Master CCI

Langage machine

Solution du contrôle continu écrit 2014

Durée 1h30, documents autorisés, calculatrices et ordinateurs interdits

Table des matières

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

1 Modulo 3 : constructions algorithmiques (45mn)

On veut calculer le reste de la division par 3 du contenu de la variable entière x, en n'utilisant ni x/3, ni x%3, mais des soustractions.

Traduire en langage d'assemblage ARM les déclarations de variables ainsi que les instructions entre les appels de sscanf et printf (exclus) avec les contraintes suivantes :

1. Bien que l'instruction ARM de division (div) existe mais vous devez essayer de ne pas l'utiliser.
2. L'ordre des déclarations de variables stockées en mémoire doit être respecté.

```
unsigned short resultat;
unsigned int x;

int main (int argc, char *argv[])
{
   register unsigned int delta;          // a stocker dans r1
   register unsigned int etapes;         // a stocker dans r3
   register unsigned short int mod3;     // a stocker dans r2

   sscanf (argv[1]," %x",&x);
   // Traduire en ARM depuis ici ...

   delta = 0x30000000;
```

```
   mod3=x;
   etapes=0;

   while (mod3>=3)
     {
     if (mod3>=delta)
       mod3 = mod3 - delta;
     else
       delta = delta/2;
     etapes++;
     }
   resultat = mod3;
  // ... jusqu'ici

   printf ("mod3=%d calcul en %u etapes\n",mod3,etapes);
   return 0;
}
```

Voici une solution, à un décalage de 3 près des numéros de registres utilisés pour stocker les variables locales de main, de manière à faciliter l'écritures des appels à print et scanf.

```
          .global resultat
          .global x
          .global main

          .bss
                @ si quelquechose précède resultat dans bss
                @ ajouter un .balign 2 avant resultat
resultat: .skip 2

          .balign 4
x:        .skip 4



@ Allocation des registres :
@ r0 : temporaire/format de printf/scanf
@ r1 : temporaire/varaible de printf/scanf
@ r4 : delta
@ r5 : etapes
@ r6 : mod3


          .text
format_scanf:   .asciz "%x"
format_printf:  .asciz "mod3=%d calcul en %u etapes\n"
```

```
                .balign 4
main:           stmfd sp!,{r1-r6,lr}


                ldr  r0,[r1,#4]        @ sscanf (argv[1],"%d",&x)
                ldr  r1,= format_scanf
                ldr  r2,= x
                bl   sscanf

                ldr  r4,= 0x30000000   @ delta=r4,= 0x30000000

                ldr  r0,=x             @ mod3 = x
                ldr  r6,[r0]

                mov  r5,#0             @ etapes = 0

                b    tw                @ goto tw

corps:          cmp  r6,r4             @ if (mod3 < delta) goto sinon
                blo  sinon
                sub  r6,r6,r4          @ mod3 = mod3 -delta
                b    finsi             @ goto finsi
sinon:          mov  r4,r4,LSR #1      @ delta = delta /2
finsi:          add  r5,r5,#1          @ etapes ++

tw:             cmp  r6,#3             @ if (mod3>=3) goto corps
                bhs  corps

                ldr  r0,=resultat      @ resultat = mod3
                strh r6,[r0]

                ldr  r0,=format_printf @ printf ("...",x)
                mov  r1,r6
                mov  r2,r5
                bl   printf

                mov r0,#0;
                ldmfd sp!,{r1-r6,pc}

                .ltorg
```

2 Modulo 3 : variables et pointeurs (20mn)

Voici un autre algorithme (très inefficace) de calcul de modulo 3, basé sur une liste circulaire de structures chaînées entre elles : le champ suivant de c_i repère la structure $c_{(i+1)\%3}$.

```
unsigned int x;
unsigned short mod3;

struct cellule {
  unsigned short  mod;
  struct cellule *suivant;
};

extern struct cellule c0;   // avec extern les declarations ne définissent
extern struct cellule c1;   // que le type des structures et ne reservent
extern struct cellule c2;   // pas de memoire de stockage

struct cellule c0 = {0,&c1};  // vraies declaration avec reservation
struct cellule c1 = {1,&c2};  // de mémoire et valeurs intiales
struct cellule c2 = {2,&c0};


int main (int argc, char *argv[])
{
   register struct cellule *p;   // un pointeur de ce genre de structure

   sscanf (argv[1]," %x",&x);
   printf ("x=0x%8x (%11d)i, mod3(x) = %u\n",x,x,x%3);

   p=&c0;
   while (x>0)
     {
     p = (*p).suivant;
     x--;
     }
   mod3 = (*p).mod;
   printf ("%u\n",mod3);

   return 0;
}
```

