Marmara University - Faculty of Engineering

Department of Computer Engineering

CSE4219 Principles of Embedded System Design (Fall 2024)

Submit Date: 03/11/2024.

Arm Cortex M4 Problems

Student Number (ID)	Name	Surname
150120998	Abdelrahman	Zahran
150120997	Mohamed Nael	Ayoubi
130319659	Cihan	Erdoğanyılmaz

Sections Of the Report: -

- Section (1): Problem (1) - ARM Assembly Program for Repeated Digit Summation
- Section (2): Problem (2) - ARM Assembly Program for Matrix Column Swap
- Section (3): Problem (3) - ARM Assembly Program for Error Correcting Code (ECC)

All Inputs are given to programs as Data Memory Inputs – For Implementation Simplicity Purposes!!

This method is common in embedded systems, assembly language programming, and certain low-level software development contexts. It simplifies the process by avoiding complex input/output handling at runtime, enabling the program to fetch inputs directly from specified memory addresses in a predictable, structured way.

Section (1): Problem (1) - ARM Assembly Program for Repeated Digit Summation

In this assembly program, we aim to calculate the value of the function F(a,n)=a+aa+aaa+…F(a, n) = a + aa + aaa + … up to nnn terms based on the input arguments a and n. We begin by defining our data section, which holds the input values for a and n.

We initialize a register to hold the total sum and then load the values of a and n from memory into registers for further processing. A loop counter is initialized to zero to track the number of terms processed.

Inside the loop, we compute the new term by multiplying the accumulated sum by 10 and adding the current value of a. This approach builds the terms in the required sequence (e.g., 3, 33, 333) efficiently without needing to construct each term explicitly. We also check if the loop counter has reached n; if so, we exit the loop.

Once all terms have been processed, the final sum is stored in the designated register (r0). This structured approach allows for a clear and efficient calculation of the expression using basic arithmetic operations in assembly language.

Inputs & Outputs:

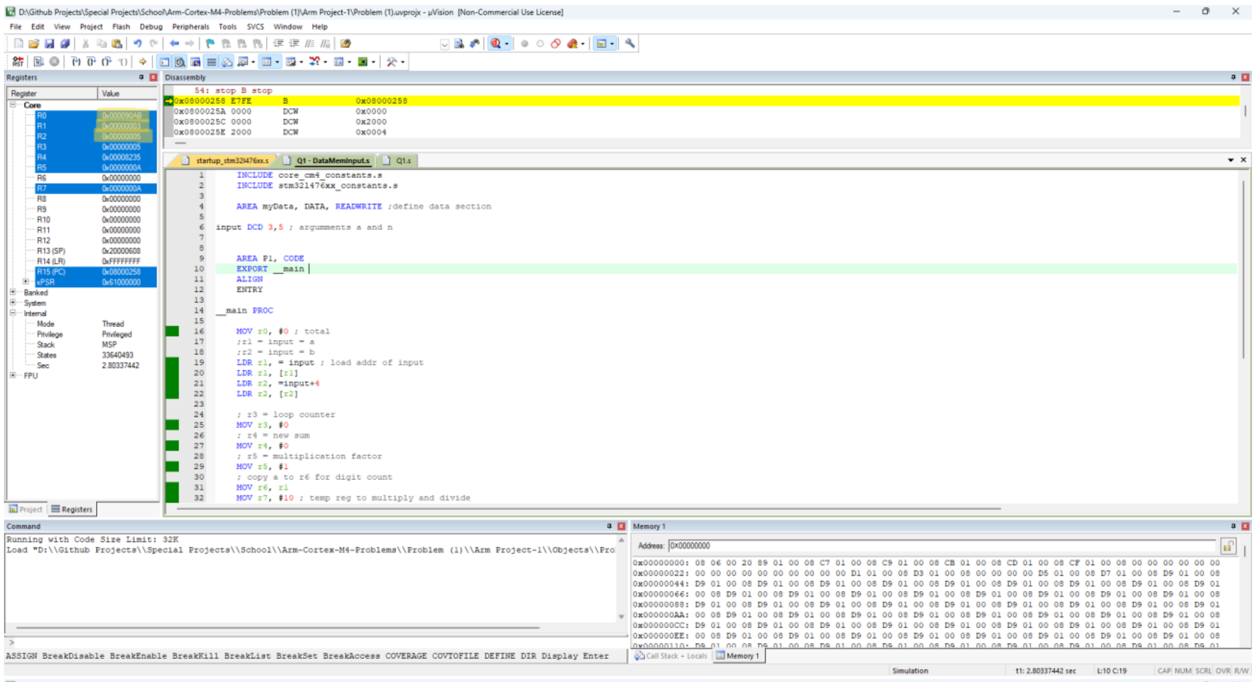
1) A = 3, B = 5 → Output: 37035 (In Hex.)

2) A = 45, B = 3 → Output: 459135 (In Hex.)

