Fixed Point

We are mostly discussing floating point numbers today, but I want to mention briefly that some computers represent non-integers numbers using *fixed* point. Fixed point means we have a constant number of bits (or digits) to the left and right of the binary (or decimal) point.

For example, we might have eight digits to the left of the decimal point and two digits to the right. An example is 23953223.49. You are familiar with representations have two digits to the right. That is how our currency works (but the number to the left is not fixed with currency). An example in binary is 10.1101 where we have two and four bits to the left and right, respectively. Note that I used the term "digits" for base 10, and "bits" for base 2.

We can represent negative numbers in fixed point using two's complement. Take the number 6.5 in decimal which is 110.1 in binary. Suppose (arbitrarily) that we represent it with four bits to the left and four to the right of the binary point. To represent -6.5 in binary, we invert the bits and add what we need to get back to 0.

$$\begin{array}{rll} & 0110.1000 & \\ & 1001.0111 & \leftarrow \text{invert bits} \\ + & \underline{0000.0001} & \leftarrow \text{add .0001} \\ & 0000.0000 & \end{array}$$

We get the representation of -6.5 by adding the 2nd and 3rd rows of the sum:

$$\begin{array}{rll} & 1001.0111 & \leftarrow \text{invert bits} \\ + & \underline{0000.0001} & \leftarrow \text{add .0001} \\ & 1001.1000 & \leftarrow \text{-6.5 as signed fixed point} \end{array}$$

Scientific Notation

Let's return to floating point and consider a number representation called "scientific notation," which most of you learned about in high school. You know that you can represent very large decimal numbers or very small decimal numbers by writing, for example:

$$300,000,000 = 3 \times 10^8 = \texttt{3.0E} + \texttt{8}$$

which is the speed of light in meters per second, and

$$.00000456 = 4.56 \times 10^{-6} = \texttt{4.56E} - \texttt{6}$$

which has no particular significance to my knowledge.

In *normalized* scientific notation in base 10, there is exactly one digit to the left of the decimal point and this digit is from 1 to 9 (not 0). *Note that the number 0 cannot be written in normalized scientific notation.* Examples of *normalized* binary numbers are shown blow on the right side of the equations:

$$(1000.01)_2 = 1.00001 \times 2^3$$

$$(0.111)_2 = 1.11 \times 2^{-1}$$

A normalized binary number has the form:

$$1.\underline{\hspace{3cm}} \times 2^E$$

where the ____________ is called the *significand* (or *mantissa*) and E is the *exponent*. Note that it is impossible to encode the number 0 in normalized form.

The number of bits in the significand is called the number of significant bits. Note that we don't count the first bit when counting the number of significant bit. (In decimal, we do have to specify the most significant digit since it can be anything from 1 to 9, so it is included in the count.)

IEEE floating point standard

To represent floating point numbers in a computer, we need to make some choices:

- how many bits to use for the significand?
- how many bit to use for the exponent?
- how to represent the exponent?
- how to represent the sign of the number?
- how to represent the number 0 ?

Until the late 1970's, different computer manufacturers made different choices. This meant that the same software produced different results when run on different computers. Obviously this was not desirable. A standard was introduced in 1985 by the IEEE organization.[1] The standard (number ”754”, i.e. there are lots of different standards) consists of two representations for floating point numbers. One is called *single precision* and uses 32 bits. The other is called *double precision* and uses 64 bits. This representation is used in nearly all computers you will find today.

Single precision

Single precision represents a floating point number using 32 bits (4 bytes, that is, one byte is 8 bits). We number the bits from right to left. Bit 31 is the leftmost and bit 0 is the rightmost.

- bits 0-22 (23 bits) are used for the significand
- bits 23-30 (8 bits) are used for the exponent
- bit 31 (1 bit) is used for the sign (0 positive, 1 negative).

Because the standard uses normalized numbers, the 1 bit to the left of the binary point in 1.____________ does not need to be coded since it is always there. The significand only refers to what is to the right of the binary point.

The coding of the exponent is subtle. You might expect the exponent should be encoded with two's complement, since you want to allow positive exponents (large numbers) and negative exponents (small numbers). This is not the way the IEEE chose to do it, however.

