In discussing numeric representation in binary, only *integer* values have been considered, but what about values in the range $(0 < x < 1)$—*fractional values*?

There are two techniques for representing and storing such numbers in binary: *fixed-point* and *floating-point*.

## Fixed-Point

For representing fractional values, *fixed-point* is very closely related to how common *decimal* numbers are written. This makes fixed-point practical and easy to use for humans, *yet the least flexible for computers*. Therefore, it is not actually used within the computer, but it is a wonderful technique to study and learn since it helps understand how *floating-point* is able to represent and store values (seen in the next lecture)

The term *fixed-point* refers to the separator between the integer portion and fractional portion of a number. In the decimal number system this separator is called the "decimal point." In binary, it is called simply called the "point" (although nitpickers would say it is must truly be called the "binary point").

Consider,

$$10.25_{10} = 1010.01_2$$

The form of a fixed-point number is similar to its decimal equivalent. This is reasonable after observing how the unit values are defined on either side of the respective points,

$$10^2\ 10^1\ 10^0\ .\ 10^{-1}\ 10^{-2}\ 10^{-3}$$

$$2^2\ \ 2^1\ \ 2^0\ .\ \ 2^{-1}\ \ 2^{-2}\ \ 2^{-3}$$

As with integer values already discussed, length of the number in decimal is irrelevant, yet the binary number is constrained (or limited) by its word size. Similarly, binary fixed-point numbers must fit within a pre-defined word size: the number of bits before and after the point is important and must be clearly defined.

## Conversion: Decimal to Fixed-Point

Conversion from decimal to fixed-point can be simply accomplished in two (2) steps:

1. Determine the integer part (before point) same as previous conversion method.
2. Determine the fractional part (after point) using a technique somewhat opposite to integer conversion.
3. If negative, perform the 2's comp. on the entire fixed-point number.

ex: (disregard word size limitations for the following two (2) examples)

$10.25_{10} = ?_2$

$10_{10}$

```
    10 / 2 = 5 + rem 0 : 0  (lsb)
     5 / 2 = 2 + rem 1 : 1
     2 / 2 = 1 + rem 0 : 0
     1 / 2 = 0 + rem 1 : 1  (msb)
```

$10_{10} = 1010_2$

$.25_{10}$

```
    0.25 * 2 = 0.5 (record integer) : 0  (msb)
    0.5 * 2  = 1.0 (record integer) : 1  (lsb)
    0.0 * 2  = 0.0
```

$0.25_{10} = .01_2$

$10 + .25_{10} = 10.25_{10}$ ; $1010 + .01_2 = 1010.01_2$

ex:

$21.375_{10} = ?_2$

$21_{10}$

```
21 / 2 = 10 + rem 1 : 1  (lsb)
10 / 2 =  5 + rem 0 : 0
 5 / 2 =  2 + rem 1 : 1
 2 / 2 =  1 + rem 0 : 0
 1 / 2 =  0 + rem 1 : 1  (msb)
```

$21_{10} = 10101_2$

$.375_{10}$

```
0.375 * 2 = 0.75 : 0  (msb)
0.75 * 2  = 1.5  : 1
0.50 * 2  = 1.0  : 1  (lsb)
0.0 * 2   = 0.0
```

$0.375_{10} = .011_2$

$21.375_{10} = 10101.011_2$

It seems straightforward when the fractional part is a nice power of $2^{-n}$, but what if a very simple fraction in decimal ends up being a little more difficult in converting to binary?

ex:

$0.137_{10} = ?_2$

In this example, when must the conversion cease?

The question actual is: what degree of precision (the length of digits after the point) is required?

The determination is once again based on the word size, performing the calculation until all fractional bits are exhausted.

## Storing Fixed-Point

Along with the total word size, fixed-point requires that a specific *point* position be defined to separate the integer part from the fractional. Yet, unlike the 2's comp. sign bit, the point itself is never stored, since only its definition is required.

ex:

```
8-bit word with 4-bit precision:  _ _ _ _ . _ _ _ _
8-bit word with 5-bit precision:  _ _ _ . _ _ _ _ _
```

As with word size, once the *point* position is set for a variable or memory location, all values must conform to it. This defines the range of possible values for the *magnitude* (integer) and the *precision* (fractional) portions of the representation.

Always remember that the *point* is not stored, since the position it occupies is just a convention. Therefore, storing a fixed-point number is the same as with any integer with a defined word size, except that the point position must be remembered by the program storing the number.

> **Note:** Once in binary, all mathematics involving fixed-point work as though the numbers are full integers. The point is irrelevant, and required only when comparing the value with another that is defined by another word size and point position, and 2's comp applies to the entire binary number, not just the integer part.
>
> Also realise that integer numbers of a word size **n** are just fixed-point numbers with precision zero (0).

## Working with Fixed-Point

Although not commonly used in the computer, fixed-point representation has its advantages (especially for humans),

1. Once the precision is defined (i.e., the *point* is set), addition and subtraction (via 2's comp.) works as before (**Note**: For addition on two fixed-point binary numbers, they must share the same word size and precision)

ex: (8-bit word, 2's comp. 3-bit precision)

$$\begin{array}{rr} 10.25 & 01010.010 \\ +\ 6.75_{10} & +\ 00110.110_2 \\ \hline 17.00 & 10001.000_2 \end{array}$$

2. 2's comp. applies as it does with integers (in conversion, recall that after the *flip*, **1** is added to lsb).

ex: (8-bit word, 2's comp. 4-bit precision)

$6.75_{10} = 0110.1100_2$

$-6.75_{10} =$ $\underline{-6.75_{10} = 1001.0100_2}$

1. flip: $1001.0011$
2. add lsb 1: $+\ .0001_2$

$1001.0100_2$

check: $6.75 + (-6.75) = 0.0_{10}$

$0110.1100 + 1001.0100 = 0000.0000_2$ ("extra" 1 is lost)