles.
Un décalage logique à droite de b bits supprime les b bits de poids faibles et ajoute (à gauche) b bits à 0 en poids forts.

Un décalage arithmétique à droite de b bits supprime les b bits de poids faibles et ajoute (à gauche) b bits égaux à l'ancien bit de poids fort (bit de signe).

Une rotation à droite est une variante de décalage à droite dans lequel les bits ajoutés à gauche sont les bits supprimés à droite.

Entier initial	opération	Résultat
$\underleftarrow{101}$ **10001001000110100010101101011**	LSL #3	**10001001000110100010101101011** $\underline{000}$
$\underleftarrow{001}$ **10001001000110100010101101010**		**10001001000110100010101101010** $\underline{000}$
10110001001000110100010101101 $\underrightarrow{011}$	LSR #3	$\underline{000}$ **10110001001000110100010101101**
00110001001000110100010101101 $\underrightarrow{010}$		$\underline{000}$ **00110001001000110100010101101**
10110001001000110100010101101 $\underrightarrow{011}$	ASR #3	$\underline{111}$ **10110001001000110100010101101**
00110001001000110100010101101 $\underrightarrow{010}$		$\underline{000}$ **00110001001000110100010101101**
10110001001000110100010101110 $\underrightarrow{1011}$	ROR #4	$\underrightarrow{1011}$ **10110001001000110100010101110**
10110001 $\underrightarrow{00100011010001010110101 1}$	ROR #25	$\underrightarrow{00100011010001010110101 1}$ **10110001**

TABLE 1 – Les bits conservés et décalés sont indiqués en gras