Program Source Code:

```
    INCLUDE core_cm4_constants.s
    INCLUDE stm32l476xx_constants.s

    AREA myData, DATA, READWRITE ;define data section

input DCD 45,3 ; argumments a and n


    AREA P1, CODE
    EXPORT __main
    ALIGN
    ENTRY

__main PROC

    MOV r0, #0 ; total
    ;r1 = input = a
    ;r2 = input = b
    LDR r1, = input ; load addr of input
    LDR r1, [r1]
    LDR r2, =input+4
    LDR r2, [r2]

    ; r3 = loop counter
    MOV r3, #0
    ; r4 = new sum
    MOV r4, #0
    ; r5 = multiplication factor
    MOV r5, #1
    ; copy a to r6 for digit count
    MOV r6, r1
    MOV r7, #10 ; temp reg to multiply and divide

count_digits
    CMP r6, #0
    BEQ loop ; no mmore digits left begin main loop
    MUL r5, r5, r7 ; multp factor increased
    UDIV r6, r6, r7
    B count_digits

loop
    ; sum = sum * 10 + a
    ; total = total + sum
    CMP r2, r3
    BEQ stop ; if b equals counter branch to stop execution

    MLA r4, r4, r5, r1; multiply with accumulate
    ADD r0, r0, r4

    ADD r3, r3, #1
    B loop


stop B stop
    ENDP
    END
```

Screenshots:



D:\Github Projects\Special Projects\School\Arm-Cortex-M4-Problems\Problem (1)\Arm Project-1\Problem (1).uvprojx - µVision [Non-Commercial Use License]

Register	Value
Core	
R0	0x000090AB
R1	0x00000003
R2	0x00000005
R3	0x00000005
R4	0x00008235
R5	0x0000000A
R6	0x00000000
R7	0x0000000A
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000608
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000258
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	33640493
Sec	2.80337442
FPU	

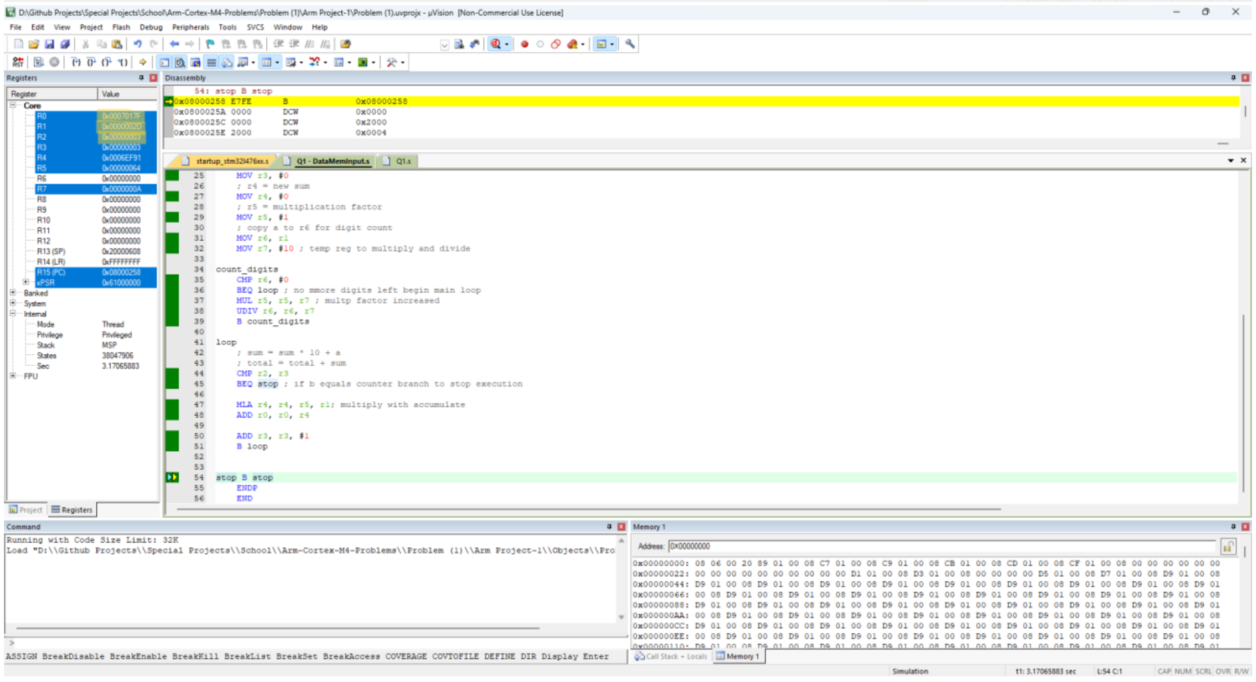
```
    54: stop B stop
0x08000258 E7FE      B          0x08000258
0x0800025A 0000      DCW        0x0000
0x0800025C 0000      DCW        0x2000
0x0800025E 2000      DCW        0x0004
```

```
    INCLUDE core_cm4_constants.s
    INCLUDE stm32l476xx_constants.s

    AREA myData, DATA, READWRITE ;define data section

input DCD 3,5 ; argumments a and n


    AREA P1, CODE
    EXPORT __main
    ALIGN
    ENTRY

__main PROC

    MOV r0, #0 ; total
    ;r1 = input = a
    ;r2 = input = b
    LDR r1, = input ; load addr of input
    LDR r1, [r1]
    LDR r2, =input+4
    LDR r2, [r2]

    ; r3 = loop counter
    MOV r3, #0
    ; r4 = new sum
    MOV r4, #0
    ; r5 = multiplication factor
    MOV r5, #1
    ; copy a to r6 for digit count
    MOV r6, r1
    MOV r7, #10 ; temp reg to multiply and divide
```

```
Running with Code Size Limit: 32K
Load "D:\\Github Projects\\Special Projects\\School\\Arm-Cortex-M4-Problems\\Problem (1)\\Arm Project-1\\Objects\\Pro
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter
```

```
0x00000000: 08 06 00 20 89 01 00 08 C7 01 00 08 C9 01 00 08 CB 01 00 08 CD 01 00 08 CF 01 00 08 00 00 00 00 00 00
0x00000022: 00 00 00 00 00 00 00 00 00 00 D1 01 00 08 D3 01 00 08 00 00 00 00 D5 01 00 08 D7 01 00 08 D9 01 00 08
0x00000044: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x00000066: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
0x00000088: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x000000AA: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
0x000000CC: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x000000EE: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
```