[1] Institute of Electrical and Electronics Engineers, Inc. See `http://www.ieee.org`

The two exponent codes (00000000, 11111111) are *reserved* for special cases.

- The exponent 00000000 is used to represent a set of numbers lying strictly in the tiny interval $(-2^{-126}, 2^{-126})$ which includes the number 0. These numbers are called *denormalized numbers.*

$$\begin{array}{rll}
 & 1.00000000000000000000000 \times 2^{-126} & \leftarrow \text{smallest positive normalized number} \\
 & 0.11111111111111111111111 \times 2^{-126} & \leftarrow \text{largest denormalized number} \\
 & 0.11111111111111111111110 \times 2^{-126} & \\
 & 0.11111111111111111111101 \times 2^{-126} & \\
 & \vdots & \\
 & 0.00000000000000000000010 \times 2^{-126} & \\
 & 0.00000000000000000000000 \times 2^{-126} & \leftarrow \quad 0 \\
- & 0.00000000000000000000001 \times 2^{-126} & \\
- & 0.00000000000000000000010 \times 2^{-126} & \\
 & \vdots & \\
- & 0.11111111111111111111110 \times 2^{-126} & \\
- & 0.11111111111111111111111 \times 2^{-126} & \leftarrow \text{smallest denormalized number} \\
 & & \\
- & 1.00000000000000000000000 \times 2^{-126} & \leftarrow \text{largest negative normalized number}
\end{array}$$

  Notice that, to represent 0, the exponent bits and significand bits are all 0's.

- The exponent 11111111 is used to represent either infinity (plus or minus, depending on the sign bit), or the case that there is no number stored (called "not a number", or NaN). This case could arise if your program has declared a floating point variable but you haven't assigned it a value yet.

  These special cases are determined by the significand. If the significand is all 0's (23 of them), then the interpretation is $\pm\infty$, depending on the sign bit. If the significant is not all zeros, then the interpretation is NaN.

There are 254 remaining 8-bit exponent codes, namely from 00000001 to 11111110, which are used for normalized numbers. These codes represent exponents ranging from -126 through 127.

exponent code	exponent value	
00000001	-126	← smallest exponent value
00000010	-125	
:	:	
01111110	-1	
01111111	0	← the code of value 0 is the "bias"
10000000	1	
10000001	2	
:	:	
11111110	127	← largest exponent value

To convert from this 8-bit exponent code to the exponent value that it represents, you interpret the 8-bit code as an unsigned number (from 1 to 254) and then subtract the value 127 (called the "bias") which is the code for the exponent value of 0.

How should you think about the discretization of the real line that is defined this way? Think of consecutive powers of 2, namely $[2^e, 2^{e+1}]$. That interval is of size 2^e. Partition that interval into 2^{23} equal sized pieces, each size being 2^{e-23}. Note that the step size between "samples" is constant within each interval $[2^e, 2^{e+1}]$ but that the step size doubles in each consecutive power of 2 interval.

An Example

How is 8.75 coded up in the IEEE single precision standard? First you show that

$$8.75 = (100011)_2 \times 2^{-2}$$

using the techniques discussed earlier. Then, you shift the binary point to normalize the number,

$$(100011)_2 \times 2^{-2} \ = \ (1.00011)_2 \times 2^3$$

The significand is $\underline{00011}000000000000000000$ where I have underlined the part that came from the above equation. Note that we have dropped the leftmost 1 bit, because we are using normalized numbers.

The exponent code is determined by adding 127 to the exponent value, i.e. $3 + 127 = 130$, and writing 130 as an unsigned number $(10000010)_2$. The sign bit is 0 since we have a positive number. Thus, 8.75 is encoded with the 32 bits

$$0 \ 10000010 \ 00011000000000000000000.$$

We can write it in hexadecimal as `0x410c0000`.

