FFT/IFFT Block Floating Point Scaling

October 2005, ver. 1.0 Application Note 404

Introduction

The Altera® FFT MegaCore® function uses block-floating-point (BFP) arithmetic internally to perform calculations. BFP architecture is a trade-off between fixed-point and full floating-point architecture.

Unlike an FFT block that uses floating point arithmetic, a block-floating-point FFT block does not provide an input for exponents. Internally, a complex value integer pair is represented with a single scale factor that is typically shared among other complex value integer pairs. After each stage of the FFT, the largest output value is detected and the intermediate result is scaled to improve the precision. The exponent records the number of left or right shifts used to perform the scaling. As a result, the output magnitude relative to the input level is:

$$output * 2^{-exponent}$$

For example, if exponent = –3, the input samples are shifted right by three bits, and hence the magnitude of the output is $output * 2^{3}$.

Block Floating Point

After every pass through a radix-2 or radix-4 engine in the FFT core, the addition and multiplication operations cause the data bits width to grow. In other words, the total data bits width from the FFT operation grows proportionally to the number of passes. The number of passes of the FFT/IFFT computation depends on the logarithm of the number of points. Table 1 on page 3 shows the possible exponents for corresponding bit growth.

A fixed-point architecture FFT needs a huge multiplier and memory block to accommodate the large bit width growth to represent the high dynamic range. Though floating-point is powerful in arithmetic operations, its power comes at the cost of higher design complexity such as a floating-point multiplier and a floating-point adder. BFP arithmetic combines the advantages of floating-point and fixed-point arithmetic. BFP arithmetic offers a better signal-to-noise ratio (SNR) and dynamic range than does floating-point and fixed-point arithmetic with the same number of bits in the hardware implementation.

In a block-floating-point architecture FFT, the radix-2 or radix-4 computation of each pass shares the same hardware, with the data being read from memory, passed through the core engine, and written back to memory. Before entering the next pass, each data sample is shifted right

(an operation called "scaling") if there is a carry-out bit from the addition and multiplication operations. The number of bits shifted is based on the difference in bit growth between the data sample and the maximum data sample detected in the previous stage. The maximum bit growth is recorded in the exponent register. Each data sample now shares the same exponent value and data bit width to go to the next core engine. The same core engine can be reused without incurring the expense of a larger engine to accommodate the bit growth. The output SNR depends on how many bits of right shift occur and at what stages of the radix core computation they occur. In other words, the signal-to-noise ratio is data dependent and you need to know the input signal to compute the SNR.

Calculating Possible Exponent Values

Depending on the length of the FFT/IFFT, the number of passes through the radix engine is known and therefore the range of the exponent is known. The possible values of the exponent are determined by the following equations:

P = ceil{$\log_4$N}, where *N* is the transform length

R = 0 if $\log_2$N is even, otherwise R=1

Single Output range = (–3P+R, P+R–4)

Quad Output range = (–3P+R+1, P+R–7)

These equations translate to the values shown in Table 1.

Table 1. Exponent Scaling Values for FFT / IFFT *Note (1)*

Single Output Engine				Quad Output Engine			
N	P	MAX *(2)*	MIN *(2)*	N	P	MAX *(2)*	MIN *(2)*
64	3	–9	–1	64	3	–8	–4
128	4	–11	1	128	4	–10	–2
256	4	–12	0	256	4	–11	–3
512	5	–14	2	512	5	–13	–1
1024	5	–15	1	1024	5	–14	–2
2048	6	–17	3	2048	6	–16	0
4096	6	–18	2	4096	6	–17	–1
8192	7	–20	4	8192	7	–19	1
16384	7	–21	3	16384	7	–20	0

Note to Table 1:

(1) This table lists the range of exponents, which is the number of scale events that occurred internally. For IFFT, the output must be divided by *N* externally. If more arithmetic operations are performed after this step, the division by *N* must be performed at the end to prevent loss of precision.

(2) The MAX and MIN values show the number of times the data is shifted. A negative value indicates shifts to the left, while a positive value indicates shifts to the right.

For details of the division by *N* in the IFFT operation, refer to Equation 2 in the *Specifications* chapter of the *FFT MegaCore Function User Guide*.

