Master CCI

Langage machine

Corrigé du contrôle continu écrit 2017

Durée 1h30, documents autorisés, calculatrices et ordinateurs interdits

Table des matières

Conventions et contraintes

Attention : ne perdez pas de temps à analyser le principe des algorithmes et focalisez-vous sur leur traduction en langage d'assemblage ARM.

L'ordre dans lquel les variables sont déclarées doit être respecté dans la traduction en langage d'assemblage. Les varioables déclarées sans attribut register **doivent** être stockées en mémoire.

Voici les types de variables entières utilisées dans le code des exercices.

types d'entier relatif		taille	types d'entier naturel	
synonyme sur ARM	type	bits	type	synonyme sur ARM
char	int8_t	8	uint8_t	unsigned char
short	int16_t	16	uint16_t	unsigned short
int	int32_t	32	uint32_t	unsigned int

1 Première partie : gestion des variables (45 mn)

1.1 Traduction des déclarations (20mn)

```
uint16_t a = 3;
int32_t x = 5;
uint16_t b = 16;
int32_t y = -2;
int32_t e = 120;
int32_t f = 24;
uint16_t *ptminab;
int32_t *pt32g = &x;
int32_t *pt32d = &y;
int32_t *ptminxy;
```

Traduire en langage d'assemblage les déclarations de variables ci-contre.

A quelle adresse sera stockée la variable y, en supposant que la section data débute à l'adresse 00018fd4 (justifier brièvement vôtre réponse) ?

```
            .data

a:          .short   3    @ a : data + 0   adresse a 0x00018fd4 prochaine 0x00018fd6
            .balign  4    @                0x00018fd6 prochaine 0x00018fd8
x:          .word    5    @ x : data + 4   adresse x 0x00018fd8 prochaine 0x00018fdc


b:          .short  16    @ b : data + 8   adresse b 0x00018fdc prochaine 0x00018fde
            .balign  4    @                         0x00018fde prochaine 0x00018fe0
y:          .word   -2    @ y : data + c   adresse y 0x00018fe0 prochaine 0x00018fe4

e:          .word  120    @                adresse e 0x00018fe4 prochaine 0x00018fe8
f:          .word   24    @                adresse f 0x00018fe8 prochaine 0x00018fec

                          @ Pas de & en ARM, mais x en ARM correspond à &x en C
pt32g:      .word    x    @            adresse pt32g 0x00018fec prochaine 0x00018ff0
                          @            contenu initial 0x00018fd8
pt32d:      .word    y    @            adresse pt32d 0x00018ff0 prochaine 0x00018ff4
                          @            contenu initial 0x00018fe0

            .bss          @ En supposant que bss commence à l'adresse 0x000199c8
ptminab:   .skip    4     @          adresse ptminab 0x000199c8 prochaine 0x000199cc
ptminxy:   .skip    4     @          adresse ptminxy 0x000199cc prochaine 0x000199d0
```

Adresse de y : 0x00018fd4+ c = 0x00018fe0

1.2 Traductions des accès aux variables (25mn)

Voici deux fonctions C qui accèdent à ces variables et un squelette, à compléter, de leur traduction en langage d'assemblage ARM.
Traduire le corps de calcul1 et calcul2, excepté l'appel de printf.
Que contiennent les variables x et y à la fin de l'exécution de main ?

