Problem 1: (14 points)

a. Consider the following C function

```
int func ( int x ) {    // x is given argument
    int c = -10;
    short sx = x;
    unsigned ux = x;
    return exp;     // exp is given expression
}
```

Assume it is executed on a ***6-bit*** machine that uses ***two's complement*** arithmetic. Assume also that right shifts of signed values are performed ***arithmetically***. "short" integers are encoded with 3 bits. Fill in the following table showing the effect of this function with the ***given argument "x"*** and ***given expression*** "***exp***": (***Note:*** *When the variable is pressed into the parameter stack, the byte is aligned. And you need not fill in entries marked with "**---**".*) (0.5'*20)

x	*exp*	*func(x)*	
		Decimal Representation	*Binary Representation*
0	x	---	**[1]000000**
-27	sx	---	**[2]111101**
27	(short)(x – c)	**[3] -3**	**[4]111101**
-27	x && 1	**[5] 1**	**[6]000001**
-27	ux	**[7]-27**	**[8]100101**
-27	c | !x	**[9]-10**	**[10]110110**
10	(short)((x<<2)>>3)	**[11]-3**	**[12]111101**
30	c - x	**[13]24**	**[14]011000**
30	c + !x	**[15]-1**	**[16]11011**
		0	**0**
-TMax	x	**[17]-31**	**[18]100001**
-27	x * c	**[19]14**	**[20]001110**

b. Given two different functions below:

```
int a ( int x ) {
    return x / 4;
}

int b ( int x ) {
    return x >> 2;
}
```

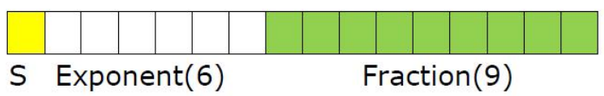
Given the same input x, if they always return the same value, we do take them as the equivalent one. Please explain the reason essentially whether these two functions are equivalent or not. (4')

Not equivalent

When the X is positive, the two functions are the same, but when the X is negative, for the division according to the compiler is to round zero and for right shift is to round down.

Problem 2: (16 points)

Number Conversion: *IEEE 754* single precision float standard ***with a little change*** is illustrated
below.



a. Filling the blanks with proper values. (1'*2)

Normalized: $(-1)^{\textbf{sign}} * (1.\text{fraction}) * 2^{\textbf{exponent-bias}}$, where **bias**=__31__;

Denormalized: $(-1)^{\textbf{sign}} * (0.\text{fraction}) * 2^{\textbf{E}}$, where **E**=__-30__;

Zero: all 0's in all 3 fields

b. Convert the number $\textbf{(-4.1171875)}_{\textbf{10}}$ into *IEEE 754* FP single precision

representation of this problem (in hex). (4')

0xC20F

$(-4.1171875)_{10}$ → $(-100.0001111)_2$ $(-1.000001111 * 2^2)_2$

S:1

E:$(2+31=33)_{10}$ → $(100001)_2$

F:000001111

$(1\ 100001\ 00000111)_2$ 0xC20F

→

c. What is the equivalent value as a decimal number? (3'*2 = 6')

$(0\ 000000\ 000001001)_2$ and **$(0\ 100100\ 100010000)_2$**

$(0.000001001 * 2^{-30})_2$ $(0.5^6 * 2^{-30} + 0.5^9 * 2^{-30})_{10}$ $(0.017578125 * 2^{-30})_{10}$

$(1.125 * 2^{-36})_{10}$

$(1.100010000 * 2^{36-31})_2$ $(110001)_2$ $(49)_{10}$

d. Calculate ***SUM*** of **$(0\ 000000\ 000001001)_2$** and **$(0\ 000100\ 000010100)_2$**, and then round the results to **5** bits to the right of the binary point with ***Round-to-Even*** rounding modes. (***NOTE:*** *Please give your steps detailed*) (4')

$(0\ 000000\ 000001001)_2 = (0.000001001 * 2^{-30})_2$

$(0\ 000100\ 000010100)_2 = (1.000010100 * 2^{4-31})_2$

$(0.000001001 * 2^{-30})_2 = (0.000000001001 * 2^{-27})_2 = (0.000000001 * 2^{-27})_2$

$(1.000010100 * 2^{-27})_2 + (0.000000001 * 2^{-27})_2 = (1.000010101 * 2^{-27})_2$

$(1.00001 * 2^{-27})_2 = (0\ 000100\ 000010000)_2$

Problem 3: (20 points)

