# Solving the Mystery of Negative Fraction Lengths

Andy Bartlett

This article will take you through a journey that will allow you to understand common fixed-point types better. At the end of this journey, the mystery of negative fraction lengths will have been revealed to you!

In this journey, the weighted-bit-columns concept will give you clarity on how bit encodings map to real-world engineering values. Next, your knowledge of scientific notation will be expanded to knowledge of binary-integer-mantissa-exponent notation. Then, your familiarity with decimal-point notation will be stretched into an understanding of binary-point notation. Your happiness will be elevated with a seemingly intuitive understanding of fraction lengths. Immediately, your confidence will be torn down with the befuddlement of negative fraction lengths! But, you'll be shown you already have the expertise to grasp this. You have experience converting from scientific notation to decimal-point notation and know that irksome adding digits can be necessary. That pesky padding needed just for a style of visual display is the key to the mystery. With the mystery revealed, it becomes clear that the underlying engineering is sound, and that negative fraction lengths are just a way to specify a type that efficiently represents big values.

## Weighted-Bit-Columns Concept

Weighted-bit-columns is a useful concept for understanding binary values and types.

Showing a value using weighted-bit-columns can make it easy to see what is the minimum resources needed to represent the value losslessly.

Weighted-bit-columns can also clarify understanding of data types that have scaling.

For confusion inducing concepts like negative fraction lengths, weighted-bit-columns can make the topic simple to understand.

### Example: 3-bit unsigned integer

Let's review what you likely already know about integers. This review will make it very easy to understand binary-scaled fixed-point types that will be introduced later. For simplicity, let's review a 3-bit type.

A 3-bit unsigned integer has eight distinct representable values. These eight values and their weighted bit column encodings are shown below.

```
dispBinPedanticExtend(0:7)
```

```
 Binary unsigned encoding shown (all values are non-negative.)

              Weighted Bit Columns
               ---  ---  ---
 Real World    2^2  2^1  2^0
    Value       4    2    1
               ---  ---  ---
      0     =   0    0    0
```

```
1     =   0    0    1
2     =   0    1    0
3     =   0    1    1
4     =   1    0    0
5     =   1    0    1
6     =   1    1    0
7     =   1    1    1
```

The right bit column has weighting of 1. The middle bit column has weighting of 2. The right bit column has weighting of 4. Equivalently, the weightings of these columns from right to left are 2^0, 2^1, and 2^2.

For any number of bits, the rightmost column is called the **least significant bit (LSB)** column. The leftmost column is called the **most significant bit (MSB)** column.

Notice that for any pair of neighboring columns, the weighting of the column on the left is always twice as big as the one on the right. This relationship will remain valid when we move beyond integers to arbitrary fixed-point types.

Notice that the real-world value is the sum of the products of each bit and its column weighting. For example, the calculation for the sixth row is the following.

5 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0

This mathematical procedure of mapping bits and weights to finite real-world values will predominantly remain the same for other numeric types. The simple exceptions will be introduced when needed.

The table above shows all eight representable values for this type. The values are sorted from the smallest at the top to the largest at the bottom. Notice that the sequence of representable values is "stepping" or "striding" by the weight of the LSB column. In other words, the absolute difference between any two neighboring representable values is always the weight of the LSB column. This property generalizes from integers to any kind of fixed-point type. But, it does not generalize to floating-point types.

## Example: Counting by eighths

Let's change example 1 to count by eighths (i.e., 1/8) instead of by 1s. To count by eighths, we simply need to multiply the weightings of all columns by 0.125.

```
dispBinPedanticExtend(0.125*(0:7))
```

```
 Binary unsigned encoding shown (all values are non-negative.)

              Weighted Bit Columns
               ----  ----  ----
 Real World    2^-1  2^-2  2^-3
    Value       .5    .25  .125
               ----  ----  ----
      0     =    0     0     0
    0.125   =    0     0     1
    0.25    =    0     1     0
    0.375   =    0     1     1
```

```
 0.5    =    1     0     0
0.625   =    1     0     1
0.75    =    1     1     0
0.875   =    1     1     1
```

Let's examine the sixth row. Notice that it obeys the rule that the real world value is the sum of the products of each bit and its column weight. The only difference from example 1 is that the column weightings are different.

