DIALLO AISSATOU BOBO
Groupe IE3-00-02
AHRICHE NAJAT
Groupe IE3-00-01

Compte rendu du Tp2
Elec3A

Exercice 1 :

a) Somme :

Le programme calcule la somme des N premiers entiers naturels (1 + 2 + … + N) et place le résultat dans le registre **r0**.

- Le registre **r0** est initialisé à 0 : il contiendra la somme finale.
- Le registre **r1** est initialisé à 1 : il sert de compteur pour parcourir tous les entiers de 1 à N.
- La constante **N** est définie avec EQU, comme demandé dans l'énoncé.

Le programme utilise une **boucle** :

1. Il compare le compteur r1 avec N.
2. Tant que r1 ne dépasse pas N, il ajoute r1 à la somme dans r0.
3. Puis il incrémente r1 pour passer à l'entier suivant.

Quand r1 devient supérieur à N, la boucle s'arrête. Le programme se termine avec r0 contenant la somme totale.

Code :

```
        AREA TP2, CODE, READONLY
        ENTRY
        EXPORT __main

__main
        MOV     r0, #0          ; r0 = somme
        MOV     r1, #1          ; r1 = compteur

N       EQU     10              ; constante N = 10

boucle
        CMP     r1, #N          ; compare compteur avec N
        BHI     fin             ; si r1 > N → fin

        ADD     r0, r0, r1      ; r0 = r0 + r1
        ADD     r1, r1, #1      ; r1 = r1 + 1

        B       boucle          ; retour au début de la boucle

fin
        B fin           ; arrêt programme

        END
```

b) Multiplication :

Le but du programme est de calculer la multiplication **r0 × r1** en utilisant uniquement des **additions successives**, tout en gérant les cas où les opérandes peuvent être **positifs ou négatifs**.

1. **Initialisation :**
    - r2 est mis à 0 et servira à stocker le résultat.

- r3 servira de compteur d'itérations.
- r4 indique si le résultat final doit être négatif.

2. **Gestion des signes :**
   Le programme vérifie si r0 et r1 sont négatifs.
   Si c'est le cas, il les rend temporairement positifs (pour faciliter la répétition) et note le changement dans r4.
3. **Multiplication par additions successives :**
   Le programme ajoute r0 dans r2 autant de fois que la valeur de r1.
   Cela reproduit la définition mathématique :
   A × B = A + A + A + … (B fois)
4. **Application du signe final :**
   Si un seul des deux nombres était négatif, le résultat doit être négatif.
   Le programme applique alors un changement de signe sur r2.

Code :

```
    AREA TP2, CODE, READONLY

    ENTRY

    EXPORT __main
__main



; r0 = A

; r1 = B

; r2 = résultat

    MOV   r2, #0          ; résultat = 0

    MOV   r3, #0          ; compteur de répétitions

    MOV   r4, #0          ; indique si le résultat doit être négatif (0 = non, 1 = oui)

; Gestion des signes

    CMP   r0, #0

    BLT   negA            ; si A < 0, on le rend positif et on note le signe

contA

    CMP   r1, #0

    BLT   negB            ; si B < 0, pareil

contB

; Boucle d'additions successives

    MOV   r3, r1          ; on répète A, B fois

boucle

    CMP   r3, #0

   BEQ   finMult
```

```
    ADD    r2, r2, r0        ; résultat = résultat + A
    SUB    r3, r3, #1        ; on décrémente le compteur
    B      boucle
finMult
;  Appliquer le signe
    CMP    r4, #1
    BNE    fin
    RSB    r2, r2, #0        ; r2 = -r2
fin
    SWI    0x123456
    END
; Sous-routines changement de signe
negA   RSB    r0, r0, #0        ; A = -A
    ADD    r4, r4, #1        ; on change le signe final
    B      contA
negB   RSB    r1, r1, #0        ; B = -B
    ADD    r4, r4, #1        ; on change le signe final
    B      contB
```

