

CSE 2421

IEEE 754 Floating Point – Part 1

Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*

- Chapter 2, Sections 2.4 through 2.4.4

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
- ▶ Floating point encodes rational numbers/values of the form
$$V = x \times 2^y$$
- ▶ Useful for computations involving:
 - Very large numbers: $|V| \gg 0$
 - Numbers very close to 0: $|V| \ll 1$
 - More generally, as *an approximation* to real numbers
- ▶ Single precision and double precision:
 - Single precision: 32 bits
 - Double precision: 64 bits

Floating point

- ▶ Limited range and precision (finite space):
- ▶ Range (for given precision):
 - Single: $+/- 2^{-149}$ to $+/- 2^{127}$
 - Compare with int: -2^{31} to $2^{31} - 1$
 - Double: $+/- 2^{-1074}$ to $+/- 2^{1023}$
 - Compare with long (ints): -2^{63} to $2^{63} - 1$
- ▶ **Overflow** means that values have grown too large for the representation, much in the same way that you can have integer overflow.
- ▶ **Underflow** (a value too small to be represented) is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Floating point

Question: What can/should you do if you are working with an application that regularly deals with numbers that are “too small to be represented” or “too large to be represented”?

Obviously, something, right???

It's not like any scientist or engineer I know to just say “It can't be done!”

What do you think?



Floating Point

- ▶ “Real numbers” having a decimal portion that is not necessarily equal to 0
- ▶ Example: 123.14 base 10
 - $1*10^2 + 2*10^1 + 3*10^0 + 1*10^{-1} + 4*10^{-2}$
 - Digit format: $d_m d_{m-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-n}$
 - $dnum \rightarrow \text{summation_of}(i = -n \text{ to } m) d_i * 10^i$
- ▶ Example: 110.11 base 2
 - $1*2^2 + 1*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2}$
 - Digit format: $b_m b_{m-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-n}$
 - $bnum \rightarrow \text{summation_of}(i = -n \text{ to } m) b_i * 2^i$
- ▶ In both cases, digits on the left of the “point” are weighted by a positive power and those on the right are weighted by a negative power

Floating Point

- ▶ Shifting the binary point one position left:
 - Divides the number by 2
 - Compare 101.11 ($5\frac{3}{4}$) base 2 with 10.111 ($2\frac{7}{8}$) base 2
- ▶ Shifting the binary point one position right:
 - Multiplies the number by 2
 - Compare 101.11 ($5\frac{3}{4}$) base 2 with 1011.1 ($11\frac{1}{2}$) base 2
- ▶ Numbers such as $0.111\dots11$ base 2 represent numbers just below 1
→ 0.1111111 base 2 = $63/64$ (decimal 0.984375), and $0.111111111 = 255/256$ (decimal 0.99609375)

Some limitations

- ▶ Only finite-length encodings can be stored
 - $1/3$ and $5/7$ cannot be represented exactly (generally, only rational numbers with denominators that are powers of 2, i.e., numbers such as $1/2$ or $3/16$, have a terminating, or finite-length binary encoding)
- ▶ Fractional binary notation can only represent numbers that can be written $x * 2^y$ (for integers x and y)
 - Example: $63/64 = 63 * 2^{-6}$; $255/256 = 255 * 2^{-8}$
 - Otherwise, value can only be approximated
 - Get increasing accuracy by lengthening the binary representation but still have finite space, so precision will always be limited

Scientific Notation

- ▶ When we represent a number in scientific notation
 - Typically only 1 digit to the left of the decimal point
 - $6.385 * 10^8$
 - $-9.66712 * 10^{17}$
 - Four fields
 - Sign
 - Left side of decimal point
 - Right side of decimal point
 - exponent
- ▶ Hold these thoughts as we move on to IEEE 754

IEEE 754 encoding

- ▶ The bits in the encoding of the real number value are divided into three fields (details about the sizes of these fields on the next slide):
 - S field (or sign field): encodes the sign of the number;
 - E field (or exponent field): encodes the exponent of the real number (positive, 0 or negative), in biased form;
 - F field (or significand/mantissa field): which encodes a number greater than or equal to 0, and less than 2, to which the exponent is raised to obtain the real value encoded by the entire IEEE 754 representation (this is x in a representation of the value of the number using $|V| = x \times 2^y$)

Biased Integer Representation

an integer representation that skews the bit pattern so as to look just like unsigned but actually represent negative numbers.

examples: given 4 bits, we BIAS values by 2^3 (8)

TRUE VALUE to be represented	3
add in the bias	+8

unsigned value	11

so the bit pattern of 3 in 4-bit biased-8 representation will be 1011

Biased Integer Representation (cont)

going the other way, suppose we were given a biased-8 representation of 0110

unsigned 0110 represents	6
subtract out the bias	- 8

TRUE VALUE represented	-2

this representation allows operations on the biased numbers to be the same as for unsigned integers, but actually represents both positive and negative values.

