M2P CCI : corrigé Langage Machine, Novembre 2017

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

Table des matières

Conventions communes à toutes les questions

Toutes les variables locales des fonctions et procédures sont stockées dans des registres (le numéro de registre à utiliser est indiqué en commentaire). Seules les variables déclarées à l'extérieur des fonctions et sans l'attribut register sont stockées en mémoire. Types uint32_t, uint16_t, uint8_t : entiers naturels sur 32, 16 et 8 bits.

1 Endianness (fontions et tableaux)

Dans la traduction des fonctions, respecter l'allocation des registres décrite en annexe.

1.1 Présentation sans question

On considère 3 tableaux d'entiers naturels de tailles 16,32 et 64 bits, représentés selon la convention Big ou Little Endian de la machine (détails en annexe 4).

```
uint64_t t64[2] = { 0x0123456789abcdef, 0xefcdab8967452301 };
uint32_t t32[4] = { 0x01234567, 0x89abcdef, 0x67452301, 0xefcdab89 };
uint16_t t16[8] = { 0x0123, 0x4567, 0x89ab,0xcdef,
                    0x6745, 0x2301, 0xefcd, 0xab89 };
```

Les 3 paramètres de la procdure reverse_endian sont :

1. tab : adresse du tableau

2. indice : numéro de l'élément de tableau à modifier
3. taille (d'élément) : sizeof(type), type ∈ {uin64_t, uint16_t,uint8_t}

Reverse_endian permet de changer la convention de représentation d'un élément de tableau :

— Après reverse_endian(t32,2,4), l'élément t32[2] contiendra 0x01234567
— Après reverse_endian(t16,0,2), l'élément t16[0] contiendra 0x2301

1.2 Traduction de swap_bytes

Traduire en langage d'assemablage ARM le corps (boucle for) de swap_bytes. Le prologue et l'épilogue ne sont pas demandés.

```
            .text

@ void swap_bytes (uint32_t size, uint8_t *address)
@
@ Convention d'appel :   size dans r0, address dans r1, adresse de retour dans lr
@
@ variables locales : tmp1 dans r3, tmp2 dans r4, i dans r5, tm1mi dans r6

            .global swap_bytes
swap_bytes: @ registres modifiés dans le corps : r3,r4,r5
prologue:   sub  sp,sp,#16;           @ variante possible
            str r6,[sp,#12]           @ sub sp,sp,#,4 str r6,[sp]
            str r5,[sp,#8]            @ sub sp,sp,#4, str r5,[sp]
            str r4,[sp,#4]            @ sub sp,sp,#4, str r4,[sp]
            str r3,[sp]               @ sub sp,sp,#4, str r3,[sp]

corps_swap: mov  r5,#0         @ i=0
            b    condfor

corpstq:
initfor:    sub  r6,r0,#1      @ tm1mi = size -1 - i
            sub  r6,r6,r5

corpsfor:   ldrb r3,[r1,r5]    @ tmp1 = address[i];
            ldrb r4,[r1,r6]    @ tmp2 = address[tm1mi];
            strb r4,[r1,r5]    @ address[i] = tmp2;
            strb r3,[r1,r6]    @ address[tm1mi] = tmp1;

majfor:     add  r5,r5,#1      @ i++

condfor:    cmp  r5,r0, LSR #1 @ while(i<size/2)
```

```
            blo  corpstq

                                @ variante
epilogue:   ldr r3,[sp]         @ ldr r3,[sp]; add sp,sp,#4
            ldr r4,[sp,#4]      @ ldr r4,[sp]; add sp,sp,#4
            ldr r5,[sp,#8]      @ ldr r5,[sp]; add sp,sp,#4
            ldr r6,[sp,#12]     @ ldr r6,[sp]; add sp,sp,#4
            add sp,sp,#16

            mov  pc,lr

@ Variaante avec test de la condition avant le corps

corps_swap: mov  r5,#0          @ i=0
confor:     cmp  r5,r0, LSR #1 @ while(i<size/2) {
            bhs  epilogue

corpsfor:   ...                 @ meme code
majfor:     add  r5,r5,#1       @ i++

            b    condfor        @ }
```

1.3 Traduction de reverse_endian

Le code de reverse_endian comprend un prologue (sauvegarde des registres), le corps proprement dit, et un épilogue (restauration des registres et retour).

