Application Report
SPRA948 − September 2003

A Block Floating Point Implementation for an N-Point FFT on the TMS320C55x DSP

David Elam and Cesar Iovescu *TMS320C5000 Software Applications*

ABSTRACT

A block floating-point (BFP) implementation provides an innovative method of floating-point emulation on a fixed-point processor. This application report implements the BFP algorithm for the Fast Fourier Transform (FFT) algorithm on a Texas Instruments (TI) TMS320C55x™ DSP by taking advantage of the CPU exponent encoder. The BFP algorithm as it applies to the FFT allows 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 application report 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 which 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.

Contents

List of Figures

List of Tables

1 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 twos 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 application report, the term fixed-point will imply use of the fractional notation.

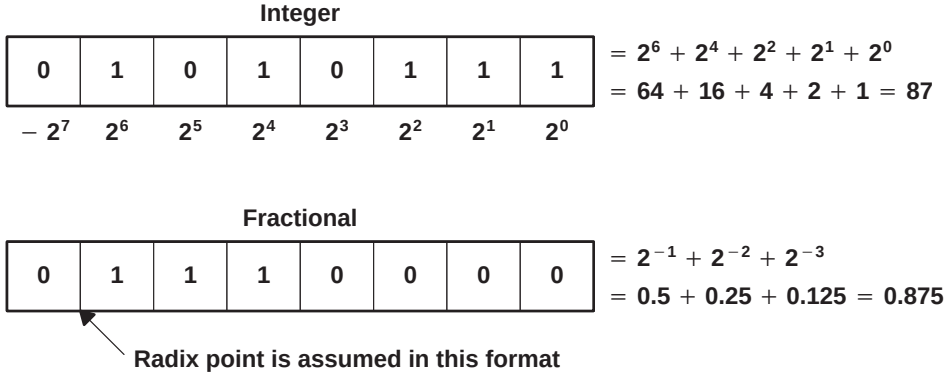
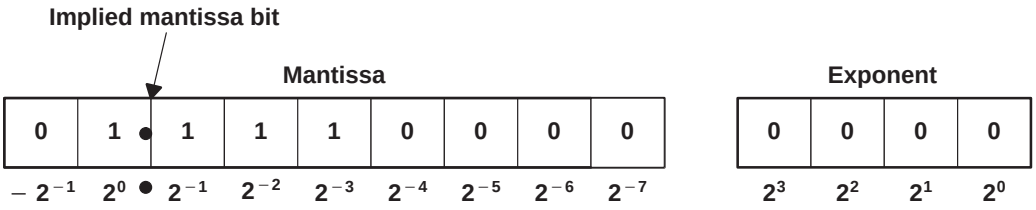


Figure 1. Diagram of Fixed-Point Representations − Integer and Fractional

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

Decimalvalue $= 1.875 \times 2^5 = 60$

Figure 2. Diagram of Floating-Point Representation

2 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, 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 its 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.

3 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 behavior 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.

