15-122: Principles of Imperative Computation — Spring 2018

Lab 3: A bit more about Bits — 23 January

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

Two's complement

Because C0's **int** type only represents integers in the range $[-2^{31}, 2^{31})$, addition and multiplication are defined in terms of modular arithmetic. As a result, adding two positive numbers may give you a negative number!

Checkpoint 0

Write a function that returns 1 if the sign bit is 1, and 0 otherwise. That is, write a function that returns the sign bit shifted to be the least significant bit. Your solution can use any of the bitwise operators, but will not need all of them.

```
1 int getSignBit(int x)
2 //@ensures \result == 0 || \result == 1;
3 {
4     return ____________________________________________________________;
5 }
```

Frames

In homework, we used a 32-bit C0 integer to store pixels as *four 8-bit unsigned quantities*, with the alpha component stored in the eight high-order bits and the blue component stored in the eight low-order bits:

Alpha	*Red*	*Green*	*Blue*
$a_7a_6a_5a_4a_3a_2a_1a_0$	$r_7r_6r_5r_4r_3r_2r_1r_0$	$g_7g_6g_5g_4g_3g_2g_1g_0$	$b_7b_6b_5b_4b_3b_2b_1b_0$
(unsigned)	(unsigned)	(unsigned)	(unsigned)

In networking protocols like Ethernet, data is transmitted in the form of a sequence of bits called *frames*. In this question we will use a 32-bit C0 integer to store data belonging to one frame. Frames have four parts:

- The *destination address*, i.e., the device number where the data is going (6 bit unsigned integer)
- The *source address*, i.e., the device number where the data is coming from (6 bit unsigned integer)
- The *data* or *payload* (18 bit unsigned integer)
- Two bits that are used to ensure that everything is received correctly (the *cyclic redundancy check* or CRC).

The interface to the `frame` type is much like the interface to the `pixel` type:

```
typedef int frame;

frame make_frame(int source, int dest, int data)
/*@requires 0 <= source && source < ______; @*/
/*@requires 0 <= dest   && dest   < ______; @*/
/*@requires 0 <= data   && data   < 262144; @*/ ;

int get_source(frame F)  /*@ensures 0 <= \result && \result < ______; @*/ ;
int get_dest(frame F)    /*@ensures 0 <= \result && \result < ______; @*/ ;
int get_data(frame F)    /*@ensures 0 <= \result && \result < 262144; @*/ ;
```

(3.a) Given this format of frames, what is the maximum number (in decimal) of addressable devices in the network?

1pt

Checkpoint 1

(4.a) For each frame, the CRC value is calculated by counting the total number of "1" bits in the source, destination, and payload fields combined, and then finding the remainder when this number is divided by 4. What would the CRC be if the source was decimal 15, the destination was decimal 40, and the payload was the decimal number 1024?

One way of packing a frame into a 32-bit integer is as follows:

Destination	*Source*	*Data*	*CRC*
$d_5d_4d_3d_2d_1d_0$	$s_5s_4s_3s_2s_1s_0$	$i_{11}i_{10}i_Fi_Ei_Di_Ci_Bi_Ai_9i_8i_7i_6i_5i_4i_3i_2i_1i_0$	c_1c_0
(unsigned)	(unsigned)	(unsigned)	(unsigned)

(4.b) In this implementation, if frame `f` is `0xFADEDBEE`, what is the value of `get_source(f)`?

2pt

Checkpoint 2

(5.a) To calculate the CRC, we're going to write a helper function that counts the number of bits set to "1" in an arbitrary *non-negative* integer. Here's the skeleton of one possible implementation. Fill in the blanks so that it correctly returns the number of bits set to "1" in the non-negative number `x`. Some notes:

- This function only has to work with non-negative inputs.
- The second loop invariant should allow you to prove that the postcondition holds.
- The third loop invariant, if completed, should allow you to prove that the assertion after the loop, `//@assert x == 0`, always evaluates to `true`.

```
int count_ones(int x)
//@requires x >= 0;
//@ensures 0 <= \result && \result <= 31;
{
    int ones = 0;
    for (int i = 0; i < 31; i++)
    //@loop_invariant 0 <= i && i <= 31;

    //@loop_invariant ____________________________________________________;

    //@loop_invariant x & (_____________________________________) == 0;
    {
        if (_____________ % _____________ ==     _____________) {

            ____________________________________________________________;
        }
        x = x >> 1;
    }
    //@assert x == 0;
    return ones;
}
```

3pt