

# Master CCI

## Langage machine

### Solution du contrôle continu écrit 2012

Durée 1h30, documents autorisés, calculatrices et ordinateurs interdits

## 1 Base2 et accès à la mémoire (50mn)

### 1.1 Précisions et rappels sur les décalages (pas de question)

- En C, l'expression  $a \ll b$  signifie valeur(a) décalée à gauche de valeur(b) bits. Réciproquement  $a \gg b$  signifie valeur(a) décalée à droite de valeur(b) bits. Le décalage est de type arithmétique si l'entier a est de type relatif (int), et logique sinon (unsigned int).
- Lorsque l'opérande droit d'un calcul est un registre, les instructions ARM permettent de lui appliquer une opération de décalage à gauche ou à droite. Le nombre de bits du décalage peut être une constante immédiate ou le contenu d'un registre. Le décalage est appliquée sur la copie du contenu du registre utilisée dans l'instruction de calcul : il ne modifie pas le contenu du registre utilisé.

### 1.2 Accès aux variables en mémoire (35mn)

On veut traduire le programme source1.c en un fichier source1.S (langage d'assemblage ARM) :

```
#include <stdio.h>

unsigned int x = 7;
unsigned int y;
unsigned int z;

int main (void)
{
    scanf("%u",&y);    // verification du code de retour de scanf omise
    printf ("x=%u, y=%u, z=%u\n",x,y,z);

    // traduire en langage d'assemblage ces deux instructions
    x = (y<<3) + y;
    z = x/4;

    printf ("x=%u, y=%u, z=%u\n",x,y,z);
    return 0;
}
```

**Traduire** la déclaration des variables en supposant que la taille d'un int ou unsigned int est un mot de 32 bits.

```
                .data
x:              .word    7

                .bss
y:              .skip    4
z:              .skip    4
```

**Traduire** en langage d'assemblage ARM chacune des deux affectations entre les appels à printf. Traduire chaque affectation indépendamment l'une de l'autre puis indiquer les éventuelles optimisations de code possible lorsque l'on traduit les deux instructions ensemble.

```
ldr    r5,= y
ldr    r3,[r5]
add    r6, r3,r3,LSL #3
ldr    r4,= x
str    r6, [r4]

ldr    r4,= x           @ optimisation : supprimer ce ldr
ldr    r6, [r4]         @ optimisation : supprimer ce ldr
mov    r3, r6, LSR #2   @ optimisation : remplacer r3 par r6
ldr    r5,= z
str    r3, [r5]

.ltorg
```

On considère à présent le programme source2.c :

```
#include <stdio.h>

unsigned char  x = 'a';
short int y;
int z;

int main (void)
{
    scanf("%hd",&y);    // verification du code de retour de scanf omise
    printf ("x=%c, y=%hd\n",x,y);

    // traduire en langage d'assemblage ces deux instructions
    x++;
    z = y*4;
```

```

printf ("x=%c, y=%hd, z=%d\n",x,y,z);
return 0;
}

```

**Traduire** en langage d'assemblage la déclaration des variables et les deux affectations en supposant que `sizeof(short)=1` et `sizeof(int)=4`.

```

        .data
x:      .byte    'a'      @ ou .byte 97

```

```

        .bss
y:      .skip 2

```

```

        .balign 4
z:      .skip 4

```

```

        .text
@ r2 : temporaire adresse
@ r3 : temporaire contenu

```

```
main:
```

```
...
```

```

@ x++
ldr     r3,= x
ldrb    r2,[r3]
add     r2, r2, #1
strb    r2, [r3]

```

```

@ z = y*4
ldr     r3,= y
ldrsh   r2,[r3]
mov     r2, r2, LSL #2
ldr     r3,= z
str     r2,[r3]

```

```
...
```

### 1.3 Base 2 (15mn)

On suppose que la machine représente les entiers relatifs selon la convention habituelle du complément à 2.

Donner en hexadécimal et en décimal le contenu de la variable `x` (de taille 32 bits) à la fin de cette séquence de code (expliquer pourquoi). Quelle est l'opération arithmétique réalisée ?

```
int x;
x = 0xcde12300;
x = x >> 8;
```

X étant de type relatif, on décale à droite en recopiant le bit de signe qui est ici 1 (bit de poids fort du chiffre C en poids fort). On injecte donc 8 bits à 1 à gauche, ce qui donne deux chiffres hexadécimal f, d'où nouvelle valeur de x : 0xffcde123. L'opération est approximativement une division par  $256 = 2^8$  (attention : ce ne serait pas vrai si x n'était pas multiple de 256).

