Module 0100: Floating point numbers

Tak Auyeung, Ph.D.

November 27, 2006

1 About this module

- Prerequisites:
- Objectives: This module explores how numbers are represented in the floating point representation.

2 What is the "point"?

Let us use base-10 for the moment. We know that 265 means $200 + 60 + 5 = 2(10)^2 + 6(10)^1 + 5(10)^0$. Then, we can naturally extend this logic to numbers that have a decimal point.

For example, 265.89 means $2(10)^2 + 6(10)^1 + 5(10)^0 + 8(10)^{-1} + 9(10)^{-2}$. because 0.8 is really 8(0.1), and 0.09 is really 9(0.01).

Now, we can apply this representation to binary numbers. We know that $1011_2 = 1(2)^3 + 0(2)^2 + 1(2)^1 + 1(2)^0$, then $1011.011_2 = 1(2)^3 + 0(2)^2 + 1(2)^1 + 1(2)^0 + 0(2)^{-1} + 1(2)^{-2} + 1(2)^{-3} = 8 + 2 + 1 + 0.25 + 0.125 = 11.375$. Of course, in this case, the point should be called a "binary point" rather than a "decimal point".

In summary, the "point" of a number notation is simply a separator to separate the digit for b^0 and the digit for b^{-1}, where b is the base of the representation (10 for decimal, 2 for binary, and etc.).

3 Fixed (implied) point representation

3.1 The representation

A fixed point representation is essentally an integer, but with an implied point between two fixed digit positions.

In some applications, it is okay to use a fixed point representation for numbers. For example, in the U.S., we frequently measure length by inches, and weight by pounds. Both are exceptionally well suited for fixed point representation.

If we need to represent weight quantities down to ounces, then this by itself is already a fixed point representation, only that the unit is a pound. Let me illustrate. The weight "5 pounds and 7 ounces" can be expressed as a fixed point number. Because there are 16 ounces in a pound, we can rewrite this as "101_2 pounds and 111_2 ounces". Next, we convert the ounce units into pound units. There are 16 (10000_2) ounces in a pound, so the weight we want to represent is "101_2 and $111_2/10000_2$ pounds".

The rule of division is the same regardless of the base. In other words, in binary, $111_2/10_2 = 11.1_2$, $111_2/100_2 = 1.11_2$ and so on. This means that the weight can be represented as 101.0111_2 pounds.

If the required precision is limited to ounces, then we can fix the number of "binary places" to 4.

3.2 Operators

The use of fixed point representation means that we can represent any weight using a regular integer, and just understand that a point is implied between bit 3 and bit 4 (remember, the least significant bit is bit 0). In other words, given an integer with a "native" value of x, it is implied that that its represented value x' is actually $x/10000_2$.

This also means that we can use the usual integral operators for addition, subtraction and comparison.

3.3 Multiplication

However, multipliciation and division require some attention. This is because natively, $z = xy$ is an integer multiplication, given x and y are integers. However, using the fixed point notation, the represented values are $z' = x'y' = \left(\frac{x}{10000_2}\right)\left(\frac{y}{10000_2}\right) = \frac{xy}{100000000_2}$. However, using the implicit binary point, the *interpreted* value of z is $\frac{xy}{10000_2}$.

In other words, the implicit binary point of the product is no longer between bit 3 and bit 4, but between bit 7 and bit 8! If we do interpret z with an implicit binary point between bit 3 and bit 4, then the value is 10000_2 too big.

This problem can be fixed in two ways. One is to understand the product term has an implicit binary point between bit 7 and bit 8. However, this method requires the programmer to keep track of different fixed point representations. It does offer one advantage: precision. No precision is lost in the multiplication.

