ELEC 334 - Homework #2

Berat KIZILARMUT - 171024086

A. Problem 0 – Test Setup

Requested files have been downloaded and has been put in a directory. Make operation was completed within Visual Studio Code Terminal. Compiled banana.exe has been ran with cmd.

```
Komut İstemi
Microsoft Windows [Version 10.0.18362.1139]
(c) 2019 Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\kuros>"D:\Datas\Dropbox\Ders\ELM334 - Microprocessors\hw2\github\banana.exe"
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\kuros>
```

Figure 1

B. Problem 1 – Pseudo-random Number Generator

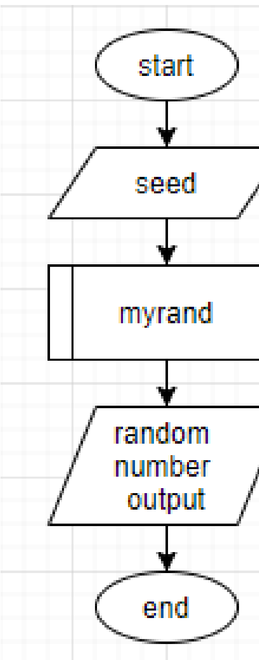


Figure 2

A Pseudo-random number generator flowchart has been designed, Figure 2, this program takes seed value from the user, puts it through the written random function and provides random outputs.

Starting from myrand.h, firstly two very specific numbers are defined which were found via research then a “static int next” value is defined. Rather than using pointers or structures, next is defined as static. Nextly random function itself and setseed function is defined.

Going to myrand.c, random function itself is defined, a fail safe if statement has been added to exclude negative results. Setseed is simple and just sets taken seed to next.

Finally on main.c, seed is taken from the user and random numbers are printed out.

```c
#ifndef MYRAND_H
#define MYRAND_H

#define a (16807)
#define b (2147483647)
static int next = 1;
int random(void);
void setseed(int seed);

#endif
```

Figure 3

Myrand.h, given Figure 3, which has function declaration.

```c
#include <stdio.h>
#include <stdlib.h>
#include "myrand.h"

int random(void)
{
    next = (a * next) % b;
    if(next<1)
        random();
    return next;
}

void setseed(int seed)
{
    next = seed;
}
```

Figure 4

Myrand.c, given Figure 4, which has pseudo random number generator function defitions.

And main.c, given Figure 5, which takes a seed input from the user and calls the pseudo number generator functions.

```c
#include <stdio.h>
#include <stdlib.h>
#include "myrand.h"

int main()
{
    int x;
    printf("Set the pseudo random number generator seed: ");
    scanf("%d",&x);
    setseed(x);
    for(int i=0; i<10; i++)
    printf("%d\n",(random()%15)+1);
    return 0;
}
```

Figure 5

C. Problem 2 – Test your random number generator

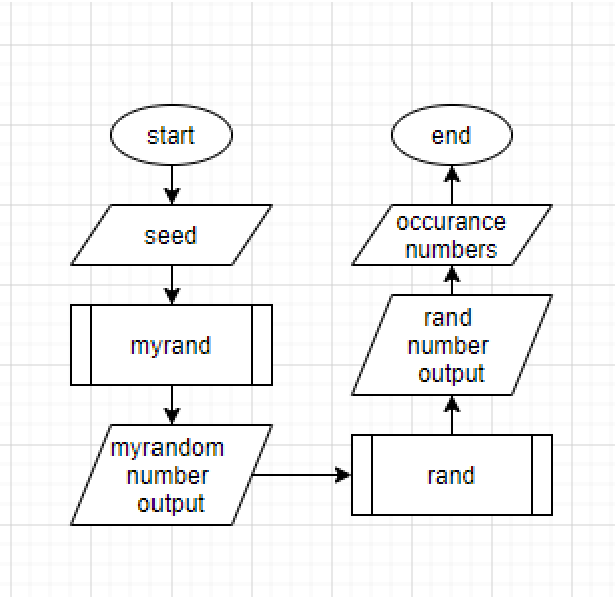


Figure 6

Self-built Pseudo-random Number Generator and C built-in random number generator has been tested with 200k samples each. Comparing the results, they were within margin of error, and are reasonable.