Simulation   t1: 2.80337442 sec   L:10 C:19   CAP NUM SCRL OVR R/W



D:\Github Projects\Special Projects\School\Arm-Cortex-M4-Problems\Problem (1)\Arm Project-1\Problem (1).uvprojx - µVision [Non-Commercial Use License]

Register	Value
Core	
R0	0x0007017F
R1	0x0000002D
R2	0x00000003
R3	0x00000003
R4	0x0006EF91
R5	0x00000064
R6	0x00000000
R7	0x0000000A
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000608
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000258
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	38047906
Sec	3.17065883
FPU	

```
    54: stop B stop
0x08000258 E7FE      B          0x08000258
0x0800025A 0000      DCW        0x0000
0x0800025C 0000      DCW        0x2000
0x0800025E 2000      DCW        0x0004
```

```
    MOV r3, #0
    ; r4 = new sum
    MOV r4, #0
    ; r5 = multiplication factor
    MOV r5, #1
    ; copy a to r6 for digit count
    MOV r6, r1
    MOV r7, #10 ; temp reg to multiply and divide

count_digits
    CMP r6, #0
    BEQ loop ; no mmore digits left begin main loop
    MUL r5, r5, r7 ; multp factor increased
    UDIV r6, r6, r7
    B count_digits

loop
    ; sum = sum * 10 + a
    ; total = total + sum
    CMP r2, r3
    BEQ stop ; if b equals counter branch to stop execution

    MLA r4, r4, r5, r1; multiply with accumulate
    ADD r0, r0, r4

    ADD r3, r3, #1
    B loop


stop B stop
    ENDP
    END
```

```
Running with Code Size Limit: 32K
Load "D:\\Github Projects\\Special Projects\\School\\Arm-Cortex-M4-Problems\\Problem (1)\\Arm Project-1\\Objects\\Pro
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter
```

```
0x00000000: 08 06 00 20 89 01 00 08 C7 01 00 08 C9 01 00 08 CB 01 00 08 CD 01 00 08 CF 01 00 08 00 00 00 00 00 00
0x00000022: 00 00 00 00 00 00 00 00 00 00 D1 01 00 08 D3 01 00 08 00 00 00 00 D5 01 00 08 D7 01 00 08 D9 01 00 08
0x00000044: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x00000066: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
0x00000088: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x000000AA: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
0x000000CC: D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01
0x000000EE: 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08 D9 01 00 08
```

