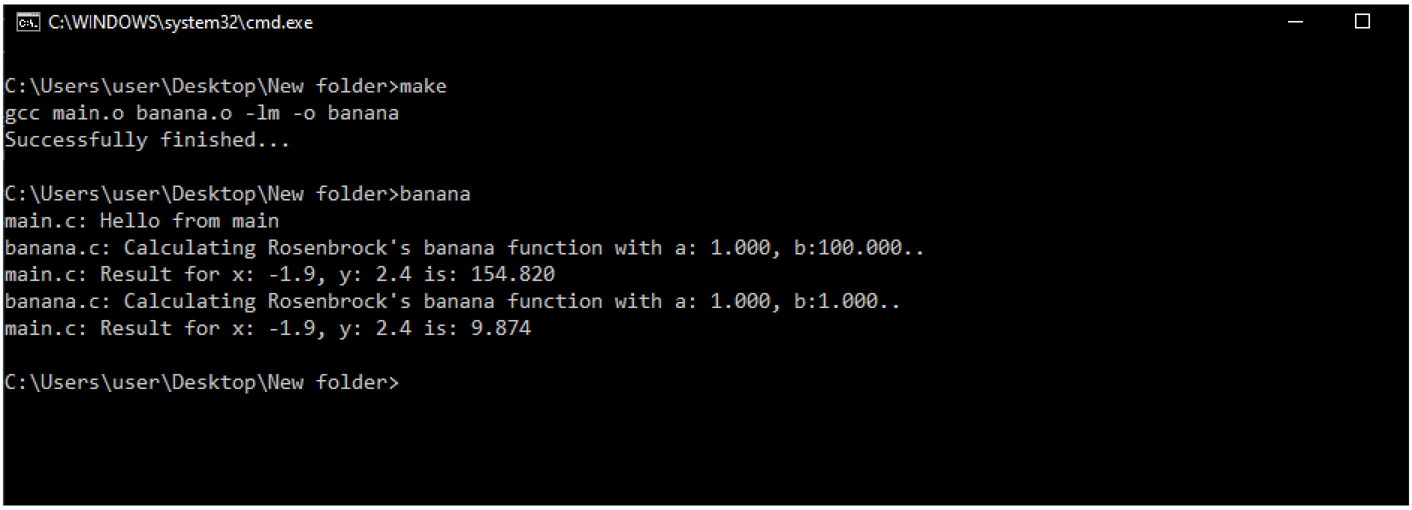
ELM 334 - Homework #2

Ömer Emre Polat
1801022037

A. Problem 0

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

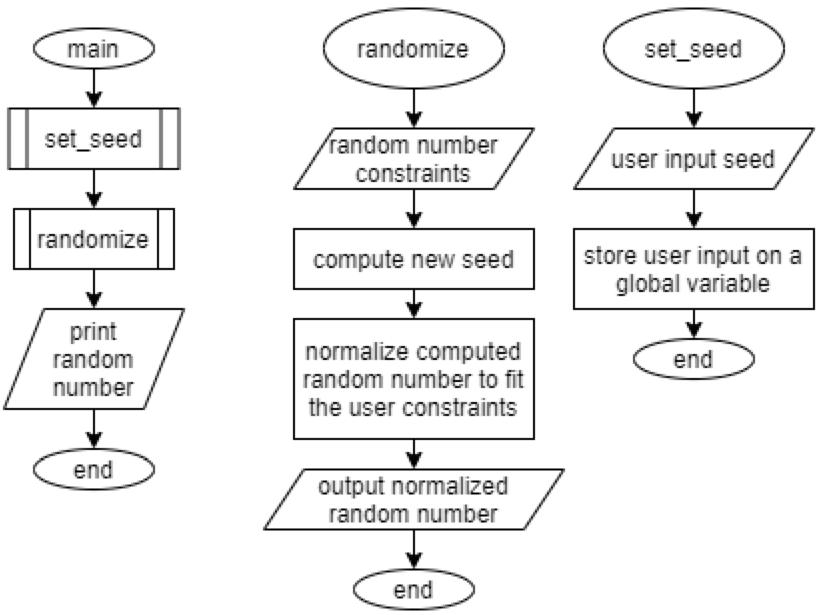


```
C:\WINDOWS\system32\cmd.exe

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;
}
```

