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

A. The number 7.0 will have \( E = 2, M = 1.112 = \frac{7}{4}, f = 0.112 = \frac{3}{4}, \) and \( V = 7. \) The exponent bits will be 100 \( \cdots \) 01 and the fraction bits will be 1100 \( \cdots \) 0.

B. The largest odd integer that can be represented exactly will have a binary representation consisting of \( n + 1 \) ones. It will have \( E = n, M = 1.11 \cdots 12 = 2 - 2^{-n}, f = 0.11 \cdots 12 = 1 - 2^{-n}, \) and a value \( V = 2^{n+1} - 1. \) The bit representation of the exponent will be the binary representation of \( n + 2^{k-1} - 1. \) The bit representation of the fraction will be 11 \( \cdots \) 11.

C. The reciprocal of the smallest positive normalized value will have value \( V = 2^{2^{k-1}-2}. \) It will have \( E = 2^{k-1} - 2, M = 1, \) and \( f = 0. \) The bit representation of the exponent will be 11 \( \cdots \) 101. The bit representation of the fraction will be 00 \( \cdots \) 00.

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 \( 2^{7-1} - 1 = 63. \) Fill in the table that follows for each of the numbers given, with the following instructions for each column:

<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>Description</th>
<th>Hex</th>
<th>M</th>
<th>E</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>−0</td>
<td>8000</td>
<td>0</td>
<td>−62</td>
<td>−0</td>
</tr>
<tr>
<td>Smallest value &gt; 2</td>
<td>4001</td>
<td>257</td>
<td>1</td>
<td>257/128</td>
</tr>
<tr>
<td>512</td>
<td>4800</td>
<td>1</td>
<td>72</td>
<td>−</td>
</tr>
<tr>
<td>Largest denormalized</td>
<td>00FF</td>
<td>255</td>
<td>−62</td>
<td>255 x 2&lt;sup&gt;−70&lt;/sup&gt;</td>
</tr>
<tr>
<td>−∞</td>
<td>FF00</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Number with hex representation</td>
<td>3BB0</td>
<td>27/16</td>
<td>−4</td>
<td>27/256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BITS</th>
<th>VALUE</th>
<th>BITS</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 01111 001</td>
<td>-9/8</td>
<td>1 01111 0010</td>
<td>-9/8</td>
</tr>
<tr>
<td>0 10110 011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01111 010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00000 111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 11100 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10111 100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
<th>Bits</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 01111 001</td>
<td>-9/8</td>
<td>1 01111 0010</td>
<td>-9/8</td>
<td></td>
</tr>
<tr>
<td>0 10110 011</td>
<td></td>
<td>0 1110 0110</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>1 00111 010</td>
<td>-5/1024</td>
<td>1 0000 0101</td>
<td>-5/1024</td>
<td></td>
</tr>
<tr>
<td>0 00000 111</td>
<td>7/131072</td>
<td>0 0000 0001</td>
<td>1/1024</td>
<td>Smallest positive denorm</td>
</tr>
<tr>
<td>1 11100 000</td>
<td>-8192</td>
<td>1 1110 1111</td>
<td>-248</td>
<td>Smallest number &gt; −∞</td>
</tr>
<tr>
<td>0 10111 100</td>
<td>384</td>
<td>0 1111 0000</td>
<td>+∞</td>
<td>Round to ∞.</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).
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:

/* 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

A. (float) x == (float) dx. Yes. Converting to float could cause rounding, but both x and
dx will be rounded in the same way.
B. dx - dy == (double) (x-y). No. Let x = 0 and y = TMin32.
C. (dx + dy) + dz == dx + (dy + dz). Yes. Since each value ranges between TMin32 and
TMax32, their sum can be represented exactly.
D. (dx * dy) * dz == dx * (dy * dz). No. Let dx = TMax32, dy = TMax32 - 1, dz =
TMax32 - 2. (Not detected with Linux/GCC)
E. dx / dx == dz / dz. No. Let x = 0, z = 1.
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 < -149) {
        /* too small. return 0.0 */
        exp = 0;
        frac = 0;
    } else if (x < -126) {
        /* denormalized result */
        exp = 1;
        frac = 1 << (x + 149);
    } else if (x < 128) {
        /* normalized result */
        exp = x + 127;
        frac = 0;
    } else
    {
        /* too big. Return +oo */
        exp = 255;
        frac = 0;
    }

    /* 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?

A. $\pi \approx 11.001001000111110110112$.
B. $22/7 = 11.001001001001001\cdots_2$.
C. They diverge in the ninth bit to the right of the binary point.

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:

/* 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.

/* 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.

```c
/* Compute -f. If f is NaN, then return f. */
float_bits float_negate(float_bits f) {
    unsigned exp = f>>23 & 0xFF;
    unsigned frac = f & 0x7FFFFFF;
    unsigned mask = 1 << 31;
    unsigned neg = f ^ mask;
    if (exp == 0xFF && frac != 0)
        /* NaN */
        return f;
    return neg;
}
```