Fixed Point Representation
And Fractional Math

By Erick L. Oberstar
© 07/17/2004 Oberstar Consulting
Table of Contents

Table of Contents............................................................................................................................ 1
Summary........................................................................................................................................... 2
1. Fixed-Point Representation............................................................................................................. 2
   1.1. Fixed Point Range - Integer Portion .................................................................................. 2
   1.2. Fixed Point Resolution - Fractional Portion ....................................................................... 3
   1.3. Range & Resolution - Putting The Word Together .............................................................. 4
   1.4. Scaling A Floating Point Number To Fixed Point ............................................................. 4
2. Math With Eight Bit Examples...................................................................................................... 5
   2.1. Q1.7 Format ........................................................................................................................ 5
   2.2. Q2.6 Format ........................................................................................................................ 5
   2.3. Multiply - Q1.7xQ2.6=Q3.13 Format .............................................................................. 5
   2.4. Addition - Q1.7+Q2.6=Q2.6 Format ............................................................................... 6
   2.5. An Example Using Real Numbers ..................................................................................... 6
3. Implementation Caveats................................................................................................................... 8
   3.1. Addition ............................................................................................................................. 8
   3.2. Multiplication .................................................................................................................... 8
   3.3. General Caveats & Example C Code ............................................................................... 9
4. References..................................................................................................................................... 9
Summary

In a majority of the commercially available processors on the market today there is no hardware support for floating-point arithmetic due to the cost the extra silicon imposes on a processors total cost. In fact a large portion of processors do not even have hardware support for integer multiplication. This necessitates software emulation for floating point arithmetic and possibly even software emulation for computing integer multiplications. This software overhead can significantly limit the rate at which algorithms can be executed.

By implementing algorithms using fixed-point (integer) mathematics, a significant improvement in execution speed can be observed because of inherent integer math hardware support in a large number of processors, as well as the reduced software complexity for emulated integer multiply and divide. This speed improvement does come at the cost of reduced range and accuracy of the algorithms variables. The purpose of this paper is to investigate the issues relating to algorithm implementation utilizing fixed-point rather than floating-point mathematics.

1. Fixed-Point Representation

To more accurately construct an algorithm, double or single precision floating point data and coefficient values should be used. However there is significant processor overhead required to perform floating-point calculations resulting from the lack of hardware based floating point support. In some cases such as with lower powered embedded processors there is not even compiler support for double precision floating point numbers. Floating point emulation overhead limits the effective iteration rate of an algorithm.

To improve mathematical throughput or increase the execution rate (i.e. increase rate the algorithm could be repetitively run) calculations can be performed using two’s complement signed fixed point representations. Fixed-point representations require the programmer to create a virtual decimal place in between two bit locations for a given length of data (variable type).

For the purposes of this paper the notion of a Q-point for a fixed point number is introduced. This labeling convention is as follows:

\[ Q[QI].[QF] \]

Where \( QI = \# \) of integer bits \& \( QF = \# \) of fractional bits

The number of integer bits (QI) plus the number of fractional (QF) bits yields the total number of bits used to represent the number. Sum QI+QF is know as the word length (WL) and this sum usually corresponds to variable widths supported on a given processor. Typical word lengths would be \{8,16,32\} bits corresponding to \{char, int, long int\} C/C++ variable types. For example: a Q3.5 number would be an 8-bit value with three integer bits and five fractional bits.

1.1. Fixed Point Range - Integer Portion
The range of floating point variable (i.e. its Min to Max range) in an algorithm sets the number of bits (QI) required to represent the integer portion of the number. This relationship for unsigned numbers (positive only) is defined by the equations:

\[ 0 \leq \alpha \leq 2^{QI} \]

\[ QI = \text{ceiling} \left( \log_2 \left( \text{abs} \left( \alpha \right) \right) \right) \]

where \( \alpha \) is the floating point variable.