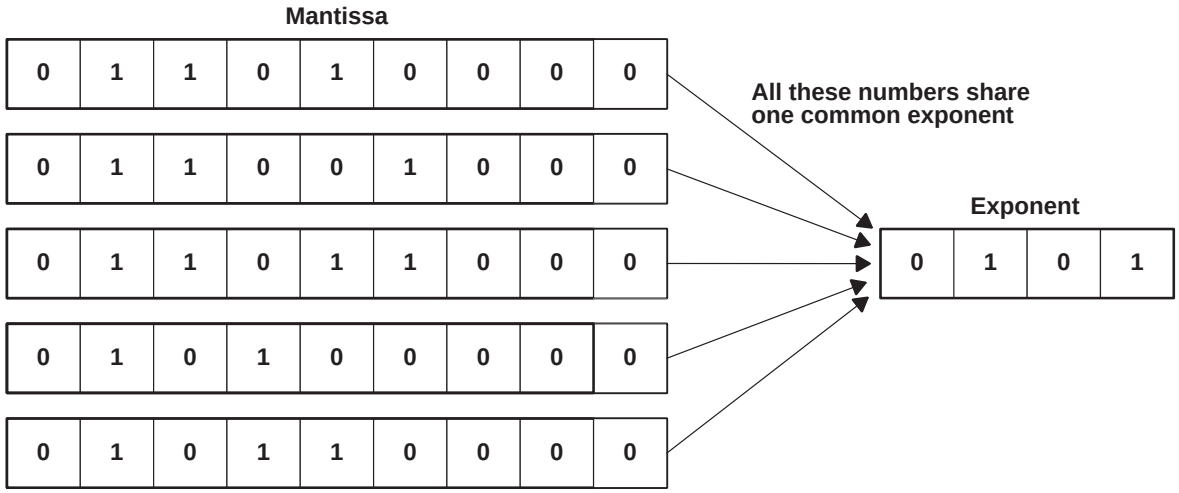


Figure 3. Diagram of Block Floating-Point Representation

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.

4 Bit-Growth of an FFT Stage

The precision of a decimation-in-time (DIT) FFT stage can be enhanced by predicting an upcoming stage's bit growth and normalizing the input block accordingly to use the maximum possible dynamic range. Several factors determine a stage's bit growth, such as the twiddle factors involved, whether or not complex arithmetic is used, and the radix of the stage.

Since precision is saved by using two radix-2 stages rather than one radix-4 stage, all stages implemented here are radix-2. The basic radix-2 butterfly computation in the DIT FFT algorithm is shown in Figure 4 where both the input and twiddle factors are complex values.

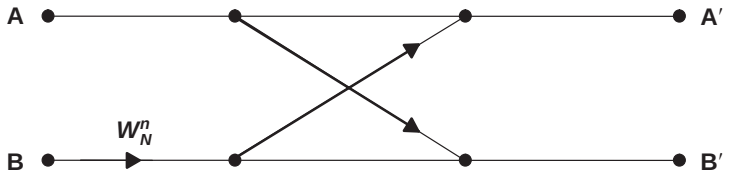


Figure 4. Radix-2 DIT FFT Butterfly Computation

$$A' = (A_r + iA_i) + (W_{Nr}^n + iW_{Ni}^n)(B_r + iB_i)$$

$$B' = (A_r + iA_i) - (W_{Nr}^n + iW_{Ni}^n)(B_r + iB_i)$$

The real and imaginary parts of the butterfly output must be considered separately when computing the theoretical bit growth since each component occupies a separate memory location.

$$A' = (A_r + B_r W_{Nr}^n - B_i W_{Ni}^n) + i(A_i + B_r W_{Ni}^n + B_i W_{Nr}^n)$$

$$B' = (A_r - B_r W_{Nr}^n + B_i W_{Ni}^n) + i(A_i - B_r W_{Ni}^n - B_i W_{Nr}^n)$$

4.1 Bit Growth in the First Two Stages

In the first stage of any N-length DIT FFT where N is a power of two, W_N^0 = 1+j0 and W_N^N = 1+j0 are the only twiddle factors involved in the butterfly computations. Since A and B are complex values with real and imaginary parts less than 1, the maximum growth factor is 2 (bit growth of 1) according to equation 1.3, such as where A = B = 1 + j1. Stage two also has a maximum growth factor of 2 since W_N^0 = 1+j0, $W_N^{N/4}$ = 0–*i*, and W_N^N = 1+j0 are the only twiddle factors involved in stage two butterflies as seen in Figure 5.