Traduire en langage d'assemblage ARM

1. les réservations de place associées aux déclaration des variables x,mod, c0 à c2.
2. l'instruction p=&c0
3. l'instruction p = (*p).suivant
4. l'instruction mod3 = (*p).mod

```
        .global mod3
        .global x
        .global c0
        .global c1
        .global c2
        .global main

        .bss
        @ si quelquehcose precede x dans bss (dans 1 autre fichier)
        @ ajouter .balign 4 avant la reservation de place pour x
x:      .skip 4
mod3:   .skip 2

        .data
        @ si quelquehcose precede c0 dans data (dans 1 autre fichier)
        @ ajouter .balign 2 avant la reservation de place pour c0
c0:     .hword 0
        .balign 4
        .word c1

c1:     .hword 1
        .balign 4
        .word c2

c2:     .hword 2
        .balign 4
        .word c0

@ Allocation des registres :
@ r0 : temporaire/format de printf/scanf
@ r1 : temporaire/varaible de printf/scanf
@ r4 : p

DELTA_MOD = 0
DELTA_SUIVANT = 4

                .text
format_scanf:   .asciz "%x"

                .balign 4
main:           stmfd sp!,{r1-r4,lr}


                ldr  r0,[r1,#4]        @ sscanf (argv[1],"%d",&x)
                ldr  r1,= format_scanf
                ldr  r2,= x
                bl   sscanf
```

```
                @ printf omis : cf solution exercice precedent

                ldr  r4,=c0             @ p = &c0
                b  tw                   @ goto tw

corps:          ldr  r4, [r4,#DELTA_SUIVANT] @ p = (*p).suivant

                ldr  r0,= x             @ x--
                ldr  r1,[r0]
                sub  r1,r1,#1
                str  r1,[r0]

tw:             ldr  r0,=x              @ if (x > 0) goto corps
                ldr  r1,[r0]
                cmp  r1, #0
                bhi  corps

                ldr  r0,[r4,#DELTA_MOD] @ mod3 = (*p).mod
                ldr  r1,= mod3
                strh r0,[r1]

                mov r0,#0;
                ldmfd sp!,{r1-r6,pc}

                .ltorg
```

3 Branchements, adresses et base 2, divers (25mn (sans le bonus))

Un programme C gère trois variables entières sur 32 bits : a, b et niter. Ce programme contient deux constructeurs algorithmiques classiques imbriqués. Voici un squelette de sa traduction en langage d'assemblage ARM :

```
@ Associations variables <-> registres : a <-> r0, b <-> r1, niter <-> r2
@ En cas de besoin de temporaire, utiliser r3

main:       ...                 @ code d'initialisation de a et b omis
            mov   r3,#0
            b     etiq4         @ b est synonyme de bal (branch always)
etiq1:      blo   etiq2
            sub   r1,r1,r2      /*   A    */
            b     etiq3         /*   B    */
etiq2:      sub   r2,r2,r1      /*   C    */
etiq3:      add   r3,r3,#1      /*   D    */
```

```
etiq4:      cmp   r1,r2
            bne   etiq1

            mov   r0,#0         @ ces 2 isntructions mov sont la traduction
            mov   pc,lr         @ de return 0 en fin de main
```

3.1 Traduction inverse de constructeurs algorithmiques

(Question bonus) Quelle(s) séquence(s) de 2 instructions sera ou seront exécutée(s) avant l'instruction blo ? Par quelle instruction seront positionnés les indicateurs Z,N,C,V utilisés dans l'évaluation de la condition lower ?

On ne peut arriver à l'étiquette etiq1 que via l'instruction de branchement bne etiq1, elle-même précédée de l'instruction cmp r1,r2. Ces 2 instructions forment la séquence exécutée avant blo etiq2, et les indicateurs sont positionnés par l'instruction cmp.