No changes were made to the myrand.h and myrand.c

A new function, test_random.c has been created, which has two functions; test_myrandom and test_crandom. Both of these are very similar to each other, they both have arrays that store occurrence of numbers, both have switch cases that increase those said occurrence numbers.

Necessary changes and update were made to the main.c file, new functions were added, self-built pseudo number generator still uses seed from user input.

These said new files are;

main.c, given Figure 7, which takes a seed input from the user and calls the test functions.

```c
#include <stdio.h>
#include <stdlib.h>
#include "myrand.h"
#include "test_random.h"

int main()
{
    int x;
    printf("Set the self built pseudo random number generator seed: ");
    scanf("%d",&x);
    setseed(x);
    test_myrandom();
    test_crandom();
    return 0;
}
```

Figure 7

```
#ifndef TEST_RANDOM_H
#define TEST_RANDOM_H
void test_myrandom(void);
void test_crandom(void);
#endif
```

Test_random.h, given Figure 8, which has function declaration.

Figure 8

And test_random.c, given Figura 9, which has the test_myrandom and test_crandom test functions.

```
#include <stdio.h>
#include <stdlib.h>
#include "myrand.h"
#include "test_random.h"

void test_myrandom(void)
{
    printf("Commencing self built pseudo-random number generator test:\n");
    int var;
    int mycounter[15]={0};

    for (int i=0; i<200000; i++)
    {
        var=(random()%15)+1;
        switch(var)
        {
            case 1:
            mycounter[0]++;
            break;

            case 2:
            mycounter[1]++;
            break;

            case 3:
            mycounter[2]++;
            break;

            case 4:
            mycounter[3]++;
            break;

            case 5:
            mycounter[4]++;
            break;

            case 6:
            mycounter[5]++;
            break;
```

```c
            case 7:
            mycounter[6]++;
            break;

            case 8:
            mycounter[7]++;
            break;

            case 9:
            mycounter[8]++;
            break;

            case 10:
            mycounter[9]++;
            break;

            case 11:
            mycounter[10]++;
            break;

            case 12:
            mycounter[11]++;
            break;

            case 13:
            mycounter[12]++;
            break;

            case 14:
            mycounter[13]++;
            break;

            case 15:
            mycounter[14]++;
            break;
        }
    }

    printf("Self made pseudo-random number generator test is completed\n");
    for(int j=0; j<15; j++)
    {
        printf("Number %d occured %d times\n",(j+1),mycounter[j]);
    }
}

void test_crandom(void)
{
    printf("Commencing C built in random number generator test:\n");
```

```
    int cvar;
    int ccounter[15]={0};
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0; i<200000; i++)
    {
        cvar=(rand()%15)+1;
        switch(cvar)
        {
            case 1:
            ccounter[0]++;
            break;

            case 2:
            ccounter[1]++;
            break;

            case 3:
            ccounter[2]++;
            break;

            case 4:
            ccounter[3]++;
            break;

            case 5:
            ccounter[4]++;
            break;

            case 6:
            ccounter[5]++;
            break;

            case 7:
            ccounter[6]++;
            break;

            case 8:
            ccounter[7]++;
            break;

            case 9:
            ccounter[8]++;
            break;

            case 10:
            ccounter[9]++;
            break;
```

```
            case 11:
            ccounter[10]++;
            break;

            case 12:
            ccounter[11]++;
            break;

            case 13:
            ccounter[12]++;
            break;

            case 14:
            ccounter[13]++;
            break;

            case 15:
            ccounter[14]++;
            break;
        }
    }

    printf("C Built in random number generator test is completed\n");
    for(int j=0; j<15; j++)
    {
        printf("Number %d occured %d times\n",(j+1),ccounter[j]);
    }
}
```

D. Problem 3 – Instruction Decode

ARMv6-M Architecture Reference Manual has been gathered. All the information below has been acquired withing “Thumb Instruction Details” section of the datasheet.

- ldr r5, [r6, #4]

This operation is an LDR immediate operation (A6.7.26) which has it’s encoding given below;

`LDR <Rt>, [<Rn>{,#<imm5>}]`

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

Rt on this operation is r5, so 3 bit Rt should be “101”. Rn on this operation is r6, so 3 bit Rn should be “110”. Immediate is #4 , address offset moves 4 by 4 so the imm5 is “00001”.