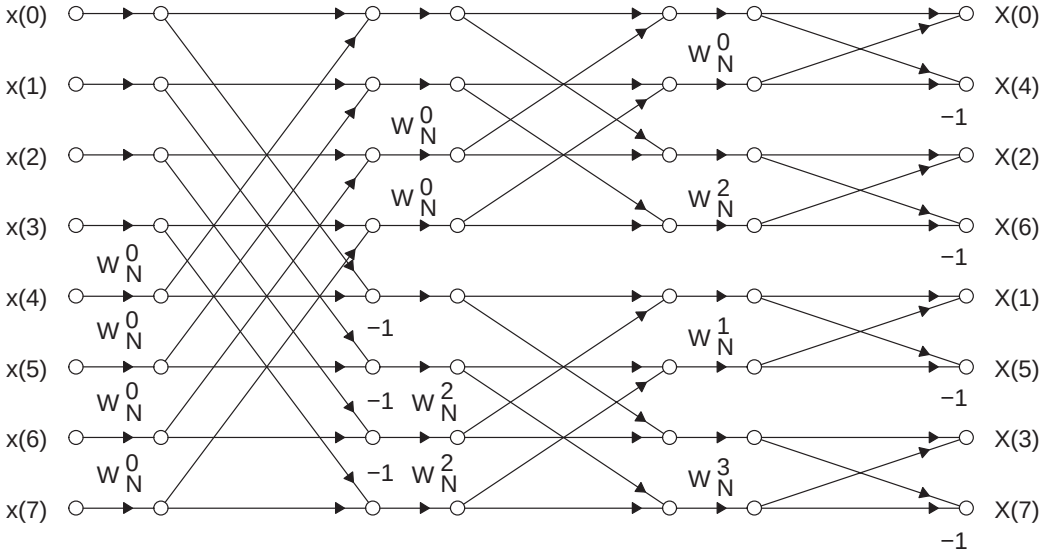


Figure 5. 8-Point DIT FFT

4.2 Bit Growth in the Third and Following Stages

The third through the final $\log_2 N$ stages involve twiddle factors that have both a real and imaginary component, enabling the butterflies to grow by more than a factor of two. In general, the maximum growth factor for any radix-2 butterfly can be found by computing the maximum absolute value of the real and imaginary parts of A' and B' as follows:

$$\begin{aligned} Max|A'| &= Max\left|A_r + B_r W_{Nr}^n - B_i W_{Ni}^n\right| \vee Max\left|A_i + B_i W_{Nr}^n + B_r W_{Ni}^n\right| \\ &= Max|A_r + B_r \cos(\theta) - B_i \sin(\theta)| \vee Max|A_i + B_i \cos(\theta) + B_r \sin(\theta)| \end{aligned}$$

$$\begin{aligned} Max|B'| &= Max\left|A_r - B_r W_{Nr}^n + B_i W_{Ni}^n\right| \vee Max\left|A_i - B_i W_{Nr}^n - B_r W_{Ni}^n\right| \\ &= Max|A_r - B_r \cos(\theta) + B_i \sin(\theta)| \vee Max|A_i - B_i \cos(\theta) - B_r \sin(\theta)| \end{aligned}$$

The real and imaginary parts of A and B include and are between −1 and 1, so

$$Max|A_r| = Max|A_i| = Max|B_r| = Max|B_i| = 1$$

A maximum is reached when each of the three components in the real or imaginary part are of the same sign and θ is at an angle that corresponds to a maximum.

$$Max|A'| = Max|B'| = Max|1 \pm \cos(\theta) \pm \sin(\theta)|$$

$$\frac{d}{d\theta}(1 \pm \cos(\theta) \pm \sin(\theta)) = 0$$

$(\pm \cos(\theta) \pm \sin(\theta)) = 0$

Maximums at $\theta = \frac{\Pi}{4} + n\frac{\Pi}{2} \quad n = 0, 1, \ldots, \infty$

The maximum growth factor is **2.4142 (2 bits)**, such as in the case where A = 1 + j0, B = 1 + j1, and W = $\Pi/4$ + j $\Pi/4$ in the equation 1.1 butterfly computation. **Thus, the input data must be prescaled to allow for one bit of growth in the first two stages and two bits of growth in each stage thereafter if overflow is to be averted**.

5 Implementing the Block Floating-Point Algorithm

The block floating-point (BFP) analysis presented here is based on its application to a 16-bit, N-point, complex DIT FFT with radix-2 stages and Q.15 data. The BFP FFT algorithm relies on scaling before each stage to maximize precision yet prevent overflow and can be implemented one of two ways.

- **Fractional scaling** – Scaling by some non-integer value to normalize the input to ½ before the first two stages and to 1/2.4142 before the following stages. Some precision can be gained by fractional scaling when less than one bit of growth occurs, but a cycle intensive computation of the reciprocal is required to normalize the data.
- **Integer scaling** – Scaling by some power of two in order to normalize the input to ½ before the first two stages and to 1/4 before the following stages. Scaling by powers of two incurs some precision loss but involves computationally efficient shifts.