For example an unsigned (positive) variable \( \alpha = 5.4321 \):

\[ QI = \text{ceiling} \left( \log_2 \left( 5 \right) \right) = \text{ceiling} \left( 2.3219 \right) = 3 \]

\[ 5.4321 \leq 2^3 = 8 \]

\[ \therefore \text{3 bits are required for the integer portion of } \alpha \]

If signed variables must be represented the above equation for QI must be slightly modified because of the need to include a single bit to designate the sign of the number (\( \pm \)). This relationship for signed numbers (\( \pm \alpha \)) is defined by the equations:

\[ -2^{QI-1} \leq \alpha < 2^{QI-1} \]

\[ QI - 1 = \text{ceiling} \left( \log_2 \left( \max \left( \text{abs} \left[ \alpha_{\text{max}}, \alpha_{\text{min}} \right] \right) \right) \right) \]

where \( \alpha \) is the floating point variable.

For example an signed (\( \pm \)) variable \( \alpha = -5.4321 \):

\[ QI - 1 = \text{ceiling} \left( \log_2 \left( \max \left( \text{abs} \left[ -5.4321 \right] \right) \right) \right) = \text{ceiling} \left( \log_2 \left( 5.4321 \right) \right) \]

\[ QI = \text{ceiling} \left( \log_2 \left( 5.4321 \right) \right) + 1 = \text{ceiling} \left( 2.3219 \right) + 1 = 3 + 1 \]

\[ QI = 4 \]

1.2. Fixed Point Resolution - Fractional Portion

The resolution for a fixed point variable is set by the remaining fractional bits (QF) for a given word length (WL) for the variable. For a given word length (WL) and dynamic range of a variable the resolution is limited. If a higher resolution is needed for a given range then the WL of the variable must be increased to provide this resolution.

The resolution \( \varepsilon \), of a fixed point number is governed by the equation:

\[ \varepsilon = \frac{1}{2^{QF}} \]
Therefore the number of fractional bits (QF) required for a particular resolution are defined by the equation:

\[ QF = \log_2 \left( \frac{1}{\varepsilon} \right) \]

However since QF is integer values only, the ceiling of the logarithm result:

\[ QF = \text{ceiling} \left( \log_2 \left( \frac{1}{\varepsilon} \right) \right) \]

For example an signed (±) variable \( \alpha = -5.4321, \varepsilon \leq 0.0001 \)

\[ QF = \text{ceiling} \left( \log_2 \left( \frac{1}{0.0001} \right) \right) \]

\[ QF = \text{ceiling} \left( \log_2 (10000) \right) = \text{ceiling} \left( 13.288 \right) = 14 \]

### 1.3. Range & Resolution - Putting The Word Together

The integer and fractional bits are combined together into a standard WL. The full range and resolution for a fixed point value are set by the integer and fractional parts of the number for a fixed WL. The combined range and resolution for a fixed point number is defined by:

\[-2^{QI-1} \leq \alpha < 2^{QI-1} - 2^{-QF}\]

Where: WL = QI+QF with the sign bit lumped in with QI.

Based on the above example of \( \alpha = -5.4321 \) the required WL=QI+QF = 4+14 = 18 (bits) to guarantee both the range and resolution in the above examples. Note this is not a standard WL on most processors. This exemplifies that there is a tradeoff between range and resolution when implemented on standard WL processors.

### 1.4. Scaling A Floating Point Number To Fixed Point

Once an appropriate fixed point format has been calculated based on WL, range, and resolution of a floating point value, the fixed point approximation for the floating point number can be calculated. This relationship is governed by the equation:

\[ FxdPt = \text{floor} \left( FltPt \times 2^{QF} \right) \]

Since the fixed point representation of a floating point number can only have integer values the floor of the scaled floating point number must be used.

From the example above, an signed (±) variable \( \alpha = -5.4321, \varepsilon \leq 0.0001, \text{QF} = 14 \).