0.625 = 1 * 2^-1 + 0 * 2^-2 + 1 * 2^-3

## Example: Counting by 1/128

Let's change example 1 to count by 1/128. To count by 1/128, we simply need to multiply the weightings of all columns from example 1 by 2^-7.

Or equivalently, we set the weight of the LSB column to 2^-7. And, we have all the other columns obey the rule of the weight doubling when moving one column to the left.

```
dispBinPedanticExtend(2^-7*(0:7))
```

```
Binary unsigned encoding shown (all values are non-negative.)

              Weighted Bit Columns
               ----  ----  ----
 Real World    2^-5  2^-6  2^-7
    Value
               ----  ----  ----
     0     =    0     0     0
 0.0078125 =    0     0     1
 0.015625  =    0     1     0
 0.0234375 =    0     1     1
  0.03125  =    1     0     0
 0.0390625 =    1     0     1
 0.046875  =    1     1     0
 0.0546875 =    1     1     1
```

Let's again examine the sixth row, and notice that it obeys the rule that the real world value is the sum of the products of each bit and its column weighting. Again, the only difference is the weightings.

0.0390625 = 1 * 2^-5 + 0 * 2^-6 + 1 * 2^-7

## Example: Counting by 16

Let's change example 1 to count by 16. To count by 16, we simple need to multiply the weightings of all columns from example 1 by 16.

```
dispBinPedanticExtend(16*(0:7))
```

```
Binary unsigned encoding shown (all values are non-negative.)

              Weighted Bit Columns
               ---  ---  ---
```

```
Real World    2^6  2^5  2^4
   Value       64   32   16
               ---  ---  ---
     0     =   0    0    0
    16     =   0    0    1
    32     =   0    1    0
    48     =   0    1    1
    64     =   1    0    0
    80     =   1    0    1
    96     =   1    1    0
    112    =   1    1    1
```

Let's again examine the sixth row, and notice that it obeys the rule that the real world value is the sum of the products of each bit and its column weighting. Again, the only difference is the weightings.

80 = 1 * 2^6 + 0 * 2^5 + 1 * 2^4

## Scientific (Mantissa-Exponent) Notation

All engineers and scientists have expertise in scientific notation. For example, engineers readily know that one and three quarters million is represented in scientific notation as 1.75e6.

Scientific notation is composed of three keep parts a signed mantissa, an exponent, and the (implied) base number system.

mantissa * base^exponent

Scientific notation is a special case of general mantissa-exponent notations because it restricts the mantissa

1.0 <= abs(mantissa) < 2.0

And, of course, scientific notation uses base 10 number system.

Engineering notation is another mantissa-exponent notation that sets slightly different constraints

1.0 <= abs(mantissa) < 1000.0

exponent is a multiple of 3

In any case, variations mantissa-exponent notation would give an engineer no difficulty. For example, an engineer would easily realize these numbers are equal.

0.00175e9  1.75e6  1750e3  1750000e0

## Binary-Integer-Mantissa-Exponent Notation

In this section, we will define binary-integer-mantissa-exponent (BIME) notation. It is flavor of mantissa-exponent notation.

mantissa * base^exponent

As the name indicates, the mantissa must be an integer value. And binary indicates that the base of the exponent is 2.

Displaying of the mantissa does not have to be binary. It could be shown in binary, decimal, or hex as best suits the situation.

The same value can have multiple binary-integer-mantissa-exponent representations.  For example, these are all equal.

112 * 2^0  56 * 2^1  28 * 2^2  14 * 2^3  7 * 2^4

For fixed-point and integer typed values, the common choice is to select the exponent that corresponds to the data types scaling. For this choice, the integer mantissa will be the integer stored in computer memory and registers. When using your C code debugger, the stored integer (mantissa) value is what will be displayed by the tool.

