Tarefa 6 - **Subroutines**

Carlos Eduardo Sena Rebouças Garcia 9345255

Diego Boccoli Gallego 9344744

Gabriel Rocha Amorim 9395112

**6.5.1 Transmission of arguments**

**Write ARM assembly code to compute the function a = b×c + d. Write three separate programs that:**

**•transmit the arguments by way of registers with one subroutine, func1**

main: LDR r1, =0x123

LDR r2, =0x124

LDR r4, =0x01

BL func1

B out

func1: MLA r0, r1, r2, r4

BX lr

out:

**•transmit the arguments by way of the addresses with one subroutine, func1**

main: LDR r1, =0x01

LDR r2, =0x06

LDR r4, =0x0f

STRB r1, a

STRB r2, b

STRB r4, c

BL func1

B out

a: .byte 0x00

b: .byte 0x00

c: .byte 0x00

d .byte 0x00

out:

func1: LDRB r6, b

LDRB r7, c

LDRB r8, d

MLA r0, r6, r7, r8

STRB r0, a

BX lr

**•transmit the arguments by way of the stack with two subroutines, func1 and func2, that demonstrate stack functionality.**

main: LDR r1, =0x123

LDR r2, =0x124

LDR r4, =0x01

BL func1

B out

a: .word 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

func1: LDR r0, =a

STMIA r0!, {r1, r2, r4, lr}

BL func2

LDMDB r0!, {r1, lr}

out1: BX lr

func2: LDMDB r0!, {r1-r3, r5}

MLA r4, r1, r2, r3

STMIA r0!, {r4, r5}

BX lr

out:

**6.5.2 Bubble sorting**

**1.Write ARM assembly code to perform an ascending bubble sort operation on a list located in memory. The length of the list is located at 0x4000 and the first element of the list is located at 0x4001. The sorted list must be stored back to the original array of memory locations starting at 0x4001. Assume an array of bytes.**

**2.Modify your code to utilize a full descending stack. Sorting must be done on the stack only. Once the stack is sorted, store the sorted stack back to the original array of memory locations starting at 0x4001. The algorithm for the bubble sort is as follows:   
 a.Compare adjacent elements. If the first element is greater than the second, swap them.  
 b.Do this for each pair of adjacent elements, starting with the first two and ending with the last two. At this point the last element should be the greatest.**

**c.Repeat the steps for all elements except the last one.**

**d.Repeat this process for one fewer element each time, until you have no more pairs to compare.**

main: LDR r0, size @ int size = 4;

LDR r1, =arr @ int [] arr = [4, 15, 13, 1];

SUB r9, r0, #4 @ aux

LDR r2, =0x0 @ i = 0

loop: CMP r2, r0 @ for(i = 0; i < size; i++)

BGE fora

MOV r3, r2 @ j = i

loop2: CMP r3, r9 @ for(j = i; j < size-1; j++)

BGE out

ADD r7, r1, r3

LDMIA r7!, {r4, r5}

CMP r4, r5

MOVLE r6, r4

MOVLE r4, r5

MOVLE r5, r6

STMDB r7, {r4, r5}

ADD r3, r3, #4

B loop2

out: ADD r2, r2, #4

B loop

arr:

.word 0x04, 0x0f, 0x0d, 0x01

size:

.word 16

fora:

**6.5.3 Magic squares  
Write ARM assembly to check whether an N×N matrix is a magic square. A magic square is an N×N matrix in which the sum of the numbers in every row, column, or diagonal is N(N^2+ 1)/2. All matrix entries are unique numbers from 1 to N^2. For example, suppose you wanted to test a famous example of a magic square: The matrix starts at location 0x4000 and ends at location (0x4000+N2). Put the 16 in location 0x4000, 3 in 0x4001, 2 in 0x4002, 13 in 0x4003, 5 in 0x4004, ..., and 1 in 0x400F. Put N in r1. Assume that everything is in bytes, which puts a constraint on N. Write the code so that, if the matrix is a magic square, r9 is set, and otherwise it is cleared. To test the algorithm, you can search the Internet for other magic square examples, such as Ben Franklin's own 8 × 8 magic square.**

**6.5.4 More stacks**

**Write ARM assembly to implement a push operation without the use of load/store multiple instructions. Write the code to handle bytes, half-words, and words. Use r0 to indicate the data type. A value of 1 in r0 indicates that a byte is to be pushed, 2 indicates a half-word, and 4 indicates a word. Put the data to push in r1.**

main: LDR r0, =0x4

LDR r1, =0x16

LDR r2, =stack

LDR r3, size

MOV r4, r3

ADD r3, r3, r2

STR r1, [r3], r0

ADD r4, r4, r0

STR r4, size

size:

.word 0x12

stack:

.word 0x04, 0xff, 0x00, 0x13, 0x0, 0x0