實驗二 ARM Assembly I

第一組 0410137 劉家麟 0416324 胡安鳳

# 實驗目的

熟悉基本ARMv7組合語言語法使用。

在這次實驗中需要同學了解

* 如何利用條件跳躍指令完成程式迴圈的操作
* 算數與邏輯操作指令使用
* 暫存器(Register)使用與基本函式參數傳遞
* 記憶體與陣列存取
* Random Number Generator 使用 (加分)
* FPU instructions使用 (加分)

# 實驗原理

請參考上課Assembly部分講義。

# 實驗步驟

## Hamming distance

計算兩個數長度為half-word(2bytes)的漢明距離，並將結果存放至result變數中。  
Please calculate the Hamming distance of 2 half-word (2 bytes) numbers, and store the result into the variable “result”.

|  |
| --- |
| .data  result: .byte 0  .text  .global main  .equ X, 0x55AA  .equ Y, 0xAA55  hamm:  //TODO  bx lr  main:  movs R0, #X //This code will cause assemble error. Why? And how to fix.  movs R1, #Y  ldr R2, =result  bl hamm  L: b L |

Note: 漢明距離主要是利用XOR計算兩數bit間差異個數，計算方式可參考下列連結。

Note: Hamming distance is basically using the XOR function to calculate the different number of “bits” of two numbers. Please check the following link for more information.

Reference: <https://en.wikipedia.org/wiki/Hamming_distance>

1. **Problem definition and algorithm abstract**

Hamming distance is the different bits between two numbers represented in binary.  
e.g. 63=(111111)bin 64=(1000000)bin, then the Hamming distance is   
0111111  
1000000 distance = 7

It is a traditional bit manipulation problem.

The algorithm is as simple as just XOR (the instruction in ARM is EOR) the two number since once there is a different bit, XOR will make it be 1.

The ***c code*** is represented as follow

|  |
| --- |
| **int** bitCount(**unsigned** **int** n) {  **int** counter = 0;  **while**(n)  {  counter += n % 2;  n >>= 1;  }  **return** counter;  } |

Once the number has a bit, %2 will cause the remainder be 1, by this method we can accumulate it into the counter of counting the bits.

Finally, we >>= the number again and again till it reaches 0.

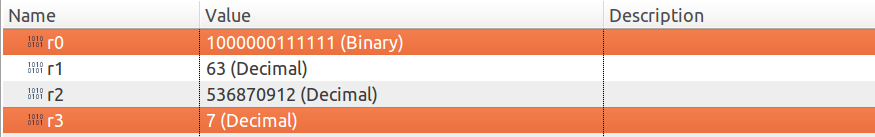
The ARM assembly is represented as follow

|  |
| --- |
| hamm:  //TODO  eor R0, R0, R1 //xor for how many bits are 1  **add** R4, R0, #0 //n=r0  whileloop:  **cmp** R4, #0 //while(n)  beq return  // counter as r3, the result of n%2 (which is the same as n&1) save at R5  **and** R5, R4, #1 //R5 for increment value in R3  **add** R3, R3, R5 // counter+=n%2  lsr R4 ,R4 ,#1 //n>>=1  b whileloop  return:  bx lr |

1. **Test cases tested**

**TEST1:**

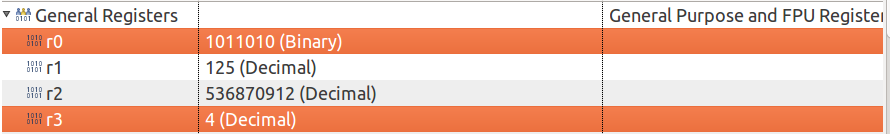
e.g. a=0x1000 =1000000000000  
 b=0x3f =0000000111111  
 a xor b =1000000111111 stored in r0



The correct answer should be 7, stored in r3.

