Microprocessor Lab 2 Report

0410137 劉家麟 0416324 胡安鳳

Problem1 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 , %2will 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 follows

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

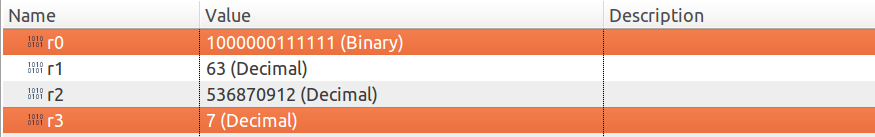
2.Test cases tested

e.g. a=0x1000 =1000000000000

b=0x3f =0000000111111

a xor b =1000000111111 stored in r0

The correct answer should be 7 ,stored in r3

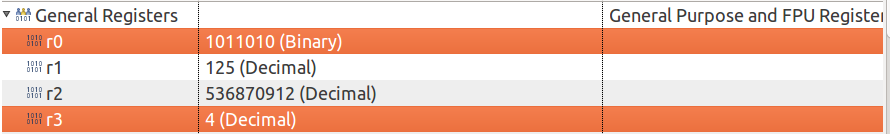


e.g. a=39 = 0100111

b=125 = 1111101

a xor b = 1011010 store in r0

The correct answer should be 4 ,stored in r3

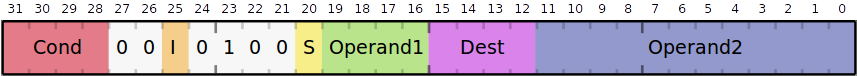


3.About the error of 0x55AA and 0xAA55

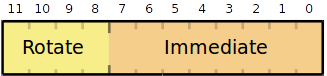
Reference link <https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/>

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.

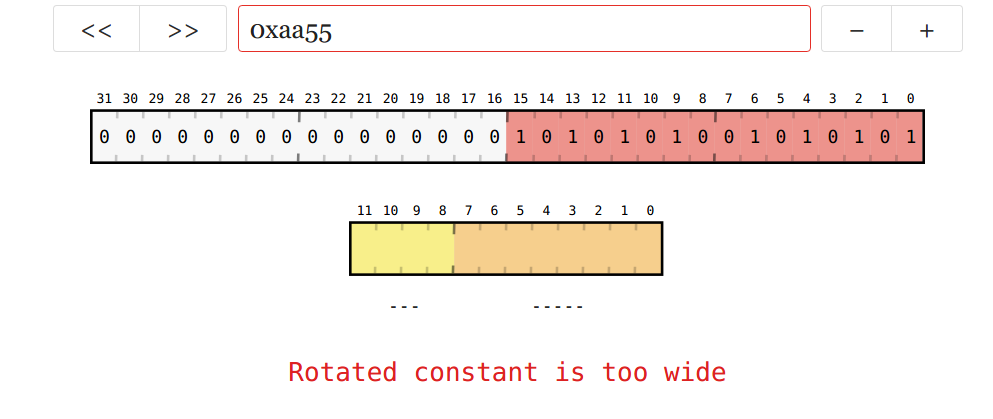
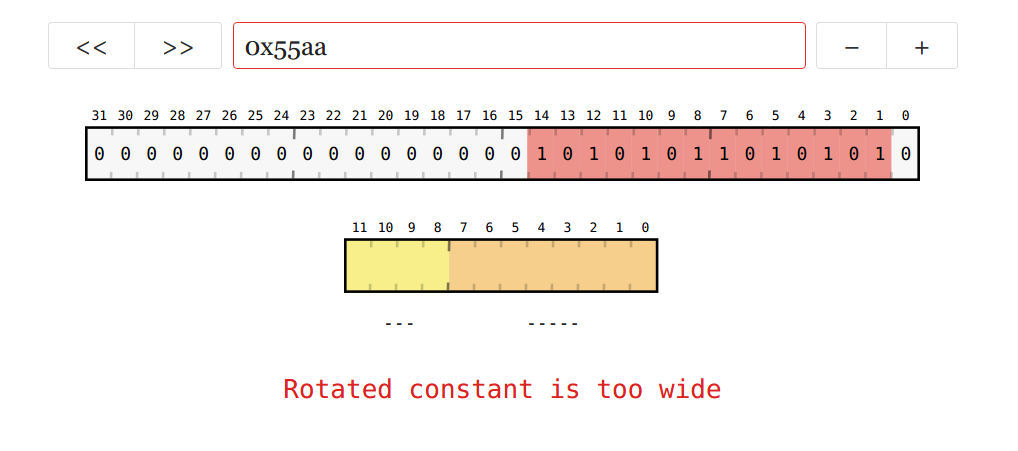
Here's the bit layout of an ARM data processing instruction:



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.