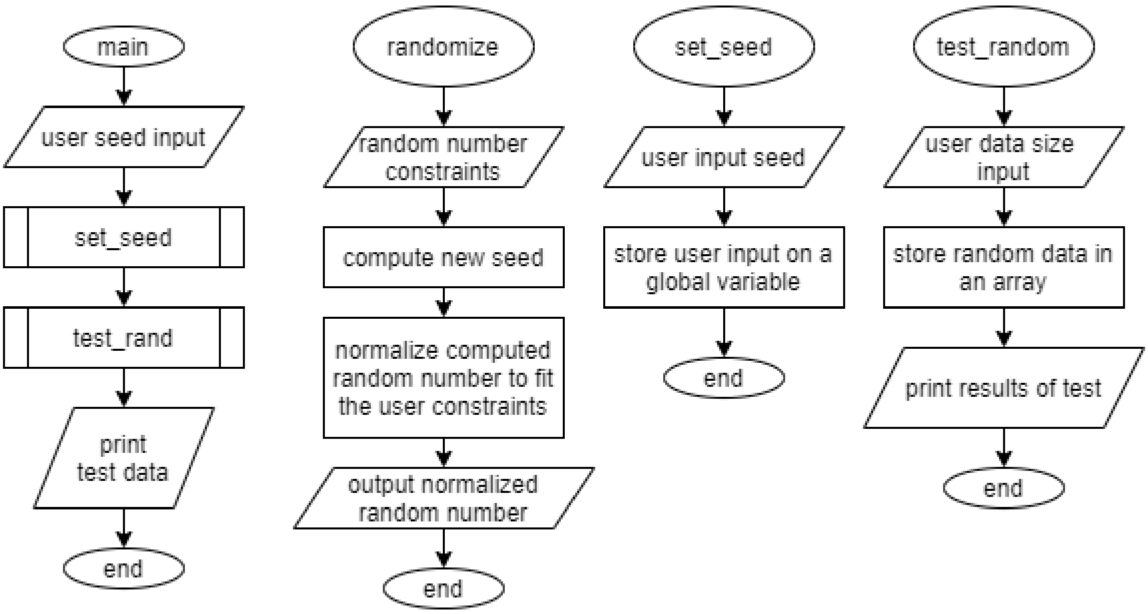
With the codes written we can test it using the makefile template given to us.

```
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 thats 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

```c
#ifndef _TEST_RANDOM_H
#define _TEST_RANDOM_H

void test_random(int);