(Question bonus) Donner une suite d'instructions C équivalente à ce programme, (d'abord avec, puis sans if ... goto) [1]

```
@ Le commentaire sur les associations registres <-> variables était
@ erroné dans le sujet : c'aurait dû être
@ r1 : a, r2 : b et r3 : niter, r0 : temporaire
@
main:       ...
            mov   r3,#0         @ niter = 0
            b     etiq4         @ goto etiq4
etiq1:      blo   etiq2         @ if (a<b) goto etiq2
            sub   r1,r1,r2      @     a = a - b
            b     etiq3         @ goto etiq3
etiq2:      sub   r2,r2,r1      @     b = b - a
etiq3:      add   r3,r3,#1      @ niter ++

etiq4:      cmp   r1,r2         @ if (a != b) goto etiq1
            bne   etiq1
```

On retrouve une structure si ... alors ... sinon imbriquée dans une boucle tant que avec test après le corps. Une seule comparaison est effectuée parce que les deux consitions comparent les même variables.

```
niter = 0;
while (a != b)
  {
```

1. Toute ressemblance avec un algorithme de calcul connu n'est absolument pas fortuite.

```
if (a >= b)
   a = a - b;
else
   b = b - a;
niter++;
}
```

Il s'agit d'un algorithme classique de calcul de pgcd par soustractions successives, auquel on a ajouté un comptage du nombre de soustractions.

3.2 Notion de branchement et base 2

Quelle est l'instruction de branchement conditionnel à utiliser pour que la condition de saut soit l'inverse de celle testée par blo ?

L'inverse est une condition supérieur ou égal, pour entier naturel tout comme blo, donc bhs.

Ecrire en C la déclaration des variables a et b.

unsigned int a,b ;

Le type de condition utilisé indique que les variables sont des entiers naturels, d'où le unsigned. Si ces variables avaient été stockées en mémoire, le suffixe de ldr/str utilisé pour accéder à leur contenu aurait donné de plus une indication sur leur taille (int,short ou char).

Supposons que l'instruction blo etiq2 soit stockée à l'adresse 0x00100010. **A quelle adresse** hexadécimale correspond l'étiquette etiq2 ?

Etiq2 est à etiq1 +3 instructions, soit 12 (0xc) octets plus loin, d'où 0x0010001c.

L'instruction blo est un branchement relatif. Son paramètre etiq2 sera traduit en un entier relatif Delta inclus dans l'intruction blo. Delta, encodé sur 24 bits suivant la méthode classique du complement à 2, est exprimé en nombre d'instructions dont il faut se déplacer.

Calculer Delta.

De combien (environ) d'instructions au maximum pourrait-on se déplacer vers l'avant avec une instruction telle que b ou blo ?

Delta est l'écart en nombre d'instructions - 2 pour tenir compte du fait que pc est en avance de 2 instructions, d'où Delta=3-2=1.

A une unité et l'avance de pc près, 2^{23}, soit environ 8 millions d'instructions.

Considérez la proposition suivante : on peut substituer à l'instruction b etiq3 une pseudo-instruction ldr (forme ,=) sans modifier le comportement du programme. ?

- Si elle vraie, **écrire** cette pseudo-instruction (avec une courte explication du pourquoi)
- Sinon expliquez **pourquoi c'est impossible**

Il s'agit d'une instruction de saut inconditionnel, donc d'affecter l'adresse destination au compteur ordinal, ce qui peut être réalisé avec une instruction classique de chargement de constantes sur 32 bits : **ldr pc,=etiq3**.

Il manque une directive de section à ce squelette de traduction en langage d'assemblage ARM. **Donner cette directive**

.text

Quelle est la différence de comportement entre les instructions add et addS ?

La version addS met à jour les indicateurs ZNCV dans le registre d'état cpsr. Ceci permet entre autre de tester si le résultat apparent est strictement négatif ou s'il est nul.

Par quelle instruction[2] pourrait-on remplacer l'instruction cmp sans changer le comportement du programme ?

L'isntruction cmp est une soustraction qui ne stocke pas son résultat dans un registre, mais positionne les indicateurs ZNCV. On peut donc utiliser à la place l'instruction **subS r0,r1,r2**. Le résulta peut être stocké dans n'importe quel registre temporaire qui ne contient aucune donnée importante (ici r0).

2. en langage d'assemblage ARM