A DISCUSSION OF THE BLOCK FLOATING-POINT FFT IMPLEMENTATION ON FIXED-POINT DSPs

Arun Chhabra, Member Technical Staff <*achhabra@ti.com*>
Ramesh A. Iyer, Member Group Technical Staff <*riyer@ti.com*>
Texas Instruments, San Jose, California

ABSTRACT

Block floating-point (BFP) implementation provides an innovative method of floating-point emulation. This paper implements the BFP algorithm for the Fast Fourier Transform (FFT) algorithm. The BFP algorithm as it applies to the FFT allows fractional signal gain adjustment in a fixed-point environment by using a block representation of input values of block size N to an N-point FFT. This algorithm is applied repetitively to all stages of the FFT. The elements within a block are further represented by their respective mantissas and a common exponent assigned to the block. This method allows for aggressive scaling with a single exponent while retaining greater dynamic range in the output. This workshop discusses the BFP FFT and demonstrates its implementation in assembly language. The implementation is carried out on a fixed-point digital signal processor (DSP). The fixed-point BFP FFT results are contrasted with the results of a floating-point FFT of the same size implemented with MATLAB. For applications where the FFT is a core component of the overall algorithm, the BFP FFT can provide results approaching floating-point dynamic range on a low-cost fixed-point processor. Most DSP applications can be handled with fixed-point representation. However, for those applications that require extended dynamic range but do not warrant the cost of a floating-point chip, a block floating-point implementation on a fixed-point chip readily provides a cost-effective solution.

FIXED AND FLOATING POINT REPRESENTATIONS

Fixed-point processors represent numbers either in fractional notation - used mostly in signal processing algorithms, or integer notation - primarily for control operations, address calculations and other non-signal processing operations. Clearly the term fixed-point representation is not synonymous with integer notation. In addition, the choice of fractional notation in digital signal processing algorithms is crucial to the implementation of a successful scaling strategy for fixed-point processors.

Integer representation encompasses numbers from zero to the largest whole number that can be represented using the available number of bits. Numbers can be represented in two's complement form with the most significant bit as the sign bit that is negatively weighted.

Fractional format is used to represent numbers between -1 and 1. A binary radix point is assumed to exist immediately after the sign bit that is also negatively weighted. For the purpose of this paper, the term fixed-point will imply use of the fractional notation.

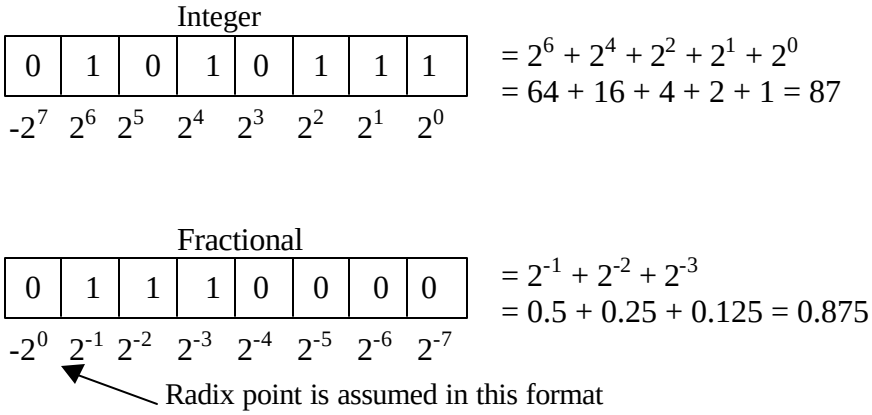
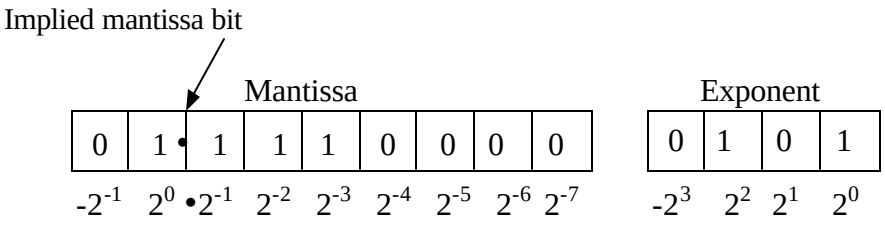