choosing a bias:

the bias chosen is most often based on the number of bits available for representing an integer. To get an approx. equal distribution of true values above and below 0, the bias should be $2^{(n-1)}$ or $2^{(n-1)} - 1$

Floating point


- ▶ The real number decimal value represented is: $(-1)^s \times f \times 2^e$



- ▶ S is one bit representing the sign of the number (it directly encodes s)
 - 0 for positive, 1 for negative
- ▶ E is an 8-bit **biased** integer (NOTE: **not B2T!**) representing the exponent e
 - $e = E - \text{bias} \rightarrow E = e + \text{bias}$
- ▶ F is an unsigned integer (ϵ depends on the E field: if E is 0, ϵ is zero; otherwise, ϵ is 1)
 - $f = (F / (2^n)) + \epsilon \rightarrow F = (f - \epsilon) * 2^n$
- ▶ For 32 bit, $n = 23$ and $\text{bias} = 127$
- ▶ For 64 bit, $n = 52$ and $\text{bias} = 1023$

	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

Representing Floating Point Numbers

- ▶ Think of IEEE 754 as similar to scientific notation.
 - ▶ IEEE 754 standard is used so that there is consistency across computers
 - ▶ This is not the only way to represent floating point numbers, it is just the IEEE standard way of doing it.
- 

Floating point example

- ▶ Put the decimal number 64.2 into the IEEE standard single precision floating point representation.
- ▶ **5 step process** to obtain Sign(S), Exponent(E), and Mantissa(F)

Floating point example

Step 1:

Get a binary representation for 64.2, with at least the number of bits for the F field + 1 (24 for single precision, or 53 for double precision). To do this, get unsigned binary representations for the stuff to the left and right of the decimal point separately.

64 is 1000000

.2 can be gotten using the algorithm:

	<u>Whole number part</u>
.2 x 2 = 0.4	0
.4 x 2 = 0.8	0
.8 x 2 = 1.6	1
.6 x 2 = 1.2	1
.2 x 2 = 0.4	0 now this whole pattern (0011) repeats.
.4 x 2 = 0.8	0
.8 x 2 = 1.6	1
.6 x 2 = 1.2	1

Write the whole number parts from top to bottom, right to left:

So, a binary representation for .2 is .001100110011...

Putting the halves back together again, and writing 24 bits:

64.2 is 1000000.00110011001100110

Floating point example


Step 2:

Normalize the binary representation. (i.e. Make it look like scientific notation) with a single 1 to the left of the binary point (for now – more on this later):

1.0000 0000 1100 1100 1100 110 $\times 2^6$

NOTE: the bit to the left of the binary point will be considered the 'bit' (it does not appear in the encoding, but is ***implied by the value of the exponent***)

The “Hidden Bit”

- ▶ There are 23 bit for the mantissa in a single precision IEEE 754 representation.
 - ▶ It turns out that, if floating point numbers are always stored in a normalized form, then the leading bit (the one on the left of the binary point) is always a 1. So, why store it at all?
 - ▶ We can put it back into the number (giving 24 bits of precision for the mantissa) for any calculation, but we only have to store 23 bits.
- 

Floating point example

Step 3:

6 is the true exponent (e). For the standard form, to get E , it needs to be in 8-bit, biased-127 representation (add 127 to the true exponent to get the biased-127 representation).

$$\begin{array}{r} 6 \\ + 127 \\ \hline 133 \end{array}$$

Convert this decimal value to binary: 133 in 8-bit, unsigned representation is 1000 0101

This is the bit pattern used for E in the standard form.

Floating point example

Step 4:

The mantissa/significand stored (F) is the stuff to the right of the radix point in the normalized form (**1.0000 0000 1100 1100 1100 110** $\times 2^6$)

We need **23 bits** of it for single precision.

0000 0000 1100 1100 1100 110

Floating point example

Step 5: Put it all together (and include the correct sign bit: the original number is positive, so the sign bit is 0):

S	E		F					
0	1000	0101	0000	0000	1100	1100	1100	110
0b	0100	0010	1000	0000	0110	0110	0110	0110
0x	4	2	8	0	6	6	6	6

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 (E) contains 127 (the bias) plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
 - The first bit of the mantissa/significand is implied to be 1 for biased exponents in the range 1 – 254 (but assumed to be 0 for exponent of 0); i.e., the mantissa/significand is interpreted as 1.F, where F is the field of fraction bits.

Floating point example

► Given binary 1 100 0100 1000 0000 0010 1100 0000 0000

$S = 1$

$E = 1000\ 1001 = 137 = 127 + 10 \rightarrow e = 10$

$F = 000\ 0000\ 0010\ 1100\ 0000\ 0000$

$f = 1.000\ 0000\ 0010\ 1100\ 0000\ 0000 \times 2^{10}$

$1000\ 0000\ 001.0\ 1100\ 0000\ 0000$

$= -1025.375$