Simulation   t1: 3.17065883 sec   L:54 C:1   CAP NUM SCRL OVR R/W

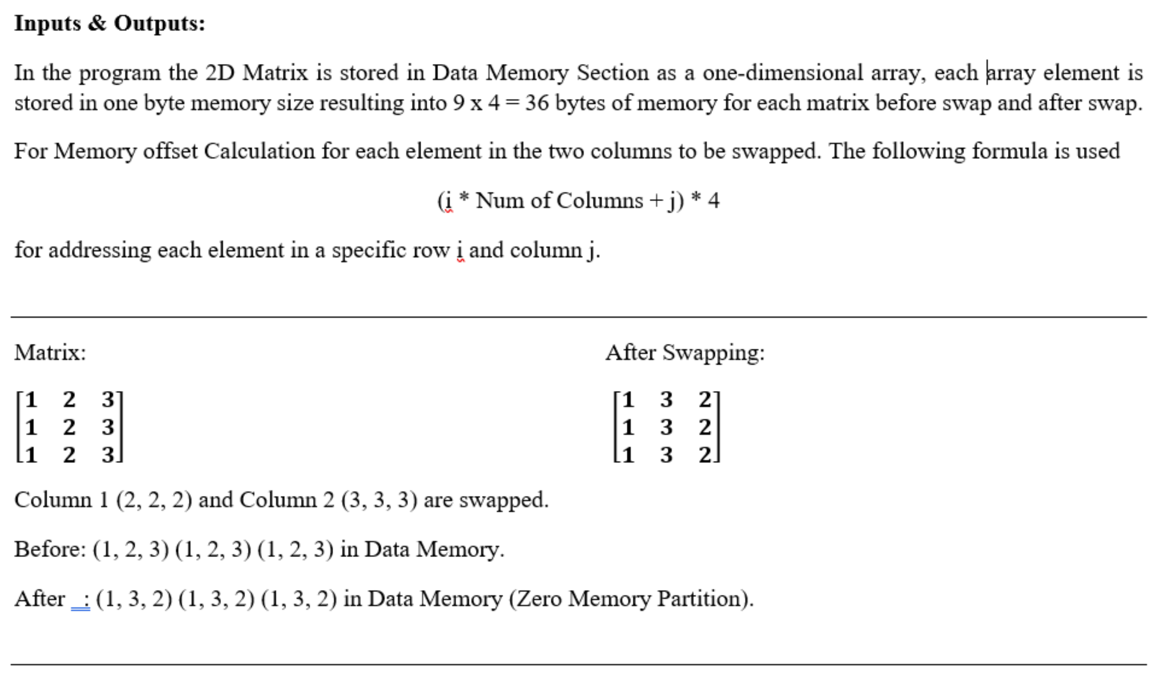
Section (2): Problem (2) - ARM Assembly Program for Matrix Column Swap

In this assembly program, our objective is to swap two columns in a 3x3 integer matrix. We start by defining the data section, which includes the matrix initialized with some integer values and a reserved block of memory to store the modified matrix after the swap.

We initialize registers to hold the column indices to be swapped and load the matrix address and the destination memory address for the swapped result. The column indices are converted to the corresponding memory addresses based on the row-major storage of the matrix.

A loop is set up to iterate through the rows of the matrix. For each row, we calculate the memory addresses for the elements in the two columns that need to be swapped. The elements are then loaded from the source matrix, and the swapped values are stored in the destination memory.

This process repeats until all rows have been processed. Once the loop is complete, the program ends with the swapped matrix stored in the designated zeroed memory block. This implementation efficiently handles the column swap using basic arithmetic operations and memory addressing in assembly language.



Inputs & Outputs:

In the program the 2D Matrix is stored in Data Memory Section as a one-dimensional array, each array element is stored in one byte memory size resulting into 9 x 4 = 36 bytes of memory for each matrix before swap and after swap.

For Memory offset Calculation for each element in the two columns to be swapped. The following formula is used

$$(i * \text{Num of Columns} + j) * 4$$

for addressing each element in a specific row i and column j.

Matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

After Swapping:

$$\begin{bmatrix} 1 & 3 & 2 \\ 1 & 3 & 2 \\ 1 & 3 & 2 \end{bmatrix}$$

Column 1 (2, 2, 2) and Column 2 (3, 3, 3) are swapped.

Before: (1, 2, 3) (1, 2, 3) (1, 2, 3) in Data Memory.

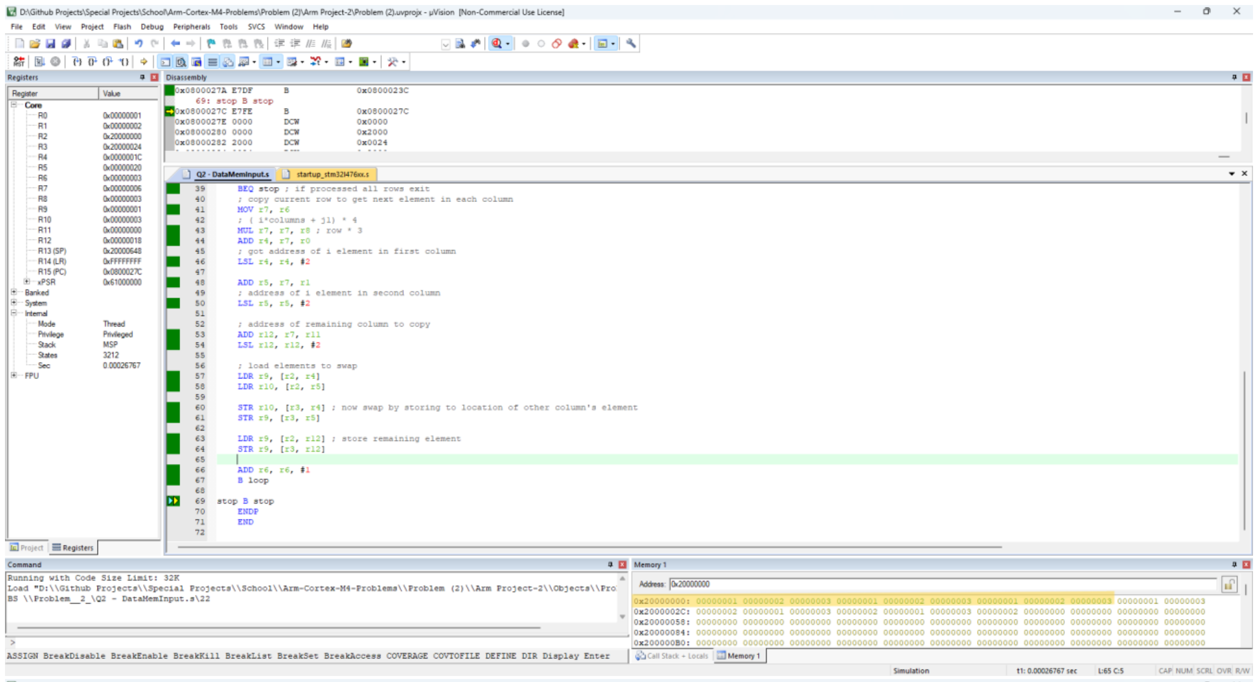
After _: (1, 3, 2) (1, 3, 2) (1, 3, 2) in Data Memory (Zero Memory Partition).