L'affectation *pt32d = *pt32g + 1 stocke dans y x+1, soit 6, x restant inchangée à 5. Puis x=y+6 affecte 12 = 6+6 à x, y restant inchangée à 6.

```
void calcul1 () {
  register int32_t regint32;  // dans registre r4
  register int32_t regvalf;   // dans registre r5
                              // registres r8 et suivants : temporaires
  regint32=e;
  regvalf=f;
  if (regvalf<regint32) {
    regint32=regvalf;
  }
  *pt32d = *pt32g + 1;
  printf ("minimum (e=%d, f=%d) = %d\n",e,f,regint32);
}

void calcul2 () {
   if (a<b) {
     ptminab=&a;
   } else   {
     ptminab=&b;
   }
   x= y + 6;
}

void main () {
  calcul1();
  calcul2();
}
```

```
           .text
           @ registres temporaires : r8 et r8

calcul1:   stmfd    sp!,{r0,r5,r8,r9,lr}
                                @ regint = e = *&e
           ldr      r8,=e       @ r8 = &e = 0x00018fe4
           ldr      r4,[r8]     @ regint32 = *r8 = Mem[r8] = Mem[0x00018fe4] =  120

                                @ regvalf  = f = *&f
           ldr      r8,=f       @ r8= &f = 0x00018fe8
           ldr      r5,[r8]     @ regvalf  = *r8 = mem[r8] = Mem[0x00018fe8] = 24

testif:    cmp      r5,r4       @ if (regvalf >= regint32) goto finsi
           bge      finsi       @    entiers relatifs : bge   (bhs si naturels)
alors:     mov      r4,r5       @   regint32=regvalf;

                                @ pt32g en mémoire : synonyme de *&pt32g
                                @ r9 = *pt32g + 1 = **&t32g + 1
```

```
finsi:      ldr      r8,=pt32g    @ r8 = &pt32g = 0x00018fec  adresse du pointeur
            ldr      r8,[r8]      @ r8 =*r8=*&pt32g=Mem[0x00018fec](= 0x00018fd8)
                                  @ r8 =contenu du pointeur=adresse var pointée (&x)

            ldr      r9,[r8]      @ r9 = *r8 = **&pt32g = Mem[0x00018fd8] (=5)
                                  @ r9 = contenu de la variable pointée par pt32g

            add      r9,r9,#1     @ r9 = **&pt32g+1

                                  @ r8 = pt32d = *&32d

            ldr      r8,=pt32d    @ r8 =&pt32d=0x00018ff0=adresse du pointeur pt32d
            ldr      r8,[r8]      @ r8 = *r8 = *&pt32d = Mem[0x00018ff0] (=0x00018fe0)
                                  @ r8 = contenu du pointeur=adresse var pointée (&y)

                                  @ **&pt32d=*&pt32g+1
            str      r9,[r8]      @ *r8 = r9  : Mem[r8] = r9
                                  @  1ère * tout à gauche du membre gauche = écriture
                                  @ r8 : adresse de var pointée par pt32d
                                  @ r9 : contenu de var pointée par pt32g + 6

callprintf:adr       r0,format
            ldr      r1,=e;ldr  r1,[r1]
            ldr      r2,=f;ldr  r2,[r2]
            mov      r3,r4
            bl       printf

            ldmfd    sp!,{r0,r5,r8,r9,,lr}
            mov      pc,lr

format:     .asciz   "minimum (e=%d, f=%d) = %d\n"

            .balign  4
            @ r0 : &a, r1 copie contenu a, r2: &b, r3 : copie contenu b, r4 : &ptminab
            @ r5 : &x,, r6 : &y, r7 : copie contenu de y

calcul2:    stmfd    sp!,{r0-r5}

                                  @ r1 = a = *&a
            ldr      r0,=a        @ r0 = &a
            ldrh     r1,[r0]      @ r1 =*r0=Mem_16_unsigned[r0]=Mem_16_unsigned[&a]

                                  @ r3 = b = *&b
            ldr      r2,=b        @ r2 = &b
            ldrh     r3,[r2]      @ r3 =*r2=Mem_16_unsigned[r2]=Mem_16_unsigned[&b]
```

```
testif2:    cmp      r1,r3         @ if (a>=b) goto sinon
            bhs      sinon2        @    entiers naturels,  bge si relatifs

                                   @ ptminab en Mem : ptminab = *&ptminab
alors2:     ldr      r4,=ptminab   @ r4 = &ptminab = adresse du pointeur
            str      r0,[r4]       @ *r4 = r0 : *&ptminab = &a
                                   @ adresse de var (&a) devient contenu du pointeur

            b        finsi2        @    goto finsi

sinon2:     ldr      r4,=ptminab   @
            str      r2,[r4]       @ *r4 = r2 : *&ptminab = &b
            b        finsi2        @    goto finsi
finsi2:

                                   @ r7 = y + 6 = *&y + 6

            ldr       r6,=y        @ r6 = &y
            ldr       r7,[r6]      @ r7 = *r6 = *&y = Mem[&y] = (contenu de) y  (= -2)
            add       r7,r7,#6     @ r7 = *&y + 6 = y + 6


            ldr       r5,=x        @ r5 = &x
            str       r7,[r5]      @ *r5 = r7 : x (ou *&x = Mem[&x]) = y + 6

            ldmfd     sp!,{r0-r5}
            mov       pc,lr
```

