Lab2 ARM Assembly I
實驗二 **ARM Assembly I**

第五組 0516322朱蝶、0516059劉嘉豪

1. **Lab objectives** 實驗目的

Familiar with basic ARMv7 assembly language.

In this Lab ,we will learn topics below.

- How to use conditional branch to finish the loop.
- How to use logic and arithmetic instructions.
- How to use registers and basic function parameter passing.
- How to access memory and array.

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

2. **Theory** 實驗原理

Please check the part of course materials of assembly language.

請參考上課Assembly部分講義。

3. **Steps** 實驗步驟

3.1. **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

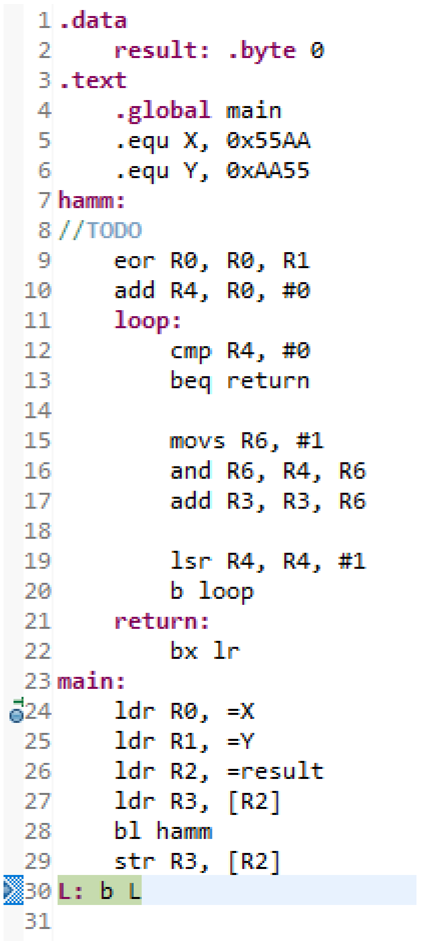
1. Problem and Code

Hamming distance is the number of different bits between two numbers. And we can simply use XOR (the instruction in ARM is EOR) to solve this problem.

The algorithm in C code is like this.

```
int hamm (int n)
{
        int counter = 0;
        while(n)
        {
                counter += n%2;
                n >> =1;
        }
        return counter;
}
```

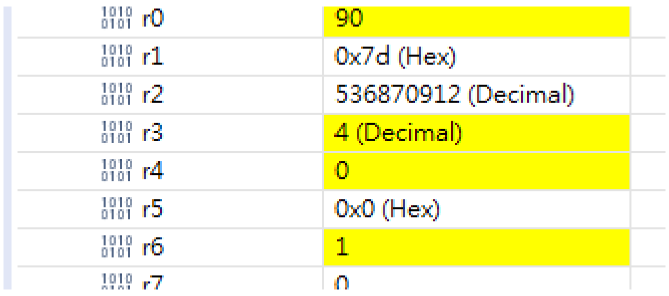
And we implement this algorithm to our assembly code, which is in the following part.



```
.data
    result: .byte 0
.text
    .global main
    .equ X, 0x55AA
    .equ Y, 0xAA55
hamm:
//TODO
    eor R0, R0, R1
    add R4, R0, #0
    loop:
        cmp R4, #0
        beq return

        movs R6, #1
        and R6, R4, R6
        add R3, R3, R6

        lsr R4, R4, #1
        b loop
    return:
        bx lr
main:
    ldr R0, =X
    ldr R1, =Y
    ldr R2, =result
    ldr R3, [R2]
    bl hamm
    str R3, [R2]
L: b L

```

2. Test cases and the results

Test1:
X = 39 =   0100111
Y = 125 = 1111101
X xor Y = 1011010 stored in r0 (90) => ans = 4 stored in r3
This test case is the same as the screen shots.



r0	90
r1	0x7d (Hex)
r2	536870912 (Decimal)
r3	4 (Decimal)
r4	0
r5	0x0 (Hex)
r6	1
r7	0

3. About 0x55AA and 0xAA55

These two numbers cannot be encoded by the method taught by the professor in class. This method is that ARM doesn't have 12-bit to store the value, while there are just 8 bits for storing numbers and 4 bits for rotation. Since these two numbers are like this, the 8 bits can't represent the value of 0x55AA and 0xAA55.
0x55AA = 0000000000000000101010110101010
0xAA55 = 0000000000000001010101001010101
So we need to change movs to ldr, so that the number can be load into r0, r1 correctly.