Program Source Code:

```
    INCLUDE core_cm4_constants.s
    INCLUDE stm32l476xx_constants.s

    AREA myData, DATA, READWRITE ;data section

Matrix DCD 1,2,3,1,2,3,1,2,3 ;row major each is a word for integer so total 36
bytes
zMem   SPACE 36 ;reserving zeroed memory in data to store the modified matrix
```

```
    AREA P2, CODE
    EXPORT __main
    ALIGN
    ENTRY

__main PROC

    ; input which columns to swap in 3x3 matrix
    MOV r0, #1 ; col1 = 2
    MOV r1, #2 ; col2 = 1

    LDR r2, =Matrix ; get matrix address
    LDR r3, =zMem

    ; get indices
    MOV r4, r0 ; get col 1 first element index in the matrix (j1)
    MOV r5, r1 ; get col 2 first element index in the matrix (j2)

    MOV r6, #0 ; row counter (i)

    MOV r8, #3 ; to multiply later for next element address

    ; remainin column
    ADD r11, r0, r1
    SUB r11, r8, r11

    ; loop until no rows left
loop
    CMP r6, #3
    BEQ stop ; if processed all rows exit
    ; copy current row to get next element in each column
    MOV r7, r6
    ; ( i*columns + j1) * 4
    MUL r7, r7, r8 ; row * 3
    ADD r4, r7, r0
    ; got address of i element in first column
    LSL r4, r4, #2

    ADD r5, r7, r1
    ; address of i element in second column
    LSL r5, r5, #2

    ; address of remaining column to copy
    ADD r12, r7, r11
    LSL r12, r12, #2

    ; load elements to swap
    LDR r9, [r2, r4]
    LDR r10, [r2, r5]

    STR r10, [r3, r4] ; now swap by storing to location of other column's element
    STR r9, [r3, r5]

    LDR r9, [r2, r12] ; store remaining element
    STR r9, [r3, r12]

    ADD r6, r6, #1
    B loop

stop B stop
    ENDP
    END
```

Screenshots:



D:\Github Projects\Special Projects\School\Arm-Cortex-M4-Problems\Problem (2)\Arm Project-2\Problem (2).uvprojx - µVision [Non-Commercial Use License]

Register	Value
Core	
R0	0x00000001
R1	0x00000002
R2	0x20000000
R3	0x20000024
R4	0x0000001C
R5	0x00000020
R6	0x00000003
R7	0x00000006
R8	0x00000003
R9	0x00000001
R10	0x00000003
R11	0x00000000
R12	0x00000018
R13 (SP)	0x20000648
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800027C
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	3212
Sec	0.00026767
FPU	

```
0x0800027A E7DF      B              0x0800023C
    69: stop B stop
0x0800027C E7FE      B              0x0800027C
0x0800027E 0000      DCW            0x0000
0x08000280 0000      DCW            0x2000
0x08000282 2000      DCW            0x0024
```

