Master CCI

Langage machine

Solution du contrôle continu écrit 2015

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 Variables et constructions algorithmiques (50mn)

On considère un extrait de code f et sa traduction en langage d'assemblage ARM. Les fichiers f.c et f.s sont tous les deux incomplets. L'objectif est de les reconstituer en entier, chaque partie de code manquant dans l'un étant présente dans l'autre.

Utiliser les registres r0 et r1 pour les variables x et y, et les registres r2 à r4 comme stockage temporaire.

```
/* fichier f.c */
...  a=...;              /* Retrouver à quoi correspondent les ... */
...  b=...;
char c;
...  res;

void f (void)
{
  register ... x;    /* a stocker dans le registre r0 */
  register ... y;    /* a stocker dans le registre r1 */

  x=a;
  y=b;
```

```
  ...
    if (x<y) {
       x=x+a;
    } else {
       y=y+b;
    }
  ...
 res = x;
}

/* fichier f.s */
                    .data
a:                  ....    15     /* retrouver quelle directive de réservation utilis
b:                  ....    12

                    .bss
/* Ajouter ici la déclaration de la variable c */
res:                .skip    2


                    .text
                    @ r0 : x   r1 : y   r2, r3, r4 : temporaires

f:                  stmfd  sp!,{r0-r4}  /* debut standard de fonction */

                    ldr   r3,=a
                    ldrh  r0,[r3]

                    ldr   r4,=b
                    ldrh  r1,[r4]

etq1:               cmp   r0,r1
                    beq   etq2

@ Debut de la traduction de if omise
                    ...
@ Fin   de la traduction de if omise

                    b     etq1
etq2:               ...                @ res = x

                    ldmfd sp!,{r0-r4}   /* fin standard de fonction */
                    mov   pc,lr
```

Déduire (expliquer brièvement le raisonnement) du fichier.s **de quel type**(parmi les 6 du tableau 3) sont les **variables** a,b,res,x et y (type commun aux cinq variables).

L'instruction utilisée pour accéder au contenu de a et b est ldrh. On peut donc en déduire :

1. que la taille des entiers est 16 bits (ce serait ldr ou ldrb pour 32 ou 8 bits)
2. qu'il s'agit d'entiers naturels (pour des relatifs l'instruction utilisée serait ldrsh).

Le fait que les 5 variables soient du même type permettait également de déduire de la traduction de la déclaration de res la taille (mais pas le type naturel/unsigned versus relatif) d'entier. La directive .skip 2 réserve 2 octets, donc res est d'un type entier codé sur 16 bits (int16_t ou uint16_t).

Compléter leurs déclarations dans f.c et f.s.

```
/* Dans f.c */

unsigned short a=15;
unsigned short b=12;
unsigned short res;

/* Dans f.s */
/* Dans .data */
a:       .short 15          @ variante équivalente   a:     .hword  15
b:       .short 12

/* Dans .bss */
res:     .skip  2
c:       .skip  1           @ sizeof(char) = 1
                            @ si c est déclaré avant res, ajouter .balign2 entre
                            @ les deux réservations de mémoire
```

Traduire ensuite la déclaration de la variable c dans la section bss.

Dans la section bss, on ne donne pas de valeur initiale. On utilise la directive skip pour réserver le nombre d'octets correspondant à la taille de l'entier. Un char occupe un octet, d'où .skip 1 pour la déclaration de la variable c.

Voici quelques remarques relatives à certaines réponses vues dans les copies :

1. De nombreuses réponses vont totalement à l'encontre de l'indication que a,b,res,x et y étaient du même type a été largement ignorée, avec des propositions du genre a et b int et res unsigned short ...
2. Une réservation de mémoire avec pour valeur initiale une constante entière négative (par exemple -15) implique une variable de type entier relatif (sans attribut unsigned). Mais le signe + étant facultatif, il n'est pas possible de déterminer si la constante 15 est la valeur initiale d'une variable de type entier naturel (unsigned) ou relatif (sans unsigned).

