M2P CCI : corrigé examen Langage Machine, Novembre 2014

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

Table des matières

1 Introduction et conventions (5mn, pas de question)

Eviter l'instruction de multiplication mul dans les traductions.

Sauf précision contraire, la convention d'appel utilisée est celle de gcc, à savoir (pour le jeu d'instructions ARM v4) :
- Passage des quatre premiers paramètres dans les registres r0 à r3
- Résultat des fonctions dans r0 à la place du premier argument
- Adresse de retour passée dans le registre lr

Les principales questions portent sur la traduction du programme de conversion de chaîne en entier (équivalent de sscanf(chaine,"%d",...), présenté en annexe. On suppose que les paramètres c (char_to_valchiffre) et string (string_to_tabval) ne contiennent que des caractères ASCII (codes<128).

2 Questions sur le programme de conversion (1h20mn)

2.1 char_to_valchiffre : paramètres valeur et résultat (20mn)

Traduire en langage d'assemblage la fonction char_to_valchiffre. La gestion de la pile sera simpifiée : pas de fp et les opérations sur la pile seront décrites par un simples commentaires empiler ou dépiler accompagnés de la liste des registres à copier.

```
.global char_to_valchiffre
.text
```

```
@ Convention d'appel
@     r0 : parametre c, puis resultat de la fonction
@     r1 : parametre intval
@
@ Variables locales :
@     r4 : val
@ Temporaires :
@      r5 : copie de c

char_to_valchiffre:
prologue:
            stmfd  sp!,{r4,r5}  @ empiler(r4,r5)
            mov    r5,r0        @ copie de c

corps:      mov    r0,#0        @ resultat par defaut

            cmp    r5,#'0'      @ if (c<'0') goto epilogue
            blt    epilogue     @ blo fonctionne aussi puisque 0<=c<=127 (ASCII)

            cmp    r5,#'9'      @ if (c>'9') goto epilogue
            bgt    epilogue

            cmp    r1,#0        @ if (intvall == NULL) goto finsi
            beq    finsi

            sub    r4,r5,#'0'   @ *intval = c-'0';
            strh   r4,[r1]

finsi:      mov    r0,#1
epilogue:   ldmfd  sp!,{r4,r5} @ depiler (r4,r5)
            mov    pc,lr


@ Seuls les 8 bits de poids faible de c reçu
@ dans r0 sont significatifs. On peut par
@ precaution forcer les bits de poids forts
@ à 0 dans le prologue de la fonction :

prologue:
            stmfd  sp!,{r4,r5}
            mov    r5,r0,LSL #24
            mov    r5,r5,LSR #24
```

Commentaires :

- Dans plusieurs copies, il n'a pas été tenu compte de la convention d'appel spécifiée dans la partie introduction et conventions.
- Voici ce qu'il fallait comprendre de la déclaration du paramètre intval, qui a été souvent mal interprétée :
  1. le nom du deuxème paramètre est intval
  2. son type est (unsigned short *) : intval est donc une adresse de variable de type entier court
  3. étant le deuxième paramètre, intval est passé dans r1
  4. *intval = expresssion signifie que la valeur de expression doit être écrite en mémoire à l'adresse intval (contenue dans r1).
- La gestion de pile simplifiée ne vous dispensait pas de gérer correctement l'adresse de retour. Char_to_valchiffre n'appelant aucune autre fonction, son corps ne modifie pas lr et il n'est pas utile d'empiler lr pour le sauvegarder (branchement de retour classique par mov pc,lr). On peut utiliser une instruction ldm_{xx} avec pc dans la liste des registres à dépiler, mais celà suppose d'avoir empilé lr correctement dans le prologue.
- Ne pas confondre '5' (synonyme de 0x35, le code ASCII du caractère chiffre cinq) et l'entier 5.