#endif
```

test.c

```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

```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		

`/////////////////////////////////////////////////////////`

0100 0100 0000 1000

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

0001 1110 1010 0010

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		

0001 0101 0110 0010

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

0101 0000 0111 0101

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		

//

0100 0111 0111 0000

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)

1101 0001 0000 0111

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	

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

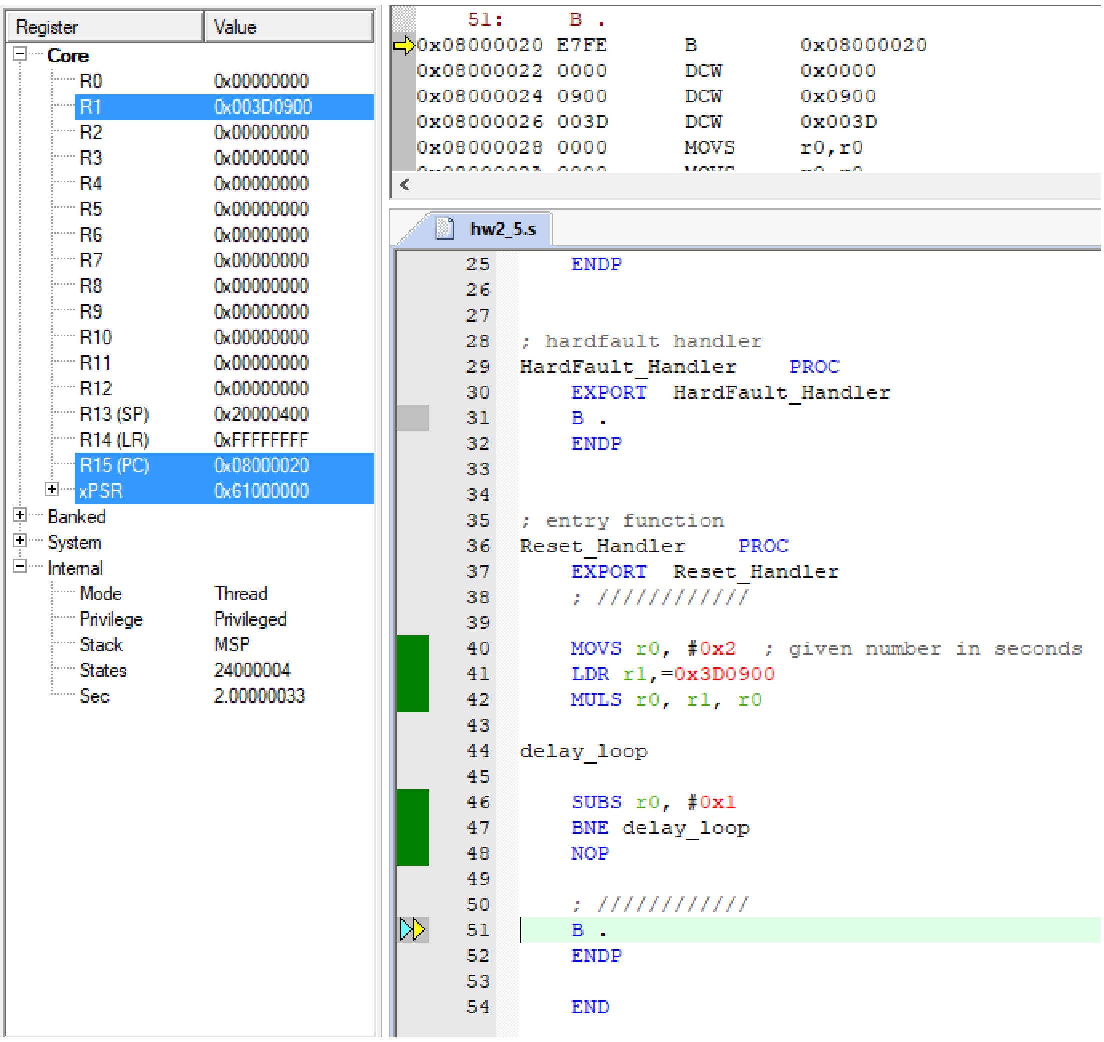
Instruction cycle times are taken from ARM11 (ARMv6) textbook.

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
```



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.

