

Master CCI

Solution du contrôle continu Langage machine 2022

Durée 1h30, documents autorisés, calculatrices et ordinateurs interdits

Les durées servent de barème indicatif

Table des matières

1	Variables en mémoire	2
1.1	Déclarations	2
1.2	Accès aux variables	2
2	Constructions algorithmiques et base 2	3

Conventions et contraintes

L'ordre dans lequel les variables sont déclarées doit être respecté dans la traduction en langage d'assemblage. Les variables déclarées sans attribut register **doivent** être stockées en mémoire (contenu mémoire mis à jour à chaque affectation).

Voici les types de variables entières utilisées dans le code des exercices. Par dérogation, considérez pour la traduction en langage d'assemblage le type char comme synonyme de uint8_t (entier naturel).

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

Les constantes symboliques peuvent être définies par = en langage d'assemblage (exemple : TAILLE=2), après quoi elles peuvent être utilisées comme opérande dans une directive ou une instruction : (exemple : mov r0,#TAILLE).

Les directives de réservation de place acceptent des expressions simples composées d'entiers, de constantes entières symboliques et d'opérateurs arithmétiques de base : +, -, * (.skip 4*TAILLE).

L'instruction machine mul existe (avec une syntaxe analogue à add ou sub), mais vous ne devriez pas avoir à l'utiliser pour des multiplications par de petites constantes

- les petites constantes sont la somme d'un nombre limité de puissances de 2.
- mul n'accepte pas d'opérande droit de type constante immédiate

1 Variables en mémoire

1.1 Déclarations (5points, 20mn)

```
int16_t x16 = 45;
int16_t y16;
char c1 = 'b';
char c2;
int16_t *ptr16 = &x16;
```

Traduire en langage d'assemblage les déclarations précédentes.

```
        .data
x16:    .hword    45            @ uint16_t x16 = 45;
c1:     .byte     'b'          @ char c1 = 'b';
        .balign   4
ptr16:  .word     x16          @ uint16_t *ptr16 = &x16;

        .bss
y16:    .skip     2            @ uint16_t y16;
c2:     .skip     1            @ char c2;
```

1.2 Accès aux variables (8 points 40mn)

```
void calcul () {
    register uint16_t *p16; // à stocker dans le registre r4
    register uint16_t v16;  // à stocker dans le registre r5
    c2 = c1 + 1;
    (*ptr16) ++;            // *ptr16=*ptr16+1
    p16 = ptr16;
    v16 = *p16;
    p16 = &y16;
    *p16 = 2;
```

Traduire individuellement en langage d'assemblage ARM chacune des affectations précédentes.

```
        @ p16 : r4
        @ v16 : r5

ldr     r1,=c2            @ c2 = c1 + 1;
        ldrb     r0,[r1]
        add     r0,r0,#1
        ldr     r1,=c2
        strb     r0,{r1}

ldr     r2,=ptr16         @ *ptr16++
        ldr     r2,[r2]
        ldrsh   r0,[r2]
```

```

add    r0,r0,#1
strh   r0,[r2]

ldr    r1,=ptr16    @ p16 = ptr16
ldr    r4,[r1]

ldrsh  r5,[r4]      @ v16 = *p16

ldr    r4,=y16      @ p16=&y16

mov    r0,#2        @ *p16=2
strh   r0,[r4]

```

2 Constructions algorithmiques et base 2 (30mn)

On considère la fonction calcul suivante :

```

// Le paramètre x est reçu dans le registre r0
int calcul (int32_t x) {
    register int nb=0; // à stocker dans r7
    while (x != 0) {
        if (x < 0) nb++;
        x = x * 2;
    }
    return nb;
}

```

Compléter le code ARM de la fonction ci-dessous (traduire la boucle while).
Donner deux variantes du while : l'une avec test de la condition avant le corps et l'autre avec test de la condition après le corps.

```

@ x stocké dans r0
@ nb stocké dans r7
@ valeur de retour dans r0

```

```

calcul:    stmfd sp!{r1-r7} @ sauvegarder r1 à r7 dans la pile
           mov    r7,#0     @ nb=0
           ...           @ traduction de la boucle while à compléter
           ...           @
suitew:    mov    r0,r7     @ return nb
           ldmfd sp!{r1-r7} @ restaurer r1 à r7 depuis la pile
           bx     lr       @ branchement retour

```

```

@ x stocké dans r0
@ nb stocké dans r7
@ valeur de retour dans r0

```

```

calcul:    stmfd sp!{r1-r7} @ sauvegarder r1 à r7 dans la pile
           mov    r7,#0     @ nb=0
           b      condw     @ traduction de la boucle while à compléter

```

```

corpsw:    cmp    r0,#0        @ redondant
           blt    finssi       @ if (x <0)
           add    r7,r7,#1     @ nb ++
finssi:    mov    r0,r0,LSL #1 @ x = x *2
condw:     cmp    r0,#0
           bne    corpsw
suitew:    mov    r0,r7        @ return nb
           ldmfd  sp!{r1-r7} @ restaurer r1 à r7 depuis la pile
           bx     lr          @ branchement retour

calcul:    stmfd  sp!{r1-r7} @ sauvegarder r1 à r7 dans la pile
           mov    r7,#0        @ nb=0
condw :    cmp    r0,#0        @
           bne    suitew
           xmp    r0,#0        @ redonant
           blt    finssi       @ if (x <0)
           add    r7,r7,#1     @ nb ++
finssi:    mov    r0,r0,LSL #1 @ x = x *2
           b      condw
suitew:    mov    r0,r7        @ return nb
           ldmfd  sp!{r1-r7} @ restaurer r1 à r7 depuis la pile
           bx     lr          @ branchement retour

```

Que calcule la fonction ?

La fonction calcule le nombre de puissances de 2 composant x (autrement dit le nombre de 1 dans la représentation binaire de x).