**TEST2:**

e.g. a=39 =0100111  
 b=125 =1111101  
 a xor b =1011010 stored in r0

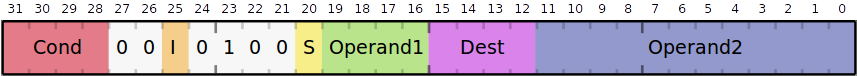


The correct answer should be 4, stored in r3.

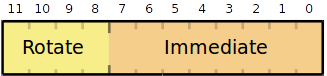
1. **About the error of 0x55AA and 0xAA55**

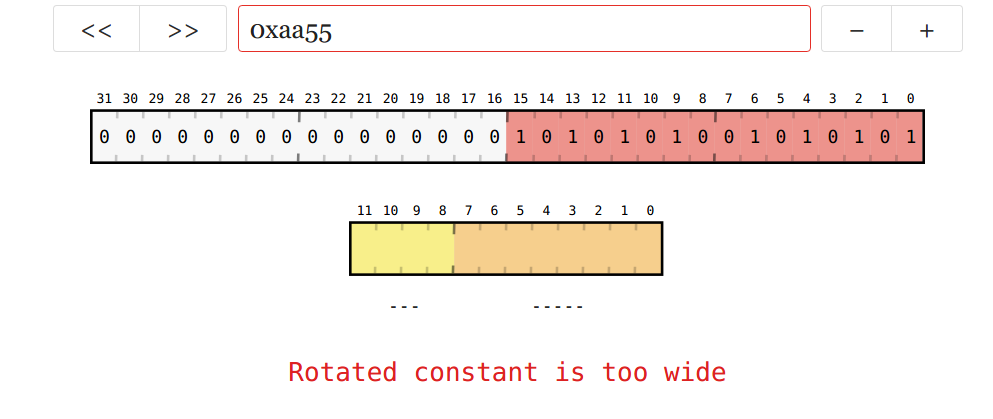
Reference link <https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/>

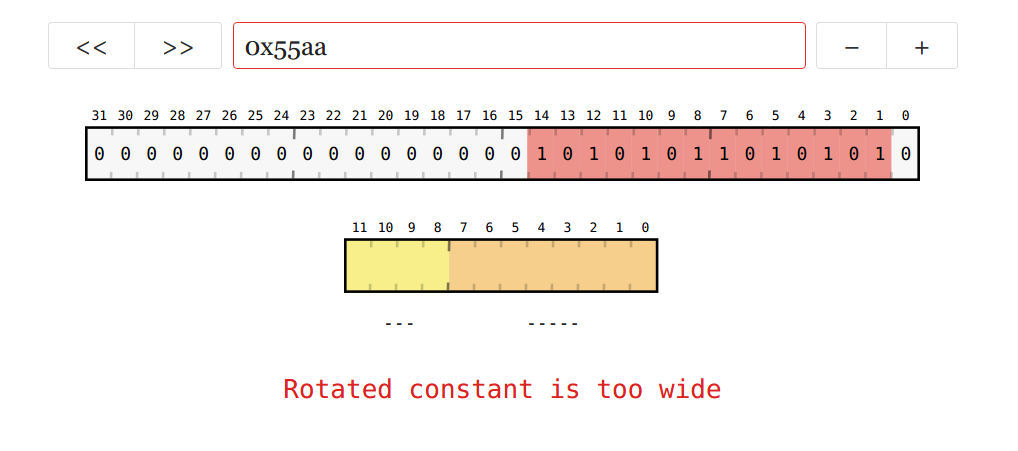
The 0x55aa and 0xaa55 cannot be encoded by the method of link provided up, namely the ROTATING METHOD of using 12 bits immediate value (in ARM instruction architecture) to represent larger immediate value.

Here's the bit layout of an ARM data processing instruction:

But ARM doesn't use the 12-bit immediate value as a 12-bit number. Instead, it's an 8-bit number with a [4-bit rotation](http://en.wikipedia.org/wiki/Circular_shift), like this:



Which, in conclusion, cannot represent the value of 0x55aa and 0xaa55.



1. **Rethink of 3**

Q: So how do we represented other values that are not accepted before?

