```
/* 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;
}
```

2.61 ◆◆

Write C expressions that evaluate to 1 when the following conditions are true and to 0 when they are false. Assume `x` is of type `int`.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the most significant byte of `x` equals 1.

D. Any bit in the least significant byte of `x` equals 0.

Your code should follow the bit-level integer coding rules (page 154), with the additional restriction that you may not use equality (`==`) or inequality (`!=`) tests.

2.62 ◆◆◆

Write a function `int_shifts_are_logical()` that yields 1 when run on a machine that uses logical right shifts for data type `int` and yields 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines.

2.63 ◆◆◆

Fill in code for the following C functions. 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. 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. You may use the computation `8*sizeof(int)` to determine w, the number of bits in data type `int`. The shift amount `k` can range from 0 to $w - 1$.

```
int sra(int x, int k) {
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;
    .
    .
    .
}
```

Hint: Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

2.76 ◆◆
Suppose we are given the task of generating code to multiply integer variable `x` by various different constant factors K. To be efficient, we want to use only the operations `+`, `-`, and `<<`. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 5$: _____
B. $K = 9$: _____
C. $K = 30$: _____
D. $K = -56$: _____

2.77 ◆◆
Write code for a function with the following prototype:

```
/* Divide by power of two.  Assume 0 <= k < w-1 */
int divide_power2(int x, int k);
```

The function should compute $x/2^k$ with correct rounding, and it should follow the bit-level integer coding rules (page 154).

2.78 ◆◆
Write code for a function `mul5div8` that, for integer argument `x`, computes `5*x/8`, but following the bit-level integer coding rules (page 154). Your code should replicate the fact that the computation `5*x` can cause overflow.

2.79 ◆◆◆
Write code for a function `fiveeighths` that, for integer argument `x`, computes the value of $\frac{5}{8}$`x`, rounded toward zero. It should not overflow. Your function should follow the bit-level integer coding rules (page 154).

2.80 ◆◆
Write C expressions to generate the bit patterns that follow, where a^n represents n repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters `m` and `n`, representing the values of m and n, but not a parameter representing w.

A. $1^{w-n}0^n$
B. $0^{w-n-m}1^n0^m$

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.

A. `(x>y) == (-x<-y)`

B. `((x+y)<<5) + x-y == 31*y+33*x`

C. `~x+~y == ~(x+y)`

D. `(int) (ux-uy) == -(y-x)`

E. `((x >> 1) << 1) <= x`

2.82 ◆◆

Consider numbers having a binary representation consisting of an infinite string of the form $0.y\,y\,y\,y\,y\,y\cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101\cdots$ $(y = 01)$, while the representation of $\frac{1}{5}$ is $0.001100110011\cdots$ $(y = 0011)$.

A. Let $Y = B2U_k(y)$, that is, the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. **Hint:** Consider the effect of shifting the binary point k positions to the right.

B. What is the numeric value of the string for the following values of y?

(a) 001

(b) 1001

(c) 000111

2.83 ◆

Fill in the return value for the following procedure, which tests whether its first argument is greater than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is *NaN*. The two flavors of zero, +0 and −0, are considered equal.

```
int float_ge(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
```

```
    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return _____ ;
}
```

2.84 ◆

Given a floating-point format with a k-bit exponent and an n-bit fraction, write formulas for the exponent E, significand M, the fraction f, and the value V for the quantities that follow. In addition, describe the bit representation.

A. The number 5.0

B. The largest odd integer that can be represented exactly

C. The reciprocal of the smallest positive normalized value

2.85 ◆

Intel-compatible processors also support an "extended precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

	Extended precision	
Description	Value	Decimal
Smallest positive denormalized	_____	_____
Smallest positive normalized	_____	_____
Largest normalized	_____	_____

2.86 ◆

Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{2}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = 1$. Our number would therefore have an exponent field of 0x40 (decimal value $63 + 1 = 64$) and a significand field 0xC0 (binary 11000000_2), giving a hex representation 40C0.

You need not fill in entries marked "—".

Description	Hex	M	E	V
-0	_____	_____	_____	—
Smallest value > 1	_____	_____	_____	_____
256	_____	_____	_____	—
Largest denormalized	_____	_____	_____	_____
$-\infty$	_____	—	—	—
Number with hex representation 3AA0	—	_____	_____	_____

2.87 ◆◆

Consider the following two 9-bit floating-point representations based on the IEEE floating-point format.

1. Format A
   - There is one sign bit.
   - There are $k = 5$ exponent bits. The exponent bias is 15.
   - There are $n = 3$ fraction bits.
2. Format B
   - There is one sign bit.
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 4$ fraction bits.

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If rounding is necessary, you should *round toward* $+\infty$. 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 or $17/2^6$).

Format A		Format B	
Bits	Value	Bits	Value
1 01110 001	$\frac{-9}{16}$	1 0110 0010	$\frac{-9}{16}$
0 10110 101	_____	_____	_____
1 00111 110	_____	_____	_____
0 00000 101	_____	_____	_____
1 11011 000	_____	_____	_____
0 11000 100	_____	_____	_____

2.88 ◆

We are running programs on a machine where values of type `int` have a 32-bit two's-complement representation. Values of type `float` use the 32-bit IEEE format, and values of type `double` use the 64-bit IEEE format.

We generate arbitrary integer values x, y, and z, and convert them to values of type double as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

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. Note that you cannot use an IA32 machine running GCC to test your answers, since it would use the 80-bit extended-precision representation for both float and double.

A. `(double)(float) x == dx`

B. `dx + dy == (double) (x+y)`

C. `dx + dy + dz == dz + dy + dx`

D. `dx * dy * dz == dz * dy * dx`

E. `dx / dx == dy / dy`

2.89 ◆

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You decide that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the code that follows to compute the correct result. Assume the function u2f returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)
{

    /* Result exponent and fraction */
    unsigned exp, frac;
    unsigned u;

    if (x < _________) {
        /* Too small.  Return 0.0 */
        exp = _________;
        frac = _________;
    } else if (x < _________) {
```