M2P CCI : examen Langage Machine, Novembre 2018

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)

La convention d'appel applicable à toutes les fonctions à traduire est inspirée de celle de gcc :
- Les paramètres explicites de gauche à droite sont stockés dans les registres r0 à r3.
- 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.

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".

Le recours à la paire de pointeurs (fp,sp) ne se justifie pas dans ce contexte où aucun paramètre n'est passé dans la pile : toute la gestion de pile pourra être effectuée avec le seul registre sp.

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

2 Fonction n'en utilisant pas d'autre : échanger (20mn)

La fonction **échanger** permet d'échanger le contenu de deux variables et en retourne la somme.

Traduire en langage d'assemblage cette fonction, qui pourra être appelée par n'importe quelle fonction (dont, mais pas uniquement, parcours).

```
int32_t echanger (int16_t *gauche, int16_t *droit) {
  register int16_t valg;     // utiliser r4
  register int16_t vald;     // utiliser r5

  valg=*gauche;
  vald=*droit;
  *gauche=vald;
  *droit=valg;

  return valg+vald;
}
```

```
.global echanger

 @ parametres
 @ r0 : g r1 : d
 @
 @ variables locales
 @ r4 : valg
 @ r5 : vald

   .text
echanger:
          @ prologue de sauvegarde des registres modifiés r4 et r5
#ifdef STMFD
           stmfd  sp!,{r4,r5}
#else // STMFD
           sub  sp,sp,#4
           str  r5,[sp]
           sub  sp,sp,#4
           str  r4,[sp]
#endif // STMFD

           ldrsh r4,[r0]       @ valg = *gauche
           ldrsh r5,[r1]       @ vald = *droit
           strh r5,[r0]        @ *gauche = vald
           strh r4,[r1]        @ *droit = valg
           add  r0,r4,r5       @ return valg + vald  (val retour dans r0)

          @ epilogue de estauration des registres modifiés r4 et r5
#ifdef STMFD
           ldmfd  sp!,{r4,r5}
#else // STMFD
```

```
            ldr  r4,[sp]
            add  sp,sp,#4
            ldr  r5,[sp]
            add  sp,sp,#4
#endif // STMFD

            mov  pc,lr
```

3 Fonction en appelant une autre et tableaux : parcours (60mn)

La procédure **parcours** utilise la fonction échanger. Elle :

— stocke dans une variable passée par l'appelante la somme des éléments du tableau reçu en paramètre et

— inverse l'ordre des éléments dans le tableau

```
void parcours (unsigned taille, int16_t t[taille], int32_t *sigma) {
  register unsigned i;      // utiliser r4
  register unsigned borne;  // utiliser r5
  register int32_t s;       // utiliser r6

  s=0;
  borne=taille/2;
  for (i=0; i<borne;i++) {
    s = s + echanger (t+i, t+taille-1-i);
  }
  if (taille%2 !=0) {       // si taille impair
    s = s + t[i];
  }
  *sigma=s;
}
```

Le premier paramètre est le nombre d'éléments et le deuxième l'adresse du tableau. A noter, la fonction pourrait être déclarée de 2 autres manières équivalentes, bien que moins lisibles :

```
void parcours (unsigned taille, int16_t t[], int32_t *sigma) {...}
void parcours (unsigned taille, int16_t *t, int32_t *sigma) {...}
```

Prêter attention aux détails suivant :

1. L'utilisation de registres supplémentaires (par exemple r7 et r8) comme temporaires sera sans doute nécessaire, notament pour l'indiçage du tableau.
2. Echanger et parcours ayant la même convention d'appel, les paramètres reçus par parcours seront remplacés dans les registres par les paramètres passés lors de l'appel de échanger. Il faudra donc dupliquer les paramètres reçus dans la pile.
3. Pour le test de parité, exploiter les propriétés de la représentation en base 2. Il y a plusieurs solutions possibles à base d'opérateurs booléens bit à bit et/ou de décalages : en plus du code ARM, expliquer brièvement le principe de votre solution.
4. Il fortement recommandé de définir des constantes symboliques (NOM_CONSTANTE=valeur) pour nommer la position relative des sauvegardes de registres par rapport au sommet de pile.