La déclaration de tabval a parfois été lue comme la déclaration d'un paramètre valeur ordinaire de nom *tabval. Ceci témoigne d'une grave incompréhension des pointeurs. Il convient de retravailler sur les notions suivantes :
- le passage de paramètre par référence (algorithmique)
- le passage de paramètre par adresse (programmation et langage)
- les pointeurs, le passage de paramètre résultat (langage machine)

2.2 string_to_tabval : while et arithmétique sur les pointeurs (30mn)

Traduire en langage d'assemblage (lignes commentées trad) :

1. la boucle while
2. l'affectation res= pshort - tabval ;

```
                .global string_to_tabval

@ Convention d'appel
@  r0 : string (string_to_tabval), c(char_to_valchiffre), resultat
@  r1 : tabval (string_to_tabval), intval
@  r4 : ok
@  r5 : pshort
@  r6, r7 : copies de string et tabval

                .text
```

```
string_to_tabval:
                stmfd   sp!, {r1,r4,r5,r6,r7,lr} @ empiler les regs modifies
                                                 @ sauf r0 : le resultat

                mov     r6, r0       @ r6 devient la reference string
                mov     r7, r1       @ r7 devient la reference tabval

                mov     r4, #1       @ ok = 1
                mov     r5, r7       @ pshort = tabval

                b       condw        @ goto condw

corps:          ldrsb   r0, [r6]     @ c = *string   (ldrb convient aussi)
                mov     r1, r5       @ intval = pshort
                bl      char_to_valchiffre  @ appel de la fonction
                movS    r4, r0       @ ok = resultat de l'appel
                                     @ movS met à jour Z et N
                                     @ (économise une instruction cmp r4,#0)
                beq     finsi        @ if (!ok) goto finsi
                add     r6,r6,#1     @ string++
                add     r5,r5,#2     @ pshort++

finsi:
condw:          cmp     r4, #1       @ if (ok>=1) goto corps
                bge     corps

                sub     r0, r5, r7   @ res= pshort-tabval
                mov     r0, r0, LSR #1

                ldmfd   sp!, {r1,r4, r5,r6,r7,pc}  @ depiler tous les regs modifies
```

Remarque : si la condition du while avait été ok != 0, on aurait pu optimiser.

```
            cmp  r4,#0
            bne  finwhile
corps:      ... idem
            bl char_to_valchiffre
            movS  r4,r0   @ ou la séquence mov r4,r0; cmp r4,#0
            beq   finsi
            add  r6,r6,#1
            add  r5,r5,#2
            b    corps
finsi:                    @ si la condition du if est fausse
finwhile:                 @ celle du while, l'est aussi
            sub r0,r5,r7
            mov r0, r0, LSR #1
```

```
mov pc,lr
```

Dans la majorité des copies, la convention d'appel a été mal comprise ou ignorée. Paramètres et résultat passés par les registres signifiait que :

- r0 contient le paramètre string (et non *string : la déclaration const char *string indique que le paramètre reçu sera désigné sous le nom string, et que la nature de ce paramètre est une adresse "d'objet" char) dans le prologue de la fonction.
- même principe pour r1 et le paramètre tabval : r0 et r1 contiennent donc les adresses des tableaux chaîne de caractères à convertir (string) et valeurs entières qu'ils représentent (tabval).
- dans l'épilogue de la fonction, le résultat retourné à l'appelante remplace le premier paramètre dans r0

Certaines traductions gèrent implicitement un passage de paramètres via une structure en mémoire, inspiré du td sur les procédures simples, ce qui revient à traiter les paramètres comme des variables globales stockées en mémoire. En soi, cette interprétation n'implique pas de pénalité substancielle à condition d'en respecter les implications et de rester cohérent :

- même interprétation côté appelante et appelée : l'appelante doit écrire les paramètres en mémoire et l'appelée les y lire.
- ne pas oublier que dans ce cas, comme pour toute variable en mémoire, tabval en C est synonyme de *&tabval et qu'accéder au contenu de tabval se fait en deux temps : charger &tabval (l'étiquette tabval en langage d'assemblage ARM) dans un registre, puis effectuer une lecture à cette adresse pour récupérer le contenu du paramètre : copier tabval dans r5 donne la séquence ldr r8,=tabval ; ldr r4,[r8].

L'arithmétique sur les pointeurs a souvent été ignorée : string et tabval diffèrent par la taille d'élément repéré : sizeof(short) = 2 et sizeof(char) = 1.