A: Maybe we can use more register to represent then (by decomposition) move some accepted value to 2 or even more registers to add up (or some linear combinations) to form the unaccepted immediate value aforementioned.

## Fibonacci serial

宣告一數值N ()，計算Fib(N)並將回傳值存放至R4暫存器  
Declare a number N() and calculate the Fibonacci serial Fib(N). Store the result into register R4.

|  |
| --- |
| .text  .global main  .equ N, 20  fib:  //TODO  bx lr  main:  movs R0, #N  bl fib  L: b L |

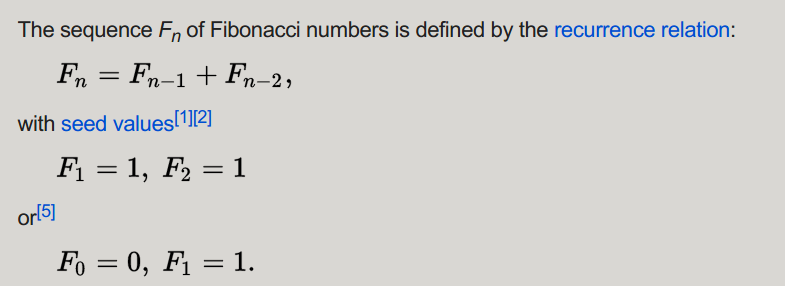
Note: 回傳值格式為signed integer，若Fib[N]結果 overflow的話回傳-2, 當N數值出過範圍時fib回傳-1，計算方式可參考下列連結

Note: The returned value should be in signed integer format. If the result of Fib(N) overflows, you should return -2. If the value of N is outside the accepted range, you should return -1. Check the following link for more details of the calculation.

Reference: <https://it.wikipedia.org/wiki/Successione_di_Fibonacci>

1. **Problem definition and algorithm abstract**

The well-known mathematical sequence defined as below

  
(source Wikipedia)

In a nutshell, I write a checker named cmp\_greater\_than\_1: and cmp\_less\_than100: to check if the input value is in the range, otherwise, minus r4 by 1 (r4 is originally initialized with 0) and return.

If it is really in the range, then go to Fibonacci main function, by using the loop method (the recursion may be too difficult to implement in ARM assembly)

Initialize the first →sec→fib, where fib=first+sec

Then move the 3 continuous numbers forward till reaches end.

By using the cmp, we may get the result of cmp value to branch, acquiring the method of conditional move like if/else or looping condition check in C.

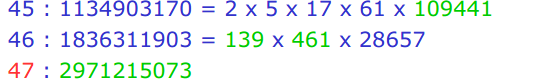
Finally, a BVS instruction, which means BRANCH IF OVERFLOW SET SIGNED.

Since we use adds, the Z V C N flags in ARM assembly will be updated and this BVS can successfully get flags with an eye to determining whether to branch or not.

The rest of detailed algorithm concepts have been written in the comments of the source code.

|  |
| --- |
| fib:  //TODO  //check if N is **in** the range, let r4= -1 for OUT\_OF\_RANGE  cmp\_greater\_than1:  **cmp** r0, #1  bge cmp\_less\_than100  //if OUT\_OF\_RANGE  **movs** r4, #0  **sub** r4, r4, #1  b return  cmp\_less\_than100:  **cmp** r0, #100  ble fibonacci\_main  //if OUT\_OF\_RANGE  **movs** r4, #0  **sub** r4, r4, 1  b return //br for better manipulation??  fibonacci\_main:  **movs** r4, #1 //first prototype for special-testcase judge  **cmp** r0, #1 //fibonacci f(1)=1 --> 1 1 2 3 5 8 13...  beq return  **cmp** r0, #2  beq return  **movs** r1, #1 //first  **movs** r2, #1 //second  **movs** r4, #0 //fib(n)  **movs** r5, #2 //fibonacci counnter start at 2 (modification for overflow detection)  for\_loop:  adds r4, r1, r2 //third=fir+sec adds will update the flag!! FOR SURE  **movs** r1, r2 //fir=sec  **movs** r2, r4//sec=third  **add** r5, r5, #1//increment the counter by 1  bvs overflow\_return //f(48)will cause overflow **in** 32bit intege    **cmp** r5, r0 //compare if it is still **in** the fib range  blt for\_loop//back to **loop** again  return:  bx lr  overflow\_return:  **movs** r4, #0  subs r4, r4, #2  bx lr |