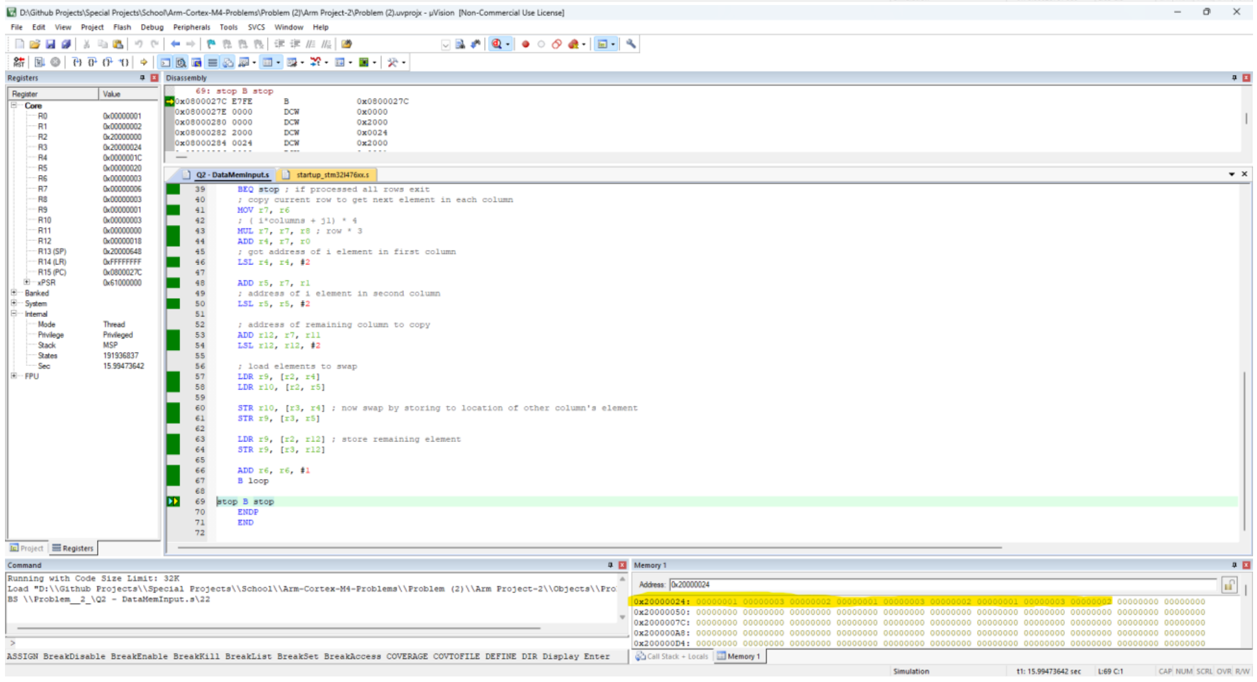
```
    BEQ stop ; if processed all rows exit
    ; copy current row to get next element in each column
    MOV r7, r6
    ; ( i*columns + j1) * 4
    MUL r7, r7, r8 ; row * 3
    ADD r4, r7, r0
    ; got address of i element in first column
    LSL r4, r4, #2

    ADD r5, r7, r1
    ; address of i element in second column
    LSL r5, r5, #2

    ; address of remaining column to copy
    ADD r12, r7, r11
    LSL r12, r12, #2

    ; load elements to swap
    LDR r9, [r2, r4]
    LDR r10, [r2, r5]

    STR r10, [r3, r4] ; now swap by storing to location of other column's element
    STR r9, [r3, r5]

    LDR r9, [r2, r12] ; store remaining element
    STR r9, [r3, r12]

    ADD r6, r6, #1
    B loop

stop B stop
    ENDP
    END
```

```
Running with Code Size Limit: 32K
Load "D:\\Github Projects\\Special Projects\\School\\Arm-Cortex-M4-Problems\\Problem (2)\\Arm Project-2\\Objects\\Pro
BS \\Problem__2_\Q2 - DataMemInput.s\22

>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter
```

Address: 0x20000000

```
0x20000000: 00000001 00000002 00000003 00000001 00000002 00000003 00000001 00000002 00000003 00000001 00000003
0x2000002C: 00000002 00000001 00000003 00000002 00000001 00000003 00000002 00000000 00000000 00000000 00000000
0x20000058: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000084: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Simulation   t1: 0.00026767 sec   L:65 C:5   CAP NUM SCRL OVR R/W



D:\Github Projects\Special Projects\School\Arm-Cortex-M4-Problems\Problem (2)\Arm Project-2\Problem (2).uvprojx - µVision [Non-Commercial Use License]

Register	Value
Core	
R0	0x00000001
R1	0x00000002
R2	0x20000000
R3	0x20000024
R4	0x0000001C
R5	0x00000020
R6	0x00000003
R7	0x00000006
R8	0x00000003
R9	0x00000001
R10	0x00000003
R11	0x00000000
R12	0x00000018
R13 (SP)	0x20000648
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800027C
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	191936837
Sec	15.99473642
FPU	

```
    69: stop B stop
0x0800027C E7FE      B              0x0800027C
0x0800027E 0000      DCW            0x0000
0x08000280 0000      DCW            0x2000
0x08000282 2000      DCW            0x0024
0x08000284 0024      DCW            0x2000
```

```
    BEQ stop ; if processed all rows exit
    ; copy current row to get next element in each column
    MOV r7, r6
    ; ( i*columns + j1) * 4
    MUL r7, r7, r8 ; row * 3
    ADD r4, r7, r0
    ; got address of i element in first column
    LSL r4, r4, #2

    ADD r5, r7, r1
    ; address of i element in second column
    LSL r5, r5, #2

    ; address of remaining column to copy
    ADD r12, r7, r11
    LSL r12, r12, #2

    ; load elements to swap
    LDR r9, [r2, r4]
    LDR r10, [r2, r5]

