Fixed-Point Representation & Fractional Math

By Erick L. Oberstar

Revision 1.2

Released August 30, 2007

Table of Contents

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 processor's 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.

The Q[QI].[QF] format fixed-point number format analyzed in this paper is broken down in subsequent sections into integer and fractional content for the purpose of study and understanding. The separate sections on integer and fractional content are subsequently combined to provide an overall understanding of the nature of Q[QI].[QF] format fixed-point numbers.

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 overhead limits the effective iteration rate of an algorithm.

To improve mathematical throughput or increase the execution rate (i.e. increase the 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 commonly implemented in compilers for microcontrollers or DSPs.

For example: a Q3.5 number would be an 8-bit value with three integer bits and five fractional bits. For signed integer variable types we will include the sign bit in QI as it does have integer weight albeit negative in sign. WL varies over processors and integer type names can infer different word lengths in various tool chains (i.e some compilers treat **int** as 16-bit, some as 32-bit) [ISO/IEC 9899:TC2]. Therefore, for the purpose of this paper the previously referenced word lengths / type names are implied and used.

The Q[QI].[QF] format fixed-point number format is broken down in subsequent sections into integer and fractional content for the purpose of study and understanding. The separate sections on integer and fractional content are subsequently combined to provide an overall understanding of the nature of Q[QI].[QF] format fixed-point numbers.

1.1. Fixed-point Range - Integer Portion

To represent a floating-point number in fixed-point a floating-point number needs to be viewed as two distinct parts, the integer content, and the fractional content. The integer range of a 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. Keep in mind that QI itself can only hold integer values because of the binary nature of a bit – it exists or doesn't.

There are two different methods of computing the number of integer bits required (QI) for each type of number, unsigned and signed.

1.1.1. Fixed-point Range for Unsigned Integers

This relationship for unsigned numbers (positive only) is defined by the minimum and maximum of any QI-bit number shown in the following equation:

$$\boxed{0 \le \alpha \le \left(2^{QI} - 1\right)}$$

Equation 1

Method 1:
Solving Equation 1 for the required number of bits QI:

$$QI \ge ceiling\left(\log_2\left(\alpha + 1\right)\right)$$

$$\boxed{QI = ceiling\left(\log_2\left(\alpha + 1\right)\right)}$$

Equation 2

where α is the floating-point variable being ranged & *ceiling* rounds towards +∞.
Note: The $\log_2()$ value in Equation 2 can never be negative which implies that QI is always ≥ 1. The benefit of QI < 1 is addressed later in this paper.

For example an unsigned (positive only) variable α = 5.4321:

$$\boxed{QI = ceiling\left(\log_2\left(5.4321 + 1\right)\right) = ceiling\left(2.6835\right) = 3}$$

Example 1

∴ 3 bits are required for the integer portion of α
As sanity check, verify α less than the maximum 3-bit unsigned number.
$5.4321 \le 2^3 - 1 = 7$ - Yes!

Method 2:

Although the previous method above (Method 1) is one possible way of computing QI for unsigned values, there is another way that is arguably better, especially when dealing with numbers that could be much smaller than |1| that must be implemented on standard variable types.

Taking our initial bounding inequality Equation 1:

$$\boxed{0 \le \alpha \le \left(2^{QI} - 1\right)}$$

The inequality can be rewritten:

$$\boxed{0 \le \alpha < 2^{QI}}$$

Equation 3

Note: The upper boundary conditions changes from ≤ to < and the boundary value changes to one integer count higher that the maximum QI-bit unsigned number.

Solving Equation 3 for QI for the constraint:

$$\begin{aligned} &\alpha < 2^{QI} \\ &\log_2\left(\alpha\right) < QI \end{aligned}$$

$$\boxed{QI > \left(\log_2\left(\alpha\right)\right)}$$

Equation 4

Knowing that QI is an integer number of bits we can create an equation to compute a QI that satisfies constraint Equation 4 by adding 1 and truncating the result (rounding tward -∞). An equation for the required number of integer bits can be generalized for this method:

$$\boxed{QI = floor\left(\log_2\left(\alpha\right)+1\right)}$$

Equation 5

where α is the floating-point variable being ranged & *floor* rounds towards -∞.
Note: The $\log_2()$ value in Equation 5 can be negative which implies that QI can be negative. The benefit of QI < 1 is addressed later in this paper.

