Problem 2.83 Solution:

This problem helps students appreciate the property of IEEE floating point that the relative magnitude of two numbers can be determined by viewing the combination of exponent and fraction as an unsigned integer. Only the signs and the handling of ± 0 requires special consideration.

code/data/floatcomp-ans.c

```c
int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    return
        (ux<<1 == 0 && uy<<1 == 0) ||  /* Both are zero */
        (sx && !sy) ||                 /* x < 0, y >=  0 */
        (!sx && !sy && ux <= uy) ||    /* x >= 0, y >= 0 */
        (sx && sy && ux >= uy);        /* x <  0, y <  0 */
}
```

Problem 2.88 Solution:

This problem requires students to think of the relationship between `int`, `float`, and `double`.

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` = $TMin_{32}$.

C. `(dx + dy) + dz == dx + (dy + dz)`. Yes. Since each value ranges between $TMin_{32}$ and $TMax_{32}$, their sum can be represented exactly.

D. `(dx * dy) * dz == dx * (dy * dz)`. No. Let `dx` = $TMax_{32}$, `dy` = $TMax_{32} - 1$, `dz` = $TMax_{32} - 2$. (Not detected with Linux/GCC)

E. `dx / dx == dz / dz`. No. Let `x = 0`, `z = 1`.

Problem 2.89 Solution:

This problem helps students understand the relation between the different categories of numbers. Getting all of the cutoff thresholds correct is fairly tricky. Our solution file contains testing code.

code/data/fpwr2-ans.c

```
/* Compute 2**x */
float fpwr2(int x) {

    unsigned exp, frac;
    unsigned u;

    if (x < -149) {
        /* Too small.  Return 0.0 */
        exp  = 0;
        frac = 0;
    } else if (x < -126) {
        /* Denormalized result */
        exp  = 0;
        frac = 1 << (x + 149);
    } else if (x < 128) {
        /* Normalized result. */
        exp  = x + 127;
        frac = 0;
    } else {
        /* Too big.  Return +oo */
        exp  = 255;
        frac = 0;
    }
    u = exp << 23 | frac;
    return u2f(u);
}
```

Problem 3.56 Solution:

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces students to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops.

A. We can see that `result` must be in register `%edi`, since this value gets copied to `%eax` at the end of the function as the return value (line 13). We can see that `%esi` and `%ebx` get loaded with the values of `x` and `n` (lines 1 and 2), leaving `%edx` as the one holding variable `mask` (line 4.)

B. Register `%edi` (`result`) is initialized to -1 and `%edx` (`mask`) to 1.

C. The condition for continuing the loop (line 12) is that `mask` is nonzero.

D. The shift instruction on line 10 updates `mask` to be `mask << n`.

E. Lines 6–8 update `result` to be `result ^ (x&mask)`.

F. Here is the original code:

code/asm/for.c

```
int loop(int x, int n)
{
    int result = -1;
    int mask;
    for (mask = 0x1; mask != 0; mask = mask << n) {
        result ^= (x & mask);
    }
    return result;
}
```

code/asm/for.c