number consists of a 1 in the rightmost bit, with the rest being 0. The exponent represents 2^{-126} and the fraction represents 2^{-23} so the value is 2^{-149}. This scheme provides for a graceful underflow by giving up significance instead of jumping to 0 when the result cannot be expressed as a normalized number.

Two zeros are present in this scheme, positive and negative, determined by the sign bit. Both have an exponent of 0 and a fraction of 0. Here too, the bit to the left of the binary point is implicitly 0 rather than 1.

Overflow cannot be handled gracefully. There are no bit combinations left. Instead, a special representation is provided for infinity, consisting of an exponent with all 1s (not allowed for normalized numbers), and a fraction of 0. This number can be used as an operand and behaves according to the usual mathematical rules for infinity. For example infinity plus anything is infinity, and any finite number divided by infinity is zero. Similarly, any finite number divided by zero yields infinity.

What about infinity divided by infinity? The result is undefined. To handle this case, another special format is provided, called **NaN (Not a Number)**. It too, can be used as an operand with predictable results.

PROBLEMS

1. Convert the following numbers to IEEE single-precision format. Give the results as eight hexadecimal digits.

   a. 9
   b. 5/32
   c. −5/32
   d. 6.125

2. Convert the following IEEE single-precision floating-point numbers from hex to decimal:

   a. 42E48000H
   b. 3F880000H
   c. 00800000H
   d. C7F00000H

3. The format of single-precision floating-point numbers on the 370 has a 7-bit exponent in the excess 64 system, and a fraction containing 24 bits plus a sign bit, with the binary point at the left end of the fraction. The radix for exponentiation is 16. The order of the fields is sign bit, exponent, fraction. Express the number 7/64 as a normalized number in this system in hex.

4. The following binary floating-point numbers consist of a sign bit, an excess 64, radix 2 exponent, and a 16-bit fraction. Normalize them.

   a. 0 1000000 0001010100000001

b. 0 0111111 0000001111111111
c. 0 1000011 1000000000000000

5. To add two floating-point numbers, you must adjust the exponents (by shifting the fraction) to make them the same. Then you can add the fractions and normalize the result, if need be. Add the single-precision IEEE numbers 3EE00000H and 3D800000H and express the normalized result in hexadecimal.

6. The Tightwad Computer Company has decided to come out with a machine having 16-bit floating-point numbers. The Model 0.001 has a floating-point format with a sign bit, 7-bit, excess 64 exponent, and 8-bit fraction. The Model 0.002 has a sign bit, 5-bit, excess 16 exponent, and 10-bit fraction. Both use radix 2 exponentiation. What are the smallest and largest positive normalized numbers on both models? About how many decimal digits of precision does each have? Would you buy either one?

7. There is one situation in which an operation on two floating-point numbers can cause a drastic reduction in the number of significant bits in the result. What is it?

8. Some floating-point chips have a square root instruction built in. A possible algorithm is an iterative one (e.g., Newton-Raphson). Iterative algorithms need an initial approximation and then steadily improve it. How can one obtain a fast approximate square root of a floating-point number?

9. Write a procedure to add two IEEE single-precision floating-point numbers. Each number is represented by a 32-element Boolean array.

10. Write a procedure to add two single-precision floating-point numbers that use radix 16 for the exponent and radix 2 for the fraction but do not have an implied 1 bit to the left of the binary point. A normalized number has 0001, 0010, ..., 1111 as the leftmost 4 bits of the fraction, but not 0000. A number is normalized by shifting the fraction left 4 bits and subtracting 1 from the exponent.