Implementing Scaling

The scaling algorithm is implemented as follows:

1. Determine the length of the resulting full scale dynamic range storage register. To get the length, add the width of the data to the number of times the data is shifted (the MAX value shown in Table 1). For example, for a 16-bit data, 256-point Quad Output FFT/IFFT, MAX = –11 and MIN = –3. The MAX value indicates 11 shifts to the left, so the resulting full scaled data width is 16 + 11, or 27 bits.

2. Map the output data to the appropriate location within the expanded dynamic range register based upon the exponent value. To continue the above example, the 16-bit output data [15..0] from the FFT/IFFT is mapped to [26..11] for an exponent of –11, to [25..10] for an exponent of –10, to [24..9] for an exponent of –9, and so on.

3. Sign extend the data within the full scale register.

A sample of Verilog HDL code that illustrates the scaling of the output data (for exponents –11 to –9) with sign extension is shown in the following example:

```
case (exp)
   6'b110101 : //-11 Set data equal to MSBs
      begin
         full_range_real_out[26:0] <= {real_in[15:0],11'b0};
         full_range_imag_out[26:0] <= {imag_in[15:0],11'b0};
      end
   6'b110110 : //-10 Equals left shift by 10 with sign extension
      begin
         full_range_real_out[26] <= {real_in[15]};
         full_range_real_out[25:0] <= {real_in[15:0],10'b0};
         full_range_imag_out[26] <= {imag_in[15]};
         full_range_imag_out[25:0] <= {imag_in[15:0],10'b0};
      end
   6'b110111 : //-9 Equals left shift by 9 with sign extension
      begin
         full_range_real_out[26:25] <= {real_in[15],real_in[15]};
         full_range_real_out[24:0] <= {real_in[15:0],9'b0};
         full_range_imag_out[26:25] <= {imag_in[15],imag_in[15]};
         full_range_imag_out[24:0] <= {imag_in[15:0],9'b0};
      end
   .
   .
   .
endcase
```

In this example, the output provides a full scale 27-bit word. You need to choose how many and which bits should be carried forward in the processing chain. The choice of bits determines the absolute gain relative to the input sample level.

Figure 1 demonstrates the effect of scaling for all possible values for the 256-point Quad Output FFT with an input signal level of 5000H. The output of the FFT is 280H when the exponent = –5. The figure illustrates all cases of valid exponent values of scaling to the full scale storage register [26..0]. Since the exponent is –5, you need to look at the register values for that column. This data is shown in the last two columns in the figure. Note that the last column represents the gain compensated data after the scaling (0005000H), which agrees with the input data as expected. If you want to keep 16 bits for subsequent processing, you can choose the bottom 16 bits that result in 5000H. However, if you choose a different bit range, such as the top 16 bits, then the result is 000AH. Therefore, the choice of bits affects the relative gain through the processing chain.

Because this example has 27 bits of full scale resolution and 16 bits of output resolution, we choose the bottom 16 bits to maintain unity gain relative to the input signal. Choosing the LSBs is not the only solution or the correct one for all cases. The choice depends on which signal levels are important. One way to empirically select the proper range is by simulating test cases that implement expected system data. The output of the simulations should tell what range of bits to use as the output register. If the full scale data is not used (or just the MSBs), you must saturate the data to avoid wraparound problems.

Figure 1. **Scaling of Input Data Sample = 5000H**

					Exponent										Looking at Exponent = -5	
Bit		Input		Output Data	-11	-10	-9	-8	-7	-6	-5	-4	-3		Taking All Bits	Sign Extend / Pad
		5000 H		280 H												
26					0											0
25					0	0										0
24					0	0	0									0
23					0	0	0	0								0
22					0	0	0	0	0							0
21					0	0	0	0	0	0						0
20					1	0	0	0	0	0	0				0	0
19					0	1	0	0	0	0	0	0			0	0
18					1	0	1	0	0	0	0	0	0		0	0
17					0	1	0	1	0	0	0	0	0		0	0
16					0	0	1	0	1	0	0	0	0		0	0
15		0		0	0	0	0	1	0	1	0	0	0		0	0
14		1		0	0	0	0	0	1	0	1	0	0		1	1
13		0		0	0	0	0	0	0	1	0	1	0		0	0
12		1		0	0	0	0	0	0	0	1	0	1		1	1
11		0		0	0	0	0	0	0	0	0	1	0		0	0
10		0		0		0	0	0	0	0	0	0	1		0	0
9		0		1			0	0	0	0	0	0	0		0	0
8		0		0				0	0	0	0	0	0		0	0
7		0		1					0	0	0	0	0		0	0
6		0		0						0	0	0	0		0	0
5		0		0							0	0	0		0	0
4		0		0								0	0			0
3		0		0									0			0
2		0		0												0
1		0		0												0
0		0		0												0