## BIME for 3-bit unsigned integer

The weighted bit column notation is instructive, but it is also quite verbose. Binary-integer-mantissa-exponent (BIME) notation has the advantage of being more compact.

Let's take another look at the 3-bit unsigned integer example, but use BIME instead.

```
dispBinIntMantExpAlignInType(fi(0:7,0,3))
```

```
Binary unsigned encoding shown (all values are non-negative.)

 Real World   Notation: Integer Mantissa
    Value           and Pow2 Exponent
      0     = 000 * 2^0
      1     = 001 * 2^0
      2     = 010 * 2^0
      3     = 011 * 2^0
      4     = 100 * 2^0
      5     = 101 * 2^0
      6     = 110 * 2^0
      7     = 111 * 2^0
```

For a 3-bit type, this representation is a bit more compact. For larger types such as 64 bits, the compactness difference between weighted-bit-columns and binary-integer-mantissa-exponent are dramatic.

## BIME for 3-bit unsigned counting by 1/8, 1/128, and 16

Let's look at Binary-integer-mantissa-exponent (BIME) notation for the three earlier examples of 3-bit unsigned counting by 1/8, 1/128, and 16, respectively.

```
dispBinIntMantExpAlignInType(fi((0:7)/8,0,3))
```

```
Binary unsigned encoding shown (all values are non-negative.)

 Real World   Notation: Integer Mantissa
    Value           and Pow2 Exponent
```

```
     0     = 000 * 2^-3
 0.125     = 001 * 2^-3
 0.25      = 010 * 2^-3
 0.375     = 011 * 2^-3
  0.5      = 100 * 2^-3
 0.625     = 101 * 2^-3
 0.75      = 110 * 2^-3
 0.875     = 111 * 2^-3
```

```
dispBinIntMantExpAlignInType(fi((0:7)/128,0,3))
```

```
Binary unsigned encoding shown (all values are non-negative.)

 Real World   Notation: Integer Mantissa
    Value           and Pow2 Exponent
      0     = 000 * 2^-7
 0.0078125  = 001 * 2^-7
 0.015625   = 010 * 2^-7
 0.0234375  = 011 * 2^-7
  0.03125   = 100 * 2^-7
 0.0390625  = 101 * 2^-7
 0.046875   = 110 * 2^-7
 0.0546875  = 111 * 2^-7
```

```
dispBinIntMantExpAlignInType(fi(16*(0:7),0,3))
```

```
Binary unsigned encoding shown (all values are non-negative.)

 Real World   Notation: Integer Mantissa
    Value           and Pow2 Exponent
      0     = 000 * 2^4
     16     = 001 * 2^4
     32     = 010 * 2^4
     48     = 011 * 2^4
     64     = 100 * 2^4
     80     = 101 * 2^4
     96     = 110 * 2^4
     112    = 111 * 2^4
```

Let's again examine the sixth row for all four cases.

101 * 2^0

101 * 2^-3

101 * 2^-7

101 * 2^4

Please take note.  Other than being aware that the mantissa is shown in base 2, there is nothing here that would confuse an engineer skilled in scientific notation.

## Binary-point notation

Binary-point notation is like the familiar decimal-point notation. Decimal-point notation uses digits. Binary-point notation uses bits.  Decimal-point notation puts the point between the digit corresponding to weighting 1 and 1/10. Equivalently, the decimal point is between the 10^0 digit column and the 10^-1 digit column. Binary-point notation puts the point between the weighted bit column with weight 1 and 1/2. Equivalently, the binary point is between the 2^0 bit column and the 2^-1 bit column.

