

ELM 334 - Homework #2

Ömer Emre Polat
1801022037



A. Problem 0

Given Github files are created in a subfolder and compiled using mingw and makefile.

```
C:\WINDOWS\system32\cmd.exe

C:\Users\user\Desktop\New folder>make
gcc main.o banana.o -lm -o banana
Successfully finished...

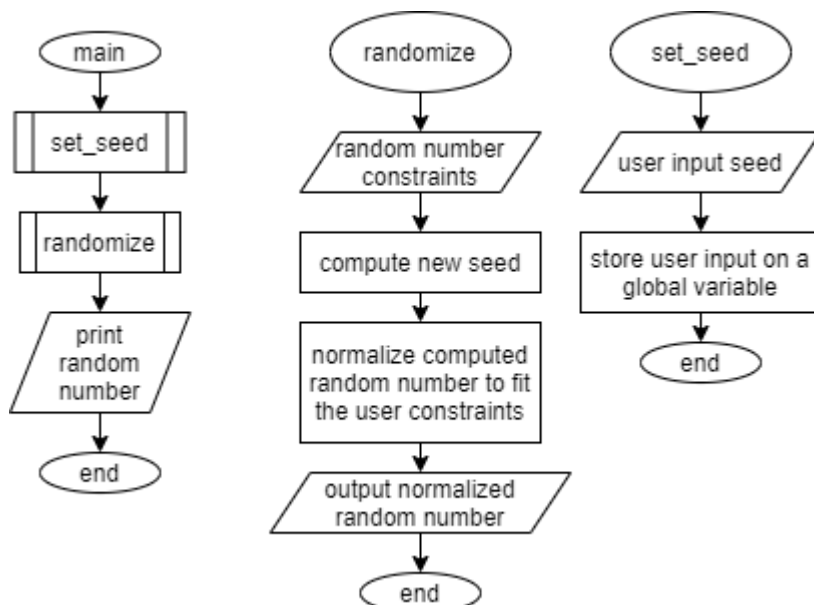
C:\Users\user\Desktop\New folder>banana
main.c: Hello from main
banana.c: Calculating Rosenbrock's banana function with a: 1.000, b:100.000..
main.c: Result for x: -1.9, y: 2.4 is: 154.820
banana.c: Calculating Rosenbrock's banana function with a: 1.000, b:1.000..
main.c: Result for x: -1.9, y: 2.4 is: 9.874

C:\Users\user\Desktop\New folder>
```

Code works as expected and gives the random numbers.

B. Problem 1

First we create a flowchart of the code.



This flowchart shows the general working of the random number generator code.

Now we write up the code using the flowchart as a helping tool. Code was written using the pseudo random number generator algorithm found on Wikipedia. This algorithm has a c constant in it which was picked randomly.

myrand.h

```
#ifndef _MYRAND_H
#define _MYRAND_H

void set_seed(int);

int randomize(int, int);

#endif
```

myrand.c

```
#include <stdio.h>

#include "myrand.h"

static const int a = 1103515245;
static const int c = 1234567;
static const int m = 2147483647;

static int seed = 1;
static int random = 0;

void set_seed(int s)
{
    seed = s;
}

int randomize(int min, int max)
{
    //uses a random number generating algorithm
    seed = ((a * seed) + c) % m;

    //normalizes values for the size
    random = (seed % (max));

    //normalizes the values to fit the minimum values and makes the negative values from the remainder o
perator positive values
    return ((random) < 0 ? (0 - (random)) + min : (random) + min);
}
```

main.c

```
#include <stdio.h>

#include "myrand.h"

int main()
{
    set_seed(1);

    printf("random number: %d", randomize(1, 15));
    return 0;
}
```

With the codes written we can test it using the makefile template given to us.