4.Rethink of 3

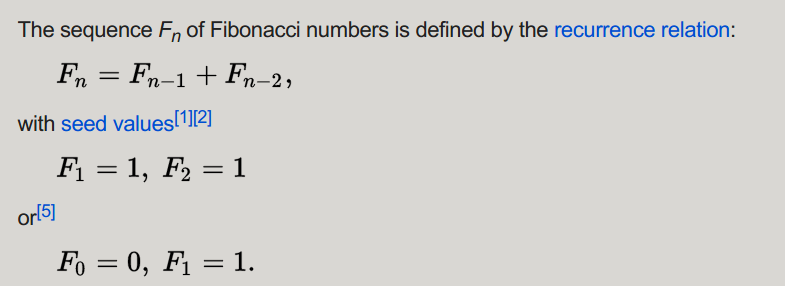
Q:So how do we represented other values that are not accepted before?

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

Problem2 Fibonacci Sequence

1.Problem definition and algorithm abstract

The well known mathematical sequence defined as:



(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, other wise, 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 BEANCH 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 en 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

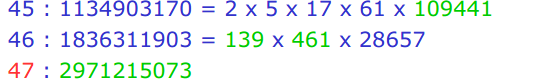
subs r4, r4, #2

bx lr

2.Test cases tested

(1) f(46) = 1836311903





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



(3) 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)

3.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>

4.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 fibonacci(47)) the overflow flag sould 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.

Problem3 Bubble Sort

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.

### Pseudocode implementation

The algorithm can be expressed as (0-based array):

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:

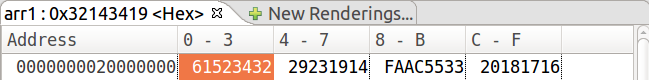
bx lr

2. Result

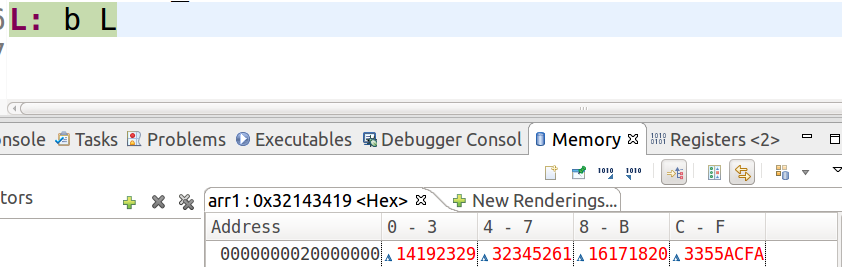
Since the data store is 0x\_ \_ which is a byte, so it fits perfectly into the memory block where a block is 4 bytes(a memory row is 4\*4=16 bytesz ), so we can easily see the array are successfully sorted descending.

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

stores arr2.



Or sorting in ascending order by bgt



3.Problem encountered and solutions

(1) 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)

(2) 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

Problem4(Bonus) FPU and RNG manipulation

4.1 Basic floating point manipulation

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.

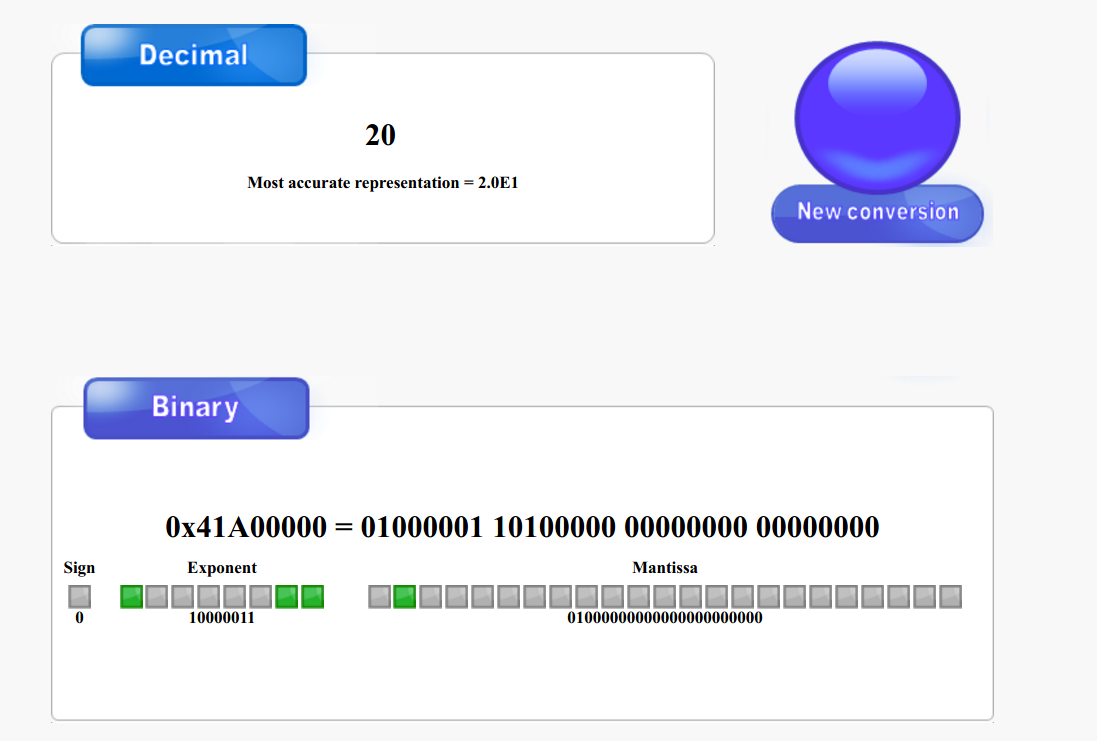
So the program ,in fact, search nothing and halt there.