4. Further thinking

How to store these kind of numbers in the 8 bit space? We think that maybe we can try to break down these numbers to several parts and store them in.

3.2. Fibonacci serial

宣告一數值N (1≤*N* ≤100)，計算Fib(N)並將回傳值存放至R4暫存器
Declare a number N(1≤*N* ≤100) 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
```

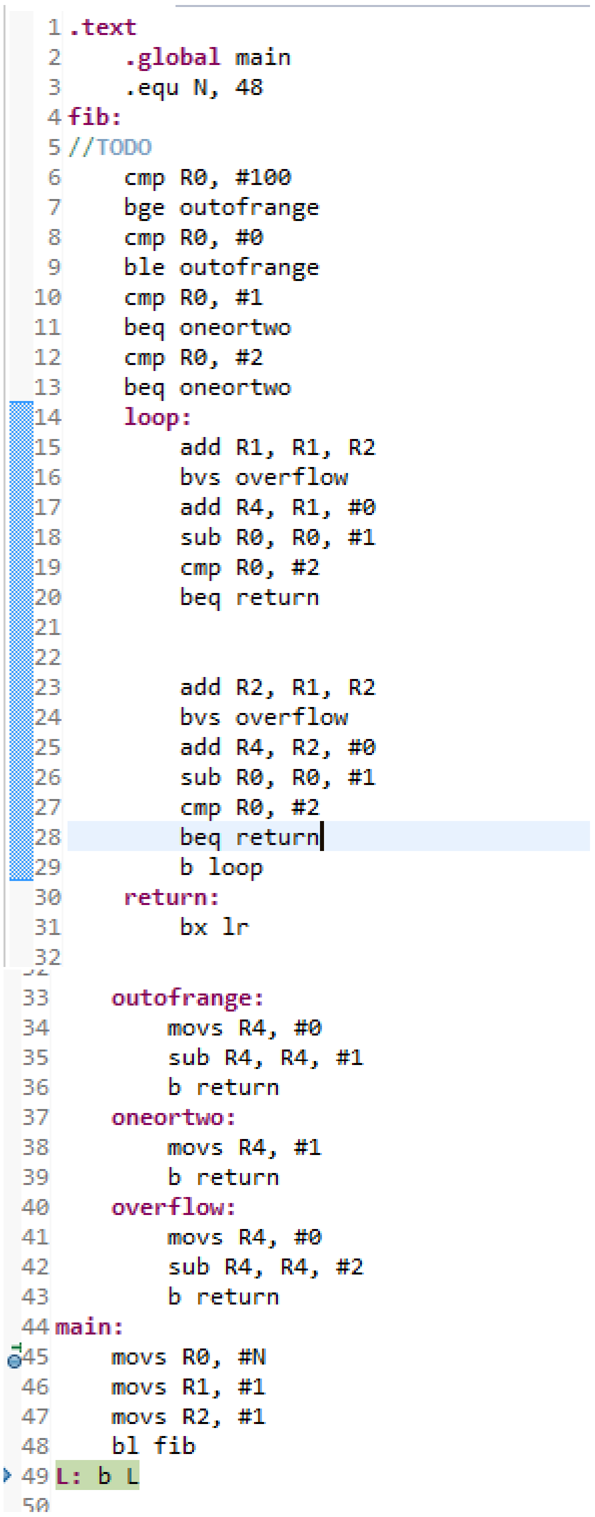
Note: 回傳值格式為signed integer，若Fib[N]結果 overflow的話回傳-2, 當N數值出過範圍時fib回傳-1，計算方式可參考下列連結

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 and Code

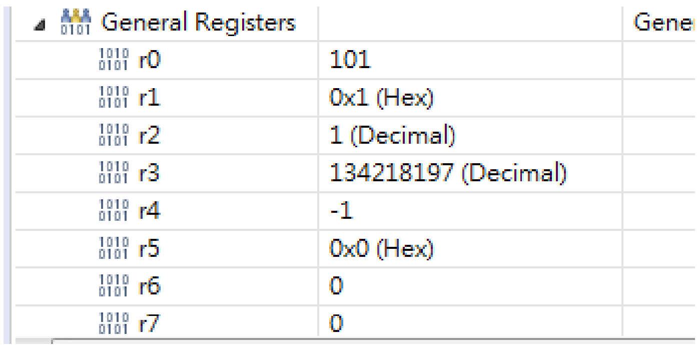
Fibonacci numbers is definced by the recurrence relation:
Fn = Fn-1 + Fn-2,
F1 = 1, F2 = 1 or F0 = 1, F1 = 1.
So first, we need to check whether the input N is between 1 and 100 (check if it is greater than one and smaller than 100), if it is out of range, than minus r4 by 1 (r4 is initialized with 0) and return.
If N is in the range, then check if N is one or two, if it is, then move 1 into r4 and return (F1 = 1, F2 = 1). If it is not one or two, use the recursion above to implement with assembly code.
After adding, we use bvs, which means branch if overflow set signed to check if there is an overflow. Since we adds, the ZVCN flags in ARM will be updated and bvs can successfully get flags to determine whether to branch or not. Then we put the result in R4 and let R0 = R0 – 1 (N-1), then compare R0 with 2, since we started from F3.
The detailed algorithm is like the code written below.



