MARMARA UNIVERSITY

CSE 4095.1- Project 1 Report

Serkan Koç - 150118073
Erkam Karaca - 150118021
Bahadır Alacan - 150118042

Question 1

Firstly we defined data area and two strings.

```
    ALIGN
    AREA    myData, DATA, READWRITE
    ALIGN
str1    DCB "ABCD", 0 ; First string
str2    DCB "CD", 0 ; Second string
    END
```

```
__main  PROC

    LDR R0, =str1 ;loading string 1 to r0
    LDRB R1, [R0], #1 ;loading 1 byte to r1 and then update r0 by r0+1

check   ; Check if the byte is a valid hexadecimal digit
    CMP R1, #'0' ;comparing the byte with ascii value of 0
    BLO invalid_digit ; if ascii value of the byte is lower than the ascii value of 0 go to invalid digit
    CMP R1, #'9' ; comparing the byte with ascii value of 9
    BHI letter_check ; if ascii value of the byte is greater than the ascii value of 9 go to letter check
    B valid_digit ; this means the byte is a valid digit

letter_check
    CMP R1, #'A' ; comparing the byte with ascii value of A
    BLO invalid_digit ; if ascii value of the byte is lower than the ascii value of A go to invalid digit
    CMP R1, #'F' ; comparing the byte with ascii value of F
    BHI lowercase_check ; if ascii value of the byte is greater than the ascii value of F go to lowercase check
    B valid_digit ; this means the byte is a valid uppercase letter

lowercase_check
    CMP R1, #'a' ; comparing the byte with ascii value of a
    BLO invalid_digit ; if ascii value of the byte is lower than the ascii value of a go to invalid digit
    CMP R1, #'f' ; comparing the byte with ascii value of f
    BHI invalid_digit ; if ascii value of the byte is greater than the ascii value of f go to invalid check
    B valid_digit ; this means the byte is a valid lowercase letter

invalid_digit B invalid_digit ;infinite loop

valid_digit B loop
```

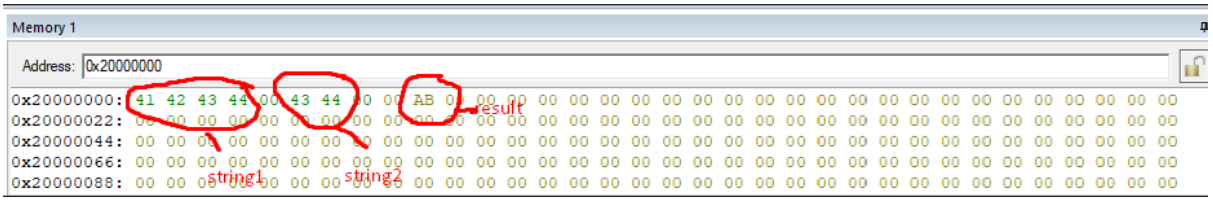
Here we are validating the given inputs are hexadecimal or not. If not the code goes infinite loop. In the check label we controlled if the read byte is digit or not. In the letter_check label we controlled if the byte is uppercase letter or not. In the lowercase_check label we check if the byte is lowercase letter or not.

```
loop
    SUB R3, R1, #48 ; r3=r1-'0'
    CMP R3, #9 ; if r3 is less than 9 that means r1 is a digit
    BLE digit_func ; if r1 is a digit go to digit_func

    SUB R1, R1, #55 ; converting the r1 into integer
    ADD R2, R1, R2 ; r2 = r1+r2
    LSL R2, #4 ; shifting r2 4 times left

    LDRB R1, [R0], #1 ;loading 1 byte to r1 and then update r0 by r0+1
    CMP R1, #0 ; checking if string loading is finished
    BNE check

one_right_of_str1
    LSR R2, #4 ; one extra left shift is cancelled here with shift right
    CMP R7, #1 ; flag to convert the second string into int
    BEQ calculate_sub ; if both strings are converted to int go to sub
    B one_right_of_str2

one_right_of_str2
    ADD R7, R7, #1 ; updating flag
    MOV R5, R2 ; moved int1 to r5 since we need r2 again
    AND R2, R2, #0 ; zeroed r2
    LDR R0, =str2 ; loading address of str2 to r0
    LDRB R1, [R0], #1 ; ;loading 1 byte of string2 to r1 and then update r0 by r0+1
    B check

calculate_sub
    SUB R6, R5, R2 ; r6=int1-int2
    STR R6, [R0] ; storing the result to the memory
    B stop

stop    B       stop            ; R2 will be str2 and R5 will be str1

digit_func
    SUB R1, R1, #48 ; r1= r1-'0'
    ADD R2, R1, R2 ; r2 = r1+r2
    LSL R2, #4 ; shifting r2 4 times left
    LDRB R1, [R0], #1 ;loading 1 byte to r1 and then update r0 by r0+1
    CMP R1, #0 ;check if end of the string
    BEQ one_right_of_str1
    B check
```

If inputs are valid the code continues from loop label. In loop label we convert the string value to int. If the string byte is digit it is converted in digit_func label. Otherwise it is converted in the loop label. Since we are starting from the most significant byte of the string we had to shift it to left. One extra shift is canceled in the one_right_of_str1 and one_right_of_str2 labels. After converting the strings to integers in the calculate_sub label we subtract them and push the result into the memory.