Full machine code should be “0110 1000 0111 0101”

➢ mvns r4, r4

This is an MVN register operation (A6.7.45), encoding procedure given below;

MVNS <Rd>,<Rm>

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		

Rm on this operation is r4, so 3 bit Rm should be “100”. Rd on this operation is r4, so 3 bit Rd should be “100” aswell.

Full machine code is;

“0100 0011 1110 0100”

➢ ands r5, r5, r4

This is an AND register operation (A6.7.7), encoding procedure given below;

ANDS <Rdn>,<Rm> ANDS{<q>} {<Rd>,} <Rn>, <Rm>

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		

Since the destination, Rd, register (r5) is the same as the Rn, they are combined as Rdn. Since Rdn on this operation is r5, 3 bit Rdn should be “101”. Rm on this operation is r4, 3 bit Rm is “100”.

Full machine code is; “0100 0000 0010 0101”

➢ adds r0, r0, r1

This is an ADD register operation (A6.7.3), encoding T1 procedure given below;

ADDS <Rd>,<Rn>,<Rm>

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

Rd on this operation is r0, 3 bit Rd is “000”. Rn on this operation is r0, 3 bit Rn is “000”. Rm on this operation is r1, 3 bit Rm is “001”.

Full machine code is; “0001 1000 0100 0000”

➢ add r0, r0, r1

This is an ADD register operation (A6.7.3), encoding T2 procedure given below;

ADD <Rdn>,<Rm>

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

Since the destination, Rd, and Rn is the same register, they are combined in to Rdn, which is 3 bit “000”. Rm is r1, Rm 4 bit is “0001”. DN is “0”.

Full machine code is: “0100 0100 0000 1000”

- subs r2, r4, #2

This is an SUB immediate operation (A6.7.65), encoding procedure given below;

`SUBS <Rd>,<Rn>,#<imm3>`

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		

Rd on this operation is r2, 3 bit Rd is “010”. Rn on this operation is r4, 3 bit Rn is “100”. Immediate number is “2”, imm3 is “010.

Full machine code is: “0001 1110 1010 0010”

- asrs r2, r4, #21

This is an ASR, Arithmetic Shift Right, immediate operation (A6.7.8), encoding procedure given below;

`ASRS <Rd>,<Rm>,#<imm5>`

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

Rd on this operation is r2, Rd 3 bit is “010”. Rm on this operation is r4, Rm 3 bit is “100”. Immediate number is 21, imm5 is “ 10101”.

Full machine code is: “0001 0101 0110 0010”

- str r5, [r6, r1]

This is an STR, store register, register operation (A6.7.60), encoding procedure given below;

`STR <Rt>,[<Rn>,<Rm>]`

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		

Rt on this operation is r5, Rt 3 bit is “101”. Rn on this operation is r6, Rn 3 bit is “110”. Rm on this operation is r1, Rm 3 bit is “001”.

Full machine code is: “0101 0000 0111 0101”

- bx lr

This is an branch and exchange operation (A6.7.15), encoding given below;

`BX <Rm>`

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)

Rm is given as lr in this operation, which means link register. Since lr is 14, Rm 4 bit is “1110”.

Full machine code is; “0100 0111 0111 0000”

- bne 0x12

  This is a B, branch, operation (A6.7.10), encoding procedure given below;

```
B<c>  <label>
```

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

Condition is given as ne, not equal, which has a flag value of “0001”. Label is given as “0x12”, imm8 should be “0001 0010”

Full machine code is:” 1101 0001 0001 0010”

E. Problem 4 – Instruction cycle times

Instruction cycle times have been gathered from ARM, Instruction set summary, reference given at the end of the report.