```
.text
    .global main
    .equ N, 48
fib:
//TODO
    cmp R0, #100
    bge outofrange
    cmp R0, #0
    ble outofrange
    cmp R0, #1
    beq oneortwo
    cmp R0, #2
    beq oneortwo
    loop:
        add R1, R1, R2
        bvs overflow
        add R4, R1, #0
        sub R0, R0, #1
        cmp R0, #2
        beq return


        add R2, R1, R2
        bvs overflow
        add R4, R2, #0
        sub R0, R0, #1
        cmp R0, #2
        beq return
        b loop
    return:
        bx lr

    outofrange:
        movs R4, #0
        sub R4, R4, #1
        b return
    oneortwo:
        movs R4, #1
        b return
    overflow:
        movs R4, #0
        sub R4, R4, #2
        b return
main:
    movs R0, #N
    movs R1, #1
    movs R2, #1
    bl fib
L: b L
```

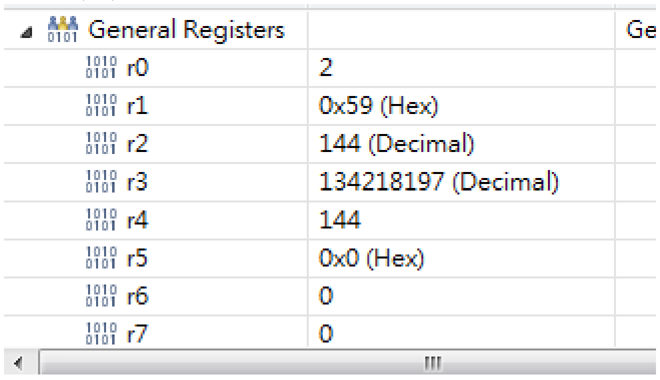
2. Test cases and the results

Test1:
N = 101 (is out of range)
r4 = -1



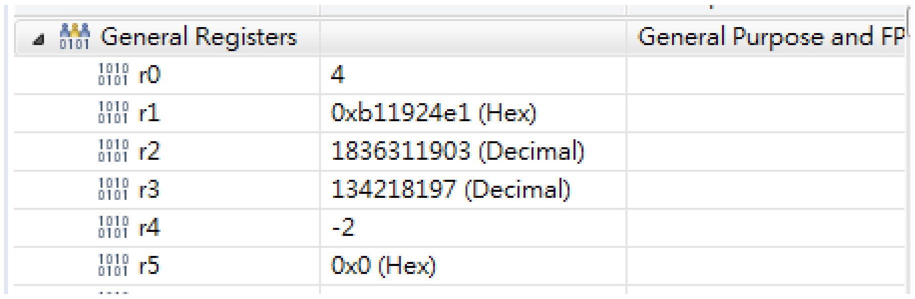
General Registers		Gene
r0	101	
r1	0x1 (Hex)	
r2	1 (Decimal)	
r3	134218197 (Decimal)	
r4	-1	
r5	0x0 (Hex)	
r6	0	
r7	0	

Test 2:
N = 12
R4 = F(12) = 144



General Registers		Ge
r0	2	
r1	0x59 (Hex)	
r2	144 (Decimal)	
r3	134218197 (Decimal)	
r4	144	
r5	0x0 (Hex)	
r6	0	
r7	0	

Test 3:
N = 48 (out of range, exceed 2^31-1)
R4 = -2



General Registers		General Purpose and FP
r0	4	
r1	0xb11924e1 (Hex)	
r2	1836311903 (Decimal)	
r3	134218197 (Decimal)	
r4	-2	
r5	0x0 (Hex)	

3.3. 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
```