Fractional scaling is feasible through an iterative algorithm but might not warrant the small gain over integer scaling. The BFP FFT algorithm implemented here scales by powers of two. It is important to note that no automatic gain control (AGC) is implemented within the actual FFT stage computation, such as in the case of the C55x DSPLIB function cfft_scale(), which indiscriminately employs binary scaling (integer scaling by ½) prevent overflow, regardless of the data magnitude. Binary scaling offers a less cycle intensive approach to scaling but decreases precision unnecessarily in most cases. The BFP method implemented here is designed to handle any size of input data, whereas simple binary scaling may overflow in some cases where the input magnitude of the maximum real or complex value exceeds 1/2.4142.

5.1 Scaling in the First Two Stages

The input data to the first stage of the complex FFT is scaled to occupy the maximum possible dynamic range that allows for a single bit growth in the upcoming stage. In other words, if the maximum datum is below ¼ (8192 in Q.15) or above ½ (16384 in Q.15) then the input array is normalized by some power of two that gives the maximum datum room for one bit of growth. On the TMS320C55x, this is feasible through the CPU exponent encoder, which provides the location of the MSB of a given datum. The data in the subsequent radix-2 stage then grows by either zero or one bit. If no growth or only fractional growth occurs, then no scaling is performed. If any real or imaginary data grows by one bit, then all values are scaled down by one bit to prepare for second stage bit growth.

5.2 Scaling in the Third and Following Stages

Before the third and rest of the $\log_2 N$ stages, the input data is scaled by some factor of two that allows for two bits of growth, i.e. the maximum datum must stay between 1/8(4096 in Q.15) and ¼(8192 in Q.15) to prevent overflow yet maximize the dynamic range.

5.3 The Block Exponent

A record is kept of the scaling factors from each stage so that the block exponent of the output magnitude can be recovered. Following the final stage, the total number of shifts, i.e. the block common exponent, is returned to allow the proper output magnitude to be recovered.

Figure 6 illustrates the computational flow of the BFT FFT algorithm.