Que peut-on dire des valeurs relatives de x et y lorsque l'indicateur C (retenue finale) est à 1 après calcul de x-y par addition du complément à 2 de y :

- pour x et y de type int ?  
on ne peut rien dire parce que pour des entiers relatifs les indicateurs à prendre en compte sont N (signe du résultat apparent) et V (indicateur de débordement en relatif)
- pour x et y de type unsigned int ?  
les retenues dans l'addition du C2 sont complémentaires des emprunts dans la vraie soustraction. C=1 signifie que le dernier emprunt dans la vraie soustraction aurait été 0 : pas d'emprunt final, d'où soustraction possible et  $x \geq y$ .

## 2 Constructions algorithmiques

On considère le squelette de programme C suivant :

```
#include <stdio.h>

#define N 10

int main (void)
{
    register unsigned int s;    // a stocker dans registre r7
    register unsigned int i;    // a stocker dans registre r6
    register unsigned int j;    // a stocker dans registre r5

    ... // instructions à retrouver

    return 0;
}
```

ainsi que sa traduction en langage d'assemblage ARM :

```
@ s : r7  i : r6  j : r5
```

```
@ Ne vous souviez pas de ce qui est avant debut:
```

```

        .global main
        .text

format:  .asciz  "%d\n"
        .balign 4

main:    stmfd  sp!,{r0-r7,lr}
        ldr     r8,= format

debut:   mov r5,#0
et1:     b      et6
et2:     mov r7,#0
        mov r6,#0
et3:     b      et5
et4 :    add r7,r7,r6
        add r6,r6,#1
et5:     cmp r6,r5
        bls et4

fin1:    @ debut de traduction de printf ("%d\n",j);
        mov r0,r8
        mov r1,r7
        bl printf
        @ fin de traduction de l'appel de printf

debut2:  add r5,r5,#1
et6:     cmp r5,#10
        bls et2

@ Ne vous souciez pas non plus de ces instructions
fin2:    ldmdfd sp!,{r0-r7,lr}
        mov r0,#0
        mov pc,lr

```

Donner une séquence d'instructions C équivalente au code en langage d'assemblage compris entre les étiquettes debut et fin.

```

debut:   mov r5,#0           @ j=0;
et1:     b      et6           @ goto      et6;
et2:     mov r7,#0           @ s=0;

        mov r6,#0           @ i=0;
et3:     b      et5           @ goto      et5
et4 :    add r7,r7,r6         @ s = s+i
        add r6,r6,#1         @ i++

```

```

et5:      cmp    r6,r5                @ if (i <= j) goto et4
          bls    et4

          @ debut de traduction de printf ("%d\n",j);
          mov    r0,r8                @ printf ("%d\n",j);
          mov    r1,r7
          bl     printf
          @ fin de traduction de l'appel de printf

          add    r5,r5,#1             @ j++
et6:      cmp    r5,#10               @ if (j <= 10) goto et2
          bls    et2

fin:      ldmfid sp!,{r0-r7,lr}
          mov    r0,#0
          mov    pc,lr

```

On reconnaît à partir de l'étiquette `et4` une boucle `while(i<=j)`. A partir de l'étiquette `et3`, on reconnaît une boucle `while (j <= 10)`, dont le corps inclut la boucle d'indice `i`.

```

j = 0;
while (j <= N)
{
    s = 0;
    i = 0;
    while (i <= j)
    {
        s = s + i;
        i++;
    }
    printf ("%d\n",s);
    j++;
}

```

On retrouve dans la boucle intérieure un algorithme de calcul de l'élément d'indice `j` de la suite de Fibonacci. La boucle sur `j` calcule `N` éléments de cette suite.

Ecrire en langage d'assemblage un code équivalent dans lequel les branchements conditionnels sont tous des branchements en avant.

```

@ s : r7  i : r6  j : r5

.global main
.text

```

```

format:      .asciz   "%d\n"
              .balign 4

main:        stmfd   sp!,{r0-r7,lr}
              ldr     r8,= format

debut:       mov     r5,#0                @@@ j = 0

testj:       cmp     r5,#10              @@@ while (j <= N)
              bhi     fin                @@@ if (j > N) goto fin

corpsj:      mov     r7,#0                @@@ s = 0
              mov     r6,#0                @@@ i = 0

testi:       cmp     r6,r5                @@@ while (i <= j)
              bhi     suitei             @@@ if (i > j) goto suitei

corpsi:      add     r7,r7,r6              @@@ s = s + i
              add     r6,r6,#1            @@@ i++
              b       testi               @@@ goto testi

suitei:      @ debut de traduction de printf ("%d\n",j);
              mov     r0,r8
              mov     r1,r7
              bl      printf
              @ fin de traduction de l'appel de printf

              add     r5,r5,#1            @@@ j++
              b       testj              @@@ goto testj

fin:         ldmfd   sp!,{r0-r7,lr}
              mov     r0,#0
              mov     pc,lr

```