The integer (Fixed Point) representation for \( \alpha \) is:

\[ \alpha_{FxdPt} = \text{floor} \left( \alpha_{FltPt} \times 2^{14} \right) = \text{floor} \left( -5.4321 \times 2^{14} \right) \]

\[ \alpha_{FxdPt} = \text{floor} \left( -5.4321 \times 16384 \right) = \text{floor} \left( -88999.5264 \right) = -88999 \]
Note that \(2^{16} < |\alpha_{\text{FadPt}}| = 88999 < 2^{17}\), this necessitates that 17-bits be used for the magnitude plus the sign bit yields the previously calculated WL of 18-bits in a Q4.14 to represent \(\alpha = -5.4321\), with an \(\varepsilon \leq 0.0001\). Note that the closest larger standard data type that can accommodate this value is a 32-bit data type. Since only four integer bits are required the remaining 28 bits of the 32-bit data can be used for fractional content (QF) which would yield \(\varepsilon = \frac{1}{2^{QF}} = \frac{1}{2^{28}} \cong 3.725290298461914\times10^{-9}\).

2. Math With Eight Bit Examples

Consider a simple example with two variables, one variable (\(\alpha\)) ranging from \(\pm 1\) and the other variable (\(\beta\)) ranging from \(\pm 2\) with both as much resolution as possible. For an 8-bit WL, this necessitates Q1.7 and Q2.6 fixed-point representations for \(\alpha\) and \(\beta\) respectively. An 8-bit example was chosen because the most common WL in low cost microcontrollers is typically 8-bits.

2.1. Q1.7 Format

Q1.7 numbers can represent fixed-point numbers ranging from –1 to 0.9921875 (-1 to 1 - 1/128) in increments 0.0078125 (1/128). The 8-bit Q1.7 number bit weighting is shown below. The decimal place is between bits 6 and 7 using the convention the most right bit is the 0-bit. The variable \(\alpha\) is in a Q1.7 format.

\[
\begin{array}{cccccccc}
| s & x & x & x & x & x & x & x |\\
| -1 & 1/2 & 1/4 & 1/8 & 1/16 & 1/32 & 1/64 & 1/128 |
\end{array}
\]

2.2. Q2.6 Format

Eight bit Q2.6 numbers can represent fixed-point numbers ranging from –2 to 1.984375 (-2 to 2 - 1/64) in increments 0.015625 (1/64). The Q2.6 representation bit weighting is shown below. The decimal place is between bits 5 and 6 using the convention the most right bit is the 0-bit. The variable \(\beta\) is in a Q2.6 format.

\[
\begin{array}{cccccccc}
| s & x & x & x & x & x & x & x |\\
| -2 & 1 & 1/2 & 1/4 & 1/8 & 1/16 & 1/32 & 1/64 |
\end{array}
\]

2.3. Multiply - Q1.7xQ2.6=Q3.13 Format

When performing an integer multiplication the product is 2xWL if both the multiplier and multiplicand are WL long. If the integer multiplication is on fixed point variables, the number of integer and fractional bits in the product is the sum of the corresponding multiplier and multiplicand Q-points as described by the following equations.

\[
\begin{align*}
Q_{\text{Product}}^I &= Q_{\text{Multiplier}}^I + Q_{\text{Multiplicand}}^I \\
Q_{\text{Product}}^F &= Q_{\text{Multiplicand}}^F + Q_{\text{Multiplier}}^F
\end{align*}
\]
When a Q1.7 and Q2.6 number are multiplied (both are signed 8-bit numbers) the result is a 16-bit Q3.13 number. Q3.13 numbers range from –4 to 3.9998779296875 (–4 to 4 – 1/8192) in increments of 0.0001220703125 (1/8192). The Q3.13 representation bit weighting is shown below.

```
|   s|x|x.|   x|   x|x|x|x|x|x|x|x|x|x|x|x|         x|
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| -4|2|1.|1/2|1/4|x|x|x|x|x|x|x|x|x|x|1/8192|
```

The 16-bit Q3.13 number can be scaled back to an 8-bit representation for subsequent use in an algorithm. The 8-bit result needs to be a Q3.5 format to maintain the range of the result of the multiplication at the price of losing the precision for the lowest 8 fractional bits. These Q3.5 bits are extracted by shifting the 16-bit Q3.13 number right eight bits and selecting only the low byte of the 16-bit value. The resulting 8-bit Q3.5 number inside the 16-bit result is shown below.

```
|   s|x|x.|   x|   x|x|x|x|x|x|x|x|x|x|x|x|         x|
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| -4|2|1.|1/2|1/4|x|x|x|x|x|x|x|x|x|x|1/8192|
```