Implement the following functions as in Lab1. ***The coding rules are also same as Lab1.***

a. isEqual (4'*1+2'*1)

```
/* [isEqual]
 *– returns 1 if two numbers are equal, 0 if not.
 *
 * Example: isEqual(2, 3) = 0, isEqual(4, 4) = 1.
 *
 * Illegal ops: + - / % !
 *
 * MaxOp: 20 (2')
 */

int isEqual( int x, int y ) {
/* Please fill your code in the answer paper */
   int z = x ^ y;
   z = z | (z >> 16);
   z = z | (z >> 8);
   z = z | (z >> 4);
   z = z | (z >> 2);
   z = z | (z >> 1);
   return (~z)&1;
```

b. logicalShift (1'*6)

```
/* [logicalShift]
 * – shift x to the right by n, using a logical shift.
 * can assume that 1 <= n <= 31.
 *
 * Examples: logicalShift(0x87654321,4) = 0x08765432,
 *
 * Legal ops: ~ & ^ | + << >>
 */

int logicalShift (int x, int n) {
   int mask= 0x7f << __[1]_24_ | 0xff << __[2]_16_ | 0xff << __[3]_8_
| 0xff;
   return ((x >> __[4]_n_) __[5]_&_ ( mask >> (__[6]_n + (~0)_));
}
```

c. satAdd (1'*8)

```
/*satAdd
 *– adds two numbers but when positive overflow occurs,
 * returns maximum possible value, and when negative
 * overflow occurs, it returns minimum positive value.
 *
 * Examples: satAdd(0x40000000,0x40000000) = 0x7fffffff
 * satAdd(0x80000000,0xffffffff) = 0x80000000
 *
 * Legal ops: ! ~ & ^ | + << >>
 */
```

```
int satAdd(int x, int y) {
  int signbit = 31;
/* buzz is all 1's if no overflow. if overflow, return MaxInt or MinInt
 * depending on which way we hit the wall
 */
  int buzz = ((x__[1]_^_y) | (x^__[2]_~_ (x+__[3]_y_))) >> signbit;
  int result = (__[4]_buzz_ & (x __[5]_+_ y)) __[6]_|_ (~buzz & (((x+__[7]_y_) >> signbit) __[8]_^_ (1L << signbit)));
  return result;
}
```

```
struct my_struct{
      char a;
      int x[2];
      char y[3];
};
union{
      struct my_struct struct_data;
      char array_data[11];
} data;
```

Problem 4: (11 points)

Suppose we have following code in C, which will run on ***Linux/X86***: (1'*11)

```
//set all bytes of data to zero
memset(&data,0,sizeof(data));

//Will output 0:0x00a43503
printf("0:0x%.8x\n",0xa43503);

printf("1:%d\n2:%d\n3:%d\n4:%d\n",sizeof(struct my_struct)
      ,sizeof(data.struct_data),sizeof(data.array_data),sizeof(dat
a));
for(int i=0;i<=13;i++)
      data.array_data[i]=i;

printf("5:0x%.8x\n",(int)data.struct_data.a);
printf("6:0x%.8x\n7:0x%.8x\n",data.struct_data.x[0],data.struc
t_data.x[1]);
printf("8:0x%.8x\n",(int)(*((short*)data.struct_data.y)));

struct my_struct my_struct_array[2][2];
memset(my_struct_array,0,sizeof(my_struct_array));

printf("9:0x%.8x\n",*((int*)my_struct_array));

int *int_ptr=(int*)my_struct_array;
for(int i=1;i<=12;i++,int_ptr++)
      *int_ptr=i;

printf("10:0x%.8x\n11:0x%.8x\n",my_struct_array[0][0].x[1]
      , (int)my_struct_array[1][0].a);
```

What is the output? Fill in the blanks (blank 0 is done for you to show the format of %.8x)

You can run it on ***Linux/X86***

0: 0x00a43503

1: (1) 16

2: (2) 16

3: (3) 11

4: (4) 16

5: (5) 0x00000000

6: (6) 0x07060504

7: (7) 0x0b0a0908

8: (8) 0x00000d0c

9: (9) 0x00000000

10: (10)0x00000003

11: (11)0x00000009

Problem 5: (21 points)

Given the following assembly code segment, it is to run on a platform in ***X86 32***:

```
1 080483a0 <foo>:                    23 lea
(%ecx,%ecx,2),%eax
2 push %ebp                          24 movsbl %bl,%ebx
3 mov %esp,%ebp                      25 add %ebx,%eax
4 push %ebx                          26 shl $0x2,%eax
5 sub $0x14,%esp                     27 jmp .L6
6 mov 0x8(%ebp),%edx                 28 .L3
7 mov 0x10(%ebp),%ecx                29 add $0x1,%ecx
8 movzbl 0xc(%ebp),%ebx                 30
mov %ecx,0x8(%esp)
9 cmp $0x6,%edx                      31 movsbl %bl,%ebx
10 je .L4                            32
mov %ebx,0x4(%esp)
11 cmp $0x6,%edx                     33 movl $0x4,(%esp)
12 jg .L1                            34 call 80483a0
<foo>
13 cmp $0x5,%edx                     35 jmp .L6
14 jne .L5                           36 .L4
15 jmp .L3                        37 subl %ebx,%ecx
16 .L1                            38 .L5
17 cmp $0x7,%edx                     39 mov %ecx,%eax
18 je .L2                            40 .L6
```