For example if:

$$\boxed{\begin{array}{l} \alpha = 2 \\ QI > \log_2\left(\alpha\right) = \log_2\left(2\right) = 1 \\ QI > 1 \\ QI = floor\left(\log_2\left(2\right)+1\right) = 1+1 = 2 \\ \therefore QI = 2 \end{array}}$$

Example 2

As sanity check, verify α is bounded by the minimum and maximum values of a 2-bit unsigned number.

$$0 \le \alpha < 2^{QI}$$
$$0 \le \left[0,2\right] < 2^2$$
$$0 \le \left[0,2\right] < 4 \text{ - Yes!}$$

1.1.2. Fixed-point Range for Signed Integers

If signed variables must be represented, the previous solution for QI changes because of the range limits of a signed integer number types.

This relationship for the integer content of signed numbers (±α) is defined is defined by the minimum and maximum values that a signed QI-bit integer number type can hold. This is number shown in the following equation:

$$\boxed{\left(-2^{QI-1}\right) \le \alpha \le \left(2^{QI-1}-1\right)}$$

Equation 6

Remember there is an asymmetry about zero for signed integer variable types: (i.e. a signed 8-bit value ranges from +127 to -128). This asymmetry yields two possible methods for computing the number of integer bits:

Method 3:

Solving for QI for the negative constraint of Equation 6 (i.e. when α is *negative*):

$$\begin{aligned} &\left(-2^{QI-1}\right) \le \alpha \\ &2^{QI-1} \ge -\alpha \\ &QI-1 \ge \log_2\left(-\alpha\right) \\ &\boxed{QI \ge \log_2\left(-\alpha\right)+1} \end{aligned}$$

Equation 7

Solving for QI for the Positive constraint Equation 6 (i.e. when α is *positive*):

$$\begin{aligned} &\alpha \le \left(2^{QI-1}-1\right) \\ &\alpha+1 \le 2^{QI-1} \\ &\log_2\left(\alpha+1\right) \le QI-1 \\ &\boxed{QI \ge \left(\log_2\left(\alpha+1\right)\right)+1} \end{aligned}$$

Equation 8

For example if:

$$\boxed{\begin{aligned} &\alpha_{\min} = -2, \quad \alpha_{\max} = 2 \\ &QI\big|_{\alpha_{\min}} \ge \log_2\left(-\alpha_{\min}\right)+1 = \log_2\left(2\right)+1 = 2 \\ &QI\big|_{\alpha_{\max}} \ge \log_2\left(\alpha_{\max}+1\right)+1 = \log_2\left(3\right)+1 = 2.5850 \end{aligned}}$$

Example 3

The positive constraint is the tighter of the two constraints due to this asymmetry of signed integer types about zero. It is not uncommon for users/programmers to define variable magnitude constraints that are symmetric about zero (for example: $-3 \le \alpha \le 3$). The computation for the required number of integer bits can be generalized for this method:

$$\boxed{QI = ceiling\left(\log_2\left(\max\left(abs\left[\alpha_{\max},\alpha_{\min}\right]\right)+1\right)\right)+1}$$

Equation 9

where α is the floating-point variable being ranged & *ceiling* rounds towards +∞.

Note: The $\log_2()$ value in Equation 9 can never be negative which implies that QI is always ≥ 1. The benefit of QI < 1 is addressed later in this paper.

