M2P CCI : corrigé examen Langage Machine, Novembre 2012

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

Table des matières

1 Tableaux et fonctions (90mn)

On considère le programme suivant qui recherche la valeur et l'indice de l'élément minimum d'un tableau :

```
#include <stdio.h>
#include <string.h>

char format1 [13] = "tab[0] = %d\n";
char format2 [24] = "mini = %hu , indice %u\n";
unsigned short tab [9] = {3,6,8,7,2,1,9,0,5};
unsigned short *p = tab;

void chercher_mini (unsigned int n,                // n dans r0
                    unsigned short **ptr_mini)     // ptr_mini dans r1
{
   register unsigned short min;      // a stocker dans r4
   register unsigned short *p_min;   // a stocker dans r5
   register unsigned short *p;   // a stocker dans r6

   min = tab[0];
   p_min = tab;

  for (p=tab+1; p<tab+n;p++)
  {
    if (*p < min)
      {
      min = *p;
      p_min = p;
```

```
      }
  }
  *ptr_mini = p_min;
}

int main (void)
{
   register unsigned short m; // a stocker dans r6
   register unsigned int indice;        // a stocker dans r7

   printf (format1,*p);
   chercher_mini (8,&p);
   m = *p;
   indice = p-tab;
   printf (format2, m,indice);

   return 0;
}
```

La convention d'appel de chercher_mini est la suivante :
- le premier paramètre (n) est passé dans le registre r0.
- le deuxième paramètre (ptr_mini) est passé dans r1.
- L'adresse de retour est passée dans lr.

1.1 Section .data (15mn)

Traduire en langage d'assemblage ARM les déclarations de format1, format2, tab et p.

```
.data
format1:        .asciz  "tab[0] = %d\n"
format2:        .asciz  "mini = %hu , indice = %u\n"

.balign 2
tab:            .half   3
                .half   6
                .half   8
                .half   7
                .half   2
                .half   1
                .half   9
                .half   0
```

```
                .half   5

                .balugn 4
p:              .word   tab
```

L'initialisateur de format1 est une chaîne de douze caractères (\ n représente un seul caractère), celui de tab un ensemble de 9 entiers.

Pourquoi le tableau format1 est-il déclaré de taille 13 alors que tab est bien de taille 9 ?

Une chaîne de n caractères est représentée sur n+1 octets à cause de la marque de fin de chaîne ('\0' ou 0) à prévoir pour délimiter la chaîne. Le problème ne concerne pas un simple tableau d'entiers.

1.2 Chercher_mini (parcours de tableau avec pointeur) (40mn)

Voici un squelette de la traduction de chercher_mini en langage d'assemblage ARM à compléter :

```
                .global chercher_mini
                .text
chercher_mini:
prologue :      stmfd   sp!, {r4-r9}     @ sauver r4 a r9 dans la pile
corps :         ...                      @ traduire ici min = tab[0]
                                         @ jusqu'a *ptr_mini = p_min

epilogue:       sp!,{r4-r9}              @ restaurer r4 a r9
                ...
```

Traduire les instructions **min = tab[0] et p_min = tab**.

Traduire la totalité de la boucle for (if inclus).

Traduire la fin de la fonction.

```
                .global chercher_mini

.text
chercher_mini:  stmfd   sp!, {r4-r8}

                ldr     r7,= tab            @  min = tab[0]
                ldrh   r4, [r7]

                mov     r5,r7               @  p_min = tab
```

```
                add     r6, r7, #2          @  p = tab+1

                add     r7, r7, r0, LSL #1  @tab+n

                b       test                @ goto test
corps:
if:             ldrh    r8, [r6]            @  if (*p >=  min) goto finsi
                cmp     r8,r4
                bhs     finsi

alors:          ldrh   r4,[r6]             @ min = *p
                mov     r5,r6               @ p_min = p;

finsi:          add     r6,r6,#2            @ p++

test:           cmp     r6,r7               @ if (p<tab) goto corps
                blo     corps

                str     r5, [r1]            @ *ptr_mini = p_min

ldmfd   sp!,{r4-r8}
mov     pc,lr
```