Figure 1. Diagram of Fixed Point representations - integer & fractional

Floating-point arithmetic consists of representing a number by way of two components - a mantissa and an exponent. The mantissa is generally a fractional value that can be viewed to be similar to the fixed-point component. The exponent is an integer that represents the number of places that the binary point of the mantissa must be shifted in either direction to obtain the original number. In floating point numbers, the binary point comes after the second most significant bit in the mantissa.



mantissa = $2^0 + 2^{-1} + 2^{-2} + 2^{-3} = 1 + 0.5 + 0.25 + 0.125 = 1.875$

exponent = $2^2 + 2^0 = 4 + 1 = 5$

decimal value = $1.875 \times 2^5 = 60$

Figure 2. Diagram of Floating-Point representation

PRECISION, DYNAMIC RANGE AND QUANTIZATION EFFECTS

Two primary means to gauge the performance of fixed-point and floating-point representations are dynamic range and precision.

Precision defines the resolution of a signal representation; it can be measured by the size of the least significant bit (LSB) of the fraction. In other words, the word-length of the fixed-point format governs precision. For floating-point format, the number of bits that make up the mantissa give the precision with which a number can be represented. Thus, for the floating-point case, precision would be the minimum difference between two numbers with a given common exponent. An added advantage of the floating-point processors is that the hardware automatically scales numbers to use the full range of the mantissa. If the number becomes too large for the available mantissa then the hardware scales it down by shifting it right. If the number consumes less space than the available word-length, the hardware scales it up by shifting it left. The exponent tracks the number of these shifts in either direction.

The dynamic range of a processor is the ratio between the smallest and largest number that can be represented. The dynamic range for a floating-point value is clearly determined by the size of the exponent. As a result, given the same word-length, a floating-point processor will always have a greater dynamic range than a fixed-point processor. On the other hand, given the same word-length, a fixed-point processor will always have greater precision than floating-point processors.

Quantization error also serves as a parameter by which the difference between fixed-point and floating-point representations can be measured. Quantization error is directly dependent on the size of the LSB. As the number of quantization levels increases the difference between the original analog waveform and it's quantized digital equivalent becomes less. As a result, the quantization error also decreases, thereby lowering the quantization noise. It is clear then that the quantization effect is directly dependent on the word-length of a given representation.

The increased dynamic range of a floating-point processor does come at a price. While providing increased dynamic range, floating-point processors also tend to cost more and dissipate more power than fixed-point processors, as more logic gates are required to implement floating-point operations.

THE BLOCK FLOATING POINT CONCEPT

At this point it is clear that fixed and floating-point implementations have their respective advantages. It is possible to achieve the dynamic range approaching that of

floating-point arithmetic while working with fixed-point processors. This can be accomplished by using floating-point emulation software routines. Emulating floating-point behaviour on a fixed-point processor tends to be very cycle intensive, since the emulation routine must manipulate all arithmetic computations to artificially mimic floating-point math on a fixed-point device. This software emulation is only worthwhile if a small portion of the overall computation requires extended dynamic range. Clearly, a cost-effective alternative for floating-point dynamic range implemented on a fixed-point processor is needed.

The block floating point algorithm is based on the block automatic gain control (AGC) concept. Block AGC only scales values at the input stage of the FFT. It only adjusts the input signal power. The block floating point algorithm takes it a step further by tracking the signal strength from stage to stage to provide a more comprehensive scaling strategy and extended dynamic range.