c) Division

Le programme calcule la division entière de r0 (dividende) par r1 (diviseur) en utilisant uniquement des soustractions successives.
Le quotient est stocké dans r2 et le reste dans r3.
Dans un premier temps, le programme traite les signes : il mémorise le signe du dividende et du diviseur, puis travaille sur leurs valeurs absolues.
Ensuite, il effectue la division en soustrayant le diviseur au dividende autant de fois que possible : à chaque soustraction, le compteur r2 (quotient) est incrémenté et r3 représente le reste courant.
Quand le reste devient strictement inférieur au diviseur, la boucle s'arrête.
Enfin, le signe du quotient est déterminé en fonction des signes du dividende et du diviseur (si un seul est négatif, le quotient est négatif), et le reste prend le signe du dividende, tout en restant de valeur absolue inférieure au diviseur.

Code :

```
    AREA TP2, CODE, READONLY
    ENTRY
    EXPORT __main

__main
    MOV    r2, #0        ; quotient = 0
    MOV    r3, #0        ; reste
```

```
    MOV    r4, #0        ; signe du dividende (0=positif, 1=negatif)
    MOV    r5, #0        ; signe du diviseur  (0=positif, 1=negatif)

; Gestion signe du dividende
    CMP    r0, #0
    BGE    ok_dividende
    RSB    r0, r0, #0    ; r0 = |r0|
    MOV    r4, #1
ok_dividende

; Gestion signe du diviseur
    CMP    r1, #0
    BGE    ok_diviseur
    RSB    r1, r1, #0    ; r1 = |r1|
    MOV    r5, #1
ok_diviseur

;Division par soustractions successives
    MOV    r3, r0        ; reste temporaire = dividende
    MOV    r2, #0        ; quotient = 0

boucle
    CMP    r3, r1        ; tant que reste >= diviseur
    BLT    fin_division
    SUB    r3, r3, r1    ; reste = reste - diviseur
    ADD    r2, r2, #1    ; quotient++
    B      boucle

fin_division
; Calcul du signe du quotient
    EOR    r6, r4, r5    ; XOR des signes
    CMP    r6, #0
    BEQ    quotient_ok
    RSB    r2, r2, #0    ; quotient négatif
quotient_ok

; Signe du reste
    CMP    r4, #0
    BEQ    reste_ok
    RSB    r3, r3, #0    ; reste négatif si dividende négatif
reste_ok

    SWI    0x123456      ; arrêt programme
    END
```

Exercice 2 :

Que fait le programme ci-dessous ?

Le programme commence par enregistrer, dans la partie __main, l'adresse mémoire des deux tableaux : srcstr dans r1 (source) et dststr dans r0 (destination).
Ensuite, dans la partie strcopy, il charge à chaque itération un octet de la chaîne source à l'adresse pointée par r1 dans le registre r2, puis il sauvegarde cet octet dans la chaîne destination à l'adresse pointée par r0. Les registres r0 et r1 sont incrémentés à chaque tour de boucle pour passer au caractère suivant.
La fonction strcopy est exécutée tant que l'octet lu n'est pas égal à 0 (caractère nul de fin de chaîne). Lorsque ce 0 est rencontré, la copie s'arrête et le programme se termine.

Quelle est la taille, en bytes (octets), de chacune des cases des deux tableaux « srcstr » et « dststr » ?

Les deux tableaux sont déclarés avec l'instruction **DCB**, ce qui signifie *Define Constant Byte*.
Donc chaque case de srcstr et dststr occupe **1 octet**.

Pour srcstr :
La chaîne "First string - source" contient 23 caractères au total, plus l'octet 0 de fin de chaîne.
Le tableau srcstr occupe donc **24 octets** en mémoire.

Pour dststr :
La chaîne "Second string - destination" contient 27 caractères, plus l'octet 0 final.
Le tableau dststr occupe donc **28 octets** en mémoire.

