M2P CCI : examen Langage Machine, Décembre 2019

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

Avant de répondre aux questions, vous devez **impérativement lire** la convention d'appel des fonctions et les contraintes de stockage à respecter.

Table des matières

1 Convention d'appel et contraintes de stockage (sans question, 5mn)

La convention d'appel applicable à toutes les fonctions à traduire est inspirée de celle de gcc :
— Les quatre premiers paramètres explicites sont stockés dans les registres r0 à r3.
— Les paramètres suivant sont empilés (paramètre de gauche en sommet de pile)
— Le paramètre implicite adresse de retour dans l'appelante est passé dans le registre lr (r14).
— Pour les fonctions, le résultat est retourné à la place du premier argument dans le registre r0.
— L'exécution de la routine appelée préserve les contenus des registres : au retour de la fonction, tous les registres [1] autres que le compteur ordinal pc (et, dans le cas d'une fonction, r0 qui contiendra le résultat) ont un contenu identique à celui d'avant l'appel.

Variables ou paramètres seront stockés en mémoire excepté si :
— l'attribut **register** est présent dans leur déclaration ou
— la convention d'appel stipule que le paramètre est passé dans un registre.

Chaque accès à'une variable en mémoire devrait générer une lecture ou une écriture en mémoire. La lecture peut être omise (de préférence avec un commentaire approprié) si un registre contient une copie à jour (datant de l'affectation la plus récente) du contenu de la variable, mais pas l'écriture en mémoire, à réaliser pour chaque affectation.

1. fp/r11, ip/r12 et sp/r13 inclus

La mémoire pour les informations locales à la procédure sera allouée dynamiquement dans la pile : vous ne pouvez pas utiliser le schéma d'allocation statique dans la section bss présenté dans le chapitre "procédures simples, sans récursion".

Vous pouvez gérer la pile au choix avec sp seul ou avec le couple (fp,sp)

2 Présentation (sans question) du programme de gestion de tableaux (5mn)

Le programme a traduire
— recopie un tableau dans un autre
— détermine au pssage la vakeur du plus grand élément du tableau
— détermine au passage l'adresse du petit élément du tableau

La recopie d'une valeur, avec mise à jour de maximum et pointeur de minimum est réalisée par la procédure traiter_element (fichier traiter_element.c en annexe 8).

Le fichier parcours.c (annexe 7) contient les déclarations des variables globables et tableaux et la routine de parcours des tableaux.

Parcours doit être traduite sans connaître le code de traiter_element et traiter_element pourrait être appelée (dans un autre programme) par une ou plusieurs fonctions autres que parcours. La seule source d'information exploitable sur la fonction appelée ou appelante est la convention d'appel rappelée en début de sujet.

3 Déclaration des variables (15mn)

Traduire les déclarations des variables globales (maximum, ptrmin, orig et dest) de parcours.c.

4 Traiter_element : indiçage de tableau, pointeur de pointeur (40mn)

Traduire en langage d'assemblage ARM la fonction traiter_element.