2 Boucle for (30mn)

La fonction somme calcule itérativement $\sum_{i=0}^{N-1} 2^i$, pour N=8.

```
#define N 8
uint32_t somme ()
{
  register unsigned int sigma;  // utiliser le registre r0
  register unsigned int i;      // utiliser le registre r1
  register unsigned int ajout;  // utiliser le registre r2

  ajout=1;
  sigma=0;
  for(i=0;i<N;i++) {
    sigma = sigma + ajout;
    ajout = ajout * 2;
  }
  return sigma;
}
```

Traduire somme en langage d'assemblage ARM. Donner 2 versions différentes : test de la condition de boucle après, puis avant le corps de la boucle.

```
            .text
            N=8
somme:      stmfd    sp!,{r1-r2}   @ sauvegarde registres

            ... à compléter ...    @ corps de somme à écrire

suite_for:  ldm    sp!,{r1-r2}     @ restauration registres
            mov    pc,lr           @ return sigma (déjà dans r0)

            .text
            N=8
@ Version avec test après

somme:      stmfd    sp!,{r1-r2}

            mov    r2,#1        @ ajout = 1
            mov    r0,#0        @ sigma = 0

            mov    r1,#0        @ i = 0

            b      cond_for    @ goto cond_for
corps_for:  add    r0,r0,r2    @ sigma = sigma + ajout
            mov    r2,r2,LSL #1     @ ajout = ajout *2
corps_for_end:    add    r1,r1,#1    @ i++
cond_for:   cmp    r1,#N        @ if (i<N) goto corps_for
            blo    corps_for

            ldm    sp!,{r1-r2}    @ return sigma (déjà dans r0)
            mov    pc,lr
```

Remarque : b cond_for est nécessaire dans la traduction d'un while(i<N) seul. Dans la traduction de la boucle for, on pourrait l'omettre en tenant compte des deux informations suivantes qui permettent de déduire que le premier test de la condition sera toujours vrai :

— i est initialisé à 0 juste avant la boucle while

— On connaît la constante N et on sait qu'elle est > 0.

```
@ Version avec test avant
somme:      stmfd    sp!,{r1-r2}

            mov    r2,#1        @ ajout = 1
            mov    r0,#0        @ sigma = 0

            mov    r1,#0        @ i = 0
```

```
cond_for:   cmp    r1,#N        @ if (i>=N) goto suite_for
            bhs    suite_for
corps_for:  add    r0,r0,r2    @ sigma = sigma + ajout
            mov    r2,r2,LSL #1     @ ajout = ajout *2
corps_for_end:    add    r1,r1,#1    @ i++
            b      cond_for    @ goto cond_for

suite_for:  ldm    sp!,{r1-r2}    @ return sigma (déjà dans r0)
            mov    pc,lr
```

Quelle serait la traduction en une seule instruction ARM de l'affectation suivante : ajout = ajout *8 ?

mov r2,r2,LSL #3

3 Arithmétique en base 2 (15mn)

On considère une machine fictive travaillant sur 5 bits.

Compléter et interpréter l'addition binaire suivante :

1. Compléter les lignes des retenues et du résultat apparent
2. Indiquer quels bits sont les indicateurs C et N → C=1 et N=0
3. Que vaut l'indicateur V (expliquer brèvement comment vous déterminez sa valeur) ?
4. A partir des indicateurs, déduire si l'opération est correcte en aupposant les entiers de type
   — naturel
   — relatif
5. Donner les valeurs décimales des opérandes et du résutat pour les deux types d'entier

	1	1	0	1	1	a
+	0	1	0	1	0	b
C=1	1	0	1	0	0	retenues
	0	0	1	0	1	résultat apparent
	N=0 ↑					

L'indicateur de débordement pour entiers relatifs V vaut 0, chacun des 2 arguments suivant suffit à lui seul à l'affirmer :
— Les 2 dernières retenues sont égales.
— Les bits de poids fort nous indiquent que nous n'avons pas deux opérandes de même signe et un résultat apparent de type opposé.

C=1 : addition sur entiers naturels fausse

V=0 : addition sur entiers naturels correcte

Addition de naturels : $27 + 10 \rightarrow 5$

Addition de relatifs : $-5 + 10 \rightarrow 5$