INF401

TP6et7 « **Parcours de tableaux** »

Shaghayegh HAJMOHAMMADKASHI , Kimia KHADEMLOU– MIN3

Pendant les séances 6 et 7, nous avons plongé dans le monde des tableaux en langage d'assemblage ARM. L'objectif était de créer un programme capable de générer et d'afficher les tables de multiplication de 1 à 10. En utilisant des techniques de programmation spécifiques à l'assemblage ARM, nous avons exploré la création, le remplissage et l'affichage de tableaux, offrant ainsi une expérience pratique de la manipulation de données structurées dans ce langage.

6.1 Tables de multiplications

Questions concernant le remplissage du tableau

- Dans quelle case du tableau table se trouve le produit 1*1 ?

Le produit 1*1 se trouve dans la case table[0][0].

- Dans quelle case du tableau table se trouve le produit 1*2 ?

Le produit 1*2 se trouve dans la case table[0][1].

- Dans quelle case du tableau table se trouve le produit 7*9 ?

Le produit 7*9 se trouve dans la case table [6][8].

- Dans quelle case du tableau table se trouve le produit 10*10 ?

Le produit 10*10 se trouve dans la case table[9][9].

6.2 Affichage du tableau

la prochaine étape consiste à afficher ce tableau à l'écran. Cet affichage doit refléter fidèlement la disposition présentée dans la figure 6.1, avec des barres verticales séparant les cellules et des tirets horizontaux pour les lignes de séparation.

```
LEXIQUE :
 N_MAX    : l'entier 10
 ESPACE   : le caractère ' '   // code ascii 32
 BARRE    : le caractère '|'   // code ascii 124
 TIRETS   : le caractère '---' // code ascii 45
 Ligne    : le type tableau  sur [0..N_MAX-1] d?entiers
 table    : le tableau sur [0..N_MAX-1] de Ligne
 n_lig,n_col : deux entiers
 mult : un entier
```

```
ALGORITHME :
  pour n_lig parcourant [0..N_MAX-1] :
       pour n_col parcourant [0..N_MAX-1] :
            ecrire_car(BARRE);
            mult <-- table[n_lig][n_col];
            si mult < 100 alors ecrire_car(ESPACE);
            si mult < 10 alors ecrire_car(ESPACE);
            ecrire_int(mult);
       ecrire_car(BARRE);
       a_la_ligne();
       répéter N_MAX fois :
            ecrire_car(BARRE);
            ecrire_chn(TIRETS);
       ecrire_car(BARRE);
       a_la_ligne();
```

La traduction de cet algorithme :
(trouvez la version plus détaillé avec des commentaires dans le code source)

```
mov     r0, #0
affichage_loop:
    cmp     r0, #N_MAX
    bge     finboucle1
    mov     r2, #0
boucle1:
    cmp     r2, #N_MAX
    bge     finboucle2
    ldr     r1, LD_barre
    ldrb    r1, [r1]
    bl      EcrCar
    ldr     r3, LD_debutTab
    mov     r10, #4
    mul     r8, r0, r10
    mov     r7, #10
    mul     r4, r8, r7
    add     r3, r3, r4
    mul     r4, r2, r10
    add     r3, r3, r4
    ldrb    r4, [r3]
    @ Gestion des espaces pour l'alignement
    cmp     r4, #100
    bge     si1
    ldr     r1, LD_espace
    ldrb    r1, [r1]
    bl      EcrCar
si1:
    cmp     r4, #10
    bge     si2
    ldr     r1, LD_espace
    ldrb    r1, [r1]
```

```
    bl      EcrCar
si2:
    mov     r1, r4
    bl      EcrNdecim32
    add     r2, r2, #1
    b       boucle1
finboucle2:
    ldr     r1, LD_barre
    ldrb    r1, [r1]
    bl      EcrCar
    bl      AlaLigne
    @ Écrire la ligne de séparation après chaque ligne du tableau
    mov     r3, #0
    ldr     r1, LD_barre
    ldrb    r1, [r1]
boucle_sep:
    cmp     r3, #N_MAX
    bge     fin_sep
    ldr     r1, LD_barre
    ldrb    r1, [r1]
    bl      EcrCar
    ldr     r1, LD_tirets
    bl      EcrChn
    add     r3, r3, #1
    b       boucle_sep
fin_sep:
    ldr     r1, LD_barre
    ldrb    r1, [r1]
    bl      EcrCar
    bl      AlaLigne
    add     r0, r0, #1
    b       affichage_loop
finboucle1:
    b       fin
```

Remarque :
Comme indiqué dans le sujet, nous avons récupéré le fichiers es.s pour la traduction des fonctions d’entrées-sorties.

```
● khademlk@im2ag-turing-01:[~/inf401/TP6et7]: arm-eabi-run tabmult
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
 |  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
 |---|---|---|---|---|---|---|---|---|---|
```

6.3 Remplissage du tableau

Dans le cadre de notre implémentation, nous avons choisi d'utiliser l'algorithme de multiplication par additions successives pour réaliser l'opération de multiplication de deux entiers positifs. Cette approche consiste à ajouter le multiplicand à lui-même autant de fois que l'indique le multiplicateur, ce qui permet d'obtenir le produit final.

```
LEXIQUE :
  mult, a et b : trois entiers positifs ou nuls

ALGORITHME :
    mult <-- 0;
    répéter a fois : mult <-- mult + b;
```