8-bit Q3.5 Number

### 2.4. Addition - Q1.7+Q2.6=Q2.6 Format

Addition is a pure integer type of operation but care must be taken to align the fixed-point decimal places and attention must be paid to handling overflow of the addition.

\[
\begin{array}{c}
| s|x|x|x|x|x|x| \\
+ | s|x|x|x|x|x|x| \\
\end{array}
\]

Right Shift & sign extend the Q1.7 to align the decimal place.

\[
\begin{array}{c}
| s|x|x|x|x|x|x| \\
+ | s|x|x|x|x|x|x| \\
+ c | s|x|x|x|x|x|x| \\
\end{array}
\]

Perform the signed addition and check the carry bit (c) to see if you overflowed the WL (8-bits in this case). Another option is to accumulate the result of the destination into a 2xWL variable and check to see if it exceeds the maximum WL value you expect. For example with the addition of two eight bit values into a sixteen bit result and checking if the sixteen bit result is in the range \(-2^7 \leq \text{result} < 2^7 - 1\), and if not saturating positive or negative.

### 2.5. An Example Using Real Numbers

Using an 8-bit WL with \(|\alpha| \leq 1.8\), \(|\beta| < 1\), and \(|\chi| \leq 2.8\) as range limits, with \(\alpha = 1.667\), \(\beta = -0.75\), and \(\chi = 2.6\), with maximal resolution on each variable, compute:
\[(\alpha \times \beta) + \chi = (1.667 \times -0.75) + 2.6 = -1.25025 + 2.6 = 1.34975\]

\[QI_{\alpha} = \text{ceiling} \left( \log_2 \left( \max (\alpha) \right) \right) + 1 = \text{ceiling} \left( \log_2 (1.8) \right) + 1 = 2\]
\[QF_{\alpha} = WL - QI_{\alpha} = 8 - 2 = 6\]
\[\text{so: } \epsilon_{\alpha} = \frac{1}{2^{QF_{\alpha}}} = \frac{1}{2^6} = \frac{1}{64} = 0.015625\]
\[-116_{10} (Q2.6) \leq \alpha_{\text{Fxp}} \leq 115_{10} (Q2.6)\]
\[\alpha_{\text{Fxp}} = 1.667 \times 2^6 = 106_{10} (Q2.6)\]

\[QI_{\beta} = \text{ceiling} \left( \log_2 \left( \max (\beta) \right) \right) + 1 = \text{ceiling} \left( \log_2 (1) \right) + 1 = 1\]
\[QF_{\beta} = WL - QI_{\beta} = 8 - 1 = 7\]
\[\text{so: } \epsilon_{\beta} = \frac{1}{2^{QF_{\beta}}} = \frac{1}{2^7} = \frac{1}{128} = 0.0078125\]
\[\beta_{\text{Fxp}} = -0.75 \times 2^7 = -96_{10} (Q1.7)\]

\[QI_{\chi} = \text{ceiling} \left( \log_2 \left( \max (\chi) \right) \right) + 1 = \text{ceiling} \left( \log_2 (3) \right) + 1 = 3\]
\[QF_{\chi} = WL - QI_{\chi} = 8 - 3 = 5\]
\[\text{so: } \epsilon_{\chi} = \frac{1}{2^{QF_{\chi}}} = \frac{1}{2^5} = \frac{1}{32} = 0.03125\]
\[-90_{10} (Q3.5) \leq \chi_{\text{Fxp}} \leq 89_{10} (Q3.5)\]
\[\chi_{\text{Fxp}} = 2.6 \times 2^5 = 83_{10} (Q3.5)\]

\[(\alpha \times \beta) + \chi = (1.667 \times -0.75) + 2.6 = -1.25025 + 2.6 = 1.34975\]

Computing the product term \((\alpha \times \beta)\),
\[
\alpha_{\text{Fxp}} \times \beta_{\text{Fxp}} = 106_{10} (Q2.6) \times -96_{10} (Q1.7) = -10176_{10} (Q3.13) = -1.2421875
\]

Notice that the fixed point approximation of the product term has an error of:
\[-1.25025 - (-1.2421875) = -0.0080625\]

Also notice that the range of the product term is essentially the range of \(\alpha\) but in a 16-bit format.