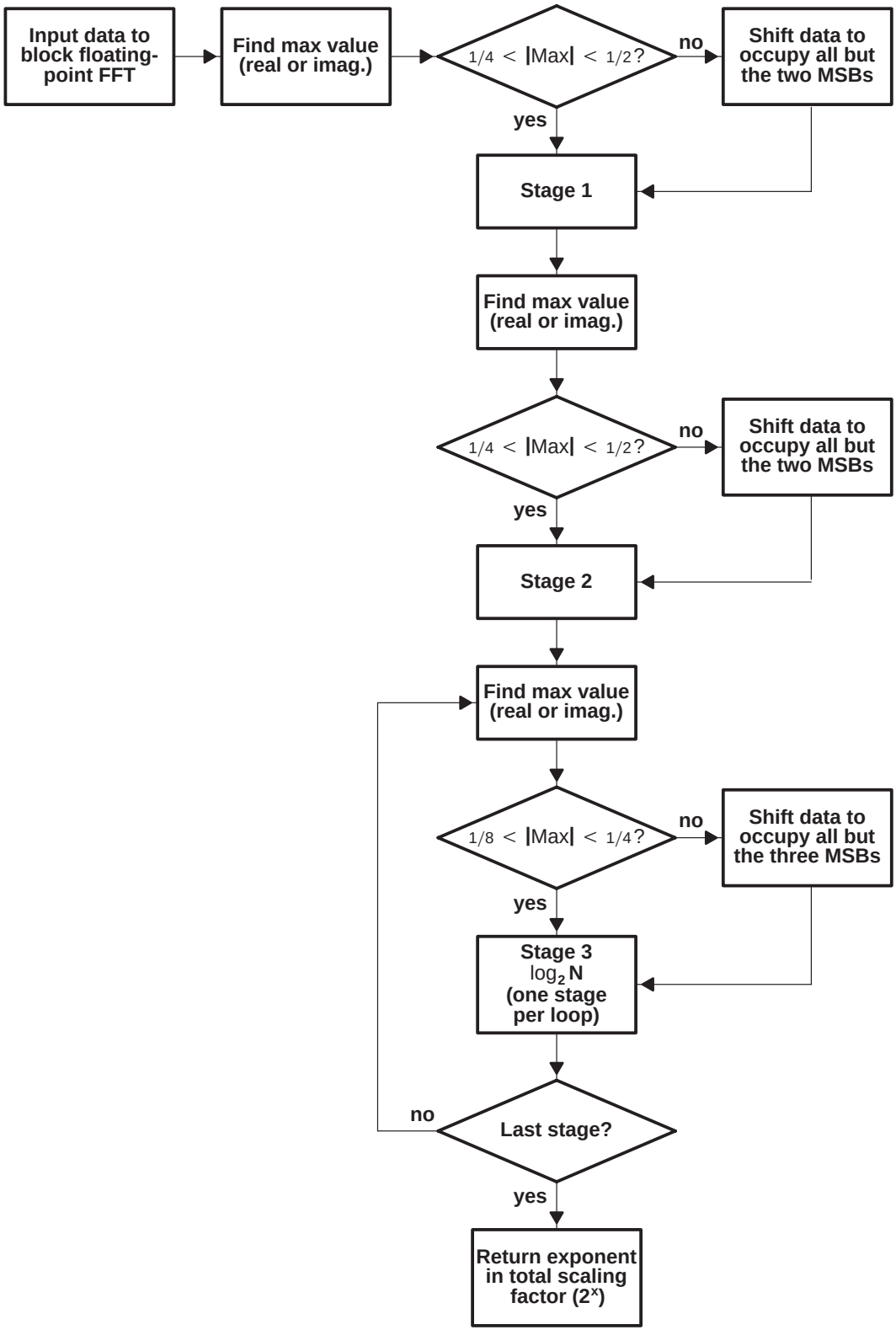


Figure 6. Block Diagram of the BFP FFT Implementation

6 BFP FFT Precision

To test the precision savings of the block floating-point FFT, the results were compared with two known good result sets – those of the floating-point MATLAB environment and of the pure fixed-point DSP environment. An integer formatted signal consisting of rail-to-rail complex random noise was generated in the MATLAB environment and input to each of the following FFTs.

- **16-Bit, Fixed-Point FFT with Binary Scaling** – The output is converted to floating-point format and then scaled by 2 ^ (log2N) = a power of two determined by the number of FFT stages.
- **16-Bit, Fixed-Point FFT with Block Floating-Point Scaling** – The output is converted to floating-point format and then scaled by 2 ^ (−returned exponent) = a power of two determined by the returned exponent.
- **32-Bit, Fixed-Point FFT with Binary Scaling** – The input consists of the same Q.15 integers scaled to occupy the upper 16-bits of each memory location. The output is converted to floating-point format and then scaled by 2 ^ (log2N) = a power of two determined by the number of FFT stages
- **32-Bit, Floating-Point FFT (Reference)** – The input consists of the same integers passed into the above FFTs but in floating point format. The output is used as a reference for computing the SNR of the FFTs computed in the fixed-point environment.

6.1 Computing the SNR

The signal-to-noise ratio (SNR) was computed as follows:

$$SNR(db) = 10 * \log\left(\frac{total\_signal\_power}{total\_noise\_power}\right)$$

Given two complex signals, A and B, where signal A is the reference signal (MATLAB) and signal B is the signal that is corrupted by noise (BFP and fixed-point), the total signal power is found by

$$\sum (A_R)^2 + \sum (A_{\mathrm{Im}})^2$$

The total noise power is found by

$$\sum (A_R - B_R)^2 + \sum (A_{\mathrm{Im}} - B_{\mathrm{Im}})^2$$

$$SNR(db) = 10 * \log \frac{\sum (A_R)^2 + \sum (A_{\mathrm{Im}})^2}{\sum (A_R - B_R)^2 + \sum (A_{\mathrm{Im}} - B_{\mathrm{Im}})^2}$$

To eliminate the dependency of precision on the characteristics of the signal, complex random noise generated in MATLAB is used for the input signal. The following figure illustrates the dependency of precision (SNR) on input magnitude by varying the input signal scaling factor by powers of 2 across the Q.15 spectrum.