Suppose we want to add two single precision floating point numbers, x and y, e.g.

$$\begin{aligned} x &= 1.1010100000000000000000000101 * 2^{-3} \\ y &= 1.0010010000101010101010101010 * 2^2 \end{aligned}$$

where x has a smaller exponent than y. We cannot just add the significands bit by bit, because then we would be adding terms with different powers of 2. Instead, we need to rewrite one of the two numbers such that the exponents agree. The usual way to do this is to rewrite the smaller of the two numbers, that is, rewrite the number with the smaller exponent. So, for the above example, we rewrite x so that it has the same exponent as y:

$$\begin{aligned} x &= 1.1010100000000000000000000101 * 2^{-3} \\ &= 1.1010100000000000000000000101 * 2^{-5} * 2^2 \\ &= 0.000011010100000000000000000000101 * 2^2 \end{aligned}$$

Notice that x is no longer normalized, that is, it does not have a 1 to the left of the binary point.

Now that x and y have the same exponent, we can add them.

$$\begin{array}{rl} & 1.0010010000101010101010101010 * 2^2 \\ + & \underline{0.000011010100000000000000000000101 * 2^2} \\ = & 1.001100010110101010101010101000101 * 2^2 \end{array}$$

The sum is a normalized number. But the significand is more than 23 bits. If we want to represent it as single precision float, then we must truncate (or round off, or round up) the extra bits.

A related point, which I discussed last lecture, is that most numbers that have a finite number of digits (base 10) require an infinite number of bits to represent exactly. This means that when you try to represent these numbers as floating point in a computer, you will to approximate them. These can lead to surprising results. Take the example 0.05 i.e. 1/20.0 that we saw last class, and consider the following code:

```
float x = 0;
for (int ct = 0; ct < 20; ct ++) {
   x += 1.0 / 20;
   System.out.println( x );
}
```

It prints out up to eight digits to the right of the decimal point.

```
0.05
0.1
0.15
0.2
0.25
0.3
0.35000002
0.40000004
0.45000005
0.50000006
etc
```

None of these numbers is represented exactly in the computer! The computer is just printing the best approximation it can with eight digits to the right of the decimal point. For example, 0.1 means 0.10000000 and it doesn't bother writing the seven 0's on the right because people don't generally want to see those 0's. But that's just convention.

Why eight digits? The answer is presumably that $2^{23} \approx 10^8$ and so the number that is being approximated with with those 23 bits of significand can be approximated just as well with eight digits.

Double precision

The 23 bits of significand might seem like alot at first glance. But 2^{23} is only about 8 million. When we try to represent integers larger than this value, we find that we have problems. We don't have enough precision and we have to skip some. (See Exercises 1 Q10c.)

In order to get more precision, we need more bits. The IEEE double precision representation uses 64 bits instead of 32. 1 bit is used for the sign, 11 bits for the exponent, and remaining 52 bits for the significand ($1 + 11 + 52 = 64$). A similar idea is used for the exponent bias except that now the bias is 1023 ($2^{10} - 1$), rather than 127 ($2^7 - 1$). Conceptually, there is nothing new here beyond single precision, though.

In the slides, I went through the same examples as I did for single precision. I wrote out the exponent codes for double precision, and I encoded 8.75 as a double. I suggest you work them out for yourself, and verify with the slides. I also gave a similar Java example for double.

```
double x = 0;
for (int ct=0; ct < 10; ct ++) {
   x += 1.0 / 10;
   System.out.println( x );
}
```

which prints out

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Note that it prints out up to about 16 digits. The reason for 16 is that

$$2^{52} = (2^1 0)^5 * 2^{\approx} (10^3)^5 * 4 = 10^{16}$$

and so with 52 bits we can represent about the same number of numbers as we can with 10 digits. Of course, we are still approximating.

Note that the 3rd line has 17 digits to the right of the decimal point. Also, it seems strange that the third line is slightly greater than .1 * 3 whereas the 7th-9th lines are slightly less than .1 * 7-9 respectively. It is not obvious why this behavior occurs. (Time permitting, I will figure it out later – or if one of you wishes to figure it out and let me know, please do.)

IEEE converter web site

Here is a useful tool to check out if you want to practice this stuff.
`http://www.h-schmidt.net/FloatConverter/IEEE754.html`