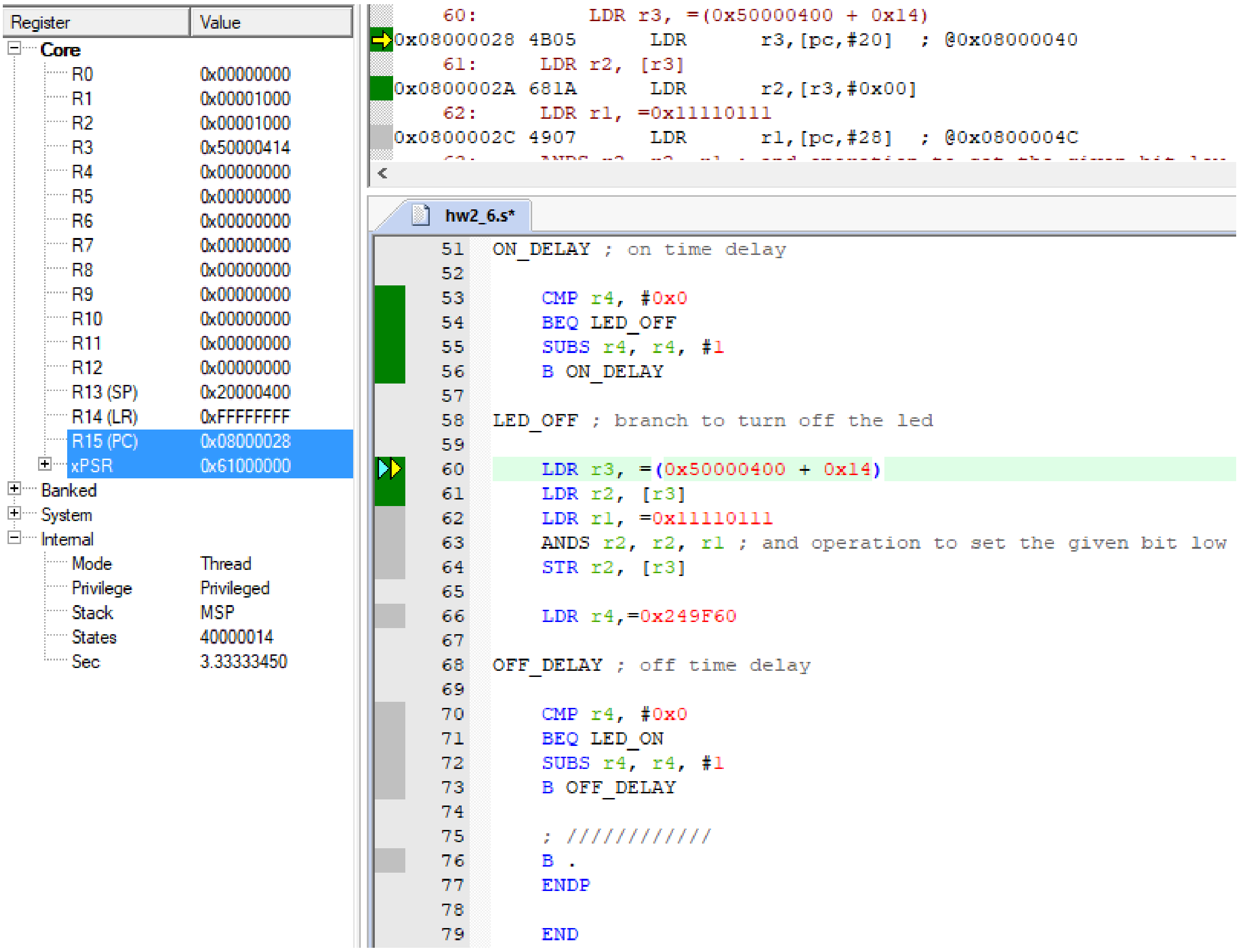
$$delay\ loop\ count = \frac{16MHz}{1+1} = 8M = 0x7A1200$$

```
Stack_Size      EQU     0x00000400


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

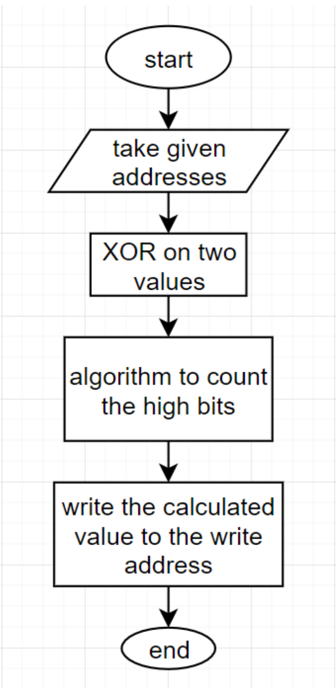
```
    THUMB

    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, =0xFFFFEFFF
        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.



