

3 – Manipulation de tableaux, modes d'adressage

Objectifs

L'objectif général est d'approfondir la programmation assembleur : manipulations de tableaux et utilisation des modes d'adressage, élaboration d'algorithmes.

Rappels

Pour les exercices qui suivent, on rappelle les modes d'adressage disponibles en assembleur ARM :

- pré-indexé :

LDR/STR Rd, [Rn, offset] accès à mem[Rn+offset]

- pré-indexé automatique :

LDR/STR Rd, [Rn, offset]! $Rn \leftarrow Rn + offset$, puis accès à mem[Rn]

- post-indexé automatique :

LDR/STR Rd, [Rn], offset accès à mem[Rn] puis $Rn \leftarrow Rn + offset$

Où *offset* est soit une valeur immédiate (*#literal*), soit un registre (*Rm*), soit un registre avec décalage (*Rm, shift*).

1.1 Opérations sur des tableaux

Soient les trois boucles suivantes décrites en langage C. Initialiser les tableaux X, Y et la variable S en mémoire, puis complétez les trois programmes assembleur suivants :

a)

```
for (i=0; i<10; i++)
    S = S + X[i] + Y[i];
```

```

                                     @ Section de données -----
S:          .word          0
X:          .word          0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Y:          .word          0, 1, 2, 3, 4, 5, 6, 7, 8, 9
                                     @ Section de code -----
                ADR          r2, X           @ r2 pointe sur X[0]
                ADR          r4, Y           @ r4 pointe sur Y[0]
                LDR          r5, S           @ r5 <- valeur de S
                MOV          r0, #0          @ I=0
LOOP:        ...
                ...
EXIT:        ...
```

b)

```
for (i=0; i<10; i++)
    if (X[i]>0) S = S + X[i];
```

```

                                     @ Section de données -----
S:          .word          0
X:          .word          0, 1, -2, 3, -4, 5, -6, 7, -8, 9
                                     @ Section de code -----
                ADR          r2, X           @ r2 pointe sur X[0]
                LDR          r5, S           @ r5 <- valeur de S
```

```

MOV          r0, #0          @ I=0
LOOP:        ...
            ...
EXIT:        ...

```

c)

for (i=1; i<=9; i++)
Y[i-1] = X[i] + X[i-1];

```

                                @ Section de données -----
X:          .word      0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Y:          .word      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
                                @ Section de code -----
                                @ r2 pointe sur X[0]
                                @ r4 pointe sur Y[0]
                                @ I=1
                                ADR          r2, X
                                ADR          r4, Y
                                MOV          r0, #1
LOOP:        ...
            ...
EXIT:        ...

```

1.2 Algorithme de tri à bulles

Le principe du tri à bulles est de comparer deux valeurs adjacentes et d'inverser leur position si elles sont mal placées. Ainsi pour trier une liste dans l'ordre croissant, si un premier nombre x est plus grand qu'un deuxième nombre y, alors x et y sont mal placés et il faut les inverser. Si, au contraire, x est plus petit que y, alors on ne fait rien et l'on compare y à z, l'élément suivant. On parcourt ainsi entièrement la liste, puis on recommence jusqu'à ce qu'il n'y ait plus aucun élément à inverser.

a) Représenter l'état de la suite 3, 107, 27, 12, 322, 155, 63 à chaque étape de l'algorithme de tri à bulles (dans l'ordre croissant).

b) Donner une description détaillée de l'algorithme.

c) Ecrire un programme assembleur qui trie les éléments d'un tableau TAB = {3, 107, 27, 12, 322, 155, 63}.

```

                                @ Section données
TAB:        .word      3, 107, 27, 12, 322, 155, 63
N:          .word      7
                                @ Section code
                                @ R0 ← @TAB
                                @ I=0
                                ADR          R0, TAB
                                MOV          R1, #0
MAINLOOP:   ...
            ...
            B           MAINLOOP
EXIT:

```

1.3 Conversion hexadécimal - ASCII

On souhaite écrire un programme qui convertit la valeur hexadécimale VALEUR en caractères ASCII (pour pouvoir l'afficher par la suite comme du texte par exemple).

Le résultat sera placé dans un tableau TABASCII où chaque élément correspond au code ASCII du caractère hexadécimal.

Exemple : 0xA76E2FF1 → (65)(55)(54)(69)(50)(70)(70)(49)

Rappel : en code ASCII, les caractères sont représentés par un nombre décimal: les chiffres 0 à 9 sont représentés par les codes 48 à 57 et les lettres A à F par les codes 65 à 70.

- Expliquer l'utilité du tableau TABCODE (voir code ci-dessous).
- Proposer et décrire un algorithme réalisant cette conversion.
- Développer le programme en complétant le code assembleur suivant :

```

                                @ Section données
TABASCII:  .byte      0, 0, 0, 0, 0, 0, 0, 0
VALEUR:    .word      0xA76E2FF1
TABCODE:   .byte      48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 65, 66, 67,
68, 69, 70

                                @ Section code
                                @ LOOP COUNTER
                                @ r1 <- VALEUR
MOV        r0, #8
LDR        r1, VALEUR

...
LOOP:      ...
...
BNE        LOOP

```

1.4 Clignotement d'une LED en assembleur

Le code C suivant, vu en TP, fait clignoter la LED connectée au port PH.3 de la carte MCBSTM32F400.

```

#include "stm32f4xx.h"
int main(void)
{
    uint8_t i;
    volatile uint32_t tick;

    // Init
    RCC->AHB1ENR    = 0x00100000; // Reset RCC AHB1
    RCC->AHB1ENR    = 0x00000080;

    GPIOH->MODER    = 0x00000040;
    GPIOH->OTYPER    = 0x00000000;
    GPIOH->OSPEEDR   = 0x00000080;
    GPIOH->PUPDR     = 0x00000080;

    // Blink 10 times
    for (i=0; i<10; i++) {
        // LED on
        GPIOH->BSRRL= 0x0008;
        // delay
        for (tick=0; tick<5000000; tick++);
        // LED off
        GPIOH->BSRRH= 0x0008;
        // delay
        for (tick=0; tick<5000000; tick++);
    }

    return 0;
}

```

On cherche à écrire un programme équivalent, mais directement en assembleur.

a) En cherchant dans les définitions du fichier `stm32f4xx.h` (qui peut être téléchargé à l'url <http://users.polytech.unice.fr/~bilavarn/>), déterminer les adresses physiques des registres `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `BSRRL` et `BSRRH` du `GPIOH`.

b) De la même façon, déterminer l'adresse physique du registre `RCC_AHB1ENR`.

c) Ecrire un programme assembleur qui fait clignoter la LED connectée au port `PH.3`.

Tester le programme en simulation.

Le programme sera ensuite testé sur la carte `MCBSTM32F400`.

On observera aussi le code assembleur généré par le compilateur `GCC ARM` pour comparaison.

e) Comment adapter le code pour la LED verte d'une carte `STM32F4-Discovery` ?