1. **Test cases tested**
2. f(46) = 1836311903



1. f(47) exceeds the value of 2^31-1 can represents so we set r4 = -2



1. f(200) is not in the range



Does not store in r0, directly back to main (r5 will be only stored in Fibonacci main function)

1. **Overflow detection**

Mentioned above with BVS instruction and its explanation.

Reference: <https://community.arm.com/processors/b/blog/posts/condition-codes-1-condition-flags-and-codes>

1. **Problem encountered and solutions**

(1) I first found that the bvs instruction was in vain, nonetheless it turned out to be that add was used instead of adds, only the adds will update the Z C V N flags in arm architecture.

(2) bvs by branching if a register is overflow, once a register exceeds 2^31-1(and in Fibonacci is f(47)) the overflow flag should be set and set r4 to be -2.

However, I originally set the group of 3 numbers in the order of fib, fir, sec where sec=fib+fir. Problem is that even though fib is only f(45), sec will be f(47)

The V flag will still be set, causing the r4 value to be -2, which is totally wrong.

So I debug by using the order of fir sec fibo, where fibo=fir+sec, once the fibo reaches the f(47) it triggers the flag and cause the bvs to branch, terminating the Fibonacci function and set r4=-2.

## Bubble sort

利用組合語言完成長度為8byte的8bit泡沫排序法。

Please implement the Bubble sort algorithm for the 8 bytes data array with each element in 8bits by assembly.

實作要求：完成do\_sort函式，其中陣列起始記憶體位置作為輸入參數R0，程式結束後需觀察arr1與arr2記憶體內容是否有排序完成。

Implementation Requirement: Fill-in the do\_sort function. The start address of the array is store in the R0 register. Observe the result of arr1 and arr2 in the memory viewer after calling the do\_sort functions. The two arrays should be sorted.

|  |
| --- |
| .data  arr1: .byte 0x19, 0x34, 0x14, 0x32, 0x52, 0x23, 0x61, 0x29  arr2: .byte 0x18, 0x17, 0x33, 0x16, 0xFA, 0x20, 0x55, 0xAC  .text  .global main  do\_sort:  //TODO  bx lr  main:  ldr r0, =arr1  bl do\_sort  ldr r0, =arr2  bl do\_sort  L: b L |

Note: 注意記憶體存取需使用byte alignment指令，例如：STRB, LDRB

Note: The memory access may require the instructions that support byte-alignment, such as STRB, LDRB.

1. **Problem definition and algorithm abstract**

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

**Pseudo Code Implementation:**

|  |
| --- |
| void bubble\_sort(int arr[], int len)  {  int i, j, temp;  **for** (i = 0; i < len - 1; i++)  **for** (j = 0; j < len - 1 - i; j++)  **if** (arr[j] > arr[j + 1])  {  temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  } |

The ARM assembly is represented as follow

More detailed code explanations have been written in the comments part.