- tabval ++ se traduit par l'ajout de 2 et non 1
- la diférence entre pshort et tabval est un nombre d'octets, qui doit être converti en différence d'indices de tableau en le divisant par 2

Rappelons aussi que toutes les adresses (étiquettes et adresses contenues dans de tout type de pointeur) sont sur 32 bits, et que les tailles des accès aux contenus repérés par les pointeurs sont définies par le type de pointeur utilisé.

Ok n'ayant pas l'attribut unsigned, devrait être traité comme un entier relatif dans les conditions de branchement (blt au lieu de blo). En pratique ceci n'est pas important, puisque les valeurs affectées à ok (0 et 1) ne sont ni négatives, ni proches de la limite supérieures des entiers positifs représentables.

2.3 Horner10 : boucle for et tableau (20mn)

Traduire en langage d'assemblage la boucle for de la fonction horner10.
Remarques :

- ne pas oublier de multiplier l'indice i par la taille d'un élément dans le calcul de l'adresse de chiffres[i]
- le décalage à gauche (LSL) de g bits multiplie par 2^g.
- ne pas oublier l'incrémentation de la variable indice de la boucle for.

```
                .global horner10

@ r6 : var locale res
@ r7 : var locale i
@ r0 : parametre nb_chiffres
@ r1 : parametre chiffres
@ r5 : temp1 : (res*10, chiffes[i])
@ r8 : temp2 : i*sizeof(unsigned short)

                .text
@ horner10:     stmfd sp!,{r5-r8}     @ empiler r5 à r8

@ Horner10 n'appelle pas de fontion : sauvegarde de lr inutile

                mov  r6,#0            @ res = 0
                mov  r7,#0            @ i = 0

@ Debut de la traduction
                b    condfor          @ goto condfor

corps:          mov  r8, r7, LSL #1   @ temp2 = i*2
                mov  r5, r6, LSL #3   @ temp2 = res * 8
                add  r6, r5, r6, LSL #1 @ res = temp2 + res *2
                ldrh r5, [r1,r8]      @ temp1 = chiffres[i]
                add  r6, r6, r5       @ res +== chiffres[i]

                add  r7,r7, #1        @ i++

condfor:        cmp  r7,r0            @ if (i<nb_chiffres) goto corps
                blo  corps

@ fin de la traduction
                mov  r0,r6            @ return res
                ldmfd sp!,{r5-r8}     @ depiler r5 à r8
                mov  pc,lr
```

3 Questions diverses (35mn)

3.1 Scanf("%s",...) est dangereux ! : chaînes, pile (20mn)

Rappel : la langage C représente une chaîne de caractères sous la forme d'un tableau de char et un octet à 0 sert de marque de fin de chaîne.

Le programme suivant est censé écrire dans un fichier ma_trace une chaîne de caractères lue au clavier.

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

#define MAX 10  // longueur max de la chaine de l'utilisateur
#define L 200   // Longueur max du nom de fichier de trace

FILE *f;

char chaine_u[MAX+1]="";
char nom_trace [L+1]="ma_trace";

void lire (void)
{
  printf ("Entrer une chaîne de (au maximum)  %d caracteres\n",MAX);
  scanf ("%s",chaine_u);
}

// Demander a l'utilisateur de saisir une chaine
// d'au maximum L caracteres puis l'écrire dans
// un fichier de nom ma_trace.

int main (void)
{
  lire ();
  f = fopen (nom_trace,"w");
  if (f != NULL)
    fprintf (f,"%s\n",chaine_u);
  return 0;
}
```

Traduire en langage d'assemblage les déclarations de chaine_u et nom_trace et **expliquer** le +1 dans la dimension de chaine_u.

Il faut prévoir un octet de plus que le nombre maximal de caractères dans la chaîne pour stocker l'octet 0 marqueur de fin de chaîne. Ainsi la chaîne vide ("") occupe un

octet contenant 0 (ou synonyme '\0').

La traduction de la déclaration comprend deux parties : des directives de réservation d'octets avec valeurs initiales pour la chaîne initialisatrice et sa marque de fin de chaîne et une directive .skip pour réserver des octets sans valeur initiale pour le reste de la taille des tableaux.