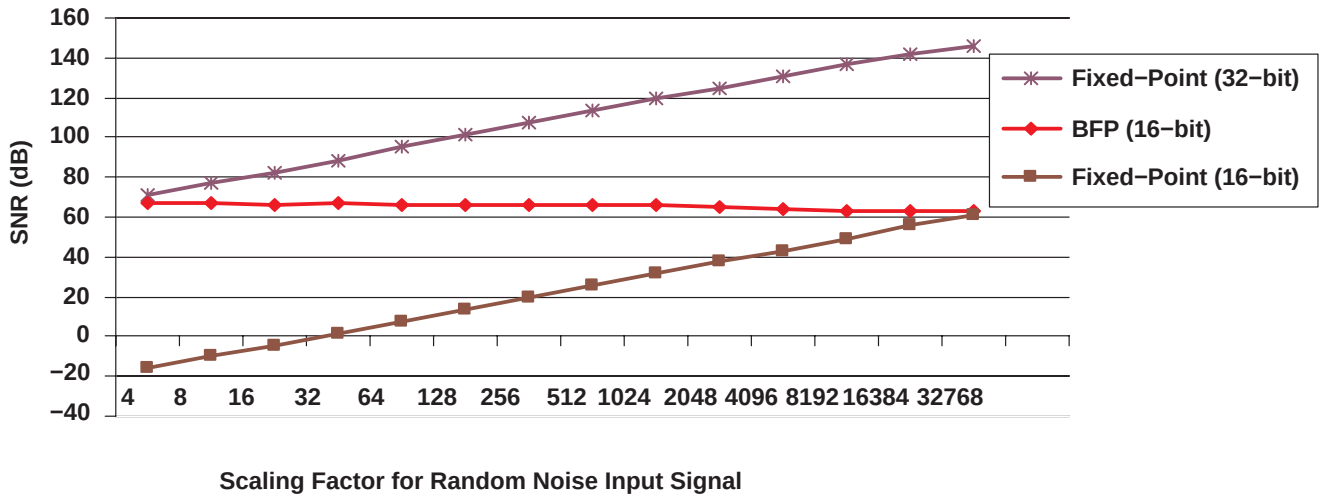


Figure 7. SNR vs. Scaling Factor for a 256-point FFT

Quantization error (noise) in this case is caused by extensive multiply-accumulates (MACs). The fixed-point FFT suffers greatly from MAC rounding since most of the dynamic range of the processor is generally not used for small input magnitudes, whereas the block floating-point does not suffer as much since most significant bits are relocated to the MSBs preceding each stage. From the above figure, it is clear that the block floating-point FFT is superior in precision to the 16-bit fixed-point FFT for all magnitudes of input data. In general, the block floating-point SNR shows more consistency but tends to decrease with increasing input magnitude. This is caused by reduced precision with increasing input magnitudes – a common feature of floating-point representations. The SNR of both the 16-bit and 32-bit fixed-point FFTs increases with increasing input magnitude since the MAC quantization error stays relatively constant as the input signal magnitude increases.

7 BFP FFT Benchmarks

Since the block floating-point FFT scales only when it is essential to prevent overflow, it has the advantage of increased precision. However, this advantage comes at the price of cycle intensive search and scale routines. The following table shows how the 16-bit BFP FFT algorithm compares with the 16-bit fixed-point FFT and 32-bit fixed-point in cycle performance.

Table 1. N-Point FFT Benchmarks (Fixed-Point vs. Block Floating-Point)

	Number of Cycles		
Number of Points (N)	cfft_scale (16-bit)	cfft_bfp (16-bit)	cfft32_scale (32-Bit)
16	356	914	723
32	621	1706	1693
64	1206	3442	3971
128	2511	7306	9217
256	5416	15906	21103
512	11841	34938	47677
1024	25946	76754	106443

Since scaling is performed only when necessary to prevent overflow in the next stage, some stage output data might not be modified, depending on the characteristics of the data. This effectively reduces the total number of cycles relative to a similar length FFT that requires scaling before every stage. The BFP FFT data in the table above assumes worst case cycle performance, i.e. bit-growth in every stage.

It is clear that the 16-bit block floating-point FFT produces SNRs rivaling that of a floating-point FFT. Though some precision can be gained by using a 32-bit, fixed-point FFT, the block floating-point FFT requires half the memory to store the input data and uses fewer cycles in most cases while offering a competitive SNR.

8 Conclusion

The benefits of the block floating-point algorithm are apparent. From the results of the experiments, it is clear that the BFP FFT implementation produces improved quantization error over the 16-bit fixed-point FFT implementation.

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 readily provides a cost-effective solution.

9 References

1. John G. Proakis and Dimitris G. Manolakis, “Digital Signal Processing, Principles, Algorithms, and Applications”, Prentice Hall, Third Edition, 1996
2. *TMS320C55x A Block Floating Point Implementation on the TMS320C54x DSP* (SPRA610)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265