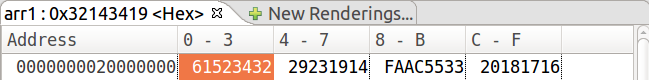
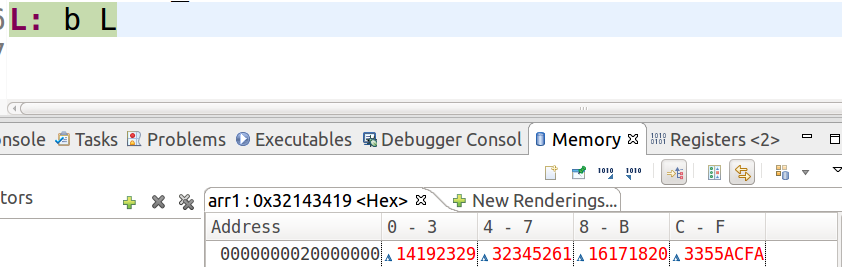
|  |
| --- |
| do\_sort:  //TODO arr from 0 to 7  **movs** r3, #7 //i<len-1-i part  **movs** r1, #0 //**int** i=0      for\_loop\_outer: //i=[0,6]  **movs** r2, #0 //**int** j=0  **sub** r9, r3, r1//len-1-i          for\_loop\_inner:  **add** r4, r0, r2 //**offset** **in** **byte** get arr+j address, store **in** r4  ldrb r5, [r4]//dereference to get value r5 as arr[j]  **add** r6, r4, #1 //**offset** **in** **byte** get arr+j address ,store **in** r6  ldrb r7, [r6]  **cmp** r5,r7  blt swap              swap\_is\_done:  //j++ then j<len-1-i  **add** r2, r2, #1  //r9 as len-1-i r9 = r3(7 which is len-1 )- r1 (which is i)  **cmp** r2,r9  blt for\_loop\_inner  //i++ then i< len -1  **add** r1, r1, #1  **cmp** r1, #7  blt for\_loop\_outer  b return //job is done      swap: //using r8 as temp value for swapping use STRB for storing **in** memory  **movs** r8, r5  **movs** r5, r7  **movs** r7, r8  //store the fucking value back in memory  strb r5, [r4]  strb r7, [r6]  b swap\_is\_done  return:  bx lr |

1. **Result**

Since the data store is 0x\_ \_ which is a byte, so it fits perfectly into the memory block where a block is 4 bytes (a memory row is 4\*4=16 bytes), so we can easily see the array are successfully sorted descending.

Address 0x20000000~0x20000007 stores arr1 and 0x20000008~0x2000000F stores arr2.

Or sorting in ascending order by bgt



1. **Problem encountered and solutions**
2. Originally the program terminated immediately after swap, and it turned out that I should back to the loop using branch rather than writing nothing, otherwise, the program will keep going to the end.

|  |
| --- |
| swap: //using r8 as temp value for swapping use STRB for storing **in** memory  **movs** r8, r5  **movs** r5, r7  **movs** r7, r8  //store the fucking value back in memory  strb r5, [r4]  strb r7, [r6]  b swap\_is\_done(without this, program will keep going till return (bx lr to main funtion), through this error, I realize that the ARM assembly is executing in sequence order) |

1. There are i and j , j is in the inner loop, once the inner loop terminated, j should be reset to 0. I once forgot to do this and the memory pointer went so far away to arr2 , causing the result error.

|  |
| --- |
| for\_loop\_outer: //i=[0,6]  **movs** r2, #0 //**int** j=0  **sub** r9, r3, r1//len-1-i |

for\_loop\_outer: Had it been the C code, it may cause the segmentation fault of ERROR\_OUT\_OF\_RANGE

## Monte-Carlo Method for Estimating Pi with FPU and RNG (加分題 10%)(Optional problems with additional 10% score)

透過STM32L476晶片上面的Random Number Generator硬體來產生亂數，並結合FPU使用進一步估算Pi的值

Using the Random Number Generator hardware on STM32L476 to generate numbers for estimating the value of Pi by using the FPU.

**3.4.1 Enabling FPU (Floating Point Unit) and Floating Point Manipulation**

請參考M4 programming manual.pdf 來開啟FPU計算功能，並進行下列運算

Please check the M4 programming manual to enable the functionality of FPU and do the following calculation.

|  |
| --- |
| .syntax unified  .cpu cortex-m4  .thumb  .data  x: .float 0.123  y: .float 0.456  z: .word 20  .text  .global main  enable\_fpu:  //Your code start from here  bx lr  main:  bl enable\_fpu  ldr r0,=x  vldr.f32 s0,[r0]  ldr r0,=y  vldr s1,[r0]  vadd.f32 s2,s0,s1  // Your code start from here  **//Calculate the following values using FPU instructions**  **//and show the register result in your report**  // s2=x-y  // s2=x\*y  // s2=x/y  // load z into r0,  // copy z from r0 to s2,  // convert z from U32 to float representation F32 in s2  // calculate s3=z+x+y  L: b L |

Q3.4.1.1: 如果enable\_fpu留空，程式會停在哪裡？為什麼？  
If the enable\_fpu function is empty in the above code, where will the program stop at and why?