```
              .global chaine_u
              .global nom_trace
MAX=10
L=200
              .data
chaine_u:     .byte 0
              .skip MAX

nom_trace:    .byte  'm'    @ l'ensemble des .byte '...' peuvent etre
              .byte  'a'    @ remplace par :
              .byte  '_'    @ .asciz "ma_trace"
              .byte  't'
              .byte  'r'
              .byte  'a'
              .byte  'c'
              .byte  'e'
              .byte  0
              .skip   192   @ .skip L+1-9
\end[verbatim}
\vspace{1ex}

Voici deux exécutions de ce programme. La commande ls | wc -l permet de
connaître le nombre de fichiers du répertoire courant.

\begin{verbatim}
turing> ls
turing> simgcc -Wall ../overflow.c -o overflow
turing> ls | wc -l
1
turing> armrun overflow
Entrer une chaîne de (au maximum)  10 caracteres
cas_normal
turing> ls
ma_trace  overflow
turing> ls | wc -l
2
turing> cat ma_trace
```

```
cas_normal
turing> armrun overflow
Entrer une chaîne de (au maximum)  10 caracteres
Ne_jamais!!_faire_confiance_a_l'utilisateur_qui_saisit_une_chaine
turing> cat ma_trace
cas_normal
turing> ls | wc -l
3
```

La deuxième exécution a créé un troisième fichier sans mettre à jour le contenu du fichier ma_trace :

1. **Expliquer** pourquoi
2. **Donner** le nom du nouveau fichier créé ainsi que son contenu

La taille du tableau n'étant pas vérifiée, une chaîne trop longue va déborder du tableau et modifier les variables déclarées juste après le tableau et dans la même section. Les deux tableaux étant initialisés dans leur déclaration, ils seront tous les deux stockés dans la section data. L'adresse de nom_trace sera l'adresse du dernier élément de chaine_u, plus un.

On retrouve donc les 10 premiers caractères de la chaîne trop longue dans le tableau chaine_u (sans marqueur de fin de chaîne) et le reste de la chaîne (avec le 0 marqueur de fin de chaîne) déborde dans le début du tableau nom_trace.

Printf %s va donc parcourir et afficher tous les caractères de chaine_u, puis continuer avec ceux dans nom_trace jusqu'à rencontrer le 0 marque de fin de chaîne (ou terminer l'exécution en erreur si la longueur de la chaîne saisie fait sortir des sections data+bss).

Le contenu déposé dans le fichier créé sera donc la totalité de la chaîne saisie. Le nom du fichier créé est donné par nom_trace, qui contiendra donc la chaîne saisie privée des 11 premiers caractères qui seront dans chaine_u.

```
turing> ls
faire_confiance_a_l_utilisateur_qui_saisit_une_chaine  ma_trace  overflow
turing> cat faire_confiance_a_l_utilisateur_qui_saisit_une_chaine
Ne_jamais!!_faire_confiance_a_l_utilisateur_qui_saisit_une_chaine
```

La question suivante est plus difficile (omettre si vous manquez de temps).

Considérons le scénario suivant :

1. main appelle la fonction f
2. f appelle scanf pour remplir un tableau chaîne qui est une variable locale de f

En fournissant une chaîne de caractères adaptée au code de f, un utilisateur peut théoriquement faire exécuter n'importe quelle séquence de code du programme au lieu de la suite du corps de main après l'appel de f.

En se basant sur le principe de gestion des fonctions avec la pile, **expliquer** le principe de réalisation de ce "tour de magie".

Le prologue de la fonction va sauvegarder les registres modifiés dans la pile, puis allouer de la mémoire dans la pile pour les variables locales, dont le tableau. Dans l'ordre des adresses en mémoire, celle du tableau va précéder celle des sauvegardes de registres.

La fin de la chaîne qui déborde du tableau va donc écraser les anciennes valeurs des registres sauvegdardés, dont lr. Modifier le contenu de la sauvegarde de lr revient à modifier l'adresse destination du branchement de retour en fin d'épilogue de f. L'exécution ne se poursuivra donc pas dans la suite du corps de l'appelante, mais à une adresse formée par la concaténation des codes ASCII des caractères de la chaîne qui ont remplacé l'ancienne valeur de lr.

On peut ainsi choisir d'aller exécuter toute fonction d'entrée/sortie X dont chaque octet de l'adresse est un code ASCII valide. Si f empile r0 à r3, le même mécanisme permet de choisir également la valeur des 4 premiers paramètres à passer à la fonction X.

3.2 Mémoire, Big/little endian (15mn)

Un entier codé sur 32 bits stocké en mémoire à une adresse 4X occupe quatre octets d'adresses consécutives : 4X, 4X+1, 4X+2 et 4X+3. Il existe deux conventions possibles ou "endianness" [1] :

1. Big endian : le poids des octets décroit avec les adresses (l'octet d'adresse 4X contient les bits de poids forts de l'entier).
2. Litte endian : le poids des octets croit avec les adresses (l'octet d'adresse 4X contient les bits de poids faibles de l'entier).

Exemple : stockage de 0x12345678 à l'adresse 0x10000				
Adresses des octets	0x10000	0x10001	0x10002	0x10003
Contenus Big endian	0x12	0x34	0x56	0x78
Contenus Little endian	0x78	0x56	0x34	0x12

La primitive de communication de base d'un réseau informatique est le transfert d'un tableau d'octets entre deux machines ("d'endianness" identiques ou pas).