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?

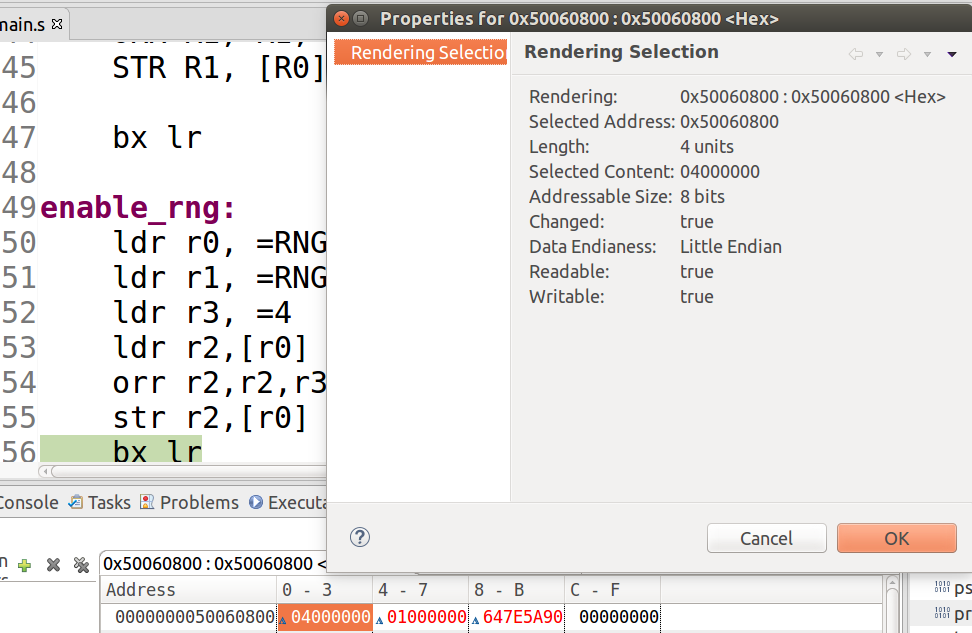
We need the pseudo type conversion since the s2 is still not the REAL FLOAT TYPE. (IEEE754 is the correct type) so wee need TYPE CONVERSION from int to float.

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

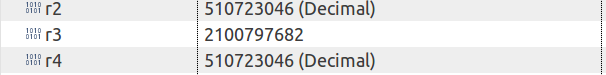
0x41A00000 = 01000001 10100000 00000000 00000000

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

4.2Generating (x,y) in the unit plane

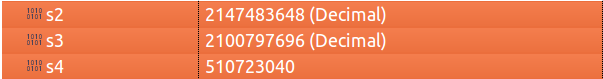
(1)The picture shows that the RNG\_ENABLE is done

.



(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.



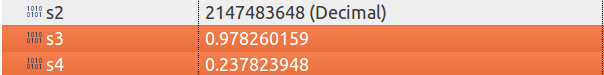
Inaccuracy is due to IEEE 754 floating point standard if the data exceeded the 23 bit

in floating point representation.

單精度二進制小數，使用32個位元存儲。

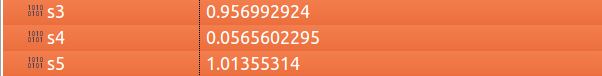
|  |  |  |
| --- | --- | --- |
| 1 | 8 | 23位長 |
| S | Exp | Fraction |
| 31 | 30至23 偏正值（實際的指數大小+127） | 22至0位編號（從右邊開始為0） |

(4)Mapping it into [0,1] by dividing by INT\_MAX = 2^32-1 (since the number generated is in SIGNED FORMAT)



(5)

Sum z(s5)=x(s3)^2+y(s4)^2 and get sqrt(z)



Inaccuracy due to IEEE 754

4.3 Monte Carlo method for pi estimation

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