Before computing the sum \((\alpha \times \beta) + \chi\), the 16-bit product term and \(\chi\) need to have the decimal places aligned. The decimal places can be aligned by right shifting the signed 16-bit product term 8-bits or by sign extending \(\chi\) to 16-bits and left shifting it 8-bits. It is not uncommon to
need to scale a 2xWL result back to a WL result for subsequent computations or system outputs such as a D/A or PWM.

So scaling the product term to align the decimal places:

\[ \alpha_{\text{FxdPt}} \times \beta_{\text{FxdPt}} = -10176_{10} (Q3.13) \gg 8 = \frac{-10176_{10}}{2^8} = -39_{10} (Q3.5) \]

Adding the scaled product term and \( \chi \)

\[ (\alpha_{\text{FxdPt}} \times \beta_{\text{FxdPt}}) + \chi_{\text{FxdPt}} = -39_{10} (Q3.5) + 83_{10} (Q3.5) = 44_{10} (Q3.5) \]

The answer is:

\[ (\alpha_{\text{FxdPt}} \times \beta_{\text{FxdPt}}) + \chi_{\text{FxdPt}} = 44_{10} (Q3.5) = 1.375 \]

Notice the error inherent between the floating point calculation and the fixed point calculation shown below:

\[ ((\alpha \times \beta) + \chi) - ((\alpha_{\text{FxdPt}} \times \beta_{\text{FxdPt}}) + \chi_{\text{FxdPt}}) = 1.34975 - 1.375 = -0.02525 \]

3. Implementation Caveats

A critical detail when implementing fixed point algorithms is that the variables must be a signed data type. I.e. use variable types: signed char, signed int, and signed long int, as opposed to unsigned char, unsigned int, and unsigned long int. This is important because of the need to preserve a variables sign when performing the inherent scaling via left or right shift operations for fixed point addition operations and sign extension for typecasting required for multiplication operations.

3.1. Addition

When an addition operation needs to be performed one of the variables may need to be shifted to align the Q-points (decimal place) of the variables before the addition. The variable with the larger number of fractional bits (larger QF) will need to be right shifted \( QF_{\text{Larger}} - QF_{\text{Smaller}} \) bits to effectively move its decimal place left to align the Q-points.

3.2. Multiplication

When a product of fixed point numbers is calculated at least one of the values must be sign extended to 2xWL to do the multiplication correctly. If this is not done only the low \( \frac{1}{2} \) of 2xWL result will be returned. This is done by type casting the multiplicand (or multiplier) to a 2xWL variable type. This will sign extend the multiplicand (or multiplier) to 2xWL to properly compute the product.
3.3. General Caveats & Example C Code

A point of caution: some compilers contain switches to make “char” data types unsigned by default as well as to allow automatic promotion of “char” types to “int”. The following C code example could be used to implement the example earlier in this document.

```c
// Variable Declarations
signed char alpha, beta, gamma; // Alpha - Q1.7, Beta - Q2.6, Gamma - Q3.5
signed int prod; // 16-bit multiply product accumulator
signed int sum; // 16-bit summation accumulator
signed char result; // 8-bit result register

// Functional Code Block
prod = (int) alpha*beta; // 8x8 to 16 multiply – Note that the type cast to integer is required otherwise the accum will only have the low 8-bits of the multiply
sum = ((signed char)(prod >>8))+gamma; // align the Qpts, cast to WL and add them
if (sum>127)  // Positive saturation point for signed 8-bit
    result = 127; // Saturate positive
else if (sum<-128) // Negative saturation point for signed 8-bit
    result = -128; // Saturate negative
else
    result = (signed char)sum; // If not saturated just use the low 8-bits
```

4. References