H. Problem 7



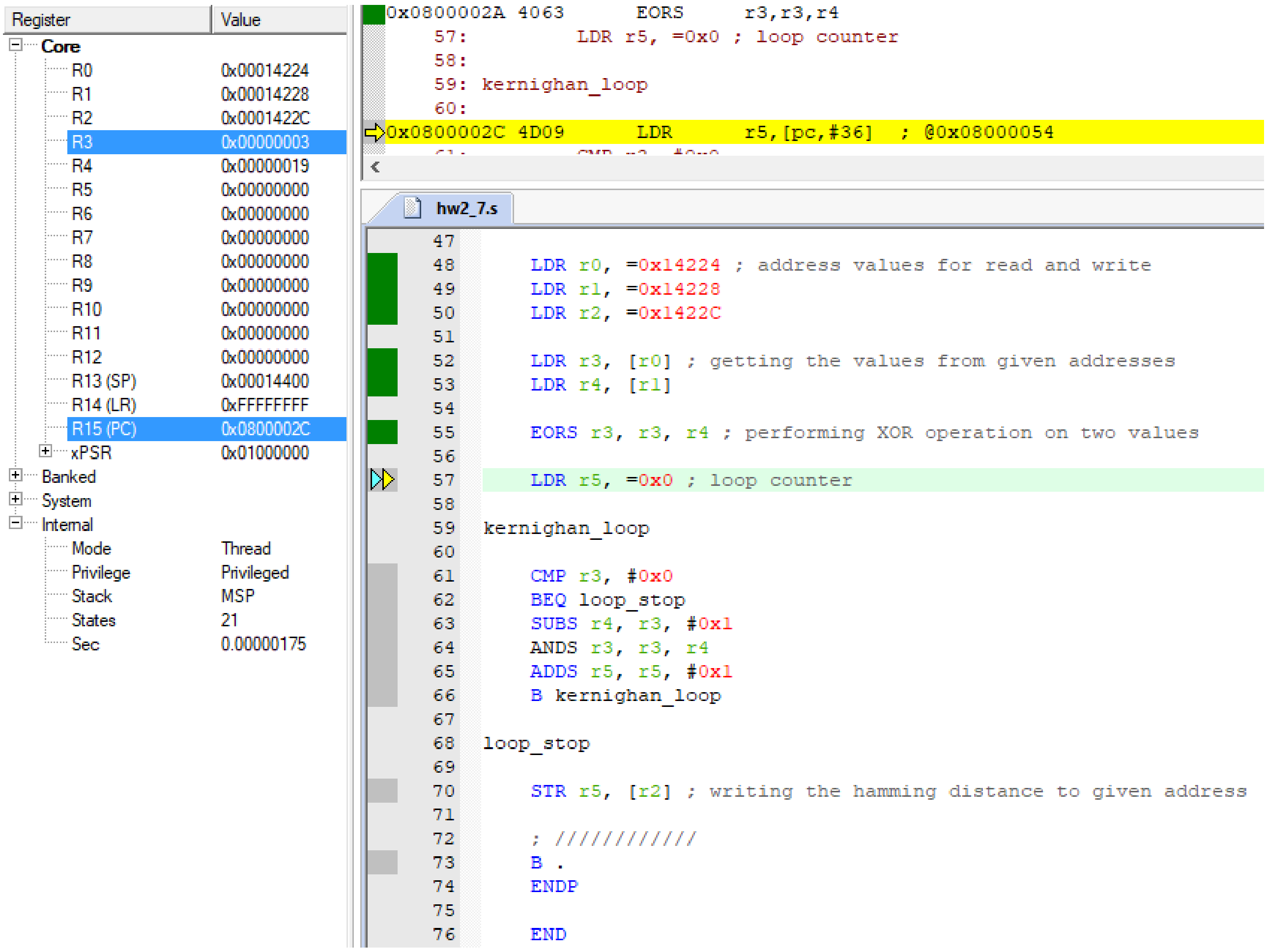
After creating the flowchart we can use it as a template to write the program code.

To calculate the hemming 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 hemming distance.