The floating-point emulation scheme discussed here is the block floating-point algorithm. The primary benefit of the block floating-point algorithm emanates from the fact that operations are carried out on a block basis using a common exponent. Here, each value in the block can be expressed in two components - a mantissa and a common exponent. The common exponent is stored as a separate data word. This results in a minimum hardware implementation compared to that of a conventional floating-point implementation.

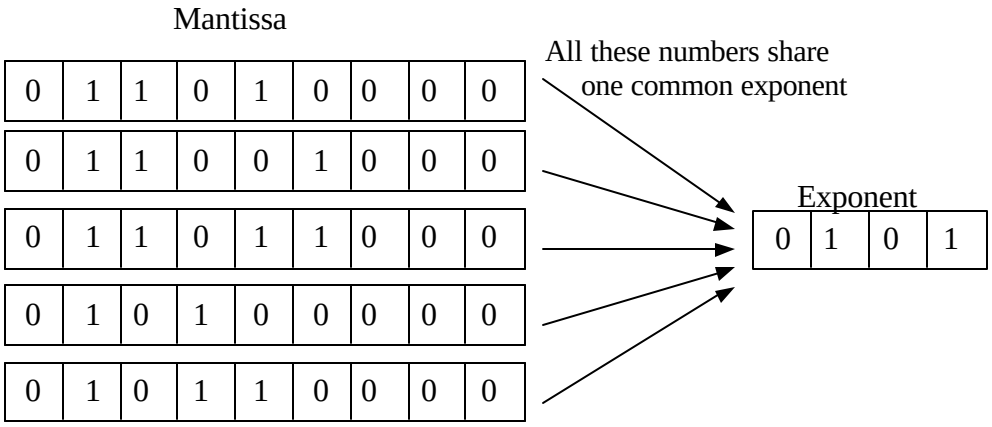


Figure 3. Diagram of Block Floating-Point representation

The value of the common exponent is determined by the data element in the block with the largest amplitude. In order to compute the value of the exponent, the number of leading bits has to be determined. This is determined by the number of left shifts required for this data element to be normalized to the dynamic range of the processor. Certain DSP processors have specific instructions, such as exponent detection and

normalization instructions, that perform this task. If a given block of data consists entirely of small values, a large common exponent can be used to shift the small data values left and provide more dynamic range. On the other hand, if a data block contains large data values, then a small common exponent will be applied. Whatever the case may be, once the common exponent is computed, all data elements in the block are shifted up by that amount, in order to make optimal use of the available dynamic range. The exponent computation does not consider the most significant bit, since that is reserved for the sign bit and is not considered to be part of the dynamic range.

As a result, block floating-point representation does provide an advantage over both, fixed and floating-point formats. Scaling each value up by the common exponent increases the dynamic range of data elements in comparison to that of a fixed-point implementation. At the same time, having a separate common exponent for all data values preserves the precision of a fixed-point processor. Therefore, the block floating-point algorithm is more economical than a conventional floating-point implementation.

THE BLOCK FLOATING POINT FOR A COMPLEX FFT

The block floating-point analysis presented here is based on its application to a 64-point complex decimation-in-time (DIT) Fast Fourier Transform (FFT). The assembly code that implements this FFT will be referred to as the "original code" through this paper. Block floating-point scaling is implemented by determining the input-scaling factor for each butterfly stage based on the actual bit growth of the previous stage. This can be implemented in a number of ways. One technique is as explained above by computing the number of leading bits and normalizing the whole array of input values by that amount. At the end of the computations of each stage, the output values are scaled down by the required amount such that they do not lead to overflow when used as the input to the next stage. This process will repeat itself so that maximum possible dynamic range is maintained while averting the possibility of an overflow. The original code only employs binary scaling. It is also possible to perform non-binary scaling with BFP that allows for fractional gain adjustments. This paper employs both of the above scaling techniques.

The first stage of our FFT implementation is a radix-4 butterfly for code and execution speed optimization. The unique characteristic of this stage is that the magnitude growth can be no more than a factor of 4, since the value of theta ( ) can only be in increments of □/2. As a result, the scaling factor for that input array was chosen to be , or a right shift of two bit places.