1.3 Main (Appel de fonction, arithmétique sur les pointeurs) (20mn)

Traduire en langage d'assemblage l'instruction **chercher_mini (8,&p)**.

```
 .text
mov  r0, #8
ldr  r1,= p
bl  chercher_mini
```

Traduire en langage d'assemblage l'instruction **indice = p-tab**. Attention : si deux pointeurs p1 et p2 repèrent respectivement tab[i] et tab[j], alors l'expression p1-p2 représente i-j.

```
.text
ldr  r5,= tab
ldr  r4,= p
ldr  r4, [r4]
sub  r7,r4,r5
mov  r7, r7, LSR #1
```

1.4 Parcours du tableau par indice (15mn)

Voici une autre version de chercher_mini utilisant une variable de boucle de type indice.

```
void chercher_mini (unsigned int n,                // n dans r0
                    unsigned short **ptr_mini)     // ptr_mini dans r1
{
   register unsigned short min;       // a stocker dans r4
   register unsigned indice_min;      // a stocker dans r5
   register unsigned int i;           // a stocker dans r6

   min = tab[0];

  for (i=0;i<n;i++)
  {
    if (tab[i] < min)
      {
      min = tab[i];
      indice_min = i;
      }
  }
  *ptr_mini = tab + i;
}
```

Répondre aux questions suivantes sans rajouter de variable de boucle de type pointeur :

- **Traduire** l'instruction if (corps du if inclus).
- **Traduire** l'instruction ***ptr_mini = tab + i**.

```
        .text
@ traduction du if
        @ traduction du if
        ldr  r7,= tab           @ r9 <- tab[i]
        mov  r8, r6, LSL #1
        ldrh r9, [r7,r8]
        cmp  r9, min
        bhs   finsi
        mov  r4,r9
        mov  r5, r6
finsi:
@ traduction de *ptr_mini = tab + i
        ldr  r7,=tab                  @ instruction redondante
        add  r8, r7, r6, LSL #1
        str  r8, [r1]
```

2 Fonction mystère (branchements et appels, if) (30mn)

Voici une traduction manuelle d'une fonction lettre dont le prototype est le suivant : **void lettre (char c)**.

```
        .global lettre

        .data
table:  .word  voy_hexa
        .word  cons_hexa
        .word  cons_hexa
        .word  cons_hexa
        .word  voy_hexa
        .word  cons_hexa
        .word  cons
        .word  cons
        .word  voy
        .word  cons
        .word  cons
        .word  cons
        .word  cons
        .word  cons
        .word  voy
        .word  cons
        .word  cons
        .word  cons
        .word  cons
        .word  cons
        .word  voy
        .word  cons
        .word  cons
        .word  cons
        .word  voy
        .word  cons

       .text
m1:    .asciz  "N'est pas une minuscule\n"
m2:    .asciz  "consonne\n"
m3:    .asciz  "voyelle\n"
m4:    .asciz  "consonne hexa\n"
m5:    .asciz  "voyelle hexa\n"

       .global lettre
       .balign  4
```

```
lettre: stmfd  sp!, {r0-r2,lr}

        ldr    lr,= fin

        cmp    r0, #'a'
        blo    pas_min
        cmp    r0, #'z'
        bhi    pas_min

        sub    r1,r0,#'a'
        mov    r1, r1, LSL #2
        ldr    r2,= table
        ldr    pc, [r2,r1]

pas_min:
        ldr  r0,= m1
        b     printf

cons_hexa:
        ldr  r0,= m4
        b   printf

cons:
        ldr  r0,= m2
        b   printf

voy_hexa:
        ldr  r0,= m5
        b   printf

voy:
        ldr  r0,= m3
        b   printf

fin:    ldmfd  sp!, {r0-r2,lr}
        mov    pc,lr

.ltorg
```

Cette fonction affiche un message décrivant une information sur la nature du caractère reçu en paramètre.

