Calvin Liu

804182525

Homework 1

Write code to implement the following function:

/\*

\* Generate mask indicating leftmost 1 in x. Assume w=32.

\* For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.

\* If x = 0, then return 0.

\*/

int leftmost\_one(unsigned x);

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type int has w = 32 bits.

Your code should contain a total of at most 15 arithmetic, bit-wise, and logical operations.

**Hint:** First transform x into a bit vector of the form [0 . . . 011 . . . 1].

int leftmost\_one(unsigned x)

{

x |= (x >> 1);

x |= (x >> 2);

x |= (x >> 4);

x |= (x >> 8);

x |= (x >> 16);

return x - (x >> 1);

}

You just started working for a company that is implementing a set of procedures to operate on a data structure where 4 signed bytes are packed into a 32-bit unsigned. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two’s-complement arithmetic and arithmetic right shifts with the following prototype:

/\* Declaration of data type where 4 bytes are packed

into an unsigned \*/

typedef unsigned packed\_t;

/\* Extract byte from word. Return as signed integer \*/

int xbyte(packed\_t word, int bytenum);

That is, the function will extract the designated byte and sign extend it to be a 32-bit int.

Your predecessor (who was fired for incompetence) wrote the following code:

/\* Failed attempt at xbyte \*/

int xbyte(packed\_t word, int bytenum)

{

return (word >> (bytenum << 3)) & 0xFF;

}

A. What is wrong with this code?

It just returns the positive 8 bit in bytenum position because you are always &ing it with 000…0 1111 1111.

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

int xbyte(packed\_t word, int bytenum)

{

return (24 – (bytenum << 3)) >> 24;

}

You are given the task of writing a function that will copy an integer val into a buffer buf, but it should do so only if enough space is available in the buffer.

Here is the code you write:

/\* Copy integer into buffer if space is available \*/

/\* WARNING: The following code is buggy \*/

void copy\_int(int val, void \*buf, int maxbytes) {

if (maxbytes-sizeof(val) >= 0)

memcpy(buf, (void \*) &val, sizeof(val));

}

This code makes use of the library function memcpy. Although its use is a bit artificial here, where we simply want to copy an int, it illustrates an approach commonly used to copy larger data structures.

You carefully test the code and discover that it *always* copies the value to the buffer, even when maxbytes is too small.

A. Explain why the conditional test in the code always succeeds. **Hint:** The sizeof operator returns a value of type size\_t.

The sizeof(val) will produce an a type of size\_t so that since it is unsigned, any math done with unsigned ints will produce an unsigned int.

B. Show how you can rewrite the conditional test to make it work properly.

To fix it you could put (int) in front of the sizeof function so it will look like

void copy\_int(int val, void \*buf, int maxbytes) {

if (maxbytes-(int)sizeof(val) >= 0)

memcpy(buf, (void \*) &val, sizeof(val));

}

We are running programs on a machine 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, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

1. (x<y) == (-x>-y) – Will not always yield 1 because if x is Tmin and y is 1 it doesn’t work
2. ((x+y)<<4) + y-x == 17\*y+15\*x – Will always yield 1 because you distribute 16 to (x+y) with the shift and then add y and subtract x. You get the same answer.
3. ~x+~y+1==~(x+y) – Will always yield 1 because when you complement a number you have to add 1 to obtain the negation or else it will be off by 1.
4. (ux-uy) == -(unsigned)(y-x) – Will always yield 1 because the left side is equal to ux + ~uy +1. The right side evaluates to ~uy + 1 + ux + 1 + ~u1 + 1. ~u1 + 1 + 1 + 1 is equivalent to 1.
5. ((x>>2)<<2)<=x Always yield 1 because you are always rounding up when it is a negative for x>>2 and then <<2 will make it even more negative. If it is positive then it will round down, <<2 will make it less.