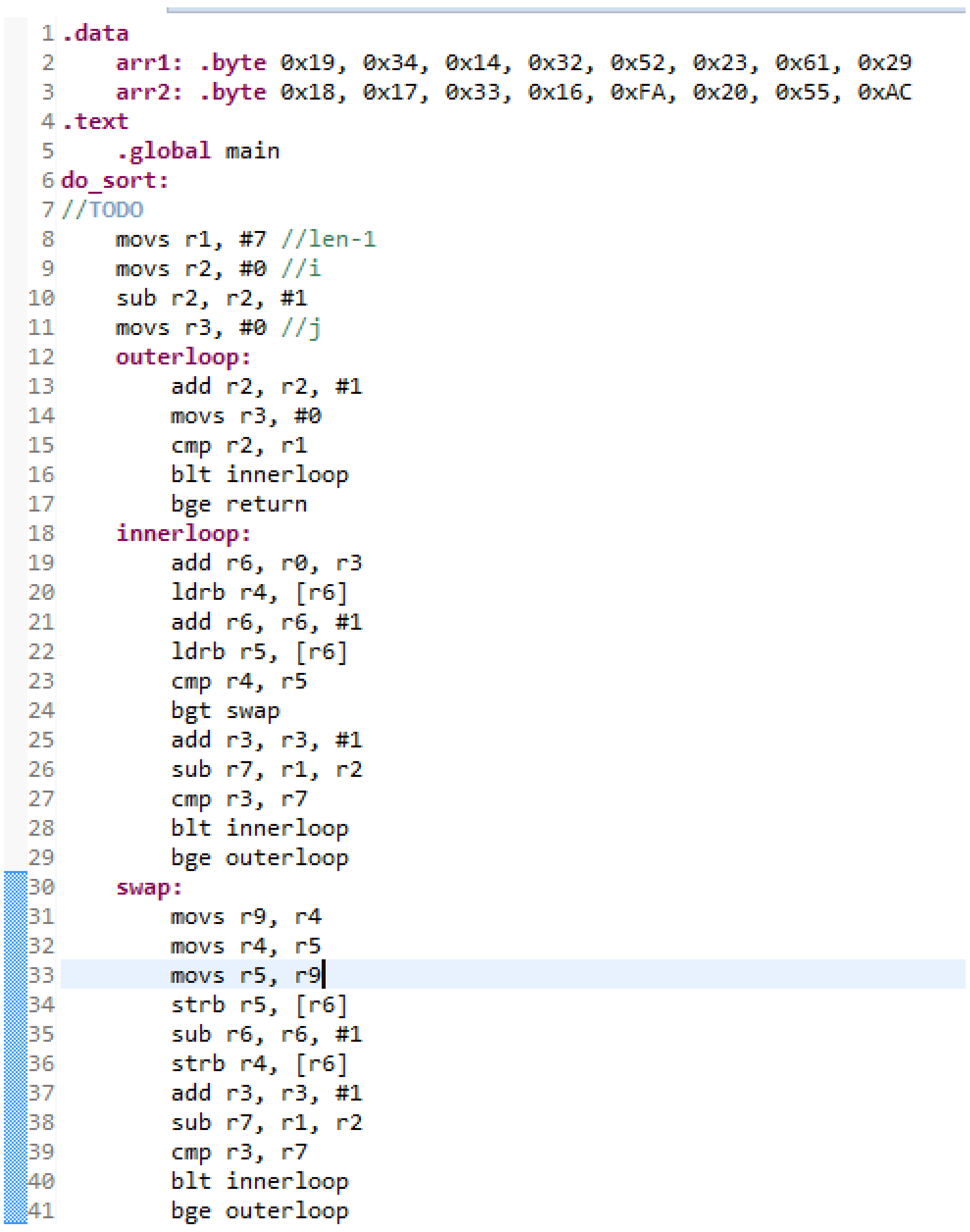
Note: 注意記憶體存取需使用byte alignment指令，例如：STRB, LDRB
Note: The memory access may require the instructions that support byte-alignment, such as STRB, LDRB.