## Binary-point for 3-bit unsigned integer counting by 1, 1/2, 1/4, and 1/8

Let's show the binary-point notation for 3-bit unsigned types that count by 1, 1/2, 1/4, and  1/8.

```
dispBinPtExtend(fi(0:7,0,3))
```

```
 Binary unsigned encoding shown (all values are non-negative.)

  Real World   Notation: Binary Point
     Value
       0     = 000.
       1     = 001.
       2     = 010.
       3     = 011.
       4     = 100.
       5     = 101.
       6     = 110.
       7     = 111.
```

```
dispBinPtExtend(fi((0:7)/2,0,3))
```

```
 Binary unsigned encoding shown (all values are non-negative.)

  Real World   Notation: Binary Point
     Value
       0     = 00.0
     0.5     = 00.1
       1     = 01.0
     1.5     = 01.1
       2     = 10.0
     2.5     = 10.1
       3     = 11.0
     3.5     = 11.1
```

```
dispBinPtExtend(fi((0:7)/4,0,3))
```

```
 Binary unsigned encoding shown (all values are non-negative.)

  Real World   Notation: Binary Point
     Value
       0     = 0.00
    0.25     = 0.01
     0.5     = 0.10
    0.75     = 0.11
       1     = 1.00
    1.25     = 1.01
     1.5     = 1.10
```

```
    1.75    = 1.11
```

```
dispBinPtExtend(fi((0:7)/8,0,3))
```

```
 Binary unsigned encoding shown (all values are non-negative.)

  Real World   Notation: Binary Point
     Value
       0     = .000
     0.125   = .001
     0.25    = .010
     0.375   = .011
      0.5    = .100
     0.625   = .101
     0.75    = .110
     0.875   = .111
```

Hopefully, binary point notation feels pretty straight forward at this point. If it is confusing, please compare the the weight-bit-columns, binary-integer-mantissa-exponent, and binary-point examples for 3-bits counting by 1 and counting by 1/8. They all represent the same values, but just use a different style.

## Defining Lengths: Word, Fraction, and Integer

This section will define word-length, fraction-length, and integer-length. For fraction-length and integer-length, both a conceptual definition and a mathematical one will be given. All the definitions are restricted to integer and (simpler) fixed-point data types.

### Definition

WordLength is the total number of bits used (per real scalar)

### Conceptual definition

FractionLength is the number of bits to the right of the binary-point

IntegerLength is the number of bits to the left of the binary-point

### Mathematical definition

FractionLength = - log2( weight of LSB )

IntegerLength = WordLength - FractionLength

Let's apply the definitions to some of the examples previously shown.

| Decimal | Binary-Point | WordLength | IntegerLength | FractionLength | WeightLSB |
|---|---|---|---|---|---|
| 5 | 101. | 3 | 3 | 0 | 2^0 |
| 2.5 | 10.1 | 3 | 2 | 1 | 2^-1 |
| 1.25 | 1.01 | 3 | 1 | 2 | 2^-2 |

| | | | | | |
|---|---|---|---|---|---|
| 0.625 | .101 | 3 | 0 | 3 | 2^-3 |

The conceptual definition should feel intuitive based on the binary-point notation. If you check the mathematical definition, you will find that it is consistent with the conceptual defintion.

## What the heck is a negative FractionLength or IntegerLength?

Based on the intuitive conceptual definition of FractionLength and IntegerLength, it's reasonable to have the impression that those values are always non-negative and that they never exceed the WordLength. But, those impressions are not correct. Depending on the data type, either FractionLength or IntegerLength, can be negative. If one is negative, then the other will be larger than the WordLength.

Earlier, you were asked to take note that these four cases all seemed reasonable.

101 * 2^0

101 * 2^-3

101 * 2^-7

101 * 2^4

Let's now apply the mathematical defintion of FractionLength and IntegerLength to these four case.

| Decimal | BIME | WeightLSB | WordLength | FractionLength | IntegerLength |
|---|---|---|---|---|---|
| 5 | 101 * 2^0 | 2^0 | 3 | 0 | 0 |
| 2.5 | 101 * 2^-3 | 2^-3 | 3 | 3 | 0 |
| 0.0390625 | 101 * 2^-7 | 2^-7 | 3 | 7 | -4 |
| 80 | 101 * 2^4 | 2^4 | 3 | -4 | 7 |

Notice that the last two rows have a negative IntegerLength and FractionLength, respectively. Also, for those rows, the FractionLength and the IntegerLength are greater than the WordLength. It is not intuitive, but it's simple consistent math.