La traduction de cet algorithme en ARM :

(trouvez le code complet de cette partie dans le fichier tabmult3.s)

```
@ Remplissage du tableau
@ multiplication par additions successives

    ldr r0, =debutTab         @ Charger l'adresse du début du tableau dans r0
    mov r1, #1                @ Initialiser le compteur de ligne à 1
remplissage:
    cmp r1, #N_MAX            @ Vérifier si le compteur de ligne dépasse N_MAX
    bgt affichage             @ Si oui, passer à l'affichage
    mov r2, #1                @ Initialiser le compteur de colonne à 1
remplissage_ligne:
    cmp r2, #N_MAX            @ Vérifier si le compteur de colonne dépasse N_MAX
    bgt prochaine_ligne       @ Si oui, passer à la prochaine ligne
    mov r3, #0                @ Initialiser le produit à 0
    mov r4, r1                @ Copier le compteur de ligne dans r4
    mov r5, r2                @ Copier le compteur de colonne dans r5
    somme_loop:
        add r3, r3, r4        @ Ajouter r4 à r3 (somme partielle)
```

```
        subs r5, r5, #1       @ Décrémenter le compteur de colonne
        bne somme_loop        @ Répéter jusqu'à ce que r5 soit nul
    str r3, [r0], #4          @ Stocker le produit dans le tableau et avancer l'adresse
    add r2, r2, #1            @ Incrémenter le compteur de colonne
    b remplissage_ligne       @ Revenir pour la prochaine colonne
prochaine_ligne:
    add r1, r1, #1            @ Incrémenter le compteur de ligne
    b remplissage             @ Revenir pour la prochaine ligne
```

6.3.1 Codage d'un tableau à 2 dimensions

Lorsque nous travaillons avec des tableaux à deux dimensions en langage d'assemblage, nous devons les stocker en mémoire de manière à ce qu'ils puissent être facilement accessibles et manipulés. Pour ce faire, nous transformons souvent le tableau à deux dimensions en un tableau à une dimension en rangeant les lignes du tableau les unes après les autres. Chaque ligne du tableau devient alors une séquence de cases contenant chacune un élément du tableau.

Ainsi, pour coder un tableau à deux dimensions en langage d'assemblage, nous utilisons généralement deux boucles imbriquées : une pour parcourir les lignes et une autre pour parcourir les colonnes. Cela nous permet d'accéder à chaque élément du tableau et de le stocker dans le tableau unidimensionnel de manière organisée et efficace.

e00	e01	e02	e03	e04	e05
e10	e11	e12	e13	e14	e15
e20	e21	e22	e23	e24	e25
e30	e31	e32	e33	e34	e35

table : e00
e01
e02
e03
e04
e05
e10
e11
...
e35

Pour calculer l'adresse de table[x][y] en fonction de table, x et y, nous utilisons la formule suivante :

table + x*40 + y*4

Car une ligne est de 40 octets (il y a 10 colonne de 4 octets dans une ligne), et chaque colonne est de 4 octets.

En langage d'assemblage, pour effectuer le calcul d'adresse et écrire la valeur en mémoire pour **table[x][y] <-- valeur**, nous utilisons l'instruction suivante :

```
    str r3, [r0], #4          @ Stocker le produit dans le tableau et avancer l'adresse
```

6.3.2 Codage du programme de multiplication (version 1)

Dans la version 1 du programme de multiplication, nous avons exploré deux approches différentes pour remplir le tableau des multiplications:

Utilisation de l'addition successive (tabmult3.s)

Utilisation de l'instruction "mul" (tabmult2.s)

Veuillez trouver le code complète et détaillé de cette partie dans le fichier de code source.

6.3.3 Codage du programme de multiplication (version 2)

dans cette version, nous avons utilisé uniquement une seule boucle pour remplissage du tableau.

Nous avons traduit cet algorithme :

```
Définir N_MAX = 10
Initialiser n_lig à 1
Initialiser n_col à 1

Tant que n_lig * n_col est inférieur ou égal à N_MAX * N_MAX :
    Calculer mult = n_lig * n_col
    Calculer l'adresse mémoire pour stocker mult dans le tableau
    Stocker mult à l'adresse mémoire calculée
    Incrémenter n_col de 1
    Si n_col dépasse N_MAX :
        Passer à la ligne suivante :
        Incrémenter n_lig de 1
        Réinitialiser n_col à 1
    Fin Si
Fin Tant que
```

Veuillez retrouver l'implémentation détaillée de cet algorithme dans le fichier tabmult.s.

Résumé :

Lors des séances 6 et 7, nous avons exploré la création et la manipulation de tableaux en langage d'assemblage ARM. En utilisant des méthodes telles que les additions successives pour la multiplication, nous avons conçu un programme pour remplir un tableau avec les tables de multiplication de 1 à 10, puis afficher son contenu. Ces séances nous ont permis de mieux comprendre la gestion des tableaux dans un environnement d'assemblage ARM.