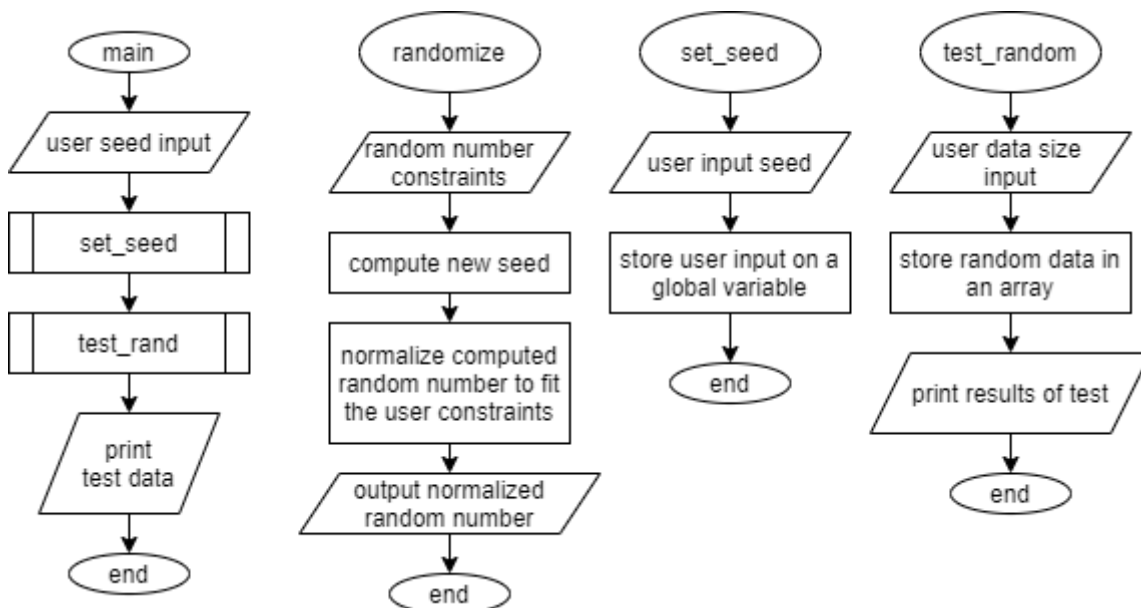
```
C:\Users\user\Desktop\micro\random_numbers>make
gcc main.o myrand.o -lm -o myrand
Successfully finished...

C:\Users\user\Desktop\micro\random_numbers>myrand
random number: 8
```

Our function gives an output of “8”.

C. Problem 2

Now we create a flowchart for the test code.



Using the flowchart we can write the program code and test it. Test code works by running randomize function by a number that's set and counts every number that comes out of the randomize function then prints the counted numbers so that we can see the distribution of values.

test.h

```
#ifndef _TEST_RANDOM_H
#define _TEST_RANDOM_H

void test_random(int);

#endif
```

test.c

```
#include <stdio.h>

#include "test_random.h"
#include "myrand.h"

void test_random(int data_size)
{
    int counter[15] = {0};

    for(int i=0; i<data_size; i++)
    {
        int x = randomize(1, 15);
        counter[x-1]++;
    }

    for(int i=0; i<15; i++)
    {
        printf("%d: %d\n", i+1, counter[i]);
    }
}
```

main.c

```
#include <stdio.h>

#include "myrand.h"
#include "test_random.h"

int main()
{
    set_seed(13);

    test_random(150000);

    return 0;
}
```

Now we test the code for the distribution of numbers with a data size of 150000 numbers.

```
C:\Users\user\Desktop\micro\random_numbers>myrand
1: 10005
2: 10187
3: 9985
4: 9991
5: 9905
6: 9969
7: 9969
8: 9874
9: 10075
10: 10162
11: 9899
12: 9896
13: 9935
14: 10102
15: 10046
```

As we can see the numbers are relatively uniformly distributed.

D. Problem 3

```
ldr r5, [r6, #4]
mvns r4, r4
ands r5, r5, r4
adds r0, r0, r1
//////////
add r0, r0, r1
subs r2, r4, #2
asrs r2, r4, #21
str r5, [r6, r1]
//////////
bx lr
bne 0x12
```

Decoding this assembly code into thumb gives us:

0110 1000 0111 0101

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn			Rt		

0100 0011 1110 0100

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

0100 0000 0010 0101

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

0001 1000 0100 0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

//////////

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0					Rm		Rdn	

DN└

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1				imm3		Rn		Rd	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0						imm5		Rm		Rd	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0				Rm		Rn		Rt	

////////////////////////////////////

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0				Rm		(0)(0)(0)	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1							cond				imm8	

Converting these into hexadecimal we get:

```
0x6875
0x43E4
0x4025
0x1840
////////////////////////////////
0x4408
0x1EA2
0x1562
0x5075
////////////////////////////////
0x4770
0xD107
```

E. Problem 4

Now we make a table for all the cycle times.