The subsequent stages of our FFT are radix-2 butterflies. The maximum theoretical magnitude growth possible for a general radix-2 FFT butterfly is a factor of 2.414. Given that a radix-2 butterfly can be expressed as $A' = A + (B*W)$, where all values are complex and W is the twiddle factor, we note that W will reach it's maximum

magnitude at □4. Given this, it is obvious that *A′* will attain it's maximum possible value when *A* = 1+ j0; *B* = 1 - j1; *W* = 0.707 + j0.707. This results in a maximum gain of 2.41421356. Thus the scaling based on this signal growth factor will be 1/2.414 » 0.4167.

IMPLEMENTING THE BLOCK FLOATING POINT - APPROACHES TAKEN

Three different approaches were adopted in implementing the block floating-point concept for a 64-point complex FFT. The input to approaches I and II was a rail-to-rail complex random noise generated with MATLAB. Approach III uses the same complex random noise but scaled down by a factor of 2.

In the first approach, prior to processing any input values to the first radix-4 butterfly stage, all values are scaled up such that the signal occupies the entire dynamic range of the processor. This radix-4 stage is scaled by a factor of 4 to prevent overflow. Each subsequent radix-2 stage is automatically scaled by a factor of 2 in the original FFT code.

In the second approach, input values to the radix-4 stage are scaled up such that the signal occupies the entire dynamic range of the processor. In addition, scaling is done on a conditional basis in the block floating-point code when the maximum input value to each butterfly stage crosses a pre-determined threshold value. Should scaling be required in the block floating-point code, a scale down factor of (2.414/2) is applied in conjunction with a binary scale factor of 2. This results in multiplying the block of values by a factor of (1/(2.414/2)) = 0.83, followed by a binary output scaling of a factor of 2. Thus, the block floating-point implementation also allows a combination of binary and fractional scaling to achieve optimal performance for a fixed-point processor.

The third approach draws on the benefits of approaches I and II. Similar to approach I, the input set of values is scaled up to fully occupy the processor dynamic range. Drawing from approach II, the automatic scaling feature from the original fixed-point FFT code is disabled. This approach was an attempt to view the impact of an input signal of smaller amplitude on the results. The input used in this approach is a signal that is identical to the input signal of the previous steps in frequency distribution. However, in this case the amplitude of each of the frequency bins is half that of the previous approaches. Scaling is carried out similar to approach two.

ANALYSIS OF RESULTS

The results of the block floating-point FFT are compared against two known good results sets - those of the floating-point MATLAB environment and of the pure fixed-point DSP environment.

The primary methods adopted to analyze the results of the block floating-point implementation are quantization error and signal-to-noise (SNR). The quantization error is a suitable study of the results since it compares the corresponding values between two types of signals. Calculating the total noise power for a given pair of signals results in the quantization error. Given two complex signals, A and B (where signal A is the reference signal against which comparison is carried out and signal B is the signal who's performance is under test), the total noise power computation for these two signals is found by

$$\sum (R_A - R_B)^2 + \sum (\mathrm{Im}_A - \mathrm{Im}_B)^2$$

where 'R' and 'Im' denote real and imaginary quantities, respectively. This is the quantization error. Similarly, the total signal power can be computed by

$$\sum (R_A)^2 + \sum (\mathrm{Im}_A)^2 .$$

The quantization error is computed for the signal under test (signal B) with respect to a signal considered to be the reference for comparison (signal A). As a result, in this analysis, the total signal power is always calculated with the reference signal - signal A in the equations above.

Armed with this knowledge, the computation for the SNR becomes relatively simple. It is the ratio of the total signal power to the total noise power. Using the equations above

$$\mathrm{SNR} = \frac{\sum (R_A)^2 + \sum (\mathrm{Im}_A)^2}{\sum (R_A - R_B)^2 + \sum (\mathrm{Im}_A - \mathrm{Im}_B)^2}$$

Or in dB the SNR can be expressed as (10 log $\mathrm{SNR}_{\text{power ratio}}$).