The second method is to shift the product four times to the right so that the binary point is at the right place again. Let us use the $\mathrm{rs}(x, n)$ notation to mean right shift a number x n bit positions. Then the most intuitive way to adjust (normalize) the product is $z = \mathrm{rs}(xy, 4)$. This means a product bit pattern 1011010110010101_2 will become 101101011001_2, which is then interpreted as 10110101.1001_2,

3.4 Rounding versus truncating

The right shift method by itself is a truncation operator. The least significant four bits are simply gone. This means the value 1.11110000_2 and the value 1.11111111_2 are both truncated to the same represented value of 1.1111_2. This appears to be okay, but it has very bad effect "in the long run".

If we carry out multiplications several times, or add the products of several terms, then the effect of truncation will become apparent. The computed values will becomes smaller and smaller compared to the actual value. This is because truncation biases to a smaller value.

Let us rethink this problem. 1.11110000_2 should definitely be *rounded* to 1.1111_2. However, 1.11111111_2 is much closer to 10.0000_2 than 1.1111_2. As a result, it makes sense to round 1.11111111_2 to 10.0000_2. We can, then, generalize and say that we round a number to a less precise representation based on whether it is closer to the smaller value or the larger value.

There is one problem left. What about 1.11111000_2? It is actually exactly half way beteween 1.11110000_2 and 10.0000_2. What we need to consider in this case is: how many values are rounded to 1.1111_2, and how many values are rounded to 10.1111_2? The two numbers should be the same.

Because 1.11110000_2 is "rounded" to 1.1111_2, this means we have $1.11110000_2, \ldots 1.11110111_2$ rounded to 1.1111_2. That makes 8 distinct values. It makes sense, then, to round 1.11111000_2 to 10.0000_2 so that all values $1.11111000_2, \ldots 1.11111111_2$ (8 of them) are arounded to 10.0000_2.

Rounding is not difficult, we only need to add 1000_2 to $z = xy$ before the right shift operation. In other words, we want to make $z = \mathrm{rs}((xy + 1000_2), 4)$.

3.5 How about signed numbers?

The only thing we have to be careful about is that the right shift operation needs to be sign-aware. In other words, we want the right shift operator to replicate the most significant bit when we right shift. This way, we can preserve the sign of a number as we right shift.

3.6 How about division?

Dividing two fixed point number presents another problem. Let n' and r' represent the actual values of the dividend and divisor, respectively. We'll also use x' to represent the actual value of $\frac{n'}{r'}$. n and r are the fixed point representations of n' and r'. Based on our assumption that the implicit binary point is between bit 3 and bit 4, $n' = n/10000_2$, and $r' = r/10000_2$.

The question is whether $x = \frac{n}{r}$ is the actual representation of x'. Using the represented values, $x' = \frac{n/10000_2}{r/10000_2}$. Let us first compute $x'' = n/r$. Note that $x' = x''$! This is bad, because the *representation* of x' is supposed to be $x'(10000_2)$. In other words, x'' does not have an implicit binary point between bit 3 and bit 4! x'' is not expressed in units of $1/10000_2$. If we interpret x'' as a fixed point number, its interpreted value will be 10000_2 smaller than it should be.

Format	Sign	Exponent	Exp. Bias,	Mantissa
Single precision	31	23-30	127	0-22
Double precision	64	52-62	1023	0-51

Table 1: IEEE floating point formats.

To make x'' express in units of $1/10000_2$, we can first multiply n by 10000_2, then carry out the division. This means that we can compute $x = \frac{n(10000_2)}{r}$ so that. This makes $x'' = (10000_2)(n'/r')$, which also means that $x''/10000_2$ is the approximate value of n'/r'.

Do we have rounding problems? Yes! This is because the actual division between n and r is done by integer division. Integer division truncates the fractional part of the result!

We can fix this problem by first multiply the dividend by 100000000_2, get the quotient, round it, then right shift it 4 times. In other words, we can compute the fixed point result as $x = \text{rs}(((n(100000000_2))/r) + 1000_2, 4)$.

4 Floating point

Fixed point representation is more useful than most programmer know. Many lossy compression algorithms (such as MP3) can be done in fixed point computations. However, there are a few problems associated with fixed point computation. First of all, the smallest and largest absolute values that can be represented by a fixed point representation are limited by the implicit fixed binary point. This makes fixed point representation not very useful for applications where the range of numbers can be huge.