## Padding needed when switching from mantissa-exponent to point notation

When switching from mantissa-exponent notation to decimal-point notation, it is often necessary to pad the representation with digits that do not alter the value.

Consider the following decimal examples.

| mantissa-exponent | decimal-point | padding |
|---|---|---|
| 1.23e10 | 12300000000. | 7 zeros padded on the right |
| 4.56e-6 | .00000456 | 5 zeros padded on the left |

It's cumbersome to add that padding, but it is a familiar and necessary action if the point notation must be used.

Converting from binary-integer-mantissa-exponent notation to binary-point notation can also require padding too.

From the previous examples that had FractionLength and IntegerLength greater than the WordLength, let's show them in binary-point.

```
dispBinPt(0.039062500000000,80)
```

```
Binary unsigned encoding shown (all values are non-negative.)

 Real World   Notation: Binary Point
    Value
 0.0390625 =        .0000101
    80     = 1010000.
```

Notice that the FractionLength shown for the first row is 7. Interestingly, the value 7 agrees with FractionLength computed previously using the mathematical definition. Also, notice that the row has four zero-padding bits on the left. It is no accident that the number of pad bits is the negative of the IntegerLength, 4 == -(-4).

The last row shows a similar pattern. The IntegerLength displayed, 7, agrees with the previously computed mathematical definition. Also, the number of pad bitts on the right, 4, is the negative of the FractionLength, 4 == -(-4).

Based on these examples, we posit the following.

### The conceptual definition holds for the larger of IntegerLength and FractionLength

After a value is converted to binary point notation with padding as needed, the conceptual definition of IntegerLength or FractionLength will hold for the larger of the two values. Moreover, the conceptual definition will hold whenever the value is positive.

If IntegerLength or FractionLength is negative, then the absolute value of that negative equals the number of pad bits needed to convert to a binary point representation.

## What about negative values?

For negative values, the principles all remain the same. Some details need to change, but they are not complicated.

For negative values, the encoding needs to change. For integer and fixed-point, two's complement has been the utterly dominant encoding for decades.

The fundamental change with two's complement is when converting from bits to real-world value. The change is simply to negate the weight of the MSB being used.

Let's illustrate this by example.

```
dispBinPedanticExtend(-5.125,5.75)
```

```
Binary two's complement encoding shown (some values are negative.)

                        Weighted Bit Columns
                ----  ----  ----  ----  ----  ----  ----
 Real World      2^3   2^2   2^1   2^0  2^-1  2^-2  2^-3
    Value          8     4     2     1    .5    .25  .125
                ----  ----  ----  ----  ----  ----  ----
   -5.125   =     1     0     1     0     1     1     1
    5.75    =     0     1     0     1     1     1     0
```

To compute the real world value, the weight of the MSB is negated

$$-5.125 = 1*(-8) + 0*(4) + 1*(2) + 0*(1) + 1*(.5) + 1*(.25) + 1*(.125)$$

$$5.75 = 0*(-8) + 1*(4) + 0*(2) + 1*(1) + 1*(.5) + 1*(.25) + 0*(.125)$$

A consequence of this rule of negating weight of the MSB used is that padding on the left is not always zero. Padding on the left should agree with the previous left most bit. For negative values, padding bits on the left must be 1. For postive values, padding bits on the left must be 0. Padding on the left is also called signed-extension. Padding on the right is always done with zeros.

Let's illustrate this with an example first shown in binary-integer-mantissa-exponent format.

```
dispBinIntMantExpAlign([-3 3]*2^-8)
```

```
Binary two's complement encoding shown (some values are negative.)

  Real World   Notation: Integer Mantissa
    Value            and Pow2 Exponent
 -0.01171875 = 101 * 2^-8
  0.01171875 = 011 * 2^-8
```

Then, let's convert those values to binary point.

```
dispBinPt([-3 3]*2^-8)
```

```
Binary two's complement encoding shown (some values are negative.)

  Real World   Notation: Binary Point
    Value
 -0.01171875 = .11111101
  0.01171875 = .00000011
```

Notice that the five pad bits are different for the signed and unsigned cases. Notice that the FractionLength for the values is eight, which, just like for positive values, is the negative of log2( LSB weight ). Since the WordLength was 3, the IntegerLength is -5. The absolute value of this negative IntegerLength equals the number of pad bits needed. This fact nicely and consistently brings us full circle back to where this paragraph started.