5 Parcours : appel de fonction, pointeurs (30mn)

Traduire en langage d'assemblage ARM la fonction parcours.

```
            .global maximum
            .global ptrmini
             .global orig
            .global dest
             .global parcours

TAILLE_DEST=20
TAILLE_ORIG=10
```

```
            .bss
maximum:    .skip 2
            .balign 4
ptrmini:    .skip 4
dest:       .skip 2*TAILLE_DEST

            .data
orig:       .hword 3
            .hword -4
            .hword 0
            .hword 7
            .hword -20
            .hword 10
            .hword 12
            .hword -25
            .hword 8
            .hword 2

            .text
            @ r6  : i
            @ r7 : ptr
            @ r8,r9,r10 : tmp1, tmp2, tmp3

parcours:   stmfd sp!,{r6-r10,lr}

            ldr   r7,=orig          @ ptr=orig
            ldrsh r9,[r7]                 @ tmp2 = *ptr
            ldr   r8,=dest                @ tmp1 = dest
            strh  r9,[r8]           @ dest[0] = *ptr
            ldr   r10,= maximum           @ tmp3 = &maximum
            strh  r9,[r10]          @ maximum = *ptr
            ldr   r9,= ptrmini            @ tmp3 = &ptrmini
            str   r8,[r9]           @ ptrmini = dest

            mov   r6,#1             @ i=1
            add   r7,r7,#2          @ ptr = ptr + 1     (*sizeof(int16_t))

            b     condw
corpsw:                         @ traiter_element(maximum,&ptrmini,dest,i,*ptr)
            ldr   r0,=maximum             @ max_de_traiter... = &maximum
            ldr   r1,=ptrmini             @ ptradrmin_de_traiter...=&ptrmini
            ldr   r2,=dest                @ tableau_de_traiter...=dest
            mov   r3,r6                   @ indice_de_traiter...=i
            ldrsh r8,[r7]                 @ val_de_traiter...=*ptr  (empile)
            sub   sp,sp,#4
            strh  r8,[sp]
            bl    traiter_element
            add   sp,sp,#4                @ liberer bloc parametre empile
```

```
                add     r6,r6,#1          @ i=i+1

                add     r7,r7,#2          @ ptr = ptr+1

condw:          cmp     r6,#TAILLE_ORIG   @ while (i<TAILLE8ORIG) {
                blt     corpsw

                ldmfd   sp!,{r6-r10,lr}
                bx      lr

.global traiter_element

TRAITER_TO_VALEUR = 0

@ Convention d'appel :
@       r0 : max  r1 :  ptradrmin   r2 : tableau  r3 : indice
@       sommet de pile : valeur
@ Locaux :
@       v   : r9
@       temporaires : r6, r7, r8

                .text
traiter_element: sub  sp,sp,#4                  @ empiler fp
                 str  fp,[sp]
                 add  fp,sp,#4                  @ fp = ancien sommet de pile
                 stmfd sp!,{r6-r9}              @ empiler temporaires modifiés

corps_traiter:   ldrsh  r9,[fp,#TRAITER_TO_VALEUR]  @ tmp2 = *&valeur

condsi1:         ldrsh  r6,[r0]                 @ tmp1 = *max
                 cmp    r9,r6                   @ if (v > *max) {
                 ble    finsi1

alors1:          strh   r9,[r0]                 @    *max=v
                                                @ }

finsi1:          add    r6,r2,r3,LSL #1         @ tmp1 = &(tableau[indice])
                 strh   r9,[r6]                 @ tableau[indice] = v

condsi2:         ldr    r7,[r1]                 @ tmp2 = *ptradrmin
                 ldrsh  r8,[r7]                 @ tmp3 = **ptradrmin
                 cmp    r9, r8                  @ if (v < **ptradrmin) {
                 bge    finsi2

alors2:          str    r6,[r1]                 @ *ptradrmin = tableau+indice
finsi2:                                         @ }

                 ldmfd  sp!,{r6-r9,fp}
```

```
        bx      lr
```

6 Questions diverses et base 2 (25mn)

6.1 Gestion de variables locales

Expliquer la différence entre l'allocation de mémoire aux variables locales des fonctions est plus simple que l'allocation de mémoire dans le cas général (par exemple pour des éléments de liste chaînée) et pourquoi la première est plus simple à gérer.

Dans quelle section de la mémoire sera alloué le doublet retourné par la fonction creer_doublet ci-dessous ?

Expliquer pourquoi retourner à l'appelante l'adresse d'une variable locale constitue une erreur grave de programmation.

```
// détail des champs de la structure omis
typedef struct _doublet { ... } doublet_t ;

doublet_t *creer_doublet () {
  struct doublet nouveau;
  return &nouveau;
}
```

La mémoire pour les variables locales est allouée lors de l'entrée dans la fonction et libérée dès le retour de la fonction. La durée de vie est donc celle de l'exécution du corps de la fonction et l'ordre des allocations et libération respecte la propriété LIFO des appels et retours de fonction. On utilise donc la pile dont le sommet est décalé à chaque allocation et ramené à sa position précédente lors de chaque retour.