Values of registers after the execution:

Register	Value
Core	
R0	0x20000008
R1	0x00000000
R2	0x000000CD
R3	0x00000014
R4	0x00000000
R5	0x0000ABCD
R6	0x0000AB00
R7	0x00000001
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000608
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000282
xPSR	0x61000000



Question 2

In the second question, we defined our data in the “myData” section of the code as shown below. We defined strings as DCB with null termination at the end. Then we defined two buffers for the strings to allocate a block of memory.

```
    ALIGN
    AREA    myData, DATA, READWRITE
    ALIGN
str1    DCB "AaBbCcDd.", 0 ; First string
str2    DCB "CDcd.", 0  ; Second string
buff1       SPACE 20  ; Buffer for first string
buff2       SPACE 20  ; Buffer for second string
    END
```

Then, we are loading memory address of the first string to “R0” and the beginning of the allocated memory address for the first string to “R1” at the first two line. Then, in the loop1, we are loading first character of the first string from “R0” and immediately increment the “R0”’s pointer as 1 byte in order to point the next character.

Then we are checking for null terminator if it is null terminator program goes to done1 to add a null terminator to buffer1 and to store it in the memory. Then make the necessary loadings for the process of the second string.

If it is not null terminator, the program compares ASCII value of that character with ASCII value of “A” in order to satisfy the first condition of the being an uppercase letter. If it is value less than value of “A” then it goes to “not_upper” to skip to next character.

If it is greater than or equal to the ASCII value of “A” then we are checking it is less than the value of “Z” to determine this character is exactly an uppercase letter. If it is not below or equal to the “Z” the program skips to the next character again.

Finally at the last row of the loop1, we are adding 32(‘a’ - ‘A’) as decimal to the ASCII value of character in R2 in order to convert uppercase letter to lowercase letter.

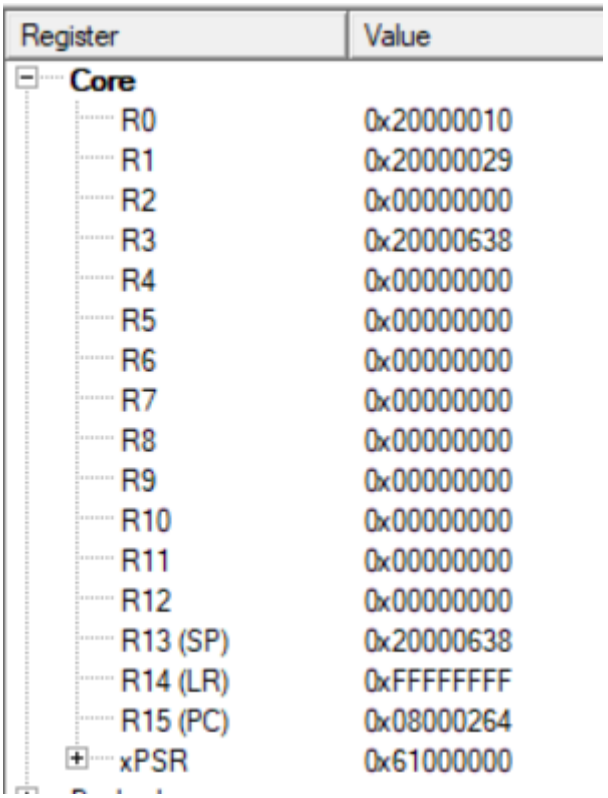
Then program goes to not_upper section again and stores converted character to buffer1 and goes to loop1 to process the next character.

```
__main  PROC

    ; Read first string and convert uppercase letters to lowercase
    LDR R0, =str1
    LDR R1, =buff1

loop1
    LDRB R2, [R0], #1       ; Load a character from str1
    CMP R2, #0              ; Check if null terminator is reached
    BEQ done1               ; If yes, terminate loop
    CMP R2, #'A'            ; Check if character is uppercase letter
    BLT not_upper          ; If not, skip to next character
    CMP R2, #'Z'            ; Check if character is within range of uppercase letters
    BGT not_upper          ; If not, skip to next character
    ADD R2, R2, #'a' - 'A'  ; Convert uppercase letter to lowercase

not_upper
    STRB R2, [R1], #1       ; Store the converted character in buff1
    B loop1

done1
    MOV R2, #0              ; Add null terminator to buff1
    STRB R2, [R1]
     ; Read second string and begin to convert lowercase letters to uppercase
    LDR R0, =str2
    LDR R1, =buff2
```