La pile n'étant utilisée que pour la sauvegarde des registres modifiés dans le corps de la fonction, seul sp sera utilisé (et pas fp).

Réécrire en C le corps de reverse_endian en supprimant tous les opérateurs d'indexation (crochets [ ]).

```
  @ Le forceur (uint8_t *) est nécessaire pour éviter un warning de compilation
  @ ad8 est de type uint8_t *, ad64+indice est (uint64_t *), ad32_t est (uint32_t *)
  @ rappel : l'entier ajouté à un pinteur est multiplié iimplicitement par sizeof(typ
  @ IMPLICITEMENT : il n'y a pas à l'écrire.
  @ On applique la régle &(t[i]) est synonyme de t+i et t[i] synonyme de *(t+i)
  @ addxx+indice est déjà l'adresse : pas de & devant
  @
  if (taille==8) {
    ad8 = (uint8_t *) (ad64+indice);
  } else if (taille==4) {
    ad8 = (uint8_t *) (ad32+indice);
  } else if (taille==2) {
```

```
    ad8 = (uint8_t *) (ad16+indice);
  } else
    return;

  @ variante possible commune aux tailles 2,4 et 8 :
    ad8 = (uint8_t *) address + taille*indice;
```

Traduire en langage d'assemblage l'appel de swap_bytes

Remarque : l'appel ne présente aucune difficulté : il suffit de copier les paramètres dans r0 et r1 d'après la conhvention d'appel avec passage par les registres, et de penser à utiliser bl au lieu de b pour sauver l'adresse de retour.

Traduire le reste du corps

Remarque : 2 points points importants à considérer ;

1. L'opérateur & qui indique qu'il s'agit juste de calculer l'adresse et non de prendre le contenu
2. L'arithmétique sur les pointeurs prend en compte le type d'objet pointé : l'entier ajouté est implicitment mutliplié par sizeof(type pointé).

```
@ void reverse_endian (void *tab, uint32_t indice, uint32_t taille)

@
@ Convention d'appel : tab dans r0, indice dans r1, taille dans r2, adresse de retour
@
@ variables locales : ad64 dans r4, ad32 dans r5, ad16 dans r6 ad8 dans r7

@ Parmi les registres à sauvegdarer, le plus important est lr qui contient l'adresse
@ retour, et qui sera modifié lors de l'appel bl swap_bytes

            .global reverse_endian
reverse_endian: @ registres modifiés dans le corps : r3 à r7 et lr
prologue:   sub  sp,sp,#24;
            str lr,[sp,#20]
            str r7,[sp,#16]
            str r6,[sp,#12]
            str r5,[sp,#8]
            str r4,[sp,#4]
            str r3,[sp]

corps_rev:  mov r4,r0        @ ad64=tab
            mov r5,r0        @ ad32=tab
            mov r6,r0        @ ad16=tab
```

```
if1:        cmp r2,#8        @ if (taille !=8) goto if2
            bne if2
alors1:     add r7,r4,r1,LSL #3 @ ad8 = ad64+indice (calcul indice*sizeof(uint64_t))
            b   suite_if
if2:        cmp r2,#4        @ if (taille !=4) goto if3
            bne if3
alors2:     add r7,r5,r1,LSL #2 @ ad8 = ad32+indice (calcul indice*sizeof(uint32_t))
            b   suite_if
if3:        cmp r2,#2        @ if (taille !=2) goto finsi
            bne sortie
alors3:     add r7,r6,r1,LSL #1 @ ad8 = ad16+indice (calcul indice*sizeof(uint16_t))

                             @ swap_bytes(taille,ad8)
suite_if:   mov r0,r2        @  size_de_swap_bytes = taille
            mov r1,r7        @  address_de_swap_bytes = ad8
            bl  swap_bytes

                             @ NB : cet appel de swap_çbytes détruit le contenu de r0
                             @ ce n'est pas grave dans la mesure où les paramètres re
                             @ dans r0 (tab) et r1 (indice) ne sont plus utilisés apr
                             @ l'appel

                             @ Dans le cas contraire, il faudrait prévoir de sauvegar
                             @ pour pouvoir récupérer tab et indice au retour de swap
sortie:
epiogue:    ldr r3,[sp]
            ldr r4,[sp,#4]
            ldr r5,[sp,#8]
            ldr r6,[sp,#12]
            ldr r7,[sp,#16]
            ldr lr,[sp,#20]

            add sp,sp,#20

            mov  pc,lr
```