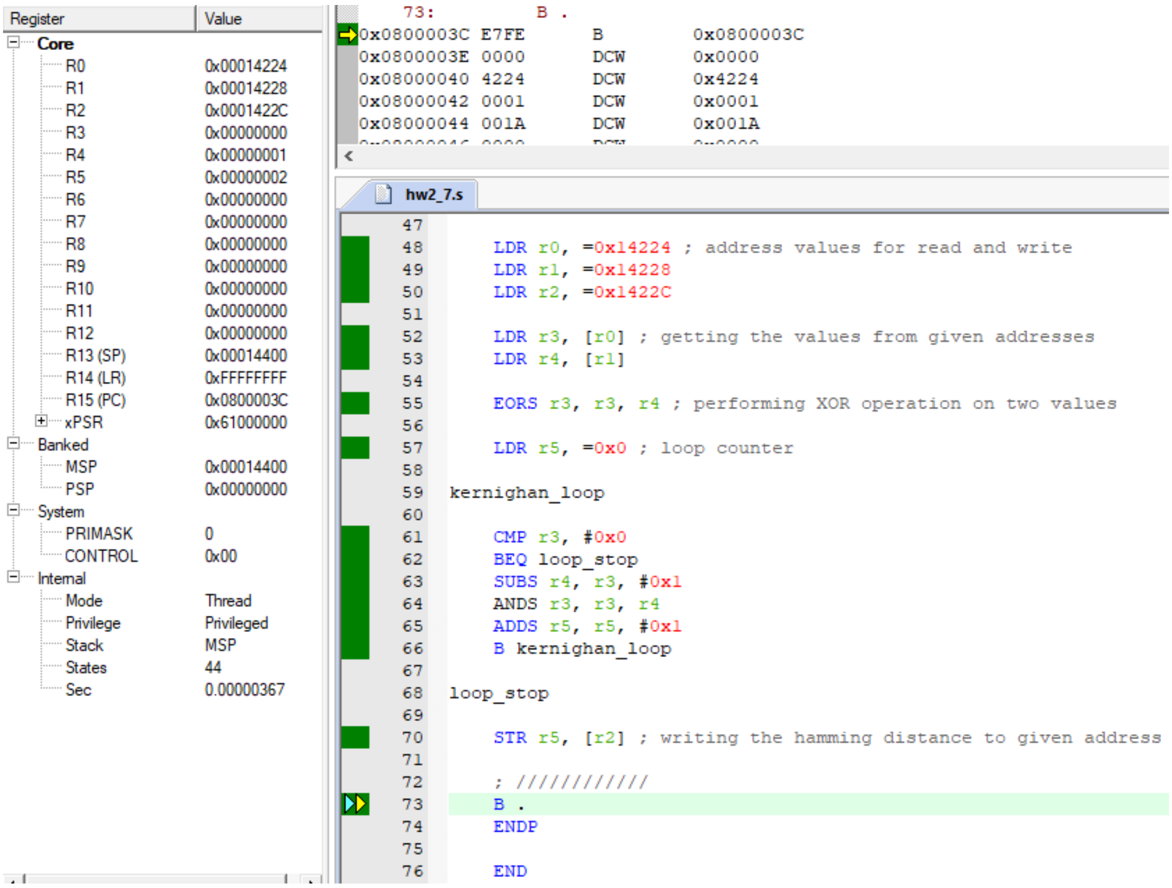
```
Stack_Size      EQU     0x00000400

                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.

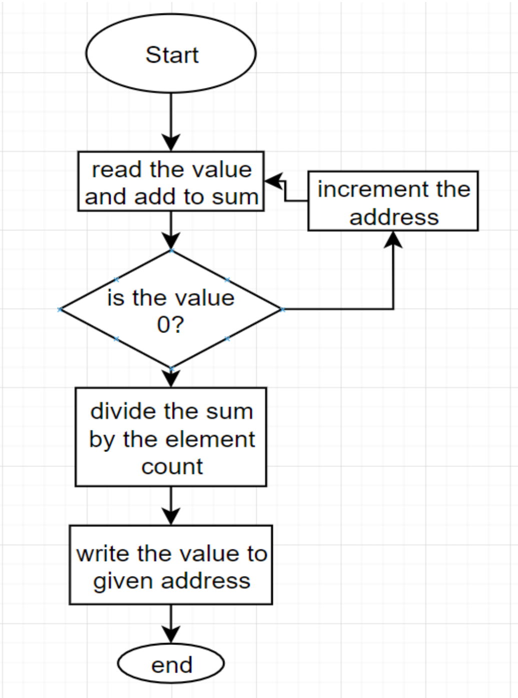


After performing XOR operation one both test values (value1=11010, value2=11001) we get the value of 00011 