Chapter 7 -- floating point arithmetic

about FLOATING POINT ARITHMETIC

arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division

the operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation) -- example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction.

ADDITION

example on decimal value given in scientific notation:

```
   3.25 x 10 ** 3
 + 2.63 x 10 ** -1
 -----------------

 first step:  align decimal points
 second step:  add


   3.25     x 10 ** 3
 + 0.000263 x 10 ** 3
 --------------------
   3.250263 x 10 ** 3
    (presumes use of infinite precision, without regard for accuracy)

 third step:  normalize the result (already normalized!)
```

example on fl pt. value given in binary:

```
.25 =   0 01111101 00000000000000000000000

100 =   0 10000101 10010000000000000000000
```

to add these fl. pt. representations,
step 1: align radix points

shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1

shifting the mantissa RIGHT by 1 bit INCREASES THE EXPONENT by 1

we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end

```
         of the mantissa

       -> choose to shift the .25, since we want to increase it's exponent.
       -> shift by  10000101
                   -01111101
                   ---------
                    00001000   (8) places.

            0 01111101 00000000000000000000000 (original value)
            0 01111110 10000000000000000000000 (shifted 1 place)
                       (note that hidden bit is shifted into msb of mantissa)
            0 01111111 01000000000000000000000 (shifted 2 places)
            0 10000000 00100000000000000000000 (shifted 3 places)
            0 10000001 00010000000000000000000 (shifted 4 places)
            0 10000010 00001000000000000000000 (shifted 5 places)
            0 10000011 00000100000000000000000 (shifted 6 places)
            0 10000100 00000010000000000000000 (shifted 7 places)
            0 10000101 00000001000000000000000 (shifted 8 places)
```

step 2: add (don't forget the hidden bit for the 100)

```
         0 10000101 1.10010000000000000000000  (100)
      +  0 10000101 0.00000001000000000000000  (.25)
      ---------------------------------------
         0 10000101 1.10010001000000000000000
```

step 3: normalize the result (get the "hidden bit" to be a 1)

it already is for this example.

```
   result is
         0 10000101 10010001000000000000000
```

SUBTRACTION

like addition as far as alignment of radix points

then the algorithm for subtraction of sign mag. numbers takes over.

before subtracting,
  compare magnitudes (don't forget the hidden bit!)
  change sign bit if order of operands is changed.

don't forget to normalize number afterward.

MULTIPLICATION

example on decimal values given in scientific notation:

```
       3.0 x 10 ** 1
     + 0.5 x 10 ** 2
     -----------------
```

```
algorithm:  multiply mantissas
            add exponents

  3.0 x 10 ** 1
+ 0.5 x 10 ** 2
-----------------
 1.50 x 10 ** 3
```

```
example in binary:    use a mantissa that is only 4 bits so that
                      I don't spend all day just doing the multiplication
                      part.
```

```
  0 10000100 0100
x 1 00111100 1100
-----------------
```

```
mantissa multiplication:           1.0100
 (don't forget hidden bit)       x 1.1100
                                 ------
                                  00000
                                 00000
                                10100
                               10100
                              10100
                              ---------
                             1000110000
                   becomes   10.00110000
```

```
 add exponents:       always add true exponents
                      (otherwise the bias gets added in twice)

  biased:
  10000100
+ 00111100
----------
```

```
  10000100         01111111  (switch the order of the subtraction,
- 01111111       - 00111100   so that we can get a negative value)
----------       ----------
  00000101         01000011
  true exp         true exp
    is 5.           is -67
```

```
add true exponents      5 + (-67) is -62.

re-bias exponent:     -62 + 127 is 65.
     unsigned representation for 65 is  01000001.
```

put the result back together (and add sign bit).

```
1 01000001  10.00110000
```

normalize the result:
(moving the radix point one place to the left increases
the exponent by 1.)

```
1 01000001  10.00110000
  becomes
1 01000010  1.000110000
```

this is the value stored (not the hidden bit!):

```
1 01000010  000110000
```

DIVISION

similar to multiplication.

true division:
do unsigned division on the mantissas (don't forget the hidden bit)
subtract TRUE exponents

The IEEE standard is very specific about how all this is done.
Unfortunately, the hardware to do all this is pretty slow.

Some comparisons of approximate times:

2's complement integer add	1 time unit
fl. pt add	4 time units
fl. pt multiply	6 time units
fl. pt. divide	13 time units

There is a faster way to do division. Its called
division by reciprocal approximation. It takes about the same
time as a fl. pt. multiply. Unfortunately, the results are
not always the same as with true division.