For example to compute QI for a signed (±) variable $-5.4321 \le \alpha \le 5.4321$:

$$\boxed{\begin{gathered}QI-1 = ceiling\left(\log_2\left(\max\left(abs\left[-5.4321,5.4321\right]\right)+1\right)\right)+1 = ceiling\left(\log_2\left(6.4321\right)\right)+1\\ QI = ceiling\left(\log_2\left(6.4321\right)\right)+1 = ceiling\left(2.6853\right)+1 = 3+1\\ QI = 4\end{gathered}}$$

Example 4

As sanity check, verify α is bounded by the minimum and maximum values of a 4-bit signed number.

$$\left(-2^{QI-1}\right) \le \alpha \le \left(2^{QI-1}-1\right)$$

$$\left(-2^{4-1}\right) \le \left[-5.4321,5.4321\right] \le \left(2^{4-1}-1\right)$$

$$\left(-2^{3}\right) \le \left[-5.4321,5.4321\right] \le \left(2^{3}-1\right)$$

$$-8 \le \left[-5.4321,5.4321\right] \le 7 \text{ - Yes!}$$

Method 4:

Although Method 3 is one possible way of computing QI for signed values, there is another way that is arguably better, especially when dealing with numbers that could be much smaller than |1| that must be implemented on standard variable types.

Taking our initial bounding inequality Equation 6:

$$\left(-2^{QI-1}\right) \le \alpha \le \left(2^{QI-1}-1\right)$$

The inequality can be rewritten:

$$\boxed{-2^{QI-1} \le \alpha < 2^{QI-1}}$$

Equation 10

Note: The upper boundary conditions changes from $\le$ to < and the boundary value changes to one integer count higher that the maximum QI-bit signed number.

Solving Equation 10 for QI for the negative constraint (i.e. when α is *negative*):

$$\begin{aligned}&\left(-2^{QI-1}\right) \le \alpha\\&2^{QI-1} \ge -\alpha\\&QI-1 \ge \log_2\left(-\alpha\right)\\&QI \ge \log_2\left(-\alpha\right)+1\end{aligned}$$

Solving Equation 10 for QI for the Positive constraint (i.e. when α is *positive*):

$$\begin{aligned}&\alpha < 2^{QI-1}\\&\log_2\left(\alpha\right) < QI-1\\&QI > \left(\log_2\left(\alpha\right)\right)+1\end{aligned}$$

The positive constraint is the tighter of the two constraints due to this asymmetry of signed integer types about zero. It is not uncommon for users/programmers to define variable magnitude constraints that are almost symmetric about zero (for example: $-4 \le \alpha < 4$). The constraint for the required number of integer bits can be generalized for this method:

$$\boxed{QI > \log_2\left(\max\left(abs\left[\alpha_{\max},\alpha_{\min}\right]\right)\right)+1}$$

Equation 11

Knowing that QI is an integer number of bits we can create an equation to compute a QI that satisfies constraint Equation 4 by adding 1 and truncating the result (rounding tward -∞). An equation for the required number of integer bits can be generalized for this method:

$$QI = floor\left(\log_2\left(\max\left(abs\left[\alpha_{\max},\alpha_{\min}\right]\right)\right)+1+1\right)$$

$$\boxed{QI = floor\left(\log_2\left(\max\left(abs\left[\alpha_{\max},\alpha_{\min}\right]\right)\right)+2\right)}$$

Equation 12

Note: The $\log_2()$ value in Equation 12 can be negative which implies that QI can be negative. The benefit of QI < 1 is addressed later in this paper.

For example if:

$$\boxed{\begin{array}{l} \alpha_{\min} = -2, \quad \alpha_{\max} = 2 \\ QI\big|_{\alpha_{\min}} \geq \log_2(-\alpha_{\min}) + 1 = \log_2(2) + 1 = 2 \\ QI\big|_{\alpha_{\max}} > \log_2(\alpha_{\max}) + 1 = \log_2(2) + 1 = 2 \\ QI\big|_{\alpha_{\min}} \geq 2, QI\big|_{\alpha_{\max}} > 2 \\ \therefore QI = 3 \end{array}}$$