on the r3 register. Now the code will perform the Kernighan loop to calculate the amount of high bits.



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



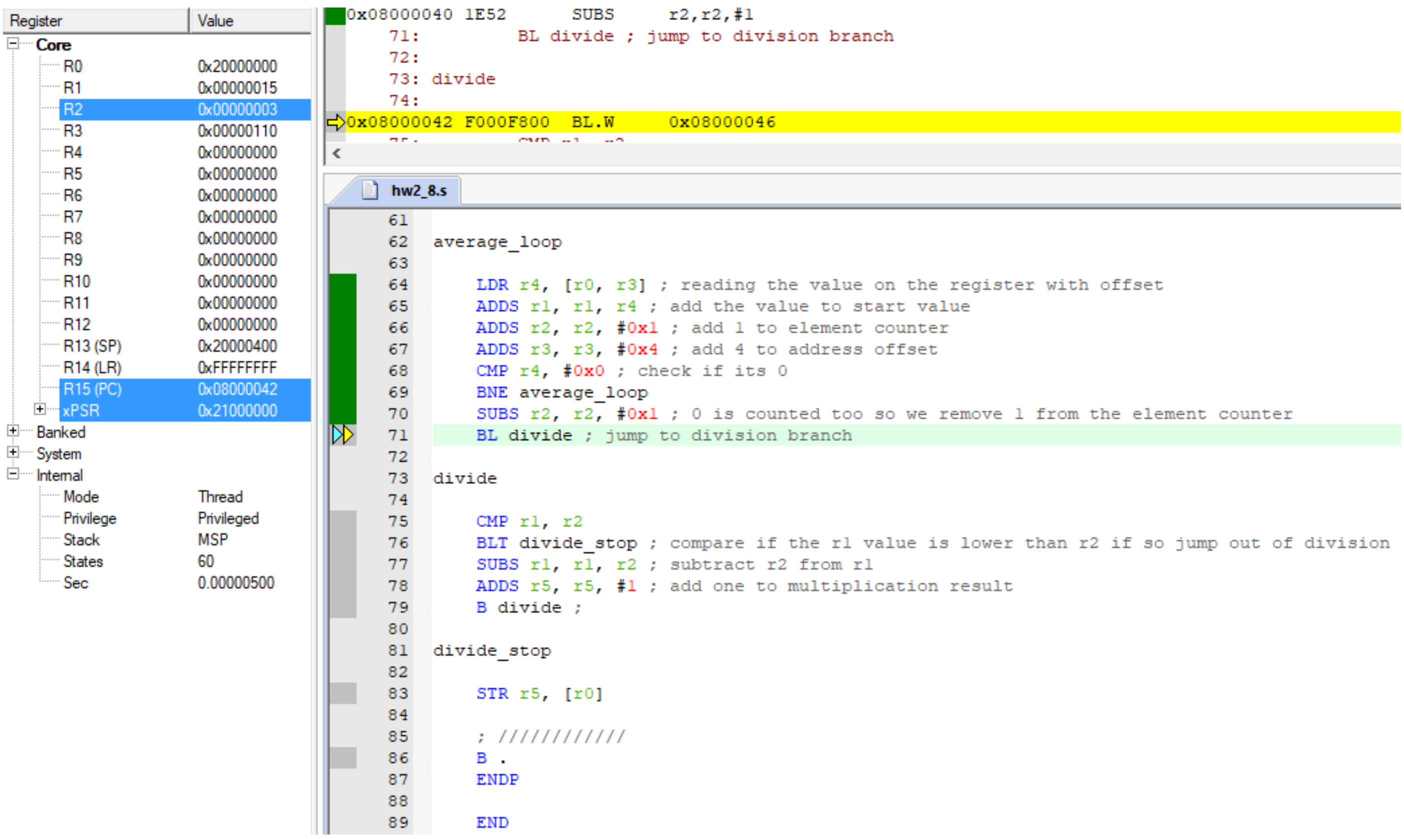
Now that we created the flowchart we can write the assembly code according to the flowchart.