- ldr r5, [r6, #4] — 1 Cycle
- mvns r4, r4 — 1 Cycle
- ands r5, r5, r4 — 1 Cycle
- adds r0, r0, r1 — 1 Cycle
- add r0, r0, r1 — 1 Cycle
- subs r2, r4, #2 — 1 Cycle
- asrs r2, r4, #21 — 1 Cycle
- str r5, [r6, r1] — 2 Cycles
- bx lr — 2 Cycles
- bne 0x12 — 1 Cycle

F. Problem 5 – Assembly delay function

```
Delayfunc
      SUBS r0, #0x1
      BNE delayfunc
      NOP
```

Figure 9

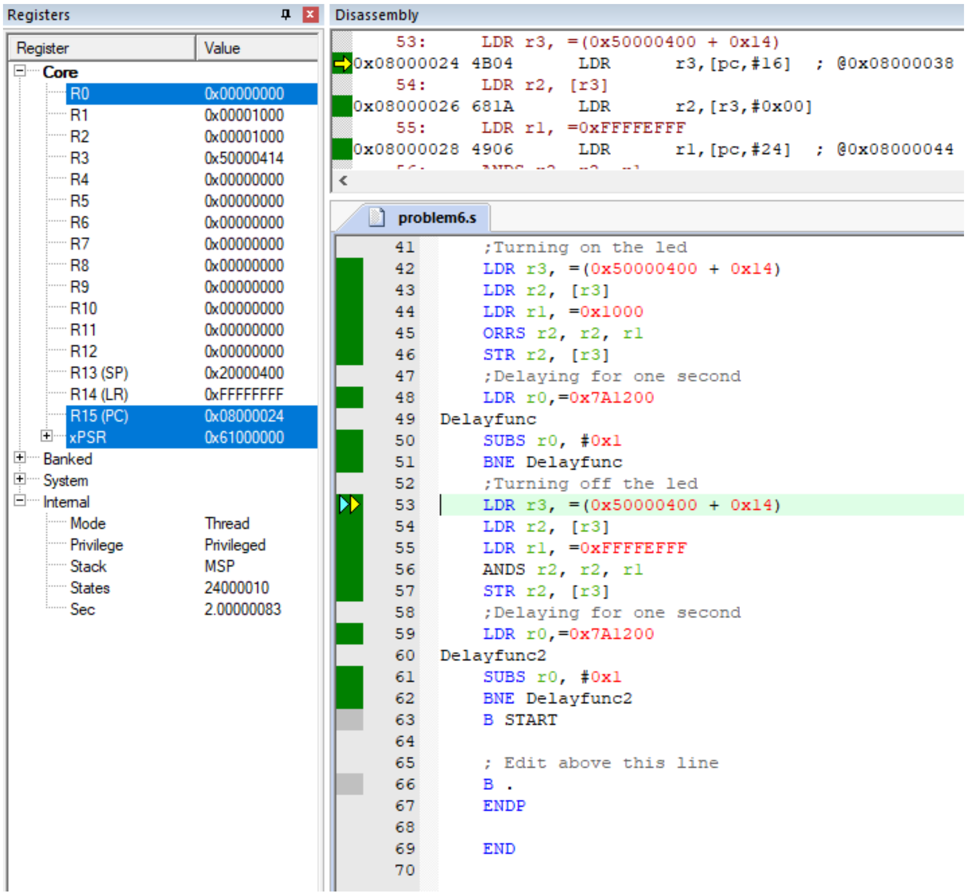
If a number were to be given to r0, delay function will work. Used this exact function on Problem 6.

G. Problem 6 – Assembly LED Toggle

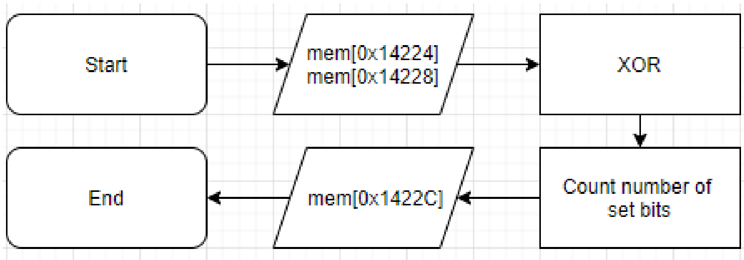
```
START
    ;Turning on the led
    LDR r3, =(0x50000400 + 0x14)
    LDR r2, [r3]
    LDR r1, =0x1000
    ORRS r2, r2, r1
    STR r2, [r3]
    ;Delaying for one second
    LDR r0,=0x7A1200
Delayfunc
    SUBS r0, #0x1
    BNE Delayfunc
    ;Turning off the led
    LDR r3, =(0x50000400 + 0x14)
    LDR r2, [r3]
    LDR r1, =0xFFFFEFFF
    ANDS r2, r2, r1
    STR r2, [r3]
    ;Delaying for one second
    LDR r0,=0x7A1200
Delayfunc2
    SUBS r0, #0x1
    BNE Delayfunc2
    B START
```