Ecrire le prologue et l'épilogue :

1. établir au préalable la liste des registres à sauvegarder
2. n'utiliser que les instructions sub, str, et mov.

2 Listes : structures, procédures et pile

La convention d'appel de la fonction listmin (voire annexe **??**) est la suivante :

1. L'adresse de retour est passée dans le registre lr
2. Tous les paramètres explicites sont passés dans la pile, le premier (cl) étant en sommet de pile.

Cette fonction parcourt une liste circulaire donnée en premier paramètre et modifie le pointeur passé en deuxième paramètre pour qu'il piointe sur l'éléement de liste de plus petite valeur.

2.1 Déclaration de variables

Le type doublet_t est défini comme suit :

```
typedef struct doublet {
struct doublet *next;
uint16_t value;
} doublet_t;
```

Le fichier data.c déclare deux listes chaînées circulaires créées statiquement composées respectivement de :

1. 5 éléments d0 à d4
2. un seul élément single_cl

```
/* Extern : type specification only */
/*          no memory allocation    */

extern doublet_t d1,d2,d3,d4,d5;

/* Circular Linked list 5,3,1,2,4 */

doublet_t d0 = {&d2,5};
doublet_t d1 = {&d0,4};
doublet_t d2 = {&d4,3};
doublet_t d3 = {&d1,2};
doublet_t d4 = {&d3,1};

doublet_t *circlist = &d0;

/* Single element circular list */
doublet_t single_cl = {&single_cl,6};

doublet_t *ptrmin;
```

Traduire en langage d'assemblage les déclarations de variables précédentes.

Commentaire : une structure avec une composante pointeur et une composante entière est traitéen exactement comme une variable pointeur et une variable entière

séparées, mais avec une seule étiquette sur la première et une définition de constantes symboliques pour définir la place des composantes par rapport à l'adrsse de début de la structure.

Le type int16_t indiquait clairement qu'il s'agissait d'unnentier codé sur 16 bits : réserver 2 octets initialisés avec .short ou .hword. Un pointeur est une adresse et occupe donc 4 octets, d'où .word et .balign4 puisque il suivra le champ value d'une structure précédente qui est de taille inférieure.

```
                    DOUBLET_T_DELTA_NEXT=0
                    DOUBLET_T_DELTA_VALUE=4

                    .data
                    .balign 4

d0:                 .word   d2
                    .hword  5

                    .balign 4
d1:                 .word   d0
                    .hword  4

                    .balign 4
d2:                 .word   d4
                    .hword  3

                    .balign 4
d3:                 .word   d1
                    .hword  2
                    .balign 4

d4:                 .word   d3
                    .hword  1
                    .balign 4

circlist:           .word d0

single_cl:          .word   single_cl
                    .hword  6

                    .bss
ptrmin:             .skip   4
```

2.2 Appel de listmin

La routine ci-dessus est une version raccourcie du programme principal donné en annexe.

```
void main ()
{
  afficher ();
  listmin(circlist,&ptrmin);
  circlist=&d2;
  afficher ();
}
```

Traduire en code ARM le corps cette fonction main (le prologue et l'épilogue de main ne sont pas demandés).

