Introduction to Fixed Point Number Representation

CS61c Spring 2006

Author: Hayden So
Last Modified: 2/28/06

Introduction: Real numbers in Real world

In real life, we deal with real numbers -- numbers with fractional part. Most modern computer have native (hardware) support for *floating point* numbers. However, the use of floating point is not necessarily the only way to represent fractional numbers. This article describes the fixed point representation of real numbers. The use of fixed point data type is used widely in digital signal processing (DSP) and game applications, where performance is sometimes more important than precision. As we will see later, fixed point arithmetic is much faster than floating point arithmetic.

It All Starts With an Integer

Recall that a binary number:

$$110101_2$$

represents the value:

$$1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0* 2^1 + 1 * 2^0$$

$$= 32 + 16 + 4 + 1$$

$$= 53_{10}$$

Now, if we divide the number 53 by 2, we know the the result should be 26.5. However, how do we represent it if we only had integer representations?

The Binary Point

The key to represent fractional numbers, like 26.5 above, is the concept of *binary point*. A binary point is like the decimal point in a decimal system. It acts as a divider

between the integer and the fractional part of a number.

In a decimal system, a decimal point denotes the position in a numeral that the coefficient should multiply by $10^0 = 1$. For example, in the numeral 26.5, the coefficient 6 has a weight of $10^0 = 1$. But what happen to the 5 to the right of decimal point? We know from our experience, that it carries a weight of 10^{-1}. We know the numeral "26.5" represents the value "twenty six and a half" because

$$2 * 10^1 + 6 * 10^0 + 5 * 10^{-1} = 26.5$$

The very same concept of decimal point can be applied to our binary representation, making a "binary point". As in the decimal system, a binary point represents the coefficient of the term $2^0 = 1$. All digits (or bits) to the left of the binary point carries a weight of 2^0, 2^1, 2^2, and so on. Digits (or bits) on the right of binary point carries a weight of 2^{-1}, 2^{-2}, 2^{-3}, and so on. For example, the number:

$$11010.1_2$$

represents the value:

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
...	1	1	0	1	0	1	0	...

$$= 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0* 2^0 + 1 * 2^{-1}$$

$$= 16 + 8 + 2 + 0.5$$

$$= 26.5$$

Shifting Is The Key

A careful reader should now realize the bit pattern of 53 and 26.5 is exactly the same. The *only* difference, is the position of binary point. In the case of 53_{10}, there is "no" binary point. Alternatively, we can say the binary point is located at the far right, at position 0. (Think in decimal, 53 and 53.0 represents the same number.)

2^5	2^4	2^3	2^2	2^1	2^0	Binary Point	2^{-1}	2^{-2}	2^{-3}
1	1	0	1	0	1	.	0	0	0

In the case of 26.5_{10}, binary point is located one position to the *left* of 53_{10}:

2^5	2^4	2^3	2^2	2^1	2^0	Binary Point	2^{-1}	2^{-2}	2^{-3}
0	1	1	0	1	0	.	1	0	0

Now, recall in class, we discuss shifting an integer to the right by 1 bit position is equivalent to dividing the number by 2. In the case of integer, since we don't have a fractional part, we simply cannot represent digit to the right of a binary point, making this shifting process an *integer division*. However, it is simply a limitation of integer representations of binary number.

In general, mathematically, given a fixed binary point position, shifting the bit pattern of a number to the right by 1 bit *always* divide the number by 2. Similarly, shifting a number to the left by 1 bit multiplies the number by 2.

Fixed Point Number Representation

The shifting process above is the key to understand fixed point number representation. To represent a real number in computers (or any hardware in general), we can define a fixed point number type simply by implicitly *fixing* the binary point to be at some position of a numeral. We will then simply adhere to this implicit convention when we represent numbers.

To define a fixed point type conceptually, all we need are two parameters:

- width of the number representation, and
- binary point position within the number

We will use the notation `fixed<w,b>` for the rest of this article, where `w` denotes the number of bits used as a whole (the Width of a number), and `b` denotes the position of binary point counting from the least significant bit (counting from 0).

