Student Name:
Student ID:

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:

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:

```
1 /* Create some arbitrary values */
2 int x = random();
3 int y = random();
4 /* Convert to unsigned */
5 unsigned ux = (unsigned) x;
6 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:

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			
2	1 00011 010			
3	0 00011 010			
4	1 11000 000			
5	0 10011 100			

Answer:

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!

```
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:

Problem 5

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

```
/* 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:

```
// 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:

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:

Appendix 1: Bit-Level Integer Coding Rules

In Problem 4, we will artificially restrict what programming constructs you can use to help you gain a better understanding of the bit-level, logic, and arithmetic operations of C. In answering these problems, your code must follow these rules:

Assumptions

1. Integers are represented in two's-complement form.
2. Right shifts of signed data are performed arithmetically.
3. Data type int is w bits long. For some of the problems, you will be given a specific value for `w`, but otherwise your code should work as long as w is a multiple of 8. You can use the expression `sizeof(int)<<3` to compute `w`.

Forbidden

1. Conditionals (`if` or `?:`), loops, switch statements, function calls, and macro invocations.
2. Division, modulus, and multiplication.
3. Relative comparison operators (`<`, `>`, `<=`, and `>=`).

Allowed operations

1. All bit-level and logic operations.
2. Left and right shifts, but only with shift amounts between `0` and `w-1`. Addition and subtraction.
3. Equality (`==`) and inequality (`!=`) tests.
4. Integer constants `INT_MIN` and `INT_MAX`. (TIP: `<limits.h>` should be included if you want to use them.)
5. Casting between data types int and unsigned, either explicitly or implicitly.

Even with these rules, you should try to make your code readable by choosing descriptive variable names and using comments to describe the logic behind your solutions.
As an example, the following code extracts the most significant byte from integer argument `x`:

```
/* Get most significant byte from x */
int get_msb(int x) {
    /* Shift by w-8 */
    int shift_val = (sizeof(int)-1)<<3; /* Arithmetic shift */
    int xright = x >> shift_val;
    /* Zero all but LSB */
    return xright & 0xFF;
}
```

Appendix 2: Bit-Level Floating-Point Coding Rules

In Problem 5, you will write code to implement floating-point functions, operating directly on bit-level representations of floating-point numbers. Your code should exactly replicate the conventions for IEEE floating-point operations, including using round-to-even mode when rounding is required. To this end, we define data type `float_bits` to be equivalent to unsigned:

```
/* Access bit-level representation floating-point number */
typedef unsigned float_bits;
```

Rather than using data type float in your code, you will use `float_bits`. You may use both int and unsigned data types, including unsigned and integer constants and operations. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating-point data types, operations, or constants. Instead, your code should perform the bit manipulations that implement the specified floating-point operations.

The following function illustrates the use of these coding rules. For argument `f` , it returns ±0 if `f` is denormalized (preserving the sign of `f`), and returns `f` otherwise.

```
/* If f is denorm, return 0. Otherwise, return f */
float_bits float_denorm_zero(float_bits f) {
  /* Decompose bit representation into parts */
  unsigned sign = f>>31;
  unsigned exp = f>>23 & 0xFF;
  unsigned frac = f & 0x7FFFFF;
  if (exp == 0) {
    /* Denormalized. Set fraction to 0 */
    frac = 0;
    }
  /* Reassemble bits */
  return (sign << 31) | (exp << 23) | frac;
}
```