```
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
```

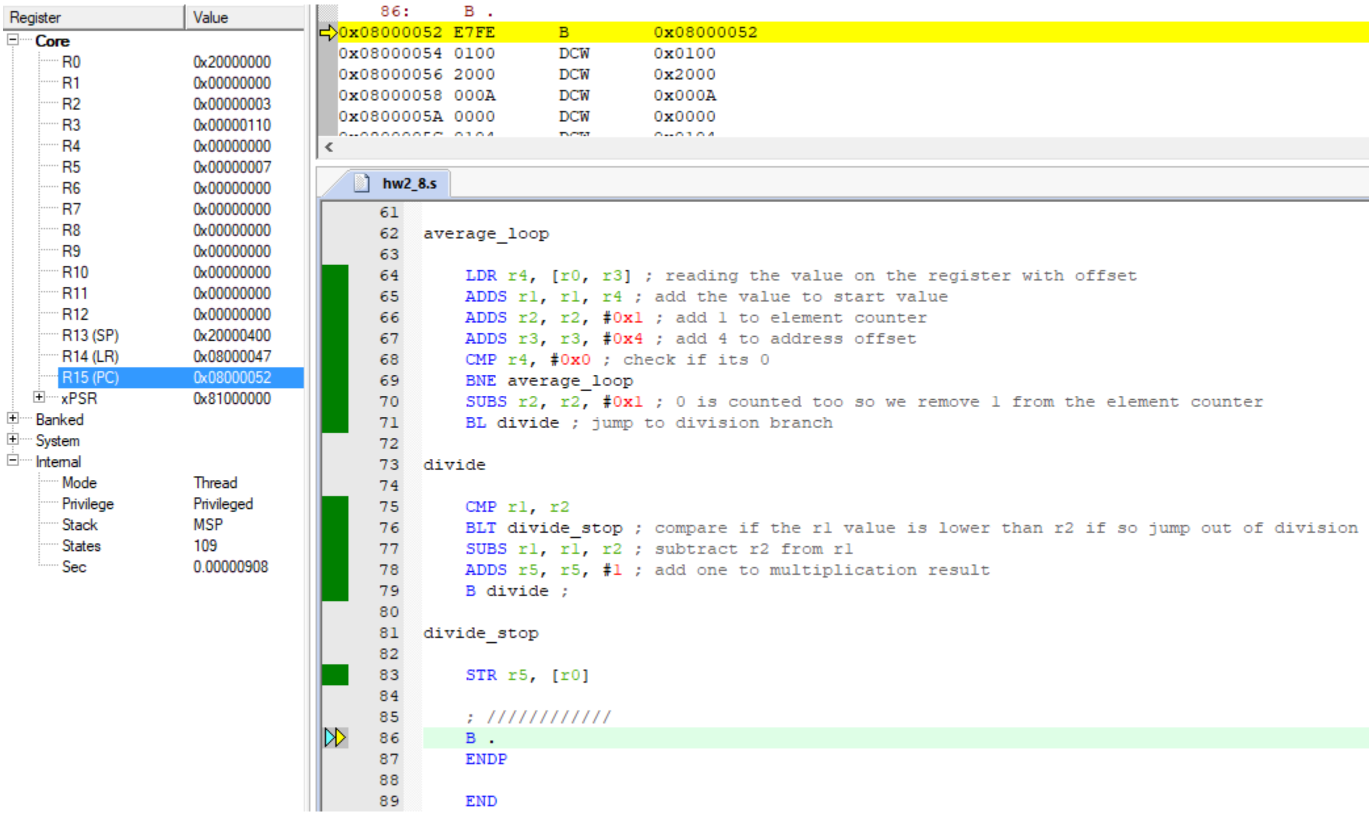
```
; hardfault handler
HardFault_Handler    PROC
    EXPORT  HardFault_Handler
    B .
    ENDP



; 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.



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