For example, `fixed<8,3>` denotes a 8-bit fixed point number, of which 3 right most bits are fractional. Therefore, the bit pattern:

0	0	0	1	0	1	1	0

represents a real number:

$$00010.110_2$$

$$= 1 * 2^{1} + 1 * 2^{-1} + 1 * 2^{-1}$$

$$= 2 + 0.5 + 0.25$$

$$= 2.75$$

Note that on a computer, a bit patter can represents anything. Therefore the same bit pattern, if we "cast" it to another type, such as a `fixed<8,5>` type, will represents the number:

$$000.10110_2$$

$$= 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-4}$$

= 0.5 + 0.125 + 0.0625

= 0.6875

If we treat this bit patter as integer, it represents the number:

10110_2

$= 1 * 2^4 + 1 * 2^2 + 1 * 2^1$

= 16 + 4 + 2

= 22

Negative Numbers

So far we talked about positive numbers, but we do want to represent negative numbers, don't we? How do we represent fixed point negative numbers then?

In computer, we use 2's complement to represent negative numbers. One of the property of 2's complement numbers is that arithmetic operations of either positive of negative numbers are identical. It includes, addition, subtraction, and not surprisingly, shifting. We can divide negative 2's complement numbers by 2 via a simple 1 bit right shift with sign extension as we can do so with positive numbers.

Recall in the beginning of this article we discuss how fixed point numbers are simply a shifted version of an integer (by setting binary point to a non-zero position). Combining with the observation that shift operation applies to 2's complement negative number as well as positive numbers, we can easily see how we can represent negative number in fixed point: Use 2's complement.

As an illustration, below are all the numbers representable with 4-bits 2's complement:

Bit Pattern				Number Represented (n)	n / 2
1	1	1	1	-1	-0.5
1	1	1	0	-2	-1
1	1	0	1	-3	-1.5
1	1	0	0	-4	-2
1	0	1	1	-5	-2.5
1	0	1	0	-6	-3
1	0	0	1	-7	-3.5
1	0	0	0	-8	-4
0	1	1	1	7	3.5
0	1	1	0	6	3
0	1	0	1	5	2.5

0	1	0	0	4	2
0	0	1	1	3	1.5
0	0	1	0	2	1
0	0	0	1	1	0.5
0	0	0	0	0	0

Looking at this table, we can then easily realize we can represent the number -2.5 with bit pattern "1011", *IF* we assume the binary point is at position 1.

Pros and Cons of Fixed Point Number Representation

By now, you should find that fixed point numbers are indeed a close relative to integer representation. The two only differs in the position of binary point. In fact, you might even consider integer representation as a "special case" of fixed point numbers, where the binary point is at position 0. All the arithmetic operations a computer can operate on integer can therefore be applied to fixed point number as well.

Therefore, the benefit of fixed point arithmetic is that they are as straight-forward and efficient as integers arithmetic in computers. We can reuse all the hardware built to for integer arithmetic to perform real numbers arithmetic using fixed point representation. In other word, fixed point arithmetic comes for free on computers.

The disadvantage of fixed point number, is than of course the loss of range and precision when compare with floating point number representations. For example, in a `fixed<8,1>` representation, our fractional part is only precise to a quantum of 0.5. We cannot represent number like 0.75. We can represent 0.75 with `fixed<8,2>`, but then we loose range on the integer part.

Using Fixed Point Number in C

C does not have native "type" for fixed point number. However, due to the nature of fixed point representation, we simply don't need one. Recall all arithmetics on fixed point numbers are the same as integer, we can simply reuse the integer type `int` in C to perform fixed point arithmetic. The position of binary point only matters in cases when we print it on screen or perform arithmetic with different "type" (such as when adding `int` to `fixed<32,6>`).

Conclusion

Fixed point is a simple yet very powerful way to represent fractional numbers in computer. By reusing all integer arithmetic circuits of a computer, fixed point arithmetic is orders of magnitude faster than floating point arithmetic. This is the

reason why it is being used in many game and DSP applications. On the other hand, it lacks the range and precision that floating point number representation offers. You, as a programmer or circuit designer, must do the tradeoff.