Program will halt at this line vldr.f32 s0,[r0], which is the very first code to use the floating point instruction.

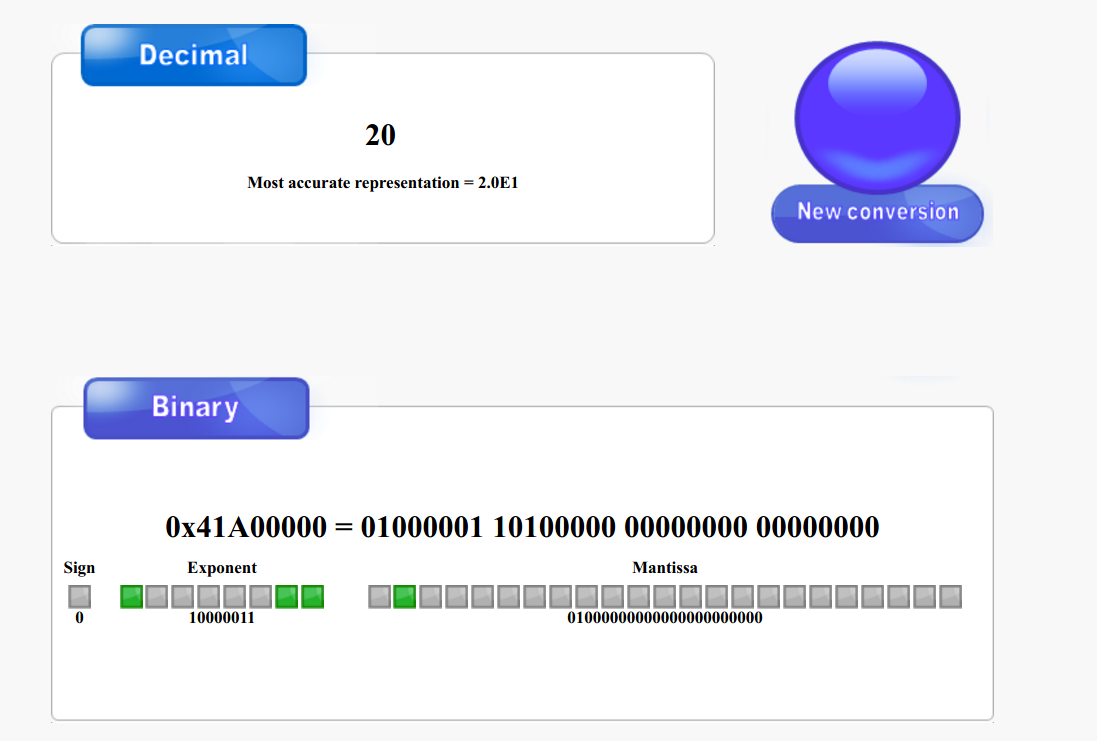
To be more specific and precise, it will halt at line96 of startup\_stm32, b infinite\_loop to search for something to cope with floating point, but nothing is there.

So the program, in fact, search nothing and halt there.

Q3.4.1.2: 為什麼需要將U32轉成F32格式再相加？如果想直接load 一個值代表20到s2中不需轉換就能運算，應該將z修改成多少才能得到相同答案？  
Why do we need to convert the U32 to F32 format before the addition? If we want to directly load a value represents 20 for calculation without further format conversion, what value should we modify to z in order to get the same answer?

We need the pseudo type conversion since the s2 is still not the REAL FLOAT TYPE. (IEEE754 is the correct type) so we need TYPE CONVERSION from int to float.