a. According to previous assembly code, please complete the following boo function in C. (2'*2+1'*4)

```
long foo ( TypeA a, TypeB b, long c ) {
  long result;
  switch (a) {
    case 7:
          result = __[1](3*c + b) << 2__;
          break;
    case 5:
          result = __[2]foo(a–1, b, c+1)/foo(4,b,c+1)__;
          break;
    case __[3]6__:
    case __[4]9__:
          __[5]c -= b;__;
    default:
          result = __[6]c__;
  }
  return result;
}
```

b. Please determine the most reasonable type of parameters a and b (instead of TypeA and TypeB) in function foo. (2’)

Type A : int / long

Type B : char

c. Suppose the function foo is recursively called at line 34, consider the value of %ebp at line 6 will become ___smaller___(larger/smaller/unchanged) ? (2’)

d. Try to transform the above assembly code into an indirect jump version. (2’+3’+4’)

(a) The minimal entries size of jump table is **__[1]**_5__ .

(b) Modify the piece of segment of assemble code in bold based on the following frame code:

Assembly Code:

```
leal    ___[2] -5(%edx)___, %edx
cmpl    __[3] $4__, %edx
ja      ___[4] .L5___
jmp         *.L8(,%edx, 4)
```

Jump Table:

```
.section .rodata
.align 4
.L8:
//Please fill entries here
```

Problem 6: (18 points)

Consider the following C program.

```
int rec (int n){
    if (n == 0)
        return 1;
    else if (n == 1)
        return 1;
    else
        return rec(n - __[1]_1_) + (rec(n - __[2]_2_) << 2);
}
int main (void){
    printf("%d\n",rec(4));
    return 0;
}
```

Here is corresponding assembly code:

```
0x08048354 <rec>:
0x08048354      pushl __[3]%ebp__
0x08048355      movl %esp, %ebp
0x08048357      pushl __[4]%ebx__
0x08048358      subl $0x8, %esp
                cmpl __[5]$0__, 0x8(%ebp)
                jne .L2
                movl $0x1, -8(%ebp)
                jmp .L4
.L2:                cmpl __[6]$1__, 8(%ebp)
                jne .L5
                movl $0x1, -8(%ebp)
                jmp .L4
.L5:                movl 0x8(%ebp), %eax
                decl %eax
0x0804837d      movl %eax, (%esp)
0x08048380      call rec
0x08048385      movl %eax, %ebx
0x08048387      movl 0x8(%ebp), %eax
0x0804838a      subl $0x2, %eax
0x0804838d      movl %eax, (%esp)
0x08048390      call rec
0x08048395      sall $0x2, %eax
                addl %eax, %ebx
                movl %ebx, -8(%ebp)
.L4:                movl -8(%ebp), __[7]%eax__
                addl $0x8, %esp
                popl %ebx
0x080483a4popl %ebp
0x080483a5ret
0x080483a6      <main>:
...
0x080483be      call rec
0x080483c3      mov %eax, 0x4(%esp)
... ...
```

a. Fill in the blanks of c program and assembly code. (1'*7)

b. Suppose the value of register %esp is 0xbfbefa7c before the program ***first*** time enter the function "rec" with parameter 0x04. Consider the stack status before we return from the ***first*** call to function "rec" with parameter 0x01. Fill in the following blanks with the corresponding address and/or value. (***Note:*** *"---" represents some space or nothing on the stack*) (0.5'*14)

ADDRESS	VALUE
0xbfbefa80	0x04
0xbfbefa7c	**[1] 0x080483c3**
...	...
[2] 0xbfbefa78	0xbfbefa98 (old %ebp)
...	...
[3] 0xbfbefa6c	0x03
...	...
[4] 0xbfbefa68	0x08048385
...	...
[5] 0xbfbefa64	**[6]** (old %ebp)0xbfbefa78
...	...
[7] 0xbfbefa58	0x02
...	...
[8] 0xbfbefa54	0x08048385
...	...
[9] 0xbfbefa50	**[10]** (old %ebp)0xbfbefa64
...	...
[11] 0xbfbefa44	0x01
...	...
[12] 0xbfbefa40	0x08048385
...	...
[13] 0xbfbefa3c	**[14]** (old %ebp)0xbfbefa50
...	...

c. Consider the stack status before we return from the ***first*** call to function "rec" with parameter 0x01, how much space is exactly allocated but not used in the stack? These empty entries are allocated by which instruction in the assembly code? (4')

16 bytes

subl $8, %esp

"subl $8, %esp" allocates 32bytes, but 12bytes are used to transfer parameter and 4bytes are used by "movl %ebx, -8(%ebp)".