Traduire parcours en langage d'assemblage ARM en trois étapes :
— Traduire l'affectation s = s + echanger (t+i t+tailel-1-i)
— Traduire la boucle for en remplaçant le code de l'affectation de s ci-dessus par un rectangle
— Dessiner l'organisation du bloc de mémoire en sommet de pile dans le corps de parcours.

```
@ Suggestion de squelette de parcours.S (à compléter)
                  .text
NB_MOTS=...       @ à compléter
...      = ...    @ ...
SAUVE_R4=...      @ compléter
...      = ...    @ ... etc

parcours:
prologue:         sub  sp,sp,#(4*NB_MOTS)
                  ...
                  str  r4,[sp,#SAUVE_R4]
                  ...

corps:            ...

epilogue          ...
                  ldr  r4,[sp,#SAUVE_R4]
                  ...
                  add  sp,sp,#(4*NB_MOTS)
                  ...
```

Solution :

```
               .global parcours
               .text

NB_REGS=8                        @ dessin du bloc en sommet de pile

                @           |----------------|
TO_LR=0         @   sp ---->|sauvegarde de lr| +0
                @           |----------------|
TO_R8=4         @           |sauvegarde de r8| +4
                @           |----------------|
TO_R7=8         @           |sauvegarde de r7| +8
                @           |----------------|
TO_R6=12        @           |sauvegarde de r6| +12
                @           |----------------|
TO_R5=16        @           |sauvegarde de r5| +16
                @           |----------------|
TO_R4=20        @           |sauvegarde de r4| +20
                @           |----------------|
TO_R1=24        @           |sauvegarde de r1| +24
                @           |----------------|
TO_R0=4         @           |sauvegarde de r0| +28
```

```
                @           |----------------|

parcours:
                sub   sp,sp,#(4*NB_REGS)
                str   lr,[sp,#TO_LR]
                str   r8,[sp,#TO_R8]
                str   r7,[sp,#TO_R7]
                str   r6,[sp,#TO_R6]
                str   r5,[sp,#TO_R5]
                str   r4,[sp,#TO_R4]
                str   r1,[sp,#TO_R1]
                str   r0,[sp,#TO_R0]


@ void parcours (unsigned taille, int16_t t[taille], int32_t *sigma) {
@
@ taille : r0 puis Mem[sp+TO_R0]
@ t :      r1 puis Mem[sp+TO_R1]
@ sigma :  r2 puis Mem[sp+TO_R2]
@
@ r4 : i            @ register unsigned int i;  // utiliser r4
@ r5 : borne        @ register unsigned borne   // utiliser r5
@ r6 : s            @ register int32_t s;       // utiliser r6
@ r7 : temporaire de calcul de t+taille-1 -i
@ r8 : tmp pour calcul de %2 != 0

                mov   r6,#0           @ s=0;

                mov   r5,r0, LSR #1   @ borne = taille/2

debutfor:
initfor:        mov   r4,#0           @ i=0
debutwhile:     b     testwhile       @
                                      @ {
@ ************************************************************************
corps:          ldr   r0,[sp,#TO_R1]  @ g de echanger (r0) = t+i
                add   r0, r4, LSL #1

                ldr   r1,[sp,#TO_R1]  @ d de echanger (r1) = t+taille-1-i
                ldr   r7,[sp,#TO_R0]      @  tmp r7 = taille
                sub   r7,r7,#1            @  tmp r7 = taille -1
                sub   r7,r7,r4            @  tmp r7 = taille -1 -i
                add   r1,r1,r7,LSL #1
                bl    echanger        @ resultat de echanger dans r0
                add   r6,r6,r0        @ s = s + echanger (t+i, t+taille-1-i);
                                      @ }
@ ************************************************************************
majfor:         add   r4,r4,#1        @ i++
```

```
testwhile:      cmp   r4,r5            @ while (i<borne)
                blo   corps               @ fi (i<borne)= goto corps

finfor:
if:             ldr   r8,[sp,#TO_R0]   @ tmp r8 = taille
                                       @ if (taille%2 ==0) goto finsi
                andS  r8,r8,#1         @ ou tst r8,#1
                beq   finsi
                                       @ s = s + t[i];
alors:          ldr   r7,[sp,#TO_R1]       @ r7 = t
                add   r7,r7,r4,LSL #1      @ r7 = t+i
                ldrsh r7,[r7]              @ r7 = t[i]
                add   r6,r6,r7
finsi:

                str   r6,[r2]          @ *sigma=s;


                ldr   lr,[sp,#TO_LR]
                ldr   r8,[sp,#TO_R8]
                ldr   r7,[sp,#TO_R7]
                ldr   r6,[sp,#TO_R6]
                ldr   r5,[sp,#TO_R5]
                ldr   r4,[sp,#TO_R4]
                ldr   r1,[sp,#TO_R1]
                ldr   r0,[sp,#TO_R0]
                add   sp,sp,#(4*NB_REGS)

                mov   pc,lr
```