This is why we have floating point numbers.

4.1 The representation

The representation of a number using the floating point method has three components:

- The mantissa.is a number $n \in [1 \ldots 2)$, which means it is at least 1, and less than 2.
  - Note that the first part of a mantissa is implied. In other words, the 1. portion is implied. This means a mantissa 1.01101_2 is represented as the bit pattern 01101_2.
- The sign s indicates whether a number is negative or not. It is conventional to use a value of 1 to indicate a negative value, and 0 to indicate a non-negative value.
- The exponent x specifies the magnitude 2^{x-b}, where b is an offset. so that x can be an unsigned number.

The value represented by such a floating number is, then, $s = 0 \Rightarrow v = n2^{x-b}, s = 1 \Rightarrow v = -n2^{x-b}$. Of course, we immediately have a problem: how do we represent 0? Let's defer this topic for now.

4.2 IEEE floating point formats

The IEEE (Institute of Electrical and Electronics Engineers) has established two floating point number formats. A single precision format (`float` in C) uses 32 bits, while a double precision format (`double` in C) uses 64 bits.

Counting bits from the least significant digit, table 1 describes the two formats.

4.2.1 Special values

Certain bit patterns of a floating number are special, and they do not represent the values that they normally should. This is necessary to represent special values for floating point related computations.

For this discussion, let us assume the exponent has x bits, e be the unsigned raw value of the exponent ($e \in [0 \ldots 2^x - 1]$). Furthermore, assume the mantissa has n bits, and m be the unsigned raw value of the mantissa bit pattern ($m \in [0 \ldots 2^n - 1)$. The exponent bias is $2^{x-1} - 1$.

- $e = 0 \wedge m = 0$: the value is zero. Note that without this special interpretation, the bit pattern represents the least absolute value that can be represented, which is $1 \cdot 2^{0-(2^{x-1}-1)}$.
- $e = 2^x - 1 \wedge m = 0$: the bit pattern represents the absolute value of infinity. If the sign bit is one, then the bit pattern represents $-\infty$. If the sign bit is zero, then the bit pattern represents ∞.
- $e = 2^x - 1 \wedge m \neq 0$: the bit pattern represents the "NAN" (Not A Number).

sign	exponent	mantissa
1	$129 = 10000001_2$	01101000000000000000000

Table 2: Representation of -5.625 as a single precision floating point number.

sign	exponent	mantissa
0	$124 = 01111100_2$	10011001100110011001101

Table 3: Representation of 0.2 as a single precision floating point number.

4.2.2 Example 1

The representation of -5.625 as a single precision floating point number.

The sign bit is 1 because the value being represented is negative.

The binary representation of 5.625 is 101_2 plus $.101_2$. This means the *unnormalized* representation is $101.101_2 \cdot 2^0$.

A floating point number is normalized when the value to the left of the binary point of the mantissa is exactly one. We can normalize a value by shifting the binary point and adjusting the exponent at the same time. Let us define $\text{ls}(x, n)$ be the left shift function, it left shifts the binary representation of value x by n bits.

Then, $m \cdot 2^e = \text{ls}(m, 1) \cdot 2^{e-1} = \text{rs}(m, 1) \cdot 2^{e+1}$.

Knowing this, $101.101_2 \cdot 2^0 = 1.01101_2 \cdot 2^2$ after two right shift operations. Remember, however, that only the digits to the right of the binary point are represented. The exponent is also represented using a biased format. The bias of a single precision floating point number is 127, as a result, $127 + 2 = 129$ should be represented as the exponent bit pattern.

As a result, the representation of -5.625 is illustrated in table 2.

4.2.3 Example 2

Let us try to represent 0.2 as a single precision floating point number. In order to figure out the binary representation of 0.2 as fractions of powers of 2, we can first multiply a number (a power of 2) to 0.2 so that it has 25 significant binary digits. Note that there is one more digit than what we need for rounding purposes.