If we want to directly use it, we should do it according to IEEE754 and represent it in binary

0x41A00000 = 01000001 10100000 00000000 00000000

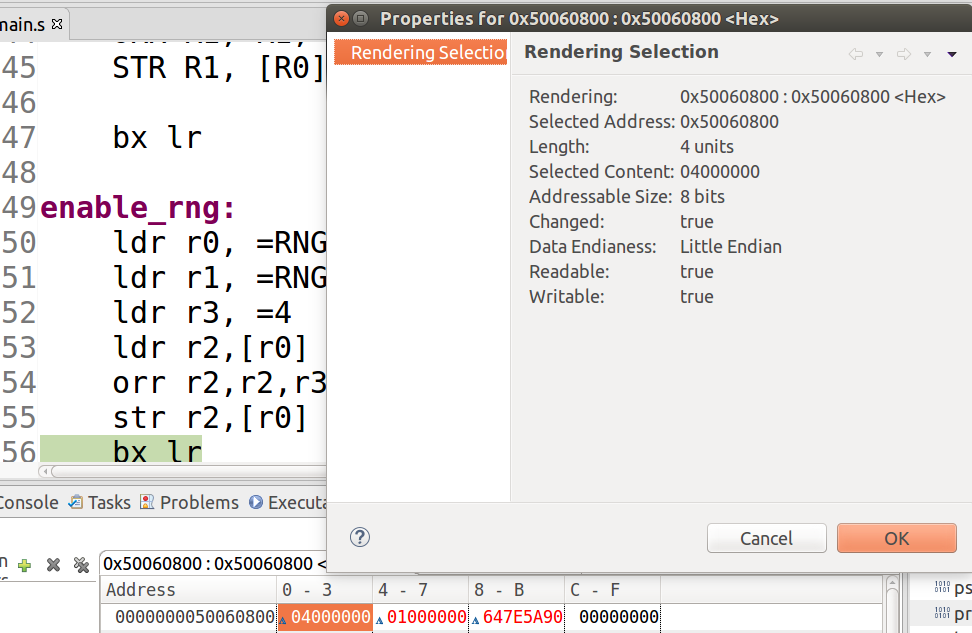
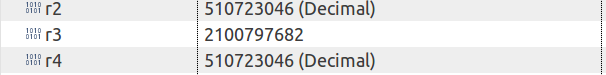
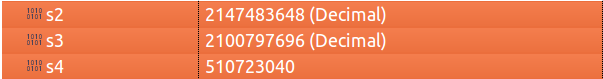
will be correct (+) (1+2^-2)\*2^(131-127)=1.25\*16=20(float)

**3.4.2 Random Number Generator**

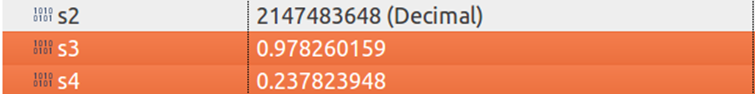
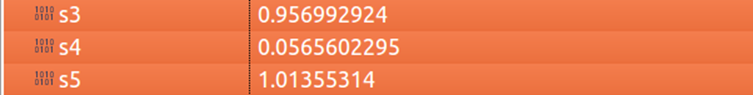
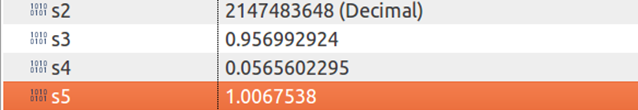
開啟RNG功能，產生一組(x,y)點在單位平面裡.