   Voici trois indices raisonnablement fiables permettant de déduire si la variable entière est de type naturel ou relatif :

(a) Le format (s'il est correctement utilisé) dans les printf et scanf : %u (naturel) ou %d (relatif)

(b) Les comparaisons d'infériorité ou supériorité : blo (naturel) versus blt (relatif). A noter : les branchements condtionnels beq/bz et bne/bnz testent l'indicateur Z, qui est vrai si tous les bits du résultat apparent sont nuls. Il s'appliquent aussi bien aux naturels qu'aux relatifs.

(c) Pour les tailles inférieures au mot, le type d'extension de format à la taille des registres : lrb/ldrh (naturel) versus lrsb/lrsh (relatif)

3. Beaucoup d'entre vous n'ont tenu aucun compte de l'attribut register qui dans le cadre de cette UE spécifie que la variable est à stocker dans un registre et non en mémoire : x et y sont donc les registres r0 et r1 et il n'y a pas de mémoire à réserver pour x et y.
4. La traduction c : .byte 'c' correspond à la déclaration char c = 'c'. Les directives de réservation de place (.byte,.hword,.word,.ascii,.asciz) ne sont utilisables que dans la section data dont le contenu initial est contenu dans le fichier exécutable. En supposant que toute la section bss soit systèmatiquement initialisé à 0 lors du chargement du programme, c : .byte 0 pourrait à la rigueur convenir).

Traduire en langage d'assemblage l'instruction res = x.

```
/* Décomposition :  *&res = x       */

etq2:               ldr   r3,=res      @ res = x
                    strh  r0,[r3]
```

Erreurs fréquement rencontrées :

1. Traduire après décomposition en *&res =x l'étoile tout à gauche du membre gauche par une lecture et une instruction load au lieu d'une écriture par store
2. Intervertir le contenu et adresse : strh r3,[r0] correspond à *x = r3
3. Ne pas tenir compte de la taille du type entier de res et x dans l'instruction str (str ou lieu de strh).

Traduire en langage d'assemblage la construction algorithmique if.

```
                    cmp   r0,r1           @ instruction redondante : déjà exécutée ava
                    bhs   etq3            @ if (x<y) {
etq5:               ldrh  r2,[r3]         @ x = x + *&a
                    add   r0,r0,r2
                    b     etq4            @ } else {

etq3:               ldrh  r2,[r4]         @ y = y + *&b
                    add   r1,r1,r2
                                          @ }
etq4:
                    b     etq1            @ }  ce branchement fait partie de while
```

Erreurs les plus fréquentes sur la structure du if :

1. Oublier l'égalité dans l'inversion de la condition : la négation de $x < y$ est $x \geq y$ et non $x > y$.
2. Il faut soit inverser la condition du if dans la traduction en if ... goto, soit permuter l'ordre des blocs alors et sinon, mais **pas les 2 à la fois** !
3. Oubli de sauter à la fin du if après le premier bloc
4. Branchement conditionnels redondants : après bhs ($\geq$) sinon, la condition inverse lo ($<$) est forcément vraie, ce genre de construction contient un branchement conditionnel de trop :
   cmp r0,r1 blo alors bhs sinon alors :
5. Erreur de moindre importance : ne pas respecter la nature de l'entier dans le choix de la condition : bge au lieu de bhs pour un entier naturel.