La mémoire pour les variables non locales allouées dynamiquement est allouée et libérée explicitement, et sans ordre LIFO entre les allocations et libération.

Dans ce code, creer_doublet retourne l'adresse de sa vraiable locale sotckée dans la pile. Retourner l'adresse d'une variable locale à l'apellante est une erreur grossière parce que le bloc de mémoire dans la pile alloué pour la variable locale sera libéré et réutilisé lors d'un appel de fonction suivant, détruisant le contenu de la variable locale. L'erreur consiste à retourner une adresse de variable dont la durée de vie est limitée à l'exécution du corps de la fonction.

6.2 Base 2

Expliquer brièvement ce que calcule la fonction mystère ci-dessous et comment.

```
\\ fichier mystere.h
int mystere(int x);

\\fichier mystere.s
        .global mystere       @ exporter mystere vers les autres fichiers
```

```
mystere:                        @ sauvegarde de r5 et r6 dans la pile omise
        movS  r5,r0,LSR #31     @ décalage logique à droite (recopie de 0)
        mvnne r0,r0             @ move not si equal
        add r0,r0,r5
        mov pc,lr               @ restauration de r5 et r6 omise
```

Le décalage met à zéro tous les bits du contenu initial, excepté le bit de poids fort, ramené en position 0. Le résultat est donc 1 si l'entier de départ est négatif, et 0 dans le cas contraire.

Si le paramètre x est négatif, l'instruction move not sera exécutée et mystere retournera le complément à 1 de x (calculé par mvn) plus 1 (r5), soit $-x : \overline{x} + 1$.

Si x est positif ou nul, l'instruction mvn est ignorée et on ajouter 0 (r5) à x, donc le paraamètre x d'origine.

La fonction qui retourne -x si $x < 0$ et x si $x \geq 0$ calcule la vakeur absolue de x.

7 Annexe : procédure parcours et variables globales

```
#include "parcours.h"
#include "traiter.h"

int16_t maximum;
int16_t *ptrmini;

// Dans parcours.h : #define TAILLE_DEST 20 et #define TAILLE_ORIG 10

int16_t orig[TAILLE_ORIG]= {3,-4,0,7,-20,10,12,-25,8,2};
int16_t dest [TAILLE_DEST];

void parcours () {
   register int32_t i;                // utiliser r6
   register int16_t *ptr;             // utiliser r7

   // traiter element orig[0] : maximum = dest[0] = orig[0], ptrmini = &(dest[0])
   ptr = orig;
   dest[0] = *ptr;
   maximum = *ptr;
   ptrmini = dest;

   i = 1;
   ptr = ptr+1;
   // Boucle de parcours
   while (i < TAILLE_ORIG) {
      traiter_element(&maximum, &ptrmini, dest, i, *ptr);
      i=i+1;
      ptr=ptr+1;
   }
```

```
}


              .text
              @ r6  : i
              @ r7 : ptr
              @ r8,r9,r10 : tmp1, tmp2, tmp3

parcours:     @ à compléter
```

8 Annexe : procédure traiter_element

```
#include "traiter.h"

void traiter_element (int16_t *max, int16_t **ptradrmin, int16_t *tableau,
                      int32_t indice, int16_t valeur) {
  register int16_t v;   // utiliser r9
  v = valeur;
  if (v > *max) {
     *max = v;
  }
  tableau [indice] = v;
  if (v < **ptradrmin) {
     *ptradrmin = &(tableau[indice]);
  }
}

@ Convention d'appel :
@      r0 : max  r1 :  ptradrmin   r2 : tableau  r3 : indice
@      sommet de pile : valeur
@ Locaux :
@      v   : r9
@      temporaires : r6, r7, r8

traiter_element:  @ à compléter
```