## Negative Fractions Lengths is Good for Big Values

Some engineering applications involve values that are big in the units that the engineering team has selected. Suppose an engineering team has decided that resistance must be represented in Ohms with no scaling qualifier. Assume that the resistance range to be represented goes from 1e5 Ohms to 1e7 Ohms. And assume accuracy greater than 2% is sufficient.

Using 10 bits, these specifications can be achieved.

```
dispBinIntMantExpAlignInType(fi([1e5 1e7],0,10))
```

```
  Binary unsigned encoding shown (all values are non-negative.)

   Real World   Notation: Integer Mantissa
      Value           and Pow2 Exponent
      98304   = 0000000110 * 2^14
     9994240  = 1001100010 * 2^14
```

Notice that the weight of the LSB is 2^14. Using this formula,

Weight of LSB = 2^-FractionLength

the FractionLength is found to be -14. The fact that it is negative is unimportant.

All that matters is that the scaling 2^-FractionLength was big enough to allow 10 bits to represent 1e5 to 1e7 with sufficient accuracy.

To achieve the same range with a non-negative FractionLength would have required the WordLength to increase from 10 bits to 24 bits. This bigger wordlength would more than double the size consumed more resources on the chip, used more power, and slowed down calculations.

The ability to have a negative fraction length allows sensible and efficient scaling to be applied to our engineering problem.

## Floating-Point Uses Super Negative Fraction Lengths and Integer Lengths

Values represented by floating-point type can be displayed in any of the notations presented in this article. Floating-point can represent huge and tiny values. So they would have a big range of exponents in binary-integer-mantissa-exponent notation. But the number of bits in the mantissa is modest. These two facts imply floating-point can have sizeable negative integer lengths and considerable negative fraction lengths. Let's see a few examples.

The fraction length of the following single precision value would be -104.

```
dispBinIntMantExpAlignInType(realmax('single'))
```

```
  Binary unsigned encoding shown (all values are non-negative.)

        Real World         Notation: Integer Mantissa
           Value                 and Pow2 Exponent
   3.4028234663852886e+38 = 01111111111111111111111 * 2^104
```

The integer length of the following value is less than -100.

```
dispBinIntMantExp(realmin('single'))
```

```
Binary unsigned encoding shown (all values are non-negative.)

       Real World         Notation: Integer Mantissa
          Value                 and Pow2 Exponent
 1.1754943508222875e-38 = 1 * 2^-126
```

Clearly, there is nothing wrong with having negative fraction and integer lengths. They just correspond to representing very big or very small values with an efficiently small number of bits.

## Summary

This article has presented topics that support a deeper understanding of fixed-point types.

- weighted-bit-columns concept
- binary-integer-mantissa-exponent notation
- binary-point notation
- utilization of pad bits to convert from exponent notation to point notation

These concepts make it easier to understand what it means to have a negative fraction or integer length and why this provides useful flexibility to support the needs of an engineering application efficiently.

## Resources

The tools demonstrated in the article for display numeric values in various formats are available on GitHub, https://github.com/mathworks/NumericEfficiencyExamples, in the folder DisplayTools.

They require MATLAB® and Fixed-Point Designer™.

At the time of writing, the tools work with releases R2020a and R2019b as well as MATLAB OnLine.

List display functions:

dispBinIntMantExp.m

dispBinIntMantExpAlign.m

dispBinIntMantExpAlignInType.m

dispBinPedanticExtend.m

dispBinPedanticInType.m

dispBinPedanticMin.m

dispBinPt.m

dispBinPtExtend.m

dispBinPtPrefer.m