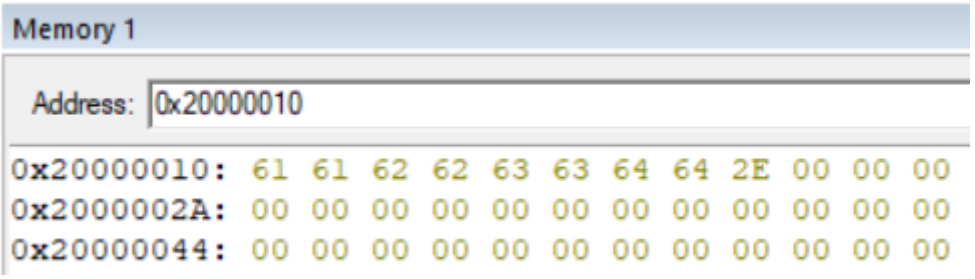
At the loop2 basically we are doing the same things that we are explained above but the differences are that firstly we are doing comparison with the ASCII values of ‘a’ and ‘z’ and subtracting ‘a’ - ‘A’. The other difference is that in done2 part we are adding null terminator and storing it to memory but we are not loading new memory address or buffer address to registers again because of the program is being finished.

```
loop2
    LDRB R2, [R0], #1       ; Load a character from str2
    CMP R2, #0              ; Check if null terminator is reached
    BEQ done2               ; If yes, terminate loop
    CMP R2, #'a'            ; Check if character is lowercase letter
    BLT not_lower          ; If not, skip to next character
    CMP R2, #'z'            ; Check if character is within range of lowercase letters
    BGT not_lower          ; If not, skip to next character
    SUB R2, R2, #'a' - 'A'  ; Convert lowercase letter to uppercase

not_lower
    STRB R2, [R1], #1       ; Store the converted character in buff2
    B loop2

done2
    MOV R2, #0              ; Add null terminator to buff2
    STRB R2, [R1]

stop    B stop              ; End of program, loop indefinitely
```

Values of the registers and memory at the end of the program execution. Address 0x20000010 contains the first strings hex values and 0x20000024 contains second strings hex values.



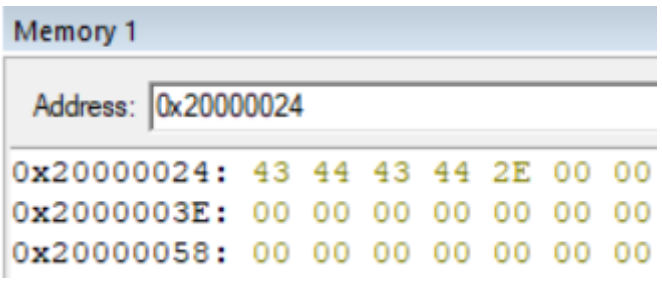
Register	Value
Core	
R0	0x20000010
R1	0x20000029
R2	0x00000000
R3	0x20000638
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000638
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000264
xPSR	0x61000000



Memory 1

Address: 0x20000010

```
0x20000010: 61 61 62 62 63 63 64 64 2E 00 00 00
0x2000002A: 00 00 00 00 00 00 00 00 00 00 00 00
0x20000044: 00 00 00 00 00 00 00 00 00 00 00 00
```



Memory 1

Address: 0x20000024

```
0x20000024: 43 44 43 44 2E 00 00
0x2000003E: 00 00 00 00 00 00 00
0x20000058: 00 00 00 00 00 00 00
```

Question 3

In the third question we defined 4-by-3 matrix with random numbers. We initialized transpose_matrix with 48 bytes space. Because we defined matrix with DCD instruction and one element keeps 4 byte space.

```
50      AREA myData, DATA, READWRITE
51      ALIGN
52  matrix DCD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
53      ; 1 4 7 10
54      ; 2 5 8 11
55      ; 3 6 9 12
56  transpose_matrix space 48
57      END
```

In main area, we defined loop variables. Initialized address of matrices and we defined outer loop. In outer loop we controlled i variable. When it becomes 3 program branches to done. If it is not three it continues and add 1 to i which is R0. Then calculates address of matrix for using in inner loop. Stores in R3 register.

