IEEE floating point

- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms
- Limited range and precision (finite space)
- Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers.
- Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Floating Point

- "real numbers" having a decimal portion != 0
- Example: 123.14 base 10
  - Meaning:
    - $1*10^2 + 2*10^1 + 3*10^0 + 1*10^{-1} + 4*10^{-2}$
  - Digit format: $d_m d_{m-1} \ldots d_1 d_0 \,.\, d_{-1} d_{-2} \ldots d_{-n}$
  - dnum → summation_of(i = -n to m) $d_i * 10^i$
- Example: 110.11 base 2
  - Meaning:
    - $1*2^2 + 1*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2}$
  - Digit format: $b_m b_{m-1} \ldots b_1 b_0 \,.\, b_{-1} b_{-2} \ldots b_{-n}$
  - bnum → summation_of(i = -n to m) $b_i * 2^i$
- "." now a "binary point"
- In both cases, digits on the left of the "point" are weighted by positive power and those on the right are weighted by negative powers

Floating Point

- Shifting the binary point one position left
  - Divides the number by 2
  - Compare 101.11 base 2 with 10.111 base 2
- Shifting the binary point one position right
  - Multiplies the number by 2
  - Compare 101.11 base 2 with 1011.1 base 2
- Numbers 0.111…11 base 2 represent numbers just below 1 → 0.111111 base 2 = 63/64
- Only finite-length encodings
  - 1/3 and 5/7 cannot be represented exactly
- Fractional binary notation can only represent numbers that can be written $x * 2^y$ i.e. $63/64 = 63*2^{-6}$
  - Otherwise, approximated
  - Increasing accuracy = lengthening the binary representation but still have finite space

Practice Page

- Fractional value of the following binary values:
  - .01 =
  - .010 =
  - 1.00110 =
  - 11.001101 =
- 123.45 base 10
  - Binary value =
  - FYI also equals:
    - 1.2345×10^{2} is normalized form
    - 12345×10^{-2} uses significand/mantissa/coeefficient and exponent

Floating point example

- Put the decimal number 64.2 into the IEEE standard single precision floating point representation…

  SEE HANDOUT

IEEE standard floating point representation

- The bit representation is divided into 3 fields
  - The single sign bit s directly encodes the sign s
  - The k-bit exponent field encodes the exponent
    - exp = $e_{k-1}...e_1e_0$
  - The n-bit fraction field encodes the significand M (but the value encoded also depends on whether or not the exponent field equals 0… later)
    - frac = $f_{n-1}...f_1f_0$
- Two most common formats
  - Single precision (float)
  - Double-Precision (double)

	Sign	Exponent	Fraction	Bias
Single Precision (4 bytes)	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision (8 bytes)	1 [63]	11 [62-52]	52 [51-00]	1023

The sign bit and the exponent

- The sign bit is as simple as it gets.
  - 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.
- The exponent field needs to represent both positive and negative exponents.
  - A *bias* is added to the actual exponent in order to get the stored exponent.
  - For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.
  - For double precision, the exponent field is 11 bits, and has a bias of 1023.

More on the “bias”

- In IEEE 754 floating point numbers, the exponent is biased in the engineering sense of the word – the value stored is offset from the actual value by the exponent bias.
- **Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**
- To solve this problem the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison.
- By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.
- When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.
- For a single-precision number, an exponent in the range −126 .. +127 is biased by adding 127 to get a value in the range 1 .. 254 (0 and 255 have special meanings).
- For a double-precision number, an exponent in the range −1022 .. +1023 is biased by adding 1023 to get a value in the range 1 .. 2046 (0 and 2047 have special meanings).

The fraction

- Typically called the “significand”
- Represents the precision bits of the number.
- It is composed of an implicit (i.e. hidden) leading bit and the fraction bits.
- In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form.
  - This basically puts the radix point after the first non-zero digit (see previous example)

FYI: A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa/significand has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it all together

- So, to sum up:
  - The sign bit is 0 for positive, 1 for negative.
  - The exponent's base is two.
  - The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
  - The first bit of the mantissa/significand is typically assumed to be 1.*f*, where *f* is the field of fraction bits.