    STR r10, [r3, r4] ; now swap by storing to location of other column's element
    STR r9, [r3, r5]

    LDR r9, [r2, r12] ; store remaining element
    STR r9, [r3, r12]

    ADD r6, r6, #1
    B loop

stop B stop
    ENDP
    END
```

```
Running with Code Size Limit: 32K
Load "D:\\Github Projects\\Special Projects\\School\\Arm-Cortex-M4-Problems\\Problem (2)\\Arm Project-2\\Objects\\Pro
BS \\Problem__2_\Q2 - DataMemInput.s\22

>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter
```

Address: 0x20000024

```
0x20000024: 00000001 00000003 00000002 00000001 00000003 00000002 00000001 00000003 00000002 00000000 00000000
0x20000050: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x2000007C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000A8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Simulation   t1: 15.99473642 sec   L:69 C:1   CAP NUM SCRL OVR R/W

Section (3): Problem (3) - ARM Assembly Program for Error Correcting Code (ECC)

In this assembly program, we start by defining our data section, which includes our 8-bit input data and a reserved zeroed memory location for storing the expanded result.

First, we load the 8-bit input value into a register and extract the bits into a 13-bit register by shifting the original value multiple times. This approach simplifies the process and avoids the need to access data memory repeatedly for bit manipulation.

Next, we apply a masking technique to isolate the bits relevant for each parity bit. For each required parity bit, we perform an AND operation between the current register (which contains the original data bits) and the mask corresponding to the parity bit position. This allows us to check only the bits that the parity bit is responsible for.

To compute the parity bits, we utilize an XOR operation to simulate even counting. By XORing each bit in the masked registers and shifting them right, we accumulate the count of 1s into a temporary register (r9). After processing all bits, we check the parity result stored in r9. If the count is odd, we set the corresponding parity bit to 1 by performing an OR operation with the expanded data register (r3). Finally, we store the complete 13-bit expanded data, including the original data bits and the calculated parity bits, into our reserved memory location.

This method ensures an efficient calculation of the error-correcting codes while maintaining clarity and simplicity in the implementation.