Utilized the information I gathered from Lab1 on ELEC335 on turning the led on. Implemented the Problem 5 code to this use case. Chose an appropriate number to load to r0 for 16 MHz processor. Substraction and branch not equal operations take 2 cycles. Divided 16 million by 2 for 2 cycles per whole operation, gathered 8 million per second. Set r0 to “0x7A1200” to get approximately 1 second delay.

Line of codes are added to provided “basic assembly template for keil ARM simulator”, only added code given left to prevent mess.



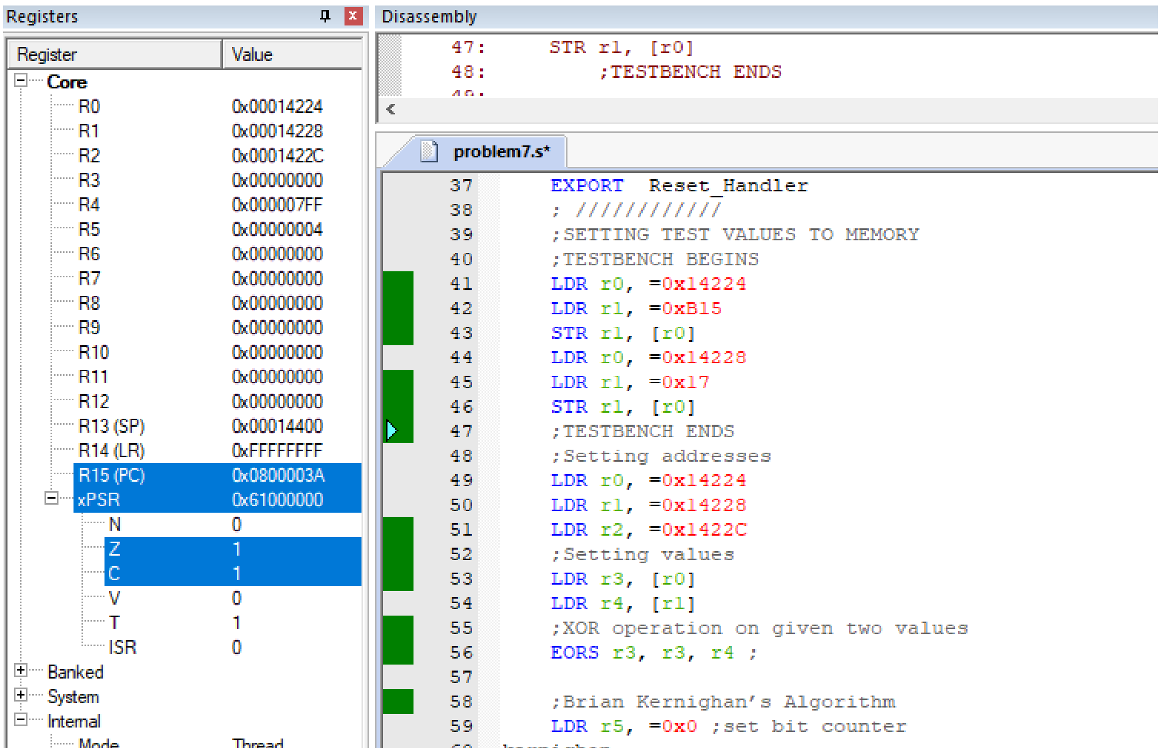
H. Problem 7 – Assembly Hamming Distance



