Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times$ significand $\times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+\text{significand}) \times 2^{exponent - bias}$
- Example:
  - decimal: $-.75 = -3/4 = -3/2^2$
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Floating Point Add/Sub

- To add/sub two numbers
  - We first compare the two exponents
  - Select the higher of the two as the exponent of result
  - Select the significand part of lower exponent number and shift it right by the amount equal to the difference of two exponent
  - Remember to keep two shifted out bit and a guard bit
  - add/sub the signifand as required according to operation and signs of operands
  - Normalize significand of result adjusting exponent
  - Round the result (add one to the least significant bit to be retained if the first bit being thrown away is a 1
  - Re-normalize the result

Floating Point Multiply

- To multiply two numbers
  - Add the two exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Multiple significand portions
  - Results will be 1x.xxxxx… or 01.xxxx….
  - In the first case shift result right and adjust exponent
  - Round off the result
  - This may require another normalization step

Floating Point Divide

- To divide two numbers
  - Subtract divisor's exponent from the dividend's exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Divide dividend's significand by divisor's significand portions
  - Results will be 1.xxxxx… or 0.1xxxx….
  - In the second case shift result left and adjust exponent
  - Round off the result
  - This may require another normalization step