Student Name: [redacted]
Student ID: [redacted]

Computer Systems I

Assignment 1

Problem 1

Complete the following 8-bit two's complement calculations, and represent the results by 8-bit two's complement:

1. 0101 0011 + 0011 0111
2. 0010 1100 + 1010 0010
3. 1111 1100 + 1011 0101
4. 0111 1010 + 0011 1011
5. 0110 0101 - 0010 1010
6. 1100 0010 - 1111 1100
7. 0110 1111 - 1111 0101
8. 0111 0010 - 1000 0010

Answer:

1\.
```
  01010011
+ 00110111
----------
  10001010
```

2\.
```
  00101100
+ 10100010
----------
  11001110
```

3\.
```
  11111100
+ 10110101
----------
  10110001
```

4\.
```
  01111010
+ 00111011
----------
  10110101
```

5\. 补: 11010110
```
  0110 0101
+ 1101 0110
-----------
  0011 1011
```

6\. 补: 00000100
```
  1100 0010
+ 00000100
----------
  11000110
```

7\. 补: 00001011
```
  0110 1111
+ 00001011
----------
  01111010
```

8\. 补: 01111110
```
  0111 0010
+ 0111 1110
-----------
  11110000
```

Problem 2

We are running programs where values of type int are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type unsigned are also 32 bits. We generate arbitrary values x and y, and convert them to unsigned values as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to True) for all values of x and y, or (2) give values of x and y for which it is false (evaluates to False):

1. `(x < 0) == (-x > 0)`
2. `~x + ~y < ~(x + y)`
3. `((x >> 6) << 6) <= x`
4. `(ux + uy) == -(unsigned)(-y - x)`

Answer:

1. $x<0$ 时　　$x \geq 0$ 时

$x<0 \equiv$ True　　$x<0 \equiv$ False

$-x>0 \equiv$ True　　$-x>0 \equiv$ False

于是 $\forall x$, 1. $\equiv$ True

2. $x = -INT\_MAX-1, y=-1$

$\sim x = INT\_MAX, \sim y = 0$

$\sim x + \sim y = INT\_MAX$

$\sim(x+y) = -INT\_MAX-1$

False.

3. $(x>>6)<<6$ 后先去最低6位

$x>0$　$((x>>6)<<6) < x$

$x<0$　$((x>>6)<<6) < x$　最高位为符号位

$x=0$　$((x>>6)<<6) == x$

Problem 3

Consider the following two 9-bit floating-point representations based on the IEEE floating point format.

- Format A: There is 1 sign bit. There are $k = 5$ exponent bits. The exponent bias is 15. There are $n = 3$ fraction bits.
- Format B: There is 1 sign bit. There are $k = 3$ exponent bits. The exponent bias is 3. There are $n = 5$ fraction bits.

In the following table, you are given some bit patterns in format A, and your task is to convert them to the value in format B. In addition, give the values of numbers given by the format A and format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64). If the value represented by bits in format A cannot be expressed by format B, please fill **NULL** in the **Value** column and explain the reason briefly in **Bits** column.

	Format A		Format B	
No.	Bits	Value	Bits	Value
1	0 10010 011	11	0110 01100	11
2	1 00011 010	-5/16384	NULL	e=0 f=0 +0
3	0 00011 010	5/16384	NULL	e=0 f=0 -0
4	1 11000 000	-512	NULL	+∞
5	0 10011 100	24	0 111 10000	24

Answer:

$(-1)^0 \times (1.011)_2 \times 2^{18-15} = (1011)_2 = 11$

$(-1)^1 \times (1.010)_2 \times 2^{3-15} = -\frac{5}{16384}$

$(-1)^0 \times (1.010)_2 \times 2^{3-15} = (1.010)_2 \times 2^{-12} = \frac{5}{16384}$

$(-1)^1 \times (1.000)_2 \times 2^{24-15} = -512$

$(-1)^0 \times (1.100)_2 \times 2^{19-15} = 16+8 = 24$

B. 最小值为 2^{-3}

最大值为 $(2-2^{-5}) \times 2^4 < 2^5 = 32$

次正规数为 $2^{-5} \times 2^{0-3} = 2^{-8}$

3

Problem 4

Fill in code for the following C functions, following the bit-level integer coding rules (Appendix 1). Function **srl** performs a logical right shift using an arithmetic right shift (given by value **xsra**), followed by other operations not including right shifts or division. Function **sra** performs an arithmetic right shift using a logical right shift (given by value **xsrl**), followed by other operations not including right shifts or division. You may use the computation **8*sizeof(int)** to determine **w**, the number of bits in data type int. The shift amount **k** ranges from 0 to w-1.

Warning: Bit-level coding is a must!

```c
unsigned srl(unsigned x, int k) {
    /* Perform shift logically */
    unsigned xsra = (int) x >> k;
    /*
     * Tip: Your code should be added here.
     */
}

int sra(int x, int k) {
    /* Perform shift arithmetically */
    unsigned xsrl = (unsigned) x >> k;
    /*
     * Tip: Your code should be added here.
     */
}
```

Answer:

```c
unsigned srl (unsigned x, int k) {
    /*...*/
    return x>>k;
}

int sra (int x, int k) {
    /*...*/
    unsigned ux = x;
    if (x<0)
        ux |= ~UINT_MAX_;
    unsigned shifted = ux>>k;
    if (x<0)
        shifted |= ~_UINT_MAX_>>k
    return (int)shifted;
}
```

4

Problem 5

Following the bit-level floating-point coding rules (Appendix 2), implement the function with the following prototype:

```c
/* Compute -f. If f is NaN, then return f. */
float float_negate(float f);
```

For floating-point number f , this function computes -f. If f is NaN, your function should simply return f. **Warning: Bit-level coding is a must!** Testing code is shown here:

```c
// Copyright 2023 Sycuricon Group
// Author: Phantom (phantom@zju.edu.cn)

#include <stdio.h>
typedef unsigned float_bits;
float float_negate(float f);

int main() {
    printf("%f\n",float_negate(32.0));
    printf("%f\n",float_negate(-12.8));
    printf("%f\n",float_negate(0.0));
    return 0;
}
```

Answer:

```c
float float_negate(float f){
    float_bits *p=(float_bits*)&f;
    if((*p>>23&0xFF)==0xFF&&(*p&0x7FFFFF)!=0x0){
        return F;
    }
    *p ^=1<<31;
    return f;
}
```

Problem 6

Explain why the following code snippets are wrong:

1. Code Snippet 1

```
unsigned i;
int cnt = 10, sum = 0;
for (i = cnt - 1; i >= 0; i --) {
    sum += i
}
print(sum);
```

2. Code Snippet 2

```
#define Delta sizeof(int)
int i;
int cnt = 10, sum = 0;
for (i = cnt; i - Delta >= 0; i -= Delta) {
    sum += i;
}
print(sum);
```

Answer:

Snippet 1: For unsigned i, ~~...~~ i>=0 ≡ True, it'll loop forever.

Snippet 2: sizeof() ~~returns~~ computes a size_t type (unsigned), When i-Delta is performed, it occurs an implicit conversion fron int to unsigned number, thus i-Delta>=0 ≡ True it'll loop forever.