Erreur fréquente sur les affectations : dans les affectations, les variables a et b sont en mémoire. Il faut donc ramener une copie de leur contenu de la mémoire dans un registre temporaire, puis additionner ce dernier à la variable x ou y qui est un registre :

```
ldr r3,=a        @ r3 = &a
ldrh r2,[r3]     @ r2 = &r3 = *&a (a)
add  r0,r0,r2    @ x = x + *&a
```

L'affectation de l'adresse de a était déjà présente dans le début de calcul, on pouvait donc omettre la première des 3 instructions, mais pas la deuxième. L'instruction add r0,r0,r3 ajouterait à x non pas une copie du contenu de a, mais l'adresse de l'emplacement auquel a est placée, ce qui est dépourvu de sens [1]

Retrouver la partie de code de f.c qui englobe la construction if. L'objectif est de reconstituer un code C standard dépourvu d'instruction goto.

```
etq1:            cmp  r0,r1              @ if (x==y) goto etq2
                 beq  etq2
                 ...                     @ corps composé du if
                 b    etq1
```

Du fait du branchement conditionnel à etq2, l'exécution ne se poursuit dans le corps (composé uniquement du if) que si x est différent de y. La construction est donc à priori de la forme if (x != y) ou while/do ... while (x != y).

A la fin du corps, on trouve un branchement inconditionnel à l'instruction de comparaison pour réexécuter le corps si la condition est encore vraie. Il s'agit donc d'une boucle, dans une traduction avec test avant le corps en inversant la condition, comme un if sans else avec un branchement de retour au test à la fin du corps. Le corps n'est pas exécuté si la condition est initialement fausse : il s'agit donc de while et non de do

1. Notons qu'une telle opération aurait un sens si r3 contenait l'adresse d'un tableau de char et r0 un indice dans ce tableau : l'addition représenterait alors le calcul de l'adresse de l'élément d'indice r0.

... while.

Il n'y a pas clairement de variable de boucle utilisée dans la condition et incrémentée ou décrémentée à chaque tour de boucle : il n'y a donc pas de conversion de la boucle while en boucle for classique à formuler.

```
  while (x != y) {
    if (x<y) {
       x=x+a;
    } else {
       y=y+b;
    }
```

Remarque : l'explication (même brève) de ce qui fait qu'on reconnait une boucle while brille par son absence dans la quasi-totalité des copies, et la condition du while a été souvent reconstituée à l'envers.

Au cas où vous n'auriez pas reconnu l'algorithme, ce programme calcul le ppmc (plus petit multiple commun) de x et y.

2 Mémoire, base 2 et boucle (40mn)

2.1 Boucle et mémoire(30mn)

On considère la fonction calcul suivante et sa traduction partielle (sans la gestion de la boucle) :

```
unsigned int    x = 0x12345678;

unsigned int calcul (void)
{
  register int i;
  register unsigned int valeur;
  register unsigned int lu;
  register unsigned char *pc;

/* (unsigned char *) ne spécifie aucun accès mémoire                */
/* C'est un forceur de type pour dire au compilateur de considérer  */
/* &x comme si x avait été déclarée de type unsigned char           */

  pc = (unsigned char *) &x;
  valeur = 0;

  i = 3;
  while (i>=0) {
    lu = *(pc+i);
    valeur = lu + (valeur << 8);
```

```
    i--;
  }
  return valeur;
}

                .data
x:              .word    0x12345678

@ r1 : pc   r2: valeur   r3 : lu  r4 : i
@ valeur de retour de calcul dans r0

calcul:         stmfd    sp!, {r1,r4}

                ldr      r1,=x              @ pc = &x
                mov      r2, #0             @valeur = 0

                mov      r4,#3              @ i = 3

                ...                         @ à compléter

corps:          ldrb     r3,[r1,r4]         @ lu = *(pc+i)
                add      r2,r3,r2, LSL #8   @ valeur = lu + valeur << 8
fincorps:       sub      r4,r4,#1           @ i--

                ...                         @ à compléter

                mov      r0,r2              @ return valeur
                ldmfd    sp!, {r1,r4}
                mov      pc,lr

                .ltorg
```