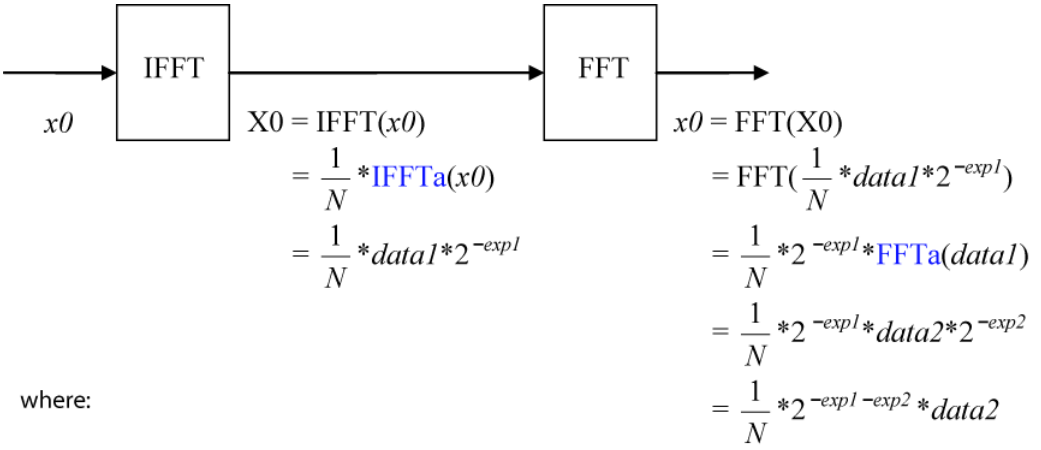
Achieving Unity Gain in an IFFT+FFT pair

Given sufficiently high precision, such as with floating-point arithmetic, you can theoretically obtain unity gain when an IFFT and FFT are cascaded. However, in BFP arithmetic, special attention must be paid to the exponent values of the IFFT/FFT blocks to achieve the unity gain. This section explains the steps required to derive a unity gain output from an Altera IFFT/FFT MegaCore pair, using BFP arithmetic.

Because BFP arithmetic does not provide an input for the exponent, you must keep track of the exponent from the IFFT block if you are feeding the output to the FFT block immediately thereafter and divide by *N* at the end to acquire the original signal magnitude.

Figure 2 shows the operation of IFFT followed by FFT and derives the equation to achieve unity gain.

Figure 2. Derivation to Achieve IFFT/FFT Pair Unity Gain



where:

x0 = Input data to IFFT
X0 = Output data from IFFT
N = Number of points
data1 = Altera IFFT MegaCore output data & Altera FFT MegaCore input data
data2 = Altera FFT MegaCore output data
exp1 = Altera IFFT MegaCore output exponent
exp2 = Altera FFT MegaCore output exponent
IFFTa = Altera IFFT
FFTa = Altera FFT

Any scaling operation on X0 followed by truncation will lose the value of *exp1* and not result in unity gain at *x0*. Any scaling operation must be done on X0 only when X0 is the final result. If the intermediate result X0 is first padded with *exp1* number of zeros and then truncated or if the data bits of X0 are truncated, then the scaling information will be lost.

One way to keep unity gain is by passing the *exp1* value to the output of the FFT block. The other way is to preserve the full precision of $data1*2^{-exp1}$ and use this value as input to the FFT block. The disadvantage

of the second method is a large size requirement for the FFT to accept the input with growing bit width from IFFT operations. The resolution required to accommodate this bit width exceeds, in most cases, the maximum data width supported by the core.

For more information, refer to the *FFT/IFFT Unity Gain* design example at **www.altera.com**.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com