Example 5

As sanity check, verify α is bounded by the minimum and maximum values of a 3-bit signed number.

$$-2^{QI-1} \leq \alpha < 2^{QI-1}$$
$$-2^{3-1} \leq [-2,2] < 2^{3-1}$$
$$-4 \leq [-2,2] < 4 \text{ - Yes!}$$

1.1.3. Fixed-point Range Comments/Conclusions

Method 1 and Method 3 exactly constrain QI based on the exact numerical range of the input parameter's (α's) integer content for unsigned and signed number types respectively. As a result the minimum number of integer content bits QI, is 1. Method 2 and Method 4 constrain QI in such a way that QI can be negative for unsigned and signed number types respectively. Negative values for QI provide benefit by allowing extended resolution (QF) for chosen WL which will be discussed later in this paper.

For Methods 2 and 4 the QI constraint equation requires QI ">" a value that is computed. Since QI itself can only have integer values, the next largest integer value must be chosen. This issue is apparent when $\log_2(\alpha)$ in either Equation 5 or Equation 11 results in exact integer value.

1.2. Fixed-point Resolution - Fractional Portion

The resolution for a fixed-point variable is set by the number of fractional bits (QF) used in the fixed-point variable. The resolution ε, of a fixed-point number is governed by the equation:

$$\boxed{\varepsilon = \frac{1}{2^{QF}}}$$

Equation 13

Therefore the number of fractional bits (QF) required for a particular resolution are defined by the equation:

$$\boxed{QF = \log_2\left(\frac{1}{\varepsilon}\right)}$$

Equation 14

However since QF is integer values only (i.e. we can only use integer numbers of bits), the *ceiling* of the logarithm is used:

$$\boxed{QF = ceiling\left(\log_2\left(\frac{1}{\varepsilon}\right)\right)}$$

Equation 15

For example an signed (±) variable α = -5.4321, $\varepsilon \le 0.0001$

$$\boxed{\begin{gathered} QF = ceiling\left(\log_2\left(\frac{1}{0.0001}\right)\right) \\ QF = ceiling\left(\log_2(10000)\right) = ceiling(13.288) = 14 \end{gathered}}$$

Example 6

It is not uncommon for users/programmers to find number of integer bits required (QI) and live with the resolution provided by the left over bits for a given word length (WL) used for the variable. For a given word length (WL) and dynamic range (QI) 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.

1.3. Range & Resolution - Putting Them Together

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 an unsigned fixed-point number is defined by:

$$\boxed{0 \le \alpha \le \left(2^{QI} - 1\right)\Big|_{\varepsilon = 2^{-QF}}}$$

Equation 16

The combined range and resolution for a signed fixed-point number is defined by:

$$\boxed{-2^{QI-1} \leq \alpha \leq \left(2^{QI-1} - 2^{-QF}\right)\Big|_{\varepsilon=2^{-QF}}}$$

Equation 17

Where: WL_{Required} = QI+QF with the sign bit lumped in with QI.

The integer and fractional bits are combined together into and used to determine a standard WL that is large enough to hold all the integer and factional bits. This implies:

$$\boxed{WL_{\text{Required}} \geq QI + QF}$$

Equation 18

For example for a Q3.5 number, an 8-bit integer variable type must be used to contain the number although larger variable types (i.e. 16-bit, 32-bit, etc…) can also contain it.

As another example for a Q4.14 number 18-bits are required to represent the number. Since 18-bits are not a standard word length in most programming languages or processors, the next longest word length variable needs to be used to contain the result. A 32-bit number would be the smallest standard WL in C/C++ that could contain a Q4.14 number. If a 32-bit number is used for the Q4.14 number, there are an additional 14-bits that are available to extend the range or resolution of the number (i.e. increase QI and/or QF). This exemplifies that there is a tradeoff between range and resolution when implemented with standard WL variables.