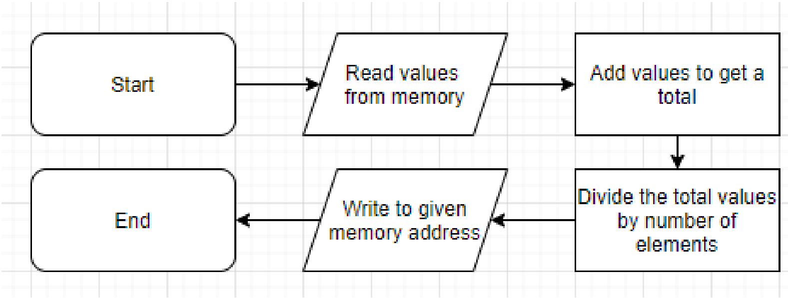

```
    ;Setting addresses
    LDR r0, =0x14224
    LDR r1, =0x14228
    LDR r2, =0x1422C
    ;Setting values
    LDR r3, [r0]
    LDR r4, [r1]
    ;XOR operation on given two values
    EORS r3, r3, r4 ;

    ;Brian Kernighan's Algorithm
    LDR r5, =0x0 ;set bit counter
kernighan
    CMP r3, #0x0
    BEQ break
    SUBS r4, r3, #0x1
    ANDS r3, r3, r4
    ADDS r5, r5, #0x1
    B kernighan

    ;breaking the loop, storing hamming distance
break
    STR r5, [r2]
```

Firstly provided addresses has been set to first three register (some values were written to mem on these addresses on the testbench, not included in the code given left), set the values to the next 2 registers to use the values of the given addresses. Used the XOR operation on r3 and r4 to get the hamming distance. Calculating the amount of set bits was not an easy task in assembly, to achieve this goal researched and found Brian Kernighan's Algorithm for set bit counting, reference added below the report. Implemented the algorithm in assembly. Lastly stored the result to register 5.

Line of codes are added to provided "basic assembly template for keil ARM simulator", only added code given left to prevent mess.