Provided that the block floating-point algorithm works as intended, its SNR results should be an improvement over the SNR for fixed-point results. The SNR in both these cases will be computed relative to the signal power for the reference MATLAB implementation. In addition, block floating-point will also promise an improved quantization error result when compared to that of the fixed-point implementation.

The results of the three different approaches are highlighted in the summary table of results below. It is clear that modifications made in the implementation of each case produced results that were better than those of the previous cases.

Approach	SNR (dB)	Quantization Error Power	Total Signal Power
I BFP Fixed Pt FFT	55.62 53.35	1.5893e-7 2.6819e-7 *41% improvement*	0.0580
II BFP Fixed Pt FFT	56.25 53.35	1.3738e-7 2.6819e-7 *49% improvement*	0.0580
III BFP Fixed Pt FFT	51.6 47.9	1.0056e-7 2.3397e-7 *58% improvement*	0.0145

Table 1. Summary of Results [1]

The results in table 1 shows that the block floating-point implementation of approach I provides a 41% improvement over the fixed-point FFT implementation. This is a benefit arrived at by scaling up all values in the input such that they fully occupy the dynamic range available. As hypothesized, this case indicates a good result, since shifting all values to the left allows for increased precision.

The block floating-point implementation of approach II produces a 49% improvement when compared to the fixed-point FFT implementation. Recall that this technique is based on disabling the automatic scaling in butterfly stages that was present in the previous approach. Since bit growth of values between butterfly stages is a possibility but not a necessity, scaling down automatically during each stage can compromise the precision of results. Scaling down only makes sense if there is evidence of bit growth during butterfly computations. However, when bit growth is not expected, the results of that stage can be directly fed to the next stage while yielding full benefit of the available dynamic range. Automatically scaling down in this last instance would use less of the full dynamic range. In this manner, optimal scaling from stage-to-stage is

[1] The results in Table 1 reflect a relative performance comparison between the fixed and floating-point approaches. The test code used to produce these results is not optimized for SNR figures.

used to prevent overflow while at the same time accuracy of the overall system is improved.

It is clear from the result table that approach III outputs the best quantization error amongst the three cases that were investigated. It is important to keep in mind that the reduced signal amplitude in the third approach will lead to a lower total signal power value. As a result, the SNR from this approach will not appear as high as the other cases. Thus, this approach is best suited for cases where a smaller quantization error is of primary interest, without concern for the slightly reduced SNR.

CONCLUSION

The benefits of the block floating point algorithm are apparent. From the results of our experiments, it is clear that the block floating-point implementation produces improved quantization error over the fixed-point implementation. It is important to note that the results in Table 1 reflect our interest to compare the relative performance between the fixed and block floating-point approaches. The test code used to produce these results is not optimized for SNR figures.

The separate common exponent is the key characteristic of the block floating point implementation. It increases the dynamic range of data elements of a fixed-point implementation by providing a dynamic range similar to that of a floating-point implementation. By using a separate memory word for the common exponent, the precision of the mantissa quantities is preserved as that of a fixed-point processor. By the same token, the block floating point algorithm is more economical than a conventional floating-point implementation.

The majority of applications are best suited for fixed-point processors. For those that require extended dynamic range but do not warrant the cost of a floating-point chip implementation, the block floating point implementation on a fixed-point chip readily provides a cost-effective solution.

REFERENCE

[1] Characteristics of DSP Processors, *Buyer's Guide to DSP Processors*, Berkeley Design Technology, Inc., 1994, pp. 33-41.

[2] *Introduction to DSP - DSP processors: Data Formats*, Bores Signal Processing, http://www.bores.com/courses/intro/chips/6_data.htm

[3] Hugh McLaughlin, SignalWorks Inc., http://www.signalworks.com

[4] Dr. John So, Texas Instruments, Inc., http://www.ti.com