Division by reciprocal approximation:

instead of doing   a / b

they do   a x 1/b.

figure out a reciprocal for b, and then use the fl. pt.
multiplication hardware.

example of a result that isn't the same as with true division.

true division: 3/3 = 1 (exactly)

reciprocal approx: 1/3 = .33333333

```
                              3 x .33333333 =  .99999999, not 1
```

It is not always possible to get a perfectly accurate reciprocal.

ISSUES in floating point

note: this discussion only touches the surface of some issues that people deal with. Entire courses could probably be taught on each of the issues.

rounding

arithmetic operations on fl. pt. values compute results that cannot be represented in the given amount of precision. So, we must round results.

There are MANY ways of rounding. They each have "correct" uses, and exist for different reasons. The goal in a computation is to have the computer round such that the end result is as "correct" as possible. There are even arguments as to what is really correct.

3 methods of rounding:

round toward 0 -- also called truncation.
figure out how many bits (digits) are available. Take that many bits (digits) for the result and throw away the rest. This has the effect of making the value represented closer to 0.

```
      example:
          .7783      if 3 decimal places available, .778
                     if 2 decimal places available, .77
```

round toward + infinity --
regardless of the value, round towards +infinity.

```
      example:
          1.23       if 2 decimal places, 1.3
         -2.86       if 2 decimal places, -2.8
```

round toward - infinity --
regardless of the value, round towards -infinity.

```
      example:
          1.23       if 2 decimal places, 1.2
         -2.86       if 2 decimal places, -2.9
```

in binary -- rounding to 2 digits after radix point

round toward + infinity --

```
              1.1101
                |
        1.11    |    10.00
                     ------

              1.001
                |
        1.00    |    1.01
                     -----
```

round toward - infinity --

```
              1.1101
                |
        1.11    |    10.00
       ------

              1.001
                |
        1.00    |    1.01
       -----
```

round toward zero (TRUNCATE) --

```
              1.1101
                |
        1.11    |    10.00
       ------

              1.001
                |
        1.00    |    1.01
       -----


             -1.1101
                |
       -10.00   |    -1.11
                     ------

             -1.001
                |
       -1.01    |   -1.00
                    -----
```

round toward nearest --

```
   ODD CASE:
   if there is anything other than 1000... to the right
   of the number of digits to be kept, then
       rounded in IEEE standard such that the least significant
       bit (to be kept) is a zero.


              1.1111
                |
        1.11    |    10.00
```

```
                       ------

                1.1101
                  |
          1.11    |    10.00
         ------

                1.001  (ODD CASE)
                  |
          1.00    |    1.01
         -----


               -1.1101  (1/4 of the way between)
                  |
         -10.00   |    -1.11
                       ------

               -1.001  (ODD CASE)
                  |
         -1.01    |   -1.00
                      -----
```

NOTE: this is a bit different than the "round to nearest" algorithm (for the "tie" case, .5) learned in elementary school for decimal numbers.

use of standards

--> allows all machines following the standard to exchange data
    and to calculate the exact same results.

--> IEEE fl. pt. standard sets
        parameters of data representation (# bits for mantissa vs. exponent)

--> Pentium architecture follows the standard

overflow and underflow

Just as with integer arithmetic, floating point arithmetic operations can cause overflow. Detection of overflow in fl. pt. comes by checking exponents before/during normalization.

Once overflow has occurred, an infinity value can be represented and propagated through a calculation.

Underflow occurs in fl. pt. representations when a number is to small (close to 0) to be represented. (show number line!)

if a fl. pt. value cannot be normalized
    (getting a 1 just to the left of the radix point would cause
     the exponent field to be all 0's)
    then underflow occurs.

HW vs. SW computing

----

floating point operations can be done by hardware (circuitry) or by software (program code).

-> a programmer won't know which is occuring, without prior knowledge of the HW.

-> SW is much slower than HW. by approx. 1000 times.

A difficult (but good) exercize for students would be to design a SW algorithm for doing fl. pt. addition using only integer operations.

SW to do fl. pt. operations is tedious. It takes lots of shifting and masking to get the data in the right form to use integer arithmetic operations to get a result -- and then more shifting and masking to put the number back into fl. pt. format.

A common thing that manufacturers used to do is to offer 2 versions of the same architecture, one with HW, and the other with SW fl. pt. ops.