1. Problem and Code

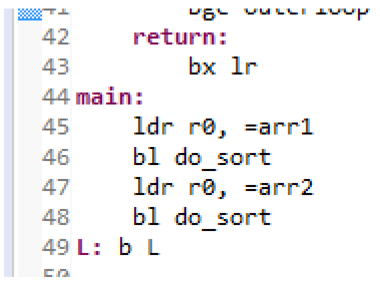
The C code of bubble sort is like this:

```
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;
                        }
}
```

And the assembly code is like this:



```
.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
    movs r1, #7 //len-1
    movs r2, #0 //i
    sub r2, r2, #1
    movs r3, #0 //j
    outerloop:
        add r2, r2, #1
        movs r3, #0
        cmp r2, r1
        blt innerloop
        bge return
    innerloop:
        add r6, r0, r3
        ldrb r4, [r6]
        add r6, r6, #1
        ldrb r5, [r6]
        cmp r4, r5
        bgt swap
        add r3, r3, #1
        sub r7, r1, r2
        cmp r3, r7
        blt innerloop
        bge outerloop
    swap:
        movs r9, r4
        movs r4, r5
        movs r5, r9
        strb r5, [r6]
        sub r6, r6, #1
        strb r4, [r6]
        add r3, r3, #1
        sub r7, r1, r2
        cmp r3, r7
        blt innerloop
        bge outerloop
```



```
    return:
        bx lr
main:
    ldr r0, =arr1
    bl do_sort
    ldr r0, =arr2
    bl do_sort
L: b L
```

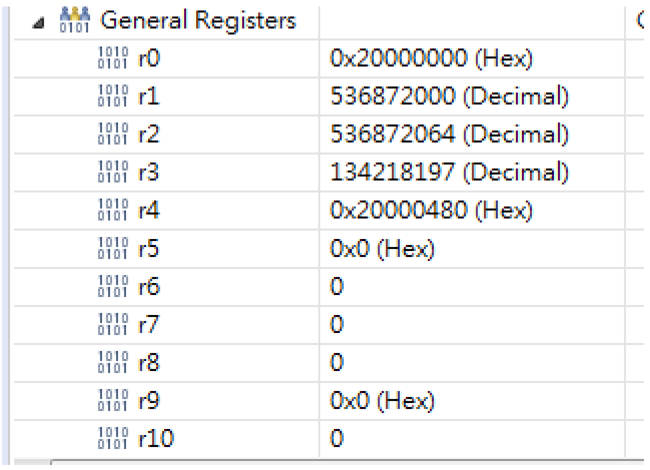
2. Test cases and the results:

Since the data 0x?? is a byte, it fits into the memory block where a block is 4 bytes (a memory row is 4*4 = 16 bytes), so we can use it to see them easily.