In inner loop we controlled j variable. When it becomes 4 program returns to outer loop. Inner loop does transpose operations. For example our matrix:

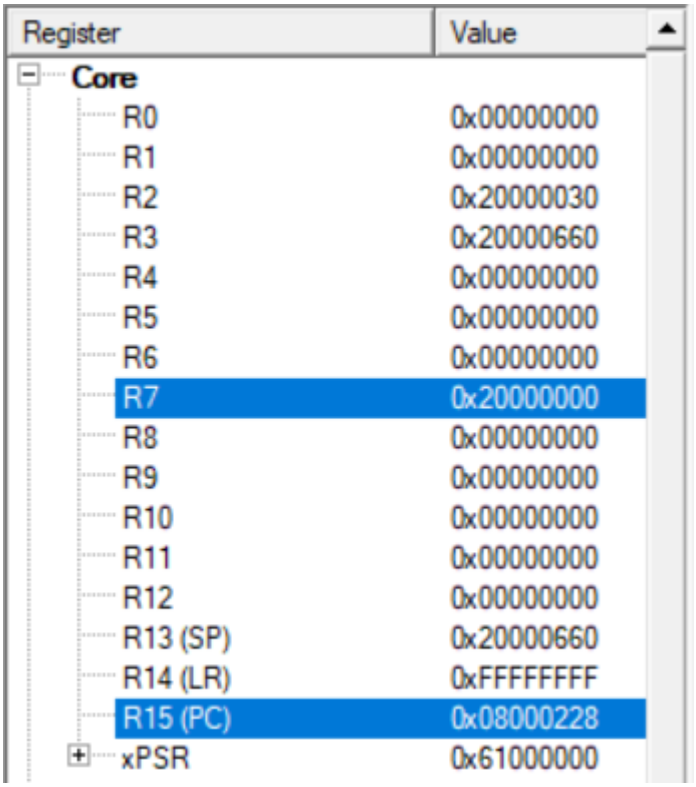
1 2 3 — 1 4 7 10
4 5 6 — So, our transposed matrix should be 2 5 8 11
7 8 9 — 3 6 9 12
10 11 12

Program execution:

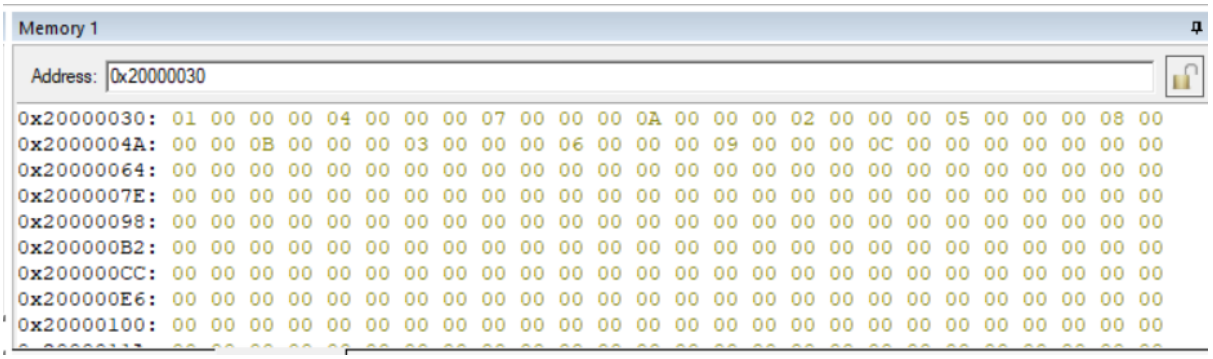
First we get 1 from matrix array and set it to first element of transposed matrix. Then in loop we added 12 to address of matrix array and get 4. And set it to second element of transposed matrix. Then set 7 and 10. Inner loop finishes and returns to outer loop. For the next iteration of inner loop, outer loop makes some arrangement. For the next iteration it adds 4 to starting address of matrix which is kept in R3. With this arrangement in inner loop it gets 2 first and set to transposed matrix address where it lefts. Continues by getting 5 8 11 and sets to transposed matrix.

```
24  __main  PROC
25      MOV R0, #0  ; i
26      MOV R1, #0  ; j
27      LDR R2, =transpose_matrix
28      LDR R7, =matrix
29      ADD R7, R7, #-4
30
31  OUTER_LOOP
32      CMP R0, #3  ; i != 3
33      BEQ DONE
34      ADD R0, R0, #1  ; i = i + 1
35      MOV R1, #0  ; j = 0
36      ADD R7, R7, #4
37      MOV R3, R7
38  INNER_LOOP
39      CMP R1, #4
40      BEQ OUTER_LOOP
41      LDR R4, [R3], #12
42      STR R4, [R2], #4
43      ADD R1, R1, #1  ; j = j + 1
44      B INNER_LOOP
45  DONE
46  stop    B       stop
47      ENDP
```

When we check registers and memories
Before outer loop:



Address of transposed matrix which is kept in R2 is 0x20000030. Starting address of matrix is 0x20000000. When program finishes transposed matrix can be seen in memory as:



It keeps transposed matrix as 1, 4, 7, A, 2, 5, 8, B, 3, 6, 9, C