L'appel de afficher, qui n'a pas de paramètre, ne présente pas de difficulté : il faut jkuste penser à utilsieer bl pour sauvegarder l'adresse de retour dans le registre lr.

La première difficulté pour l'appel de listmin est de déterminer quoi passer dans les paramètres pmin et cl. La variable ptrmin est stockée dans la section bss et on en passe l'adresse : il faut donc récupérer l'étiquette ptrmin, qu'on ne peut charger dans un registre par mov reg,#ptrmin parce que les constantes immédiates des instructions mov et de calcul sont liimtées à 8 bits (utiliser ldr reg,=ptrmin).

L'autre paramètre cl est circlist, une variable stockée en mémoire dans la section data : on pourrait écrire *&circlist. Cest donc le contenu de cette variable qu'il faut passer, et non son adresse. Il faut charger son adresse dans un registre comme pour ptrmin, puis faire une lecture du contenu avec ldr pour récupérer le contenu.

L'autre point important est de respecter la convention d'appel qui décrit où ces paramètres doivent être déposés pour que listmin les trouve. La convention indiquait de les passer dans la pile, circlist au sommet.

Il fallait donc avant le branchement :

1. allouer un mot en sommet de pile pour le paramètre pmin : sp ← sp - 4
2. mettre dedans &ptrmin : Mem[sp] ← étiquette ptrmin
3. allouer un mot en sommet de pile pour le paramètre cl : sp ← sp - 4
4. mettre dedans circlist : Mem[sp] ← Mem[étiquette circlist]

Le dernier paramètre empilé cl sera donc bien en comment de pile lors du branchement. De plus, il ne faut pas oublier de libérer l'espace alloué aux paramètres après le retour de la procédure appelée.

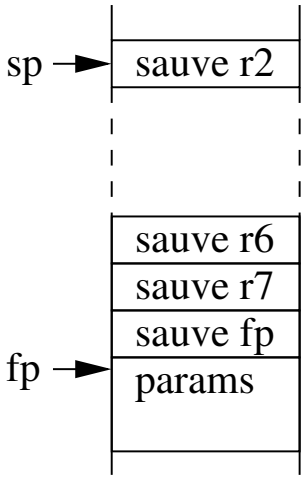
```
main:           stmfd  sp!,{r4-r7,lr}
                bl   afficher              @ afficher ()

                                           @ listmin(circlist,&ptrmin)
                ldr  r4,=ptrmin
                sub  sp,sp,#4; str r4,[sp] @ pmin_de_listmin = &ptrmin

                ldr  r5,=circlist          @ cl_de_listmin = *&circlist
                ldr  r6,[r5]
                sub  sp,sp,#4; str r6,[sp]

                bl   listmin

                add  sp,sp,#8

                bl   afficher              @ afficher ()

                ldr  r5,=circlist          @ circlist = &d2
                ldr  r7,=d2
                str  r7,[r5]

                bl   afficher              @ afficher ()

                ldmfd  sp!,{r4-r7,lr}
                mov  pc,lr
```

Remarque : dans de nombreuses copies, le code correspond à une convention d'appel avec passage de paramètre par les registres analogue à celle utilisée dans la section Endianness au lieu du passage par la pile demandé.

Pour une telle convention qui spécifierait que cl et ptrmin sont passés respectivement dans les registres r0 et r1, il faudrait utiliser ldr r1,=ptrmin et ldr r0,=circlist ; ldr r0,[r0] (r0 et r1 à sauvegarder dans prologue et épilogue de main).

