Course: **Computer Architecture** (2020 Fall)

(Always write code in valid C99/C11 standard.)

(You are allowed to read lecture materials. But you shall not disclose your answer during the quiz.)

1. What is the hexadecimal representation for decimal **`-51`** encoded as an 8-bit two's complement number? [5 points]

51 = 32+16+2+1 = 0b0011_0011, (using 8-bit binary)

-51 = 0b1100_1101 = 0xCD

2. Can the value of the sum of two 2's complement numbers **0xB3 + 0x47** be represented using an 8-bit 2's complement representation? If so, what is the sum in hex? If not, write NO. [5 points]

Yes: negative + positive is always within range. 0xB3 + 0x47 = 0b1011_0011 + 0b0100_0111 = 0b1111_1010 = 0xFA

3. Suppose we have defined a linked list `struct` as follows. Assume `*lst` points to the first element of the list, or is NULL if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node *rest;
}
```

(a) [5 points] Implement `add_front`, which adds one new value to the front of the linked list. Function prototype: `void add_front(struct ll_node **lst, int value);`

```
void add_front(struct ll_node **lst, int value) {
        struct ll_node* item = malloc(sizeof(struct ll_node));
        item->first = value;
        item->rest = *lst;
        *lst = item;
}
```

(b) [5 points] Implement `free_ll`, which frees all the memory consumed by the linked list. You must prevent users from writing to unusable memory after free. Function prototype: `void free_ll(struct ll_node **lst);`

```
void free_ll(struct ll_node **lst) {
        if (*lst) {
                free_ll(&((*lst)->rest));
                free(*lst);
        }
        *lst = NULL;
}
```

4. The IEEE 754 standard defines a binary representation for floating point values using three fields.

- The sign determines the sign of the number (0 for positive, 1 for negative).
- The exponent is in biased notation. For instance, the bias is `-127` which comes from $-(2^{8-1} - 1)$ for single-precision floating point numbers.
- The significand or mantissa is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

$\texttt{Value} = (-1)^{\texttt{Sign}} * 2^{\texttt{Exp+Bias}} * \texttt{1.significand}_2$

For denormalized floats:

$\texttt{Value} = (-1)^{\texttt{Sign}} * 2^{\texttt{Exp+Bias+1}} * \texttt{0.significand}_2$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

Note that in the above table, our exponent has values from `0` to `255`. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

(a) [5 points] What is the largest finite positive value that can be stored using a single precision float?

$\texttt{0X7F7FFFFF} = (1 + (1 - 2^{-23})) * 2^{127}$

Since this is a normalized number, it has a 1 to the left of the decimal point.

(b) [5 points] What is the smallest positive value that can be stored using a single precision float?

$\texttt{0x00000001} = 2^{-23} * 2^{-126}$

(c) [5 points] What is the smallest positive normalized value that can be stored using a single precision float?

$\texttt{0x00800000} = 2^{-126}$

5. Not every number can be represented perfectly using floating point. For example, 13 can only be approximated and thus must be rounded in any attempt to represent it. For this question, we will only look at positive numbers.

(a) [5 points] What is the next smallest number larger than 4 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.

$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$

(b) [10 points] If we are given a normalized number that is not the largest representable normalized number with exponent value `x` and with significand value `y`, what is the stepsize at that value? (There are 23 significand bits.)

This is the same approach just increment the signifcand by the 1.

curr_number = $2^{x-127} + 2^{x-127} * y$

next_number = $2^{x-127} + 2^{x-127} * y + 2^{x-127} * 2^{-23}$

stepsize = next_number − curr_number = 2^{x-150}

6. [15 points] Implement C function which gets a number and returns if it is positive (>0).

if **<=0** then return false, if **>0** then return true.

example: 0 => false, -12 => false, 29 => true

However, you CAN NOT use any flow control (e.g. `if, else, for, switch, case, goto, while, "?"`) nor logical operators (e.g. `&&, ||, >, <, ==, <=, >=`). Instead, you SHALL use operator `<<, >>, &, |, +, ^, ~, !` to implement such routine. Assume `sizeof(int) = 4`

Function prototype: `int is_positive(int x)`

```
int is_positive(int x){
        int mask = 1;
        /* place x's MSB in the rightest bit
        if x negative- temp is 11111111, if positive/0- temp is 00000000 */
        int temp = x >> 31;
        /* keep just the rightest bit by & to mask(00000001)
        if x negative- temp is 00000001, if positive/0- temp is 00000000*/
        temp = temp & mask;
        // if x is 0- now it is 00000001, else- its 0;
        x = (!x);
        /* if x is negative or '0'- make x to 1
        if x is positive - make x to 0 */
        x = (x | temp);
        // if x positive - make it 1, if negative/0- make it 0
        x = (!x);
        // return 1 if >0, else- return 0
        return x;
}
```

7. Assume a new execution mode called "turbo mode" provides a 2.5x speedup to the sections of programs where it applies. What percentage of a program (measured by original execution time) must run in "turbo mode" for an overall speedup of 10%? [5 points]

Let f be the required fraction.

1 / ((1 – f) + f/2.5) = 1.1

f = 0.152 or 15.2%

8. Following the bit-level floating-point coding rules, implement the function with the

following prototype:

```
/*
 * Compute (int) f
 * If conversion causes overflow or f is NaN, return 0x80000000
 */
typedef unsigned float_bits;
int float_to_int(float_bits f);
```

For floating-point number f, this function computes `(int) f`. Your function should round toward zero. If f cannot be represented as an integer, then the function should return `0x80000000`. [20 points]

```
float u2f(unsigned u) { return *(float *) &u; }
int float_to_int(float_bits f) {
        unsigned sign = f >> 31;
        unsigned exp = f >> 23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        unsigned bias = 0x7F;
        int result;
        unsigned E;
        unsigned M;
        if (exp < bias) {
                /* the float number is less than 1 */
                result = 0;
        } else if (exp >= 31 + bias) {
                /* overflow */
                result = 0x80000000;
        } else {
                /* normal */
                E = exp - bias;
                M = frac | 0x800000;
                if (E > 23) {
                        result = M << (E - 23);
                } else {
                        /* round to zero */
                        result = M >> (23 - E);
                }
        }
        return sign ? -result : result;
}
```

9. Write code for a function with the following prototype:

```
/* Divide by power of two. Assume 0 <= k < w-1 */
int div_power2(int x, int k);
```

The function should compute $x/2^k$ with correct rounding. [10 points]

```
#include <limits.h>

int div_power2(int x, int k) {
        /*
        * if x >= 0, then the result is x >> k;
        * if x < 0, we should add x with bias to make sure the correct rounding.
        * (x + (x + (1 << k) - 1)) >> k
        */
        int neg_flag = x & INT_MIN;
        neg_flag && (x = x + (1 << k) - 1);
        return x >> k;
}
```

10. [10 points] Write code for the function with the following prototype:

```
/*
 * Return 1 when x can be represented as an n-bit, 2's complement
 * number; 0 otherwise
 * Assume 1 <= n <= w
 */
int fits_bits(int x, int n);
```

```
int fits_bits(int x, int n) {
/*
* 1 <= n <= w
*
* assume w = 8, n = 3
* if x > 0
* 0b00000010 is ok, 0b00001010 is not, and 0b00000110 is not yet (thanks itardc@163.com)
* if x < 0
* 0b11111100 is ok, 0b10111100 is not, and 0b11111000 is not yet
*
* the point is
* x << (w-n) >> (w-n) must be equal to x itself.
*
*/
        int w = sizeof(int) << 3;
        int offset = w - n;
```

```
        return (x << offset >> offset) == x;
    }
```