```
void sw16 (unsigned short *val, unsigned short *res)
```

1. Quelquefois francisées en "gros boutisme ou petit boutisme".

```
{
  register unsigned char *pv;
  register unsigned char *pr;
  // l'adresse val est copiée sans modification dans pv
  // le forceur (unsigned char *) permet d'ignorer la différence de type
  // entre pv et val
  pv=(unsigned char *) val;
  pr=(unsigned char *) res;
  *(pr+1)= *pv;
  *pr    = *(pv+1);
}
```

Expliquer l'utilité de la procédure sw16 pour copier un tableau d'entiers courts entre deux machines.

Sw16 recopie un entier 16 bits d'une variable de type entier court vers une autre, en permuttant les deux octets de contenu. Ceci permet d'adopter une convention unique de représentation des entiers pour l'ensemble d'un réseau. Une machine utilisant la convention opposée à celle du réseau utilisera sw16 pour émettre ou recevoir dans le bon ordre les octets de contenu d'entiers courts.

Ecrire en langage d'assemblage une version de sw16 qui utilise des opérations de décalages au lieu de pointeurs.

```
//  En C :
    unsigned short v1,v2;
//  extraire octet de poids fort et le mettre en poids faible
    v1 = *val >> 8;          // décalage logique à droite
//  extraire octet de poids faible et le mettre en poids fort
    v2 = *val << 8);         // décalage logique à gauche
//  combiner les deux avec un ou bit à bit (ou avec une addition)
    *res = v1 | v2;          // ou bit à bit

@ En langage d'assemblage ARM :
@ r4 : v1   r5 : v2
    stmfd sp!,{r4,r5}
    ldrh  r5, [r0]
    mov   r4,r5,LSR #8;
    mov   r4,r4,LSL #8
    orr   r5,r5,r4
    strh  r5,[r1]
```

Ecrire en langage d'assemblage le corps d'une procédure similaire sw32 pour entiers 32 bits (vous pouvez omettre le prologue et l'épilogue de sauvegarde/restauration des registres modifiés).

```
@ avec des pointeurs

void sw32 (unsigend long *val, unsigned long *res)
{
  register unsigned char *pv;     @ stmfd sp!, {r4-r6}
  register unsigned char *pr;

  pv = (unsigned char *) val;     @   mov r4,r0
  pr = (unsigned char *) res;     @   mov r5,r1

  *pr     = *(pv+3)               @   ldrb r6,[r4,#3]; strb r6,[r5]
  *(pr+1) = *(pv+2);              @   ldrb r6,[r4,#2]; strb r6,[r5,#1]
  *(pr+2) = *(pv+1)               @   ldrb r6,[r4,#1]; strb r6,[r5,#2]
  *(pr+3) = *pv;                  @   ldrb r6,[r4; strb r6,[r5,#3]

                                  @     ldmfd sp!, {r4-r6}
}                                 @     mov  pc,lr
```