**2.64** ◆

Write code to implement the following function:

|  |
| --- |
| /\* Return 1 when any even bit of x equals 1; 0 otherwise.  Assume w=32. \*/  int any\_even\_one(unsigned x)  {  // your code  unsigned max = 0xffffffff;  unsigned ans = x ^ max;  return !ans;  } |
|  |

Your function should follow the bit-level integer coding rules, except

that you may assume that data type int has *w* = 32 bits.

**2.73** ◆◆

Write code for a function with the following prototype:

|  |
| --- |
| /\* Addition that saturates to TMin or TMax \*/  int saturating\_add(int x, int y)  {  int ans = x + y;  int num\_x = x & 0x7fffffff; //maxint = 0x7fffffff  int num\_y = y & 0x7fffffff; //maxint = 0x7fffffff  int sign\_x = x >> 31; //31 = w - 1  int sign\_y = y >> 31; //31 = w - 1  int sign\_ans = ans >> 31; //31 = w - 1    int whether\_is\_different\_sign = (sign\_x ^ sign\_y);  //if this == 1, cannot satuate  int whether\_up\_satuate = (!sign\_x) && (sign\_ans) && (!whether\_is\_different\_sign);  //if this == 1, up satuate  int whether\_down\_satuate = (sign\_x) && (!sign\_ans) && (!whether\_is\_different\_sign);  //if this == 1, down satuate  int whether\_change = whether\_up\_satuate | whether\_down\_satuate;  int step = (whether\_change << 31) - whether\_change; //31 = w-1  //not satuate: step all 0  //up satuate: step maxint  //down satuate: step maxint  ans -= (whether\_up\_satuate << 31); //31 = w - 1  ans = ans | step;  ans += whether\_down\_satuate;  return ans;  } |
|  |

Instead of overflowing the way normal two’s-complement addition does, saturating

addition returns *TMax* when there would be positive overflow, and *TMin*

when there would be negative overflow. Saturating arithmetic is commonly used

in programs that perform digital signal processing.

Your function should follow the bit-level integer coding rules.

**2.81** ◆

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)**

**x=-2147483648 y=-114514 x>y = 0**

**-x=-2147483648 -y = -114514 -x<-y = 1**

**Can Yield 0**

1. **((x+y)<<5) + x-y == 31\*y+33\*x**

**if x and y are unsigned, the formula will always be true, since all steps equals to (a calculate b) % (2^32), and % is commutative with all other calculations. If unsigned(x) == unsigned(y), x must == y**

**so it always yields 1**

1. **~x+~y == ~(x+y)**

**x= 0, y = 0**

**~x = -1**

**~y=-1**

**~(x+y)=-1**

**Can Yield 0**

1. **(int)(ux-uy) == - (y-x)**

**The binary code of ux-uy = the binary code of x-y**

**So the formula becomes**

**x+(-y)==-(y-x)**

**since + forms an abelian group, it always yields 1**

1. **((x >> 1) << 1) <= x**

**x=-2147483648**

**x>>1=-2147483648**

**(x>>1)<<1=0**

**Can Yield 0**

(PAGE 81)

**Bit-level integer coding rules**

In several of the following problems, 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

Integers are represented in two’s-complement form.

Right shifts of signed data are performed arithmetically.

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

Conditionals (if or ?:), loops, switch statements, function calls, and macro

invocations.

Division, modulus, and multiplication.

Relative comparison operators (<, >, <=, and >=).

Casting, either explicit or implicit.

. Allowed operations

All bit-level and logic operations.

Left and right shifts, but only with shift amounts between 0 and *w* − 1.

Addition and subtraction.

Equality (==) and inequality (!=) tests. (Some of the problems do not allow

these.)

Integer constants INT\_MIN and INT\_MAX.