ldr r5, [r6, #4]	1
mvns r4, r4	1
ands r5, r5, r4	1
adds r0, r0, r1	1
add r0, r0, r1	1
subs r2, r4, #2	1
asrs r2, r4, #21	1

str r5, [r6, r1]	2
bx lr	2
bne 0x12	1
Total = 12	

<div>B <immed></div> <div>BL <immed></div> <div>BLX <immed></div>	1	<div>Assumes successful dynamic prediction. Some dynamically predicted branches may be folded, to be zero cycles.</div> <div>+3 for successful static prediction.</div> <div>+4 for unsuccessful static or dynamic prediction. In this case the flags are required two cycles early.</div>
---	---	--

Instruction cycle times are taken from ARM11 (ARMv6) textbook.

F. Problem 5

This problem requires for an empty loop with enough cycles for it to reach a given number in seconds. This can be done using NOP inside the loop and we can multiply the given number in seconds with the cycle amount that takes 1 seconds to create a delay function that can create delays as long as the given number in seconds.

```

Stack_Size      EQU      0x00000400

                AREA     STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE    Stack_Size
__initial_sp


                THUMB


                AREA     RESET, DATA, READONLY
                EXPORT   __Vectors

__Vectors
    DCD      __initial_sp          ; Top of Stack
    DCD      Reset_Handler        ; Reset Handler
    DCD      NMI_Handler          ; NMI Handler
    DCD      HardFault_Handler    ; Hard Fault Handler


                AREA     |.text|, CODE, READONLY

; nmi handler

```

```
NMI_Handler    PROC
    EXPORT  NMI_Handler
    B .
ENDP
```

```
; hardfault handler
```

```
HardFault_Handler  PROC
    EXPORT  HardFault_Handler
    B .
ENDP
```

```
; entry function
```

```
Reset_Handler    PROC
    EXPORT  Reset_Handler
    ; //////////////////////////////////
```

```
    MOVS r0, #0x2 ; given number in seconds
    LDR r1,=0x3D0900
    MULS r0, r1, r0
```

```
delay_loop
```

```
    SUBS r0, #0x1
    BNE delay_loop
    NOP
```

```
; //////////////////////////////////
```

```
B .
ENDP
```

```
END
```


Register	Value
Core	
R0	0x00000000
R1	0x003D0900
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000020
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	24000004
Sec	2.00000033


```

51:      B .
0x08000020 E7FE      B      0x08000020
0x08000022 0000      DCW      0x0000
0x08000024 0900      DCW      0x0900
0x08000026 003D      DCW      0x003D
0x08000028 0000      MOVS     r0,r0
0x0800002A 0000      MOVS     r0,r0

hw2_5.s
25      ENDP
26
27
28      ; hardfault handler
29      HardFault_Handler  PROC
30      EXPORT HardFault_Handler
31      B .
32      ENDP
33
34
35      ; entry function
36      Reset_Handler  PROC
37      EXPORT Reset_Handler
38      ; ///////////
39
40      MOVS r0, #0x2 ; given number in seconds
41      LDR r1,=0x3D0900
42      MULS r0, r1, r0
43
44      delay_loop
45
46      SUBS r0, #0x1
47      BNE delay_loop
48      NOP
49
50      ; ///////////
51      B .
52      ENDP
53
54      END
  
```

As we can see for a given number of 2 the time for execution takes around 2 seconds.

G. Problem 6

In this problem (assuming that we don't have to clock the peripherals) we have to find the base address for GPIOB then look for the ODR offset value. With these set up we can change the 12th bit to 1 by performing an OR operation with the value and a 0x1000. Same with turning the bit to low, we can use AND operation to turn off the certain bit. To calculate the delay loop cycle count we can use the 16MHz clock speed then divide it with instruction cycle count per loop.

$$\text{delay loop count} = \frac{16\text{MHz}}{1 + 1} = 8\text{M} = 0x7A1200$$

```

Stack_Size      EQU      0x00000400

                AREA     STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE     Stack_Size
__initial_sp
  
```

THUMB

AREA RESET, DATA, READONLY

EXPORT __Vectors

__Vectors

DCD __initial_sp ; Top of Stack

DCD Reset_Handler ; Reset Handler

DCD NMI_Handler ; NMI Handler

DCD HardFault_Handler ; Hard Fault Handler

AREA |.text|, CODE, READONLY

; nmi handler

NMI_Handler PROC

EXPORT NMI_Handler

B .