Expliquez pourquoi on incrémente de un '1' r0 et r1 dans ce programme.

Les instructions :

```
LDRB r2, [r1], #1
STRB r2, [r0], #1
```

Lisent et écrivent **un seul octet** à chaque fois.
Après chaque accès mémoire, r1 et r0 sont **automatiquement incrémentés de 1** :

- r1 est avancé vers l'octet suivant de la chaîne source srcstr,
- r0 est avancé vers l'octet suivant de la chaîne destination dststr.

On incrémente donc r0 et r1 de 1 pour parcourir les deux chaînes **caractère par caractère.**

Exécution pas à pas et affichage en ASCII

Lorsqu'on exécute le programme pas à pas et qu'on affiche la mémoire en ASCII :

- on voit d'abord srcstr contenant "First string - source" et dststr contenant "Second string - destination",
- puis, à chaque itération, le caractère courant de srcstr est recopié dans dststr,
- à la fin de l'exécution, la zone mémoire de dststr contient la même chaîne que srcstr, suivie du caractère nul 0 (fin de chaîne).

Exercice 3 :

Le but du programme est de recopier le tableau src dans le tableau dst en inversant l'ordre des éléments. Les éléments sont des mots de 32 bits (4 octets), déclarés avec DCD.

Dans __main, on charge l'adresse de src dans r0 et l'adresse de dst dans r1. On positionne ensuite r0 sur le **dernier élément** du tableau source en ajoutant l'offset (N−1)*4. Le registre r2 contient le nombre d'éléments à traiter.

La boucle lit à chaque itération un mot de 32 bits depuis src en partant de la fin (LDR r3, [r0], #-4), puis le recopie dans dst en avançant vers l'avant (STR r3, [r1], #4). Le compteur r2 est décrémenté jusqu'à ce que tous les éléments aient été copiés.
À la fin de l'exécution, le tableau dst contient les mêmes valeurs que src, mais dans l'ordre inverse.

Code :

```
    AREA    TP2, CODE, READONLY

    ENTRY

    EXPORT  __main

N     EQU    20           ; nombre d'éléments dans le tableau

OFFSET  EQU    76          ; (N-1)*4 = 19 * 4 bytes = 76

__main

    LDR    r0, =src      ; r0 = adresse de src

    LDR    r1, =dst      ; r1 = adresse de dst

    ADD    r0, r0, #OFFSET ; r0 pointe maintenant sur le DERNIER élément de src

    MOV    r2, #N        ; compteur = nombre total d'éléments

boucle

    LDR    r3, [r0], #-4   ; lire un mot depuis src puis r0 -= 4 (on recule dans src)

    STR    r3, [r1], #4    ; écrire ce mot dans dst puis r1 += 4 (on avance dans dst)

    SUB    r2, r2, #1      ; décrémenter compteur

    BNE    boucle          ; tant qu'il reste des éléments, continuer

stop

    B stop ; fin de programme

    END

    AREA    DATA_TP2, DATA, READWRITE
```

```
src     DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4

dst     DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Conclusion :

Ce TP nous a permis de comprendre les bases essentielles de la programmation en assembleur ARM Cortex-M3 . Dans la première partie, nous avons manipulé les opérations arithmétiques fondamentales en réalisant nous-mêmes la somme, la multiplication et la division, uniquement à l'aide d'additions et de soustractions successives, ce qui nous a amenés à gérer explicitement les signes et les registres.
Dans la deuxième partie, nous avons appris à manipuler la mémoire en recopiant une chaîne de caractères octet par octet, puis un tableau de mots de 32 bits en inversant son ordre, en observant pas à pas l'évolution des adresses et du contenu mémoire.
Ce TP a ainsi renforcé notre maîtrise des instructions ARM (LDR, STR, LDRB, STRB, CMP, BNE…), de l'adressage mémoire et de la gestion des pointeurs, tout en montrant l'importance de la compréhension bas niveau du fonctionnement d'un microprocesseur.