The minimum power of 2 to multiply to 0.2 to make it greater than or equal to 1 is 8 because $0.2 \cdot 2^3 = 1.6$. After that, we multiply the value by 2^{24} to get a 25-bit representation (25 significant digits to the left of the binary point).

$0.2 \cdot 2^{3+24} = 26853545 = 11001100110011001100_2$. Do you recognize the repeating pattern of 1100_2? This is because 0.2 cannot be represented by a binary number with a finite number of digits to the right of the binary point! This is much like trying to represent $\frac{1}{3}$ in decimal.

At any rate, because the least significant digit is 1, we round up the mantissa as $11001100110011001100110_2 \cdot 2^{-27}$. After normalizing it (right shift 24 times), we end up with the representation of $1.10011001100110011001101_2 \cdot 2^{-3}$. The exponent bias is 127, this means the exponent binary pattern needs to represent $127 - 3 = 124 = 01111100$.

As a result, the representation of 0.2 is illustrated in table 3.

4.2.4 Visualizing floating point representation

This can done using a good debugger and a simple sample program. We'll use the program in listing 1 as an example to find out the bit pattern of 0.2 as a single precision floating point number.

Listing 1: check format

```
1 int main(void)
2 {
3   float f;
4
5   f = 0.2;
6   return 0;
7 }
```

We compile the program using the command `gcc -g -o test test.c` to make sure the debug symbol table remains intact. Next, we invoke the debugger using the command `gdb test`. Once we enter the debugger, we use the command `b 6` to insert a breakpoint on line 6.

Then, we use the command `run` in `gdb` to run the program. The programs stops on line 6. Here, we can examine the value of variable `f` as a 32-bit binary number using the commmand `x/tw &f` to examine the variable in binary.

We should see exactly the same bit pattern as table 3.

4.3 Floating point calculations

4.3.1 Multiplication

Multiplication for floating point numbers is actually the simplest operation! It is almost the same as the multiplication of two fixed point numbers, where the implied binary point is between bit 22 and bit 23. Recall that bit 23 is the implied 1. of the mantissa.

Let us assume that we are computing $a \cdot b$. a_s, a_e and a_m represent the sign bit, the exponent value (not the raw bit pattern) and the mantissa value (not the raw bit pattern) of a, respectively. Similar symbols are defined for b and the product term p.

Then, the absolute value of the product is $a_m \cdot b_m \cdot 2^{a_e+b_e}$. The sign follows this rule: $p_s = a_s \oplus b_s$.

Of course, things are not quite as simple as they seem. First of all, the multiplication of the mantissas is a 24×24 multiplication, which yields a 48-bit result. Each operand is a 24-bit number because there are 23 represented mantissa digits, plus one implied 1. on the left hand side.

Let p'_m represent the raw bit pattern 48-bit product. We need to round this value because only 24 of these 48 bits can be represented in the product's mantissa. The rounding is done by adding 1 to the 48-bit raw product pattern.

Next, the raw product bit pattern needs to be right shifted $23 + 23 = 46$ times to represent the actual value p_m. Note that the value of digits on the left hand side of the binary point can range from 01_2 to 11_2. This means we may need to normalize the product.

4.3.2 Division

Division of two floating point numbers is about the same as division for two fixed point numbers. In this case, we can assume the implied binary point is between bit 22 and bit 23 for the representation of the mantissa of a single precision floating point number.

The exponent of the result is $p_e = a_e - b_e$, assuming a is the dividend, and b is the divisor (a_e and b_e are the exponent values of a and b, respectively).

The result of the division may need to be normalized.

4.3.3 Addition/subtraction

Addition and subtraction are actually somewhat complicated for floating point numbers. This is because the exponent of two numbers may not be the same. As a result, the number with the smaller exponent value needs to be shifted to match the exponent of the the number with the larger exponent.