ENDP

; hardfault handler

HardFault_Handler PROC

EXPORT HardFault_Handler

B .

ENDP

; entry function

Reset_Handler PROC

EXPORT Reset_Handler

; ////////////

LED_ON ; branch to turn the led on

LDR r3, =(0x50000400 + 0x14)

LDR r2, [r3]

LDR r1, =0x1000

ORRS r2, r2, r1 ; or operation to set the given bit high

STR r2, [r3]

LDR r4,=0x7A1200 ; cycles for 1 sec total delay

ON_DELAY ; on time delay

CMP r4, #0x0

BEQ LED_OFF

```

        SUBS r4, r4, #1
        B ON_DELAY

LED_OFF ; branch to turn off the led

        LDR r3, =(0x50000400 + 0x14)
        LDR r2, [r3]
        LDR r1, =0xFFFFFFFF
        ANDS r2, r2, r1 ; and operation to set the given bit low
        STR r2, [r3]

        LDR r4,=0x249F60

OFF_DELAY ; off time delay

        CMP r4, #0x0
        BEQ LED_ON
        SUBS r4, r4, #1
        B OFF_DELAY

; ///////////
B .
ENDP

END

```

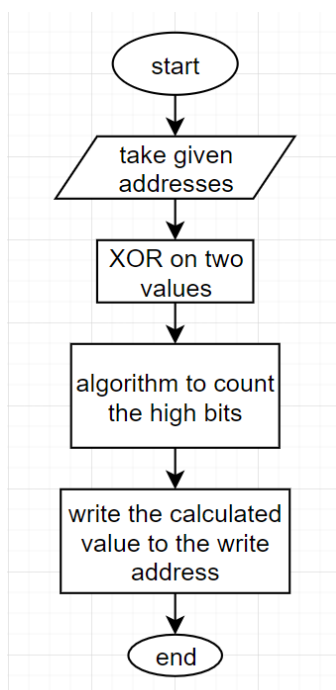
Now to simulate the code.

Register	Value
Core	
R0	0x00000000
R1	0x00001000
R2	0x00001000
R3	0x50000414
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000028
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	40000014
Sec	3.33333450


```

60:          LDR r3, =(0x50000400 + 0x14)
0x08000028 4B05      LDR      r3,[pc,#20] ; @0x08000040
61:          LDR r2, [r3]
0x0800002A 681A      LDR      r2,[r3,#0x00]
62:          LDR r1, =0x11110111
0x0800002C 4907      LDR      r1,[pc,#28] ; @0x0800004C
63:          ANDS r2, r2, r1 ; and operation to set the given bit low
64:          STR r2, [r3]
65:
66:          LDR r4,=0x249F60
67:
68: OFF_DELAY ; off time delay
69:
70:          CMP r4, #0x0
71:          BEQ LED_ON
72:          SUBS r4, r4, #1
73:          B OFF_DELAY
74:
75:          ; ///////////
76:          B .
77:          ENDP
78:
79:          END
  
```

H. Problem 7



After creating the flowchart we can use it as a template to write the program code.

To calculate the hamming distance we can use the XOR operation to find the different bits between two positions and count the different amount of bits to find the hamming distance.

```

Stack_Size      EQU      0x00000400

                AREA      STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem        SPACE    Stack_Size
__initial_sp


                THUMB


                AREA      RESET, DATA, READONLY
                EXPORT    __Vectors


__Vectors
    DCD      __initial_sp          ; Top of Stack
    DCD      Reset_Handler        ; Reset Handler
    DCD      NMI_Handler          ; NMI Handler
    DCD      HardFault_Handler    ; Hard Fault Handler


                AREA      |.text|, CODE, READONLY


; nmi handler
NMI_Handler      PROC
    EXPORT      NMI_Handler
    B .
    ENDP


; hardfault handler
HardFault_Handler  PROC
    EXPORT      HardFault_Handler
    B .
    ENDP


; entry function
Reset_Handler     PROC
    EXPORT      Reset_Handler
    ; ////////////

    LDR r0, =0x14224 ; setting the test values to the addresses
    LDR r1, =0x1A
    STR r1, [r0]

    LDR r0, =0x14228
    LDR r1, =0x19
    STR r1, [r0]

    LDR r0, =0x14224 ; address values for read and write
    LDR r1, =0x14228

```

```

    LDR r2, =0x1422C

    LDR r3, [r0] ; getting the values from given addresses
    LDR r4, [r1]

    EORS r3, r3, r4 ; performing XOR operation on two values

    LDR r5, =0x0 ; loop counter

kernighan_loop

    CMP r3, #0x0
    BEQ loop_stop
    SUBS r4, r3, #0x1
    ANDS r3, r3, r4
    ADDS r5, r5, #0x1
    B kernighan_loop

loop_stop

    STR r5, [r2] ; writing the hamming distance to given address

    ; ///////////
    B .
    ENDP

END

```

Now that we have written the code using Kernighan algorithm we can test the code with the test values.