2.3 Listmin

Voici le prologue et l'épilogue de listmin, ainsi qu'un dessin de la pile au début du corps.




```
            .text
@ void listmin (doublet_t *cl, doublet_t **pmin)

listmin:
prologue:   stmfd  sp!,{r2-r7,fp}
            add    fp,sp,#28

corps:      /* non détaillé , dessin de pile ici */

epilogue:   ldmfd  sp!,{r2-r7,fp}
            mov    pc,lr
```

Traduire l'extrait de code C suivant :

Après le prologue, fp à la position où l'appelante avait laissé sp, sur le paramètre cl, donc cl = Mem[fp] et cl->value = Mem[Mem[fp]+DOUBLET_T_DELTA_VALUE]

Sur le même principe, pmin est en Mem[fp+4] et *pmin = cl correspond à Mem[Mem[fp +4]] = cl

```
 valmin = (*c1).value;
*pmin = cl;
```

```
            ldr    r8,[fp]          @ r8 = cl
            ldr    r4,[r8,#DOUBLET_T_DELTA_VALUE]          @ valmin = (*r8).value
            ldr    r3,[fp]          @ *pmin = cl
            str    r2,[r3]
```

Dans l'alternative de convention de passage des paramètres par registres :

```
            ldr    r4,[r0,#DOUBLET_T_DELTA_VALUE]          @ valmin = (*cl).value
            str    r0,[r1]          @ *pmin = cl
```

3 Annexe 1 : code de sawp_bytes et reverse_endian

Pour la traduction en lanagage d'assemblage ARM, la convention d'affectation des variables locales aux registres est imposée (cf tableaux suivant,les déclarations de variables locales).

```
void swap_bytes(uint32_t size,
            uint8_t *address) {

  register uint8_t   tmp1;
  register uint8_t   tmp2;
  register uint32_t     i;
  register uint32_t tm1mi;

/* Si un temporaire supplémentaire
est nécessaire, utiliser r7 */
```

choix regs imposé			
tmp1	tmp2	i	tm1mi
r3	r4	r5	r6

```
  for (i=0; i<size/2; i++) {
    tm1mi = size - 1 - i;
    tmp1 = address[i];
    tmp2 = address[tm1mi];
    address[i] = tmp2;
    address[tm1mi] = tmp1;
  }
}
```

```
void reverse_endian (void *tab,
                     uint32_t indice,
                     uint32_t taille) {

  register uint64_t  *ad64;
  register uint32_t  *ad32;
  register uint16_t  *ad16;
  register uint8_t   *ad8;
```

choix regs imposé			
ad64	ad32	ad16	ad8
r4	r5	r6	r7

```
  ad64=tab;
  ad32=tab;
  ad16=tab;

  if (taille==8) {
    ad8 = (uint8_t *) & (ad64[indice]);
  } else if (taille==4) {
    ad8 = (uint8_t *) & (ad32[indice]);
  } else if (taille==2) {
    ad8 = (uint8_t *) & (ad16[indice]);
  } else
    return;

  swap_bytes(taille,ad8);
}
```

4 Annexe 3 : endianness et contenu des tableaux

Dans la représentation Big Endian, le premier octet de l'entier contient la paire de chiffres hexacimaux de poid forts et le dernier octet ceux de poids faibles. Avec la convention Little Endian, le premier octet contient au contraire les chiffres de poids faibles.

Contenu des premiers octets de t64 si stocké en 0x60000								
Endian	60000	60001	60002	60003	60004	60005	60006	60007
Big	0x01	0x23	0x45	0x67	0x89	0xab	0xcd	0xef
Little	0xef	0xcd	0xab	0x89	0x67	0x45	0x23	0x01
Contenu des premiers octets de t32 si stocké en 0x30000								
Endian	30000	30001	30002	30003	30004	30005	30006	30007
Big	0x01	0x23	0x45	0x67	0x89	0xab	0xcd	0xef
Little	0x67	0x45	0x23	0x01	0xef	0xcd	0xab	0x89
Contenu des premiers octets de t16 si stocké en 0x10000								
Endian	10000	10001	10002	10003	10004	10005	10006	10007
Big	0x01	0x23	0x45	0x67	0x89	0xab	0xcd	0xef
Little	0x23	0x01	0x67	0x45	0xab	0x89	0xef	0xcd