Qu'affiche-t-elle pour

1. '0' : N'est pas une minuscule
2. 'Z' : N'est pas une minuscule

3. 'a' : voyelle hexa
4. 'b' : consonne hexa
5. 'e' : voyelle hexa
6. 'g' : consonne
7. 'i' : voyelle

Donner une séquence de code ARM se comportant de la même manière que la fonction lettre et n'utilisant aucun tableau. Les deux méthodes sont-elles comparables en temps d'exécution (justifier brièvement votre conclusion) ?

```
        .text
.global lettre
        .balign  4

lettre: stmfd  sp!, {r0-r2,lr}

        cmp    r0, #'a'
        blo    pas_min
        cmp    r0, #'z'
        bhi    pas_min

        cmp    r0, #'a'
        beq    voy_hexa
        cmp    r0, #'e'
        beq    voy_hexa

        cmp    r0, #'g'
        blo    cons_hexa

        cmp    r0,#'i'
        beq    voy
        cmp    r0,#'o'
        beq    voy
        cmp    r0,#'u'
        beq    voy
        cmp    r0,#'y'
        beq    voy

cons:
        ldr  r0,= m2
        bl   printf
        b    fin

voy:
```

```
        ldr  r0,= m3
        bl   printf
        b    fin

pas_min:
        ldr  r0,= m1
        bl     printf
        b    fin

cons_hexa:
        ldr  r0,= m4
        bl   printf
        b    fin


voy_hexa:
        ldr  r0,= m5
        bl   printf

fin:    ldmfd  sp!, {r0-r2,lr}
        mov    pc,lr

.ltorg
```

Ci-dessous, l'estimation de vitesse de traitement ne tient pas compte des instructions appartenant au prologue et à l'épilogue.

La vitesse de traitement des non minuscules est comparable entre les deux versions : 7 instructions.

La version tableau traite les minuscules à coût constant : 11 instructions. L'autre version demande un temps variable selon la minuscule rencontrée : 8 et 10 instructions pour 'a' et 'e' respectivement et de 13 à 21 instructions pour les autres. Sauf fréquence élévée de 'a' et de 'e', la version tableau exécute en moyenne moins d'instructions.

Expliquer pourquoi

- Printf n'est exécuté qu'une seule fois par appel de lettre alors qu'il n'y a aucun branchement entre deux appels de printf et le branchement à printf s'effectue par b et non par bl.
- Lr doit être sauvé dans le prologue et restauré dans l'épilogue.

Le principe consiste à transmettre à printf l'étiquette fin comme adresse de retour (instruction ldr lr,=fin). De ce fait, il n'y a plus à sauver d'adresse de retour pour le branchement aller vers printf (b au lieu de bl). D'autre part, au retour de printf, l'exécution sautera directement à fin au lieu de continuer avec l'instruction qui suivrait l'appel à printf. Il n'est donc pas nécessaire d'ajouter un branchement vers fin après

chaque appel de printf : le saut à fin est assuré par l'instruction de retour à la fin du corps de printf.

L'appel de printf sauvegarde l'adresse de retour dans lettre dans lr, détruisant ainsi l'adresse de retour de lettre vers main. Comme tous les registres modifiés par le corps de lettre, lr doit être sauvegardé dans le prologue et restauré dans l'épilogue.

Pourrait-on effectuer le branchement à printf sans utiliser de branchement relatif (ni b ni bl) ?

– Si oui, avec quelle instruction ou séquence d'instructions ARM ?

– Si non, expliquer pourquoi ce n'est pas possible.

Oui, avec l'instruction ldr pc,= printf. Il s'agit alors d'un branchement de type absolu, mais celà reste un branchement. Cette instruction doit être précédée d'une affectation de l'adresse de retour dans le registre lr.