Register	Value
Core	
R0	0x0014224
R1	0x0014228
R2	0x001422C
R3	0x0000003
R4	0x00000019
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x0014400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800002C
xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	21
Sec	0.00000175


```

0x0800002A 4063      EORS      r3,r3,r4
57:      LDR r5, =0x0 ; loop counter
58:
59: kernighan_loop
60:
0x0800002C 4D09      LDR      r5,[pc,#36] ; @0x08000054
61:      CMP r3, #0x0
62:      BEQ loop_stop
63:      SUBS r4, r3, #0x1
64:      ANDS r3, r3, r4
65:      ADDS r5, r5, #0x1
66:      B kernighan_loop
67:
68: loop_stop
69:
70:      STR r5, [r2] ; writing the hamming distance to given address
71:
72:      ; ///////////
73:      B .
74:      ENDP
75:
76:      END
  
```

After performing XOR operation on both test values (value1=11010, value2=11001) we get the value of 00011 on the r3 register. Now the code will perform the Kernighan loop to calculate the amount of high bits.

Register	Value
Core	
R0	0x00014224
R1	0x00014228
R2	0x0001422C
R3	0x00000000
R4	0x00000001
R5	0x00000002
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00014400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800003C
xPSR	0x61000000
Banked	
MSP	0x00014400
PSP	0x00000000
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	44
Sec	0.00000367

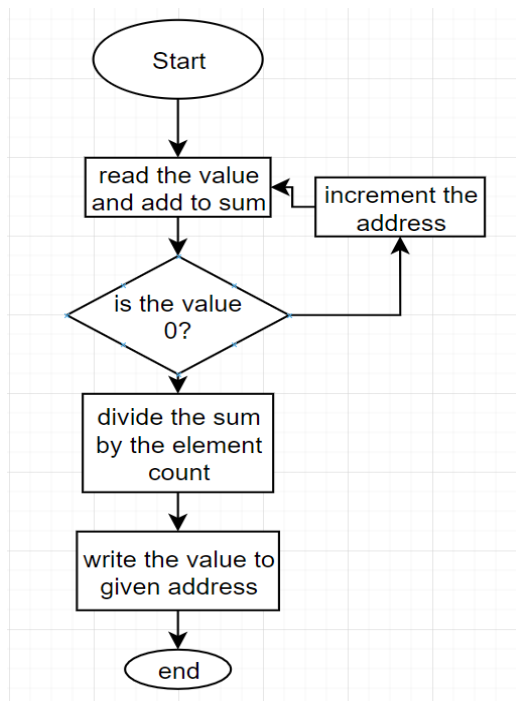

```

73:      B .
0x0800003C E7FE      B      0x0800003C
0x0800003E 0000      DCW      0x0000
0x08000040 4224      DCW      0x4224
0x08000042 0001      DCW      0x0001
0x08000044 001A      DCW      0x001A
0x08000046 0000      DCW      0x0000
<
hw2_7.s
47
48      LDR r0, =0x14224 ; address values for read and write
49      LDR r1, =0x14228
50      LDR r2, =0x1422C
51
52      LDR r3, [r0] ; getting the values from given addresses
53      LDR r4, [r1]
54
55      EORS r3, r3, r4 ; performing XOR operation on two values
56
57      LDR r5, =0x0 ; loop counter
58
59      kernighan_loop
60
61      CMP r3, #0x0
62      BEQ loop_stop
63      SUBS r4, r3, #0x1
64      ANDS r3, r3, r4
65      ADDS r5, r5, #0x1
66      B kernighan_loop
67
68      loop_stop
69
70      STR r5, [r2] ; writing the hamming distance to given address
71
72      ; ///////////
73      B .
74      ENDP
75
76      END

```

After performing Kernighan loop we get the amount of high bits as 2 on the r5 register. After hamming distance is calculated we write the value to the give address.