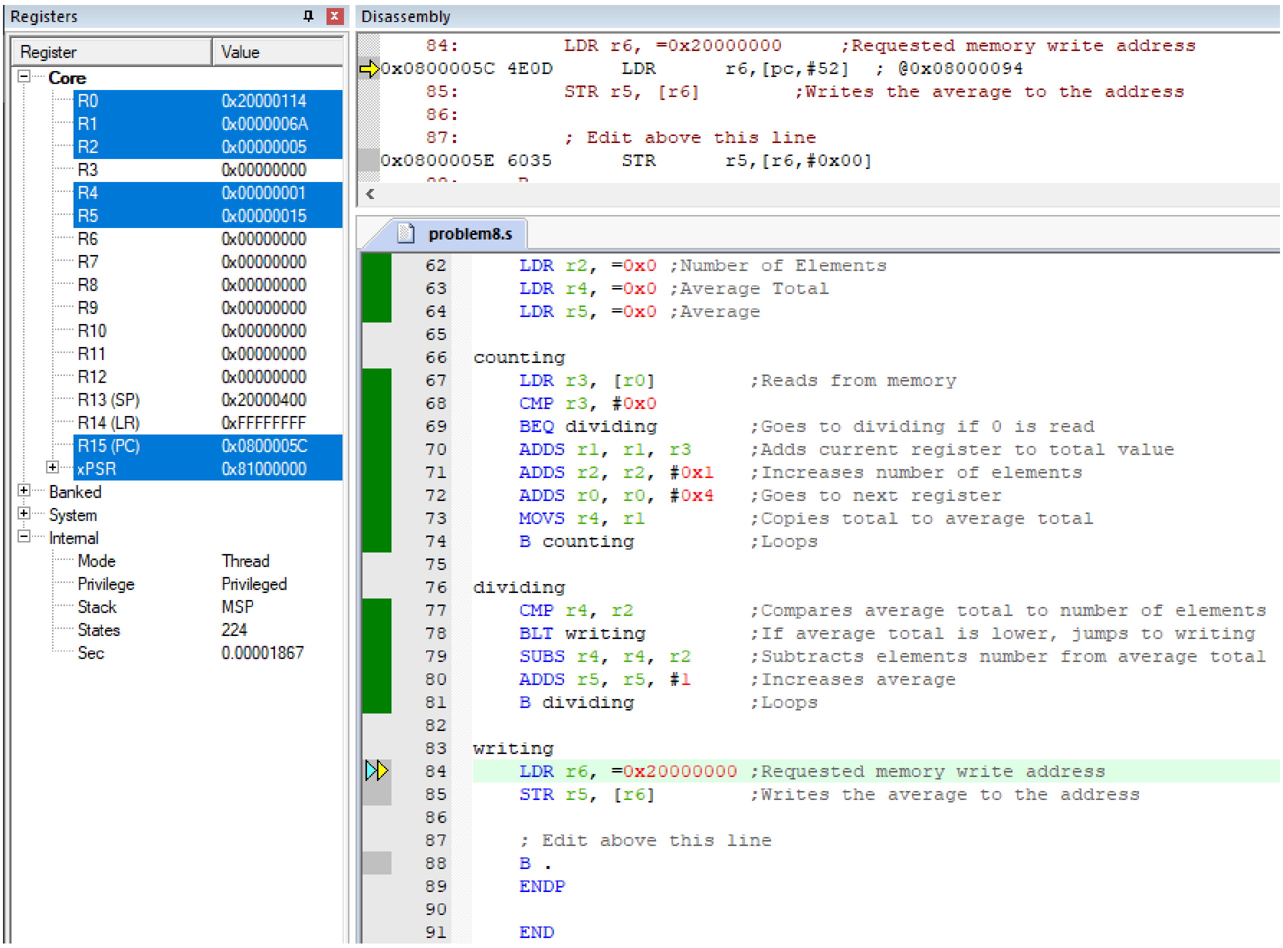
i. Problem 8 – Assembly Average



On the assembly code, firstly values were sent to specific addresses on memory. Unlike the hw2 manual, memory addresses are increased 4 by 4, not 1 by 1. After the memory setup, values are read from the memory. Read values are summed to get a total. Since this microprocessor does not have a divide function, sum is divided by subtraction using the element number. After the dividing, average result is written the required address, “0x20000000”.

```
    ;Setting up the memory
    LDR r0, =0x20000100
    LDR r1, =0xC
    STR r1, [r0]
    LDR r0, =0x20000104
    LDR r1, =0x1B
    STR r1, [r0]
    LDR r0, =0x20000108
    LDR r1, =0x1E
    STR r1, [r0]
    LDR r0, =0x2000010C
    LDR r1, =0x1D
    STR r1, [r0]
    LDR r0, =0x20000110
    LDR r1, =0x8
    STR r1, [r0]
    LDR r0, =0x20000114
    LDR r1, =0x0
    STR r1, [r0]

    LDR r0, =0x20000100 ;Start Address
    LDR r1, =0x0        ;Total Value
    LDR r2, =0x0        ;Number of Elements
    LDR r4, =0x0        ;Average Total
    LDR r5, =0x0        ;Average

counting
    LDR r3, [r0]        ;Reads from memory
    CMP r3, #0x0
    BEQ dividing        ;Goes to dividing if 0 is read
    ADDS r1, r1, r3     ;Adds current register to total value
    ADDS r2, r2, #0x1   ;Increases number of elements
```

```
    ADDS r0, r0, #0x4   ;Goes to next register
    MOVS r4, r1         ;Copies total to average total
    B counting          ;Loops

dividing
    CMP r4, r2          ;Compares average total to number of elements
    BLT writing         ;If average total is lower, jumps to writing
    SUBS r4, r4, r2     ;Subtracts elements number from average total
    ADDS r5, r5, #1     ;Increases average
    B dividing          ;Loops

writing
    LDR r6, =0x20000000 ;Requested memory write address
    STR r5, [r6]        ;Writes the average to the address
```



J. References

1. C Random Number Generation, stackoverflow, https://stackoverflow.com/questions/9492581/c-random-number-generation-pure-c-code-no-libraries-or-functions
2. C Library function, Tutorialspoint, https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm
3. Instruction set summary, arm.com, https://developer.arm.com/documentation/ddi0484/b/Programmers-Model/Instruction-set-summary
4. Brian Kernighan’s Algorithm, geeksforgeeks, https://www.geeksforgeeks.org/count-set-bits-in-an-integer