4 Paramètre pointeur, questions générales (40mn)

4.1 Paramètre pointeur (20mn)

La fonction main suivante contient une boucle d'affichage d'une chaîne de caractères. La chaîne est parcourue avec une variable pointeur pc incrémentée à chaque tour de boucle.

```
char ch[] = "bonjour ";

void incptr (...) { ... }     @ procédure incptr à définir

void main () {
  char *pc;

  pc=ch;
  while (*pc != 0) {
    putchar (*pc);
#ifndef INC_BY_FUNC
    pc++;
```

```
#else
    incptr(...);       // à compléter
#endif
  }
  putchar ('\n');
}
```

Par défaut (macro INC_BY_FUNC non définie), la boucle contient une instuction pc++ pour metre à jour le pointeur.

Le programme est recompilé avec l'option -DINC_BY_FUNC, et c'est l'appel d'une procédure incptr qui effectue à présent la mise à jour de pc, et non l'instruction pc++.

Ecrire en C la procédure fonction incptr et son appel dans main.

```
#include "incptr.h"

void incptr(char **p) {
  *p = *p +1;      // ou (*p)++;
}
```

Main doit passer à incptr l'adresse du pointeur pour que celle-ci puisse le modifier et le type du paramètre de incptr doit donc être char **, l'appel dans main étant :
incptr(&pc) ;

En supposant que le compilateur C ne tienne aucun compte de la présence ou non de l'attribut register dans la déclaration de pc.

- Pourrait-il décider de placer pc dans un registre ?
- Sur quel critère basez-vous votre réponse et celle-ci dépend-elle de l'utilisation de la procédure incptr ?

Oui dans la version avec pc++, non dans la version avec la procédure incptr parce qu'on a besoin d'en prendre l'adresse, ce qui implique qu'il soit en mémoire.

Traduire en langage d'assemblage la procédure incptr.

```
        .global incptr

        .text

incptr: sub  sp,sp,#4
        str  r5,[sp]

        ldr  r5,[r0]
        add r5,r5,#1
        str  r5,[r0]

        ldr  r5,[sp]
        add  sp,sp,#4
        mov  pc,lr
```

4.2 Questions générales (20mn)

Commenter les propositions suivantes. **Expliquer** pour chacune pourquoi elle est vraie, fausse ou partiellement vraie (et dans ce cas à quelle condition) :

1. Une fonction ou procédure doit toujours sauvegarder le registre lr.

   Partiellement vrai : seulement si la fonction contient un appel à une routine : lr est alors réutilisé pour passer l'adresse de retour locale et l'adresse de retour reçue de l'appelante dans lr est écrasée.

2. La mémoire à réserver pour stocker un pointeur est définie par le type du pointeur.

   Faux : tous les pointeurs ont la même taille (celle d'une adresse), quelque soit le type d'objet pointé. Ce dernier définit la taille d'accès à utiliser lorsqu'on applique l'opérateur * sur le pointeur, ou le coefficient multiplicateur à appliquer lorsqu'on ajoute un entier au pointeur.

3. Lorsqu'une directive .short[2] suit une directive .word dans la même section, on doit insérer une directive .balign 2 entre les deux.

   Faux : la directive d'alignement est nécéssaire lorsqu'on réserve de la mémoire pour un objet plus grand que le précédent, la paramètre du balignétant la tail ;le de l'objet qui suit. Lorsque l'objet suivant est plus petit, si le premier est aligné, le deuxième le sera forcément aussi.

4. Une fonction est généralement appelée par une instruction bl, mais on pourrait aussi l'appeler avec une séquence composée uniquement d'instruction mov et ldr (préciser laquelle ou pourquoi ce n'est pas possible).

```
@ fonctionnement equivalent à bl f, mais avec un branchement de type absolu
          ldr  lr,= suite_du_bl  @ ou mov lr,pc  @ pc 2 instr en avance : lr <- suite_bl
          ldr  pc,=f                  ldr pc,=f
suite_bl:
```

2. ou .hword la directive existant sous les 2 noms