I. Problem 8



Now that we created the flowchart we can write the assembly code according to the flowchart.

```
Stack_Size      EQU      0x00000400

                AREA     STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE    Stack_Size
__initial_sp


                THUMB


                AREA     RESET, DATA, READONLY
                EXPORT   __Vectors

__Vectors
    DCD      __initial_sp          ; Top of Stack
    DCD      Reset_Handler        ; Reset Handler
    DCD      NMI_Handler          ; NMI Handler
    DCD      HardFault_Handler    ; Hard Fault Handler


                AREA     |.text|, CODE, READONLY


; nmi handler
NMI_Handler     PROC
    EXPORT      NMI_Handler
    B .
ENDP
```

```
; hardfault handler
```

```
HardFault_Handler    PROC
```

```
    EXPORT HardFault_Handler
```

```
    B .
```

```
    ENDP
```

```
; entry function
```

```
Reset_Handler    PROC
```

```
    EXPORT Reset_Handler
```

```
    ; //////////////////////////////////
```

```
        LDR r1, =0x20000100 ; setting up data to read to
```

```
        LDR r2, =0xA
```

```
        STR r2, [r1]
```

```
        LDR r1, =0x20000104
```

```
        LDR r2, =0x4
```

```
        STR r2, [r1]
```

```
        LDR r1, =0x20000108
```

```
        LDR r2, =0x7
```

```
        STR r2, [r1]
```

```
        LDR r1, =0x2000010C
```

```
        LDR r2, =0x0
```

```
        STR r2, [r1]
```

```
        LDR r0, =0x20000000 ; start address
```

```
        LDR r1, =0x0 ; start value
```

```
        LDR r2, =0x0 ; number of elements
```

```
        LDR r3, =0x00000100 ; offset value
```

```
average_loop
```

```
        LDR r4, [r0, r3] ; reading the value on the register with offset
```

```
        ADDS r1, r1, r4 ; add the value to start value
```

```
        ADDS r2, r2, #0x1 ; add 1 to element counter
```

```
        ADDS r3, r3, #0x4 ; add 4 to address offset
```

```
        CMP r4, #0x0 ; check if its 0
```

```
        BNE average_loop
```

```
        SUBS r2, r2, #0x1 ; 0 is counted too so we remove 1 from the element counter
```

```
        BL divide ; jump to division branch
```

```
divide
```

```
        CMP r1, r2
```

```
        BLT divide_stop ; compare if the r1 value is lower than r2 if so jump out of division
```

```

SUBS r1, r1, r2 ; subtract r2 from r1

ADDS r5, r5, #1 ; add one to multiplication result

B divide ;

divide_stop

    STR r5, [r0]

; ///////////

B .

ENDP

END

```

This code first writes a bunch of values to the given addresses for the testing of the code. These values add up to 21 in total and there are 3 values. This makes it so that the average of these values should be calculated as 7.

Register	Value
R0	0x20000000
R1	0x00000015
R2	0x00000003
R3	0x00000110
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000042
xPSR	0x21000000