Enable the functionality of RNG and generate a sample point in the unit area.

|  |
| --- |
| .syntax unified  .cpu cortex-m4  .thumb  .text  .global main  .equ RCC\_BASE,0x40021000  .equ RCC\_CR,0x0  .equ RCC\_CFGR,0x08  .equ RCC\_PLLCFGR,0x0c  .equ RCC\_CCIPR,0x88  .equ RCC\_AHB2ENR,0x4C  .equ RNG\_CLK\_EN,18  // Register address for RNG (Random Number Generator)  .equ RNG\_BASE,0x50060800 //RNG BASE Address  .equ RNG\_CR\_OFFSET,0x00 //RNG Control Register  .equ RNGEN,2 // RNG\_CR bit 2  .equ RNG\_SR\_OFFSET,0x04 //RNG Status Register  .equ DRDY,0 // RNG\_SR bit 0  .equ RNG\_DR\_OFFSET,0x08 //RNG Data Register (Generated random number!)  //Data Settings for 3.4.4  .equ SAMPLE,1000000  set\_flag:  ldr r2,[r0,r1]  orr r2,r2,r3  str r2,[r0,r1]  bx lr  enable\_fpu:  //Your code in 3.4.1    bx lr  enable\_rng:  //Your code start from here  //Set the RNGEN bit to 1    bx lr  get\_rand:  //Your code start from here  //read RNG\_SR  //check DRDY bit, wait until to 1  //read RNG\_DR for random number and store into a register for later usage  bx lr  main:  //RCC Settings  ldr r0,=RCC\_BASE  ldr r1,=RCC\_CR  ldr r3,=#(1<<8) //HSION  bl set\_flag  ldr r1,=RCC\_CFGR  ldr r3,=#(3<<24) //HSI16 selected  bl set\_flag  ldr r1,=RCC\_PLLCFGR  ldr r3,=#(1<<24|1<<20|1<<16|10<<8|2<<0)  bl set\_flag  ldr r1,=RCC\_CCIPR  ldr r3,=#(2<<26)  bl set\_flag  ldr r1,=RCC\_AHB2ENR  ldr r3,=#(1<<RNG\_CLK\_EN)  bl set\_flag  ldr r1,=RCC\_CR  ldr r3,=#(1<<24) //PLLON  bl set\_flag  chk\_PLLON:  ldr r2,[r0,r1]  ands r2,r2,#(1<<25)  beq chk\_PLLON  //Your code start from here  //Enable FPU,RNG  //Generate 2 random U32 number x,y  //Map x,y in unit range [0,1] using FPU  //Calculate the z=sqrt(x^2+y^2) using FPU  //Show the result of z in your report  L: b L |

1. The picture shows that RNG\_ENABLE is done.  
   
2. The picture shows that 2 random numbers x for r3 and y for r4 respectively have been successfully generated.  
   
3. using vcvt.f32.s32 and vabs.f32 to firstly convert the data in int into float then using abs to get the absolute value for mapping in the future.  
     
   Inaccuracy is due to IEEE 754 floating point standard if the data exceeded the 23 bit in floating point representation.  
   單精度二進制小數，使用32個位元存儲。

|  |  |  |
| --- | --- | --- |
| 1 | 8 | 23位長 |
| S | Exp | Fraction |
| 31 | 30至23偏正值 （實際的指數大小+127） | 22至0位編號（從右邊開始為0） |

1. Mapping it into [0,1] by dividing by INT\_MAX = 2^32-1 (since the number generated is in SIGNED FORMAT)  
   
2. Sum z(s5)=x(s3)^2+y(s4)^2 and get sqrt(z)  
     
     
   Inaccuracy due to IEEE 754

**3.4.3 Estimation of Pi**

使用Monte Carlo Method來估算Pi的值

Using Monte Carlo Method to estimate the value of Pi.

Note:

1. Report中請至少附上三次使用一百萬個點估算完的Pi值的register結果截圖

Please attach the screenshots of the register for at least 3 estimation results using 1 million sample points.

1. 請使用3.4.2的程式模板進行修改，以避免修改到RCC設定影響RNG功能

Please use the code template provided in 3.4.2 for this problem. RNG may raise error if the settings of RCC are incorrect.

Reference : http://www.eveandersson.com/pi/monte-carlo-circle

**Results:**

pi=4\*inner\_point\_cnt/sample\_cnt, tried 3 times