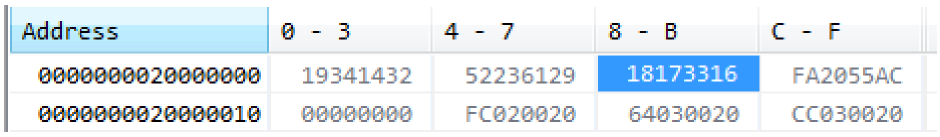
Address 0x20000000 – 0x20000007 stores arr1;

Address 0x20000008 – 0x2000000F stores arr2;

The following is the registers and memory before bubble sorting.

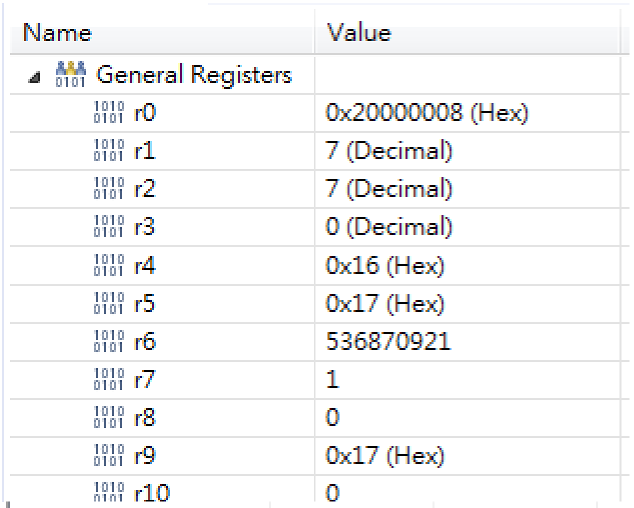


General Registers	
r0	0x20000000 (Hex)
r1	536872000 (Decimal)
r2	536872064 (Decimal)
r3	134218197 (Decimal)
r4	0x20000480 (Hex)
r5	0x0 (Hex)
r6	0
r7	0
r8	0
r9	0x0 (Hex)
r10	0

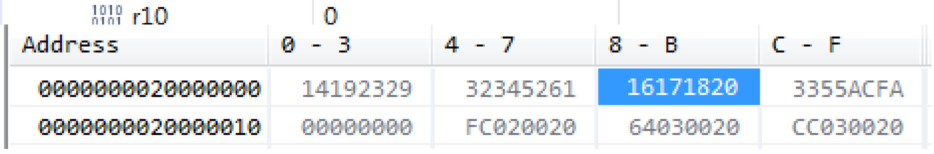


Address	0 - 3	4 - 7	8 - B	C - F
0000000020000000	19341432	52236129	18173316	FA2055AC
0000000020000010	00000000	FC020020	64030020	CC030020

The following are the registers and memory after bubble sorting, we can see that the numbers are ascending.



Name	Value
General Registers	
r0	0x20000008 (Hex)
r1	7 (Decimal)
r2	7 (Decimal)
r3	0 (Decimal)
r4	0x16 (Hex)
r5	0x17 (Hex)
r6	536870921
r7	1
r8	0
r9	0x17 (Hex)
r10	0



Address	0 - 3	4 - 7	8 - B	C - F
0000000020000000	14192329	32345261	16171820	3355ACFA
0000000020000010	00000000	FC020020	64030020	CC030020

3. Problems encountered and solutions

(1) We met many problems and warnings from the assembler because we are not familiar with the instructions. Some of them requires that the destination should be one of the source register, and some requires only r1-r7 register, we can't use r8, r9. The solutions are following the words and don't use those illegal registers or rules.

(2) We forget to use strb to put the value back to the memory and we even accidentally move r0 and then the array is gone. The solutions are to use the debugger step by step and carefully see what is wrong.

Conclusion:

We made several mistakes while finishing lab2. Sometimes we forgot to add the counter, sometimes we forget to put the value back in the memory, and sometimes we forgot to branch out the loop. After all, we think that leaving comments beside the code is really helpful, and using the debugger step by step really helps us a lot to solve our problems. In conclusion, we just need to be focus and carefully so that we won't make stupid mistakes. ☺