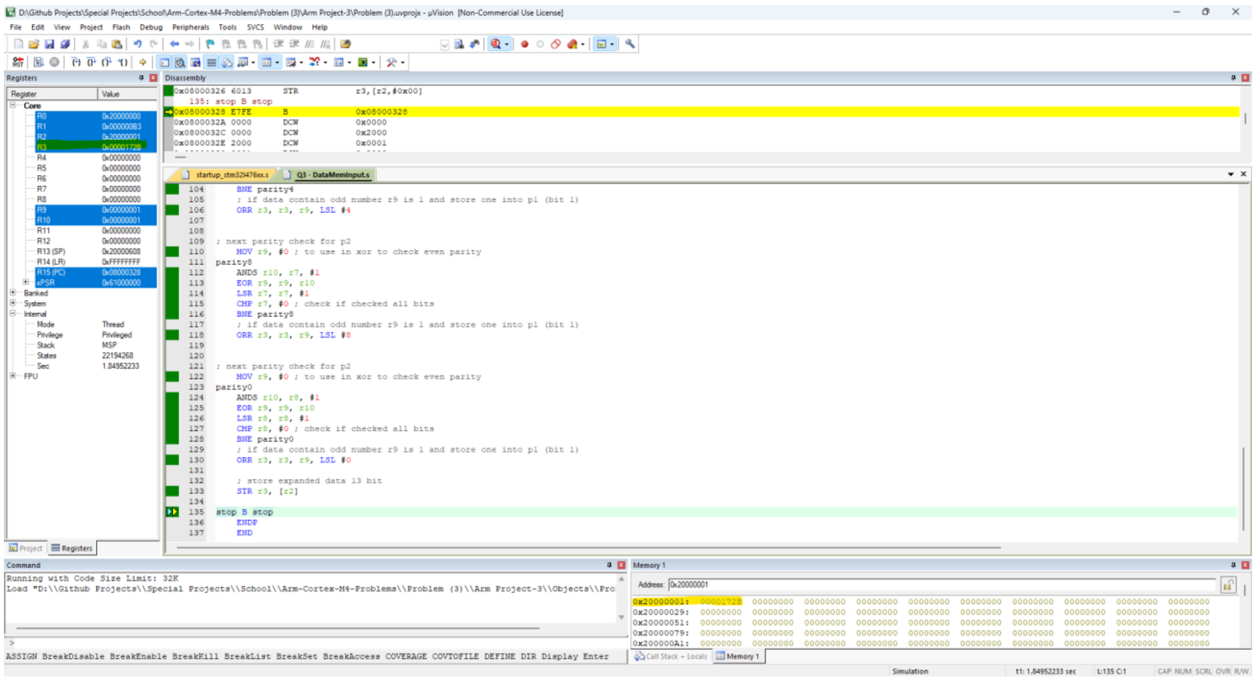
Program Source Code:

```
    INCLUDE core_cm4_constants.s
    INCLUDE stm32l476xx_constants.s

    AREA myData, DATA, READWRITE ; data section

input DCB 0xB3 ; 8 bit data input
zMem SPACE 4    ; reserving zeroed memory in data to store the expanded data


    AREA P3, CODE
    EXPORT __main
    ALIGN
    ENTRY

__main PROC

    LDR r0, =input
    MOV r1, #0
    LDRB r1, [r0] ; load input data to a register
    LDR r2, =zMem

    MOV r3, #0 ; 13-bit extended

    ; extract bits and store inside r3

    ; bit 7 to 12
    MOV r4, r1, LSR #7
    ORR r3, r3, r4, LSL #12
    ; bit 6 to 11
    MOV r4, r1, LSR #6
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #11
    ; bit 5 to 10
    MOV r4, r1, LSR #5
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #10
    ; bit 4 to 9
    MOV r4, r1, LSR #4
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #9
    ; bit 3 to 7
    MOV r4, r1, LSR #3
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #7
    ; bit 2 to 6
    MOV r4, r1, LSR #2
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #6
    ; bit 1 to 5
    MOV r4, r1, LSR #1
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #5
    ; bit 0 to 3
    MOV r4, r1
    AND r4, r4, #1 ; mask LSB
    ORR r3, r3, r4, LSL #3

    ; masked parity reg to check correspnding bits
    LDR r4, =0x0AA8 ; p1
    LDR r5, =0x0CC8 ; p2
    LDR r6, =0x10E0 ; p4
    LDR r7, =0x1E00 ; p8
    LDR r8, =0x1FFE ; p0

    AND r4, r3, r4 ; its for checking p1
    AND r5, r3, r5 ; p2
    AND r6, r3, r6 ; P4
    AND r7, r3, r7 ; p8
    AND r8, r3, r8 ; p0

    MOV r9, #0 ; to use in xor to check even parity
```

```
parity1
    ANDS r10, r4, #1
    EOR r9, r9, r10
    LSR r4, r4, #1
    CMP r4, #0 ; check if checked all bits
    BNE parity1
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #1


    ; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity2
    ANDS r10, r5, #1
    EOR r9, r9, r10
    LSR r5, r5, #1
    CMP r5, #0 ; check if checked all bits
    BNE parity2
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #2

; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity4
    ANDS r10, r6, #1
    EOR r9, r9, r10
    LSR r6, r6, #1
    CMP r6, #0 ; check if checked all bits
    BNE parity4
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #4


; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity8
    ANDS r10, r7, #1
    EOR r9, r9, r10
    LSR r7, r7, #1
    CMP r7, #0 ; check if checked all bits
    BNE parity8
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #8
```

```
; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity0
    ANDS r10, r8, #1
    EOR r9, r9, r10
    LSR r8, r8, #1
    CMP r8, #0 ; check if checked all bits
    BNE parity0
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #0

    ; store expanded data 13 bit
    STR r3, [r2]

stop B stop
    ENDP
    END
```

Screenshots:



D:\Github Projects\Special Projects\School\Arm-Cortex-M4-Problems\Problem (3)\Arm Project-3\Problem (3).uvprojx - µVision [Non-Commercial Use License]

Registers

Register	Value
Core	
R0	0x20000000
R1	0x000000B3
R2	0x20000001
R3	0x0000172B
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000001
R10	0x00000001
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000608
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000328
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	22194268
Sec	1.84952233
FPU	

Disassembly

```
0x08000326 6013       STR          r3,[r2,#0x00]
   135: stop B stop
0x08000328 E7FE       B            0x08000328
0x0800032A 0000       DCW          0x0000
0x0800032C 0000       DCW          0x2000
0x0800032E 2000       DCW          0x0001
```

startup_stm32l476xx.s | Q3 - DataMemInput.s

```
    BNE parity4
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #4


; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity8
    ANDS r10, r7, #1
    EOR r9, r9, r10
    LSR r7, r7, #1
    CMP r7, #0 ; check if checked all bits
    BNE parity8
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #8


; next parity check for p2
    MOV r9, #0 ; to use in xor to check even parity
parity0
    ANDS r10, r8, #1
    EOR r9, r9, r10
    LSR r8, r8, #1
    CMP r8, #0 ; check if checked all bits
    BNE parity0
    ; if data contain odd number r9 is 1 and store one into p1 (bit 1)
    ORR r3, r3, r9, LSL #0

    ; store expanded data 13 bit
    STR r3, [r2]

stop B stop
    ENDP
    END
```

Command

```
Running with Code Size Limit: 32K
Load "D:\\Github Projects\\Special Projects\\School\\Arm-Cortex-M4-Problems\\Problem (3)\\Arm Project-3\\Objects\\Pro
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter
```

Memory 1

Address: 0x20000001

```
0x20000001: 0000172B 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000029: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000051: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000079: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000A1: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Simulation   t1: 1.84952233 sec   L:135 C:1