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:

$$\boxed{FxdPt = \left(FltPt \times 2^{QF}\right)\Big|_{\text{Rounded twards 0}}}$$

Equation 19

Since the fixed-point representation of a floating-point number can only have integer values the integer portion or *truncation* of the scaled floating-point number must be used. This means round towards zero. For example -1.4 becomes -1 and 1.4 becomes 1.

From the example above, a signed (±) variable α = -5.4321, $\varepsilon \leq 0.0001$, QF = 14.
The integer (Fixed-point) representation for α is:

$$\boxed{\begin{aligned} \alpha_{FxdPt} &= \left(\alpha_{FltPt} \times 2^{QF_\alpha}\right)\Big|_{\text{Rounded twards 0}} = \left(-5.4321 \times 2^{14}\right)\Big|_{\text{Rounded twards 0}} \\ \alpha_{FxdPt} &= \left(-5.4321 \times 16384\right)\Big|_{\text{Rounded twards 0}} = \left(\text{-}88999.5264\right)\Big|_{\text{Rounded twards 0}} = -88999 \end{aligned}}$$

Example 7

Note that $2^{16} < |\alpha_{FxdPt}| = 88999 < 2^{17}$, 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 α = -5.4321, with an $\varepsilon \le 0.0001$. 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 $\varepsilon = \frac{1}{2^{QF}} = \frac{1}{2^{28}} \cong 3.725290298461914\text{e-}009$.

2. Math With Eight Bit Examples

Consider a simple example with two variables, one variable (α) ranging from ~±1 (–1 to 0.9921875) and the other variable (β) ranging from ~±2 (–2 to 1.984375) with both as much resolution as possible. For an 8-bit WL, this necessitates Q1.7 and Q2.6 fixed-point representations for α and β 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 in increments 0.0078125 (-1 to 1 - 1/128). The 8-bit Q1.7 number bit weighting is shown below. The decimal place is between bits 6 and 7. The variable α is in a Q1.7 format.

s.	x	x	x	x	x	x	x
-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128

2.2. Q2.6 Format

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

s	x.	x	x	x	x	x	x
-2	1	1/2	1/4	1/8	1/16	1/32	1/64

2.3. 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.

$$\begin{array}{r} |\text{s}|x.|x|x|x|x|x|x| \\ + \underline{|\text{s}|x|x.|x|x|x|x|x|} \end{array}$$

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

$$\begin{array}{rl} & |s|s|x.|x|x|x|x|x| \\ + & |\mathrm{s}|x|x.|x|x|x|x|x| \\ \hline c & |\mathrm{s}|x|x.|x|x|x|x|x| \end{array}$$

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 $-2^7 \le result < 2^7 - 1$, and if not saturating positive or negative.

2.4. 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:

$$\boxed{QI_{\text{Product}} = QI_{\text{Multiplicand}} + QI_{\text{Multiplier}}}$$

Equation 20

$$\boxed{QF_{\text{Product}} = QF_{\text{Multiplicand}} + QF_{\text{Multiplier}}}$$

Equation 21

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 in increments of 0.0001220703125 (-4 to 4 – 1/8192). The Q3.13 representation bit weighting is shown below.

$$\begin{array}{c} |\,s\,|\,x\,|\,x.\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,| \\ \hline |\,-4\,|\,2\,|\,1.\,|\,1/2\,|\,1/4\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,1/8192\,| \end{array}$$

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 loosing 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.

$$\begin{array}{c} |\,s\,|\,x\,|\,x.\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,| \\ \hline \underbrace{|\,-4\,|\,2\,|\,1.\,|\,1/2\,|\,1/4\,|\,x\,|\,x\,|\,x}_{\text{8-bit Q3.5 Number}}\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,x\,|\,1/8192\,| \end{array}$$

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 (i.e. α, β, χ are signed), with $\alpha = 1.667$, $\beta = -0.75$, and $\chi = 2.6$, with maximal resolution on each variable, compute:

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

$$QI_\alpha = floor\Big(\log_2\big(\max\big(abs(\alpha_{\min}, \alpha_{\max})\big)\big) + 2\Big)$$
$$QI_\alpha = floor\big(\log_2(1.8) + 2\big) = floor(0.848 + 2) = 2$$
$$QF_\alpha = WL - QI_\alpha = 8 - 2 = 6$$
$$\text{so: } \varepsilon_\alpha = \frac{1}{2^{QF_\alpha}} = \frac{1}{2^6} = \frac{1}{64} = 0.015625$$
$$-115_{10}(Q2.6) \leq \alpha_{FxdPt} \leq 115_{10}(Q2.6)$$
$$\alpha_{FxdPt} = 1.667 \times 2^6 = 106_{10}(Q2.6)$$

$$QI_\beta = floor\Big(\log_2\big(\max\big(abs(\beta_{\min}, \beta_{\max})\big)\big) + 2\Big)$$
$$QI_\beta = floor\big(\log_2(1 - \varepsilon) + 2\big)$$
$$QI_\beta = floor\big(\log_2(0.999999999) + 2\big) = 1$$
$$QF_\beta = WL - QI_\beta = 8 - 1 = 7$$
$$\text{so: } \varepsilon_\beta = \frac{1}{2^{QF_\beta}} = \frac{1}{2^7} = \frac{1}{128} = 0.0078125$$
$$-127_{10}(Q1.7) \leq \beta_{FxdPt} \leq 1127_{10}(Q1.7)$$
$$\beta_{FxdPt} = -0.75 \times 2^7 = -96_{10}(Q1.7)$$

$$QI_\chi = floor\Big(\log_2\big(\max\big(abs(\chi_{\min}, \chi_{\max})\big)\big) + 2\Big)$$
$$QI_\chi = floor\big(\log_2(2.8) + 2\big) = 3$$
$$QF_\chi = WL - QI_\chi = 8 - 3 = 5$$
$$\text{so: } \varepsilon_\chi = \frac{1}{2^{QF_\chi}} = \frac{1}{2^5} = \frac{1}{32} = 0.03125$$
$$-90_{10}(Q3.5) \leq \chi_{FxdPt} \leq 89_{10}(Q3.5)$$
$$\chi_{FxdPt} = 2.6 \times 2^5 = 83_{10}(Q3.5)$$

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

Computing the product term $(\alpha \times \beta)$:

$$\alpha_{FxdPt} \times \beta_{FxdPt} = 106_{10}(Q2.6) \times -96_{10}(Q1.7) = -10176_{10}(Q3.13) = \text{-}1.2421875$$

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 α but in a 16-bit format.

Before computing the sum $(\alpha \times \beta) + \chi$, the 16-bit product term and χ 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 χ 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.

Scaling the product term to align the decimal places:

$$\alpha_{FxdPt} \times \beta_{FxdPt} = -10176_{10}(Q3.13) >> 8 = \frac{-10176_{10}}{2^8} = -39_{10}(Q3.5)$$

Adding the scaled product term and χ

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

The answer is:

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

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

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

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. Computing QI & QI≤0

The number of integer bits may be computed in several ways. It is arguably preferable to compute QI using Equation 5 and Equation 12 from Method 2 and Method 4 respectively. Because of the modified constraints in methods 2 & 4, "<" as opposed to "≤" it is important to evaluate if $\log_2(|\alpha|)$ or "$\log_2(|\alpha|) + 1$" compute to an exact an integer value. If it does QI must be incremented to the next largest integer value of bits. This is because *ceiling* of an integer value is itself. Incrementing QI by 1 when the log2(|a|) is an integer value is done by adding 1 to and taking the *floor* of the constraint equations (Equation 5 & Equation 12).

Equation 5 and Equation 12 from Method 2 and Method 4 respectively are arguably preferable because they can yield negative QI values. If QI is negative (i.e. the number is fractional only), QF can be increased to the standard WL used to increase resolution of the fractional content. QI is the weight of the most significant bit in the fixed-point number. QI<0 implies fractional weight.

