PROBLEM 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 7.0
B. the largest odd integer that can be represented exactly
C. the reciprocal of the smallest positive normalized value

PROBLEM 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 $2^{63} - 1 = 63$. 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 $x/y$, where $x$ is an integer and $y$ is an integral power of 2. Examples include: 0, 67/64, and 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 $7/8$, we would have $s=0$, $M = 7/4$ and $E = -1$. Our number would therefore have an exponent field of 0x3E (decimal value 63 - 1 = 62) and a significand field 0xC0 (binary 11000000) giving a hex representation of 3EC0. You need not fill in entries marked “---”.

<table>
<thead>
<tr>
<th>Description</th>
<th>Hex</th>
<th>M</th>
<th>E</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>smallest value &gt; 2</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>512</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>largest denormalized</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>negative infinity</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>number with hex representation</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3BB0</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
PROBLEM 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 positive infinity. In addition, give the values of number given by Format A and Format B bit patterns. Give these as whole numbers (i.e. 17) or as fractions (i.e. 17/64 or 17/2^6).

<table>
<thead>
<tr>
<th>Format A</th>
<th>Format B</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS</td>
<td>VALUE</td>
</tr>
<tr>
<td>1 01111 001</td>
<td>-9/8</td>
</tr>
<tr>
<td>0 10110 011</td>
<td>-9/8</td>
</tr>
<tr>
<td>1 00111 010</td>
<td>-</td>
</tr>
<tr>
<td>0 00000 111</td>
<td>1/16</td>
</tr>
<tr>
<td>1 11100 000</td>
<td>1/2</td>
</tr>
<tr>
<td>0 10111 100</td>
<td>9/16</td>
</tr>
</tbody>
</table>

PROBLEM 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:

```c
/* 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 zero. 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. (float) x == (float) dx
B. dx – dy == (double) (x-y)
C. (dx + dy) + dz == dx + (dy + dz)
D. (dx * dy) * dz == dx * (dy * dz)
E. dx / dx == dz / dz
PROBLEM 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 positive infinity. 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 of its unsigned argument.

```c
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 < ________________)  {
        /* denormalized result */
        exp = _______________;  
        frac = _______________; 
    } else if (x < ____________________)  {
        /* normalized result */
        exp = _______________;  
        frac = _______________; 
    }  else  
        /* too big. Return +oo */
        exp = _______________;  
        frac = _______________; 
    }

    /* pack exp and frac into 32 bits */
    u = exp << 23 | frac;
    /* return as float */
    Return u2f(u);
}
PROBLEM 2.90

Around 250 B.C., the Greek mathematician Archimedes proved that $223/71 < \pi < 22/7$. Had he had access to computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of $\pi$ has the hexadecimal representation 0x40490FDB. Of course, all of these are just approximations, since $\pi$ is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $22/7$? HINT: see problem 2.82.

C. At what bit position (relative to the binary point) do these two approximations to $\pi$ diverge?

PROBLEM 2.91

Bit level floating point coding rules

For the following problem, you will write code to implement floating-point functions, operating directly on bit-level representations of floating-point numbers. Your code should exactly replicate the conventions for IEEE floating-point operations, including using round-to-even mode when rounding is required.

Toward this end, we define data type `float_bits` to be equivalent to unsigned:

```c
/* Access bit-level representation floating-point number */
typedef unsigned float_bits;
```

Rather than using data type `float` in your code, you will use `float_bits`. You may use both `int` and unsigned data types, including unsigned and integer constants and operations. You may not use any union, structs or arrays. Most significantly, you may not use any floating-point data types, operations or constants. Instead, your code should perform the bit manipulations that implement the specified floating-point operations.

The following function illustrates the use of these coding rules. For argument `f`, it returns plus/minus 0 if `f` is denormalized (preserving the sign of `f`) and returns `f` otherwise.

```c
/* if f is denorm, return 0. Otherwise, return f */
float_bits float_denorm_zero(float_bits f) {
    /* decompose bit-representation into parts */
    unsigned sign = f >> 31;
    unsigned exp = f>>23 & 0xFF;
    unsigned frac = f  &  0x7FFFFF;
    if (exp == 0) {
        /* denormalized. Set fraction to 0 */
        frac = 0;
    }
    /* reassemble bits */
    return (sign << 31)  |  (exp << 23)  |  frac;
}
```

Implement the function with the following prototype:

```c
/* compute –f. if f is NaN then return f */
float_bits float_negate(float_bits f);
```

For floating-point number `f`, this function computes $–f$. if `f` is NaN, your function should simply return `f`. Test your function by evaluating it for all $2^{32}$ values of argument `f` and comparing the result to what would be obtained using your machine’s floating-point operations.