Compléter la traduction de calcul en langage d'assemblage

La traduction classique avec test après le corps sans inversion de la condition ne présente pas de difficulté particulière. La variable i n'ayant pas l'attribut unsigned, il faut utiliser le branchement conditionnel destiné aux entiers relatifs (donc bge et non bhs).

```
calcul:         stmfd    sp!, {r1,r4}

                ldr      r1,=x              @ pc = &x
                mov      r2, #0             @valeur = 0

                mov      r4,#3              @ i = 3
                b        testw
```

```
corps:          ldrb    r3,[r1,r4]        @ lu = *(pc+i)
                add     r2,r3,r2, LSL #8  @ valeur = lu + valeur << 8
                sub     r4,r4,#1          @ i--

testw:          cmp     r4,#0             @ while (i>=0)
                bge     corps

finw:           mov     r0,r2             @ return valeur
                ldmfd   sp!, {r1,r4}
                mov     pc,lr
```

Si l'on préfère tester avant le corps, il faut prendre soin d'inverser la condition d'une part et d'ajouter après le corps un branchement inconditionnel vers le test pour réxécuter éventuellement le corps si la condition est restée vraie.

```
                mov     r4,#3             @ i = 3

testw:          cmp     r4,#0
                blt     finw

corps:          ldrb    r3,[r1,r4]        @ lu = *(pc+i)
                add     r2,r3,r2, LSL #8  @ valeur = lu + valeur << 8
                sub     r4,r4,#1          @ i--

                b       testw

finw:           mov     r0,r2             @ return valeur
```

La valeur retournée par la fonction dans r0 est celle de x : 0x12345678. En **déduire** (avec explications) si la machine est de type big ou little endian (cf définition en annexe).

Le contenu de valeur (r2) est bien identique à la valeur initiale de x dans sa déclaration.

L'octet de poids fort de r2 est le même que celui de x : 0x12. Il s'agit de l'octet d'adresse &x+3 octets, qui est lu en premier et sera décalé trois fois à gauche dans le registre.

L'octet de poids faible de r2 est égal à celui de x : 0x78. Il s'agit du dernier octet lu, d'adresse &x, qui ne sera pas décalé à gauche.

En parcourant les octets de x par ordre croissant des adresses, on rencontre donc des bits de x de poids croissant : poids faible en tête à l'adresse de x, poids fort en dernier à l'adresse de x plus 3 octets. Cet ordre de rangement correspond à la convention de représentation Little Endian.

Avec la convention Big Endian, on trouverait 0x12 à l'adresse de x, et 0x78 à l'adresse de x + 3 octets, et valeur vaudrait 0x78563412.

Proposer une séquence d'instructions ARM n'utilisant aucune variante de ldr et qui charge la constante 0xabcdef12 dans le registre r0.

Il suffit de décomposer la constante en tranches de 8 bits à assembler par décalage :

```
mov   r0,#0xab
mov   r0,r0,LSL #8
add   r0,r0,#0xcd
mov   r0,r0,LSL #8
add   r0,r0,#0xef
mov   r0,r0,LSL #8
add   r0,r0,#0x12
```

Que peut-on affirmer à propos du saut en fin de calcul (expliquer brièvement vos choix) :

1. (a) conditionnel
   (b) inconditionnel
   (c) propriété impossible à déterminer
2. (a) absolu
   (b) relatif
   (c) propriété impossible à déterminer
3. (a) saut en avant
   (b) saut en arrière
   (c) propriété impossible à déterminer

Si l'instruction mov pc,lr était conditionnelle, elle serait suivie d'un signe de condition, par exemple movne pc,lr. Sans suffixe, il s'agit de la condition implicite ALways, toujours vraie, donc d'un branchement inconditionnel.

L'instruction mov est un branchement absolu puisqu'il s'agit d'une affectation de l'adresse fournie dans lr à pc et non de l'addition d'un déplacement au compteur ordinal (dans ce cas la destination serait alors définie relativement à l'instruction courante mov pc,lr).