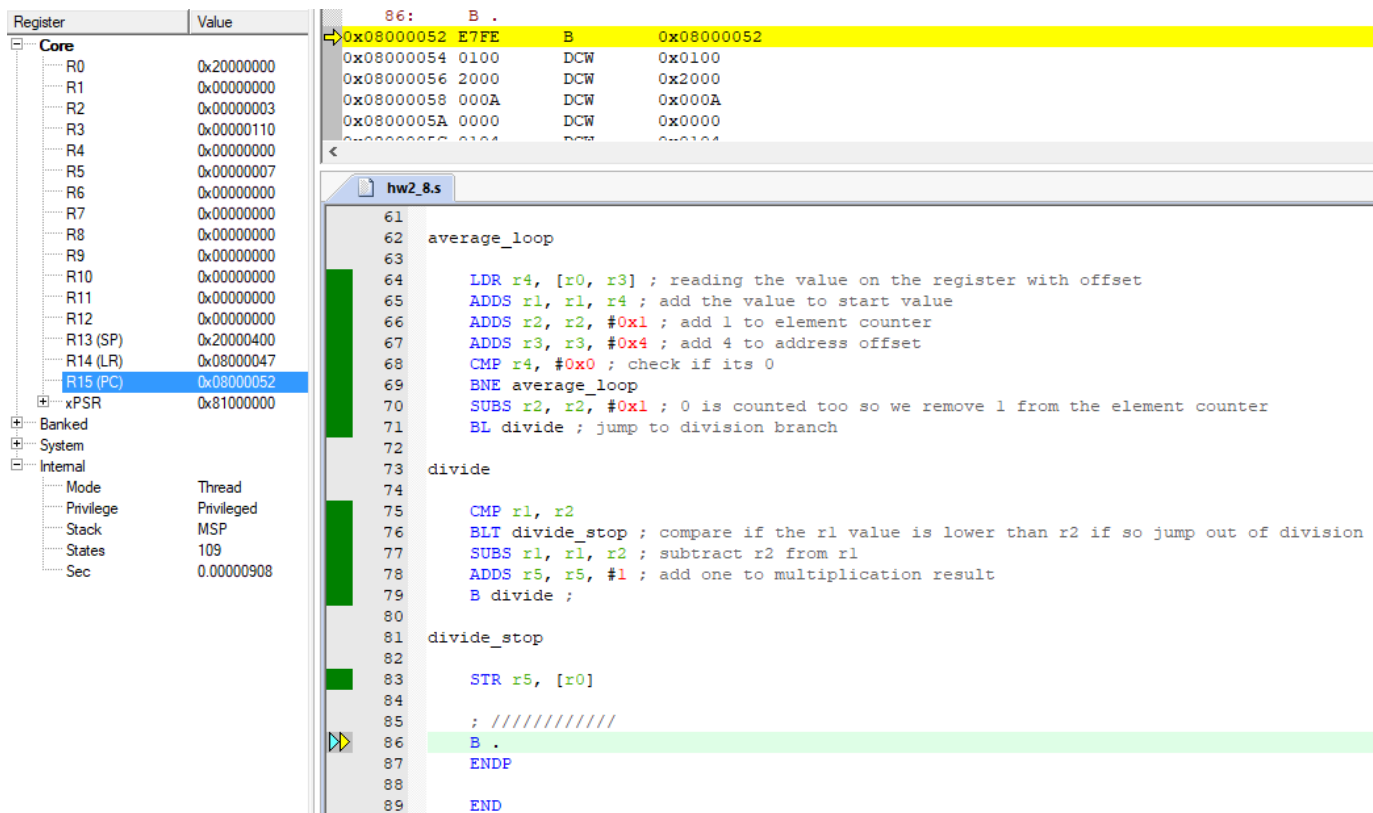
```

0x08000040 1E52      SUBS      r2,r2,#1
71:         BL divide ; jump to division branch
72:
73: divide
74:
=>0x08000042 F000F800 BL.W      0x08000046
75:         CMP r1, r2

hw2_8.s
61
62 average_loop
63
64     LDR r4, [r0, r3] ; reading the value on the register with offset
65     ADDS r1, r1, r4 ; add the value to start value
66     ADDS r2, r2, #0x1 ; add 1 to element counter
67     ADDS r3, r3, #0x4 ; add 4 to address offset
68     CMP r4, #0x0 ; check if its 0
69     BNE average_loop
70     SUBS r2, r2, #0x1 ; 0 is counted too so we remove 1 from the element counter
71     BL divide ; jump to division branch
72
73 divide
74
75     CMP r1, r2
76     BLT divide_stop ; compare if the r1 value is lower than r2 if so jump out of division
77     SUBS r1, r1, r2 ; subtract r2 from r1
78     ADDS r5, r5, #1 ; add one to multiplication result
79     B divide ;
80
81 divide_stop
82
83     STR r5, [r0]
84
85     ; ///////////
86     B .
87     ENDP
88
89     END

```

As we can see we have 0x15 on the total sum register r1 and a 3 value on the element count register r2.



As we can see the division value is stored on r5 as 7 which is the correct number for the division of 21 and 3. Then we store the r5 value on the given address with STR.

References

[1]. Random number generation algorithm

https://en.wikipedia.org/wiki/Linear_congruential_generator

[2]. Brian kernighans algorithm for hemming distance calculation

<https://medium.com/@sanchit3b/brian-kernighans-algorithm-9e0ca5989148>

[3]. Arm instruction set summary for clock cycle counts and instruction set details

<https://developer.arm.com/documentation/ddi0484/b/Programmers-Model/Instruction-set-summary>

[4]. Division algorithm for average calculation

https://en.wikipedia.org/wiki/Division_algorithm