For example if:

$$-0.05 \le \alpha \le 0.05 \text{ with } \varepsilon \le 0.0001$$

$$QI_\alpha = floor\left(\log_2\left(\max\left(abs\left(\alpha\right)\right)\right)+2\right)$$
$$QI_\alpha = floor\left(\log_2\left(0.05\right)+2\right)$$
$$QI_\alpha = floor\left(-4.3219+2\right) = floor\left(-2.3129\right)$$
$$\therefore QI_\alpha = -3$$

$$QF_\alpha = ceiling\left(\log_2\left(\frac{1}{\varepsilon_\alpha}\right)\right)$$
$$QF_\alpha = ceiling\left(\log_2\left(\frac{1}{0.0001}\right)\right)$$
$$QF_\alpha = ceiling\left(\log_2\left(10000\right)\right)$$
$$QF_\alpha = ceiling\left(13.288\right) = 14$$

$QI_\alpha + QF_\alpha = -3+14 = 11$ bits are required to represent α

$$0.05\times 2^{14} = 819.2 \rightarrow 0x0333 = 0000\ 0\underline{011\ 0011\ 0011}$$

0	0.	0	0	0	0	1	1	0	0	1	1	0	0	1	1
-2	1.	½	¼	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048	1/4096	1/8192	1/16384
S	S.	S	S	S	-1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048	1/4096	1/8192	1/16384

So for a 16-bit WL we can increase QF by 5-bits

$QI_\alpha + QF_\alpha = -3+19 = 16$ bits to represent α

This improves resolution to $\varepsilon = 2^{-19} = 1.9073486328125\times 10^{-6}$

$$0.05\times 2^{19} = 26214.4 \rightarrow 0x6666 = \underline{0110\ 0110\ 0110\ 0110}$$

.x	x	x	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
.x	x	x	S	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Example 8

3.2. 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.

3.3. 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 lower half of 2xWL result will be returned. This is done by typecasting the multiplicand (or multiplier) to a 2xWL variable type. This will sign extend the multiplicand (or multiplier) to 2xWL to properly compute the product.

3.4. General Caveats & Example C Code

A point of caution: some compilers contains 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 assuming that char and int variable types are signed 8-bit and 16-bit respectively. This code example also assumes appropriate variables are loaded with fixed-point values of the listed Q format elsewhere.

```
/* 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

1. Joint Technical Committee ISO/IEC JTC1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces.*, Working Group WG14, "ISO/IEC 9899:TC2 Committee Draft – May 6, 2005 WG14/N1124", 2005, Retreived August 21, 2007 from the World Wide Web:

   http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

2. K.-I. Kum, J. Kang, W. Sung, "A Floating-point to Fixed-point C Converter For Fixed-point Digital Signal Processors", Second SUIF Compiler Workshop, 1997, Retrieved July 12, 2004 from the World Wide Web:

   http://suif.stanford.edu/suifconf/suifconf2/papers/4.ps

3. K.-I. Kum, J. Kang, W. Sung , "A Floating-Point to Integer C Converter with Shift Reduction for Fixed-Point Digital Signal Processors," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing 1997*, ICASSP'99, pp. 2163-2166, March 1999

4. S. Kim and W. Sung "A Floating-point to Fixed-point Assembly Program Translator for the TMS320C25," *IEEE Transactions on Circuits and Systems*, vol. 41, no.11, pp.730-739, November 1994

5. J. Kang and W. Sung, "Fixed-Point C Compiler for TMS320C50 Digital Signal Processor," *Proceeding of the International Conference on Acoustics, Speech, and Signal Processing 1997*, pp. 707-710, 1997.

6. A. G. M. Cilio and H. Corporaal, "Floating Point to Fixed Point Conversion of C Code," *Delft University of Technology: Computer Architecture and Digital Techniques Dept.*, Retrieved July 12, 2004 from the World Wide Web: http://citeseer.nj.nec.com/cache/papers/......./floating-point-to-fixed.pdf