Il est impossible de déterminer le sens du branchement : l'adresse du branchement dépend de la position de la ou les fonctions qui appelle(nt) calcul, qui dépendra de l'ordre dans lequel les fichiers seront fusionnés lors de l'édition de liens.

2.2 Base 2 (10mn)

Soient r0 et r2 deux variables de type entier naturel et la séquence de code suivante :

```
mvn    r1,r0
add    r1,r1,#1
add    r2,r0,r1
```

Donner la valeur de r2 en fin de séquence pour les valeurs suivantes (écrites en base 10) de r0 : 10, 260, 1040 et 1054632.

La description de l'instruction mvn (move not) page 5 de la documentation arm simplifiée indique que cette instruction copie le **complément à 1** de l'opérande droit (r0) dans le registre destination (r1). L'instruction suivante ajoute 1, ce qui calcule le complément à 2 (autrement dit l'opposé) de l'opérande initialement dans r0.

Rappel : la représentation du complément à 1 $\overline{x}$ de x est obtenue en inversant tous les bits de x ($\overline{x}_i = 1 - x_i$) et on a la propriété $\overline{x} = 2^n - 1 - x$ (n étant la taille de la représetation, exprimée en nombre de bits). Le complément à 2 de x $\overline{x} + 1$ est égal à $2^n - x$, assimilable à -x puisque sur n bits le résultat d'une addition est obtenu à 2^n près.

On pourrait obtenir le même résultat avec la soustraction inversée :

```
rsb   r1, r0, #0
```

La dernière instruction ajoute donc le nombre présent dans r0 et son opposé calculé dans r1. Le résultat apparent est donc 0 quelque soit le nombre r0 de départ. Le vrai résultat serait 2^n, ce qui transparait dans le fait que la dernière retenue de l'addtition sera à 1.

Le choix de grands entiers difficiles à convertir manuellement en binaire était volontaire pour vous dissuader de le faire. L'entier 10 était un exemple facile à traiter et permettait de comprendre le fonctionnement.

```
10   sur 32 bits  0000 0000 0000 0000 0000 0000 0000 1010  = 0x0000000A
/10               1111 1111 1111 1111 1111 1111 1111 0101  = 0xFFFFFFF5 mvn r1,r0
+1                0000 0000 0000 0000 0000 0000 0000 0001  = 0x00000001
--->              1111 1111 1111 1111 1111 1111 1111 0110  = 0xFFFFFFF6 add r1,r1,#1
+10               0000 0000 0000 0000 0000 0000 0000 1010  = 0x0000000A
-->          C=1  0000 0000 0000 0000 0000 0000 0000 0000  = 0x00000000 add r2,r0,r1
```

Remarque : je regrette que dans presque toutes les copies ayant abordé la question, l'instruction mvn (move not) ait été interprétée comme synonyme de mov.

3 Annexe 1 : tableau des types d'entiers de stdint.h

Après utilisation de la directive **#include <stdint.h>**, on peut utiliser les types C d'entiers de taille certaine intxx_t et uintxx_t regroupés dans le tableau ci-dessous, avec leurs synonymes classiques[2] pour un processeur ARM 32 bits.

2. dont la taille effective peut varier d'une machine à l'autre

types d'entier naturel		taille	types d'entier relatif	
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

4 Annexe 2 : conventions little et big endian

Un entier E stocké sur plus de 8 bits en mémoire occupe des octets d'adresses consécutives (A,A+1,A+2 et A+3 pour un entier codé sur 32 bits). L'adresse A de l'entier est celle de son premier octet.

La représentation de l'entier en binaire est donc découpée en paquets de 8 bits stockés chacun dans un octet de mémoire selon un ordre croissant ou décroissant.

Dans la convention Big Endian, l'ordre est décroissant : le premier octet (A) contient les bits de poids forts de l'entier et le dernier octet (A+3) les bits de poids faibles de E.

La convention opposée est dite Little Endian : les bits de poids faibles occupe le premier octet (A) et ceux de poids forts le dernier (A+3).